TIDE and Tibbo BASIC User Manual

Tibbo Technology Inc.

Table of Contents

Taiko R2

Last update: 01SEP2012

Taiko is a solution which allows you to create programs for Tibbo modules capable of running TiOS (Tibbo Operating System), and products based on these modules.

With Taiko, you write your program in a language called Tibbo Basic (a close relative of any other BASIC you might already know), using a PC software called TIDE - Tibbo Integrated Development Environment. Your program is then compiled into a binary file and uploaded onto a Tibbo module. The Virtual Machine of TiOS then executes this binary.

Taiko allows you to easily create programs for a variety of Tibbo-based products. These may include:

- Alarm Panels
- Security Systems (Access control terminals, etc)
- Data Collection terminals, such as time clocks
- Sensor monitors
- Interface converters
- Vending machines
- Industrial process controllers

The solutions created with Taiko are very flexible. They are written using a language similar to BASIC, and are stored on a Tibbo module separately from the core OS of the module (TiOS). This allows for simple modification of your device functionality, even by the end-user (if you so allow).

Tibbo Basic itself is exactly the same for all TiOS-enabled devices. Hardware differences are expressed through so-called *platforms*. Change the platform, and you're programming for a different device.

Documentation Map

The documentation for Taiko includes:

Legal Information

Tibbo Technology ("TIBBO") is a Taiwan corporation that designs and/or manufactures a number of hardware products, software products, and applications ("PRODUCTS"). In many cases, Tibbo PRODUCTS are combined with each other and/or third-party

products thus creating a PRODUCT COMBINATION.

Whereas you (your Company) wish to purchase any PRODUCT from TIBBO, and/or whereas you (your Company) wish to make use of any documentation or technical information published by TIBBO, and/or make use of any source code published by TIBBO, and/or consult TIBBO and receive technical support from TIBBO or any of its employees acting in an official or unofficial capacity,

You must acknowledge and accept the following disclaimers:

1. Tibbo does not have any branch office, affiliated company, or any other form of presence in any other jurisdiction. TIBBO customers, partners and distributors in Taiwan and other countries are independent commercial entities and TIBBO does not indemnify such customers, partners or distributors in any legal proceedings related to, nor accepts any liability for damages resulting from the creation, manufacture, importation, advertisement, resale, or use of any of its PRODUCT or PRODUCT COMBINATION.

2. BASIC-programmable devices ("PROGRAMMABLE DEVICES") manufactured by TIBBO can run a variety of applications written in Tibbo BASIC ("BASIC APPLICATIONS"). Combining a particular PROGRAMMABLE DEVICE with a specific BASIC APPLICATION, either written by TIBBO or any third party, may potentially create a combinatorial end product ("END PRODUCT") that violates local rules, regulations, and/or infringes an existing patent granted in a country where such combination has occurred or where the resulting END PRODUCT is manufactured, exported, advertised, or sold. TIBBO is not capable of monitoring any activities by its customers, partners or distributors aimed at creating any END PRODUCT, does not provide advice on potential legal issues arising from creating such END PRODUCT, nor explicitly recommends the use of any of its PROGRAMMABLE DEVICES in combination with any BASIC APPLICATION, either written by TIBBO or any third party.

3. TIBBO publishes a number of BASIC APPLICATIONS and segments thereof ("CODE SNIPPETS"). The BASIC APPLICATIONS and CODE SNIPPETS are provided "as is" without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The entire risk as to the quality and performance of BASIC APPLICATIONS and CODE SNIPPETS resides with you. BASIC APPLICATIONS and CODE SNIPPETS may be used only as a part of a commercial device based on TIBBO hardware. Modified code does not have to be released into the public domain, and does not have to carry a credit for TIBBO. BASIC APPLICATIONS and CODE SNIPPETS are provided solely as coding aids and should not be construed as any indication of the predominant, representative, legal, or best mode of use for any PROGRAMMABLE DEVICE.

4. BASIC-programmable modules ("PROGRAMMABLE MODULES"), such as the EM1000 device, are shipped from TIBBO in either a blank state (without any BASIC APPLICATION loaded), or with a simple test BASIC APPLICATION aimed at verifying correct operation of PROGRAMMABLE MODULE's hardware. All other BASIC-programmable products including boards, external controllers, and developments systems ("NON-MODULE PRODUCTS"), such as the DS1000 and NB1000, are normally shipped with a BASIC APPLICATION pre-loaded. This is done solely for the

convenience of testing by the customer and the nature and function of pre-loaded BASIC APPLICATION shall not be construed as any indication of the predominant, representative, or best mode of use for any such NON-MODULE PRODUCT.

5. All specifications, technical information, and any other data published by TIBBO are subject to change without prior notice. TIBBO assumes no responsibility for any errors and does not make any commitment to update any published information.

6. Any technical advice provided by TIBBO or its personnel is offered on a purely technical basis, does not take into account any potential legal issues arising from the use of such advice, and should not be construed as a suggestion or indication of the possible, predominant, representative, or best mode of use for any Tibbo PRODUCT.

7. Any advance product or business information posted as news or updates of any kind (including Tibbo Blog posts, Tibbo Newsflashes, site news, forum posts and any other timely information posted by Tibbo personnel) shall not be construed as obligatory to TIBBO in any way, shape or form. TIBBO may change or delay any of its plans and product roadmaps without prior notice, and shall not be held liable for such changes to the extent permissible by the applicable law.

8. Neither TIBBO nor its employees shall be held responsible for any damages resulting from the creation, manufacture, or use of any third-party product or system, even if this product or system was inspired, fully or in part, by the advice provided by Tibbo staff (in an official capacity or otherwise) or content published by TIBBO or any other third party.

9. TIBBO may make non-English documentation or other information available at its discretion. Such texts may be the result of work done by third parties, and may not always be reviewed by TIBBO personnel. As such, these are not to be considered official statements by TIBBO. Any apparent inaccuracies, conflicts or differences in meaning between English-language and non-English texts shall always be resolved in favor of the English-language version.

10. TIBBO reserves the right to halt the production or availability of any of its PRODUCTS at any time and without prior notice. The availability of a particular PRODUCT in the past is not an indication of the future availability of this PRODUCT. The sale of the PRODUCT to you is solely at TIBBO's discretion and any such sale can be declined without explanation.

11. TIBBO makes no warranty for the use of its PRODUCTS, other than that expressly contained in the Standard Warranty located on the Company's website. Your use of TIBBO PRODUCTS is at your sole risk. TIBBO PRODUCTS are provided on an "as is" and "as available" basis. TIBBO expressly disclaims the warranties of merchantability, future availability, fitness for a particular purpose and non-infringement. No advice or information, whether oral or written, obtained by you from TIBBO shall create any warranty not expressly stated in the Standard Warranty.

12. LIMITATION OF LIABILITY. BY USING TIBBO PRODUCTS YOU EXPRESSLY AGREE THAT TIBBO SHALL NOT BE LIABLE TO YOU FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL OR EXEMPLARY DAMAGES, INCLUDING, BUT

NOT LIMITED TO, DAMAGES FOR LOSS OF PROFITS, GOODWILL, OR OTHER INTANGIBLE LOSSES (EVEN IF TIBBO HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES) RESULTING FROM THE USE OR THE INABILITY TO USE OF TIBBO PRODUCTS.

13. "Tibbo" is a registered trademark of Tibbo Technology, Inc.

14. Terms and product names mentioned on TIBBO website or in TIBBO documentation may be trademarks of others.

Overview

Below is a summary of the major fundamentals and theory behind TIDE. This may sound intimidating, but it's actually quite simple. You *should* at least skim over the material herein, because it explains much of what comes next. In here you will find:

- Our Language Philosophy 4
- System Components 7
- Objects 8 (a very brief overview)
- Events 8

Our Language Philosophy

Several principles have guided us through the development process of Tibbo Basic. Understanding them would help you understand this manual better, and also the language itself. See below:

A Bit of History

Years ago, programming for the PC was the nearly exclusive domain of engineers. The languages traditionally available, such as C, simply required you to be an engineer to program.

However, one day something interesting happened. Visual Basic* and Delphi** saw the light of day. And that changed quite a lot on the PC front. Suddenly, people who were not engineers were finding out that they could actually create something cool on their PC. You could say VB* and Delphi democratized the PC software market.

The situation on the embedded systems market today is quite similar to the situation which existed for the PC market in the pre-VB era. Many embedded systems vendors do offer customizable or programmable solutions -- but to implement those solutions, you would really have to be an engineer and know C/C++ quite well. So, there was clearly a need for an easy-to-use programming system which would democratize this market, as well.

Principle One: Easy To Write, Easy to Debug

Choosing BASIC as our inspiration was the natural thing to do, for us. It's a language which doesn't require you to be a professional engineer. It is easy to understand. This is why it is embedded into many non-programmer products, such as the Office suite. So we went for BASIC.

Another part of the user experience, and a major one, too, is debugging. Writing your application is just half the job. You also need to debug it and for embedded systems, this is where things typically start getting rough around the edges. Many times you have to buy expensive tools, such as ICE machines (In-Circuit

Emulators), just to figure out what your code is doing. Sometimes you don't even have the luxury of such a machine, and you actually debug by guessing and trying different things in your code.

With our system, one of our major goals was to offer a user experience which is close to debugging on the PC -- without the need for special tools, such as an ICE machine.

While your program is running on the target (embedded device), you actually see how it runs *on your PC*. You can step through it, jump to specific functions, check values of variables etc -- all from the comfort of your own PC.

Principle Two: Easy Doesn't Mean Sloppy

Some modern programming languages use certain techniques to make life 'easier' for programmers. They might not require the programmer to explicitly declare the variables he's going to use ('implicit declaration'), or might do away with the need to specify the type for the variable (i.e, use 'variant variables' which can contain anything).

This has several disadvantages. For one, it is just sloppy. After several days of writing code like that, a programmer might not have a very clear-cut idea of what his program is doing, or where things come from. While this is something which may be subject to debate, the next disadvantage is quite real:

This is simply wasteful programming. These techniques can consume quite a lot of resources, specifically memory. On the PC, a variant used to store just 2 bytes of data might take up to 100 bytes. This isn't a problem, because PCs have so much memory these days that it is barely felt.

However, embedded systems are often low-cost and bare-bones, so physical memory is a truly valuable resource. Waste too much of it -- and you would find that your code can do very little. But manage it prudently, and your code will be capable of quite impressive feats even on your 'low-power' embedded system.

So our systems requires you to be more organized. The effort is worth it.

Principle Three: The Purity of Language

Programming systems on the PC usually make no clear distinction between the 'pure' language constructs which perform calculations and control program flow, and hardware-dependant input/output. For example, many languages contain a print statement which prints something to the screen.

Since all PCs in the world are similar, this works. However, this makes little sense for embedded platform, which have vastly different input/output resources. Depending on the device, it may or may not have a screen, a serial port, networking etc etc.

In our system, we separated the language itself (what we call *the core language)* from the input/output of a particular device. Thus, the language itself remains the same, no matter what device you are programming for. The input/output part is hardware dependant, and changes from platform to platform.

When writing for a specific platform, you are provided with a set of platform-specific *objects.* These provide rich functionality and allow you to do 'real' work, such as printing messages to the serial port, communicating on the Internet or controlling motors and sensors.

Ideally, Tibbo Basic could run on a fridge just as well as it could run on a time and attendance terminal.

Principle Four: Thin and Agile

A lot of embedded systems are built by scaling down larger desktop systems, and it shows. What's the point of using a super-fast processor if you load it with dozens of layers of nested calls?

All the code TiOS includes has been designed from scratch for running on a very simple processor, and optimized for control applications. It has been crafted to have the minimum possible ROM and RAM footprint and to run as fast as possible.

We built TiOS with Pareto's principle in mind. In other words, if a certain functionality is required by only 5% of applications and yet its existence adds 90% overhead, we did not include it. For example, we decided to use a static memory model for procedure variables. Memory is not allocated and deallocated dynamically -- It is assigned on compile-time, which results in great performance improvements.

Principle Five: No B.S

... that is, no babysitting. Development systems intended for rapid application development on the PC will often try to handle every little error or problem the programmer may encounter. If a variable overflows, for example, they will *halt* execution and pop up an error to let him know. This makes sense for a PC-based product, because you are right there to see it halt.

However, when you are creating an embedded system, you expect it to run *at all times,* without halting. Nobody will be there to see any errors and babysit your system. Your device is simply expected *to work*.

This is a major difference also for the development process. In essence, since the whole language is built this way, you will also get much less errors even when doing seemingly 'strange' things, such as putting large values into variables that cannot hold them. The language will deal with it silently, in a very predictable and logical way -- but will not pop up an error.

Principle Six: Event-Driven Programming

Users of VB and Delphi and other Windows-based tools will find this principle familiar. However, if most of your experience with BASIC was under DOS, you might find this slightly odd. Under DOS, you would expect a program to begin from the beginning, then continue and stop. They execute from top to bottom. This may be called *linear execution*.

For Tibbo Basic, this is not the case. The programs you will write will be *event-driven*. Your program will consist of a number of *event handlers* which will be fired (invoked) in response to specific things which happen to your system in real life. If your platform was a fridge, you might want to write a handler for a 'door opening' event. When the door is opened, an event is generated, and an event handler, with your code in it, is fired.

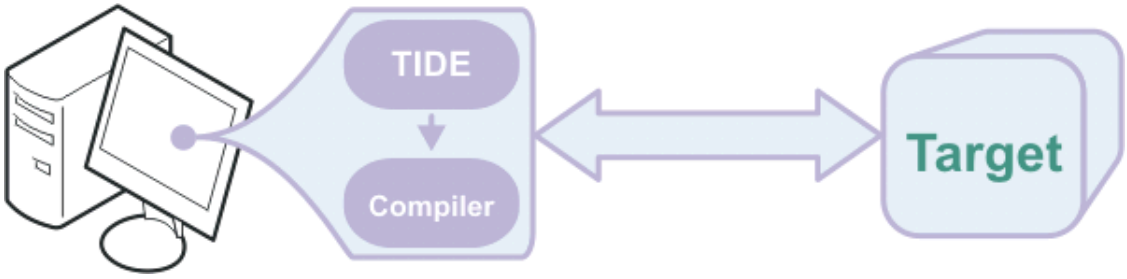
So, you could say that your event-driven application has no beginning and no end. Event handlers are called when events are generated, and in the order in which they were generated.

* Windows, Visual Basic and VB are registered trademarks of Microsoft Corporation Inc.

** Delphi is a registered trademark of Borland Inc.

System Components

Taiko is a compound system. It consists of the following components:



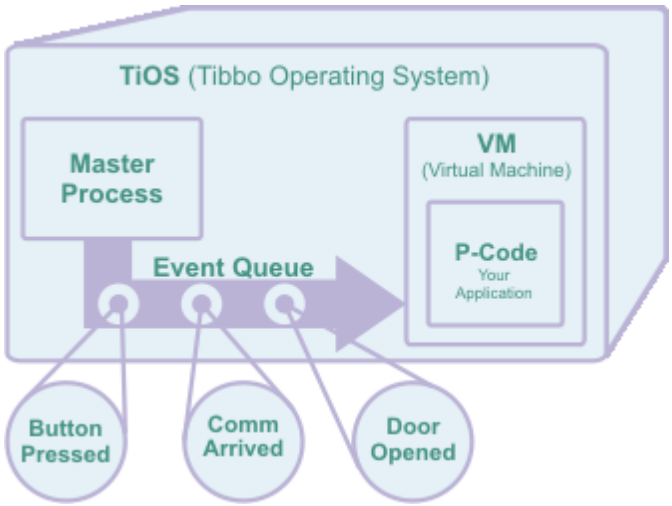
TIDE is an acronym for Tibbo Integrated Development Environment. This is the PC program in which you will write your applications and compile them, and from which you will upload them to your target and debug them.

The **compiler** is a utility program, used by TIDE. The compiler processes your project files and creates an *executable binary file* (with a .tpc suffix, for Tibbo PCode).

The **target** is a separate hardware device, on which your program actually runs. When debugging code, it is connected to your computer running TIDE (see the link above) and TIDE can monitor and control it. This is called *cross-debugging*.

As covered under Our Language Philosophy 4, Tibbo Basic is capable of running on various hardware devices. Each type of hardware device on which Tibbo Basic runs is called a *platform*.

And now, the anatomy of the target:



The **target** runs an operating system called TiOS (Tibbo Operating System).

TiOS runs two processes. One is the **Master Process**. This is the process which is in charge of communications (including communications with TIDE) and of generating events. The second process, which is under the control of the Master Process, is called the **VM** (Virtual Machine).

The VM is what actually executes your application. In essence, the VM is a processor implemented in firmware, which executes the compiled form of your application. The instructions it understands are called **P-Code**, which is short for pseudo-code. This is what the compiler produces. It is called pseudo-code because

it is not native binary code which the hardware processor can understand directly; instead, it is interpreted by the VM.

Since the VM is under the complete control of the Master Process, the actual hardware processor will not crash because of an error in your Tibbo Basic application. Your application may operate incorrectly, but you still will be able to debug it. The Master Process can stop or restart the Virtual Machine at will, and can exchange debug information with TIDE, such as report current execution state, variable values, etc.

Simply put, you can think of the VM as a sort of a 'sandbox' within the processor. Your application can play freely, without the possibility of crashing or stalling TiOS due to some error.

The **queue** is used to 'feed' your program with events 8 which it should handle. The Master Process monitors the various interfaces of the platform and generates events, putting them into the queue. The Virtual Machine extracts these events from the other side of the queue and feeds your program with them. Various parts of your program execute in response to events.

Objects

Objects represent the various component part of your platform. For example, a platform with a serial port might have a *ser* object. A platform can be described as a collection of objects.

Under Tibbo Basic, the set of object you get for each platform is fixed. You cannot add new objects or create multiple instances of the same object.

Objects have properties, methods and events. A *property* can be likened to an attribute of the object, and a *method* is an action that the object can perform. *Events* 8 are described in the next section.

Objects are covered in further detail under Objects, Events and Platform Functions 82.

Events

An *event* is something which happens to an object. Plain and simple. A fridge might have door object with an on_door_open event, and a paper shredder might have a detector object with an on_paper_detected event.

Events are a core concept in Tibbo Basic. They are the primary way in which code gets executed.

The target device maintains an *event queue*. All events registered by the system go into this queue. On the other end of the queue, the Virtual Machine takes out one event at a time and calls an event handler for each event.

Event handlers are subroutines in your code which are 'fired' (executed) to handle an event. Often, event handlers contain function calls which run other parts of the program.

While processing an event, other events may happen. These events are then queued for processing, and patiently wait for the first event to complete before beginning execution.

All Tibbo Basic programs are single-threaded, so there is only one event queue. All events are executed in the exact order in which they were queued.

It may sometimes seem that some events should get priority over other events. This functionality is not supported under Tibbo Basic. This is not crucial, as events

tend to execute very quickly, and the queue ensures events are not forgotten.

Getting Started

Below is a walk-through for a starter project which is written specifically for the EM202-EV and DS202.

Once you are done with this project, you will be able to press the button on the EM202-EV or DS202 and watch the LEDs blink "Hello World!" in Morse code.

This project would actually run also on the EM202, EM200 and EM120 modules. However, these modules cannot work on their own, and you cannot easily test with them.

Preparing Your Hardware

Preparing a DS202

Before starting to use TIDE, you should upload the correct firmware to a DS202. Perform the following steps:

- Get tios_EM202_xxx.bin firmware file (the latest version) from the Tibbo website. _100 in this filename stands for version 1.00, for example.
- Connect the DS202 to power (preferably, use adaptor supplied by Tibbo).
- Connect the DS using a network cable (WAS-1499 or similar) to the same hub your computer is connected to, or directly to the computer with a cross network cable (WAS-1498 or similar).
- Make sure your local firewall (such as the XP SP2 firewall) is disabled or does not block broadcast UDP messages. This is essential for communications between TIDE and the DS202 while debugging.
- Run Device Explorer (Start > Programs > Tibbo > Tibbo IDE > Device Explorer).
- You should see your device on the list. Select it.
- Click Upload > Load Firmware Through the Network.
- Select the firmware file, and click OK.
- The firmware will now be uploaded.
- For some firmware versions, you now have to manually reboot the DS (Disconnect and reconnect the power cable). The red Status LED should now blink rapidly. This is OK -- it means the TiOS firmware is loaded and the application program memory is empty.
- Proceed to Starting a New Project 10.

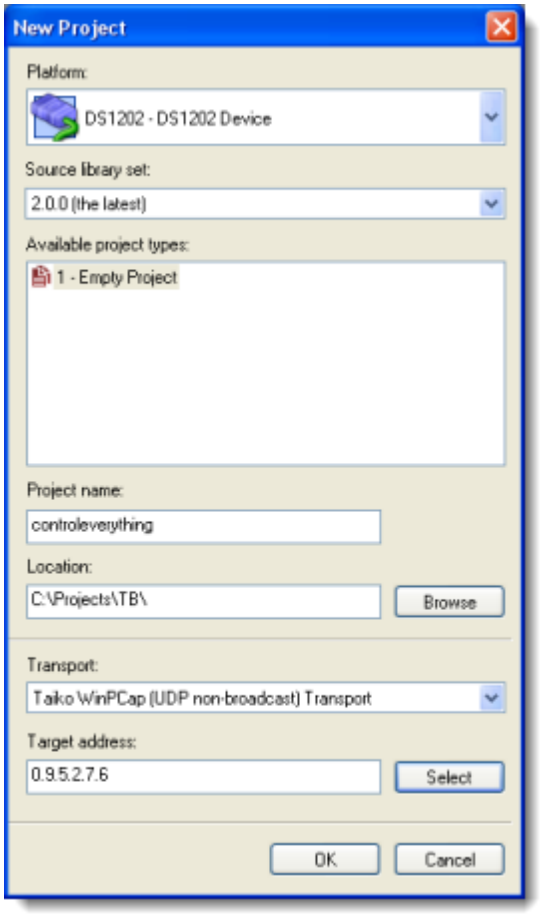
Of course, once you upload a device with the TiOS firmware, it is no longer a Device Server! So you cannot see it under DS Manager. You could program it so it would respond to DS Manager -- but by default it is a 'clean slate', and does not respond to DS Manager broadcasts.

If for some reason you cannot perform a network upload, you can perform a serial upload by selecting Upload > Load firmware Through the Serial Port. You

will then be prompted to select a COM port, turn the device off and turn it back on while pressing the SETUP button. Upload will then commence.

Starting a New Project

To begin a new project, select File > New. You will be presented with the following dialog:



Platform: Select EM202 (you can use EM1000 as well)

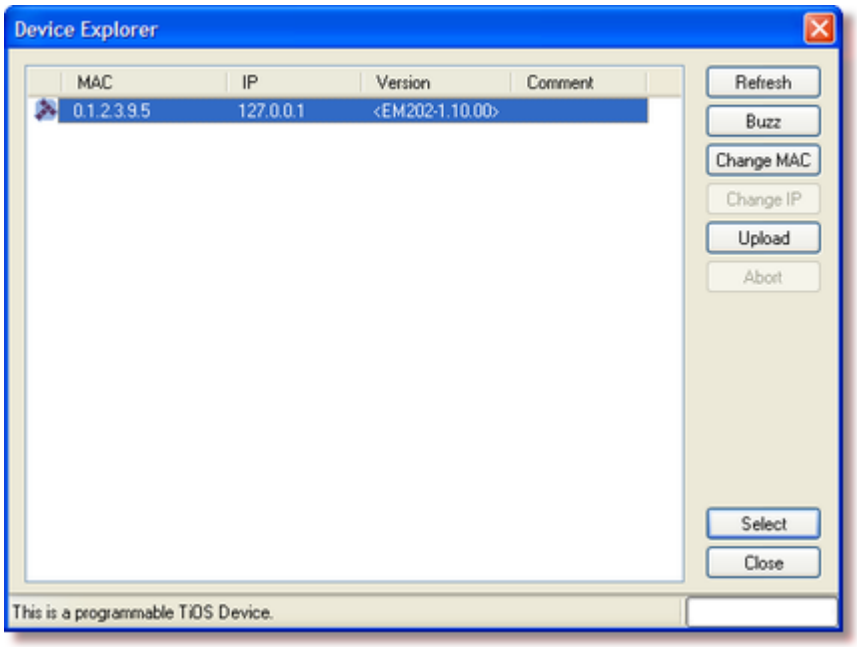
Available project types: Select Empty Project.

Project name: Type 'Hello World'.

Location: Leave untouched, unless you have a good reason to change it.

Transport: leave it as is ("Taiko UDP Broadcast Transport")

Target Address: Click Browse. You will be presented with the following dialog:



The number (hopefully) displayed is the MAC address of your target. If you select it and click Buzz, you should see the LED pattern on your target switch off momentarily. This means it is correctly detected.

If you see nothing in this dialog, it means your target isn't in communication with the computer. This is probably a power problem, or a networking problem. Perhaps you have a local firewall on the computer which blocks UDP broadcasts, such as the Windows XP Firewall. To fix this, disable the firewall or configure it to open a specific port.

Once you have located your target, click Select. You will be returned to the previous dialog, and the MAC address for your target will appear under Target Address.

You have now specified all of the required settings for a new project. Click OK and proceed.

Writing Code

Once you have started your new project, you will be presented with a blank file (main.tbs).

We will now begin writing the actual code in this file. We will construct this project from beginning to end, step by step. For your convenience, the end of this section contains a complete copy of the project without comments. You can copy and paste the whole thing into TIDE, or just copy and paste the commented sections one by one as they appear below.

Here goes:

```
' Comments cannot spill over to the next line. If you see this happening in
this manual, it is a result of the help system -- not an actual feature.

Dim hello_world As String ' define a variable which will hold the whole
pattern we will play.

Dim length, play_position As Integer ' length is a calculated integer which
will contain the whole length of the string we will play, and play_position
```

```
will contain our current position in this string (how much we have played so
far).

Const PAT_PLAY_CHUNK_LENGTH = 15 ' define a constant for the size of the
chunk we will play. We will play one chunk of the pattern at a time, and
then move on to the next chunk. Each chunk is 15 'steps' long.

Declare Sub play_Next ' let the compiler know that there is a sub called
play_next. This sub will be used in code before being created so we must
declare it.
```

Notice that we are defining a *chunk* above. The reason for this is that we are going to play quite a long and complex pattern (over 130 steps in length), but the pattern object (pat.) used to play the pattern only supports patterns of up to 16 steps. So we have to play our pattern in parts, one after the other, and track our progress through the pattern (this is what the counters are for).

So far, we have prepared the ground. Let us move to the first piece of executable code:

```
sub on_sys_init ' event handler for the init event. Fires whenever the
device powers on.
      hello_world = ' here we define the contents of our string, in morse.

            'R is Red LED, G is Green LED. GGG means a long pulse of the
green LED (line). R means a short pulse of the Red LED (dot). Line (-) means
both off.


            'HELLO .... . .-.. .-.. ---
            "R-R-R-R---R---R-GGG-R-R---R-GGG-R-R---GGG-GGG-GGG" +
            "-------" + ' A period of silence between words
            'WORLD .-- --- .-. .-.. -..
            "R-GGG-GGG---GGG-GGG-GGG---R-GGG-R---R-GGG-R-R---GGG-R-R" +
            "-------" +
            '! ..--..
            "R-R-GGG-GGG-R-R-"
      length = len(hello_world) ' Calculate total length of string.
      play_position = 1 ' Initialize play_position as we haven't played
anything yet.
end sub
```

We will now write the event handlers for our code.

First, we want the pattern to start playing whenever you press the button. For this, our platform offers a button object, which generates an on_button_pressed event. Instead of typing, you can create the event handler for this event by double-clicking on the event name in the project tree 134.

```
sub on_button_pressed ' event handler fired whenever the button is pressed
      play_position = 1 ' start playing from the beginning of the pattern
      play_next ' call the routine which plays the next chunk (the first
chunk, in this case)
end sub
```

Notice that the play_next routine is not yet defined. In our code, it is first used and then defined. This is why we have to declare 84 it at the beginning.

Now, let us move on to the next event handler:

```
sub on_pat   ' this fires whenever a pattern (a chunk, in our case) finishes
playing.
      play_next ' call the routine which plays the next chunk
end sub
```

We have now completed writing our event handlers. Our program now knows what it's supposed to do whenever you press the button, and whenever a chunk of the pattern finishes playing. It just doesn't know *how* to do it yet. This comes next:

```
sub play_next ' plays the next chunk of our large pattern.
      if length < play_position then exit sub ' if we have reached the end of
the pattern, stop.

      dim chunk_len as integer ' internal integer for the length of current
chunk to be played.

      chunk_len = length - play_position + 1 ' calculate how much of the
large string is left.

      if chunk_len > PAT_PLAY_CHUNK_LENGTH then chunk_len =
PAT_PLAY_CHUNK_LENGTH ' if too much is left, we bite off only a chunk we can
process.

      dim chunk as string ' will contain the chunk which will actually play
now.
      chunk = mid(hello_world, play_position, chunk_len) ' chunk is the part
of hello_world which begins at play_position and is as long as chunk_len.

      pat.play(chunk, YES) ' Play this chunk. YES means the pattern may be
interrupted -- you can press the button while the pattern is playing, and it
will start again from the top.

      play_position = play_position + chunk_len ' advance play_position to
account for the chunk we played.

end sub
```

Here is the whole project, without comments:

```
'=============================================================================
===
'                   HELLO WORLD IN MORSE CODE (for EM202-EV, DS202)
'=============================================================================
===

dim hello_world as string
dim length, play_position as integer

const PAT_PLAY_CHUNK_LENGTH = 15
```

```
declare sub play_next

'--------------------------------------------------------------------------------
---
sub on_sys_init
      hello_world =
             "R-R-R-R---R---R-GGG-R-R---R-GGG-R-R---GGG-GGG-GGG" +
             "-------" +
             "R-GGG-GGG---GGG-GGG-GGG---R-GGG-R---R-GGG-R-R---GGG-R-R" +
             "-------" +
             "R-R-GGG-GGG-R-R-"
      length = len(hello_world)
      play_position = 0
end sub

'--------------------------------------------------------------------------------
---
sub on_button_pressed
      play_position = 1
      play_next
end sub

'--------------------------------------------------------------------------------
---
sub on_pat
      play_next
end sub

'--------------------------------------------------------------------------------
---
sub play_next
      if length < play_position then exit sub

      dim chunk_len as integer
      chunk_len = length - play_position + 1
      if chunk_len > PAT_PLAY_CHUNK_LENGTH then chunk_len =
PAT_PLAY_CHUNK_LENGTH

      dim chunk as string
      chunk = mid(hello_world, play_position, chunk_len)
      pat.play(chunk, YES)
      play_position = play_position + chunk_len
end sub
```

Building, Uploading and Running

Once you are done with writing your project, it is time to build, upload and run it. These three operations can be done by pressing F5.

Press F5 and wait. You will see your project compiling. The output pane 133 will display any errors (if you copied the project as it is, there should be no errors).

The status bar 130 will show you the project building, uploading, and running.

Once the status bar says RUNNING, you may press the button on your device to see it blink "Hello World" in Morse.

For further information about these topics, please see Making, Uploading and Running an Executable Binary 26 and Debugging Your Project 28 below.

Compiling a Final Binary

The binary executable file you compiled in the previous step is called a debug binary 27. This type of binary is used while creating your project and debugging it.

When you decide your project is ready to be deployed in the real world, you should compile a release binary 27. To do this, select Project > Settings and uncheck the Debug version checkbox.

Use Project > Build and Upload to upload the compiled binary into the target. It will automatically start running whenever the device is powered up, and all debug functions will be disabled.

This compiled application binary file will also remain on your hard drive, inside your project folder (see Starting a New Project 10). You may upload it 41 to any number of devices using the Device Explorer 39.

You can optionally protect the firmware and application loaded into your device with a password 41 (strongly recommended).

Programming with TIDE

The topics below attempt to give you a general understanding about working with TIDE. An attempt has been made to lay them out as logically as possible; it would be advised to just read the section from top to bottom and follow the links every time you don't understand a term.

The section called Managing Projects 15 provides an overview of the general structure of a Tibbo Basic project, and also discusses the debugging process.

The next section, Programming Fundamentals 43, then delves into the specifics of Tibbo Basic programming, including the differences between Tibbo Basic and other languages you may know.

Managing Projects

Each program you will make with Tibbo Basic is actually a project. Projects include certain files, and have a specific structure. They are compiled into binary files, uploaded onto your target and debugged.

In this section:

- Protecting Your Device with Password 41

The Structure of a Project

A *project* is a collection of related files and resources, which are then compiled into one final binary file, uploaded onto a target and run. Projects include actual source files, HTML files (if any), images (if any), etc.

Your project's files come from two distinctive places: the *project folder* and the *source libraries folder*. Files in the project folder are really "your" project's files. Your project may (and should) make use of free libraries 580 provided by Tibbo. You don't have to copy library files into your project folder -- you can add them directly into you project (see Adding, Removing, and Saving Files 18, "Adding existing files to your project").

Want to modify a library? Then you really better copied this library's files into your project's folder. It is a very bad practice to modify the files in the library folder.

Here are the files that form a project:

Project file: A single file with a **.tpr** extension. Contains project settings, and a list of all files included with the project. You don't have to edit this file manually -- TIDE handles it for you. This file is always kept in the project folder.

Header files: Multiple files with a **.tbh** extension. Used for inclusion into other files; usually contain declarations for global variables, constants, etc.

BASIC files: Multiple files with a **.tbs** extension. Contain the actual body of your program.

HTML files: Multiple files with an **.html** extension (displayed with the currently associated icon). Contain webpages to be displayed by the embedded webserver of your device. These can include blocks of Tibbo Basic code. See Working with HTML 79.

(Any icon)

Resource files: Multiple files without any preset extension. Contain resources (such as images) needed for other files. Some resource files (.cfg, .txt, .ini) can be edited from within TIDE:

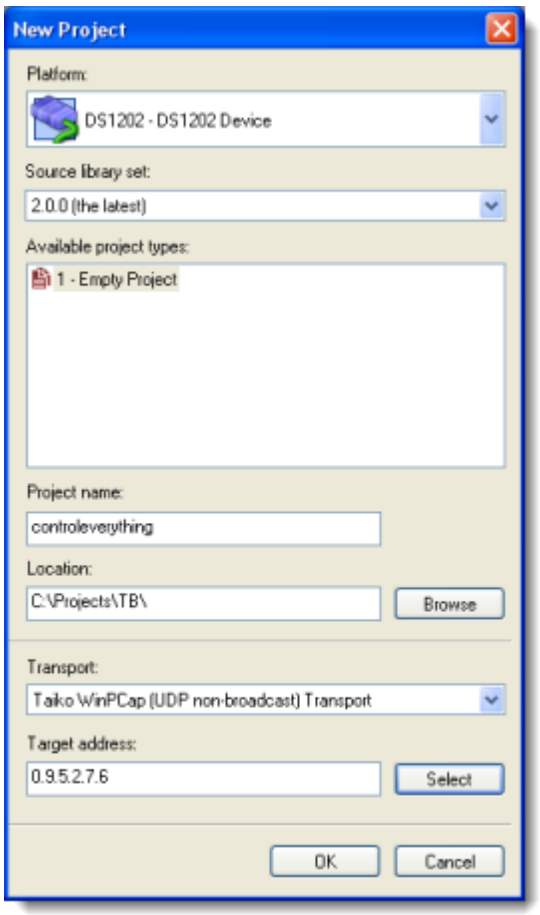
- *.cfg*, *.txt*, and *.ini* files are considered to be text files and can be edited using TIDE's built-in text editor.
- *.bmp*, *.jpg*, and *.png* files are graphical files; these will be opened using TIDE's built-in image editor 20.

Note that the only really *fixed* extension is the one for the project file -- **tpr**. This file contains references to other files within the project. These may use any extension, as long as their type is correctly stated in the project file (this is selected when adding the file, as described here 18).

The extensions above are the default extensions which are associated with TIDE, and we recommend keeping them.

Creating, Opening and Saving Projects

To create a new project, select File > New. The following dialog will appear:



Platform: The platform 138 on which your project will run.

Source library set: Tibbo libraries come in sets. Each set has a version number. Once you choose the library set for your project, it will be used until you change it. Tibbo will keep releasing new library sets. With each new release we will check if backward compatibility has been lost due to new features. If backward compatibility is lost, we will publish the new library set under the new version number. Old library sets will still be distributed and your project will keep compiling without any errors. For new projects, always choose the current library set as defined in the Library Reference 580.

Available project types: Only an Empty Project template is currently offered.

Project name: A short name for your project. TIDE will use this name to create a folder for your project, and also to create a project file (.tpr) within this folder.

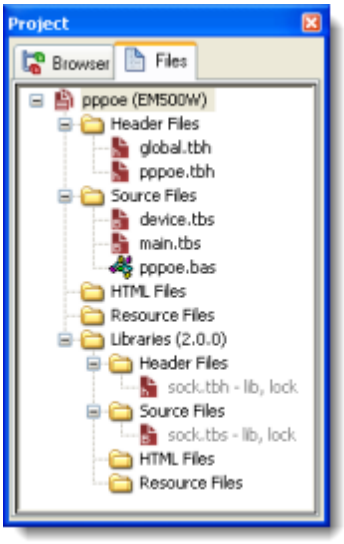
Location: The folder which will contain your project folder.

Target address: *Platform-specific.* See platform 138 documentation. The address of the target you will use for debugging and testing this project. This should be a reachable address with a live target. Your project will still be created even if you do not specify this parameter, but you will not be able to upload or debug until you specify the target using the Project Settings 38 dialog.

Adding, Removing and Saving Files

Files tab

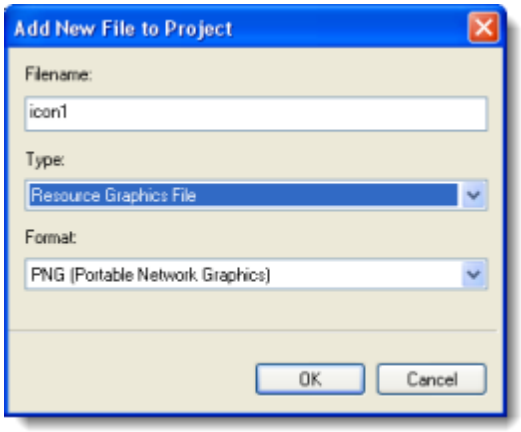
You can see what files are included in your project at any given time using the Files tab of the Project pane 134. Notice that the project structure includes a separate branch for library files 580. These come from the currently selected library set. Library set is defined when creating new project 17 or editing project settings 38.



Adding New Files to Your Project

To add new files to your project, click Project > Add New File or click the Add new file button on the Project Toolbar.

You will be presented with the following dialog:



Specify a name for your file under Filename. If you also specify an extension, the Type listbox will update, too. Automatically recognized file types are:

- *.bas* -- basic files
- *.tbh* -- header files

- *.htm, .html* -- HTML files
- *.bmp*, *.jpg*, *.png*, *.gif*, *.ico*, *.pcx* -- resource graphics files
- *.txt* -- resource text files
- all other file extensions are classified as binary resource files by default.

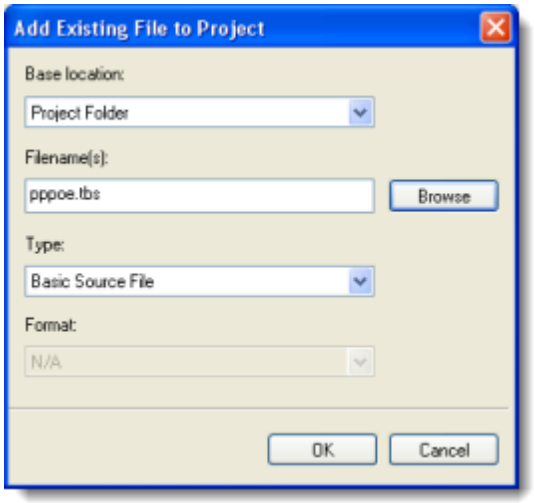
You can still set the file type regardless of the extension -- just select what you need in the Type listbox.

Adding image 20 files to the project will prompt an additional request for size in pixels, etc.

Adding Existing Files to Your Project

To add existing files to your project, click Project > Add Existing File or click the Add existing file button on the Project Toolbar.

You will be presented with the following dialog:



- Under Base location you can select either Project Folder or Source Library Set Folder. In the latter case the file you add is expected to reside in the source libraries folder, inside the currently selected library set. You defined the set to be used when creating 17 the project. You can also change it through the Project Settings 38 dialog.
- Once you selected the location, browse for the filename.
- You may need to make an appropriate selection of file Type and Format.

Removing Files from Your Project

To remove a file from your project, first select (single-click) it in the project tree. Then click Project > Remove File or click the Remove button on the Project Toolbar.

You will be presented with a prompt. If you're sure you want to remove the file,

select OK, and the file will be removed from the project. Note that it is not physically deleted -- only removed from the project tree.

Saving Files

To manually save your work, select File > Save, press Ctrl+S, or click the Save button on the Project Toolbar.

Any of these actions would save all open and modified files in your project, including the project file itself.

In addition, every time your project is compiled, all open and modified files are saved.

Resource Files

Sometimes a project may need access to certain files which are not Tibbo Basic code per se; these may be image 20 files, sound files or any other fixed binary data which is not to be interpreted or modified by the Tibbo Basic compiler but simply used as-is within the project.

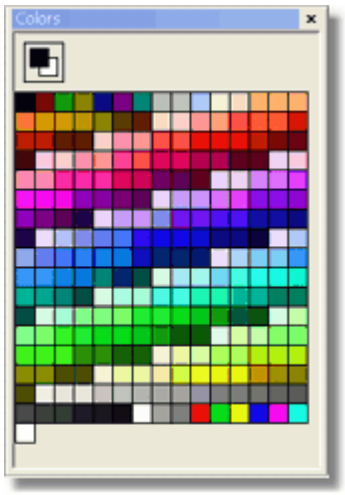
These files are not modified or compressed in any way; they are merely included within the final, compiled binary file 26 and may be accessed from within the program or by the built-in HTTP server.

Resource files are included in the project tree 134 under the Resource Files branch.

Built-in Image Editor

Beginning with release **2.0**, TIDE features a simple image editor that "knows" how to work with *.bmp*, *.jpg*, *.png,* and *.gif* files. The editor is primarily intended for bitmap-level jobs such as preparing "screens" for a device with an LCD display.

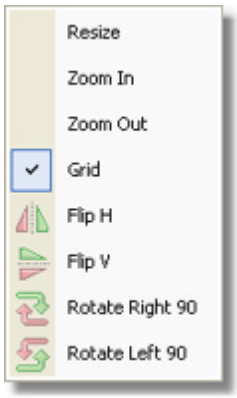
All features of the image editor are fairly standard and we see no need in discussing image editor's functionality in details. Image files are added 18 to the project as any other resource files. When adding an image file to the project you will be presented with a choice of selecting RGB or palette color mode for this file. Depending on your choice the Colors pane 135 for this image will display available palette colors...



... or RGB color selector:

Double-clicking on the image file with *.bmp*, *.jpg*, *.png,* or *.gif* extension opens the image editor.

All image-related editing functionality is concentrated within Image menu 124...



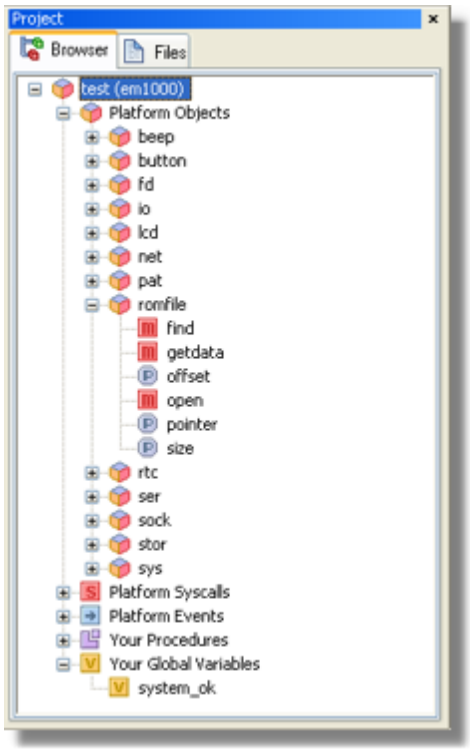
...and Image Editor toolbar 127:

Coding Your Project

TIDE contains a code editor with the following facilities:

- Syntax highlighting
- Auto-Completion 23
- Hinting 24
- Tooltips 24 for properties, events, functions, and even user-configurable tooltips for user-defined functions.

Project Browser



The Project Browser contains a tree of all objects in the platform (with their methods, properties and events), as well as all procedures and global variables of your project. The tree is updated in real time, using a dynamic background parser which constantly analyzes your source code.

The tree features icons 136 for the various constructs.

An icon next to an event is grayed (inactive) if this event does not have an event handler implemented in the project. The icon becomes "active" when the event handler is created. An icon next to a procedure is grayed if this procedure is merely declared but does not yet have a body. The icon becomes active once the procedure is implemented ("gets" a body). Same applies to global variables -- gray

icon next to variables that are declared but not yet defined, active icon for defined global variables.

Double-clicking on an event which does not yet have an event handler will create an empty event handler procedure for this event at the bottom of the currently active file. Double clicking on an event which already has an event handler will make the cursor jump to this event handler.

Double-clicking on a procedure which does not yet have a body will make the cursor jump to the location where this procedure is defined. Double-clicking on a procedure which already has a body will make the cursor jump to this body.

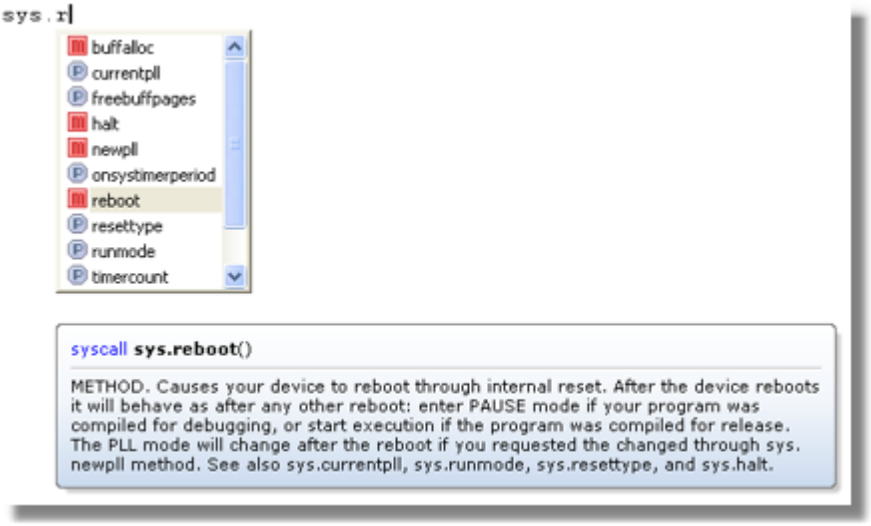
Double-clicking on a global variable which is not yet defined (using the **dim** 86 statement 86) will make the cursor jump to the location where this variable is declared (using the **declare** 84 statement 84). Double-clicking on a defined global variable will make the cursor jump to the location where this variable is defined.

Hovering the cursor over an item in the displays a tooltip 24 for this item. Additionally, when in debug mode, hovering the cursor over global variables and object properties displays their current values.

Notice, that currently selected platform is displayed next to the project name in the topmost tree node.

Code Auto-completion

If you type an object name followed by a dot, the TIDE will pop-up a code-completion box. It looks like this:

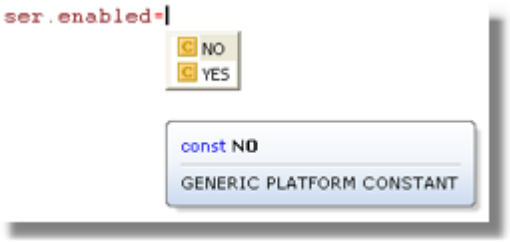


This list also supports tooltips 24 -- they are displayed as you scroll down the list. You can also hover the mouse over an item in the list to see its tooltip.

Code completion box is also displayed for the structures 58 in your project:

You can also get a list of constants related to your construct. For example, by

typing "ser.enabled=" you will get a list of possible values of this property:



- By pressing Alt+Ctrl+T when the cursor is directly to the right of any meaningful Tibbo Basic construct, you will get a pop-up list with the appropriate contents for the current context. If the cursor isn't immediately to the right of any such construct, you will get a list containing all platform enumeration types, constants, objects and events.

Code Hinting

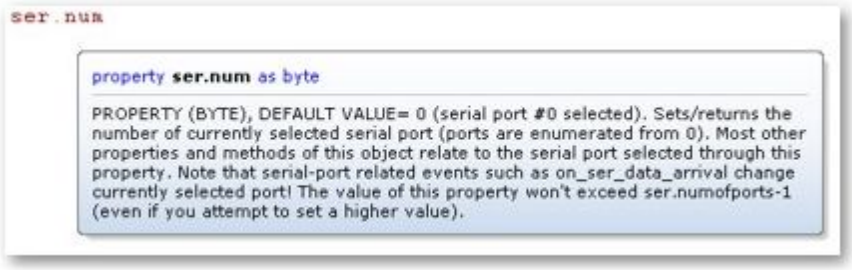
Code hinting is a feature which helps you see what are the arguments for a function, while writing the code for it. It appears as soon as you type the opening parentheses for a procedure. It looks like this:

You can see the number of arguments required and their types, as well as the return type (if any). The highlighted part shows what syntax element you should type in next.

- You may invoke code hinting manually by pressing Ctrl+Shift+space.

Tooltips

When you hover your mouse over event handlers, object properties and methods, constants, procedures, and variables the TIDE displays *tooltips*. These look like this:



The tooltips are displayed when hovering over constructs in the code editor, the Project Browser 134, Watch 33, and the Stack 133 pane. They show a formal construct definition and a comment, if available.

- In the code editor, you may also display a tooltip with the keyboard by

pressing **Ctrl+T** when the cursor is within an event handler, a procedure, a constant or a global variable.

Tooltip text for properties, methods, and events comes from the platform file for the platform[82] selected in your project. You can add your own custom comments to the tooltips displayed for procedures and variables of your projects. What's more, you can use HTML formatting to make these comments look more readable! Here is an example:

```
function blink(num as integer) as boolean ' Blinks the lights. Input:
<br><b>num</b>- pattern # to "play". <b><font color=red> Do not set to 0.</
font>.

'... your code here ...

End Function
```

This would yield the following tooltip when hovering over the "blink" identifier (notice how HTML formatting improves tooltip readability):

Supported HTML Tags[26] section details which tags you can use to beautify your tooptips.

Your comment must be on the same line as the function definition, or immediately following it. The comment can contain multiple lines, if every line begins with a comment character. These lines must be consecutive -- with no blank lines in between. For example:

```
function blink(num as integer) as boolean ' USER-DEFINED. Blinks the lights.
' This is a very important function.

' And must be included in every application.

'... your code here ...

End Function
```

This would yield a tooltip with the text "USER DEFINED. Blinks the lights. This is a very important function." This would be as one paragraph -- line breaks are not displayed within the tooltip, unless you use **
** element. The third comment would not be included because it is preceded by a blank line.

As we will explain in the Introduction to Procedures[66], there is a procedure definition (procedure body) and procedure declaration that merely states that the procedure exists. If both have comments than the comment in the procedure definition wins (takes precedence) over the comment in the procedure declaration.

Variables also have declaration (**declare** 84 statement 84) and definition (**dim** 86 s 86 tatement 86). Comment in the definition wins.

Finally, your own comment placed in the event handler definition takes over the comment for this event that comes from the platform file.

Supported HTML Tags

Here are the tags (HTML elements) that you can use:

- Presentation markup tags: ****, **<i>**, **<big>**, **<small>**, **<s>**, **<u>**, **<font** [color=*color*] [size=*size*] [face=*face*]**>**
- Headings **<h1>**...**<h6>**
- Line break **
**
- Comments: **<!-- -->**

All other tags (elements) cannot be used. Most of these tags are simply ignored, but some lead to scrambled text output.

One peculiarity of the HTML renderer used in the TIDE software is that it often requires you to add an extra space before the closing tag in the tag pair. For example, if you write "Bold text" then you will get this output: "**Bold**text". Writing "Bold text" or Boldtext will produce correct result: "**Bold** text".

Making, Uploading and Running an Executable Binary

An *Executable Binary File* is a file (.tpc type) which contains your project in compiled form, along with any resource files. It is uploaded to the target, where it is actually executed by the TiOS Virtual Machine.

Making a Binary

Once you have the code which you wish to try out, you may build it by selecting Project > Build, by pressing the shortcut key F7 or by clicking the Build button on the Debug toolbar.

If this is not the first time you're building this project, the build process will skip any files which were unmodified since the last time the project was built. This optimizes build speed.

To force the build of all files, even those which were not modified since the last time, select Project > Rebuild All.

On build, the Output pane 133 will display any errors. You can double-click on the line describing an error to jump directly to the problematic line in your code.

Uploading a Binary

Before uploading a binary, you must select the target device. You can do so using the Device Explorer 39.

- To upload your project, you must select Project > Upload or click the Upload button on the Debug toolbar.

Before uploading, TIDE checks if the project has been changed since it was last built. If so, it builds the project again and attempts to upload the new build.

Also, the current project hosted on the target will be checked. If it is the same (same project and same build number) as the project you are trying to upload, uploading will not occur. Thus, trying to upload a project twice without making any change in the project will not result in a second upload. Also, before uploading, the firmware version is checked and if it is incompatible with the firmware version specified in the platform file, the upload is aborted with an error message.

As of version 2.20.33, TIDE supports *incremental uploads.* It caches the previous build on the computer; when you make a modification, it (1) makes sure the previous build is indeed what's currently running on your device, and then (2) compares the current build with the previous build, block by block. It then attempts to upload only what's changed. This can make rebuild times significantly faster. Your device needs to be running TiOS 3.10.00 or later to be able to take advantage of this feature.

Running a Binary

For a debug version 38 (the default version type), uploading the binary does not automatically start its execution on the target. Once uploaded, it just sits there, waiting to be executed.

- To begin execution, select Debug > Run, press the shortcut key F5 or click the Run button on the Debug toolbar.

This action optionally builds and uploads your application, if needed. If a new upload was just performed, it also *reboots* the target before running it. This ensures that target starts running the newly uploaded program from a 'fresh' state.

- You may also reboot your device manually at any time by selecting Debug > Restart or clicking the Restart button on the Debug toolbar.
- These actions are *incremental.* This means that when uploading, a build is performed if needed. When running, a build and an upload are performed if needed.

Two Modes of Target Execution

When you execute a program on the target, it can run in either of two modes (depending on the setting selected under Project Settings) 38:

Debug Mode

In debug mode, your project runs with the assumption that you are right there, watching the monitor and trying to see what's going on. This means that the Debug menu is active. You can set up breakpoints 30, or step 33 through your project, watch the variables, etc.

This also means the project might stop if there's an error, such as division by 0. And you can pause execution, stop it, etc.

Also, when uploading a project in debug mode, it does not begin to run by default.

By default, it waits for you to run it.

Selecting Debug > Run, pressing F5 or clicking the Run button on the Debug toolbar would send an explicit command to the target, to start running the project.

If the device reboots while a project is running in Debug Mode, the project will not start running automatically after the reboot. You would have to run it explicitly.

Release Mode

Release mode means business. This is the mode in which you compile the final files, deployed in the field. Under this mode, the working assumption is that you, or anybody else, isn't there. Your box is just supposed to run and run, despite any and all problems and errors.

This means that a release version does not respond to any debug commands. You cannot stop it. It does not stop even when critical errors occur. It also means that when you upload a release version to your target, it starts running immediately.

Even if you reboot your device, when it has a Release Mode binary in memory, it will start running.

Debugging Your Project

One of the most common operations you will perform during your development process is *debugging*. In essence, this involves controlled execution of your project. While debugging you can step through your program, set breakpoints, watch and change the state of various variables, see how control and decisions statements are executed, etc.

One of the aspects of TIDE is that it employs a technique called *cross-debugging*. Simply put, this means your code runs on a different machine than the one on which you wrote it, and you can debug it from the computer on which you wrote the program.

Thus, code is not debugged using some PC emulator or anything of this sort. It is truly uploaded and run on your target -- just like it would run in real life.

As covered above 26, the first thing you would have to do to begin debugging a debug binary would be to run it, using F5. Once you press F5 (or Debug > Run), your project will be built (if necessary), uploaded (if necessary) and started.

Once execution has started, there are several ways in which you may control and inspect it. These are listed below.

Target States

In debug mode, your target may be in one of several states at any given moment.

For this, the status bar 130 displays several different status messages:

Run: This message means your program is currently running. It doesn't mean any specific code is actually being executed -- perhaps the target device is just sitting idle, waiting for an event to happen. But the program is still running -- not paused. This state is entered by pressing F5 or Debug > Run.

Break: This message occurs when the Virtual Machine on the target was stopped while executing code. The easiest way to get to this state is by setting and reaching a breakpoint 30 in code. You might also get to this state by selecting Debug > Pause, if you happen to catch the Virtual Machine in the midst of code execution. Once in this

state, the program pointer 30 (a yellow line) is displayed and indicates the next instruction that the Virtual Machine will execute when started. You can now inspect and change various properties and variables (both global and local) using the watch 33. This is the only state which allows stepping 33.

Pause: This message occurs when the Virtual Machine on the target was stopped while not executing code (in other words, it was caught between events). No program pointer is displayed in this state, because no code is being executed. You can check and modify the state of properties and global variables, etc using the watch. This state is entered by selecting Debug > Pause.

Abort (exception): This message indicates that an internal error has occurred. The message in parentheses is a short error code. If you hover your mouse over it, you will see a more detailed report of the error. The program pointer will appear at the problematic line. This state is similar to a break, only it is not caused by a breakpoint but by an abnormal condition. All possible causes for exception are listed in Exceptions 29.

Communication States

While TIDE is in communication with the target, the status bar displays a moving indicator of the communication state.

Communication in progress: The circle is green, and moves from side to side. It advances one step whenever TIDE gets a reply to a debug command.

Communication problem: The circle is yellow, and does not move. This state means TIDE did not receive any reply to debug commands for more than 6 seconds. The program may be still running on the target.

No Communication: The circle is red, and does not move. Occurs when TIDE did not receive any reply to debug commands for more than 12 seconds. The program may be still running on the target.

Exceptions

Exceptions are "emergency" halts of program execution. Exceptions are generated when the Virtual Machine encounters something that really prevents it from continuing normal operation. When exception happens you see "ABORT" target state in the status bar, like this:

"(DIV0)" is an abbreviated problem description. Hover the mouse over this and you will get a more detailed description.

Listed below are all possible exceptions. When you are in the debug 27 mode any exception from the list below causes the Virtual Machine to abort execution. In the release 27 mode, some "lesser" problems do not cause the halt. The logic here is that there will probably be nobody to restart the problem or check what happened anyway, so the Virtual Machine just tries to continue operation.

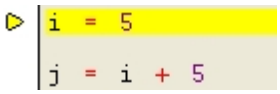
Code	Description	Halt in debug	Halt in release

		mode?	mode?
DIV 0	Division by zero	Yes	No
OOR	Out of range (attempt to access past the highest array member)	Yes	No
FPERR	Floating point error	Yes	No
IOC	Invalid opcode*	Yes	Yes
OUM	Access outside of user memory*	Yes	Yes
TDLF	Failed to load binary library*	Yes	Yes

**This exception indicates that either* TiOS *or* Tibbo Basic *compiler is not functioning properly. Let us know if you encounter this exception!*

Program Pointer

The *program pointer* is a line, highlighted in yellow, which shows the present location of program execution. It looks like this:



This means that the yellow line is now pending execution. It hasn't been executed yet. The machine is waiting for you to tell it what to do. You can now control it by stepping 33.

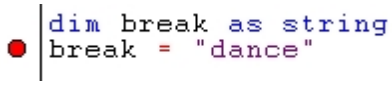
This line is displayed whenever the Virtual Machine has been paused while executing code. This can be achieved by setting a breakpoint 30, or simply selecting Debug > Pause at the "right" time.

The program pointer will only stop on lines which contain *actual executable code*.

```
dim x as byte ' the program pointer won't stop here, as this isn't
executable code.
x = 1 ' the program pointer will stop here -- this is an actual instruction
to do something.
```

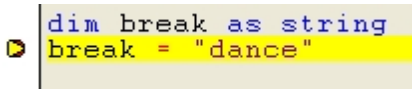
Breakpoints 3

A *breakpoint* is a point marking a line of code in which you wish to have the debugger pause. It is seen as a little red dot on the left margin of the code. Like this:



This is what it looks like when the code is not executing.

Once the code begins to execute and the breakpoint is reached, the program pointer 30 is displayed at the line in which the breakpoint is placed:



The yellow arrow over the red dot merely marks the program pointer; a breakpoint is always marked by a red dot.

Where a Breakpoint May Be Placed

A breakpoint may be placed only on a line which contains executable code; before compiling your project, you could place breakpoints anywhere. However, on compile, these breakpoints will be shifted to the nearest lines following them which contain executable code.

You could add breakpoints to your code at any time -- even while the code is running. However, while the code is running, you may only add breakpoints next to lines which contain executable code. If you click next to a line which does not contain executable code, the closest line following this line which does contain executable code will get a breakpoint.

You may have up to 16 breakpoints in your entire project. Breakpoints are saved when the project is saved.

- Adding breakpoints slows down the performance of the Virtual Machine. Having 16 breakpoints will have a noticeable effect on the speed of execution of your program.

Toggling Breakpoints

Breakpoints may be toggled (set/cleared) by putting the cursor in the line in which you wish to place (or remove) the breakpoint and pressing F9 or selecting Debug > Toggle Breakpoint. Alternatively, you may also toggle a breakpoint by clicking on the margin of the code at the point in which you wish to have a breakpoint.

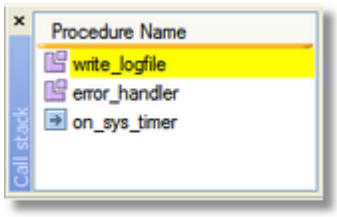
You may remove all breakpoints from an entire project (including any files which are not currently open) by selecting Debug > Remove All Breakpoints.

The Call Stack and Stack Pointer

During the execution of a program, procedures usually call other procedures. The calling procedures are not just left and forgotten; they are placed on the *call stack*.

The call stack can be toggled by View > Call Stack. It lists the sequence of procedure calls which lead to the current procedure (the one in which the Program Pointer 30 is located). The current procedure is listed at the top of the stack. Once it finishes executing, execution returns to the caller (the procedure one line below in the stack). In the caller procedure, execution resumes from the line immediately following the one which called the procedure which has just ended.

So, if the procedure (event handler, actually) **on_timer** called the procedure **error_handler**, which in turn called the procedure **write_logfile**, our call stack would look like this:



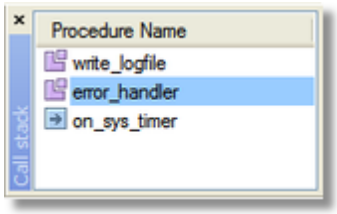
When pausing a program in the midst of code execution, the program pointer 30 appears. In the Call Stack pane, the function which currently contains the program pointer is highlighted in yellow. It is the currently executing function, so it is always the first one on the Call Stack list.

* Technically speaking, the top function on the call stack isn't actually a part of the *stack* itself, because it is currently executing, and the real stack only contains functions to which execution should later return. It still appears on the same list, for consistency and convenience.

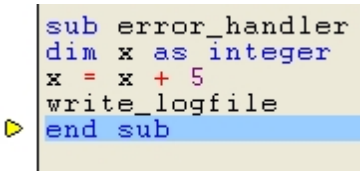
The Stack Pointer

Double-clicking on any procedure within the call stack which is not the currently executing procedure would display the *stack pointer.* This pointer would be displayed in the source code, within the procedure you double-clicked, and would highlight the line from which execution would resume once control returns to this procedure. The watch pane 33 would also interpret variables as relative to the procedure you've just highlighted.

The stack pointer is light blue in color. On the call stack list, it looks like this:



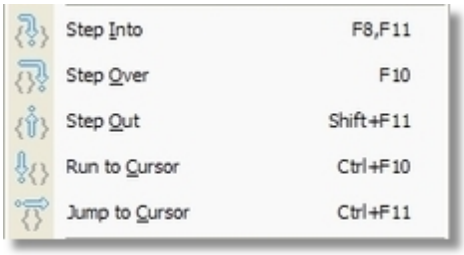
In the code editing pane, it looks like this:



Once again, double-clicking on the functions in th call stack *does not move actual execution* (the program pointer). Any sort of stepping would bring back the yellow program pointer, both in the source code and in the call stack.

Stepping 1.8.5

The following commands in the Debug menu control stepping into, through and out of various sections of your code:



A *step* is an instruction to move the program pointer 30 to another line in source code. Stepping allows you to execute your code in a very controlled way, and work your way along the program in a pace which you can analyze and understand.

- **Step Into:** Steps through your code line by line. When the program pointer reaches a procedure call, you would actually see it step *into* this procedure (hence, the name). You could then see the inner workings of this procedure as it is being executed, line by line.
- **Step Over:** Steps through your code line by line. When the program pointer reaches a procedure call, it executes this entire procedure, but just does it all at once, and stops at the next line after the procedure call. This is useful when you want to debug a body of code which contains a call to a complex or lengthy procedure, which you do not want to debug right now.
- **Step Out:** If you are currently stepping through a function and wish to exit it while you're still in the middle, use Step Out. This would bring you to the line immediately following the line which called the function you were in. This option is disabled when you cannot step out of the current function (i.e, when your other function calls it -- such as in the case of an event handler 8).
- **Run to Cursor:** The *cursor*, in this case, is the text insertion point. The little blinking black line. You can place the cursor anywhere within the body of your program and have the program execute until it reaches that point. When, and if, that point is reached, the program pointer would display.
- **Jump to Cursor:** This command makes the program pointer unconditionally move to the point where the cursor is. It will just *jump* there, possibly skipping the execution of some code. This is explicit control of program flow.

The Watch 8.6

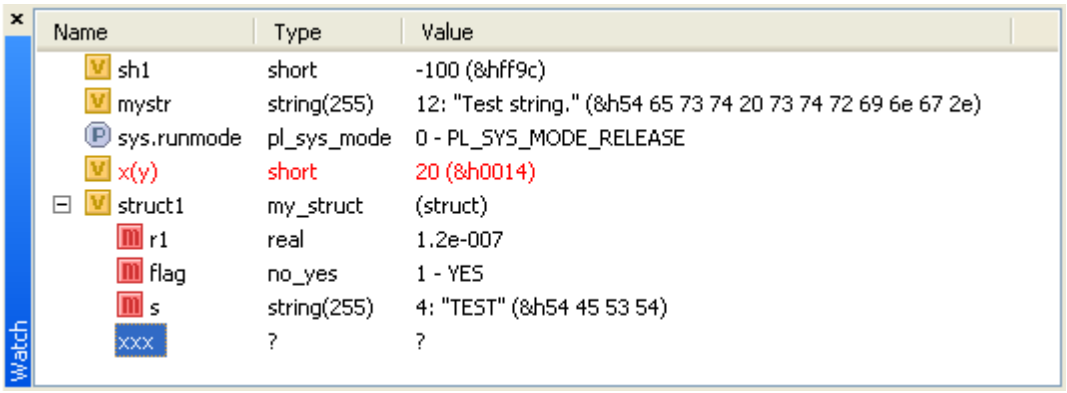
The *watch* is a facility which allows you to inspect and change the current value of variables and object properties. You can only use this facility when you are in debug mode and when the program execution is stopped. TIDE is unable to fetch variable values while the Virtual Machine is running. Depending on the scope of the variable, there may be additional limitations 37 as to when you can inspect this variable's value.

The watch updates variable values by reading them from the target every time the Virtual Machine is stopped. If any item on the list changed its value since the previous fetch, this item will be displayed in red.

Watch facility may be accessed by three different ways:

The Watch Pane

The watch can be toggled by View > Watch. It looks like this:

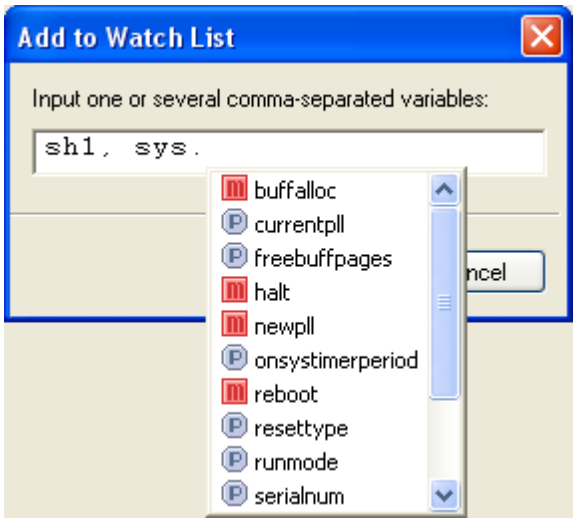


On the above screenshot we can see the status of a **short**, a **string**, a **property** of an object, a member of an **array**, one user-defined **structure**, and an undefined **variable**. Notice, that the "x(y)" is displayed in red indicating that its value has just changed. Notice also that this line shows an array indexed by another variable (y). This is not the limit of the watch pane's abilities -- try entering more complex expressions, it will work!

There are several ways of adding variables and object properties to the watch list. You can:

- Press Insert while the watch pane has focus -- this will bring up an Add to Watch List dialog. Type the name of a variable or property to watch and press OK.
- Alternatively, double-click on the empty space in the watch pane to obtain the same result.
- You can also select Debug > Add to Watch List from the Menu.
- Additionally, there is an Add to Watch List button on the Debug toolbar.
- You can right-click on the variable or property in the source code and select Add to Watch List from the context-sensitive menu.
- You can also right-click on the property in the Project pane (Browser tab) and select Add to Watch List.

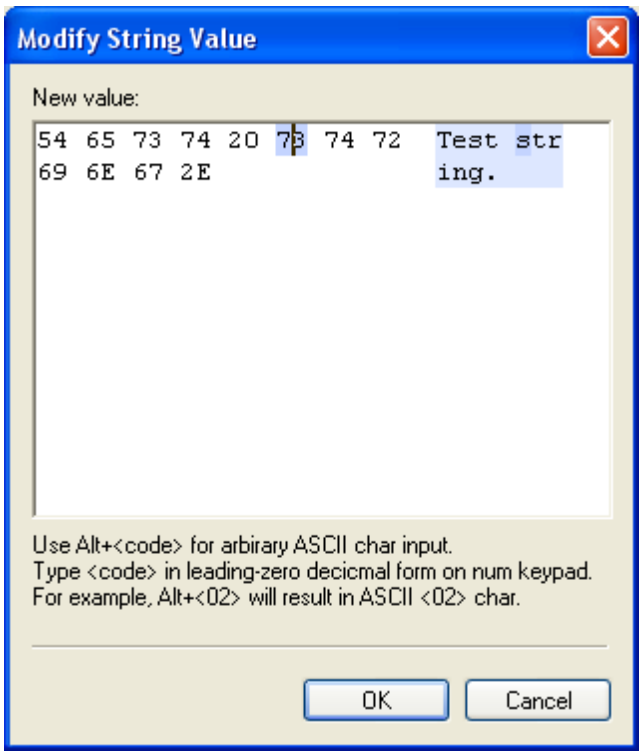
In the Add to Watch List dialog you may type multiple items to be added to the watch by separating their names with commas (i.e. "i, j, k, ser.num"). For objects you will get a drop-down list of available members:



Double-clicking the Name column in the watch pane will allow you to edit the name of the item you want to watch.

The watch pane also allows you to change the value of any variable or object property (provided it is not a read-only property). Double-click on the value field - you will be prompted for a new value.

For numerical variables, you may use hex or binary notation (&h, &b). For strings, you will be presented with the "HEX editor" allowing you to modify the string or the HEX codes of its characters:



Once a new value is entered, it will be actually written to memory on the target, and will be read again before being displayed in the watch. So when you see your new value in the value column, that means it's actually in memory now -- what you

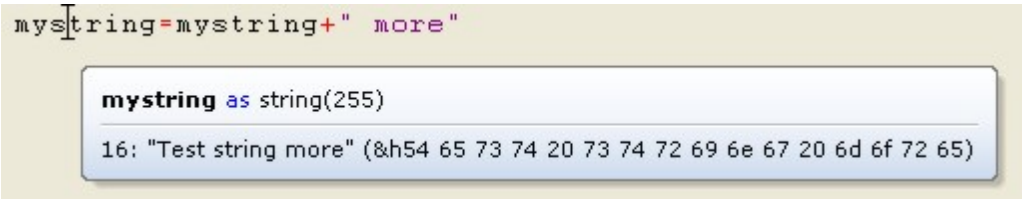
see is what you get.

The watch pane is one of the places where enumeration types become very useful. Look at sys.runmode above. Because its possible values are described through an enum, you can see a meaningful state description, rather than just a number. This is one of the main reasons for the existence of enumeration types in Tibbo Basic.

To remove a variable from the watch, select it and press Delete, select Debug > Remove from Watch List or click the Remove from Watch List button on the Debug toolbar.

The Watch Tooltip

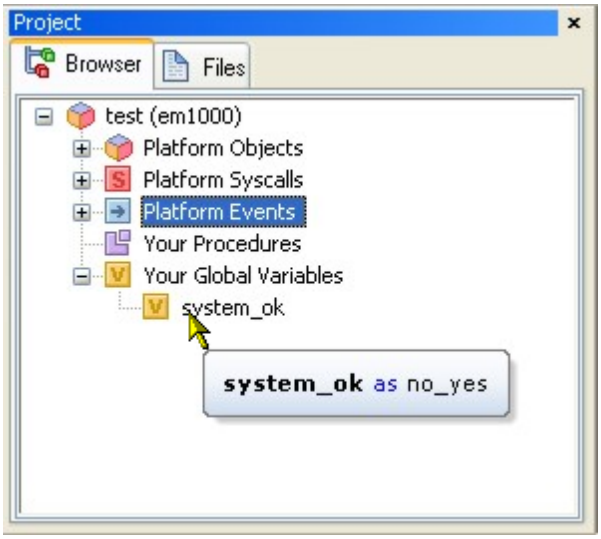
This is the watch tooltip:



When you hover the mouse cursor over an identifier in the source while in debug mode you will see a tooltip with the current value for this identifier.

For now, this won't work for arrays. If you want to inspect the values of an array add this array to the watch list.

The Project Browser



The Project Browser is a tab in the Project pane, which can be toggled by View > Project. It displays all objects for your platform, with the properties and methods for each object. It also displays all procedures and global variables in your project.

While debugging, you can see the value of an object property or a global variable by hovering over its identifier in the tree.

Scopes in Watch

The watch facility is only active when the Virtual Machine is not running, i.e. the execution is stopped. Naturally, the TIDE cannot fetch variable values while the Virtual Machine is executing your program.

You already know that when the Virtual Machine it stopped, the state of your target 28 may be either "BREAK" or "PAUSE". Properties and global variables may be inspected in the either state. Local variables only exist in their *context*. Hence, a particular local variable can only be inspected when the state is "BREAK" (the Virtual Machine is in the middle of a code execution -- there is an execution pointer visible) and when this variable exists in the *current context*.

For example:

```
sub sub_one
      dim x as byte
      x = 1
end sub

sub sub_two
      dim x as byte
      x = 2
end sub

sub sub_two
      dim i as integer
      i = 5
end sub
```

Let us say you add x to the watch list.

When the execution pointer highlights the **end sub** keyword for sub_one, the value of x in the watch would be 1. When the pointer highlights the **end sub** for sub_two, the value of x in the watch would be 2. Note that these are two *different local variables!*

Similarly, when the execution pointer is at the **end sub** of sub_three, the x variable in the watch will be undefined -- it will display a question mark.

These same rules apply to the watch tooltip, as well. Even if you hover the mouse over the x of sub_one when the program pointer is at the end of sub_two, the value displayed would be the one set in sub_two -- because this is the current context!

Code Profiling

Code Profiling is the practice of measuring how long it takes for different portions of your program to execute.

This is the timer, located on the Status Bar.

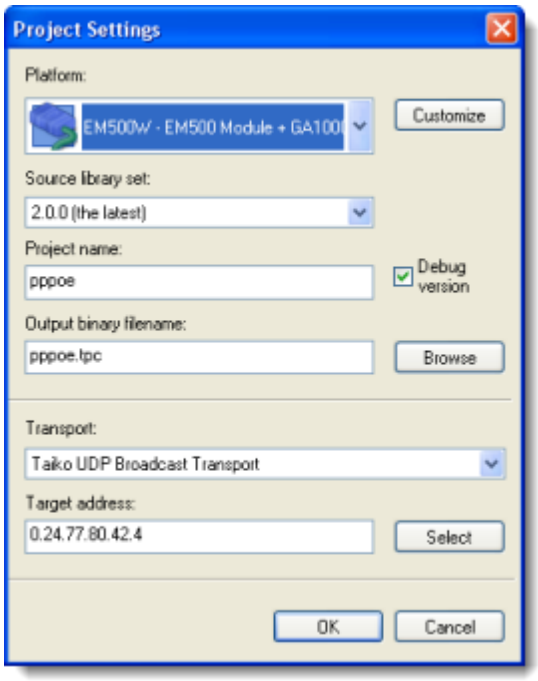
Each time execution begins or resumes in debug mode, the timer is reset and starts counting. Once execution is stopped for whatever reason, the timer displays the time elapsed since execution has begun or resumed.

Remember, as covered above 30, every additional breakpoint somewhat slows

down the speed of execution of your project.

Project Settings

The Project Settings dialog is platform specific. It can look like this:



Platform: This is the platform selected for the project. In the image above, the EM1000 143 platform is selected.

Customize: click to access the list of option available for your platform. These are "global defines" (for a preprocessor) 76, which typically include options such as whether a display is present, display type, etc. For example, on the EM1000 platform you can enable/disable display support (lcd. 317 object), flash disk support (fd. 236 object), and keypad support (kp. 304 object). Additionally, you get to choose the type of the display panel. Disabling objects when they are not in use reduces the size of your compiled application binary.

Source Libraries Version: Tibbo libraries come in sets. Each set has a version number. Once you choose the library set for your project, it will be used until you change it. Tibbo will keep releasing new library sets. With each new release we will check if backward compatibility has been lost due to new features. If backward compatibility is lost, we will publish the new library set under the new version number. Old library sets will still be distributed and your project will keep compiling without any errors. For new projects, always choose the current library set as defined in the Library Reference 580.

Project name: A descriptive name for the project.

Debug version: If checked, the compiler will build a version of this project which can be debugged using the various debugging facilities within TIDE, such as step, watch, etc. Also, debug versions do not automatically start running after the upload or reboot. Release versions run automatically and can't be debugged. By default, this option is checked. Once you've debugged your project and wish to deploy it, uncheck this and build your final version.

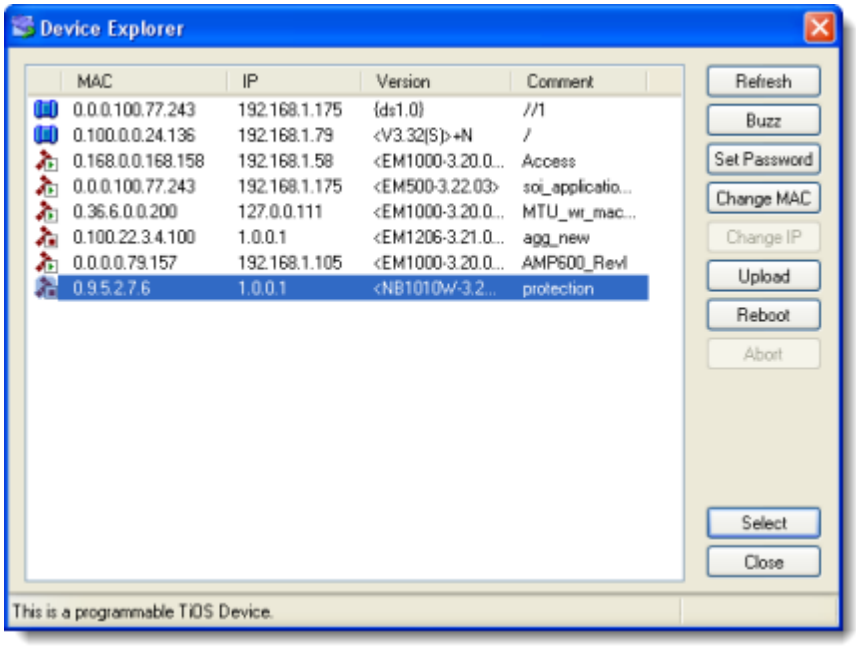
Output binary file name: The name for the .bin file of the compiled project.

Transport: Selects the way that TIDE will use to communicate with your target. Available selections are platform-specific. See your device's platform documentation for more information on available debugging means. Also, Debug Communications 204 topic lists all currently available debug transports.

Target address: Different platforms may use different communication media between TIDE and target. Thus, their target address may be expressed differently. For devices with Ethernet port, the target address is the MAC address of the device.

Select button: click it to select the target device's address. For Ethernet devices, clicking the button brings up the Device Explorer 39.

Device Explorer



The Device Explorer is actually a standalone program. It can be found in the TIDE installation folder and launched independently. It is also accessible from within TIDE as a dialog (Project > Select/Manage Target).

The Device Explorer shows all Tibbo devices found on the current network segment. These may include serial-over-IP devices and programmable devices running TiOS. Some discovered devices may have two entries -- this happens when you have a TiOS device running serial-over-IP Tibbo BASIC application.

Only TiOS devices can be debug targets.

Device list columns

Icon: The icon to the left of the MAC address shows the status of the device. It can be any of the following:

- Serial-over-IP device, valid IP address.

- Serial-over-IP device, invalid (unreachable) IP address.
- <%TIOS% device, no Tibbo BASIC application in its memory.
- <%TIOS% device, Tibbo BASIC application is loaded and running (executing).
- <%TIOS% device, Tibbo BASIC application is loaded and stopped.
- <%TIOS% device, Tibbo BASIC application is loaded and paused (for debugging).

MAC: This is the current MAC address of the device. It can be changed.

IP: The current IP address of the device. It can be changed for serial-over-IP devices. On <%TIOS% devices the IP can only be changed via Tibbo BASIC code.

Version: The version of the firmware this device is running.

Comment: Additional information. For serial-over-IP devices, this is the owner and device names. For <%TIOS% devices, this is the name of the Tibbo BASIC project currently loaded in memory.

You can click on any column header to sort the list by this column.

- Can't see your device in the list? It may not be connected to the same LAN segment (there is a router between your PC and the device), or your PC's firewall may be interfering. To fix this, configure the firewall to allow broadcast UDP datagrams from TIDE.

Buttons

Button functions apply to the device currently selected (highlighted) in the device list. You can select several devices at once -- **Buzz**, **Upload**, and **Reboot** can be applied to multiple devices simultaneously!

Buttons are...

Refresh: repeat the search for devices.

Buzz: Make the selected device identify itself by "playing" a pattern on its status LEDs 200. This is useful for finding out which physical device corresponds to the selected entry in the list. Serial-over-IP devices identify themselves by quickly flickering their red and green LEDs several times. <%TIOS% devices respond by turning their LEDs off for two seconds, but this only happens when Tibbo BASIC application is not executing. <%TIOS% devices that are running an application will not respond to buzz at all.

Set Password: you can protect the firmware and compiled Tibbo BASIC application loaded into your device by setting an access password 41 (strongly recommended). This password will then be required for MAC address changing, as well as firmware and/or application uploading.

Change MAC: Change the MAC address of the device. Note that you have to reboot the device for the change to become effective.

Change IP: Change the IP address of the device. This function is only available for serial-over-IP devices.

Upload: see Upload Function 41.

Reboot: reboot the device.

Abort: Aborts the current operation. Device Explorer operations never time out --

you must abort them manually.

Select: This button is only shown when accessing the Device Explorer from within TIDE. Selecting the device means that TIDE will use the device as its debug target for the currently opened project. You can achieve the same by double-clicking on the device in the list.

Close: Close the dialog.

Upload Function

Upload button has a fly-out menu:

- **Load Firmware Through the Network:** Upload new firmware, optionally with compiled Tibbo BASIC application binary attached, through the Ethernet interface. This can be simultaneously applied to a group of devices.
- **Load Application Through the Network:** Upload compiled Tibbo BASIC application through the Ethernet interface. This can be simultaneously applied to a group of devices.
- **Load Firmware Through the Serial Port (X-modem):** Upload firmware, optionally with compiled application binary attached, through the serial port of the device. For this to work, the device must be in the firmware upload mode.
- **Load Firmware Through the Debug Serial Port:** No publicly released Tibbo devices currently support this function.
- **Load Application Through the Debug Serial Port:** No publicly released Tibbo devices currently support this function.

Loading firmware vs. loading application

"Load firmware" functions allow you to load the following:

- Firmware for "traditional" fixed-function serial-over-IP devices such as DS203.
- Firmware for TiOS devices.
- Firmware for TiOS devices combined with the compiled Tibbo BASIC application binary. This is a single file that in fact contains two parts.

"Load Application" function allows you to load compiled Tibbo BASIC application binary into TiOS devices.

Protecting Your Device with a Password

Once you have created your Tibbo application, you can start deploying it into multiple devices. At this point, you are recommended to password-protect your device. Since Tibbo devices can be uploaded to and debugged through the network, anyone with the basic understanding of TIDE software can "mess up" with you device in many different ways and without even touching it physically. "They" can simply render the device unusable or, even worse, replace your application with a hacked one one that has additional undesirable "features". All this can be prevented by setting the device password.

Setting device password

Passwords are set with the Device Explorer 39. Select the device in the list, click

Password button to bring up **Change Device Password** dialog. Enter any password you want. From that point on, firmware and application uploads to the device will require this password. TIDE will prompt you to **Enter Device Password** every time it is needed. Check **Save password for this device** in the **Enter Device Password** dialog and TIDE will memorize the password (it will be stored in an encrypted file). This way, you won't have to input the password over and over again.

Your Tibbo BASIC application can also set the password on its own. This can spare you the trouble of manually protecting each device.You can find a code example in Special Configuration Section of the EEPROM 197.

Clearing the password

To remove the password, use the **Password** button of the Device Explorer and set an empty password. This, however, will require you to input the old (current) password as well. And what if you don't remember it? Read on...

When you forgot the password

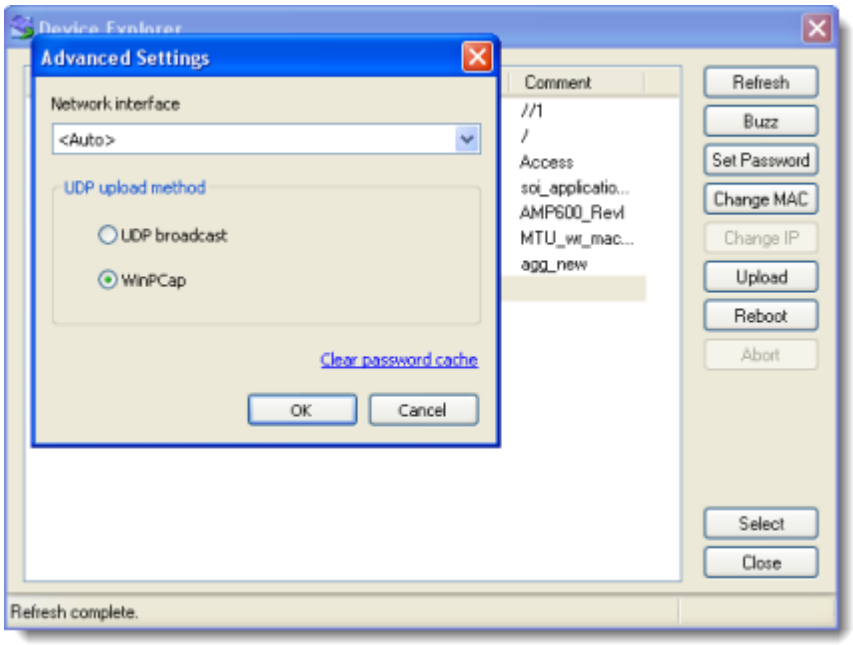
Simply press (and keep pressed) the MD button 201 of your device while clicking **OK** button in the **Change Device Password** or **Enter Device Password** dialog. This will make you device accept any password you enter. Pressing the MD button will not erase the password, but rather bypass it. This way you can change the old password (or remove it completely) without knowing what it was.

How secure is it?

When the password is being set or changed, it is transmitted across the network "in the open". Therefore, set passwords in the secure environment where you can make sure that no one is "eavesdropping". Subsequent transmissions of the password to "login" onto the device (in order to upload firmware, etc.) are "safe". No data is transmitted in the open and eavesdropping on the data transmission will not reveal the password or render any useful data.

How to remove passwords stored by TIDE

Open the Device Explorer and click on its icon (top left corner of the window). Select **Advanced Settings.** Click on **Clear Password Cache**. You are done.



Programming Fundamentals

This chapter attempts to provide a very quick run through the fundamentals of creating a program in Tibbo Basic. It was written under the assumption that the reader has some experience in programming for other languages.

There is a marked resemblance between Tibbo Basic and other types of BASIC that you may already know. Thus, we stressed the differences between Tibbo Basic and other BASIC implementations. Some sections begin with a seemingly introductory statement, but the material quickly escalates into more advanced explanations and examples.

This is not a programming tutorial. We do not attempt to teach you how to program in general. There are many excellent books which already exist on this subject, and we did not set out to compete with any of them. This is a mere attempt at explaining Tibbo Basic -- no more, no less.

Good luck!

Program Structure

A typical Tibbo Basic source code file (a compilation unit 136 with the .tbs 16 extension) contains the following sections of code:

Include Statements

These are used to include 95 other files from the same project (such as header files [.tbh 16] containing global variable definitions or utility functions). See:

```
include "global.tbh"
```

Global Variables Definitions

Here you define any variables you wish to be accessible throughout the current compilation unit:

```
dim foo, bar as integer
```

Subs and Functions

These are procedures which perform specific tasks, and may be called from other places within the project.

```
sub foo
...
end sub

function bar (a as integer) as byte
...
end function
```

Event Handlers

Tibbo Basic is event-driven. Event handlers are platform-dependent subs, that are executed when something happens. This 'something' depends upon your platform. If your Tibbo Basic program runs on a refrigerator, you would probably have an event handler for the door opening.

```
sub on_door_open
      light = 1 ' turn on the light when someone opens the door to your
fridge.
end sub
```

Just like any other sub, events can have arguments (input parameters):

```
sub on_door_state(state as byte)
   ' turn the light on and off as the fridge door gets opened and closed
   if state=0 then
      light=0
   else
      light=1
   end if
end sub
```

Event handles are always subs and never functions (i.e. they never return any value as there is nobody to return it to)!

Code Basics

There are several important things you should know about writing in Tibbo Basic:

You Can Put Comments in Your Code

The apostrophe character marks the beginning of a comment. Anything following this character until the end of a line is considered to be a comment, and will not be processed by the compiler 136.

```
x = 1 + 1 ' I am a comment!
' x = y/0 <--- this line would not cause an error, because it won't even
execute! It is commented.
```

The only exception to this is that when including an apostrophe within a string (between double quotes) it is not counted as a comment. See:

```
s = "That's a string!" ' Notice that the word that's contains an apostrophe.
```

Comments cannot span multiple lines. A line break terminates a comment. If you want to make a multi-line comment, each line of your comment must begin with an apostrophe.

Tibbo Basic Doesn't Care About Spaces!

See:

```
y =    x     +              5 ' is just like

y=x+5 ' and even this is OK:

y =
x
+
5
```

So, spaces, tabs and linefeeds carry no special meaning in Tibbo Basic. Use them or lose them, as you like. The only exceptions are the If.... Then... Else Statement 94 and comments.

Tibbo Basic Is Not Case Sensitive!

See:

```
Z = X + Y ' is just like
z = x + y
r = Q + KeWL ' even something like this is legal.
DiM MooMoo As iNTeGER ' this, distatesful as it may be, is still legal. :)
```

Capital letters just don't matter. Really.

How to Use Double Quote Marks

Double quote marks are used for marking string literals. Simply put, a *string literal* is a constant string value -- like "hello world".

```
s = "I am a string literal!"
s1 = s
```

How to Use Single Quote Marks

These are different than the apostrophes which begin a comment. An apostrophe looks like this ' while a single quote mark looks like this ` and is usually located on the tilde (~) key. Single quote marks are used to define a numerical constant which contains the value for an ASCII code. For example:

```
dim b as byte
dim w as word
b = `q` ' the variable b now contains the value 113, which is the character
code for q.
w = `LM` ' this is also legal, see below.
```

The notation used in the second example above actually places the ASCII value of L into the higher byte of a word-type 48 variable, and the ASCII value of M into the lower byte of that variable. It may seem confusing at first, but it is also legal.

Note that a this isn't the same character as an apostrophe. An apostrophe is ' and a single quote-mark is a `. Usually, the single quote mark is found on the tilde (~) key, and the apostrophe is found on the double-quote (") key.

How to Define Constants In Different Bases

Tibbo Basic allows you to assign values to constants using decimal, hexadecimal or binary notation. Decimal notation is the default. To assign a hexadecimal value, you must use the prefix **&h** before the first hexadecimal digit of your value. To assign a binary value, you must use the prefix **&b** in the same manner. So:

```
q = 15 ' this is 15, in decimal.
q = &hF ' this is still 15, just in hex.
q = &b1111 ' and this is also 15, just in binary notation.
```

These prefixes hold true whenever values are used -- there are no exceptions to this rule. Whenever a numeric value is used, it may be preceded by one of these prefixes and will be interpreted correctly.

How to Use Colons

Colons are actually not necessary in most parts of Tibbo Basic. They are a traditional part of many BASIC implementations, and are often used to group several statements in one line. However, since Tibbo Basic doesn't really care about spaces anyway, they lose their relevance.

No Tibbo Basic statements require the use of colons.

Naming Conventions

Identifiers

An identifier is the 'name' of a constant, a procedure or a variable. It is case insensitive. It may include letters (A-Z, a-z), digits (0-9) and the following special characters: . ~ $!. It must start with a letter, and cannot contain spaces.

For example:

```
dim a123 as integer ' A legal example
dim 123a as integer ' An illegal example
dim a.something as integer ' Can contain dots.
sub my_sub (ARG1 as integer, ARG2 as string) ' This is also legal
```

Platform Objects, their Properties and Methods

The name of an object is used as a prefix, followed by a dot, followed by the name of the property or the method you would like to access. For example:

```
ser.send ' Addressing the 'send' method of a 'ser' object.
x = ser.baudrate ' Assigning variable X with the value of the 'baudrate' of
a 'ser' object
```

Events

Events are related to specific objects, just like properties and methods. However, events are named differently. The pattern is on_*objectname_eventname*. Such as on_door_open.

```
sub on_ser_data_arrival ' event names may contain more than one word after
the object name.
```

Introduction to Variables, Constants and Scopes

Variables and constants are a major part of any programming language, and Tibbo Basic is no exception; below you will find explanations on the following:

Variables And Their Types

Variables are used to store values during the execution of an application. Each variable has a *name* (the identifier 137 used to refer to the value the variable contains) and a *type*, which specifies how much data this variable can contain, and also what kind of data it may contain.

Not every variable type is supported on every platform. You will find related information in the "Supported Variable Types" topic in the platform documentation (for example, EM1000's is here 143). If you attempt to use a type which is not supported by your platform you will most probably get "platform does not export *XXX* function" error during compilation.

Variables are defined using the Dim Statement 86 prior to being used in code. The simplest syntax for defining a variable would be something like:

```
dim x as integer ' x is a variable of type 'integer'.
dim str as string(32) ' str is a variable of type 'string' with a maximum
length of 32 characters (bytes).
```

A variable name must begin with a letter, and must be unique within the same scope, which is the range from which the variable can be referenced.

When you define a variable, some space is reserved for it in memory; later, while the program executes, this memory space can hold a value.

Variables are assigned values like so:

```
x = 15 ' x is now 15.
x = y ' x is now equal to y.
str = "foobar" ' str now contains the string 'foobar' (with no quotes).
```

Types of Variables

Tibbo Basic supports the following variable types:

Name	Description
byte	Hardware-level. Unsigned. Takes 1 byte in memory. Can hold integer numerical values from 0 to 255 (&hFF).

word	Hardware-level. Unsigned. Takes 2 bytes in memory. Can hold integer numerical values from 0 to 65535 (&hFFFF).
dword (new in V2.0, not available on all platforms)	Hardware-level. Unsigned. Takes 4 bytes in memory. Can hold integer numerical values from 0 to 4294967295 (&hFFFFFFFF).
char	Hardware-level. Signed. Takes 1 byte in memory. Can hold integer numerical values from -128 to 127.
short	Hardware-level. Signed. Takes 2 bytes in memory. Can hold integer numerical values from -32768 to 32767.
integer	Compiler-level. Synonym for **short**; substituted for **short** at compilation. Exists for compatibility with other BASIC implementations.
long (new in V2.0, not available on all platforms)	Hardware-level. Signed. Takes 4 bytes in memory. Can hold integer numerical values from -2147483648 to 2147483647
real (new in V2.0, not available on all platforms)	Hardware-level. Signed, in standard "IEEE" floating-point format. Can hold integer and fractional numerical values from +/- 1.175494E-38 to +/- 3.402823E+38. Real calculations are intrinsically imprecise. Result of floating-point calculations may also differ slightly on different computing platforms. Additionally, floating-point calculations can lead to floating-point errors: #INF, -#INF, #NaN. In the debug mode, any such error causes an FPERR exception 29.
float (new in V2.0, not available on all platforms)	Compiler-level. Synonym for **real**; substituted for **real** at compilation. Exists for compatibility with other BASIC implementations.
string	Hardware-level. Takes up to 257 bytes in memory (max string size can be defined separately for each string variable). Strings can actually be up to 255 bytes long but always begin with a 2-byte header -- 1 byte specifies current length, and 1 byte specifies maximum length. Each character is encoded using an ASCII (single-byte) code.
boolean	Compiler-level. Intended to contain one of two possible values (**true** or **false**). Substituted for **byte** at compilation.
user-defined structures(new in V2.0)	Each user-defined structure can include several *member variables* of different types. More about structures here 58.
user-defined enumeration types	Compiler-level. These are additional, user-defined, data types. More about these under User-Defined Types 59.

* *Hardware-level* types are actually implemented on the machine which is used to run the final program produced by the compiler.

Compiler-level types are substituted by other variable types on compile-time. The actual machine uses other variable types to represent them; they are implemented for convenience while programming.

Type Conversion

Variables can derive their value from other variables; in other words, you can assign a variable to another variable. A simple example of this would be:

```
dim x, y as byte
x = 5 ' x is now 5
y = x ' y is now 5 as well
```

However, as covered above, there are several types of variables, and not all of them can handle the same data. For example, what would happen if you assigned a variable of type **byte** the value intended for a variable of type **word**?

Table below details all possible conversion situations.

		Convert into							
		Byte	**Word**	**Dword**	**Char**	**Short**	**Long**	**Real**	**String**
Convert from	**Byte**	---	**OK**	**OK**	Reinterpret	**OK**	**OK**	**OK**	**OK** str 225
	Word	Truncate	---	**OK**	Reinterpret Truncate	Reinterpret	**OK**	**OK**	**OK** str 225
	Dword	Truncate	Truncate	---	Truncate	Truncate	Reinterpret	**OK**	**OK** lstr 216
	Char	Reinterpret	Reinterpret	Reinterpret	---	**OK**	**OK**	**OK**	**OK** stri 226
	Short	Reinterpret Truncate	Reinterpret	Reinterpret	Truncate	---	**OK**	**OK**	**OK** stri 226
	Long	Reinterpret Truncate	Reinterpret	Truncate	Truncate	Truncate	---	**OK**	**OK** lstri 217
	Real	~~Fraction~~ ???	~~Fraction~~ ???	~~Fraction~~ ???	~~Fraction~~ ???	~~Fraction~~ ???	~~Fraction~~ ????	---	**OK** fstr*
	String	**OK** val 229	**OK** val 229	**OK** lval 218	**OK** val 229	**OK** val 229	**OK** lval 218	**OK** str tof 228	---

**fstr is a functional equivalent of ftostr 211, but without mode and rnd parameters.*

Conversions without loss

Conversions marked as "**OK**" incur no loss -- the value being passed from variable of one type to variable of another type remains unchanged. For example, conversion from **word** into **dword** is done without any loss, because 32-bit **word** variable can hold any value that the 16-bit **word** variable can hold.

Conversions that cause reinterpretation

Conversions marked with "Reinterpret" mean that although binary data held by the receiving variable may be the same this binary variable *may* be interpreted differently on the destination "side". As an example, see this conversion from **byte** into **char**:

```
dim x as byte
dim c as char
x = 254
c = x ' c now contains the binary value of 254, which is interpreted as -2
```

In the above example, both x and c will contain the same binary data. However, c is a signed 8-bit value, so binary contents of 254 mean -2. Strictly speaking, this reinterpretation will only happen if the value of x exceeds maximum positive number that c can hold -- 127. If x<=127 conversion will not cause reinterpretation. For example, if x=15 then doing c=x will result in c=15 as well.

In fact, in some cases, conversion from unsigned type to a signed time will never result in the reinterpretation. This is when the maximum value that the source unsigned variable can hold can always fit in the range of positive values that the signed destination variable can hold. Example: conversion from byte (value range 0-255) to short (value range -32768 to 32767) will never result in the reinterpretation.

Conversion from signed type into unsigned type will always cause reinterpretation if the source variable contained a negative value.

Conversions that cause truncation

Conversions marked with "Truncate" mean that part of the binary data (on the most significant side) *may* be lost during the conversion. For example, converting from **word** type into **byte** type will only leave 8 bits of the original 16-bit value:

```
dim x as byte
dim w as word
w = 12345 'hex representation of 12345 is 3039
x = w ' now x contains 57. Why? Because only '39' of '3039' could fit in,
and decimal of &h39 is 57.
```

Notice, that some conversions will cause reinterpretation and truncation at the same time!

Conversions that round the number (remove fractions)

Conversions from real type into any other numerical type will cut off the fraction, as real is the only type that can hold fractions. Such conversions are marked as "Fraction" in the table above.

Conversions that implicitly invoke functions available to the user

Some conversions automatically invoke functions (syscalls) available for use in your program. In such cases the table above lists the name of the function invoked. For example, conversion from byte into string relies on the str 225 function. Two ways of conversion below produce identical result:

```
dim x as byte
dim s as string
s = str(x) ' explicit invocation
s = x ' implicit invocation. Compiler is smart enough to use str for this
conversion
```

Conversion of Boolean Variables

Boolean variables are actually stored as **byte** type variables; thus, all notes above for **byte** type variables hold true for **boolean** variables as well.

Conversion of Enumeration Types

User-defined enumeration types are held in various variable types, depending on the values associated with the constants within the enumeration type. This is described in detail under User-Defined Types 59. Thus, they are converted according to the variable type used to store them (described above).

Type conversion in expressions

In the Type Conversion 50 we already explained what happens when you assign the value of a variable of one type to a variable of another type. This section deals the cases where variables of different types are used in expression. For example, if x is **byte** and i is **integer**, what will happen when you do "x+i"?

The Virtual Machine operates on 16-bit values by default

Native data width for the Virtual Machine is 16 bits. When you are performing calculations on 8-bit and/or 16-bit variables, result is always truncated to 16 bits. Also, all intermediary calculations are done using 16-bit arithmetic. Consider the following example first:

```
dim x,y,z as byte
x=150
y=200
z=(x+y)/10 ' result of 35 is within byte variable's range, but intermidiary
calculations require 16-bit ariphmetic.
```

The above example will give correct result. Even though all three variables are of **byte** type, internal calculations are 16-bit, so when x+y produces 350, which is beyond the range of the **byte** variable, the Virtual Machine handles this in the right way. Now, let's see another example:

```
dim i,j,k as word
i=50000
j=60000
k=(i+j)/10 ' result will be 4446, which is incorrect. 32-bit ariphmetic was
not automatically used!
```

This example requires 32-bit calculations because i+j will produce a number which is outside the scope of 16-bit calculations. However, our compiler will not invoke 32-bit calculations automatically. Read on, this is not all...

Mixing in a 32-variable will cause the compiler to use 32-bit calculations

For the example above, turn i or j into a dword. Now, your calculations will be correct. Mixing in any 32-bit variable will aotumatically upgrade calculations to 32 bits!

```
dim i,k as word
dim d as dword
i=50000
d=60000
k=(i+d)/10 ' result will be 11000. This is correct!
```

String and non-string variables cannot be mixed!

Yes, sorry, but you cannot do the following (compiler will generate type mismatch 110 error):

```
'Wrong!!!
dim x as byte
x=5+"6"
```

In case you are wondering why x="6" would work but x=5+"6" doesn't: the latter is a mixed expression in which the compiler cannot decide what is implied: conversion of string to value and then addition, or conversion of value to string and then string concatenation!

Compile-time Calculations

Tibbo Basic always tries to pre-calculate everything that can be pre-calculated during compilation. For example, if you write the following code:

```
dim x,y as byte
y=5
x=5+10+y 'compiler will precalculate 5+10 and then the Virtual Machine will
only have to do 15+y
```

Pre-calculation reduces program size and speeds up execution.

When processing a string like "x=50000" compiler first determines the variable of what type would be necessary to hold the fixed value and chooses the smallest sufficient variable type. For example, for 50000 it is, obviously, **word**. Next, compiler applies the same rules of type conversion 50 as between two actual variables. Hence, in our example this will be like performing **byte**= **word**.

One additional detail. Large fixed values assigned to variables of real type must be written with fractional part, even if this fractional part is 0. Consider the following example:

```
dim r as real
r=12345678901 'try to compile this and you will get "constant too big"
error.
```

Compiler will notice that this constant does not fit even into **dword** and will generate an error. Now, try this:

```
dim r as real
r=12345678901.0 'that will work!
```

On seeing ".0" at the end of the value compiler will realize that the value must be treated as a **real** one (floating-point format) and process the value correctly.

So, why is it possible for compiler to automatically process values that fit into 8, 16, and 32 bits, but at the same time requires your conscious effort to specify that the value needs to be treated as a floating-point one?

This is because floating-point calculations are imprecise. The value "12345678901", when converted into a floating-point format, will not be exactly the same! The floating-point value will only *approximate* the value we intended to have!

For this reason we require you, the programmer, to make a conscious choice when specifying such values. By adding ".0" you acknowledge that you understand potential imprecision of the result.

Arrays4.2.4.5

An *array* is a single variable which contains several elements of the same type. Each element has a value and an index number, and may be accessed using this number.

An example of a simple array would be:

Index:	0	1	2	3	4
Value:	15	32	4	100	-30

The code to produce such an array in Tibbo Basic would look like this:

```
dim x(5) as char
x(0) = 15
x(1) = 32
x(2) = 4
```

```
x(3) = 100
x(4) = -30
```

The first index in an array is **0.** Thus, the array above contains 5 values, with indices from 0 to 4.

- This is different from some BASIC implementations that understand x(5) as "array x with the maximum element number of 5" (that is, with 6 elements). In Tibbo Basic declaring x(5) means "array of 5 elements, with indices from 0 to 4".

Variable Types For Arrays

In the example above, the array was assigned the type **char**. This means that each element within this array will be stored in a variable of type **char** 48. Starting from Tibbo Basic **V2.0** you can have arrays of variables of any type.

Accessing a Value Within an Array

To access a specific value within an array, include its index immediately after the name of the array, in parentheses. The index may also be expressed through a variable.

```
y = x(4) ' y would get a value of -30, according to the previous array
y = x(z) ' y would get the value of index z within array x.
```

As an example, you could iterate through an array with a loop (such as a For... Next Statement 90) and execute code on each element in the array, by using a variable to refer to the index of an element. Let's see how to calculate the sum of the first three elements in the array we previously defined:

```
dim x(5) as char
x(0) = 15
x(1) = 32
x(2) = 4
x(3) = 100
x(4) = -30

dim sum as integer
sum = 0

for i = 0 to 4 ' note that you do not necessarily have to iterate through
all elements in the array.
      sum = sum + x(i)
next i
' now, at the end of this loop, sum contains the sum for the first three
elements (51)
```

The TIDE and Tibbo Basic **V2.0** introduced correct handling of array indices. It is no longer possible for your program to point to an array element that does not exist. For example, if your array only has 5 elements and you try to access element number 5 the Virtual Machine will:

- Generate an OOR (Out Of Range) exception 29 and halt if your program is in the debug 27 mode. If you attempt to continue the Virtual Machine will access x(4) -- the element with maximum available index.
- When in the release 27 mode, the OOR exception will not be generated but "index limiting" will still occur.

Example:

```
dim x(5) as char
dim f as byte

f=5
x(f)=3 'index limiting will happen here (preceded by the OOR exception if
you are in the debug mode)
```

Compiler is smart enough to notice out-of-range situations even at compile time. For example, the following code will generate an error during compilation:

```
dim x(5) as char

x(5)=3 'compiler will determine that this operation will be eccessing an
array element which does not exist!
```

Multi-Dimensional Arrays

The array in the example above is called a *one-dimensional* array. This is because every element in the array has just a single index number. However, we could also have an array which looks like this:

Index:	0	1	2	3	4
0	15	32	4	100	-30
1	78	15	-3	0	55
2	32	48	97	5	22
3	13	18	9	87	54
4	32	35	79	124	3
5	7	-9	48	8	99

This is called *two-dimensional array*. Each element in the array is now identified by an index consisting of two numbers (two coordinates). For example, the element **2, 0** contains the value **32.** To create such an array, you would use the following code:

```
dim x(5,6) as char
x(0,0) = 15
x(0,1) = 32
x(0,2) = 4
```

```
......
x(5,2) = 48
x(5,3) = 8
x(5,4) = 99
```

Iterating through such an array would be done using a nested loop for each dimension in the array. The array above contains only two dimensions, so we would nest one loop within another. For an array containing six dimensions, we would have to use six such nested loops. See:

```
dim x(5,6) as char
x(0,0) = 15
x(0,1) = 32
x(0,2) = 4
......
x(5,2) = 48
x(5,3) = 8
x(5,4) = 99


dim i, j, sum as integer
sum = 0
for i = 0 to 5
      for j = 0 to 4
             sum = sum + x(i,j)
      next j
next i
' here, sum would be equal to the sum of the whole array. How much is that?
Try and see.
```

In Tibbo Basic, you may define up to 8 dimensions in an array.

Alternative way of defining arrays

We have already explained that the following string means "an array x containing 10 elements of type byte":

```
dim x(10) as byte
```

In Tibbo Basic, the same can be expressed in a different way:

```
dim x as byte(10) 'you can think of it as '10 times of byte type' :-)
```

Both ways of defining arrays are completely identical and you can even mix them together, as we can see on the following examples of 2-dimensional arrays:

```

```

```
dim i(20,10) as byte 'two-dimensional array, 20x10 elements
dim i2(20) as byte(10) 'same! that is, 20 groups of byte x 10 -- exactly same
meaning
dim i3 as byte(20,10) 'yet another way -- same result!
```

Now, Tibbo Basic strings can be defined with an optional parameter specifying maximum string length, for example:

```
dim s as byte(30) 'this string will have maximum length of 30 characters
(bytes)
```

So, how do we declare an array of string variables? Here are some examples:

```
dim s(20,10) as string(30) 'two-dimensional array of 30-byte strings, 20x10
elements
dim s2(20) as string(30)(10) 'same!
dim s2 as string(30)(20,10) 'same!
```

Arrays introduce slight overhead

Each array occupies more space than the sum total of space needed by all elements of an array. This is because each array also includes housekeeping data that, for instance, defines how many elements are there in an array, array of what type that is, etc.

Structures4.6

Beginning with **V2**, Tibbo Basic supports structures. Structure is a combinatorial user-defined data type that includes one or several *member* variables. Structures are declared using **type ... end type** 100 statements, as shown in the example below:

```
'this is a structure with three members
type my_struct
     x as byte
     y as Long
     s as string
end type
```

In the above example, we declared a structure my_struct that has three member variables: x, y, and s. This is just a declaration -- you still have to define a variable

with the new type you have created if you want to use the structure of this type in your program:

```
dim var1 as my_struct 'this is how you define a variable with type
'my_struct'
```

After that, you can address individual elements of the structure as follows:

```
var1.x=5
var1.y=12345678
var1.s="Test"
```

Structures you define may include members that are arrays or other structures. Structure variables like var1 above can be arrays as well, of course. In total, Tibbo Basic supports up to eight nesting levels (as in "array within a structure within a structure within and array" -- each structure or array is one level and up to 8 levels are possible). Here is a complex example:

```
type foo_struct 'structure with two members, and both are arrays
   x(10) as byte
   s(10) as string(4)
end type

type bar_struct 'structure with two members, one if which is another
structure
   foo as foo_struct 'so, this member is a structure
   w as word
end type

dim bar(20) as bar_struct 'we define an array of type 'bar_struct'

bar(1).foo.s(2)="test" 'oh-ho! we address element 2 of member s of member
foo of element 1 of bar!
```

Structures introduce slight overhead

Each structure occupies more space than the sum total of space needed by all of its members. This is because each structure also includes housekeeping data that, for instance, defines how many members are there, what type they have, etc.

Enumeration Types

At times, it may be useful for a programmer to define his own enumeration data types; for example, when working with the days of the week, it may be useful to refer to them by name in code, rather than by number:

```
dim i as integer
i = 2 ' i is an integer, and can only be assigned a numerical value. you
would have to remember that 2 is Monday.
```

```
dim d as dayofweek
d = monday ' now d is a user-defined type, dayofweek, and can be assigned a
value using the symbol Monday.
```

Enumeration definitions are made like this:

```
enum dayofweek
   sunday = 1,    ' if 1 is not specified, the default value associated with
the first constant is 0.
   monday,        ' by default, increments the previous value by 1. can also
be explicitly specified.
   tuesday,
   wednsday,
   thursday,
   friday,
   saturday,
   holiday = 99, ' as described above, values can be explicitly associated
with any constant in the list.
   holiday2      'this will have the value of 100
end enum
```

An enumeration type would then be used within the code as shown above (d = Monday). Note that even though Monday is an identifier 137 (i.e., an actual word, and not just some number) it does not have to be surrounded by quote marks (because it's not a string).

The value associated with each identifier within the enumeration type doesn't necessarily have to be unique; however, when you associate the same value with several constants, the distinction between these constants will be lost on compile time.

```
enum dayofweek
      Sunday = 1,
      Monday,
      Tuesday,
      Wednsday,
      Thursday,
      Friday,
      Saturday,
      holiday = 99
      bestdayofweek = 7 ' bestdayofweek and Saturday are actually the same!
      fridaythethirteenth = -666 ' a negative constant.
end enum
```

Note that above, `saturday` was implicitly (automatically) associated with the value 7, and `bestdayofweek` was explicitly associated with the same value. Now, on compile-time, they would both be considered to be just the same.

Type Mapping for Enum Types

When the project is being compiled, all enumeration types are substituted with plain numerical values. the platform on which the code is running doesn't have to know anything about Saturday or about the best day of the week; for the platform, the number 7 is informative enough.

Hence, enumeration types are converted to various built-in numerical variable types. The actual numerical type used to store the enumeration type depends on the values associated with the constants within this enumeration type:

Values associated with constants do not exceed range	Actual variable type used to store enum type
-128 to 127	**char**
0 to 255	**byte**
-32768 to 32767	**short**
0 to 65535	**word**
-2147483648 to 2147483647	**long**
0 to 4294967295	**dword**

Notice, that enumeration types cannot be converted into real values, so you cannot use fractional numbers in you enums.

Some examples:

```
enum tiny
      tinyfoo,
      tinybar
end enum ' this enum will be associated with a char type

enum medium
      mediumfoo = 254
      mediumbar
end enum ' this enum will be associated with a byte type

enum impossible
      baddy = -1
      cruise = 4294967295
end enum ' this enum will raise a compile error, as no single variable type
can hold its values

enum nofractions
      thisisok = 4294967295
      thisisnot = 125.25
end enum ' compiler won't accept this! Integer values only, please!
```

Enumeration types are helpful when debugging your code

For each enumeration type variable, the watch facility 33 of the TIDE shows this variable's current numerical value with a correct identifier associated with this value. This usually proves to be very useful during debugging! After all, seeing "dayofweek= 2- Monday" is much less cryptic than just "dayofweek= 2"!

Understanding the Scope of Variables

A *scope* is a section of code, from which you can 'see' a variable (i.e, assign it a value, or read its value). For example:

```
sub foobar(x as byte)
      dim i, y as byte
      y = x * 30
      for i = 1 to y
            dim r as short
            r = x + 5
      next i
      r = x + 5 ' this would produce a compiler error
end sub
```

So, in the example above, **x** and **y** could be seen from anywhere within the **sub** procedure called **foobar**. However, **r** could be seen only from within the **for... next** statement. Thus, trying to assign **r** a value from outside the **for... next** statement would result in a compiler error, because it actually doesn't exist outside of that loop.

One identifier can refer to several different variables, depending on the scope in this identifier is used:

```
dim x as byte ' this creates the variable x in the global scope.

sub foobar(x as byte) ' here we create x once more, in the scope of the
local sub (x as an argument).
      dim f, y as byte
      x = 5 ' right now, only the locally-created x (foobar agrument) equls
5; global x remains unchanged.
      y = x * 30
      for f = 1 to y
            dim x as byte
            x = 30 ' the argument x outside the for... next statement still
equals 5. only this local x equals 30.
      next f
end sub
```

Tibbo Basic supports several scopes:

Global Scope

Every compilation unit has, in itself, one global scope. Variables declared in this scope are accessible from within any **sub** or **function** in this compilation unit.

```
dim s as string

sub foo
      s = "foo" ' assigning a value to the global string variable s.
end sub

sub bar
      dim i as short
      i = 0
      s = ""  ' initialize s, in case it contains anything already (such as
'foo').
      for i = 1 to 5
```

```
            s = s + "bar" ' note
     next i
end sub ' at this point, s contains 'barbarbarbarbar'.
```

Local Scope

This is the scope which is between the beginning and the end of each of the following statements:

Beginning	End	Notes
sub	**end sub**	Cannot be nested.
function	**end function**	Cannot be nested.
for	**next**	
while	**wend**	
if... then... else	**end if**	No **exit** statement for **if... then... else**.
do	**loop**	

Variables declared in this scope are accessible from within the construct in which they were declared. Local scopes may be nested, for example, **for...next** scope inside **sub...end** sub scope.

A locally defined variable with the same name as a global variable takes precedence in its context over any variable with the same name which is defined in a 'wider' scope.

Local variable names also take precedence over procedure names. For example:

```
sub prc1(x as byte)
   'some code here
end sub

sub prc2
   dim prc1 as byte 'define a local variable with the same name as one
procedure we have
   prc1=0 'this will generate no error -- in the current scope prc1 is a
variable
   prc1(2) 'here, we try to invoke sub prc1 and this will cause a compiler
error.
end sub
```

HTML Scope

This section applies only to platforms which include an HTTP server.

This is a special scope, implemented in Tibbo Basic. HTML files included within a project may contain embedded Tibbo Basic code. This code is executed when the HTTP server processes an HTTP GET (or POST) request. Statements within an HTML file are considered to be within one scope -- similarly to a **function** or **sub** scope, with the exceptions that **include** 95 and **declare** 84 statements are allowed.

```

```

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD W3 HTML//EN">
<HTML>
<BODY>

      BEGINNING OF OUTPUT<br>

<?
include "global.tbh"
declare i as integer ' i is defined somewhere else
for i = 1 to 10
?>

      <i> Foo </i>

<?
next i
?>

      <br>END OF OUTPUT<br>

</BODY>
</HTML>
```

Designing dynamic HTML pages always presents you with a choice: what to do- include the BASIC code right into the HTML file or create a set of subs and functions and just call them from the HTML file? In the first place you put a lot of BASIC code into the HTML file itself, in the second case you just call subs and functions from the HTML file. So, which way is better?

Generally, we recommend to use the second way. First of all, this style of programming is cleaner- a mixture of BASIC code and static HTML text usually looks messy. Second, the second method *consumes less variable memory*.

Although the HTML scope is similar to a local scope of a function or a sub, its variables get exclusive memory allocation as if they were global. When you avoid writing a lot of BASIC code in the HTML file itself you usually avoid having to create a lot of variables in the HTML scope and this saves you memory!

Declaring Variables

Usually, a variable is first defined by the **dim** 86 s 86 tatement 86 and then used in code; however, at times, a single global variable must be accessible from several compilation units. In such cases, you must use the **public** 104 modifier when defining this variable, and use the **declare** 84 statement 84 in each compilation unit from which this variable needs to be accessed.

For example, let us say this is the file **foo.tbs**:

```
public dim i as integer ' the integer i is defined in this file, and made
into a public variable. It can now be used from other compilation units.
```

And this is the file **bar.tbs**:

```
```

```
i = i + 5 ' this would cause a compiler error. What is i?

' the correct way:
declare i as integer ' lets the compiler know that i is defined elsewhere.
i = i + 5
```

Also, if for some reason you would attempt to use a variable in a single compilation unit before defining it using the **dim** statement, you will have to use a **declare** statement before using it to let the compiler know that it exists. For example:

```
declare i as integer
i = 7 ' i has not been defined yet, but we let the compiler know that it is
defined elsewhere.

...

dim i as integer ' we now define i. Note that here it doesn't have to be
public, because it is used in the same compilation unit.

' this example is rather pointless, but just illustrates this single
principle.
```

Constants 4.10

Constants are used to represent values which do not change throughout the program; these values may be strings or numbers. They may either be stated explicitly, or be derived as the result of an expression.

Some examples:

```
const universal_answer = 42
const copyright = "(c) 2005 Widget Systems Inc." ' this is a string constant
const escape_char = `@`  ' this constant will contain a numerical value --
the ASCII code for the char @.

const hexi = &hFB ' would create a constant with a value of 251 (&hFB in
hex)
const bini = &b00110101 ' would create a constant with a value of 53
(&b00110101 in binary)

const width = 10
const height = 15
const area = width * height ' constants may contain expressions which
include other constants

dim x as byte
const foo = x + 10 ' this will produce a compiler error. Constant
expressions may contain only constants.
```

Constants can be useful when you have some values which are used throughout the code; with constants, you can define them just once and then refer to them by their meaningful name. This has the added benefit of allowing you to easily change the value for the constant any time during the development process -- you will just have to change the definition of the constant, which is a single line of code.

When defining a list of related constants, it is often convenient to use the Enum Statement 88 and create one data type which contains this list of constants. See also User-Defined Types 59 above.

When defining a constant within a scope 61, this constant is visible only from within this scope. It is a good idea to define all constants within header files, and include 95 these files into each compilation unit.

String constants

String constants can include escape sequences to define unprintable characters. This functionality is borrowed from C. Adding unprintable characters to the string has always been rather inconvenient in BASIC language. The only way to do so was like this:

```
s = "abc"+chr(10)+chr(13) 'add LF/CR in the end
```

In Tibbo Basic you can achieve the same by using escape sequences -- C style:

```
s = "abc\n\f" ''\n' means LF, '\f' -- CR
const STR1 = "abc\x10\x13" 'same result can be achieved using HEX codes of
the characters
```

The following standard escape sequences are recognized:

- "\0" for ASCII code 0
- "\a" for ASCII code 7 (&h7)
- "\b" for ASCII code 8 (&h8, BS character)
- "\t" for ASCII code 9 (&h9)
- "\n" for ASCII code 10 (&hA, LF character)
- "\v" for ASCII code 11 (&hB)
- "\f" for ASCII code 12 (&hC)
- "\r" for ASCII code 13 (&hD, CR character)
- "\e" for ASCII code 27 (&h1B, ESC character)

Any ASCII character, printable or unprintable, can be defined using its HEX code. The following format should be used: "\x00" where "00" is the HEX code of the character. Notice, that two digits should be present on the code, for example: "\x0A" -- leading zero must not be omitted.

\

Introduction to Procedures

A *procedure* is a named piece of code, which performs a designated task, and can be called (used) by other parts of the program. In Tibbo Basic, there are two types of procedures:

Function Procedures

A function is defined using the Function Statement 91. Functions can optionally have one or several arguments. Functions always return a single value. They can, however, change the value of the arguments passed to them using **ByRef** 68 and thus indirectly return more than one value. This would be an example of a function:

```
function multiply(a as integer, b as integer) as integer
      multiply = a * b
end function
```

Note how the function above returns a value: via a local variable with the same name as the function itself. Such a variable is automatically created by the compiler for each function.

Sub Procedures

Sub is short for *subroutine*; just like a function, a sub procedure can optionally accept one or more arguments. However, unlike functions, sub procedures do not return a value. It is defined using the Sub Statement 99. This would be an example of a sub:

```
dim a(10) as byte ' a is a global variable -- outside the scope of the
function.

sub init_array
      dim i as integer
      for i = 0 to 9
            a(i) = 0   ' the global variable gets changed.
      next i
end sub
```

Subs change the value of the arguments passed to them using **ByRef** 68 and thus indirectly return a value, or even several values. Of course, they may also change the value of global variables.

Event handlers are like subs

Event handlers defined in the platform work exactly like sub procedures. Event handler subs can accept arguments. Event handlers can never be function procedures as each function has to return a value and the event handler has nobody to return this value to.

Declaring Procedures

Usually, a procedure is first defined by the **function** 91 or **sub** 99 statements and then used in code; however, at times, functions can reside in a different compilation unit. In such a case, you must use the **public** 104 modifier when defining this function, and use the **declare** 84 statement 84 to let the compiler know that the function exists.

For example, let us say this is the file **utility_functions.tbs**:

```
public function multiply(a as integer, b as integer) as integer ' the value
returns by this function is an integer
      multiply = a * b
end function
```

And this is the file **program.tbs**:

```
declare function multiply(a as integer, b as integer) as integer ' declaring
just the name isn't enough. Include also the arguments and the types.
dim i as integer
i = multiply(3, 7)
```

Declare statements are usually used within header files which are then included into compilation units. Also, if for some reason you would attempt to use a procedure in a single compilation unit before defining it, you will have to use a **declare** statement to let the compiler know that it exists. For example:

```
declare function multiply(a as integer, b as integer) as integer
dim i as integer
i = multiply(3, 7)

...

function multiply(a as integer, b as integer) as integer ' now this function
doesn't have to be public.
      multiply = a * b
end function
```

Event handler subs require no declaration as they are already declared in your device's platform.

No Recursion

One thing you have to know is that procedures cannot call themselves. Also, two procedures cannot call each other. This is due to TiOS not using dynamic memory allocation. Such allocation would create a serious overhead for the system, and would drastically slow everything down -- not just recursive procedures. For more information, see Memory Allocation for Procedures 70.

Passing Arguments to Procedures

When calling subroutines or functions, it is often necessary to pass a certain value for processing within the procedure. An example of this would be a function which calculates the sum of two values; naturally, such a function would need to get two arguments -- the values which are to be added up.

There are two different ways to pass arguments to such a procedure:

The Default: Passing By Value

When passing an argument to a procedure *by value*, this argument is *copied* to a location in memory which was reserved for the local variables of this function. Processing is then done on this local copy -- the original remains untouched. For

example:

```
sub foo(x as byte)
...
x = 1
...
end sub

sub bar
      dim y as byte
      y = 2
      foo(y)
      ' at this point in code, y is still 2.
end sub
```

This way of passing variables is the default used in Tibbo Basic.

Passing By Reference

In certain cases, copying is not the preferred solution; for example, when a procedure has to modify several arguments passed to it and these later have to be accessible. Another example would be when processing large strings -- copying them would cause significant overhead.

In such cases, arguments are passed *by reference*. When passing by reference, the actual values are not copied. Instead, the procedure receives a reference to the location of the original values in memory -- hence, the name. For example:

```
sub foo(byref x as byte)
...
x = 1
...
end sub

sub bar
      dim y as byte
      y = 2
      foo(y)
      ' at this point in code, y is 1!
end sub
```

When passing arguments by reference, the code within the procedure will access these arguments using indirect addressing. This may cause possible overhead. The only case where it does not cause overhead (relative to passing by value) is when working with large strings; in this case, passing them by reference saves the need to copy the whole string to another location in memory.

Here is our advice: when dealing with strings it is usually better (in terms of performance) to pass them by reference. For all other types, passing by value yields better performance.

Strict byref argument match is now required!

Beginning with Tibbo Basic release **2.0**, strict match is required between the type of byref argument and the type of variable being passed. For example, trying to pass a string for a byte will now cause a compiler error:

```
sub bar(byref x as byte)
   ...
end sub

sub foo
   bar("123") 'attempt to pass a string will generate a compiler error
   bar(val("123")) 'this will work!
end sub
```

In the above example, sub bar takes a byte argument and we are trying to pass a string. Wrong! Compiler understands that byref arguments can be manipulated by the procedure that takes them. Therefore, it is important that actual variables being passed match the type of argument that procedure expects. Automatic type conversion 50 won't apply here.

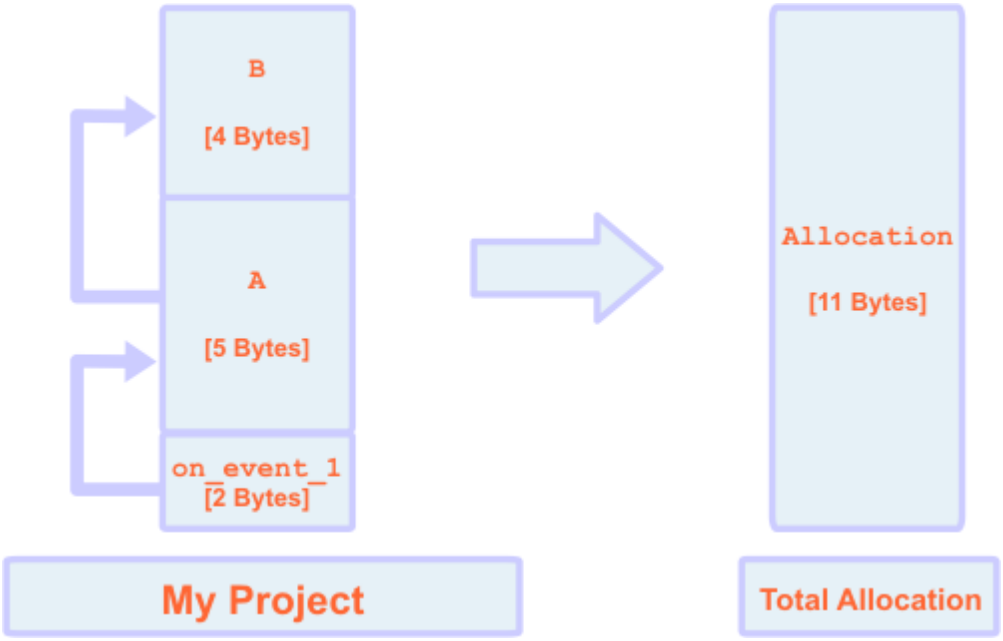
Memory Allocation for Procedures

Variable memory (RAM) allocation in TiOS is not dynamic 48. Memory is allocated for variables at compile-time.

The compiler builds a "tree" reflecting procedure calls within your project. When two different procedures never call each other, it is safe to allocate the same memory space for the variables of each of them. They will never get mixed.

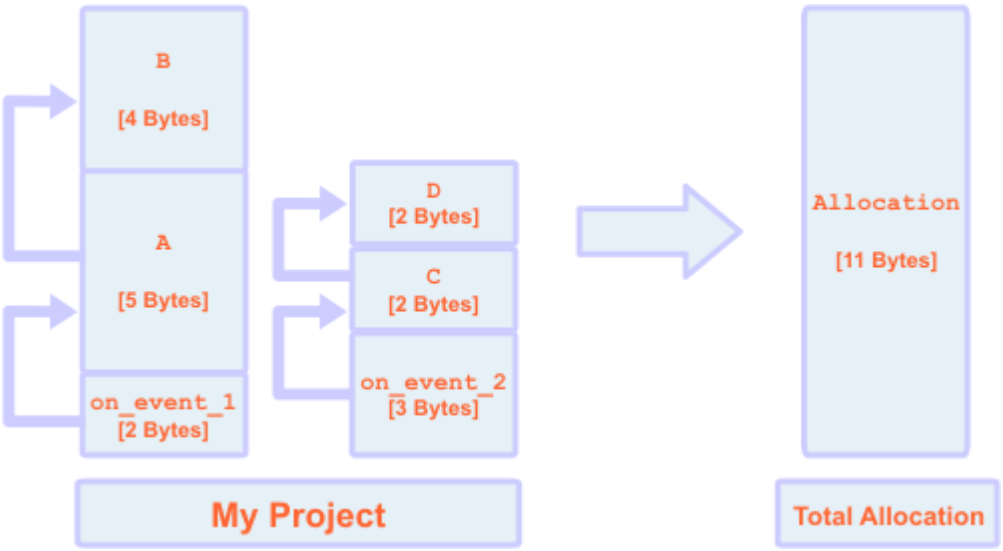
Let us say we have two event handlers in our project: on_event_1, which needs 7 bytes of memory, on_event_2, which needs 5 bytes of memory. They do not call each other. In this case, the total memory required for our project will be 7 bytes -- they will share the same memory space because only one will be executing at any given time.

However, sometimes procedures call other procedures. This affects memory allocation.



As seen above, the event handler on_event_1 calls procedure A, which in turn calls procedure B. The memory required for each procedure is listed in brackets. Since on_event_1, procedure A and procedure B call each other, they may not share the same memory space. If procedure A keeps a variable in memory, then obviously procedure B cannot use the same space in memory to store its own variables, because procedure A may need its variable once control returns to it after procedure B has completed. Thus, the total memory required for this tree is 11 bytes.

Now, let us say we also have on_event_2 in our project, which calls procedure C, which in turn calls procedure D. This is a completely separate chain:



As can be seen, this chain takes up 7 bytes of memory. However, this memory can be the same memory used for the on_event_1 chain, because these two chains will never execute at the same time. Thus, the total memory required for our project remains at 11 bytes.

A typical project usually includes a number of global variables 61. Naturally, these variables are allocated their exclusive space that is not shared with local variables of procedures. Variables of HTML scope, which are local by nature, are allocated exclusive memory space as if they were global (this is an unfortunate byproduct of

the way compiler handles HTML pages). Hence, it is more economical to implement necessary functionality in procedures invoked from HTML pages rather than include BASIC code directly into the body of HTML files.

Introduction to Control Structures

Control Structures are used to choose what parts of your program to execute, when to execute them, and whether to repeat certain blocks of code or not (and for how many iterations).

The two main types of control structures are decision structures 72 and loop structures 72.

Decision Structures

Decision Structures are used to conditionally execute code, according to the existence or absence of certain conditions.

Common uses for decision structures are to verify the validity of arguments, to handle errors in execution, to branch to different sections of code, etc.

An example of a simple decision structure would be:

```
dim x, y as byte
dim s as string

if x < y then
      s = "x is less than y"
else
      s = "x is greater than, or equal to, y"
end if
```

The following decision structures are implemented in Tibbo Basic:

- If.... Then... Else Statement 94
- Select-Case Statement 97

Loop Structures

Loop structures are used to iterate through a certain piece of code more than once. This is useful in many scenarios, such as processing arrays, processing request queues, performing string operations (such as parsing), etc.

An example of a simple loop structure would be:

```
dim f, i as integer

f = 1

for i = 1 to 6
      f = f * i
next i
' f is now equal to 1*2*3*4*5*6 (720).
```

The following loop structures are implemented in Tibbo Basic:

- Do... Loop Statement 87
- For... Next Statement 90
- While-Wend Statement 101

Doevents 6.3

Although under Tibbo Basic, event-driven programming is the norm, there may be special cases in which you just have to linger an overly long time in one event handler. This will block execution of other events. They will just keep accumulating in the queue (see System Components 7).

To resolve this, a doevents 87 statement has been provided. When this statement is invoked within a procedure, the execution of this procedure is interrupted. The VM then handles events which were present in the queue as of the moment of doevents invocation. Once these events are handled, control is returned to the procedure which invoked doevents.

If new events are added to the queue while doevents is executing, they will not be processed on the same doevents 'round'. Doevents only processes those events present in the queue at the moment it was invoked.

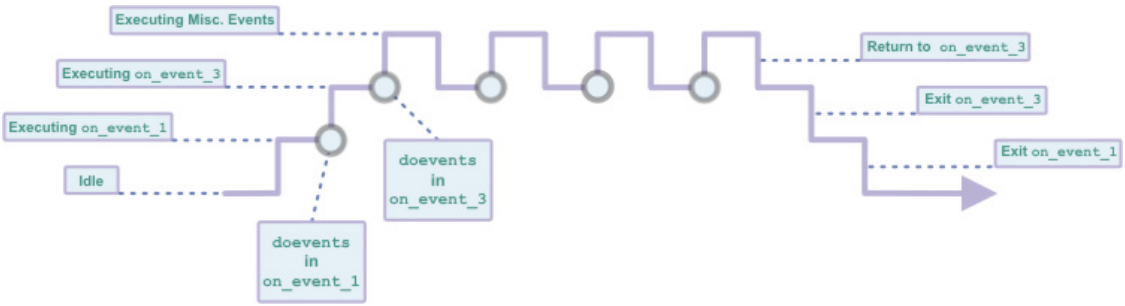
To summarize, doevents provides a way for a procedure to let other events execute while the procedure is doing something lengthy.

```
'calculate sum of all array elements -- this will surely take time!
dim sum, f as word
sum = 0
for f = 0 to 49999
      sum = sum + arr(f)
      doevents 'don't want to stall other events so allow their execution
while we are crunching numbers
next f
```

Multiple Doevents Calls

Let us say we are in event handler on_event_1. This event handler executes a doevents call. The VM begins processing other events in the queue, and starts executing an event handler on_event_3, which also contains a doevents statement.

In this case, a new doevents round will begin. Only when it completes, control will be returned to event handler on_event_3, which will complete, and then return control to the previous doevents (the one from event handler on_event_1).

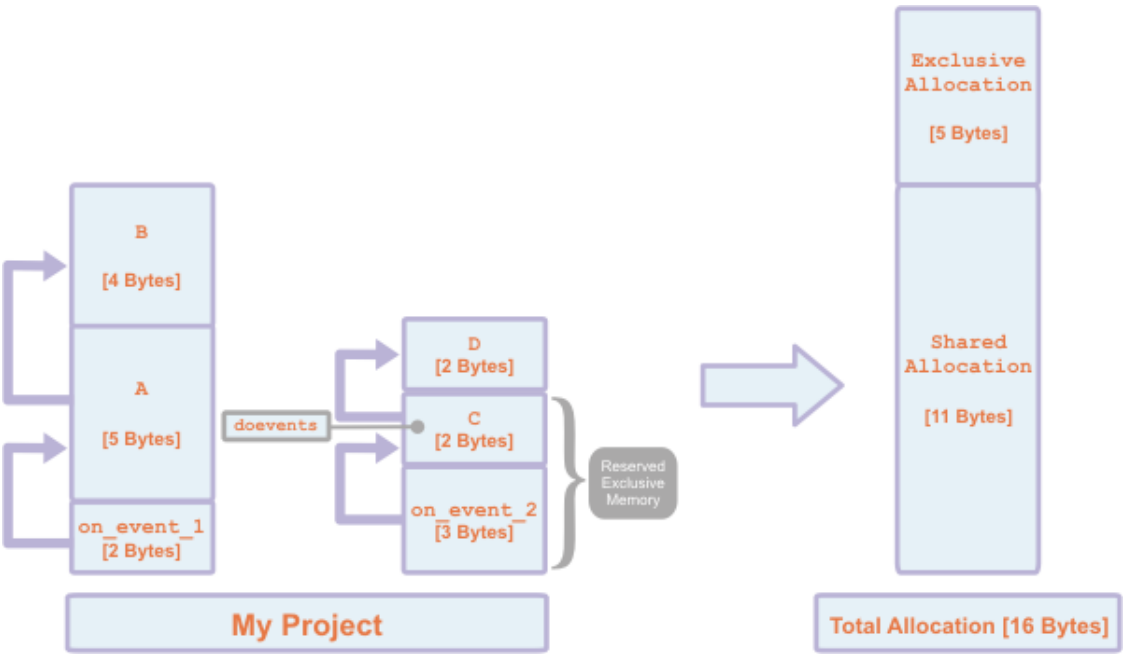


The point here is that control will not be returned to on_event_1 until on_event_3 fully completes execution, since on_event_3 contains a doevents statement in itself.

Memory Allocation When Using Doevents

When a procedure utilizes doevents, it means its execution gets interrupted, while other procedures get control. We have no way to know which other procedures will get control, as this depends on the events which will wait in the queue.

As a result, a procedure which utilizes doevents, and any procedures calling it (directly or through other procedures) cannot share any memory space with any other procedure in the project. This would lead to variable corruption -- procedures newly executed will use the shared memory and corrupt variables which are still needed for the procedures previously interrupted by doevents.



Above we have the same two chains of procedures which appear under Memory Allocation for Procedures 70, with one noticeable difference: Procedure C includes a doevents statement. The on_event_1 chain takes up 11 bytes. But on_event_2 and procedure C (which together take up 5 bytes) cannot share the same space with the on_event_1 chain, because when the doevents statement is invoked, the state of variables for on_event_2 and procedure C must be preserved. So these two get their own exclusive memory space.

Procedure D, which is also a part of the on_event_2 chain, does not get its own exclusive memory space. This is because it comes later on the chain than the procedure which contains the doevents statement. There will never be a situation

where the variables of procedure D must be preserved while other chains are executing.

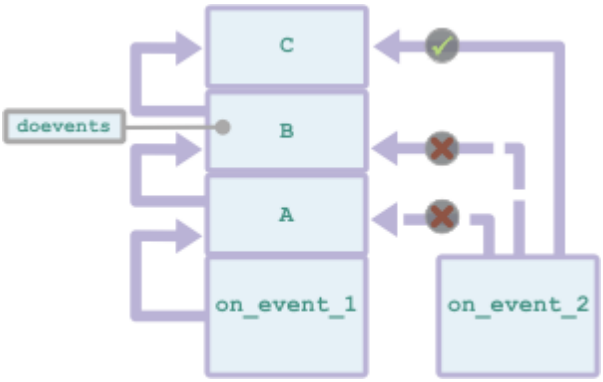
Thus, the total memory requirements of the project depicted above would be 16 bytes -- 11 shared bytes plus 5 exclusive bytes. This is more than would have been required had we not used doevents.

Sharing Procedures Which Utilize Doevents

Procedures which contain doevents statements, as well as all procedures which call them (directly or through other procedures) cannot be shared between chains.

Let us say event handler on_event_1 calls procedure A. Procedure A calls procedure B. Procedure B contains a doevents statement, and also calls procedure C.

Next, we have event handler on_event_2. It cannot call procedure A or B, because procedure B contains a doevents statement (and A calls B). If it *could* call procedure A, we might get a situation whereby event handler on_event_1 fires, calls procedure A. In it, procedure B is called, and doevents is executed. During doevents, an event handler on_event_2 fires, and calls procedure A *again* -- while the previous instance of A still holds variables in memory, waiting for control to return to it. This would corrupt the values of variables used by the first A (if you try to do something like this, the compiler will raise an error).



However, note that in the example above we also have procedure C (which is called by procedure B). This procedure can be shared by everyone -- because it is later on the chain than the procedure which contains the doevents statement.

Doevents for Events of The Same Type

For some events, only one instance of the event may be present in the queue at any given time. The next event of the same kind may only be generated after the current one has completed processing. For other events, multiple instances in the queue are allowed.

Let us say that for event_1, multiple instances are allowed, and that this event's handler contains a doevents statement. When this statement executes, it may happen that another instance of event_1 will be found on the queue, waiting to be processed. If this happens, this new instance will just be skipped -- execution will move on to the next event on the queue. Otherwise, we would once again get recursion (execution of an event handler while a previous instance of this event handler is already executing), which is not allowed under Tibbo Basic.

Using Preprocessor

Tibbo Basic compiler includes a preprocessor that understands several directives.

#define and #undef

The **#define** directive assigns a replacement token for an identifier. Before compilation, each occurrence of this identifier will be replaced with the token, for example:

```
#define ABC x(5) 'now ABC will imply 'x(5)'
ABC=20 'now it is the same as writing x(5)=20
```

The **#undef** directive "destroys" the definition made earlier with #define :

```
#define ABC x(5) 'define
...
#undef ABC 'destroy
...
ABC=20 'you will get a compilation error on this line (compiler will try to
process this as "=20")
```

#if - #else - #elif -- #endif

These directives are used to conditionally include certain portions of the source code into the compilation process (or exclude from it). Here is an example:

```
#define OPTION 0 'set to 0, 1, or 2 to select different blocks of code for
compilation
...
#If OPTION=0
   s="ABC" 'will be compiled when OPTION=0
#elif OPTION=1
   s="DEF" 'will be compiled when OPTION=1
#Else
   s="123" 'will be compiled when OPTION=2, 3, etc.
#endif
```

You can improve on this example and add meaning to 0, 1, and 2:

```
#define RED 0
#define GREEN 1
#define BLUE 2
...
#define OPTION BLUE
...
#If OPTION=RED
   s="ABC" 'will be compiled when OPTION=RED (0)
```

```
#elif OPTION=GREEN
   s="DEF" 'will be compiled when OPTION=GREEN (1)
#Else
   s="123" 'will be compiled when neither RED, nor GREEN
#endif
```

You can also write like this:

```
#If OPTION
   x=33 'will be compiled in if OPTION evaluates to any value except 0. Will
not be compiled in if OPTION evaluates to 0.
#endif
```

Preprocessor directives are not "strict". You don't have to define something before using it. During #if evaluation, all undefined identifiers will be replaced with 0:

```
#If WEIRDNESS 'undefined identifier
   x=33 'will not be compiled in
#endif
```

Do not confuse compile-time definitions such as "#define OPTION 2" and actual application code like "const COLOR=2" or "x=3". They are from two different worlds and you can't use the latter as part of #if directives. For example, the following will not work:

```
#define OPTION 0 'preprocessor directive
Const COLOR=2 'constant used by your application

#If OPTION=COLOR 'to a confused programmer, this looks like 0=2, but COLOR
is not #defined, hence, it will be evaluated to 0
   'hence, this code will be compiled in!
#endif
```

The #if directive also understands expressions, for example:

```
#define RED 0
#define GREEN 1
#define BLUE 2
...
#define OPTION 2
...
#If OPTION=GREEN+1
   'will be compiled in, because GREEN=1, hence the entire expression
evaluates to 2, and OPTION=2
#endif
```

#ifdef - #else - #endif

#ifdef and #ifndef# are like #if, but instead of evaluating an expression they simply checks if specified definition exists:

```
#ifdef OPTION
   s="X" 'will be compiled if OPTION is defined
#Else
   s="1" 'will be compiled if OPTION is not defined
#endif
```

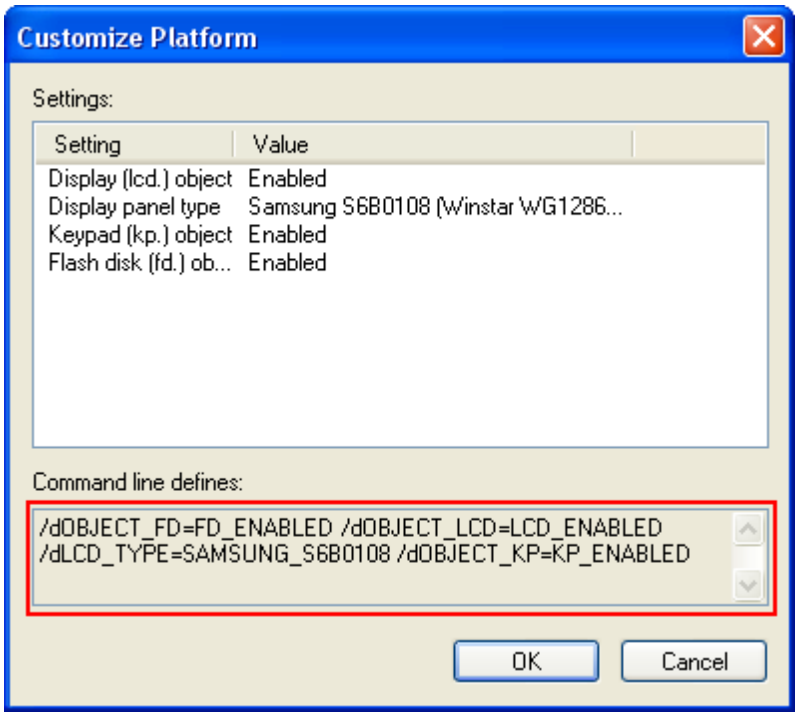
#ifndef is like #ifdef, but in reverse:

```
#ifndef OPTION
   s="X" 'will be compiled if OPTION is not defined
#Else
   s="1" 'will be compiled if OPTION is defined
#endif
```

Scope of Preprocessor Directives

Each preprocessor directive applies only to its own compilation unit 43, not the entire project. So, a #define directive in *main.h* will not be "visible" in *main.tbs* unless the latter includes the *main.h*.

The only exception to the above are platform defines. These are globally visible throughout your entire project. Platform defines determine options such as the presence or absence of a display, display type, etc. These options are selected through the Customize Platform dialog, accessible through the Project Settings 131 dialog.



Working with HTML

One of the strengths of Tibbo programmable devices is that they feature a built-in webserver (of course, this is only true for devices that have a network interface and support TCP communications). You can use this webserver as an engine for server-side scripting; simply put, you can output dynamic HTML content by including Tibbo Basic instructions within HTML pages.

Here is a simple example:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD W3 HTML//EN">
<HTML>
<BODY>

      BEGINNING OF OUTPUT<br>

<?
dim i as integer
for i = 1 to 10
?>

      <i> Foo </i> <br>

<?
next i
?>

      <br>END OF OUTPUT<br>

</BODY>
</HTML>
```

In effect, this file would cause the following to appear in the browser window of the user accessing this page:

```
BEGINNING OF OUTPUT
foo
foo
foo
foo
foo
foo
foo
foo
foo
foo
END OF OUTPUT
```

When to Use HTML Pages In Your Project

For an embedded device, built-in webserver can provide a convenient interface to this device; in fact, it is one of the best ways to allow your users to access your device remotely. They would just have to enter an address in a web browser, and voila, up comes your interface.

HTML support, as implemented in the Tibbo Basic, allows you complete control over page structure. You can use client-side technologies such as JavaScript, CSS etc., while still being able to dynamically generate HTML content within Tibbo device.

Further information about creating HTML files with dynamic content can be found in the next topic 80 as well as Using HTTP 461 topic (part of the sock 421 object documentation).

Embedding Code Within an HTML File

As covered in Understanding the Scope of Variables 61, each HTML file has a special scope, and all code within the file resides within this scope.

To begin a block of Tibbo Basic code within an HTML file, you must use an escape sequence -- **<?** . To close the section of code, use the reverse escape sequence -- **?>** .

When the embedded HTTP server receives a GET (or POST) request, it begins to output the requested HTML file. It simply reads the HTML file from top to bottom, and transmits its contents with no alteration. However, the moment is encounters a block of Tibbo Basic code, it begins executing it.

Tibbo Basic code inside HTML files does not differ from the code in "basic" files, but it may not contain procedures. This is because the Tibbo Basic code in the HTML file is considered to constitute a procedure in itself. Notice, that all code in one HTML file is considered to be a single procedure, even if there are several fragments of code in this HTML file. Consider this example:

```
<!DOCTYPE HTML public "-//W3C//DTD W3 HTML//EN">
<HTML>
<BODY>

   BEGINNING OF OUTPUT<br>

<?
'<--------------- BASIC procedure starts here
dim i as integer
for i = 1 to 10
?>

   <i> Foo </i> <br>

<?
next i
'<--------------- procedure ends here
?>

   <br>end OF OUTPUT<br>

</BODY>
</HTML>
```

There two code fragments, yet they both form one procedure. For example, variable *i* declared in the first fragment is still visible in the second fragment.

The fact that entire code within each HTML file is considered to be a part of a single procedure has implications in the way events are handled (reminder: there is a single queue 71 for all events). The next event waiting in the event queue won't be executed until the end of the HTML procedure is reached. Just because the HTML procedure consists of two or more fragments does not mean that other events will somehow be able to get executed while the HTTP server outputs the static data between those fragments (" Foo
" in our example). Use doevents 73 if you want other event handlers to squeeze in!

Tibbo Basic code in the code fragments may include decision structures 72 or loop structures 72 that may cause various segments of HTML code to be output more than once, to be skipped altogether, or to be output only when certain conditions are true or false. In the above example the line " Foo
" will be output 10 times because this line resides between two code fragments that implement a cycle!

The same result could be achieved in a different manner:

```
<!DOCTYPE HTML public "-//W3C//DTD W3 HTML//EN">
<HTML>
<BODY>

   BEGINNING OF OUTPUT<br>

<?
dim i as integer
dim s as string

s="<i> Foo </i> <br>"
```

```
for i = 1 to 10
   while sock.txfree<len(s) 'these free lines can be omitted for simple
      doevents              'tests but are actually essential for
   wend                     'reliable output of large data chunks

   sock.setdata(s)          'this prepares data for output
   sock.send                'and this commits it for sending (see sock
object)
next i
?>

   <br>end OF OUTPUT<br>

</BODY>
</HTML>
```

Here we have a single code block and "printing" the same line several times is achieved by using sock.setdata 500 and sock.send 500 methods of the sock 421 object.

So, which of the two examples shows a better way of coding? Actually, both ways are correct and equally efficient. The first way will have an advantage in case you have large static blocks that may be harder to deal with when you need to print them using sock.setdata method.

Further information about creating HTML files with dynamic content can be found in Using HTTP 461 (part of the sock 421 object documentation).

Understanding Platforms

As an embedded language, Tibbo Basic may find itself within many various hardware devices; each such device may have different capabilities in terms of storage, physical interfaces, processing power, and other such parameters.

Thus, Tibbo Basic is not a one-size-fits-all affair; it is customized specifically for every type of physical device on which TiOS runs. A function which initializes a WiFi interface would make very little sense on a device which does not support WiFi. The same would go for a function which clears the screen -- what if you have no screen? This holds true even for string functions -- some platforms are so tiny, they do not even need to support string processing!

Because of this, the 'core' of the Tibbo Basic language is actually very minimalistic -- we call it "pure" -- it contains only the statements listed under Statements 83 below. Any other functionality is implemented specifically for each platform, and is documented in detail for your platform under Platforms 138.

Objects, Events and Platform Functions

Each platform 138 provides the following types of custom language constructs:

Objects

These provide a way to access the various facilities and subsystems of the host platform. For example, in a platform which supports networking, we would have a *socket* object that handles TCP/IP communication.

Each object has *properties, methods* and *events:*

A *property* of an object allows you to read or change an internal variable for this object. For example, a *serial port* object may have a *baudrate* property. Change

the value of this property, and the actual physical baudrate changes. There are also read-only properties which only provide information.

```
ser.baudrate = 3 ' set the baudrate
x = ser.numofports ' find out how many serial ports the device has.
```

A *method* of an object is a way to make the object perform a certain action. It is basically a procedure. It can optionally take arguments or return values. Our *ser* object could have *getdata* and *setdata* methods, for instance.

```
s = ser.getdata(50) ' gets up to 50 bytes of data into variable s.
ser.setdata(s) ' prepares up to 50 bytes of data for sending.
ser.send ' no arguments, returns nothing. Sends data.
```

An *event* 8 of an object is something that 'happens' to this object in reality. When TiOS registers an event, an *event handler* for it is automatically called, if it exists in your source code. E*vent handlers* are simply subs with no arguments.

```
sub on ser_data_arrival
' ... do something! ... <-- will be called when data arrives into the serial
port.
end sub
```

Platform Functions

Many functions commonly available in other BASIC versions are implemented in Tibbo Basic on the platform level and not on the "pure" language level; these include also seemingly universal functions, such as string processing, or various date and time functions. This is done so because not every platform would actually need these functions, universal as they may seem. Some platforms may have very limited resources, and not every platform needs to know what the time is, or how to parse strings.

Language Reference

The text below provides a complete rundown of all built-in language statements 83, keywords 102 and operators 105. If you can't find something here, that means it is platform-specific, and you would find it in your platform 138 documentation.

The examples provided herein may not work on your platform -- they are given for reference only.

Statements

Statements are used for directing the machine to perform a specific operation. A *statement* is the smallest possible unit of code which would compile by itself. You could say they are the programming-language equivalent of a sentence in human speech.

Statements are built into Tibbo Basic itself, and are not platform-specific.

Const Statement

Function: Declares constants for use in place of literal values.

Syntax: **const** name = value

Scope: Global, HTML and local

See Also: Enum Statement 88

Part	Description
name	The name of the constant, later used to refer to this constant in code.
value	The value for the constant; this can be an expression.

Details

This statement defines an identifier, and binds a constant value to it. During compilation, when the compiler finds an identifier, it substitutes the identifier with the value given for the constant.

When defining a constant, you can use any valid constant expression; this can be another constant, a string, or a mathematical expression. Of course, constant expressions cannot include variables, functions, or other elements of code which are not constant (and, hence, cannot be resolved during compilation).

Global constants are usually declared in the header file 16.

Examples

```
const foo = "abc" + "fddf"
const bar = 123 + 56 * 56
const foobar = "dfdffgfg"
```

Declare Statement

Function: Declares a function or a subroutine or a variable for later use.

Syntax: **declare function** name [([**byref**] argument1 **as** type1, [**byref**] argument2 **as** type2,...)] **as** ret_type

or:
declare sub name [([**byref**] argument1 **as** type1, [**byref**] argument2 **as** type2,...)]

or:
declare name [(bounds1)] [, name2 [(bounds2)]] **as** type [(max_string_size)]

Scope: Global and HTML

See Also: Function Statement 91, Sub Statement 99, Dim Statement 86

Part	Description
function	Optional. Used to specify that you are declaring a function procedure 66.
sub	Optional. Used to specify that you are declaring a sub procedure 66. If neither **sub** nor **function** appear, it is assumed that the **declare** is used to declare a variable.
name	Required. Used to specify the name of the function, sub or variable you are declaring.
byref	Optional. If present, arguments are passed by reference 68. If not, arguments are passed By Value 68.
argument1[1, 2...]	Optional. The name of the argument to be passed to the procedure. Only used if **sub** or **function** appear.
as	Required. Precedes the type definition.
type[1, 2...]	Optional (required if arguments are present). Specifies the type 48 of the argument to be passed to the procedure.
ret_type	Optional (required for functions, cannot be used for subs). Used to specify the type of the return value from the procedure.
bounds[1, 2...]	Optional (used only when declaring variables). Specifies the boundary (finite size) of a dimension in an array.

Details

In large projects, you often define a function or variable in one compilation unit, and use it from other units, so it is external to those units.

The unit which uses this external variable or function should refer to it in a way which lets the compiler know that it does indeed exist externally.

The **declare** statement is used to refer to a variable or function in this manner, but doesn't actually allocate any memory or produce any code; rather, it tells the compiler about this external entity, so that the compiler knows about it and can deal with it (see Dim Statement 86).

Usually, variables and functions which are shared between compilation units are declared in a header file, and this header is then included in the units (see Include Statement 95).

Example

```
declare function hittest(x as integer, y as integer) as boolean
declare sub dosomething (s as byref string)
declare devicestate as integer
```

Dim Statement

Function: Defines a variable and allocates memory for it.

Syntax: [**public**] **dim** name1 [(bounds1)] [, name2 [(bounds2)]]
as type [(max_string_size)]

Scope: Global, HTML and local

See Also: Declare Statement 84

Part	Description
public	Optional; may only be used in a global scope. If present, makes the variable(s) public 61.
name[1, 2...]	Required. Specifies the name for the variable.
bounds[1, 2...]	Optional. Specifies the boundary (finite size) of a dimension in an array. Several comma-delimited boundary values make a multi-dimensional array.
as	Required. Precedes the type definition.
type	Required. Specifies the type 48 of the variable.
max_string_size	Optional (can be used only when *type* is string). Sets the maximum size for a string (default size is 255 bytes).

Details

The **dim** statement creates a variable in the current scope. It reserves memory space for this variable. Hence, as part of a **dim** statement, you have to specify the *type* of the variable; specifying the type also defines how much memory will be allocated for this variable.

When creating strings, you can explicitly define their maximum size by including it in parentheses immediately following the **string** keyword. The default maximum size of strings is 255 bytes, but if you're sure a string will contain less than 255 bytes of data, it is better to constrain it to a lower size (and thus reduce the memory footprint of your program).

The **dim** statement can also be used to create *arrays* 54. This is done by specifying the number of elements in the array in parentheses immediately following the name of the variable. Multi-dimensional arrays are created by specifying the number of elements in each dimension, separated by commas.

Note that creating a variable using **dim** does not assign any value (i.e, 0) to this variable.

There are alternative ways of specifying the number and size of each array dimension in an array. Please, examine the examples below.

Examples

```
dim x, y(5) as integer ' x is an integer; y is a one-dimensional array of 5 integers.
dim z(2, 3) as byte ' a two-dimensional array, 2x3 bytes.
```

```
dim z2(2) as byte(3) ' same -- a two-dimensional array, 2x3 bytes
dim z2 as byte(2,3) ' same again -- a two-dimensional array, 2x3 bytes

dim s as string(32) ' s is a string which can contain up to 32 bytes
dim s(10) as string(32) 'array of 10 strings with 32-byte capacity
dim s as string(32)(10) 'alternative way to make the same definition
```

Doevents Statement

Function: Interrupts the current event handler and processes all events in the queue at the moment of invocation.

Syntax: **doevents**

Scope: Global, HTML and local

See Also: Declare Statement 84

Details

Executes events pending in the queue, then returns execution to current event handler. See doevents 73 above.

Examples

```
'calculate sum of all array elements -- this will surely take time!
dim sum, f as word
sum = 0
for f = 0 to 49999
      sum = sum + arr(f)
      doevents 'don't want to stall other events so allow their execution
while we are crunching numbers
next f
```

Do... Loop Statement

Function: Repeats a block of statements while a condition is **True** or until a condition becomes **True**.

Syntax:

do [**while** | **until**] expression
statement1
statement2
...
[exit do]
...
statementN
loop

or:

do
statement1

statement2
...
[exit do]
...
statementN
loop [**while** | **until**] expression

Scope: Local and HTML

See Also: For... Next Statement 90, While-Wend Statement 101, Exit Statement 89

Part	Description
expression	A logical expression which is evaluated either before or after the first time the statements are executed.
statement[1, 2...]	Lines of code to be executed.

Details

The **do-loop** statement repeats a block of code. If the condition (**while** or **until**) is included at the end of the loop (after the **loop** keyword), the block of code is executed at least once; If the condition is included at the beginning of the loop (after the **do** keyword), the condition must evaluate to **true** for the code to execute even once.

Any number of **exit do** 89 statements may be placed anywhere in the **do... loop** as an alternate way to exit the loop. These can be used as an alternate way to exit the loop, such as after evaluating a condition mid-loop using an **if... then** 94 statement. **Exit do** statements used within nested **do-loop** statements will transfer control to the loop which is one nested level above the loop in which the **exit do** occurs.

Examples

```
   = =~ =

D= ñ~     =   =    =       = ó   ~ñW
 =Z=M

       =Z= =H=N
    =       = =Z=NM

D= ñ~     =   =    =       = ó   ~ñW
 =Z=M
  =       = =Z=NM
       =Z= =H=N
    =
```

Enum Statement

Function: Declares a type for an enumeration.

Syntax: **enum** name

```
    const1 [ = value1],
    const2 [ = value2],
    ...
end enum
```

Scope: Global and HTML

See Also: Const Statement 84

Part	Description
name	Required. The name of the **enum** type. The name must be a valid Tibbo Basic identifier, and is specified as the type when declaring variables or parameters of the **enum** type.
const[1, 2...]	Required. The name for the constant in the enum.
value1	Optional. A value associated with the constant.

Details

Enum types can be useful for debugging purposes. When you add an enum type to a watch, you will see the constant name within the enum, and not a meaningless number.

By default, constants get incremental values, starting from 0 for the first item. You can think of this as a counter, enumerating the items in the list. Explicit values may be specified for any constant on the list. This also sets the counter to this explicit value. The counter will continue to increment from this new value.

Examples

```
enum my_enum
      my_const1, ' implicit value -- 0 is assumed
      my_const2 = 5, ' explicit value of 5, counter is set to 5 too.
      my_const3 ' Counter increments to 6,implicit value of 6.
end enum
```

Exit Statement

Function: Immediately terminates execution of function or a loop.

Syntax:

```
exit do
exit for
exit function
exit sub
exit while
```

Scope: Local and HTML

See Also: Do-Loop Statement 87, For... Next Statement 90, Function Statement 91, Sub Statement 99, While-Wend Statement 101

Part	Description
exit do	Provides an alternative way to leave a **do... loop** statement. Can be used only from within such a loop. **exit do** transfers the control to the statement following the **loop** statement. When used within nested **do... loop** statements, **exit do** transfers control to the loop which is one nested level above the loop in which **exit do** occurs.
exit for	Provides a way to exit a **for** loop. Can be used only within **for... next** or **for each... next** loops. When using **exit for**, control is passed to the statement which is immediately after the **next** statement. When used within nested **for** loops, **exit for** transfers control to the loop which is one nested level above the loop in which **exit for** occurs.
exit function	Exits the current function. Execution resumes from the point where the function was originally called.
exit sub	The same as **exit function**, only used for subroutine procedures.
exit while	Exists a **while** loop before the condition it is dependant upon evaluates as **false**.

Details

Do not confuse **exit** (which just quits) with **end** (which defines the end of a section of code).

Examples

```
function send_data as integer
      ' before sending data, we want to make sure Ethernet interface is OK
      if net.failure <> 0 then
            send_data = 1 ' this way we notify the caller of an error
            exit function
      end if
      ' Ethernet interface is OK, proceed with sending data
end function
```

For... Next Statement

Function: Repeats a block of statements while a counter increments until it reaches a set value.

Syntax:

for name = start_expression **to** end_expression [**step** step_number]
    statement1
    statement2
    ...
    [exit for]
    ...
    statementN
next name

Scope: Local and HTML

See Also: Do-Loop Statement 87, Exit Statement 89, While-Wend Statement 101

Part	Description
name	Required. The name of the counter. This has to be a numeric variable, which was previously explicitly defined using a Dim Statement 86.
start_expression	Required. The initial value of the counter. This actually sets the value of the *name* counter to the result of this expression. Can be a numerical constant, or a more complex expression.
end_expression	Required. The end value for the counter; once the counter reaches it, execution of the **for... next** loop stops. Can be a numerical constant, or a more complex expression.
step_number	Optional. Defines the intervals in which the *name* counter is incremented on every pass of the loop. This must be a numerical constant, and can be either positive or negative.
statement[1, 2...]	Required. Lines of code to be executed as long as the counter is 'in range' -- between the *start_expression* and the *end_expression*.

Details

It is not advised to change the value of the counter while within a **for... next** loop. You might get unexpected results, such as infinite loops.

For... next loops may be nested, using different counter variables (*name* above). There is also an exit 89 statement which can be used to terminate them abruptly.

Under Tibbo Basic, you are required to explicitly state the name of the cycle variable to be incremented immediately following the **next** keyword.

Examples

```
dim a(10) as integer
dim f as byte

for f = 0 to 9 ' in a 10-member array, element indices are 0 to 9
      sum = sum + a(f)
next f
```

Function Statement

Function: Used to define functions 66 -- distinct units in code which perform specific tasks and always return a value.

Syntax: [**public**]
function name [([**byref**] argument1 **as** type1, [**byref**] argument2 **as** type2...)]
as ret_type

```
    statement1
    statement2
    ...
    [exit function]
    ...
    statementN
end function
```

Scope: Global

See Also: Declare Statement 84, Exit Statement 89, Sub Statement 99

Part	Description
public	Optional. If set, allows other compilation units (files in a project) to access the function.
name	Required. Specifies the name for the function (used to call it, etc).
byref	Optional. If present, the argument immediately following this modifier will be passed by reference 68.
argument[1, 2...]	Optional. The name of the argument(s) passed to the function; arguments must have a name which is a valid identifier. This is a local identifier, used to refer to these arguments within the body of the function.
as	Optional (required if arguments are specified). Precedes the type definition.
type[1, 2...]	Optional (required if arguments are specified). Specifies the data type 48 for the argument. Each argument name must be followed with a type definition, even when specifying several arguments of the same type.
ret_type	Required. Specifies the type of the value the function will return. In effect, this is the data type of the function.
statement[1, 2...]	Required. The body of code executed within the function; specifies the actual 'work' done by the function.

Details

Functions cannot be nested (which is why their scope is defined as global or HTML) . Function always return a single value. Functions can call other functions and subroutines.

The return value of a function must be explicitly set from within the body of the function, by referring to the name of the function as a variable (of type *ret_type*) which is then assigned a value.

Examples

```
'this is just an example to show how functions call each other. It's not
actually useful.
declare subtract (x as byte, y as byte) as integer ' have to declare,
because it's invoked before its body.
```

```
function distance(x as byte, y as byte) as integer
      if x>y then
            distance = subtract(x,y)
      else
            distance = subtract(y,x)
      end if
end function
...
function subtract (x as byte, y as byte) as integer
      subtract = x - y
end function
```

Goto Statement

Function:	Jumps to a specific point in code, marked by a label.
Syntax:	**goto** label ... label:
Scope:	Local and HTML
See Also:	---

Part	Description
label	Required. Marks a specific point in code.

Details

Unconditionally jumps to label in code. Notice that all goto labels are local -- you cannot use goto statement to jump from within one procedure into another procedure!

Examples

```
dim arr1(5),arr2(5),f as byte

sub on_sys_init

arr1(0) = 1
arr1(1) = 2
arr1(2) = 3
arr1(3) = 4
arr1(4) = 5

arr2(0) = 1
arr2(1) = 2
arr2(2) = 2
arr2(3) = 4
arr2(4) = 5

'compare arrays and jump if not exactly the same
for f=0 to 4
```

```
      if arr1(f)<>arr2(f) then goto not_the_same
next f
'here when both arrays contain the same data
exit sub

'here when arrays are not the same
not_the_same:
'... place code here ...
```

If.... Then... Else Statement

Function: A way to conditionally execute code.

Syntax:

if expression **then**
    statement1
    statement2
    …
[**else**
    statement1
    statement2
    …
]
end if

or:

if expression **then** true_statement1 **:** true_statement2 …

Scope: Local and HTML

See Also: Select-Case Statement 97

Part	Description
expression	Required. The result of this expression (**true** or **false**) is then used to determine what code will execute.
statement[1, 2...]	Required. Statements to execute.
:	Optional. Separator for multiple statements on a single line. Covered under Programming Fundamentals 43.

Details

When the expression evaluates to **true**, the block of code immediately following the **then** keyword is executed. If it evaluates to **false**, the code immediately following the **else** keyword is executed; if there is no **else** keyword, program flow resumes from the line immediately following the **end if** keyword.

When using the single-line syntax (as in the lower example above), an **if** statement must not be terminated using an **end if**. This is the only construct under Tibbo Basic where line end matters. So, if you want to have several statements in such a construct, you need to place them all in the same line, and separate them with colons.

If... then... else statements may be nested.

* Currently, single-line **if... then** statements cannot contain an **else** clause.

  The **elseif** syntax is not currently supported at all (even on multi-line **if... then** statements).

Examples

```
if net.failure=1 then
      if sys.runmode=0 then ser.setdata("Ethernet failure!"):ser.send
'massage when in debug mode
      sys.halt
else
      if net.linkstate=0 then
            ser.setdata("No Ethernet link!")
      else
            if net.linkstate=1 then
                  ser.setdata("Linked at 10Mbit/s.")
            else
                  ser.setdata("Linked at 100Mbit/s.")
            end if
      end if
      ser.send
end if
```

Include Statement

Function:	Includes a file (such as a header file) at the point of the statement.
Syntax:	**include** "filename"
Scope:	Global and HTML
See Also:	Includepp Statement 96

Part	Description
"filename"	Contains the filename to be included. The path can be a relative path to the project path 16, an absolute path (such as c:\myfolder\myfile.tbs) or even a UNC path (such as \\MY-SERVER\Main\myfile.tbs).

Details

Makes compiler include the contents of a file at the point of the include statement. Usually used to include header files with declarations 84, definitions for constants 84, enum types 88, etc.

Examples

File: global.tbh (a header file)

```
Declare Function multiply(x As Byte, y As Byte) As Integer
Const k=3 'a crucially vital global constant.
```

File: main.tbs (a source code file)

```
Include "global.tbh" ' now we have access to multiply and to K.

Sub on_sys_init
'     ...
      Dim result As Integer
      result = multiply(k, 3)
'     ...
End Sub
```

File: library.tbs (a source code file)

```
Include "global.tbh"

Public Function multiply(x As Byte, y As Byte) As Integer

      multiply = x * y

End Function
```

Includepp Statement

Function: Includes preprocessor directives 76 found in a specified file at the point of the statement.

Syntax: **includepp** "filename"

Scope: Global and HTML

See Also: Include Statement 95

Part	Description
"filename"	Contains the filename to be parsed for preprocessor directives. The path can be a relative path to the project path 16, an absolute path (such as c:\myfolder\myfile.tbs) or even a UNC path (such as \\MY-SERVER\Main\myfile.tbs).

Details

Makes compiler parse the specified file and include all lines that start with "#". This is mainly intended to facilitate inclusion of preprocessor directives from library configuration files.

Examples

As an example, consider the setting configuration file 670, which defines the list of settings to be maintained by the STG library 668. The file is supposed to provide a

list of #define 76 statements to override default #define statements of the STG library.

File: global.tbh (a header file)

```
includepp "settings.xtxt"
include "settings\trunk\settings.tbh"
```

File: settings.xtxt (setting descriptor file) -- actual contents

```
>>BT  E      B      1      0      255    A      0      This is a Byte setting
>>WD  E      W      1      0      65535  A      0      This is a word setting
>>ST  E      S      1      0      16     A      ^      This is a String
setting
>>DD  E      D      1      0      4      A      ^      This is a dot-decimal
String setting
#define STG_DESCRIPTOR_FILE "settings.xtxt"
#define STG_MAX_NUM_SETTINGS 4
#define STG_RAM_ARRAY_SIZE 0
#define STG_MAX_SETTING_NAME_LEN 2
#define STG_MAX_SETTING_VALUE_LEN 16
```

As a result, all #define statements from settings.xtxt will be included. Lines that do not start with "#" will be ignored.

Select-Case Statement

Function: A way to conditionally execute code.

Syntax:

```
select select_expression
        case expression1_1 [ , expression1_2, … ] [ : ]
                statement1_1
                statement1_2
                …
        case expression2_1 [ , expression2_2, … ] [ : ]
                statement2_1
                statement2_2
                …
        …
        [ case else [ : ]
                statementN_1
                statementN_2
                …
        ]
end select
```

Scope: Local and HTML

See Also: If.... Then... Else Statement 94

Part	Description

select_expression	Required. The expression which is evaluated first; subsequent expressions are tested to match this expression. If a match is found, the statements contained within this **case** clause are executed, and execution then resumes from the line immediately following **end select**.
expression[1_1...N_1]	Required. An expression to evaluate; If it matches the select_expression, the statements included in this **case** clause are executed.
statement[1_1...N_1]	Required. Statements to execute when expression[1_1...N_1] matches select_expression.
:	Optional. Maintained for backwards compatibility -- some versions of BASIC in the past required a colon following every **case** expression.
case else	Optional. Precedes a block statements which is executed if neither of the earlier **case** clause match the select_expression. If present, must be the last **case** clause.

Details

It is of note that once a matching **case** clause is found, no other **case** clauses are tested; the code within the matching clause is simply executed, and execution resumes from the line following **end select**.

Also, remember that writing two select_expressions at the same time does not mean that there will be a shared code for both of them. Rather, it means that the first expression will have no code associated with it!

```
...
case 1 : 'this expression has no code associated with it
case 2 : 'code 'x=5' belongs to this expression
   x=5
...
```

Correct way to have a single block of code for two expressions is as follows:

```
...
case 1,2 : 'x=5 will be done both for 1 and 2
   x=5
...
```

Examples

```
sub print_weekday (weekday as byte)
      select case weekday
             case 1 : ser.setdata("Monday")
             case 2 : ser.setdata("Tuesday")
             case 3 : ser.setdata("Wednesday")
             case 4 : ser.setdata("Thursday")
             case 5 : ser.setdata("Friday")
             case 6 : ser.setdata("Saturday")
```

```
            case 7 : ser.setdata("Sunday")
            case else : ser.setdata("Did you just invent a new day?")
      end select
      ser.send
end sub
```

Sub Statement

Function: Used to define subs[66] -- distinct units in code which perform specific tasks. These never return any value.

Syntax:

[**public**]
sub name [([**byref**] argument1 **as** type1, [**byref**] argument2 **as** type2...)]
    statement1
    statement2
    ...
    [exit sub]
    ...
    statementN
end sub

Scope: Global

See Also: Declare Statement[84], Exit Statement[89], Function Statement[91]

Part	Description
public	Optional. If set, allows other compilation units (files in a project) to access the function.
name	Required. Specifies the name for the function (used to call it, etc).
byref	Optional. If present, the argument immediately following this modifier will be passed by reference[68].
argument[1, 2...]	Optional. The name of the argument(s) passed to the function; arguments must have a name which is a valid identifier. This is a local identifier, used to refer to these arguments within the body of the function.
as	Optional (required if arguments are specified). Precedes the type definition.
type[1, 2...]	Optional (required if arguments are specified). Specifies the data type[48] for the argument. Each argument name must be followed with a type definition, even when specifying several arguments of the same type.
statement[1, 2...]	Required. The body of code executed within the function; specifies the actual 'work' done by the function.

Details

Subroutines cannot be nested (which is why their scope is defined as global or HTML). Subroutines do not return values. Subroutines can call other subroutines

and functions.

Examples

```
sub print_to_serial(s as byref string)
      ser.setdata(s)
      ser.send
      s = "OK" ' This sub actually returns something, if indirectly.
end sub
```

Type Statement

Function: Used to declare structures 58 -- combinatorial data types that includes one or several *member* variables.

Syntax:

type type_name
    name1 [(bounds1)] **as** type [(max_string_size)]
    ...
    nameN [(boundsN)] **as** type [(max_string_size)]
end type

Scope: Global, HTML and local

See Also: Dim Statement 86

Part	Description
type_name	Required. Specifies the name for this structure *type* (not a particular variable).
name[1, 2...]	Required. Specifies the name for the member variable.
bounds[1, 2...]	Optional. Specifies the boundary (finite size) of a dimension in an array. Several comma-delimited boundary values make a multi-dimensional array.
as	Required. Precedes the type definition.
type	Required. Specifies the type 48 of the variable.
max_string_size	Optional (can be used only when *type* is string). Sets the maximum size for a string (default size is 255 bytes).
end type	Required. Closes type declaration.

Details

Structures can include any number of members, and each member can be of any type. Any member can also be an array or another structure. Nesting of up to 8 levels is allowed (i.e. "array within a structure within a structure within and array" -- each structure or array is one level and up to 8 levels are possible).

Type...end type is only a declaration, not a variable definition! You still need to use a regular dim 86 statement to define a variable of the type you have declared.

Examples

```
'declare new type
```

```
type my_struct
   x as byte
   y as Long
   s as string(10)
end type

'define a variable of this type
dim my as my_struct
```

While-Wend Statement

Function: Executes a block of code as long as an expression evaluates to **true**.

Syntax:

while expression
    statement1
    statement2
    ...
    [exit sub]
    ...
    statementN
wend

Scope: Local and HTML

See Also: Do-Loop Statement 87, For... Next Statement 90, Exit Statement 89

Part	Description
expression	Required. The expression which is to be evaluated.
statement[1, 2...]	Required. The code to run when the expression is **true**.

Details

Makes a pre-conditional loop. First, the expression is evaluated and then if it is **true** statement1, statement2, etc are executed. Then expression is evaluated again and so on until expression becomes **false**.

Examples

```
function wait_char(ch as byte) as byte
' waits for specific character with ASCII code ch to arrive into the serial
port.
' returns 0 if char was encountered or 1 if this character was encountered

dim s as string(1)

      ' input data byte by byte for as long as there is some data left to
process
      s = ser.getdata(255) ' will input byte by byte as s only can contain a
single char!
      while len(s)<>0
```

```
            if s = chr(ch) then
                  wait_char = 0 ' character encountered!
                  exit function
            end if
            s = ser.getdata(255)
      wend

      wait_char = 1 ' did not encounter ch character and there is no more
data to input (for now)!

End Function
```

Keywords

This chapter contains links from single keywords to the statements in which you may find them. It is meant to be used as a resource for context-sensitive help.

As

A keyword designating data type. Appears as part of the following statements:

Declare Statement 84
Dim Statement 86
Function Statement 91
Sub Statement 99

Boolean

A data type. Please see Variables And Their Types 48.

ByRef

A keyword designating a way of passing arguments. Appears as part of the following statements:

Function Statement 91
Sub Statement 99

For further details, see Passing Arguments to Procedures 68.

Byte

A data type. Please see Variables And Their Types 48.

ByVal

A keyword designating a way of passing arguments. Appears as part of the following statements:

Function Statement 91

Sub Statement 99

For further details, see Passing Arguments to Procedures 68.

Char

A data type. Please see Variables And Their Types 48.

Else

A keyword used to denote conditions. Appears as part of the following statements:

If.... Then... Else Statement 94

Select-Case Statement 97

End

This keyword is used in the following statements:

Enum Statement 88

Function Statement 91

Select Statement 97

Sub Statement 99

Type Statement 100

False

A byte constant with a value of 0, associated with boolean 48 variables.

For

Appears as part of the following statement:

For... Next Statement 90

Integer

A data type. Please see Variables And Their Types 48.

Next

Appears as part of the following statement:

For... Next Statement 90

Public

A keyword used to denote visibility. Appears as part of the following statements:

Dim Statement 86
Function Statement 91
Sub Statement 99

For further details, see Understanding the Scope of Variables 61.

Short

A data type. Please see Variables And Their Types 48.

Step

Appears as part of the following statement:

For... Next Statement 90

String

A data type. Please see Variables And Their Types 48.

Then

A keyword used to denote conditional execution. Appears as part of the following statement:

If.... Then... Else Statement 94

Type

Appears as part of the following statement:

Type Statement 100

To

Appears as part of the following statement:

For... Next Statement 90

True

A byte constant with a value of 1, associated with boolean 48 variables.

Word

A data type. Please see Variables And Their Types 48.

Operators

Tibbo Basic supports the following operators:

+ Operator

Addition operator (applies to strings as well).

```
i = 1 + 2 ' this would be 3
s = "foo" + "bar" ' this would be "foobar"
```

* Operator

Multiplication operator.

```
i = 5 * 2 ' 10.
```

- Operator

Subtraction Operator.

```
i = 20 - 5 ' 15
```

/ Operator

Division operator.

```
i = 30 / 10 ' 3
i = 10 / 3 ' also 3 -- only integers are supported, decimal part is removed.
```

MOD Operator

Used to divide two numbers and return the remainder.

```
i = 10 mod 3 ' this would be 1
```

= Operator

(1) Equality operator. **(2)** Assignment operator.

```
if i = 5 then .... ' as an equality operator

i = 5 ' as an assignment operator
```

AND Operator

(1) Logical AND. **(2)** Bitwise AND.

```
if i = 5 AND j = 10 then.... ' as a logical AND


x =
&b01011001 AND
&b10101011 ' this would be
&b00001001
```

NOT Operator

(1) Logical NOT. **(2)** Bitwise NOT.

```
if NOT b then.... ' as a logical NOT -- b is a boolean value

x = NOT &b01011001 ' this would be &b10100110
```

OR Operator

(1) Logical OR. **(2)** Bitwise OR.

```
if i = 5 OR j = 10 then.... ' as a logical OR

x =
&b10110101 OR
&b01011001 ' this would be
&b11111101
```

XOR Operator

(1) Logical XOR. **(2)** Bitwise XOR.

```
if i = 5 XOR j = 10 then.... ' as a logical XOR

&b10110101 XOR
&b01011001 ' this would be
&b11101100
```

Error Messages

Below is a listing of all error messages which may appear when trying to build and upload your Tibbo BASIC program onto a target.

C1001

Description:

This error occurs when a source file contains an "illegal" character (like a special character, a non-English letter, etc) outside of a string literal or a comment.

Example:

```
%dim x as byte ' error C1001: invalid char '%' (25)
```

See Also

- Naming Conventions 47

C1002

Description:

This error occurs when source code contains a line break inside a string literal or a character constant.

Example:

```
S = "I am a string
literal "  ' error C1002: newline in constant
```

See Also

- Variables And Their Types 48

C1003

Description:

Unlike string literals, character constants may not be empty.

Example:

```
x = `` ' error C1003: empty char constant
```

See Also

- Constants 65

C1004

Description:

Character constants may contain two characters at most (in which case, the constant type will be 'word').

Example:

```
x = `abc` ' error C1004: too many chars in constant
```

See Also

- Constants 65

*** 7

C1005

Description:

The compiler detected an attempt to specify a numerical constant using an incorrect format.

Example:

```
x = &G12 ' error C1005: invalid numeric constant: unknown base 'G'
x = &b102 ' error C1005: invalid numeric constant
x = 10a ' error C1005: invalid numeric constant
```

See Also

- Type Conversion 50

C1006

Description:

This error occurs when Tibbo BASIC syntax is violated. Refer to the documentation of a particular statement to see its syntax.

Example:

```
select x ' error C1006: 'Case' expected
case 1: case 2:
       b = true
case 3:
       b = false
```

```
      case else:
            sys.halt
      end select

sub on_init
      ...
end function ' error C1006: 'End Sub' expected
```

See Also

- Language Reference 83

C1007

Description:

A numerical constant does not fit in any supported numerical type.

Example:

```
x = 65536 ' error C1007: Constant too big
x = -32769 ' error C1007: Constant too big

enum my_enum
      my_val1 = -32768,
      my_val2 = 32768
end enum ' error C1007: enum range is too wide
```

See Also

- Variables And Their Types 48
- Constants 65

C1008

Description:

This error occurs when an expression which should be constant is not actually constant.

Example:

```
dim x as integer
const a = 5 * x ' error C1008: constant expression expected
```

See Also

- Constants 65

C1009

Description:

This error occurs when the requested operation cannot be performed using the data types provided.

Example:

```
dim s as string
s = "abc" + 5 ' error C1009: type mismatch
```

See Also

- Variables And Their Types 48

C1010

Description:

This error occurs when attempting to re-use an identifier that is already used (and thus, cannot be reused in the current scope) in a definition or a declaration (of an enumeration type, a constant, a variable or a function).

Example:

```
dim x as string
sub x ' error C1010: redefinition of identifier 'x'
      …
end sub
```

See Also

- Identifier 137

*** 7

C1011

Description:

This error occurs when attempting to define a procedure twice.

Example:

```
sub x
end sub

sub x  ' error C1011:  'x' already has body
end sub
```

See Also

- Sub Statement 99
- Function Statement 91

C1012

Description:

The definition for a procedure does not match a previous declaration for that procedure (a different number or type of arguments and/or a return value)

Example:

```
declare sub x

sub x(i as integer) ' error C1012: argument count mismatch (see previous
declaration of 'x')
end sub
```

See Also

- Declare Statement 84
- Sub Statement 99
- Function Statement 91

C1013

Description:

This error occurs when a statement references an identifier which has not previously been defined.

Example:

```
dim x as integer
x = 15 * y ' error C1013: undeclared identifier 'y'
```

See Also

- Dim Statement 86

C1014

Description:

Identifiers can refer to entities of different types: labels, variables, procedures, enumeration types, constants. This error occurs when you cannot use identifier of a certain type in the current statement.

Example:

```
dim x as integer
goto x ' error C1014: 'x' is not a label
```

See Also

- Identifier 137

C1015

Description:

The **next** clause of a **for/next** statement must use the same index variable as the **for** clause.

Example:

```
dim i,j as integer
for i = 1 to 10
next j ' error C1015: 'For'/'Next' arguments mismatch
```

See Also

- For... Next Statement 90

C1016

Description:

The **exit** statement may be used only from within certain statements (**for**, **while**, **do-loop**, **sub**, **function**). This error occurs when the compiler encounters an **exit** statement which is used not from within one of these statements.

Example:

```
for i = 1 to 10
      exit while   ' error C1016: 'Exit' is of scope
next i
```

See Also

- Exit Statement 89

*** 7

C1017

Description:

Assignment statements must contain a value on left side of the equal sign (called an l-value). This can be a variable, an array element or a read-write property, and may not be a constant.

Example:

```
1 = x ' error C1017: l-value expected on the left of '='
```

C1018

Description:

Assignment statements must contain a value on right side of the equal sign (called an r-value). This error occurs when an expression on the right side of such a statement does not return a value.

Example:

```
sub sub1
…
end sub
```

```
…
x = sub1 ' error C1018: subroutine cannot be on the right of '='
```

C1019

Description:

This error occurs when attempting to access a variable which is not an array as if it were an array (by using an index number).

Example:

```
dim i,j as integer
i = j(3)  ' error C1019: j is not array
```

See Also

- Introduction to Variables, Constants and Scopes 47
- Arrays 54

C1020

Description:

This error occurs when trying to define an array with more than 8 dimensions (a maximum of 8 dimensions are allowed).

Example:

```
dim i(2,2,2,2,2,2,2,2,2) as integer ' error C1020: Too many array dimensions
(8 max)
```

See Also

- Arrays 54

C1021

Description:

This error occurs when a property was defined for read-only or for write-only, but the program tries to access this property in a different way.

Example:

```
sys.runmode = PL_SYS_MODE_DEBUG  ' error C1021: write access to property is
denied
```

See Also

- Understanding Platforms 82

C1022

Description:

There are several system calls which the compiler uses directly, and are invoked implicitly in code (string comparison, string copy, conversion from string to number, etc). This error occurs when a platform does not export these functions, but source code requires them.

Example:

```
doevents '  error C1022: platform does not export 'doevents' syscall
```

See Also

- Type Conversion 50

*** 7

C1023

Description:

The global scope may contain only declarations/definitions of procedures, variables, enumeration types and constants. This error occurs when some other statement is encountered at the global scope.

Example:

```
for i = 1 to 10  ' error C1023: Unexpected at global scope
next i
```

See Also

- Understanding the Scope of Variables 61

***[7]

C1024

Description:

This error occurs when an **include** statement references a file which cannot be read. Most commonly, it means the filename was misspelled.

Example:

```
include "utileties.tbh"  ' error C1024: unable to read file 'utileties.tbh'
```

See Also

- Include Statement[95]

L1001

Description:

This error is occurs when the linker tries to link two object files which have different data base addresses.

Lower addresses are reserved for passing arguments and returning values from platform syscalls. The data base address for program variables is calculated according to how much memory platform syscalls require for arguments and return values.

Most commonly, this error means that you are trying to link object files built for different platforms.

See Also

- System Components[7]

L1002

Description:

This error occurs when the linker attempts to link two object files with different counts of platform event handlers. Most commonly this error means that you are trying to link object files built for two different platforms.

See Also

- System Components[7]

L1003

Description:

During linking, one or more addresses remained unresolved. That means that these addresses are referenced from one or more compilation units but are never defined.

See Also

- Declare Statement 84

L1004

Description:

This error means that when linking two object files, the linker encountered a situation where the import is a data address and the export is a code address or vice versa.

L1005

Description:

Since all memory allocation is static, recursion is not supported. This error occurs whenever the linker encounters recursion (direct or indirect).

See Also

- Introduction to Procedures 66
- Our Language Philosophy 4

L1006

Description:

The TiOS Virtual Machine may hold up to 255 stack locations. This error occurs when the linker needs to reserve more stack locations. That does not necessarily mean that the direct call chain in program is 255 calls long. Each independent **doevents** statement approximately doubles the required needed stack locations.

See Also

- Doevents 73
- System Components 7

L1007

Description:

A function which contains a **doevents** statement may not be called from more than one independent call chain.

See Also

- Doevents 73

*** 7

L1008

Description:

The amount of RAM needed to store variables for your project exceeds the maximum possible size for this platform. In simple terms, your variables take up too much space.

See Also

- Platform Specifications 138

*** 7

L1009

Description:

The amount of FLASH (program memory) needed to store your project exceeds the maximum possible size for this platform. I.e, your program is too large.

See Also

- Platform Specifications 138

Objects, Properties, Methods, Events

Under Tibbo Basic, these are all platform-specific constructs.

Please refer to your Platform 138 documentation for details about the objects, properties, methods and events for your platform.

Development Environment

Below is an overview of the TIDE GUI.

As a general rule, those things which you know from common Windows programs (such as Window > Tile) will work just as you would expect them to work. We will concentrate on the more unique parts of the TIDE GUI.

Installation Requirements

The recommended system requirements for TIDE are:

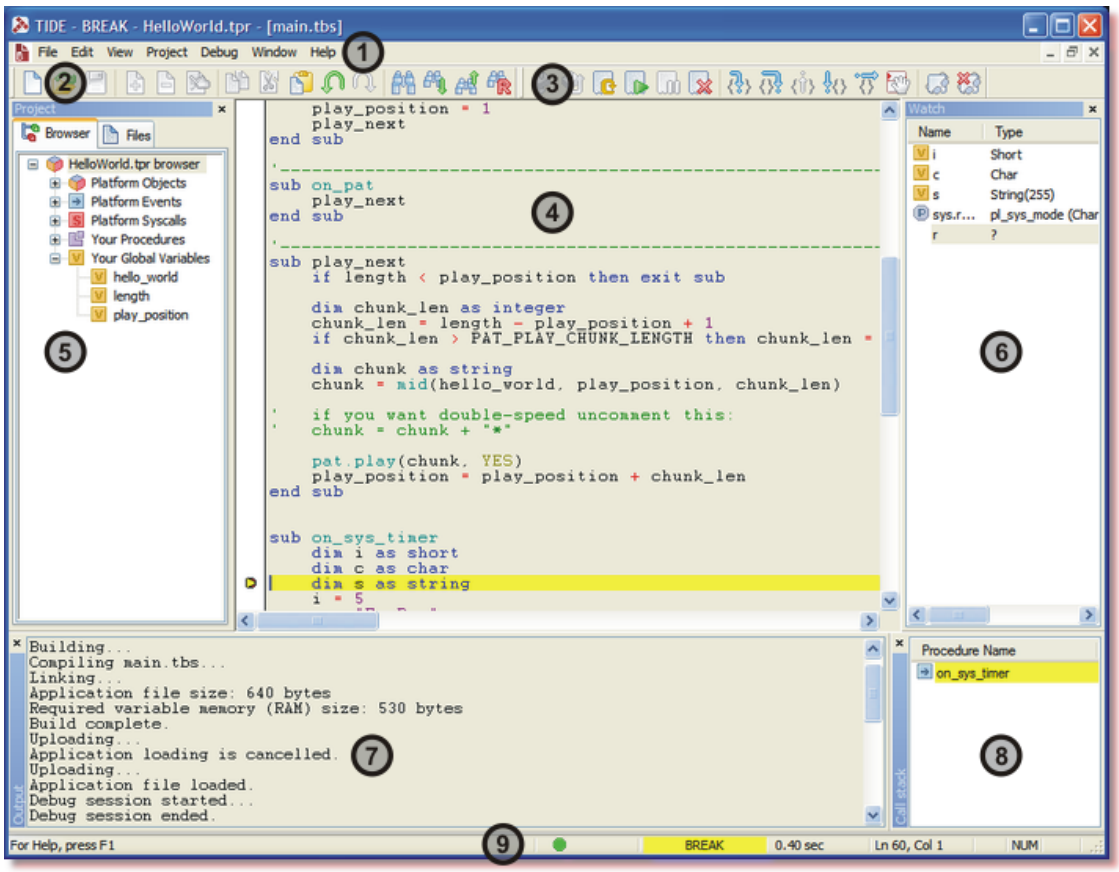
- 800 MHz Intel Pentium III processor (or equivalent) and later
- Windows 98, ME, 2000, XP, 2003
- 256 MB RAM
- 20MB of available disk space
- A communication medium with your target (platform-specific)
- At least one target device to work with (required for debugging)
- 1024 x 768, 24-bit display recommended

User Interface

The following is a systematic overview of the TIDE graphical user interface.

Main Window

The main window for TIDE looks like this:



The parts in the screenshot above are:

(1) The Menu Bar 121

(2) The Project Toolbar 126

(3) The Debug Toolbar 126

(4) The Code Editor 120

(5) The Project Pane 134

(6) The Watch Pane 135

(7) The Output Pane 133

(8) The Call Stack Pane 133

(9) The Status Bar 130

Operation Modes

Essentially, the TIDE GUI has two primary modes: The Edit Mode, in which you write the code for your application, and the Debug Mode, which is used while your code is running on the target and you are debugging it.

In each of these modes you may show, hide or resize various interface elements. The changes you make in one mode do not affect the other state. I.e, you could display the Debug Toolbar 126 while in Debug Mode and hide it while in Edit Mode. Every time you will go into Debug Mode, the toolbar would appear. When you switch back to the Edit Mode, the toolbar will disappear. Thus, you may customize your

workspace so it would serve you best both while editing code and while debugging.

Another key difference is that while in Debug Mode you cannot enter any new code. Whenever you try to type code while in this mode, you will be prompted to stop program execution on the target and switch back to Edit Mode.

You can easily tell the two modes apart by the background color of the code editor pane: When the background is white, you are in Edit Mode. When it's grey, you are in Debug Mode.

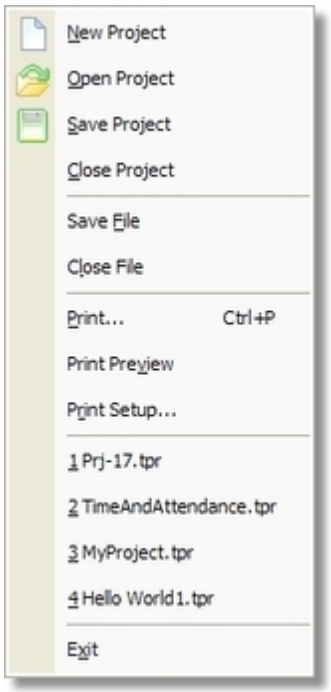
Menu Bar

A standard bar, modeled after the classic Windows menu bar.

See below:

- File Menu 121
- Edit Menu 122
- View Menu 122
- Project Menu 123
- Debug Menu 124
- Image Menu 124 (visible only when graphical resource is selected for editing)
- Window Menu 125
- Help Menu 125

File Menu 3.1



New Project: Displays the New Project 132 dialog.

Open Project: Opens an existing project

Save Project: Saves all modified files on this project. Happens automatically on build.

Close Project: Closes current project.

Save File: Saves the current file.

Close File: Closes the current file.

Print: Prints the current file.

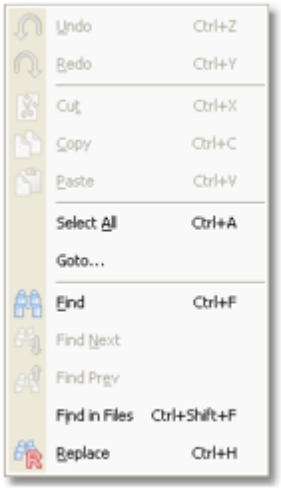
Print Preview: Check output before printing.

Print Setup: Configure printing.

1... 4: Recent files.

Exit: Quits TIDE.

Edit Menu3.2



Undo: Cancels last action.

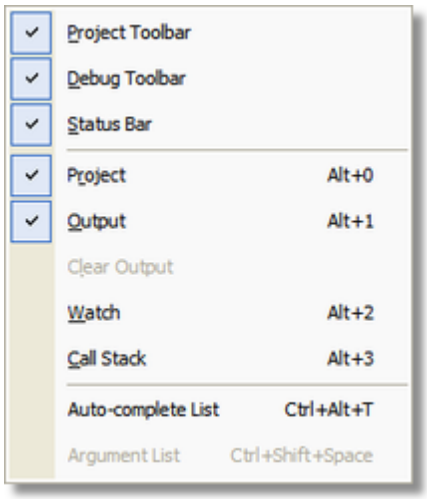
Redo: Cancels last undo.

Cut, Copy, Paste: Standard edit actions.

Select All: Select all text.

Find, Find Next, Find Prev, Find in Files, Replace: Standard find & replace actions.

View Menu3.3



Project Toolbar: Toggles (shows/hides) the Project 126 toolbar.

Debug Toolbar: Toggles the Debug 126 toolbar.

Status Bar: Toggles the Status Bar 130.

Project: Toggles the Project 134 pane.

Output: Toggles the Output 133 pane.

Clear Output: Clears the contents of the Output pane.

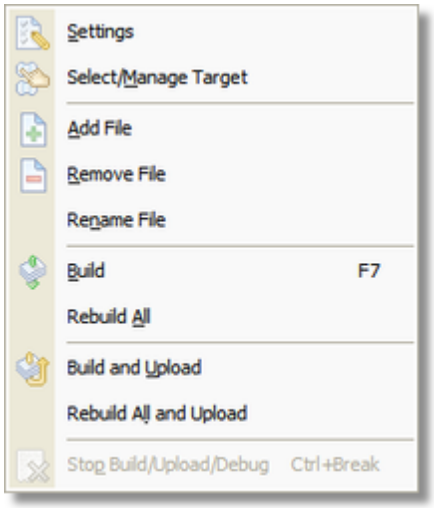
Watch: Toggles the Watch 135 pane.

Call Stack: Toggles the Call Stack 133 pane.

Auto-complete List: Show an appropriate auto-complete list 23 for the current context.

Argument List: Show a list of arguments for the current procedure call.

Project Menu



Settings: Displays the Project Settings 131 dialog.

Select/Manage Target: Shows the platform-specific dialog used to select a target for your project and upload firmware.

Add File: Displays the Add file to project 132 dialog.

Remove File: Removes the file currently selected in the Project Tree from the project. Does not delete file from disk.

Rename File: Renames the file currently selected in the Project Tree.

Build: Builds project without uploading. Builds only files modified since last build.

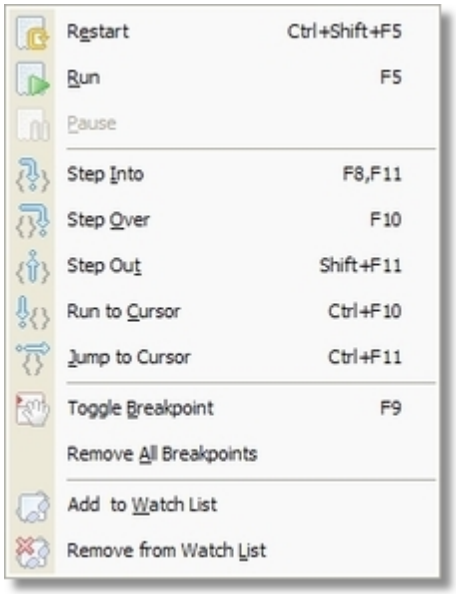
Rebuild All: Rebuilds project including all files -- even those not modified since last build.

Build and Upload: Builds if necessary, and uploads project to target without running it.

Rebuild All and Upload: Rebuilds project including all files, uploads and does not run.

Stop Build/Upload/Debug: Stops building, uploading or debugging. Exits debug mode. If the target was running, it will continue running.

Debug Menu5



Restart: Reboots the target device. Rebooting the device will not resume execution. Once the device has finished the reboot process, execution will be paused, pending further debug instructions.

Run: Begins or resumes program execution on target. If switching from Edit mode, this optionally compiles and uploads the project (when needed).

Pause: Pauses program execution. Covered under Target States 28 above.

Step Into: Covered under Stepping 33 above.

Step Over: Covered under Stepping 33 above.

Step Out: Covered under Stepping 33 above.

Run to Cursor: Covered under Stepping 33 above.

Jump to Cursor: Covered under Stepping 33 above.

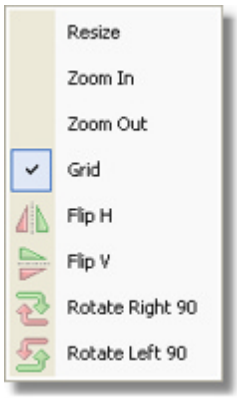
Toggle Breakpoint: Covered under Breakpoints 30 above.

Remove All Breakpoints: Covered under Breakpoints 30 above.

Add to Watch List: Covered under The Watch 33 above.

Remove from Watch List: Covered under The Watch 33 above.

Image Menu6



This menu is visible only when a graphical resource is selected for editing. TIDE

"knows" how to work with *.bmp*, *.jpg*, *.gif*, and *.png* files.

Resize: Changes the size of the image's "canvas". This is not re-sampling of the image: existing image elements won't shrink or get larger, just the total image size in pixels will change. If the new image size is smaller than portions of the image at the right and on the bottom will be lost.

Zoom In: Selects higher magnification. x1, x2, x4, x8, x16 magnification levels are available.

Zoom Out: Selects lower magnification. x1, x2, x4, x8, x16 magnification levels are available.

Grid: Toggles image grid visible/invisible.

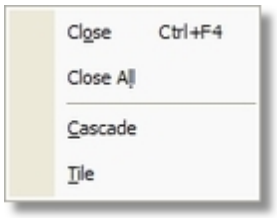
Flip H: Mirrors entire image or selected rectangular area (if selection is made) horizontally.

Flip V: Mirrors entire image or selected rectangular area (if selection is made) vertically.

Rotate Right 90: Rotates entire image or selected rectangular area (if selection is made) 90 degrees clockwise.

Rotate Left 90: Rotates entire image or selected rectangular area (if selection is made) 90 degrees counter-clockwise.

Window Menu

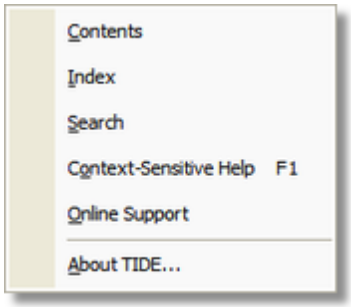


Close: Closes current window. Can also be done with Ctrl+W.

Close All: Closes all open documents without closing projects.

Cascade, Tile: Standard window actions.

Help Menu3.8



Contents: Opens the help file at the contents.

Index: Opens the help file at the index.

Search: Opens the help file in search mode.

Context-Sensitive Help: Opens the help file at the topic for the currently selected keyword in the code editor.

Online Support: Open http://www.tibbo.com/taiko.php using the default browser.

About TIDE: Displays version information, etc.

Toolbars

The TIDE has the following toolbars:

- Project Toolbar 126
- Debug Toolbar 126
- Image Editor Toolbar 127 (visible only when graphical resource is selected for editing)
- Tool Properties Toolbar 128 (visible only when graphical resource is selected for editing, contents depend on the selected tool)

Project Toolbar

- **New Project:** Displays the New Project 132 dialog.
- **Open Project:** Opens an existing project
- **Save Project:** Saves all modified files on this project. Happens automatically on build.
- **Add File:** Displays the Add file to project 132 dialog.
- **Remove File:** Removes current file from project. Does not delete file from disk.
- **Settings:** Displays the Project Settings 131 dialog.
- **Copy:** Self-explanatory.
- **Cut:** Self-explanatory.
- **Paste:** Self-explanatory.
- **Undo:** Cancels last action.
- **Redo:** Cancels last undo.
- **Find:** Self-explanatory.
- **Find Next:** Self-explanatory.
- **Find Prev:** Self-explanatory.
- **Replace:** Self-explanatory.

Debug Toolbar

Select/Manage Target: Shows the platform-specific dialog used to select a target for your project and upload firmware.

Build: Builds project without uploading. Builds only files modified since last build.

Build and Upload: Uploads project to target without running it. Builds prior to upload, if required.

Restart: Reboots the target device. Rebooting the device will not resume execution. Once the device has finished the reboot process, execution will be paused, pending further debug instructions.

Run: Begins or resumes program execution on target. If switching from Edit mode, this optionally compiles and uploads the project (when needed).

Pause: Pauses program execution. Covered under Target States 28 above.

Stop Build/Upload/Debug: Stops building, uploading or debugging. Exits debug mode. If the target was running, it will continue running.

Step Into: Covered under Stepping 33 above.

Step Over: Covered under Stepping 33 above.

Step Out: Covered under Stepping 33 above.

Run to Cursor: Covered under Stepping 33 above.

Jump to Cursor: Covered under Stepping 33 above.

Toggle Breakpoint: Covered under Breakpoints 30 above.

Add to Watch List: Covered under The Watch 33 above.

Remove from Watch List: Covered under The Watch 33 above.

Image Editor Toolbar

This toolbar is visible only when graphical resource is selected for editing. When you select a certain tool an additional Tool Properties Toolbar 128 specifically for this tool is displayed. This toolbar provides all options for the selected tool.

Selection Tool: Selects rectangular image area to move, copy, or transform.

Hand Tool: Allows you to scroll the image using "hand grip" (position the tool over the image, click and hold left mouse button, then drag the image).

Paint Tool: Use this tool to do "freehand" painting over the image.

Eraser Tool: Erases portions of the image (paints with background color).

Text Tool: Adds text to the image.

Line Tool: Draws straight lines.

Rectangle Tool: Draws rectangles.

Ellipse Tool: Draws ellipses.

Eyedropper Tool: "Picks" the color off the image (left-click over the image to pick the fore-color; right click to pick the background color).

Zoom Tool: Changes the zoom level (magnification). x1, x2, x4, x8, x16 zoom levels are available.

Tool Properties Toolbar

When you select a certain image edit tool[127] an additional properties toolbar specifically for this tool is displayed. This toolbar provides all options for the selected tool. The following toolbars are available:

- Selection Tool Properties Toolbar[128]
- Paint Tool Properties Toolbar[128]
- Eraser Tool Properties Toolbar[128]
- Text Tool Properties Toolbar[129]
- Line Tool Properties Toolbar[129]
- Rectangle Tool Properties Toolbar[129]
- Ellipse Tool Properties Toolbar[130]
- Zoom Tool Properties Toolbar[130]

Selection Tool Properties

Transparency Mode: For drag/drop and copy/paste operations selects whether pixels of background color will be copied to the destination as well. When the transparency mode is OFF the rectangular image fragment being copied or moved will overlay original image underneath completely. When the transparency mode is ON the original image will still be visible through the copied data.

Flip Horizontally: Mirrors entire image or selected rectangular area (if selection is made) horizontally.

Flip Vertically: Mirrors entire image or selected rectangular area (if selection is made) vertically.

Rotate Left 90: Rotates entire image or selected rectangular area (if selection is made) 90 degrees counter-clockwise.

Rotate Right 90: Rotates entire image or selected rectangular area (if selection is made) 90 degrees clockwise.

Paint Tool Properties

Inversion Mode: Toggles paint tool's inversion mode on/off. When inversion is OFF, left-clicking on an image pixel changes this pixel's color to fore-ground color; right-clicking changes the color to background color. With inversion ON, left-clicking causes the pixel to alternate between fore-ground and background colors.

Size: "Pencil tip" width in pixels.

Eraser Tool Properties

Size: "Eraser width" in pixels.

Text Tool Properties

Transparency Mode: selects whether original image will still be visible underneath the text. When the transparency mode is OFF, original image will not be visible. When the transparency mode is ON, original image will be visible.

Inversion Mode: Toggles text tool's inversion mode on/off. When inversion is OFF, the text will always be printed in the fore-ground color. When inversion is ON, the text will appear in fore-ground color over the areas originally painted in background color and in background color over the areas originally painted with non-background color.

Font Selector: Selects the font to print the text with.

Font Size: Selects the size of the font.

Bold: Toggles bold attribute for the font on/off.

Italic: Toggles italic attribute for the font on/off.

Underline: Toggles underline attribute for the font on/off.

Left: Selects left alignment for the text (result is visible with multi-line text only).

Center: Selects center alignment for the text (result is visible with multi-line text only).

Right: Selects right alignment for the text (result is visible with multi-line text only).

Line Tool Properties

Inversion Mode: Toggles line tool's inversion mode on/off. When inversion is OFF, the line will always be drawn in the fore-ground color. When inversion is ON, the line will appear in fore-ground color over the areas originally painted in background color and in background color over the areas originally painted with non-background color.

Size: Line width in pixels.

Rectangle Tool Properties

Inversion Mode: Toggles rectangle tool's inversion mode on/off. When inversion is OFF, the rectangle's border will always be drawn in the fore-ground color and filling (for filled rectangles) will always be done with background color. When inversion is ON, the rectangle's border will appear in fore-ground color over the areas originally painted in background color and in background color over the areas originally painted with non-background color. The color for the "filling" (when enabled) will be exactly opposite.

Solid Rectangle: Creates a rectangle filled with fore-ground color (+ inversion mode effect is applied with enabled).

Filled Rectangle: Creates a rectangle with the border of fore-ground color and filled with the background color (+ inversion mode effect is applied with enabled) .

Unfilled Rectangle: Creates a rectangle with the border of fore-ground color and no filling (+ inversion mode effect is applied with enabled).

Size: Border width (irrelevant for solid rectangles).

Corner rounding: Defines the radius of rectangle corners.

Ellipse Tool Properties

Inversion Mode: Toggles ellipse tool's inversion mode on/off. When inversion is OFF, the ellipse's border will always be drawn in the fore-ground color and filling (for filled ellipses) will always be done with background color. When inversion is ON, the ellipse's border will appear in fore-ground color over the areas originally painted in background color and in background color over the areas originally painted with non-background color. The color for the "filling" (when enabled) will be exactly opposite.

Solid Ellipse: Creates an ellipse filled with fore-ground color (+ inversion mode effect is applied with enabled).

Filled Ellipse: Creates an ellipse with the border of fore-ground color and filled with the background color (+ inversion mode effect is applied with enabled).

Unfilled Ellipse: Creates an ellipse with the border of fore-ground color and no filling (+ inversion mode effect is applied with enabled).

Size: Border width (irrelevant for solid ellipses).

Zoom Tool Properties

Zoom Level: Selects one of available zoom (magnification) levels.

Status Bar

Below is a screenshot of the status bar:

Target: Shows selected debug transport and target address (in this case, MAC address).

Progress Bar: Shows progress during long operations (uploading binary, etc).

Communication State indicator: Covered under Target States 28 above.

Target State indicator: Covered under Target States 28 above.

Timer: Covered under Code Profiling 37 above.

Cursor location: Lines and columns.

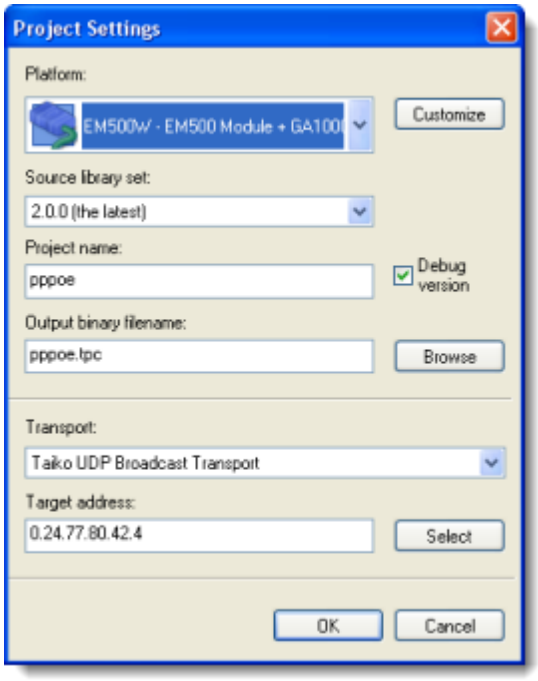
CAP, NUM, SCRL: Status indicators for Caps Lock, Num Lock and Scroll Lock.

Dialogs

Not all dialogs are reviewed -- only the ones which are not self-explanatory.

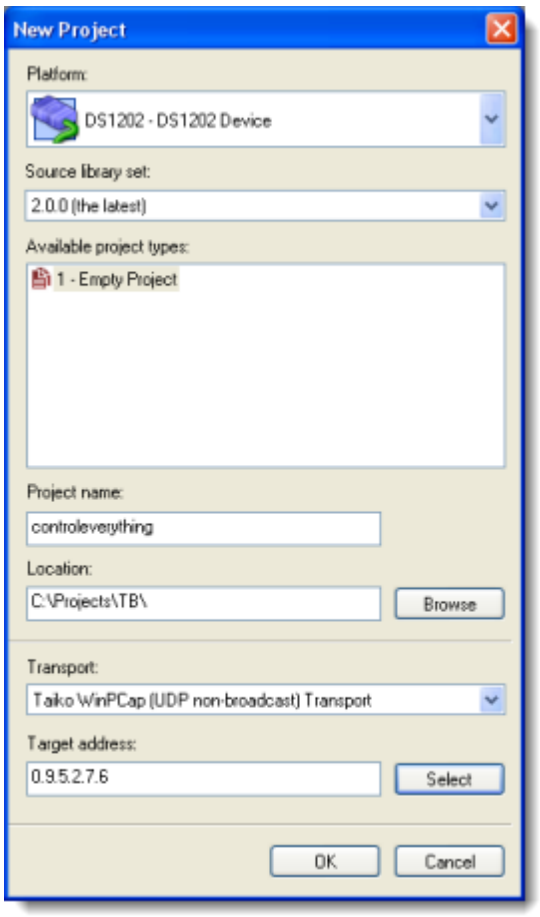
In this section:

Project Settings



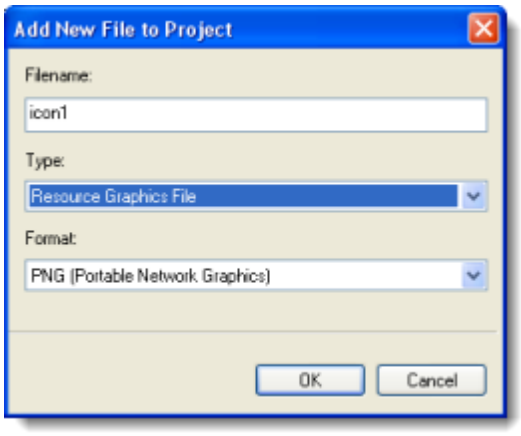
This dialog has been covered under Project Settings 38 above.

New Project 2



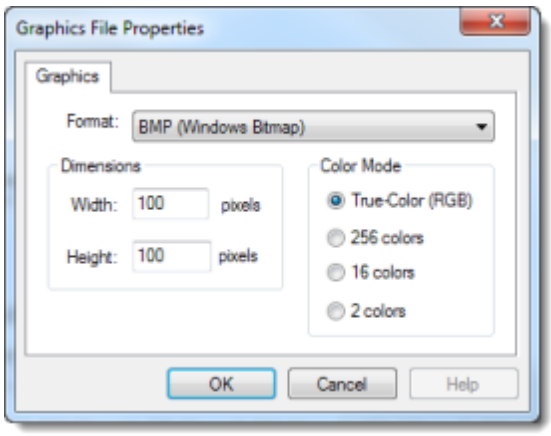
This dialog has been covered under Starting a New Project 10 above.

Add File to Project



This dialog has been covered under Adding, Removing and Saving Files 18 above.

Graphic File Properties Dialog



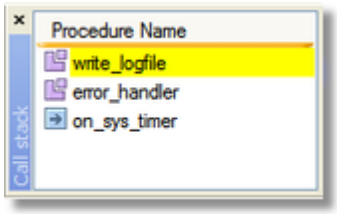
This dialog has been covered under Adding, Removing and Saving Files 18 above.

Panes

Some panes may be toggled using shortcut keys or the View Menu 122. Colors pane 135 is displayed automatically when an image resource file is opened for editing.

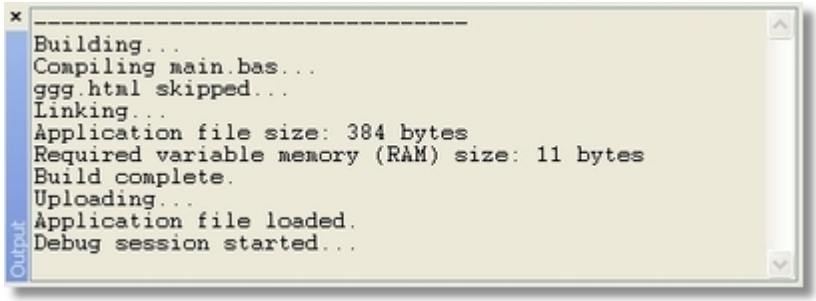
In this section:

Call Stack.7.1



The Call Stack pane is covered under The Call Stack 31 above.

Output6.2.7.2



Displays status messages while compiling, linking, uploading and debugging. Double clicking on an error message would move the cursor to the line of code which

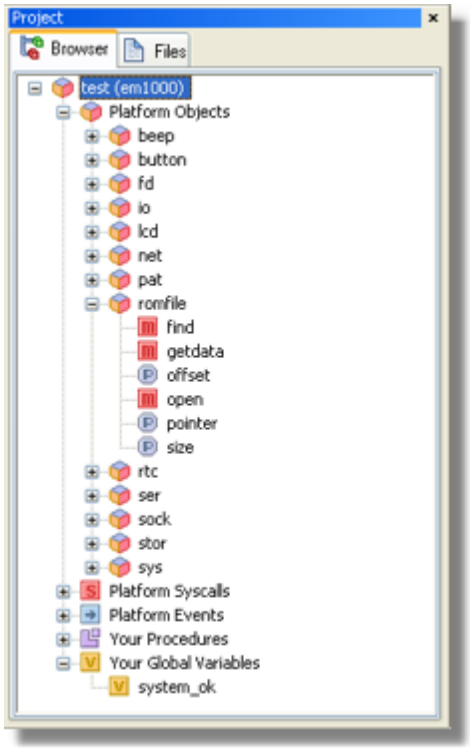
caused the error.

Project

3.2.7.3

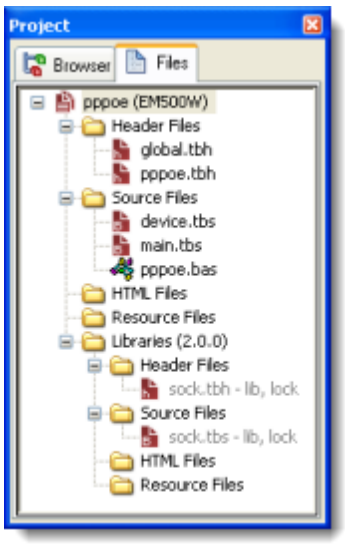
The Project pane contains two tabs: Browser[134] and Files[134].

Browser



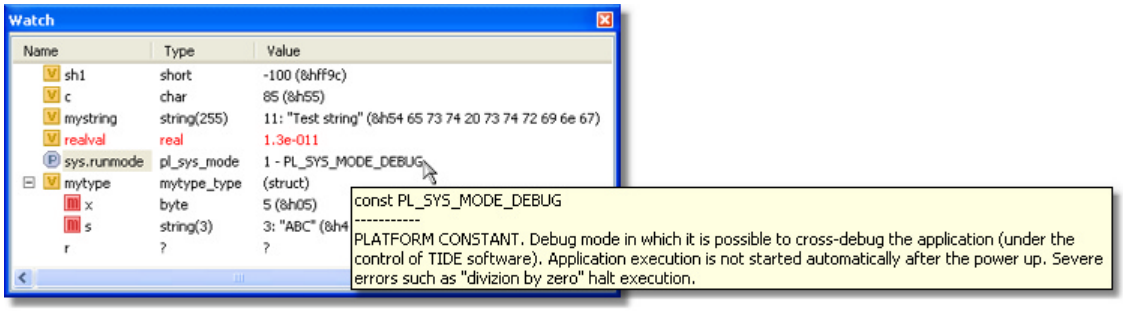
The Browser tab has been covered under Using the Project Browser[22] and under The Watch[33] above.

Files



The Files tab has been covered under Adding, Removing and Saving Files[18] above.

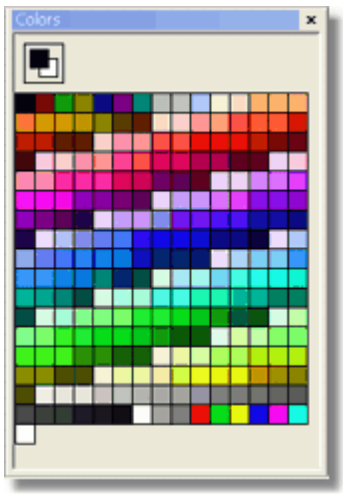
Watch 6.2.7.4



The Watch pane has been covered under The Watch 33 above.

Colors 6.2.7.5

Colors pane is displayed whenever a graphical resource is opened for editing in TIDE. Depending on the color mode selection you've made when adding an image file to the project, the pane will either show available palette colors...



... or RGB color selector:

Language Element Icons

Throughout TIDE, many icons are used for various Tibbo Basic constructs. Below is a complete listing:

- **Constants** (see Constants 65)
- **Enumeration Types** (see User-Defined Types 59)
- **System Calls** (see Function Reference 205)
- **Objects** (see Object Reference 231)
- **Properties** (see Object Reference 231)
- **Methods** (see Object Reference 231)
- **Event Handlers** (Implemented in current project -- see Object Reference 231). Grayed if no event handler exists for an event.
- **Procedures** (see Introduction to Procedures 66). Grayed if a procedure is not implemented (i.e. doesn't have a body).
- **Variables** (see Introduction to Variables, Constants and Scopes 47). Grayed if the variable is not defined.

Glossary of Terms

Below are several key definitions for terms used throughout the text.

Compilation Unit

A single file containing source code, to be processed by the compiler 136. Projects may contain many compilation units. Under Tibbo Basic, BASIC source files (.tbs) and HTML files (.html) are compilation units.

Compiler

For Tibbo Basic, a software program which takes compilation units and converts each of them, individually, to executable P-Code 137. The Tibbo Basic compiler is a single-pass compiler, which means it goes over each compilation unit from beginning to end, and does it just one time.

Construct

A meaningful combination of language elements, including keywords, identifiers, constants, etc. An example of a construct would be A = B + 5.

Cross-Debugging

This is the practice of using one device to observe and control the state of a program running on another device, in order to find bugs in it. Under Tibbo Basic, your computer displays various status messages and information about variables etc, but the actual code is executed on the target 137. The target state is periodically polled and displayed on your computer. Hence, cross-debugging.

Identifier

Any 'name' for a variable, a function, a subroutine, a constant, or any other 'thing' you may call or refer to within a program. In the statement x = 5, x is an identifier. An existing keyword 137 cannot be used as an identifier, since it already has a fixed meaning as part of Tibbo Basic syntax.

Keyword

A single word which carries a specific meaning within Tibbo Basic. Keywords are listed under Keywords 102 above.

Label

An identifier 137 marking the beginning of a block of code which will then be called using a Goto Statement 93. Labels are declared in code by writing their name, followed by a colon, on a single line.

Linker

A software program processing the output of the compiler 136, to look for any cross-references between the units. If compilation unit A calls a procedure which is in compilation unit B, the linker associates between the two, and provides compilation unit A with the proper memory addresses so that it could actually reach the procedure it needs in unit B.

P-Code

Pseudo-Code. This is code which is not executed directly by a processor, but by a 'virtual processor' (called a Virtual Machine) which is a part of TiOS that emulates a processor, interprets the P-Code and executes it.

The Tibbo Basic compiler produces P-Code.

Syscall

A *sys*tem *call*. This is an internal platform function -- not to be used explicitly. It is expressed as a numeric value. Syscalls are automatically invoked when you perform certain operations in code, such as variable type conversion.

You do not have to invoke syscalls directly within your code -- it is not recommended.

Target

The hardware device with which you are working. This is the device connected to the computer while debugging. The code you are writing actually runs on this device, and the debug messages originate from the device -- not from anywhere within your computer.

Virtual Machine

This a part of TiOS. In essence, it is a processor implemented in software. It executes the P-Code 137 of your application (produced by the compiler).

Using a Virtual Machine, we can achieve full control over it code execution. You can think of your application as if it runs in a designated 'sandbox' -- you can do

anything, but the Operating System will stay unharmed. So no code executed in the Virtual Machine can crash TiOS itself.

This approach also greatly enhances your control over your program execution during debugging.

Platforms

This section contains specifications for all platforms included into the current documentation.

Each platform supports a number of functions (syscalls) and objects. Actual functions and object description is not included into each platform's spec. Instead, they are documented in the Function Reference 205 and Object Reference 231 sections, while Platform Specifications 138 section only contains the lists of functions and objects supported. This is because most functions and objects are shared by different platforms.

Platform Specifications

The following platforms are included into this documentation:

Platform	Devices
EM500W 138	EM500 + GA1000
EM1000 143	EM1000, DS1000
EM1000W 143	EM1000 + GA1000
EM1202 151	EM1202, EM1202EV*, DS1202*
EM1202W 151	EM1202 + GA1000
DS1100 164	DS1100
DS1101W 168	DS1101 + GA1000
DS1102W 174	DS1102 + GA1000
DS1202 181	EM1202EV*, DS1202*
EM1206 158	EM1206
EM1206W 158	EM1206 + GA1000
DS1206 186	DS1206, DS1206N

** These devices can be used with the EM1202 or DS1202 platform. Notice, however, that the EM1202EV and DS1202 interconnect certain pins of the EM1202. See Programmable Hardware Manual for details.*

EM500W

Memory space

RAM	**EM500W: 17,920*** bytes for application variables and data
Flash	**327,680** bytes for application storage, data cannot be stored in this memory
EEPROM	**200** bytes for application data

** RAM available in the debug mode is smaller by 257 bytes. All memory is available in the release mode.*

Supported Objects, variable types, and functions

- Sock 421 — socket communications (up to 16 UDP, TCP, and HTTP sessions);
- Net 358 — controls the Ethernet port;
- Wln 536 — handles the Wi-Fi interface (requires GA1000 201 add-on module);
- Ser 378 — in charge of the serial port (UART, Wiegand, and clock/data modes);
- Io 294 — handles I/O lines, ports, and interrupts;
- Fd 236* — manages flash memory file system and direct sector access (requires an externally connected flash IC 143);
- Stor 522 — provides access to the EEPROM;
- Romfile 370 — facilitates access to resource files (fixed data);
- Pppoe 369 — provides access to the Internet over an ADSL modem;
- Ppp 366 — provides access to the Internet over a serial modem (GPRS, POTS, etc.);
- Pat 363 — "plays" patterns on a pair of LEDs;
- Button 234 — monitors the MD line (setup button 201);
- Sys 526 — in charge of general device functionality.

** Fully supported with the exception of fd.copyfirmware 269. Disabled by default -- enable it in Project Settings 38 -> Customize.*

This platform supports the standard set of variable types 192 and functions 192 (a.k.a. "syscalls") **except** aes128enc 206 and aes128dec 205. These AES128 functions are **not present** on the EM500W platform.

Platform-specific constants

You can find them here 140.

Miscellaneous information

Available network interfaces	**Ethernet (net.)**(1)
GPIO type 194	**Bidirectional**
RTS/CTS remapping 195	**Not supported**(2)
GA1000 lines remapping 201	**Not supported**(3)
Serial port FIFOs 196	**16 byte for TX, 16 bytes for RX**
Clock frequency (PLL) control 196	**Not supported, frequency is fixed at 80MHz**
Special configuration section of the EEPROM 197	**8 bytes for MAC storage**
Device serial number 199	**4 bytes, the number can't be changed**
Flash memory configuration 200	**Dedicated memory for firmware/**

	application **Data cannot be stored in this memory**
LEDs 200	**Green (SG) and red (SR) Status LED lines** **(their brightness indicates Ethernet link status)** **One additional dedicated link status LED line**
Debug communications 204	**Ethernet / UDP Broadcast transport** **Ethernet /WinPCap transport**

Comments:

1. The sock.allowedinterfaces 474 property refers to the Ethernet interface as "NET". Sock.targetinterface 506 and sock.currentinterface 478 properties rely on the pl_sock_interfaces 150 enum, whose members differ depending on the platform.
2. CTS is permanently mapped to 0- PL_INT_NUM_0 142 (0- PL_IO_NUM_0_INT0 141). RTS is permanently mapped to 2- PL_IO_NUM_2 141.
3. Connect the GA1000 add-on module as follows (also see schematic diagram C in Connecting GA1000 201):

CS	7- PL_IO_NUM_7 141
CLK	6- PL_IO_NUM_6 141
DI	5- PL_IO_NUM_5 141 (yes, DI and DO lines are on the same GPIO line)
DO	
RST	Choose any suitable GPIO or use NAND gates 201 to generate reset

Platform-specific Constants

The following constant lists are platform-specific:

- Enum pl_redir 140 - the list of constants that define buffer redirection (shorting) for this platform.
- Enum pl_io_num 141 - the list of constants that define available I/O lines.
- Enum pl_io_port_num 142 - the list of constants that define available 8-bit I/O ports.
- Enum pl_int_num 142 - the list of constants that define available *interrupt* lines.
- Enum pl_sock_interfaces 142 - the list of available network interfaces.

Enum pl_redir

Enum pl_redir contains the list of constants that define buffer redirection (shorting). The following objects support buffers and buffer redirection:

- Ser. 378 object (see ser.redir 414 method)
- Sock. 421 object (see sock.redir 493 method)

Enum pl_redir for this platform includes the following constants:

0- PL_REDIR_NONE: Cancels redirection for the serial port or socket.

1- PL_REDIR_SER: Redirects RX data of the serial port or socket to the TX

buffer of the serial port.

6- PL_REDIR_SOCK0: Redirects RX data of the serial port or socket to the TX buffer of socket 0. This constant can be used as a "base" for all other sockets i.e. in expressions like sock.redir= PL_REDIR_SOCK0+f.

7- PL_REDIR_SOCK1: Redirects RX data of the serial port or socket to the TX buffer of socket 1.

8- PL_REDIR_SOCK2: Redirects RX data of the serial port or socket to the TX buffer of socket 2.

9- PL_REDIR_SOCK3: Redirects RX data of the serial port or socket to the TX buffer of socket 3.

10- PL_REDIR_SOCK4: Redirects RX data of the serial port or socket to the TX buffer of socket 4.

11- PL_REDIR_SOCK5: Redirects RX data of the serial port or socket to the TX buffer of socket 5.

12- PL_REDIR_SOCK6: Redirects RX data of the serial port or socket to the TX buffer of socket 6.

13- PL_REDIR_SOCK7: Redirects RX data of the serial port or socket to the TX buffer of socket 7.

14- PL_REDIR_SOCK8: Redirects RX data of the serial port or socket to the TX buffer of socket 8.

15- PL_REDIR_SOCK9: Redirects RX data of the serial port or socket to the TX buffer of socket 9.

16- PL_REDIR_SOCK10: Redirects RX data of the serial port or socket to the TX buffer of socket 10.

17- PL_REDIR_SOCK11: Redirects RX data of the serial port or socket to the TX buffer of socket 11.

18- PL_REDIR_SOCK12: Redirects RX data of the serial port or socket to the TX buffer of socket 12.

19- PL_REDIR_SOCK13: Redirects RX data of the serial port or socket to the TX buffer of socket 13.

20- PL_REDIR_SOCK14: Redirects RX data of the serial port or socket to the TX buffer of socket 14.

21- PL_REDIR_SOCK15: Redirects RX data of the serial port or socket to the TX buffer of socket 15.

Enum pl_io_num

Enum pl_io_num contains the list of constants that refer to available GPIO lines. Use these constants when selecting the line with the io. 294 object (see the io.num 301 property).

Note that GPIO lines are of bi*directional* type 194 and do not require explicit configuration as outputs or inputs.

Enum pl_io_num includes the following constants:

0- PL_IO_NUM_0_INT0: General-purpose I/O line 0 (P0.0). This is also the interrupt line 0.

1- PL_IO_NUM_1_INT1: General-purpose I/O line 1 (P0.1). This is also the interrupt line 1.

2- PL_IO_NUM_2:	General-purpose I/O line 2 (P0.2).
3- PL_IO_NUM_3:	General-purpose I/O line 3 (P0.3).
4- PL_IO_NUM_4:	General-purpose I/O line 4 (P0.4).
5- PL_IO_NUM_5:	General-purpose I/O line 5 (P0.5).
6- PL_IO_NUM_6:	General-purpose I/O line 6 (P0.6).
7- PL_IO_NUM_7:	General-purpose I/O line 7 (P0.7).
8- PL_IO_NULL:	This is a NULL line that does not physically exist. The state of this line is always detected as LOW. Setting this line has no effect.

Enum pl_io_port_num

Enum pl_io_port_num contains the list of available 8-bit GPIO ports. Use these constants when selecting the port with the io. 294 object (see the io.portnum 302 property).

Note that GPIO lines are of b*idirectional* type 194 and do not require explicit configuration as outputs or inputs.

Enum pl_io_port_num includes the following constants:

0- PL_IO_PORT_NUM_0:	8-bit port 0 (P0). Contains I/O lines 0-7.

Enum pl_int_num

Enum pl_int_num contains the list of constants that refer to available *interrupt* lines. Interrupt lines are mapped to GPIO lines 141 (this mapping can't be altered).

Enum pl_int_num includes the following constants:

0- PL_INT_NUM_0:	Interrupt line 0 (mapped onto I/O line 0).
1- PL_INT_NUM_1:	Interrupt line 1 (mapped onto I/O line 1).
2- PL_INT_NULL:	This is a NULL interrupt line that does not physically exist.

Enum pl_sock_interfaces

Enum pl_sock_interfaces contains the list of network interfaces supported by the platform:

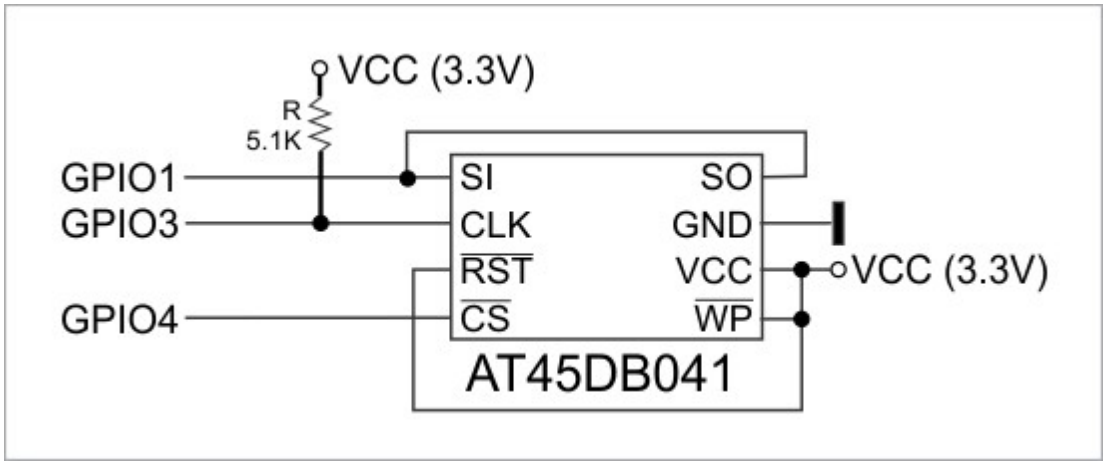
0- PL_SOCK_INTERFACE_NULL:	Null (empty) interface.
1- PL_SOCK_INTERFACE_NET **(default)**:	Ethernet interface 358.
2- PL_SOCK_INTERFACE_WLN:	Wi-Fi interface 536.
3- PL_SOCK_INTERFACE_PPP:	PPP interface 366.
4- PL_SOCK_INTERFACE_PPPOE:	PPPoE interface 369.

Connecting External Flash IC

The EM500W platform includes fd. 236 object (with the exception of the fd.copyfirmware 269 method). For this to work, an external flash IC must be connected to the EM500.

As shown on the schematic diagram below, this flash IC is ATMEL AT45DB041. Since the EM500 has a *dedicated* flash memory configuration 200, the flash IC will be used exclusively by the fd. object and provide 1MB of storage.

The 5.1K pull-up resistor is needed to "sharpen" SPI clock signal. EM500's bidirectional GPIOs 194 allow interconnecting SI and SO lines (this saves one GPIO line!).



For the fd. 236 object to work, it must be enabled first. Do this through Project Settings 38 -> Customize dialog.

- **DO NOT** enable fd. object unless you actually have the flash IC attached. When the fd. object is enabled, the EM500 will attempt to detect the flash IC presence. That is, the EM500 will try to access the flash through the SPI interface. This means that there will be signals on GPIO4 (CS), GPIO3 (CLK), and GPIO1 (DI/DO). Flash IC detection may interfere with the operation of your device (if you have something else connected to these GPIOs).

EM1000 and EM1000W Platforms

The difference between the EM1000 and EM1000W platforms is that the EM1000W additionally includes the Wi-Fi (wln. 536) object (requires an external GA1000 add-on module). All other features of these two platforms are exactly the same.

Memory space

RAM	**22,528*** bytes for application variables and data
Flash	**983,040** bytes for application and data storage (shared 200 flash memory)**
EEPROM	**2040** bytes for application data

**RAM available in the debug mode is smaller by 257 bytes. All memory is available*

in the release mode.

***Some earlier devices had only 458,752 bytes of flash memory available*

Supported Objects, variable types, and functions

- Sock 421 — socket communications (up to 16 UDP, TCP, and HTTP sessions);
- Net 358 — controls the Ethernet interface;
- Wln 536 — handles the Wi-Fi interface (only available on the **EM1000W platform**, requires GA1000 201 add-on module);
- Ser 378 — in charge of serial ports (UART, Wiegand, and clock/data modes);
- Ssi 512 — implements up to four serial synchronous interface (SSI) channels, supports SPI, I2C, clock/data, etc.;
- Io 294 — handles I/O lines, ports, and interrupts;
- Lcd 317 — controls graphical display panels (several types supported);
- Kp 304 — scans keypads of matrix and "binary" types;
- Rtc 375 — keeps track of date and time;
- Fd 236 — manages flash memory file system and direct sector access;
- Stor 522 — provides access to the EEPROM;
- Romfile 370 — facilitates access to resource files (fixed data);
- Pppoe 369 — provides access to the Internet over an ADSL modem;
- Ppp 366 — provides access to the Internet over a serial modem (GPRS, POTS, etc.);
- Pat 363 — "plays" patterns on up to five LED pairs;
- Beep 232 — generates buzzer patterns;
- Button 234 — monitors the MD line (setup button 201);
- Sys 526 — in charge of general device functionality.

These platforms support the standard set of variable types 192 and functions 192 (a.k.a. "syscalls").

Platform-specific constants

You can find them here 146.

Miscellaneous information

Available network interfaces	**EM1000: Ethernet (net.)** **EM1000W: Ethernet (net. 358), Wi-Fi (wln. 536)**(1)
GPIO type 194	**Unidirectional**
RTS/CTS remapping 195	**Supported**(2)
GA1000 lines remapping 201	**Supported**(3)
Serial port FIFOs 196	**16 byte for TX, 16 bytes for RX**

Clock frequency (PLL) control 196	**PLL on: 88.4736Mhz, PLL off: 11.0592Mhz**(4)
Special configuration section of the EEPROM 197	**8 bytes for MAC storage**
Device serial number 199	**128 bytes: 64 OTP bytes + 64 fixed bytes**(5)
Flash memory configuration 200	**Shared**
LEDs 200	**Green (SG) and red (SR) Status LED lines** **Green (EG) and yellow (EY) Ethernet LED lines**
Debug communications 204	**Ethernet / UDP Broadcast transport** **Ethernet /WinPCap transport**

Comments:

1. The sock.allowedinterfaces 474 property refers to the Ethernet interface as "NET", Wi-Fi -- as "WLN". Sock.targetinterface 506 and sock.currentinterface 478 properties rely on the pl_sock_interfaces 150 enum, whose members differ depending on the platform.
2. Default CTS/RTS mapping is different for each serial port:

	RTS	CTS
Port 1	0- PL_IO_NUM_0 147	0- PL_INT_NUM_0 147 (16- PL_IO_NUM_16_INT0 150)
Port 2	1- PL_IO_NUM_1 147	1- PL_INT_NUM_1 147 (17- PL_IO_NUM_17_INT1 150)
Port 3	2- PL_IO_NUM_2 147	2- PL_INT_NUM_2 147 (18- PL_IO_NUM_18_INT2 150)
Port 4	3- PL_IO_NUM_3 147	3- PL_INT_NUM_3 147 (19- PL_IO_NUM_19_INT3 150)

3. Mapping of GA1000 201 control lines is fully flexible on the EM1000W. However, if the GA1000 module is installed *on top* of the EM1000 (so called EM1000G module combination), then the following mapping *must* be applied:

CS	49- PL_IO_NUM_49 147
CLK	53- PL_IO_NUM_53 147
DI	52- PL_IO_NUM_52 147
DO	50- PL_IO_NUM_50 147
RST	51- PL_IO_NUM_51 147

4. Default PLL state after the external reset depends on the PM pin of the EM1000 (W).
5. Older EM1000 and EM1000W devices did not contain the serial number. To find out if your EM1000(W) has the serial number onboard, try to read this serial number with the sys.serialnum 534 R/O property. If this property returns an empty string, then the serial number is not present. Sys.serialnum returns all 128 bytes of the serial number. First 64 bytes are one-time-programmable (OTP) with the sys.setserialnum 535 method.

Platform-specific Constants

The following constant lists are platform-specific:

- Enum pl_redir 146 - the list of constants that define buffer redirection (shorting) for this platform.
- Enum pl_io_num 147 - the list of constants that define available I/O lines.
- Enum pl_io_port_num 149 - the list of constants that define available 8-bit I/O ports.
- Enum pl_int_num 150 - the list of constants that define available *interrupt* lines.
- Enum pl_sock_interfaces 150 - the list of available network interfaces.

Enum pl_redir

Enum pl_redir contains the list of constants that define buffer redirection (shorting). The following objects support buffers and buffer redirection:

- Ser. 378 object (see ser.redir 414 method)
- Sock. 421 object (see sock.redir 493 method)

Enum pl_redir for this platform includes the following constants:

0- PL_REDIR_NONE:	Cancels redirection for the serial port or socket.
1- PL_REDIR_SER:	Redirects RX data of the serial port or socket to the TX buffer of the serial port 0. This constant can be used as a "base" for all other serial ports, i.e. in expressions like ser.redir= PL_REDIR_SER+f.
1- PL_REDIR_SER0:	Redirects RX data of the serial port or socket to the TX buffer of the serial port 0.
2- PL_REDIR_SER1:	Redirects RX data of the serial port or socket to the TX buffer of the serial port 1.
3- PL_REDIR_SER2:	Redirects RX data of the serial port or socket to the TX buffer of the serial port 2.
4- PL_REDIR_SER3:	Redirects RX data of the serial port or socket to the TX buffer of the serial port 3.
6- PL_REDIR_SOCK0:	Redirects RX data of the serial port or socket to the TX buffer of socket 0. This constant can be used as a "base" for all other sockets i.e. in expressions like sock.redir= PL_REDIR_SOCK0+f.
7- PL_REDIR_SOCK1:	Redirects RX data of the serial port or socket to the TX buffer of socket 1.
8- PL_REDIR_SOCK2:	Redirects RX data of the serial port or socket to the TX buffer of socket 2.
9- PL_REDIR_SOCK3:	Redirects RX data of the serial port or socket to the TX buffer of socket 3.
10- PL_REDIR_SOCK4:	Redirects RX data of the serial port or socket to the TX buffer of socket 4.
11- PL_REDIR_SOCK5:	Redirects RX data of the serial port or socket to the TX buffer of socket 5.
12- PL_REDIR_SOCK6:	Redirects RX data of the serial port or socket to the TX buffer of socket 6.
13- PL_REDIR_SOCK7:	Redirects RX data of the serial port or socket to the TX

buffer of socket 7.

14- PL_REDIR_SOCK8: Redirects RX data of the serial port or socket to the TX buffer of socket 8.

15- PL_REDIR_SOCK9: Redirects RX data of the serial port or socket to the TX buffer of socket 9.

16- PL_REDIR_SOCK10: Redirects RX data of the serial port or socket to the TX buffer of socket 10.

17- PL_REDIR_SOCK11: Redirects RX data of the serial port or socket to the TX buffer of socket 11.

18- PL_REDIR_SOCK12: Redirects RX data of the serial port or socket to the TX buffer of socket 12.

19- PL_REDIR_SOCK13: Redirects RX data of the serial port or socket to the TX buffer of socket 13.

20- PL_REDIR_SOCK14: Redirects RX data of the serial port or socket to the TX buffer of socket 14.

21- PL_REDIR_SOCK15: Redirects RX data of the serial port or socket to the TX buffer of socket 15.

Enum pl_io_num

Enum pl_io_num contains the list of constants that refer to available GPIO lines. Use these constants when selecting the line with the io. 294 object (see the io.num 301 property).

Note that GPIO lines are of *unidirectional* type 194 and require explicit configuration as outputs or inputs. Some lines are configured as inputs or outputs automatically -- see the notes at the bottom of the page.

Enum pl_io_num includes the following constants:

0- PL_IO_NUM_0: General-purpose I/O line 0 (P0.0).

1- PL_IO_NUM_1: General-purpose I/O line 1 (P0.1).

2- PL_IO_NUM_2: General-purpose I/O line 2 (P0.2).

3- PL_IO_NUM_3: General-purpose I/O line 3 (P0.3).

4- PL_IO_NUM_4: General-purpose I/O line 4 (P0.4).

5- PL_IO_NUM_5: General-purpose I/O line 5 (P0.5).

6- PL_IO_NUM_6: General-purpose I/O line 6 (P0.6).

7- PL_IO_NUM_7: General-purpose I/O line 7 (P0.7).

8- PL_IO_NUM_8_RX0(1): General-purpose I/O line 8 (P1.0). This line is also the RX/W1in/din 379 input of the serial port 0.

9- PL_IO_NUM_9_TX0(2): General-purpose I/O line 9 (P1.1). This line is also the TX/W1out/dout 379 output of the serial port 0.

10- PL_IO_NUM_10_RX1(1): General-purpose I/O line 10 (P1.2). This line is also the RX/W0&1in/din input of the serial port 1.

11- PL_IO_NUM_11_TX1(2): General-purpose I/O line 11 (P1.3). This line is also the TX/W1out/dout output of the serial port 1.

12- PL_IO_NUM_12_RX2(1): General-purpose I/O line 12 (P1.4). This line is also the RX/W0&1in/din input of the serial port 2.

13- PL_IO_NUM_13_TX2(2): General-purpose I/O line 13 (P1.5). This line is also

	the TX/W1out/dout output of the serial port 2.
14- PL_IO_NUM_14_RX3[1]:	General-purpose I/O line 14 (P1.6). This line is also the RX/W0&1in/din input of the serial port 3.
15- PL_IO_NUM_15_TX3[2]:	General-purpose I/O line 15 (P1.7). This line is also the TX/W1out/dout output of the serial port 3.
16- PL_IO_NUM_16_INT0:	General-purpose I/O line 16 (P2.0). This is also the interrupt line 0.
17- PL_IO_NUM_17_INT1:	General-purpose I/O line 17 (P2.1). This is also the interrupt line 1.
18- PL_IO_NUM_18_INT2:	General-purpose I/O line 18 (P2.2). This is also the interrupt line 2.
19- PL_IO_NUM_19_INT3:	General-purpose I/O line 19 (P2.3). This is also the interrupt line 3.
20- PL_IO_NUM_20_INT4:	General-purpose I/O line 20 (P2.4). This is also the interrupt line 4.
21- PL_IO_NUM_21_INT5:	General-purpose I/O line 21 (P2.5). This is also the interrupt line 5.
22- PL_IO_NUM_22_INT6:	General-purpose I/O line 22 (P2.6). This is also the interrupt line 6.
23- PL_IO_NUM_23_INT7:	General-purpose I/O line 23 (P2.7). This is also the interrupt line 7.
24- PL_IO_NUM_24:	General-purpose I/O line 24 (P3.0).
25- PL_IO_NUM_25:	General-purpose I/O line 25 (P3.1).
26- PL_IO_NUM_26:	General-purpose I/O line 26 (P3.2).
27- PL_IO_NUM_27:	General-purpose I/O line 27 (P3.3).
28- PL_IO_NUM_28:	General-purpose I/O line 28 (P3.4).
29- PL_IO_NUM_29:	General-purpose I/O line 29 (P3.5).
30- PL_IO_NUM_30:	General-purpose I/O line 30 (P3.6).
31- PL_IO_NUM_31:	General-purpose I/O line 31 (P3.7).
32- PL_IO_NUM_32:	General-purpose I/O line 32 (P4.0).
33- PL_IO_NUM_33:	General-purpose I/O line 33 (P4.1).
34- PL_IO_NUM_34:	General-purpose I/O line 34 (P4.2).
35- PL_IO_NUM_35:	General-purpose I/O line 35 (P4.3).
36- PL_IO_NUM_36:	General-purpose I/O line 36 (P4.4).
37- PL_IO_NUM_37:	General-purpose I/O line 37 (P4.5).
38- PL_IO_NUM_38:	General-purpose I/O line 38 (P4.6).
39- PL_IO_NUM_39:	General-purpose I/O line 39 (P4.7).
40- PL_IO_NUM_40:	General-purpose I/O line 40 (does not belong to any 8-bit port).
41- PL_IO_NUM_41:	General-purpose I/O line 41 (does not belong to any 8-bit port).
42- PL_IO_NUM_42:	General-purpose I/O line 42 (does not belong to any 8-bit port).
43- PL_IO_NUM_43:	General-purpose I/O line 43 (does not belong to any 8-bit port).

44- PL_IO_NUM_44:	General-purpose I/O line 44 (does not belong to any 8-bit port).
45- PL_IO_NUM_45_CO(3):	General-purpose I/O line 45 (does not belong to any 8-bit port). This line is also used by the beep 232 object to generate square wave output that is primarily intended for driving beeper (buzzer).
46- PL_IO_NUM_46:	General-purpose I/O line 46 (does not belong to any 8-bit port).
47- PL_IO_NUM_47:	General-purpose I/O line 47 (does not belong to any 8-bit port).
48- PL_IO_NUM_48:	General-purpose I/O line 48 (does not belong to any 8-bit port).
49- PL_IO_NUM_49:	General-purpose I/O line 49 (does not belong to any 8-bit port).
50- PL_IO_NUM_50:	General-purpose I/O line 50 (does not belong to any 8-bit port).
51- PL_IO_NUM_51:	General-purpose I/O line 51 (does not belong to any 8-bit port).
52- PL_IO_NUM_52:	General-purpose I/O line 52 (does not belong to any 8-bit port).
53- PL_IO_NUM_53:	General-purpose I/O line 53 (does not belong to any 8-bit port).
54- PL_IO_NULL:	This is a NULL line that does not physically exist. The state of this line is always detected as LOW. Setting this line has no effect.

Notes:

1. When a serial port is in the UART 380 mode (ser.mode 409 = 0- PL_SER_MODE_UART) this line is automatically configured to be an input when this serial port is enabled (ser.enabled 405 = 1- YES) and returns to the previous input/output and high/low state when this serial port is closed (ser.enabled= 0- NO). When a serial port is in the Wiegand 383 or clock/data 386 mode (ser.mode= 1- PL_SER_MODE_WIEGAND or ser.mode= 2- PL_SER_MODE_CLOCKDATA), the line has to be configured as input by the application- this will not happen automatically.
2. When a serial port is in the UART mode (ser.mode= 0- PL_SER_MODE_UART) this line is automatically configured to be an output when the serial port is enabled (ser.enabled= 1- YES) and returns to the previous input/output and high/low state when the serial port is closed (ser.enabled= 0- NO). When a serial port is in the Wiegand or clock/data mode (ser.mode= 1- PL_SER_MODE_WIEGAND or ser.mode= 2- PL_SER_MODE_CLOCKDATA), the line has to be configured as output by the application- this will not happen automatically.
3. When the beeper pattern stars playing, this line is configured as output automatically. When the beeper pattern stops playing, the output returns to the input/output and high/low state that it had before the pattern started playing.

Enum pl_io_port_num

Enum pl_io_port_num contains the list of available 8-bit GPIO ports. Use these constants when selecting the port with the io. 294 object (see the io.portnum 302 property).

Note that GPIO lines are of *unidirectional* type[194] and require explicit configuration as outputs or inputs. Some lines are configured as inputs or outputs automatically -- see Enum pl_io_num[147] for details.

Enum pl_io_port_num includes the following constants:

0- PL_IO_PORT_NUM_0:	8-bit port 0 (P0). Contains I/O lines 0-7.
1- PL_IO_PORT_NUM_1:	8-bit port 1 (P1). Contains I/O lines 8-15.
2- PL_IO_PORT_NUM_2:	8-bit port 2 (P2). Contains I/O lines 16-23.
3- PL_IO_PORT_NUM_3:	8-bit port 3 (P3). Contains I/O lines 24-31.
4- PL_IO_PORT_NUM_4:	8-bit port 4 (P4). Contains I/O lines 32-39.

Enum pl_int_num

Enum pl_int_num contains the list of constants that refer to available *interrupt* lines. Interrupt lines are mapped to GPIO lines[147] (this mapping can't be altered). For example, interrupt line 0 corresponds to GPIO line 16, interrupt line 1- to GPIO line 17, and so on. Keep in mind that for an interrupt line to work you need to configure[194] the corresponding GPIO line as input.

Enum pl_int_num includes the following constants:

0- PL_INT_NUM_0:	Interrupt line 0 (mapped onto I/O line 16).
1- PL_INT_NUM_1:	Interrupt line 1 (mapped onto I/O line 17).
2- PL_INT_NUM_2:	Interrupt line 2 (mapped onto I/O line 18).
3- PL_INT_NUM_3:	Interrupt line 3 (mapped onto I/O line 19).
4- PL_INT_NUM_4:	Interrupt line 4 (mapped onto I/O line 20).
5- PL_INT_NUM_5:	Interrupt line 5 (mapped onto I/O line 21).
6- PL_INT_NUM_6:	Interrupt line 6 (mapped onto I/O line 22).
7- PL_INT_NUM_7:	Interrupt line 7 (mapped onto I/O line 23).
8- PL_INT_NULL:	This is a NULL interrupt line that does not physically exist.

Enum pl_sock_interfaces

Enum pl_sock_interfaces contains the list of available network interfaces. The EM1000 and EM1000W platforms differ in that the EM1000W has the W-Fi interface (requires an external GA1000 add-on module), while the EM1000 does not, and the pl_sock_interfaces reflects this.

EM1000 platform

0- PL_SOCK_INTERFACE_NULL:	Null (empty) interface.
1- PL_SOCK_INTERFACE_NET **(default)**:	Ethernet interface[358].
3- PL_SOCK_INTERFACE_PPP:	PPP interface[366].

4- PL_SOCK_INTERFACE_PPPOE: PPPoE interface 369.

EM1000W platform

0- PL_SOCK_INTERFACE_NULL: Null (empty) interface.
1- PL_SOCK_INTERFACE_NET **(default)**: Ethernet interface 358.
2- PL_SOCK_INTERFACE_WLN: Wi-Fi interface 536.
3- PL_SOCK_INTERFACE_PPP: PPP interface 366.
4- PL_SOCK_INTERFACE_PPPOE: PPPoE interface 369.

EM1202 and EM1202W Platforms

The difference between the EM1202 and EM202W platforms is that the EM1202W additionally includes the Wi-Fi (wln. 536) object (requires an external GA1000 add-on module). All other features of these two platforms are exactly the same.

Memory space

RAM	**22,528*** bytes for application variables and data
Flash	**983,040** bytes for application and data storage (shared 200 flash memory)**
EEPROM	**2040** bytes for application data

** RAM available in the debug mode is smaller by 257 bytes. All memory is available in the release mode.*

*** Some earlier devices had only 458,752 bytes of flash memory available*

Supported Objects, variable types, and functions

- Sock 421 — socket communications (up to 16 UDP, TCP, and HTTP sessions);
- Net 358 — controls the Ethernet port;
- Wln 536 — handles the Wi-Fi interface (only available on the **EM1202W platform**, requires GA1000 201 add-on module);
- Ser 378 — in charge of serial ports (UART, Wiegand, and clock/data modes);
- Ssi 512 — implements up to four serial synchronous interface (SSI) channels, supports SPI, I2C, clock/data, etc.;
- Io 294 — handles I/O lines, ports, and interrupts;
- Lcd 317 — controls graphical display panels (several types supported);
- Kp 304 — scans keypads of matrix and "binary" types;
- Fd 236 — manages flash memory file system and direct sector access;
- Stor 522 — provides access to the EEPROM;
- Romfile 370 — facilitates access to resource files (fixed data);
- Pppoe 369 — provides access to the Internet over an ADSL modem;

- Ppp 366 — provides access to the Internet over a serial modem (GPRS, POTS, etc.);
- Pat 363 — "plays" patterns on up to five LED pairs;
- Beep 232 — generates buzzer patterns;
- Button 234 — monitors the MD line (setup button 201);
- Sys 526 — in charge of general device functionality.

These platforms support the standard set of variable types 192 and functions 192 (a.k.a. "syscalls").

Platform-specific constants

You can find them here 153.

Miscellaneous information

Available network interfaces	**EM1202: Ethernet (net.)** **EM1202W: Ethernet (net. 358), Wi-Fi (wln. 536)**(1)
GPIO type 194	**Unidirectional**
RTS/CTS remapping 195	**Supported**(2)
GA1000 lines remapping 201	**Supported**
Serial port FIFOs 196	**16 byte for TX, 16 bytes for RX**
Clock frequency (PLL) control 196	**PLL on (def): 88.4736Mhz, PLL off: 11.0592Mhz**(3)
Special configuration section of the EEPROM 197	**8 bytes for MAC storage**
Device serial number 199	**128 bytes: 64 OTP bytes + 64 fixed bytes**(4)
Flash memory configuration 200	**Shared**
LEDs 200	**Green (SG) and red (SR) Status LED lines** **Green (EG) and yellow (EY) Ethernet LED lines**
Debug communications 204	**Ethernet / UDP Broadcast transport** **Ethernet /WinPCap transport**

Comments:

1. The sock.allowedinterfaces 474 property refers to the Ethernet interface as "NET", Wi-Fi -- as "WLN". Sock.targetinterface 506 and sock.currentinterface 478 properties rely on the pl_sock_interfaces 150 enum, whose members differ depending on the platform.
2. Default CTS/RTS mapping is different for each serial port:

	RTS	CTS

Port 1	0- PL_IO_NUM_0 154	0- PL_INT_NUM_0 157 (16- PL_IO_NUM_16_INT0 154)
Port 2	1- PL_IO_NUM_1 154	1- PL_INT_NUM_1 157 (17- PL_IO_NUM_17_INT1 154)
Port 3	2- PL_IO_NUM_2 154	2- PL_INT_NUM_2 157 (18- PL_IO_NUM_18_INT2 154)
Port 4	3- PL_IO_NUM_3 154	3- PL_INT_NUM_3 157 (19- PL_IO_NUM_19_INT3 154)

3. Default PLL state after the external reset is ON.
4. Older EM1202 and EM1202W devices did not contain the serial number. To find out if your EM1202(W) has the serial number onboard, try to read this serial number with the sys.serialnum 534 R/O property. If this property returns an empty string, then the serial number is not present. Sys.serialnum returns all 128 bytes of the serial number. First 64 bytes are one-time-programmable (OTP) with the sys.setserialnum 535 method.

Platform-specific Constants

The following constant lists are platform-specific:

- Enum pl_redir 146 - the list of constants that define buffer redirection (shorting) for this platform.
- Enum pl_io_num 147 - the list of constants that define available I/O lines.
- Enum pl_io_port_num 156 - the list of constants that define available 8-bit I/O ports.
- Enum pl_int_num 150 - the list of constants that define available *interrupt* lines.
- Enum pl_sock_interfaces 157 - the list of available network interfaces.

Enum pl_redir

Enum pl_redir contains the list of constants that define buffer redirection (shorting) for these platforms. The following objects support buffers and buffer redirection:

- Ser. 378 object (see ser.redir 414 method)
- Sock. 421 object (see sock.redir 493 method)

Enum pl_redir for this platform includes the following constants:

0- PL_REDIR_NONE:	Cancels redirection for the serial port or socket.
1- PL_REDIR_SER:	Redirects RX data of the serial port or socket to the TX buffer of the serial port 0. This constant can be used as a "base" for all other serial ports, i.e. in expressions like ser.redir= PL_REDIR_SER+f.
1- PL_REDIR_SER0:	Redirects RX data of the serial port or socket to the TX buffer of the serial port 0.
2- PL_REDIR_SER1:	Redirects RX data of the serial port or socket to the TX buffer of the serial port 1.
3- PL_REDIR_SER2:	Redirects RX data of the serial port or socket to the TX buffer of the serial port 2.
4- PL_REDIR_SER3:	Redirects RX data of the serial port or socket to the TX buffer of the serial port 3.

6- PL_REDIR_SOCK0:	Redirects RX data of the serial port or socket to the TX buffer of socket 0. This constant can be used as a "base" for all other sockets i.e. in expressions like sock.redir= PL_REDIR_SOCK0+f.
7- PL_REDIR_SOCK1:	Redirects RX data of the serial port or socket to the TX buffer of socket 1.
8- PL_REDIR_SOCK2:	Redirects RX data of the serial port or socket to the TX buffer of socket 2.
9- PL_REDIR_SOCK3:	Redirects RX data of the serial port or socket to the TX buffer of socket 3.
10- PL_REDIR_SOCK4:	Redirects RX data of the serial port or socket to the TX buffer of socket 4.
11- PL_REDIR_SOCK5:	Redirects RX data of the serial port or socket to the TX buffer of socket 5.
12- PL_REDIR_SOCK6:	Redirects RX data of the serial port or socket to the TX buffer of socket 6.
13- PL_REDIR_SOCK7:	Redirects RX data of the serial port or socket to the TX buffer of socket 7.
14- PL_REDIR_SOCK8:	Redirects RX data of the serial port or socket to the TX buffer of socket 8.
15- PL_REDIR_SOCK9:	Redirects RX data of the serial port or socket to the TX buffer of socket 9.
16- PL_REDIR_SOCK10:	Redirects RX data of the serial port or socket to the TX buffer of socket 10.
17- PL_REDIR_SOCK11:	Redirects RX data of the serial port or socket to the TX buffer of socket 11.
18- PL_REDIR_SOCK12:	Redirects RX data of the serial port or socket to the TX buffer of socket 12.
19- PL_REDIR_SOCK13:	Redirects RX data of the serial port or socket to the TX buffer of socket 13.
20- PL_REDIR_SOCK14:	Redirects RX data of the serial port or socket to the TX buffer of socket 14.
21- PL_REDIR_SOCK15:	Redirects RX data of the serial port or socket to the TX buffer of socket 15.

Enum pl_io_num

Enum pl_io_num contains the list of constants that refer to available GPIO lines. Use these constants when selecting the line with the io. 294 object (see the io.num 301 property).

Note that GPIO lines are of *unidirectional* type 194 and require explicit configuration as outputs or inputs. Some lines are configured as inputs or outputs automatically -- see the notes at the bottom of the page.

Enum pl_io_num includes the following constants:

0- PL_IO_NUM_0:	Selects general-purpose I/O line 0 (P0.0).
1- PL_IO_NUM_1:	Selects general-purpose I/O line 1 (P0.1).
2- PL_IO_NUM_2:	Selects general-purpose I/O line 2 (P0.2).
3- PL_IO_NUM_3:	Selects general-purpose I/O line 3 (P0.3).

4- PL_IO_NUM_4:	Selects general-purpose I/O line 4 (P0.4).
5- PL_IO_NUM_5:	Selects general-purpose I/O line 5 (P0.5).
6- PL_IO_NUM_6:	Selects general-purpose I/O line 6 (P0.6).
7- PL_IO_NUM_7:	Selects general-purpose I/O line 7 (P0.7).
8- PL_IO_NUM_8_RX0(1):	Selects general-purpose I/O line 8 (P1.0). This line is also the RX/W1in/din 379 input of the serial port 0.
9- PL_IO_NUM_9_TX0(2):	Selects general-purpose I/O line 9 (P1.1). This line is also the TX/W1out/dout 379 output of the serial port 0.
10- PL_IO_NUM_10_RX1(1):	Selects general-purpose I/O line 10 (P1.2). This line is also the RX/W0&1in/din input of the serial port 1.
11- PL_IO_NUM_11_TX1(2):	Selects general-purpose I/O line 11 (P1.3). This line is also the TX/W1out/dout output of the serial port 1.
12- PL_IO_NUM_12_RX2(1):	Selects general-purpose I/O line 12 (P1.4). This line is also the RX/W0&1in/din input of the serial port 2.
13- PL_IO_NUM_13_TX2(2):	Selects general-purpose I/O line 13 (P1.5). This line is also the TX/W1out/dout output of the serial port 2.
14- PL_IO_NUM_14_RX3(1):	Selects general-purpose I/O line 14 (P1.6). This line is also the RX/W0&1in/din input of the serial port 3.
15- PL_IO_NUM_15_TX3(2):	Selects general-purpose I/O line 15 (P1.7). This line is also the TX/W1out/dout output of the serial port 3.
16- PL_IO_NUM_16_INT0:	Selects general-purpose I/O line 16 (P2.0). This is also the interrupt line 0.
17- PL_IO_NUM_17_INT1:	Selects general-purpose I/O line 17 (P2.1). This is also the interrupt line 1.
18- PL_IO_NUM_18_INT2:	Selects general-purpose I/O line 18 (P2.2). This is also the interrupt line 2.
19- PL_IO_NUM_19_INT3:	Selects general-purpose I/O line 19 (P2.3). This is also the interrupt line 3.
20- PL_IO_NUM_20_INT4:	Selects general-purpose I/O line 20 (P2.4). This is also the interrupt line 4.
21- PL_IO_NUM_21_INT5:	Selects general-purpose I/O line 21 (P2.5). This is also the interrupt line 5.
22- PL_IO_NUM_22_INT6:	Selects general-purpose I/O line 22 (P2.6). This is also the interrupt line 6.
23- PL_IO_NUM_23_INT7:	Selects general-purpose I/O line 23 (P2.7). This is also the interrupt line 7.
24- PL_IO_NUM_24:	Selects general-purpose I/O line 24 (does not belong to any 8-bit port).
25- PL_IO_NUM_25:	Selects general-purpose I/O line 25 (does not belong to any 8-bit port).
26- PL_IO_NUM_26:	Selects general-purpose I/O line 26 (does not belong to any 8-bit port).
27- PL_IO_NUM_27:	Selects general-purpose I/O line 27 (does not belong to any 8-bit port).

28- PL_IO_NUM_28:	Selects general-purpose I/O line 28 (does not belong to any 8-bit port).
29- PL_IO_NUM_29_CO(3):	Selects general-purpose I/O line 29 (does not belong to any 8-bit port). This line is also used by the beep 232 object to generate square wave output that is primarily intended for driving beeper (buzzer).
30- PL_IO_NUM_30:	Selects general-purpose I/O line 30 (does not belong to any 8-bit port).
31- PL_IO_NUM_31:	Selects general-purpose I/O line 31 (does not belong to any 8-bit port).
32- PL_IO_NULL:	This is a NULL line that does not physically exist. The state of this line is always detected as LOW. Setting this line has no effect.

Notes:

1. When a serial port is in the UART 380 mode (ser.mode 409 = 0- PL_SER_MODE_UART) this line is automatically configured to be an input when this serial port is enabled (ser.enabled 405 = 1- YES) and returns to the previous input/output and high/low state when this serial port is closed (ser.enabled= 0- NO). When a serial port is in the Wiegand 383 or clock/data 386 mode (ser.mode= 1- PL_SER_MODE_WIEGAND or ser.mode= 2- PL_SER_MODE_CLOCKDATA), the line has to be configured as input by the application- this will not happen automatically.
2. When a serial port is in the UART mode (ser.mode= 0- PL_SER_MODE_UART) this line is automatically configured to be an output when the serial port is enabled (ser.enabled= 1- YES) and returns to the previous input/output and high/low state when the serial port is closed (ser.enabled= 0- NO). When a serial port is in the Wiegand or clock/data mode (ser.mode= 1- PL_SER_MODE_WIEGAND or ser.mode= 2- PL_SER_MODE_CLOCKDATA), the line has to be configured as output by the application- this will not happen automatically.
3. When the beeper pattern stars playing, this line is configured as output automatically. When the beeper pattern stops playing, the output returns to the input/output and high/low state that it had before the pattern started playing.

Enum pl_io_port_num

Enum pl_io_port_num contains the list of available 8-bit GPIO ports. Use these constants when selecting the port with the io. 294 object (see the io.portnum 302 property).

Note that GPIO lines are of *unidirectional* type 194 and require explicit configuration as outputs or inputs. Some lines are configured as inputs or outputs automatically -- see Enum pl_io_num 154 for details.

Enum pl_io_port_num includes the following constants:

0- PL_IO_PORT_NUM_0:	8-bit port 0 (P0). Contains I/O lines 0-7.
1- PL_IO_PORT_NUM_1:	8-bit port 1 (P1). Contains I/O lines 8-15.
2- PL_IO_PORT_NUM_2:	8-bit port 2 (P2). Contains I/O lines 16-23.

Enum pl_int_num

Enum pl_int_num contains the list of constants that refer to available *interrupt* lines. Interrupt lines are mapped to GPIO lines 154 (this mapping can't be altered). For example, interrupt line 0 corresponds to GPIO line 16, interrupt line 1- to GPIO line 17, and so on. Keep in mind that for an interrupt line to work you need to configure 194 the corresponding GPIO line as input.

Enum pl_int_num for this platform includes the following constants:

0- PL_INT_NUM_0:	Interrupt line 0 (mapped onto I/O line 16).
1- PL_INT_NUM_1:	Interrupt line 1 (mapped onto I/O line 17).
2- PL_INT_NUM_2:	Interrupt line 2 (mapped onto I/O line 18).
3- PL_INT_NUM_3:	Interrupt line 3 (mapped onto I/O line 19).
4- PL_INT_NUM_4:	Interrupt line 4 (mapped onto I/O line 20).
5- PL_INT_NUM_5:	Interrupt line 5 (mapped onto I/O line 21).
6- PL_INT_NUM_6:	Interrupt line 6 (mapped onto I/O line 22).
7- PL_INT_NUM_7:	Interrupt line 7 (mapped onto I/O line 23).
8- PL_INT_NULL:	This is a NULL interrupt line that does not physically exist.

Enum pl_sock_interfaces

Enum pl_sock_interfaces contains the list of available network interfaces. The EM1202 and EM1202W platforms differ in that the EM1202W has the W-Fi interface (requires an external GA1000 add-on module), while the EM1202 does not, and the pl_sock_interfaces reflects this.

EM1202 platform

0- PL_SOCK_INTERFACE_NULL:	Null (empty) interface.
1- PL_SOCK_INTERFACE_NET **(default)**:	Ethernet interface 358.
3- PL_SOCK_INTERFACE_PPP:	PPP interface 366.
4- PL_SOCK_INTERFACE_PPPOE:	PPPoE interface 369.

EM1202W platform

0- PL_SOCK_INTERFACE_NULL:	Null (empty) interface.
1- PL_SOCK_INTERFACE_NET **(default)**:	Ethernet interface 358.
2- PL_SOCK_INTERFACE_WLN:	Wi-Fi interface 536.
3- PL_SOCK_INTERFACE_PPP:	PPP interface 366.
4- PL_SOCK_INTERFACE_PPPOE:	PPPoE interface 369.

EM1206 and EM1206W Platforms

The difference between the EM1206 and EM1206W platforms is that the EM1206W additionally includes the Wi-Fi (wln. 536) object (requires an external GA1000 add-on module). All other features of these two platforms are exactly the same.

Memory space

RAM	**22,528*** bytes for application variables and data
Flash	**983,040** bytes for application and data storage (shared 200 flash memory)**
EEPROM	**2040** bytes for application data

** RAM available in the debug mode is smaller by 257 bytes. All memory is available in the release mode.*

***Some earlier devices had only 458,752 bytes of flash memory available*

Supported Objects, variable types, and functions

- Sock 421 — socket communications (up to 16 UDP, TCP, and HTTP sessions);
- Net 358 — controls the Ethernet port;
- Wln 536 — handles the Wi-Fi interface (only available on the **EM1206W platform**, requires GA1000 201 add-on module);
- Ser 378 — in charge of serial ports (UART, Wiegand, and clock/data modes);
- Ssi 512 — implements up to four serial synchronous interface (SSI) channels, supports SPI, I2C, clock/data, etc.;
- Io 294 — handles I/O lines, ports, and interrupts;
- Kp 304 — scans keypads of matrix and "binary" types;
- Rtc 375 — keeps track of date and time;
- Fd 236 — manages flash memory file system and direct sector access;
- Stor 522 — provides access to the EEPROM;
- Romfile 370 — facilitates access to resource files (fixed data);
- Pppoe 369 — provides access to the Internet over an ADSL modem;
- Ppp 366 — provides access to the Internet over a serial modem (GPRS, POTS, etc.);
- Pat 363 — "plays" patterns on up to five LED pairs;
- Beep 232 — generates buzzer patterns;
- Button 234 — monitors the MD line (setup button 201);
- Sys 526 — in charge of general device functionality.

These platforms support the standard set of variable types 192 and functions 192 (a.k.a. "syscalls").

Platform-specific constants

You can find them here 160.

Miscellaneous information

Available network interfaces	**EM1206: Ethernet (net.)** **EM1206W: Ethernet (net. 358), Wi-Fi (wln. 536)(1)**
GPIO type 194	**Unidirectional**
RTS/CTS remapping 195	**Supported(2)**
GA1000 lines remapping 201	**Supported(3)**
Serial port FIFOs 196	**16 byte for TX, 16 bytes for RX**
Clock frequency (PLL) control 196	**PLL on (def): 88.4736Mhz, PLL off: 11.0592Mhz(4)**
Special configuration section of the EEPROM 197	**8 bytes for MAC storage**
Device serial number 199	**128 bytes: 64 OTP bytes + 64 fixed bytes(5)**
Flash memory configuration 200	**Shared**
LEDs 200	**Green (SG) and red (SR) Status LED lines** **Green (EG) and yellow (EY) Ethernet LED lines**
Debug communications 204	**Ethernet / UDP Broadcast transport** **Ethernet /WinPCap transport**

1. The sock.allowedinterfaces 474 property refers to the Ethernet interface as "NET", Wi-Fi -- as "WLN". Sock.targetinterface 506 and sock.currentinterface 478 properties rely on the pl_sock_interfaces 150 enum, whose members differ depending on the platform.
2. Default CTS/RTS mapping is different for each serial port. Notice that CTS and RTS lines in default mapping interfere with each other. Do remember to select meaningful mapping for these lines!

	RTS	CTS
Port 1	0-PL_IO_NUM_0_RX0_INT0 161	0- PL_INT_NUM_0 163 (0-PL_IO_NUM_0_RX0_INT0 161)
Port 2	1-PL_IO_NUM_1_TX0_INT1 161	1- PL_INT_NUM_1 163 (1-PL_IO_NUM_1_TX0_INT1 161)
Port 3	2-PL_IO_NUM_2_RX1_INT2 161	2- PL_INT_NUM_2 163 (2-PL_IO_NUM_2_RX1_INT2 161)
Port 4	3-PL_IO_NUM_3_TX1_INT3 161	3- PL_INT_NUM_3 163 (3-PL_IO_NUM_3_TX1_INT3 161)

3. Mapping of GA1000 201 control lines is fully flexible on the EM1206W. However, if

the EM1206EV board is used, then the following mapping *must* be applied:

CS	15- PL_IO_NUM_15
CLK	14- PL_IO_NUM_14
DI	12- PL_IO_NUM_12
DO	13- PL_IO_NUM_13
RST	11- PL_IO_NUM_11

4. Default PLL state after the external reset is ON.
5. Older EM1206 and EM1206W devices did not contain the serial number. To find out if your EM1206(W) has the serial number onboard, try to read this serial number with the sys.serialnum R/O property. If this property returns an empty string, then the serial number is not present. Sys.serialnum returns all 128 bytes of the serial number. First 64 bytes are one-time-programmable (OTP) with the sys.setserialnum method.

Platform-specific Constants

The following constant lists are platform-specific:

- Enum pl_redir - the list of constants that define buffer redirection (shorting) for this platform.
- Enum pl_io_num - the list of constants that define available I/O lines.
- Enum pl_io_port_num - the list of constants that define available 8-bit I/O ports.
- Enum pl_int_num - the list of constants that define available *interrupt* lines.
- Enum pl_sock_interfaces - the list of available network interfaces.

Enum pl_redir

Enum pl_redir contains the list of constants that define buffer redirection (shorting) for these platforms. The following objects support buffers and buffer redirection:

- Ser. object (see ser.redir method)
- Sock. object (see sock.redir method)

Enum pl_redir for this platform includes the following constants:

0- PL_REDIR_NONE:	Cancels redirection for the serial port or socket.
1- PL_REDIR_SER:	Redirects RX data of the serial port or socket to the TX buffer of the serial port 0. This constant can be used as a "base" for all other serial ports, i.e. in expressions like ser.redir= PL_REDIR_SER+f.
1- PL_REDIR_SER0:	Redirects RX data of the serial port or socket to the TX buffer of the serial port 0.
2- PL_REDIR_SER1:	Redirects RX data of the serial port or socket to the TX buffer of the serial port 1.
3- PL_REDIR_SER2:	Redirects RX data of the serial port or socket to the TX buffer of the serial port 2.
4- PL_REDIR_SER3:	Redirects RX data of the serial port or socket to the TX buffer of the serial port 3.

6- PL_REDIR_SOCK0: Redirects RX data of the serial port or socket to the TX buffer of socket 0. This constant can be used as a "base" for all other sockets i.e. in expressions like sock.redir= PL_REDIR_SOCK0+f.

7- PL_REDIR_SOCK1: Redirects RX data of the serial port or socket to the TX buffer of socket 1.

8- PL_REDIR_SOCK2: Redirects RX data of the serial port or socket to the TX buffer of socket 2.

9- PL_REDIR_SOCK3: Redirects RX data of the serial port or socket to the TX buffer of socket 3.

10- PL_REDIR_SOCK4: Redirects RX data of the serial port or socket to the TX buffer of socket 4.

11- PL_REDIR_SOCK5: Redirects RX data of the serial port or socket to the TX buffer of socket 5.

12- PL_REDIR_SOCK6: Redirects RX data of the serial port or socket to the TX buffer of socket 6.

13- PL_REDIR_SOCK7: Redirects RX data of the serial port or socket to the TX buffer of socket 7.

14- PL_REDIR_SOCK8: Redirects RX data of the serial port or socket to the TX buffer of socket 8.

15- PL_REDIR_SOCK9: Redirects RX data of the serial port or socket to the TX buffer of socket 9.

16- PL_REDIR_SOCK10: Redirects RX data of the serial port or socket to the TX buffer of socket 10.

17- PL_REDIR_SOCK11: Redirects RX data of the serial port or socket to the TX buffer of socket 11.

18- PL_REDIR_SOCK12: Redirects RX data of the serial port or socket to the TX buffer of socket 12.

19- PL_REDIR_SOCK13: Redirects RX data of the serial port or socket to the TX buffer of socket 13.

20- PL_REDIR_SOCK14: Redirects RX data of the serial port or socket to the TX buffer of socket 14.

21- PL_REDIR_SOCK15: Redirects RX data of the serial port or socket to the TX buffer of socket 15.

Enum pl_io_num

Enum pl_io_num contains the list of constants that refer to available GPIO lines. Use these constants when selecting the line with the io. 294 object (see the io.num 301 property).

Note that GPIO lines are of *unidirectional* type 194 and require explicit configuration as outputs or inputs. Some lines are configured as inputs or outputs automatically -- see the notes at the bottom of the page.

Enum pl_io_num includes the following constants:

0- PL_IO_NUM_0_RX0_INT0(1): General-purpose I/O line 0 (P0.0). This line is also the RX/W1in/din 379 input of the serial port 0 and the interrupt line 0.

1- PL_IO_NUM_1_TX0_INT1(2): General-purpose I/O line 1 (P0.1). This line is also

	the TX/W1out/dout 379 output of the serial port 0 and the interrupt line 1.
2- PL_IO_NUM_2_RX1_INT2(1):	General-purpose I/O line 2 (P0.2). This line is also the RX/W0&1in/din input of the serial port 1 and the interrupt line 2.
3- PL_IO_NUM_3_TX1_INT3(2):	General-purpose I/O line 3 (P0.3). This line is also the TX/W1out/dout output of the serial port 1 and the interrupt line 3.
4- PL_IO_NUM_4_RX2_INT4(1):	General-purpose I/O line 4 (P0.4). This line is also the RX/W0&1in/din input of the serial port 2 and the interrupt line 4.
5- PL_IO_NUM_5_TX2_INT5(2):	General-purpose I/O line 5 (P0.5). This line is also the TX/W1out/dout output of the serial port 2 and the interrupt line 5.
6- PL_IO_NUM_6_RX3_INT6(1):	General-purpose I/O line 6 (P0.6). This line is also the RX/W0&1in/din input of the serial port 3 and the interrupt line 6.
7- PL_IO_NUM_7_TX3_INT7(2):	General-purpose I/O line 7 (P0.7). This line is also the TX/W1out/dout output of the serial port 3 and the interrupt line 7.
8- PL_IO_NUM_8:	General-purpose I/O line 8 (P1.0).
9- PL_IO_NUM_9:	General-purpose I/O line 9 (P1.1).
10- PL_IO_NUM_10:	General-purpose I/O line 10 (P1.2).
11- PL_IO_NUM_11:	General-purpose I/O line 11 (P1.3).
12- PL_IO_NUM_12:	General-purpose I/O line 12 (P1.4).
13- PL_IO_NUM_13:	General-purpose I/O line 13 (P1.5).
14- PL_IO_NUM_14:	General-purpose I/O line 14 (P1.6).
15- PL_IO_NUM_15:	General-purpose I/O line 15 (P1.7).
16- PL_IO_NUM_16_CO(3):	General-purpose I/O line 16 (does not belong to any 8-bit port). This line is also used by the beep 232 object to generate square wave output that is primarily intended for driving beeper (buzzer).
17- PL_IO_NULL:	This is a NULL line that does not physically exist. The state of this line is always detected as LOW. Setting this line has no effect.

Notes:

1. When a serial port is in the UART 380 mode (ser.mode 409 = 0- PL_SER_MODE_UART) this line is automatically configured to be an input when this serial port is enabled (ser.enabled 405 = 1- YES) and returns to the previous input/output and high/low state when this serial port is closed (ser.enabled= 0- NO). When a serial port is in the Wiegand 383 or clock/data 386 mode (ser.mode= 1- PL_SER_MODE_WIEGAND or ser.mode= 2- PL_SER_MODE_CLOCKDATA), the line has to be configured as input by the application- this will not happen automatically.
2. When a serial port is in the UART mode (ser.mode= 0- PL_SER_MODE_UART) this line is automatically configured to be an output when the serial port is enabled (ser.enabled= 1- YES) and returns to the previous input/output and high/low state when the serial port is closed (ser.enabled= 0- NO). When a serial port is in

the Wiegand or clock/data mode (ser.mode= 1- PL_SER_MODE_WIEGAND or ser.mode= 2- PL_SER_MODE_CLOCKDATA), the line has to be configured as output by the application- this will not happen automatically.

3. When the beeper pattern stars playing, this line is configured as output automatically. When the beeper pattern stops playing, the output returns to the input/output and high/low state that it had before the pattern started playing.

Enum pl_io_port_num

Enum pl_io_port_num contains the list of available 8-bit GPIO ports. Use these constants when selecting the port with the io. 294 object (see the io.portnum 302 property).

Note that GPIO lines are of *unidirectional* type 194 and require explicit configuration as outputs or inputs. Some lines are configured as inputs or outputs automatically -- see Enum pl_io_num 161 for details.

Enum pl_io_port_num includes the following constants:

0- PL_IO_PORT_NUM_0: 8-bit port 0 (P0). Contains I/O lines 0-7.

1- PL_IO_PORT_NUM_1: 8-bit port 1 (P1). Contains I/O lines 8-15.

Enum pl_int_num

Enum pl_int_num contains the list of constants that refer to available *interrupt* lines. Interrupt lines are mapped to GPIO lines 161 (this mapping can't be altered). Keep in mind that for an interrupt line to work you need to configure 194 the corresponding GPIO line as input.

Enum pl_int_num for this platform includes the following constants:

0- PL_INT_NUM_0: Interrupt line 0 (mapped onto I/O line 0).

1- PL_INT_NUM_1: Interrupt line 1 (mapped onto I/O line 1).

2- PL_INT_NUM_2: Interrupt line 2 (mapped onto I/O line 2).

3- PL_INT_NUM_3: Interrupt line 3 (mapped onto I/O line 3).

4- PL_INT_NUM_4: Interrupt line 4 (mapped onto I/O line 4).

5- PL_INT_NUM_5: Interrupt line 5 (mapped onto I/O line 5).

6- PL_INT_NUM_6: Interrupt line 6 (mapped onto I/O line 6).

7- PL_INT_NUM_7: Interrupt line 7 (mapped onto I/O line 7).

8- PL_INT_NULL: This is a NULL interrupt line that does not physically exist.

Enum pl_sock_interfaces

Enum pl_sock_interfaces contains the list of available network interfaces. The EM1206 and EM1206W platforms differ in that the EM1206W has the W-Fi interface (requires an external GA1000 add-on module), while the EM1206 does not, and the pl_sock_interfaces reflects this.

EM1206 platform

0- PL_SOCK_INTERFACE_NULL:	Null (empty) interface.
1- PL_SOCK_INTERFACE_NET **(default)**:	Ethernet interface 358.
3- PL_SOCK_INTERFACE_PPP:	PPP interface 366.
4- PL_SOCK_INTERFACE_PPPOE:	PPPoE interface 369.

EM1206W platform

0- PL_SOCK_INTERFACE_NULL:	Null (empty) interface.
1- PL_SOCK_INTERFACE_NET **(default)**:	Ethernet interface 358.
2- PL_SOCK_INTERFACE_WLN:	Wi-Fi interface 536.
3- PL_SOCK_INTERFACE_PPP:	PPP interface 366.
4- PL_SOCK_INTERFACE_PPPOE:	PPPoE interface 369.

DS1100 Platform

Memory space

RAM	**17,920*** bytes for application variables and data
Flash	**327,680** bytes for application storage, data cannot be stored in this memory
EEPROM	**200** bytes for application data

** RAM available in the debug mode is smaller by 257 bytes. All memory is available in the release mode.*

Supported Objects, variable types, and functions

- Sock 421 — socket communications (up to 16 UDP, TCP, and HTTP sessions);
- Net 358 — controls the Ethernet port;
- Ser 378 — in charge of the RS232 port;
- Io 294 — handles I/O lines, ports, and interrupts;
- Stor 522 — provides access to the EEPROM;
- Romfile 370 — facilitates access to resource files (fixed data);
- Pppoe 369 — provides access to the Internet over an ADSL modem;
- Ppp 366 — provides access to the Internet over a serial modem (GPRS, POTS, etc.);
- Pat 363 — "plays" patterns on a pair of LEDs;
- Button 234 — monitors the MD line (setup button 201);
- Sys 526 — in charge of general device functionality.

** Fully supported with the exception of fd.copyfirmware 269. Disabled by default --*

enable it in Project Settings 38 -> Customize.

This platform supports the standard set of variable types 192 and functions 192 (a.k.a. "syscalls") **except** aes128enc 206 and aes128dec 205. These AES128 functions are **not present** on the DS1100 platform.

Platform-specific constants

You can find them here 165.

Miscellaneous information

Available network interfaces	**Ethernet (net. 358)**(1)
GPIO type 194	**Bidirectional**
RTS/CTS remapping 195	**Not supported**(2)
Serial port FIFOs 196	**16 byte for TX, 16 bytes for RX**
Clock frequency (PLL) control 196	**Not supported, frequency is fixed at 80MHz**
Special configuration section of the EEPROM 197	**8 bytes for MAC storage**
Device serial number 199	**4 bytes, the number can't be changed**
Flash memory configuration 200	**Dedicated memory for firmware/ application** **Data cannot be stored in this memory**
LEDs 200	**Green (SG) and red (SR) Status LEDs** **A dedicated Ethernet link status LED (yellow)**
Debug communications 204	**Ethernet / UDP Broadcast transport** **Ethernet /WinPCap transport**

Comments:

1. The sock.allowedinterfaces 474 property refers to the Ethernet interface as "NET". Sock.targetinterface 506 and sock.currentinterface 478 properties rely on the pl_sock_interfaces 150 enum, whose members differ depending on the platform.
2. CTS is permanently mapped to 0- PL_INT_NUM_0 167 (0- PL_IO_NUM_0_INT0 167). RTS is permanently mapped to 2- PL_IO_NUM_2 167.

Platform-specific Constants

The following constant lists are platform-specific:

- Enum pl_redir 166 - the list of constants that define buffer redirection (shorting) for this platform.
- Enum pl_io_num 167 - the list of constants that define available I/O lines.
- Enum pl_io_port_num 167 - the list of constants that define available 8-bit I/O ports.

- Enum pl_int_num 167 - the list of constants that define available *interrupt* lines.
- Enum pl_sock_interfaces 168 - the list of available network interfaces.

Enum pl_redir

Enum pl_redir contains the list of constants that define buffer redirection (shorting). The following objects support buffers and buffer redirection:

- Ser. 378 object (see ser.redir 414 method)
- Sock. 421 object (see sock.redir 493 method)

Enum pl_redir for this platform includes the following constants:

0- PL_REDIR_NONE:	Cancels redirection for the serial port or socket.
1- PL_REDIR_SER:	Redirects RX data of the serial port or socket to the TX buffer of the serial port.
6- PL_REDIR_SOCK0:	Redirects RX data of the serial port or socket to the TX buffer of socket 0. This constant can be used as a "base" for all other sockets i.e. in expressions like sock.redir= PL_REDIR_SOCK0+f.
7- PL_REDIR_SOCK1:	Redirects RX data of the serial port or socket to the TX buffer of socket 1.
8- PL_REDIR_SOCK2:	Redirects RX data of the serial port or socket to the TX buffer of socket 2.
9- PL_REDIR_SOCK3:	Redirects RX data of the serial port or socket to the TX buffer of socket 3.
10- PL_REDIR_SOCK4:	Redirects RX data of the serial port or socket to the TX buffer of socket 4.
11- PL_REDIR_SOCK5:	Redirects RX data of the serial port or socket to the TX buffer of socket 5.
12- PL_REDIR_SOCK6:	Redirects RX data of the serial port or socket to the TX buffer of socket 6.
13- PL_REDIR_SOCK7:	Redirects RX data of the serial port or socket to the TX buffer of socket 7.
14- PL_REDIR_SOCK8:	Redirects RX data of the serial port or socket to the TX buffer of socket 8.
15- PL_REDIR_SOCK9:	Redirects RX data of the serial port or socket to the TX buffer of socket 9.
16- PL_REDIR_SOCK10:	Redirects RX data of the serial port or socket to the TX buffer of socket 10.
17- PL_REDIR_SOCK11:	Redirects RX data of the serial port or socket to the TX buffer of socket 11.
18- PL_REDIR_SOCK12:	Redirects RX data of the serial port or socket to the TX buffer of socket 12.
19- PL_REDIR_SOCK13:	Redirects RX data of the serial port or socket to the TX buffer of socket 13.
20- PL_REDIR_SOCK14:	Redirects RX data of the serial port or socket to the TX buffer of socket 14.
21- PL_REDIR_SOCK15:	Redirects RX data of the serial port or socket to the TX buffer of socket 15.

Enum pl_io_num

Enum pl_io_num contains the list of constants that refer to available GPIO lines. Use these constants when selecting the line with the io. 294 object (see the io.num 301 property).

Note that GPIO lines are of bi*directional* type 194 and do not require explicit configuration as outputs or inputs.

Enum pl_io_num includes the following constants:

0- PL_IO_NUM_0_CTS_INT0:	General-purpose I/O line 0 (P0.0), also the CTS input of the serial port and the interrupt line 0.
1- PL_IO_NUM_1_DSR_INT1:	General-purpose I/O line 1 (P0.1), also the DSR input of the serial port and the interrupt line 1.
2- PL_IO_NUM_2_RTS:	General-purpose I/O line 2 (P0.2), also the RTS output of the serial port.
3- PL_IO_NUM_3_DTR:	General-purpose I/O line 3 (P0.3), also the DTR output of the serial port.
4- PL_IO_NUM_4_DCD:	General-purpose I/O line 4 (P0.4), also the DCD input of the serial port.
5- PL_IO_NUM_5:	Not implemented.
6- PL_IO_NUM_6:	Not implemented.
7- PL_IO_NUM_7:	Not implemented.
8- PL_IO_NULL:	This is a NULL line that does not physically exist. The state of this line is always detected as LOW. Setting this line has no effect.

Enum pl_io_port_num

Enum pl_io_port_num contains the list of available 8-bit GPIO ports. Use these constants when selecting the port with the io. 294 object (see the io.portnum 302 property).

Note that GPIO lines are of b*idirectional* type 194 and do not require explicit configuration as outputs or inputs.

Enum pl_io_port_num includes the following constants:

0- PL_IO_PORT_NUM_0:	8-bit port 0 (P0). Contains I/O lines 0-7. Lines 5-7 are not implemented on this platform.

Enum pl_int_num

Enum pl_int_num contains the list of constants that refer to available *interrupt* lines. Interrupt lines are mapped to GPIO lines 167 (this mapping can't be altered).

Enum pl_int_num includes the following constants:

0- PL_INT_NUM_0:	Interrupt line 0 (mapped to I/O line 0, which serves as the CTS input of the serial port).
1- PL_INT_NUM_1:	Interrupt line 1 (mapped to I/O line 1, which serves as

	the DSR input of the serial port).
2- PL_INT_NULL:	This is a NULL interrupt line that does not physically exist.

Enum pl_sock_interfaces

Enum pl_sock_interfaces contains the list of network interfaces supported by the platform:

0- PL_SOCK_INTERFACE_NULL:	Null (empty) interface.
1- PL_SOCK_INTERFACE_NET **(default)**:	Ethernet interface 358.
3- PL_SOCK_INTERFACE_PPP:	PPP interface 366.
4- PL_SOCK_INTERFACE_PPPOE:	PPPoE interface 369.

DS1101W Platform

- **Memory space**

RAM	**22,528*** bytes for application variables and data
Flash	**983,040** bytes for application and data storage (shared 200 flash memory)**
EEPROM	**2040** bytes for application data

** RAM available in the debug mode is smaller by 257 bytes. All memory is available in the release mode.*

*** Some earlier devices had only 458,752 bytes of flash memory available*

Supported Objects, variable types, and functions

- Sock 421 — socket communications (up to 16 UDP, TCP, and HTTP sessions);
- Net 358 — controls the Ethernet port;
- Wln 536 — handles the Wi-Fi interface (requires GA1000 201 add-on module, i.e. "G" device option);
- Ser 378 — in charge of the serial ports (on this device, serial channels 205 of the RS232 port);
- Io 294 — handles I/O lines, ports, and interrupts;
- Lcd 317 — controls the 96x32 monochrome OLED display (therefore, requires the "D" device option);
- Fd 236 — manages flash memory file system and direct sector access;
- Stor 522 — provides access to the EEPROM;
- Romfile 370 — facilitates access to resource files (fixed data);
- Pppoe 369 — provides access to the Internet over an ADSL modem;

- Ppp 366 — provides access to the Internet over a serial modem (GPRS, POTS, etc.);
- Pat 363 — "plays" patterns on a pair of LEDs;
- Beep 232 — generates buzzer patterns;
- Button 234 — monitors the MD line (setup button 201);
- Sys 526 — in charge of general device functionality.

As all displays of this type, the DS1101's OLED display has a limited life span. There will be a decrease in the display brightness after ~10000 hours of operation. To prolong display life, use the lcd.lock 348 method to turn the display off whenever possible. Display image is preserved when the display is "locked".

These platforms support the standard set of variable types 192 and functions 192 (a.k.a. "syscalls").

Platform-specific constants

You can find them here 170.

Miscellaneous information

Available network interfaces	**Ethernet (net. 358), Wi-Fi (wln. 536)**(1)
GPIO type 194	**Unidirectional**
RTS/CTS remapping 195	**Supported**(2)
GA1000 lines remapping 201	**Supported**(3)
Serial port FIFOs 196	**16 byte for TX, 16 bytes for RX**
Clock frequency (PLL) control 196	**PLL on (def): 88.4736Mhz, PLL off: 11.0592Mhz**(4)
Special configuration section of the EEPROM 197	**8 bytes for MAC storage**
Device serial number 199	**128 bytes: 64 OTP bytes + 64 fixed bytes**
Flash memory configuration 200	**Shared**
LEDs 200	**Green (SG) and red (SR) Status LEDs** **A dedicated Ethernet link status LED (yellow)** **An LED bar consisting of five blue LEDs**(5)
Debug communications 204	**Ethernet / UDP Broadcast transport** **Ethernet /WinPCap transport**

Comments:

1. The sock.allowedinterfaces 474 property refers to the Ethernet interface as "NET", Wi-Fi -- as "WLN". Sock.targetinterface 506 and sock.currentinterface 478

properties rely on the pl_sock_interfaces 174 enum, whose members differ depending on the platform.

2. The hardware of this platform supports serial channels 205, which means that CTS/RTS and DTR/DSR pairs of the DB9 connector can also be used as RX/TX pairs of additional serial channels. Therefore, it is up to you to decide which lines are used as RTS and CTS, and which channels they will belong to. Proper mapping will be required to implement each particular arrangement.
3. Although the platform itself supports remapping, actual "wires" connecting the system to the GA1000 are fixed and your mapping should reflect this:

CS	15- PL_IO_NUM_15 171
CLK	14- PL_IO_NUM_14 171
DI	12- PL_IO_NUM_12 171
DO	13- PL_IO_NUM_13 171
RST	11- PL_IO_NUM_11 171

4. Default PLL state after the external reset is ON.
5. The LEDs of the LED bar are mapped to GPIO lines 171. The LEDs are primarily intended for the Wi-Fi signal strength indication but can also be used for other purposes.

Platform-specific Constants

The following constant lists are platform-specific:

- Enum pl_redir 170 - the list of constants that define buffer redirection (shorting) for this platform.
- Enum pl_io_num 171 - the list of constants that define available I/O lines.
- Enum pl_io_port_num 173 - the list of constants that define available 8-bit I/O ports.
- Enum pl_int_num 174 - the list of constants that define available *interrupt* lines.
- Enum pl_sock_interfaces 174 - the list of available network interfaces.

Enum pl_redir

Enum pl_redir contains the list of constants that define buffer redirection (shorting) for these platforms. The following objects support buffers and buffer redirection:

- Ser. 378 object (see ser.redir 414 method)
- Sock. 421 object (see sock.redir 493 method)

Enum pl_redir for this platform includes the following constants:

0- PL_REDIR_NONE:	Cancels redirection for the serial port* or socket.
1- PL_REDIR_SER:	Redirects RX data of the serial port or socket to the TX buffer of the serial port 0. This constant can be used as a "base" for all other serial ports, i.e. in expressions like ser.redir= PL_REDIR_SER+f.
1- PL_REDIR_SER0:	Redirects RX data of the serial port or socket to the TX buffer of the serial port 0.
2- PL_REDIR_SER1:	Redirects RX data of the serial port or socket to the TX

	buffer of the serial port 1.
3- PL_REDIR_SER2:	Redirects RX data of the serial port or socket to the TX buffer of the serial port 2.
4- PL_REDIR_SER3:	Redirects RX data of the serial port or socket to the TX buffer of the serial port 3.
6- PL_REDIR_SOCK0:	Redirects RX data of the serial port or socket to the TX buffer of socket 0. This constant can be used as a "base" for all other sockets i.e. in expressions like sock.redir= PL_REDIR_SOCK0+f.
7- PL_REDIR_SOCK1:	Redirects RX data of the serial port or socket to the TX buffer of socket 1.
8- PL_REDIR_SOCK2:	Redirects RX data of the serial port or socket to the TX buffer of socket 2.
9- PL_REDIR_SOCK3:	Redirects RX data of the serial port or socket to the TX buffer of socket 3.
10- PL_REDIR_SOCK4:	Redirects RX data of the serial port or socket to the TX buffer of socket 4.
11- PL_REDIR_SOCK5:	Redirects RX data of the serial port or socket to the TX buffer of socket 5.
12- PL_REDIR_SOCK6:	Redirects RX data of the serial port or socket to the TX buffer of socket 6.
13- PL_REDIR_SOCK7:	Redirects RX data of the serial port or socket to the TX buffer of socket 7.
14- PL_REDIR_SOCK8:	Redirects RX data of the serial port or socket to the TX buffer of socket 8.
15- PL_REDIR_SOCK9:	Redirects RX data of the serial port or socket to the TX buffer of socket 9.
16- PL_REDIR_SOCK10:	Redirects RX data of the serial port or socket to the TX buffer of socket 10.
17- PL_REDIR_SOCK11:	Redirects RX data of the serial port or socket to the TX buffer of socket 11.
18- PL_REDIR_SOCK12:	Redirects RX data of the serial port or socket to the TX buffer of socket 12.
19- PL_REDIR_SOCK13:	Redirects RX data of the serial port or socket to the TX buffer of socket 13.
20- PL_REDIR_SOCK14:	Redirects RX data of the serial port or socket to the TX buffer of socket 14.
21- PL_REDIR_SOCK15:	Redirects RX data of the serial port or socket to the TX buffer of socket 15.

**In this list, the term "serial port" actually refers to "serial channel".*

Enum pl_io_num

Enum pl_io_num contains the list of constants that refer to available GPIO lines. Use these constants when selecting the line with the io. 294 object (see the io.num 301 property).

Note that GPIO lines are of *unidirectional* type 194 and require explicit configuration as outputs or inputs. Some lines are configured as inputs or outputs automatically

-- see the notes at the bottom of the page.

- The DS1101 device has the RS232 transceiver IC onboard. This IC "dictates" the GPIO line direction for certain lines. Do not try to use GPIO lines 0, 2, 4, and 6 as outputs -- this can permanently damage the hardware.

Enum pl_io_num includes the following constants:

0- PL_IO_NUM_0_RX0_INT0(1):	General-purpose I/O line 0 (P0.0). This line is also the RX/W1in/din input of the serial port(3) 0 and the interrupt line 0.
1- PL_IO_NUM_1_TX0_INT1(2):	General-purpose I/O line 1 (P0.1). This line is also the TX/W1out/dout output of the serial port 0 and the interrupt line 1.
2- PL_IO_NUM_2_RX1_INT2(1):	General-purpose I/O line 2 (P0.2). This line is also the RX/W0&1in/din input of the serial port 1 and the interrupt line 2.
3- PL_IO_NUM_3_TX1_INT3(2):	General-purpose I/O line 3 (P0.3). This line is also the TX/W1out/dout output of the serial port 1 and the interrupt line 3.
4- PL_IO_NUM_4_RX2_INT4(1):	General-purpose I/O line 4 (P0.4). This line is also the RX/W0&1in/din input of the serial port 2 and the interrupt line 4.
5- PL_IO_NUM_5_TX2_INT5(2):	General-purpose I/O line 5 (P0.5). This line is also the TX/W1out/dout output of the serial port 2 and the interrupt line 5.
6- PL_IO_NUM_6_RX3_INT6(1):	General-purpose I/O line 6 (P0.6). This line is also the RX/W0&1in/din input of the serial port 3 and the interrupt line 6.
7- PL_IO_NUM_7_EMPTY:	Not implemented.
8- PL_IO_NUM_8_PWROUT:	Controls the power output on pin 9 of the DB9M connector. Power will be ON when this output is enabled (io.enabled= 1- YES) and set to LOW (io.state= 0- LOW).
PL_IO_NUM_9:	Pin 8 on the wireless interface connector.
PL_IO_NUM_10:	Pin 6 on the wireless interface connector.
PL_IO_NUM_11:	The RST line of the wireless interface.
PL_IO_NUM_12:	The DI line of the wireless interface.
PL_IO_NUM_13:	The DO line of the wireless interface.
PL_IO_NUM_14:	The CLK line of the wireless interface.
PL_IO_NUM_15:	The CS line of the wireless interface.
PL_IO_NUM_16:	Pin 9 on the wireless interface connector.
PL_IO_NUM_17_EMPTY:	Not implemented.
PL_IO_NUM_18_EMPTY:	Not implemented.
PL_IO_NUM_19_SB1:	LED bar, LED#1 (the "weakest signal" LED).
PL_IO_NUM_20_SB2:	LED bar, LED#2.

PL_IO_NUM_21_SB3:	LED bar, LED#3.
PL_IO_NUM_22_SB4:	LED bar, LED#4.
PL_IO_NUM_23_SB5:	LED bar, LED#5 (the "strongest signal" LED).
PL_IO_NUM_24_OLED_D0:	OLED dispay data bus, line 0 (P0.0).
PL_IO_NUM_25_OLED_D1:	OLED dispay data bus, line 1 (P0.0).
PL_IO_NUM_26_OLED_D2:	OLED dispay data bus, line 2 (P0.1).
PL_IO_NUM_27_OLED_D3:	OLED dispay data bus, line 3 (P0.2).
PL_IO_NUM_28_OLED_D4:	OLED dispay data bus, line 4 (P0.3).
PL_IO_NUM_29_OLED_D5:	OLED dispay data bus, line 5 (P0.4).
PL_IO_NUM_30_OLED_D6:	OLED dispay data bus, line 6 (P0.5).
PL_IO_NUM_31_OLED_D7:	OLED dispay data bus, line 7 (P0.6).
PL_IO_NUM_32_OLED_CS:	OLED display, CS control line.
PL_IO_NUM_33_OLED_RD:	OLED display, RD control line.
PL_IO_NUM_34_OLED_WR:	OLED display, WR control line.
PL_IO_NUM_35_OLED_DC:	OLED display, DC control line.
PL_IO_NUM_36_OLED_RST:	OLED display, RST control line.
PL_IO_NUM_37_CO:	General-purpose I/O line 37 (does not belong to any 8-bit port), also a square wave output controlled by the beep object. This output is driving a buzzer.
PL_IO_NULL:	This is a NULL line that does not physically exist. The state of this line is always detected as LOW. Setting this line has no effect.

Notes:

1. When a serial port is in the UART 380 mode (ser.mode 409 = 0- PL_SER_MODE_UART) this line is automatically configured to be an input when this serial port is enabled (ser.enabled 405 = 1- YES) and returns to the previous input/output and high/low state when this serial port is closed (ser.enabled= 0- NO).
2. When a serial port is in the UART mode (ser.mode= 0- PL_SER_MODE_UART) this line is automatically configured to be an output when the serial port is enabled (ser.enabled= 1- YES) and returns to the previous input/output and high/low state when the serial port is closed (ser.enabled= 0- NO).
3. In this list, the term "serial port" actually refers to "serial channel".

Enum pl_io_port_num

Enum pl_io_port_num contains the list of available 8-bit GPIO ports. Use these constants when selecting the port with the io. 294 object (see the io.portnum 302 property).

Note that GPIO lines are of *unidirectional* type 194 and require explicit configuration as outputs or inputs. Some lines are configured as inputs or outputs automatically -- see Enum pl_io_num 171 for details.

- The DS1101 device has the RS232 transceiver IC onboard. This IC "dictates" the GPIO line direction for certain lines. Do not try to use GPIO lines 0, 2, 4, and 6 as outputs -- this can permanently damage the hardware.

Enum pl_io_port_num includes only one constant:

PL_IO_PORT_NUM_0:	8-bit port 0, contains I/O lines 24-31 (the data bus of the OLED display).

Enum pl_int_num

Enum pl_int_num contains the list of constants that refer to available *interrupt* lines. Interrupt lines are mapped to GPIO lines 171 (this mapping can't be altered). Keep in mind that for an interrupt line to work you need to configure 194 the corresponding GPIO line as input.

Enum pl_int_num for this platform includes the following constants:

PL_INT_NUM_0:	Interrupt line 0 (mapped onto I/O line 0).
PL_INT_NUM_1:	Interrupt line 1 (mapped onto I/O line 1).
PL_INT_NUM_2:	Interrupt line 2 (mapped onto I/O line 2).
PL_INT_NUM_3:	Interrupt line 3 (mapped onto I/O line 3).
PL_INT_NUM_4:	Interrupt line 4 (mapped onto I/O line 4).
PL_INT_NUM_5:	Interrupt line 5 (mapped onto I/O line 5).
PL_INT_NUM_6:	Interrupt line 6 (mapped onto I/O line 6).
PL_INT_NUM_7:	Interrupt line 7 (mapped onto I/O line 7).
PL_INT_NULL:	This is a NULL interrupt line that does not physically exist.

Enum pl_sock_interfaces

Enum pl_sock_interfaces contains the list of network interfaces supported by the platform:

0- PL_SOCK_INTERFACE_NULL:	Null (empty) interface.
1- PL_SOCK_INTERFACE_NET **(default)**:	Ethernet interface 358.
2- PL_SOCK_INTERFACE_WLN:	Wi-Fi interface 536.
3- PL_SOCK_INTERFACE_PPP:	PPP interface 366.
4- PL_SOCK_INTERFACE_PPPOE:	PPPoE interface 369.

DS1102W Platform

- **Memory space**

RAM	**22,528*** bytes for application variables and data
Flash	**983,040** bytes for application and data storage (shared 200 flash

	memory)**
EEPROM	**2040** bytes for application data

** RAM available in the debug mode is smaller by 257 bytes. All memory is available in the release mode.*

*** Some earlier devices had only 458,752 bytes of flash memory available*

Supported Objects, variable types, and functions

- Sock 421 — socket communications (up to 16 UDP, TCP, and HTTP sessions);
- Net 358 — controls the Ethernet port;
- Wln 536 — handles the Wi-Fi interface (requires GA1000 201 add-on module, i.e. "G" device option);
- Ser 378 — in charge of the serial ports (on this device, serial channels 205 of the RS232 port);
- Io 294 — handles I/O lines, ports, and interrupts;
- Lcd 317 — controls the 96x32 monochrome OLED display (therefore, requires the "D" device option);
- Fd 236 — manages flash memory file system and direct sector access;
- Stor 522 — provides access to the EEPROM;
- Romfile 370 — facilitates access to resource files (fixed data);
- Pppoe 369 — provides access to the Internet over an ADSL modem;
- Ppp 366 — provides access to the Internet over a serial modem (GPRS, POTS, etc.);
- Pat 363 — "plays" patterns on a pair of LEDs;
- Beep 232 — generates buzzer patterns;
- Button 234 — monitors the MD line (setup button 201);
- Sys 526 — in charge of general device functionality.

These platforms support the standard set of variable types 192 and functions 192 (a.k.a. "syscalls").

Platform-specific constants

You can find them here 176.

Miscellaneous information

Available network interfaces	**Ethernet (net. 358), Wi-Fi (wln. 536)**(1)
GPIO type 194	**Unidirectional**
RTS/CTS remapping 195	**Supported**(2)
GA1000 lines remapping 201	**Supported**(3)
Serial port FIFOs 196	**16 byte for TX, 16 bytes for RX**

Clock frequency (PLL) control 196	**PLL on (def): 88.4736Mhz, PLL off: 11.0592Mhz**(4)
Special configuration section of the EEPROM 197	**8 bytes for MAC storage**
Device serial number 199	**128 bytes: 64 OTP bytes + 64 fixed bytes**
Flash memory configuration 200	**Shared**
LEDs 200	**Green (SG) and red (SR) Status LEDs** **A dedicated Ethernet link status LED (yellow)** **An LED bar consisting of five blue LEDs**(5)
Debug communications 204	**Ethernet / UDP Broadcast transport** **Ethernet /WinPCap transport**

Comments:

1. The sock.allowedinterfaces 474 property refers to the Ethernet interface as "NET", Wi-Fi -- as "WLN". Sock.targetinterface 506 and sock.currentinterface 478 properties rely on the pl_sock_interfaces 180 enum, whose members differ depending on the platform.
2. The hardware of this platform supports serial channels 205, which means that CTS/RTS and DTR/DSR pairs of the DB9 connector can also be used as RX/TX pairs of additional serial channels. Therefore, it is up to you to decide which lines are used as RTS and CTS, and which channels they will belong to. Proper mapping will be required to implement each particular arrangement.
3. Although the platform itself supports remapping, actual "wires" connecting the system to the GA1000 are fixed and your mapping should reflect this:

CS	15- PL_IO_NUM_15 178
CLK	14- PL_IO_NUM_14 178
DI	12- PL_IO_NUM_12 178
DO	13- PL_IO_NUM_13 178
RST	11- PL_IO_NUM_11 178

4. Default PLL state after the external reset is ON.
5. The LEDs of the LED bar are mapped to GPIO lines 178. The LEDs are primarily intended for the Wi-Fi signal strength indication but can also be used for other purposes.

Platform-specific Constants

The following constant lists are platform-specific:

- Enum pl_redir 177 - the list of constants that define buffer redirection (shorting) for this platform.
- Enum pl_io_num 178 - the list of constants that define available I/O lines.
- Enum pl_io_port_num 180 - the list of constants that define available 8-bit I/O ports.

- Enum pl_int_num 180 - the list of constants that define available *interrupt* lines.
- Enum pl_sock_interfaces 180 - the list of available network interfaces.

Enum pl_redir

Enum pl_redir contains the list of constants that define buffer redirection (shorting) for these platforms. The following objects support buffers and buffer redirection:

- Ser. 378 object (see ser.redir 414 method)
- Sock. 421 object (see sock.redir 493 method)

Enum pl_redir for this platform includes the following constants:

0- PL_REDIR_NONE:	Cancels redirection for the serial port* or socket.
1- PL_REDIR_SER:	Redirects RX data of the serial port or socket to the TX buffer of the serial port 0. This constant can be used as a "base" for all other serial ports, i.e. in expressions like ser.redir= PL_REDIR_SER+f.
1- PL_REDIR_SER0:	Redirects RX data of the serial port or socket to the TX buffer of the serial port 0.
2- PL_REDIR_SER1:	Redirects RX data of the serial port or socket to the TX buffer of the serial port 1.
3- PL_REDIR_SER2:	Redirects RX data of the serial port or socket to the TX buffer of the serial port 2.
4- PL_REDIR_SER3:	Redirects RX data of the serial port or socket to the TX buffer of the serial port 3.
6- PL_REDIR_SOCK0:	Redirects RX data of the serial port or socket to the TX buffer of socket 0. This constant can be used as a "base" for all other sockets i.e. in expressions like sock.redir= PL_REDIR_SOCK0+f.
7- PL_REDIR_SOCK1:	Redirects RX data of the serial port or socket to the TX buffer of socket 1.
8- PL_REDIR_SOCK2:	Redirects RX data of the serial port or socket to the TX buffer of socket 2.
9- PL_REDIR_SOCK3:	Redirects RX data of the serial port or socket to the TX buffer of socket 3.
10- PL_REDIR_SOCK4:	Redirects RX data of the serial port or socket to the TX buffer of socket 4.
11- PL_REDIR_SOCK5:	Redirects RX data of the serial port or socket to the TX buffer of socket 5.
12- PL_REDIR_SOCK6:	Redirects RX data of the serial port or socket to the TX buffer of socket 6.
13- PL_REDIR_SOCK7:	Redirects RX data of the serial port or socket to the TX buffer of socket 7.
14- PL_REDIR_SOCK8:	Redirects RX data of the serial port or socket to the TX buffer of socket 8.
15- PL_REDIR_SOCK9:	Redirects RX data of the serial port or socket to the TX buffer of socket 9.
16- PL_REDIR_SOCK10:	Redirects RX data of the serial port or socket to the TX buffer of socket 10.

17- PL_REDIR_SOCK11: Redirects RX data of the serial port or socket to the TX buffer of socket 11.

18- PL_REDIR_SOCK12: Redirects RX data of the serial port or socket to the TX buffer of socket 12.

19- PL_REDIR_SOCK13: Redirects RX data of the serial port or socket to the TX buffer of socket 13.

20- PL_REDIR_SOCK14: Redirects RX data of the serial port or socket to the TX buffer of socket 14.

21- PL_REDIR_SOCK15: Redirects RX data of the serial port or socket to the TX buffer of socket 15.

**In this list, the term "serial port" actually refers to "serial channel".*

Enum pl_io_num

Enum pl_io_num contains the list of constants that refer to available GPIO lines. Use these constants when selecting the line with the io. 294 object (see the io.num 301 property).

Note that GPIO lines are of *unidirectional* type 194 and require explicit configuration as outputs or inputs. Some lines are configured as inputs or outputs automatically -- see the notes at the bottom of the page.

- The DS1102 device has the RS232 transceiver IC onboard. This IC "dictates" the GPIO line direction for certain lines. Do not try to use GPIO lines 0, 2, 4, and 6 as outputs -- this can permanently damage the hardware.

Enum pl_io_num includes the following constants:

0- PL_IO_NUM_0_RX0_INT0(1): General-purpose I/O line 0 (P0.0). This line is also the RX/W1in/din 379 input of the serial port(3) 0 and the interrupt line 0.

1- PL_IO_NUM_1_TX0_INT1(2): General-purpose I/O line 1 (P0.1). This line is also the TX/W1out/dout 379 output of the serial port 0 and the interrupt line 1.

2- PL_IO_NUM_2_RX1_INT2(1): General-purpose I/O line 2 (P0.2). This line is also the RX/W0&1in/din input of the serial port 1 and the interrupt line 2.

3- PL_IO_NUM_3_TX1_INT3(2): General-purpose I/O line 3 (P0.3). This line is also the TX/W1out/dout output of the serial port 1 and the interrupt line 3.

4- PL_IO_NUM_4_RX2_INT4(1): General-purpose I/O line 4 (P0.4). This line is also the RX/W0&1in/din input of the serial port 2 and the interrupt line 4.

5- PL_IO_NUM_5_TX2_INT5(2): General-purpose I/O line 5 (P0.5). This line is also the TX/W1out/dout output of the serial port 2 and the interrupt line 5.

6- PL_IO_NUM_6_RX3_INT6(1): Not implemented.

7- PL_IO_NUM_7_EMPTY: Not implemented.

8- PL_IO_NUM_8_PWROUT: Not implemented.

PL_IO_NUM_9: Pin 8 on the wireless interface connector.

PL_IO_NUM_10:	Pin 6 on the wireless interface connector.
PL_IO_NUM_11:	The RST line of the wireless interface.
PL_IO_NUM_12:	The DI line of the wireless interface.
PL_IO_NUM_13:	The DO line of the wireless interface.
PL_IO_NUM_14:	The CLK line of the wireless interface.
PL_IO_NUM_15:	The CS line of the wireless interface.
PL_IO_NUM_16:	Pin 9 on the wireless interface connector.
PL_IO_NUM_17_RSMD:	Serial port mode: HIGH for RS422/485, LOW for RS232.
PL_IO_NUM_18_HDFD:	Serial port mode: HIGH for full-duplex, LOW for half-duplex.
PL_IO_NUM_19_SB1:	LED bar, LED#1 (the "weakest signal" LED).
PL_IO_NUM_20_SB2:	LED bar, LED#2.
PL_IO_NUM_21_SB3:	LED bar, LED#3.
PL_IO_NUM_22_SB4:	LED bar, LED#4.
PL_IO_NUM_23_SB5:	LED bar, LED#5 (the "strongest signal" LED).
PL_IO_NUM_24_OLED_D0:	OLED dispay data bus, line 0 (P0.0).
PL_IO_NUM_25_OLED_D1:	OLED dispay data bus, line 1 (P0.0).
PL_IO_NUM_26_OLED_D2:	OLED dispay data bus, line 2 (P0.1).
PL_IO_NUM_27_OLED_D3:	OLED dispay data bus, line 3 (P0.2).
PL_IO_NUM_28_OLED_D4:	OLED dispay data bus, line 4 (P0.3).
PL_IO_NUM_29_OLED_D5:	OLED dispay data bus, line 5 (P0.4).
PL_IO_NUM_30_OLED_D6:	OLED dispay data bus, line 6 (P0.5).
PL_IO_NUM_31_OLED_D7:	OLED dispay data bus, line 7 (P0.6).
PL_IO_NUM_32_OLED_CS:	OLED display, CS control line.
PL_IO_NUM_33_OLED_RD:	OLED display, RD control line.
PL_IO_NUM_34_OLED_WR:	OLED display, WR control line.
PL_IO_NUM_35_OLED_DC:	OLED display, DC control line.
PL_IO_NUM_36_OLED_RST:	OLED display, RST control line.
PL_IO_NUM_37_CO:	General-purpose I/O line 37 (does not belong to any 8-bit port), also a square wave output controlled by the beep object. This output is driving a buzzer.
PL_IO_NULL:	This is a NULL line that does not physically exist. The state of this line is always detected as LOW. Setting this line has no effect.

Notes:

1. When a serial port is in the UART 380 mode (ser.mode 409 = 0- PL_SER_MODE_UART) this line is automatically configured to be an input when this serial port is enabled (ser.enabled 405 = 1- YES) and returns to the previous input/output and high/low state when this serial port is closed (ser.enabled= 0- NO).
2. When a serial port is in the UART mode (ser.mode= 0- PL_SER_MODE_UART) this line is automatically configured to be an output when the serial port is enabled (ser.enabled= 1- YES) and returns to the previous input/output and high/low

state when the serial port is closed (ser.enabled= 0- NO).

3. In this list, the term "serial port" actually refers to "serial channel".

Enum pl_io_port_num

Enum pl_io_port_num contains the list of available 8-bit GPIO ports. Use these constants when selecting the port with the io. 294 object (see the io.portnum 302 property).

Note that GPIO lines are of *unidirectional* type 194 and require explicit configuration as outputs or inputs. Some lines are configured as inputs or outputs automatically -- see Enum pl_io_num 178 for details.

- The DS1102 device has the RS232 transceiver IC onboard. This IC "dictates" the GPIO line direction for certain lines. Do not try to use GPIO lines 0, 2, 4, and 6 as outputs -- this can permanently damage the hardware.

Enum pl_io_port_num includes only one constant:

PL_IO_PORT_NUM_0:	8-bit port 0, contains I/O lines 24-31 (the data bus of the OLED display).

Enum pl_int_num

Enum pl_int_num contains the list of constants that refer to available *interrupt* lines. Interrupt lines are mapped to GPIO lines 178 (this mapping can't be altered). Keep in mind that for an interrupt line to work you need to configure 194 the corresponding GPIO line as input.

Enum pl_int_num for this platform includes the following constants:

PL_INT_NUM_0:	Interrupt line 0 (mapped onto I/O line 0).
PL_INT_NUM_1:	Interrupt line 1 (mapped onto I/O line 1).
PL_INT_NUM_2:	Interrupt line 2 (mapped onto I/O line 2).
PL_INT_NUM_3:	Interrupt line 3 (mapped onto I/O line 3).
PL_INT_NUM_4:	Interrupt line 4 (mapped onto I/O line 4).
PL_INT_NUM_5:	Interrupt line 5 (mapped onto I/O line 5).
PL_INT_NUM_6:	Interrupt line 6 (mapped onto I/O line 6).
PL_INT_NUM_7:	Interrupt line 7 (mapped onto I/O line 7).
PL_INT_NULL:	This is a NULL interrupt line that does not physically exist.

Enum pl_sock_interfaces

Enum pl_sock_interfaces contains the list of network interfaces supported by the platform:

0- PL_SOCK_INTERFACE_NULL:	Null (empty) interface.
1- PL_SOCK_INTERFACE_NET **(default)**:	Ethernet interface 358.
2- PL_SOCK_INTERFACE_WLN:	Wi-Fi interface 536.
3- PL_SOCK_INTERFACE_PPP:	PPP interface 366.
4- PL_SOCK_INTERFACE_PPPOE:	PPPoE interface 369.

DS1202 Platform

Memory space

RAM	**22,528*** bytes for application variables and data
Flash	**983,040** bytes for application and data storage (shared 200 flash memory)**
EEPROM	**2040** bytes for application data

** RAM available in the debug mode is smaller by 257 bytes. All memory is available in the release mode.*

*** Some earlier devices had only 458,752 bytes of flash memory available*

Supported Objects, variable types, and functions

- Sock 421 — socket communications (up to 16 UDP, TCP, and HTTP sessions);
- Net 358 — controls the Ethernet port;
- Ser 378 — in charge of the serial ports (on this device, serial channels 205 of the RS232 port);
- Ssi 512 — implements up to four serial synchronous interface (SSI) channels, supports SPI, I2C, clock/data, etc.;
- Io 294 — handles I/O lines, ports, and interrupts;
- Fd 236 — manages flash memory file system and direct sector access;
- Stor 522 — provides access to the EEPROM;
- Romfile 370 — facilitates access to resource files (fixed data);
- Pppoe 369 — provides access to the Internet over an ADSL modem;
- Ppp 366 — provides access to the Internet over a serial modem (GPRS, POTS, etc.);
- Pat 363 — "plays" patterns on a pair of LEDs;
- Button 234 — monitors the MD line (setup button 201);
- Sys 526 — in charge of general device functionality.

These platforms support the standard set of variable types 192 and functions 192 (a.k.a. "syscalls").

Platform-specific constants

You can find them here 182.

Miscellaneous information

Available network interfaces	**Ethernet (net.)**(1)
GPIO type 194	**Unidirectional**
RTS/CTS remapping 195	**Supported**(2)
GA1000 lines remapping 201	**This platform does not support Wi-Fi at all**
Serial port FIFOs 196	**16 byte for TX, 16 bytes for RX**
Clock frequency (PLL) control 196	**PLL on (def): 88.4736Mhz, PLL off: 11.0592Mhz**(3)
Special configuration section of the EEPROM 197	**8 bytes for MAC storage**
Device serial number 199	**128 bytes: 64 OTP bytes + 64 fixed bytes**(4)
Flash memory configuration 200	**Shared**
LEDs 200	**Green (SG) and red (SR) Status LEDs** **Green (EG) and yellow (EY) Ethernet LEDs**
Debug communications 204	**Ethernet / UDP Broadcast transport** **Ethernet /WinPCap transport**

Comments:

1. The sock.allowedinterfaces 474 property refers to the Ethernet interface as "NET". Sock.targetinterface 506 and sock.currentinterface 478 properties rely on the pl_sock_interfaces 186 enum, whose members differ depending on the platform.
2. The hardware of this platform supports serial channels 205, which means that CTS/RTS and DTR/DSR pairs of the DB9 connector can also be used as RX/TX pairs of additional serial channels. Therefore, it is up to you to decide which lines are used as RTS and CTS, and which channels they will belong to. Proper mapping will be required to implement each particular arrangement.
3. Default PLL state after the external reset is ON.
4. Older DS1202 and DS1202W devices did not contain the serial number. To find out if your DS1202(W) has the serial number onboard, try to read this serial number with the sys.serialnum 534 R/O property. If this property returns an empty string, then the serial number is not present. Sys.serialnum returns all 128 bytes of the serial number. First 64 bytes are one-time-programmable (OTP) with the sys.setserialnum 535 method.

Platform-specific Constants

The following constant lists are platform-specific:

- Enum pl_redir 183 - the list of constants that define buffer redirection (shorting) for this platform.
- Enum pl_io_num 184 - the list of constants that define available I/O lines.

- Enum pl_io_port_num 185 - the list of constants that define available 8-bit I/O ports.
- Enum pl_int_num 185 - the list of constants that define available *interrupt* lines.
- Enum pl_sock_interfaces 186 - the list of available network interfaces.

Enum pl_redir

Enum pl_redir contains the list of constants that define buffer redirection (shorting) for these platforms. The following objects support buffers and buffer redirection:

- Ser. 378 object (see ser.redir 414 method)
- Sock. 421 object (see sock.redir 493 method)

Enum pl_redir for this platform includes the following constants:

0- PL_REDIR_NONE:	Cancels redirection for the serial port* or socket.
1- PL_REDIR_SER:	Redirects RX data of the serial port or socket to the TX buffer of the serial port 0. This constant can be used as a "base" for all other serial ports, i.e. in expressions like ser.redir= PL_REDIR_SER+f.
1- PL_REDIR_SER0:	Redirects RX data of the serial port or socket to the TX buffer of the serial port 0.
2- PL_REDIR_SER1:	Redirects RX data of the serial port or socket to the TX buffer of the serial port 1.
3- PL_REDIR_SER2:	Redirects RX data of the serial port or socket to the TX buffer of the serial port 2.
4- PL_REDIR_SER3:	Redirects RX data of the serial port or socket to the TX buffer of the serial port 3.
6- PL_REDIR_SOCK0:	Redirects RX data of the serial port or socket to the TX buffer of socket 0. This constant can be used as a "base" for all other sockets i.e. in expressions like sock.redir= PL_REDIR_SOCK0+f.
7- PL_REDIR_SOCK1:	Redirects RX data of the serial port or socket to the TX buffer of socket 1.
8- PL_REDIR_SOCK2:	Redirects RX data of the serial port or socket to the TX buffer of socket 2.
9- PL_REDIR_SOCK3:	Redirects RX data of the serial port or socket to the TX buffer of socket 3.
10- PL_REDIR_SOCK4:	Redirects RX data of the serial port or socket to the TX buffer of socket 4.
11- PL_REDIR_SOCK5:	Redirects RX data of the serial port or socket to the TX buffer of socket 5.
12- PL_REDIR_SOCK6:	Redirects RX data of the serial port or socket to the TX buffer of socket 6.
13- PL_REDIR_SOCK7:	Redirects RX data of the serial port or socket to the TX buffer of socket 7.
14- PL_REDIR_SOCK8:	Redirects RX data of the serial port or socket to the TX buffer of socket 8.
15- PL_REDIR_SOCK9:	Redirects RX data of the serial port or socket to the TX buffer of socket 9.

16- PL_REDIR_SOCK10: Redirects RX data of the serial port or socket to the TX buffer of socket 10.

17- PL_REDIR_SOCK11: Redirects RX data of the serial port or socket to the TX buffer of socket 11.

18- PL_REDIR_SOCK12: Redirects RX data of the serial port or socket to the TX buffer of socket 12.

19- PL_REDIR_SOCK13: Redirects RX data of the serial port or socket to the TX buffer of socket 13.

20- PL_REDIR_SOCK14: Redirects RX data of the serial port or socket to the TX buffer of socket 14.

21- PL_REDIR_SOCK15: Redirects RX data of the serial port or socket to the TX buffer of socket 15.

**In this list, the term "serial port" actually refers to "serial channel".*

Enum pl_io_num

Enum pl_io_num contains the list of constants that refer to available GPIO lines. Use these constants when selecting the line with the io. 294 object (see the io.num 301 property).

Note that GPIO lines are of *unidirectional* type 194 and require explicit configuration as outputs or inputs. Some lines are configured as inputs or outputs automatically -- see the notes at the bottom of the page.

- The EM1202EV-RS board and the DS1202 controller have the RS232 transceiver IC onboard. This IC "dictates" the GPIO line direction for certain lines. Do not try to use GPIO lines 0, 2, 4, and 6 as outputs -- this can permanently damage the hardware.

Enum pl_io_num includes the following constants:

0- PL_IO_NUM_0_RX0_INT0(1): General-purpose I/O line 0 (P0.0). This line is also the RX/W1in/din 379 input of the serial port(3) 0 and the interrupt line 0.

1- PL_IO_NUM_1_TX0_INT4(2): General-purpose I/O line 1 (P0.1). This line is also the TX/W1out/dout 379 output of the serial port 0 and the interrupt line 4.

2- PL_IO_NUM_2_RX1_INT1(1): General-purpose I/O line 2 (P0.2). This line is also the RX/W0&1in/din input of the serial port 1 and the interrupt line 1.

3- PL_IO_NUM_3_TX1_INT5(2): General-purpose I/O line 3 (P0.3). This line is also the TX/W1out/dout output of the serial port 1 and the interrupt line 5.

4- PL_IO_NUM_4_RX2_INT2(1): General-purpose I/O line 4 (P0.4). This line is also the RX/W0&1in/din input of the serial port 2 and the interrupt line 2.

5- PL_IO_NUM_5_TX2_INT6(2): General-purpose I/O line 5 (P0.5). This line is also the TX/W1out/dout output of the serial port 2 and the interrupt line 6.

6- PL_IO_NUM_6_RX3_INT3(1): General-purpose I/O line 6 (P0.6). This line is also

	the RX/W0&1in/din input of the serial port 3 and the interrupt line 3.
7- PL_IO_NUM_7_INT7(2):	General-purpose I/O line 7 (P0.7). This line is also the interrupt line 7.
PL_IO_NULL:	This is a NULL line that does not physically exist. The state of this line is always detected as LOW. Setting this line has no effect.

Notes:

1. When a serial port is in the UART 380 mode (ser.mode 409 = 0- PL_SER_MODE_UART) this line is automatically configured to be an input when this serial port is enabled (ser.enabled 405 = 1- YES) and returns to the previous input/output and high/low state when this serial port is closed (ser.enabled= 0- NO).
2. When a serial port is in the UART mode (ser.mode= 0- PL_SER_MODE_UART) this line is automatically configured to be an output when the serial port is enabled (ser.enabled= 1- YES) and returns to the previous input/output and high/low state when the serial port is closed (ser.enabled= 0- NO).
3. In this list, the term "serial port" actually refers to "serial channel".

Enum pl_io_port_num

Enum pl_io_port_num contains the list of available 8-bit GPIO ports. Use these constants when selecting the port with the io. 294 object (see the io.portnum 302 property).

Note that GPIO lines are of *unidirectional* type 194 and require explicit configuration as outputs or inputs. Some lines are configured as inputs or outputs automatically -- see Enum pl_io_num 184 for details.

- The EM1202EV-RS board and the DS1202 controller have RS232 transceiver IC onboard. This IC defines which I/O lines of the device should be configured as inputs, and which- as outputs. Specifically, do not try to use I/I lines 0, 2, 4, 6, and 7 as outputs -- this can permanently damage the hardware.

Enum pl_io_port_num includes only one constant:

0- PL_IO_PORT_NUM_0:	8-bit port 0 (P0). Contains I/O lines 0-7.

Enum pl_int_num

Enum pl_int_num contains the list of constants that refer to available *interrupt* lines. Interrupt lines are mapped to GPIO lines 184 (this mapping can't be altered). Keep in mind that for an interrupt line to work you need to configure 194 the corresponding GPIO line as input.

Enum pl_int_num for this platform includes the following constants:

PL_INT_NUM_0:	Interrupt line 0 (mapped onto I/O line 0).
PL_INT_NUM_1:	Interrupt line 1 (mapped onto I/O line 2).
PL_INT_NUM_2:	Interrupt line 2 (mapped onto I/O line 4).
PL_INT_NUM_3:	Interrupt line 3 (mapped onto I/O line 6).

PL_INT_NUM_4:	Interrupt line 4 (mapped onto I/O line 1).
PL_INT_NUM_5:	Interrupt line 5 (mapped onto I/O line 3).
PL_INT_NUM_6:	Interrupt line 6 (mapped onto I/O line 5).
PL_INT_NUM_7:	Interrupt line 7 (mapped onto I/O line 7).
PL_INT_NULL:	This is a NULL interrupt line that does not physically exist.

Enum pl_sock_interfaces

Enum pl_sock_interfaces contains the list of network interfaces supported by the platform:

0- PL_SOCK_INTERFACE_NULL:	Null (empty) interface.
1- PL_SOCK_INTERFACE_NET **(default)**:	Ethernet interface 358.
3- PL_SOCK_INTERFACE_PPP:	PPP interface 366.
4- PL_SOCK_INTERFACE_PPPOE:	PPPoE interface 369.

DS1206 Platform

- **Memory space**

RAM	**22,528*** bytes for application variables and data
Flash	**983,040** bytes for application and data storage (shared 200 flash memory)**
EEPROM	**2040** bytes for application data

** RAM available in the debug mode is smaller by 257 bytes. All memory is available in the release mode.*

*** Some earlier devices had only 458,752 bytes of flash memory available*

Supported Objects, variable types, and functions

- Sock 421 — socket communications (up to 16 UDP, TCP, and HTTP sessions);
- Net 358 — controls the Ethernet port;
- Ser 378 — in charge of the serial ports (on this device, serial channels 205 of the RS232 port);
- Ssi 512 — implements up to four serial synchronous interface (SSI) channels, supports SPI, I2C, clock/data, etc.;
- Io 294 — handles I/O lines, ports, and interrupts;
- Fd 236 — manages flash memory file system and direct sector access;

- Stor 522 — provides access to the EEPROM;
- Romfile 370 — facilitates access to resource files (fixed data);
- Pppoe 369 — provides access to the Internet over an ADSL modem;
- Ppp 366 — provides access to the Internet over a serial modem (GPRS, POTS, etc.);
- Pat 363 — "plays" patterns on a pair of LEDs;
- Button 234 — monitors the MD line (setup button 201);
- Sys 526 — in charge of general device functionality.

These platforms support the standard set of variable types 192 and functions 192 (a.k.a. "syscalls").

Platform-specific constants

You can find them here 188.

Miscellaneous information

Available network interfaces	**Ethernet (net.)**(1)
GPIO type 194	**Unidirectional**
RTS/CTS remapping 195	**Supported**(2)
GA1000 lines remapping 201	**This platform does not support Wi-Fi at all**
Serial port FIFOs 196	**16 byte for TX, 16 bytes for RX**
Clock frequency (PLL) control 196	**PLL on (def): 88.4736Mhz, PLL off: 11.0592Mhz**(3)
Special configuration section of the EEPROM 197	**8 bytes for MAC storage**
Device serial number 199	**128 bytes: 64 OTP bytes + 64 fixed bytes**(4)
Flash memory configuration 200	**Shared**
LEDs 200	**Green (SG) and red (SR) Status LEDs** **Green (EG) and yellow (EY) Ethernet LEDs**
Debug communications 204	**Ethernet / UDP Broadcast transport** **Ethernet /WinPCap transport**

Comments:

1. The sock.allowedinterfaces 474 property refers to the Ethernet interface as "NET". Sock.targetinterface 506 and sock.currentinterface 478 properties rely on the pl_sock_interfaces 191 enum, whose members differ depending on the platform.
2. The hardware of this platform supports serial channels 205, which means that CTS/RTS and DTR/DSR pairs of the DB9 connector can also be used as RX/TX pairs of additional serial channels. Therefore, it is up to you to decide which lines

are used as RTS and CTS, and which channels they will belong to. Proper mapping will be required to implement each particular arrangement.

3. Default PLL state after the external reset is ON.
4. Older DS1206 and DS1206W devices did not contain the serial number. To find out if your DS1206(W) has the serial number onboard, try to read this serial number with the sys.serialnum 534 R/O property. If this property returns an empty string, then the serial number is not present. Sys.serialnum returns all 128 bytes of the serial number. First 64 bytes are one-time-programmable (OTP) with the sys.setserialnum 535 method.

Platform-specific Constants

The following constant lists are platform-specific:

- Enum pl_redir 188 - the list of constants that define buffer redirection (shorting) for this platform.
- Enum pl_io_num 189 - the list of constants that define available I/O lines.
- Enum pl_io_port_num 190 - the list of constants that define available 8-bit I/O ports.
- Enum pl_int_num 191 - the list of constants that define available *interrupt* lines.
- Enum pl_sock_interfaces 191 - the list of available network interfaces.

Enum pl_redir

Enum pl_redir contains the list of constants that define buffer redirection (shorting) for these platforms. The following objects support buffers and buffer redirection:

- Ser. 378 object (see ser.redir 414 method)
- Sock. 421 object (see sock.redir 493 method)

Enum pl_redir for this platform includes the following constants:

0- PL_REDIR_NONE:	Cancels redirection for the serial port* or socket.
1- PL_REDIR_SER:	Redirects RX data of the serial port or socket to the TX buffer of the serial port 0. This constant can be used as a "base" for all other serial ports, i.e. in expressions like ser.redir= PL_REDIR_SER+f.
1- PL_REDIR_SER0:	Redirects RX data of the serial port or socket to the TX buffer of the serial port 0.
2- PL_REDIR_SER1:	Redirects RX data of the serial port or socket to the TX buffer of the serial port 1.
3- PL_REDIR_SER2:	Redirects RX data of the serial port or socket to the TX buffer of the serial port 2.
4- PL_REDIR_SER3:	Redirects RX data of the serial port or socket to the TX buffer of the serial port 3.
6- PL_REDIR_SOCK0:	Redirects RX data of the serial port or socket to the TX buffer of socket 0. This constant can be used as a "base" for all other sockets i.e. in expressions like sock.redir= PL_REDIR_SOCK0+f.
7- PL_REDIR_SOCK1:	Redirects RX data of the serial port or socket to the TX buffer of socket 1.
8- PL_REDIR_SOCK2:	Redirects RX data of the serial port or socket to the TX

buffer of socket 2.

9- PL_REDIR_SOCK3: Redirects RX data of the serial port or socket to the TX buffer of socket 3.

10- PL_REDIR_SOCK4: Redirects RX data of the serial port or socket to the TX buffer of socket 4.

11- PL_REDIR_SOCK5: Redirects RX data of the serial port or socket to the TX buffer of socket 5.

12- PL_REDIR_SOCK6: Redirects RX data of the serial port or socket to the TX buffer of socket 6.

13- PL_REDIR_SOCK7: Redirects RX data of the serial port or socket to the TX buffer of socket 7.

14- PL_REDIR_SOCK8: Redirects RX data of the serial port or socket to the TX buffer of socket 8.

15- PL_REDIR_SOCK9: Redirects RX data of the serial port or socket to the TX buffer of socket 9.

16- PL_REDIR_SOCK10: Redirects RX data of the serial port or socket to the TX buffer of socket 10.

17- PL_REDIR_SOCK11: Redirects RX data of the serial port or socket to the TX buffer of socket 11.

18- PL_REDIR_SOCK12: Redirects RX data of the serial port or socket to the TX buffer of socket 12.

19- PL_REDIR_SOCK13: Redirects RX data of the serial port or socket to the TX buffer of socket 13.

20- PL_REDIR_SOCK14: Redirects RX data of the serial port or socket to the TX buffer of socket 14.

21- PL_REDIR_SOCK15: Redirects RX data of the serial port or socket to the TX buffer of socket 15.

**In this list, the term "serial port" actually refers to "serial channel".*

Enum pl_io_num

Enum pl_io_num contains the list of constants that refer to available GPIO lines. Use these constants when selecting the line with the io. 294 object (see the io.num 301 property).

Note that GPIO lines are of *unidirectional* type 194 and require explicit configuration as outputs or inputs. Some lines are configured as inputs or outputs automatically -- see the notes at the bottom of the page.

- The DS1206N-RS board and the DS1206 controller have the RS232 transceiver IC onboard. This IC "dictates" the GPIO line direction for certain lines. Do not try to use GPIO lines 0, 2, 4, and 6 as outputs -- this can permanently damage the hardware.

Enum pl_io_num includes the following constants:

0- PL_IO_NUM_0_RX0_INT0(1): General-purpose I/O line 0 (P0.0). This line is also the RX/W1in/din 379 input of the serial port(3) 0 and the interrupt line 0.

1- PL_IO_NUM_1_TX0_INT1(2):	General-purpose I/O line 1 (P0.1). This line is also the TX/W1out/dout 379 output of the serial port 0 and the interrupt line 1.
2- PL_IO_NUM_2_RX1_INT2(1):	General-purpose I/O line 2 (P0.2). This line is also the RX/W0&1in/din input of the serial port 1 and the interrupt line 2.
3- PL_IO_NUM_3_TX1_INT3(2):	General-purpose I/O line 3 (P0.3). This line is also the TX/W1out/dout output of the serial port 1 and the interrupt line 3.
4- PL_IO_NUM_4_RX2_INT4(1):	General-purpose I/O line 4 (P0.4). This line is also the RX/W0&1in/din input of the serial port 2 and the interrupt line 4.
5- PL_IO_NUM_5_TX2_INT5(2):	General-purpose I/O line 5 (P0.5). This line is also the TX/W1out/dout output of the serial port 2 and the interrupt line 5.
6- PL_IO_NUM_6_RX3_INT6(1):	General-purpose I/O line 6 (P0.6). This line is also the RX/W0&1in/din input of the serial port 3 and the interrupt line 6.
7- PL_IO_NUM_7_EMPTY:	Not implemented.
8- PL_IO_NUM_8_PWROUT:	Controls the power output on pin 9 of the DB9M connector (this applies only to DS1206N-RS and DS1206 devices). Power will be ON when this output is enabled (io.enabled= 1- YES) and set to HIGH (io.state= 1- HIGH).
PL_IO_NULL:	This is a NULL line that does not physically exist. The state of this line is always detected as LOW. Setting this line has no effect.

Notes:

1. When a serial port is in the UART 380 mode (ser.mode 409 = 0- PL_SER_MODE_UART) this line is automatically configured to be an input when this serial port is enabled (ser.enabled 405 = 1- YES) and returns to the previous input/output and high/low state when this serial port is closed (ser.enabled= 0- NO).
2. When a serial port is in the UART mode (ser.mode= 0- PL_SER_MODE_UART) this line is automatically configured to be an output when the serial port is enabled (ser.enabled= 1- YES) and returns to the previous input/output and high/low state when the serial port is closed (ser.enabled= 0- NO).
3. In this list, the term "serial port" actually refers to "serial channel".

Enum pl_io_port_num

Enum pl_io_port_num contains the list of available 8-bit GPIO ports. Use these constants when selecting the port with the io. 294 object (see the io.portnum 302 property).

Note that GPIO lines are of *unidirectional* type 194 and require explicit configuration as outputs or inputs. Some lines are configured as inputs or outputs automatically -- see Enum pl_io_num 189 for details.

- The DS1206N-RS board and the DS1206 controller have RS232 transceiver IC onboard. This IC defines which I/O lines of the device should be configured as

inputs, and which- as outputs. Specifically, do not try to use I/I lines 0, 2, 4, and 6 as outputs -- this can permanently damage the hardware.

Enum pl_io_port_num includes only one constant:

0- PL_IO_PORT_NUM_0: 8-bit port 0 (P0). Contains I/O lines 0-7.

Enum pl_int_num

Enum pl_int_num contains the list of constants that refer to available *interrupt* lines. Interrupt lines are mapped to GPIO lines 189 (this mapping can't be altered). Keep in mind that for an interrupt line to work you need to configure 194 the corresponding GPIO line as input.

Enum pl_int_num for this platform includes the following constants:

PL_INT_NUM_0:	Interrupt line 0 (mapped onto I/O line 0).
PL_INT_NUM_1:	Interrupt line 1 (mapped onto I/O line 1).
PL_INT_NUM_2:	Interrupt line 2 (mapped onto I/O line 2).
PL_INT_NUM_3:	Interrupt line 3 (mapped onto I/O line 3).
PL_INT_NUM_4:	Interrupt line 4 (mapped onto I/O line 4).
PL_INT_NUM_5:	Interrupt line 5 (mapped onto I/O line 5).
PL_INT_NUM_6:	Interrupt line 6 (mapped onto I/O line 6).
PL_INT_NUM_7:	Interrupt line 7 (mapped onto I/O line 7).
PL_INT_NULL:	This is a NULL interrupt line that does not physically exist.

Enum pl_sock_interfaces

Enum pl_sock_interfaces contains the list of network interfaces supported by the platform:

0- PL_SOCK_INTERFACE_NULL:	Null (empty) interface.
1- PL_SOCK_INTERFACE_NET **(default)**:	Ethernet interface 358.
3- PL_SOCK_INTERFACE_PPP:	PPP interface 366.
4- PL_SOCK_INTERFACE_PPPOE:	PPPoE interface 369.

Common Information

- Supported Variable Types 192
- Supported Functions 192
- GPIO Type 194
- RTS/CTS Remapping 195

- Serial port FIFOs 196
- Clock Frequency (PLL) Control 196
- Special Configuration Section of the EEPROM 197
- Device Serial Number 199
- Flash Memory Configuration 200
- LEDs 200
- Setup (MD) Button (Line) 201
- Connecting GA1000 201
- Debug Communications 204

Supported Variable Types

The following variable types are supported by all devices:

- Byte
- Word
- Dword
- Char
- Short (integer)
- Long
- Real (float)
- Boolean
- User-defined structures
- User-defined enumeration types

For general type description see Variables and Their Types 48.

Supported Functions

The following syscalls (platform functions) are supported by all:

- String-related:
  - Asc 206 string character --> ASCII code;
  - Chr 208 ASCII code --> string character;
  - Val 229 numerical string--> 16-bit value (word or short);
  - Lval 218 numerical string --> 32-bit value (dword or long);
  - Strtof 228 numerical string --> real value;
  - Bin 207 unsigned 16-bit numeric value (word) --> binary numerical string;
  - Lbin 214 unsigned 32-bit numeric value (dword) --> binary numerical string;
  - Str 225 unsigned 16-bit numeric value (word) --> decimal numerical string;
  - Stri 226 signed 16-bit numeric value (short) --> decimal numerical string;
  - Lstr 216 unsigned 32-bit numeric value (dword) --> decimal numerical string;

- Lstri 217 signed 32-bit numeric value (long) --> decimal numerical string;
- Hex 212 unsigned 16-bit numeric value (word) --> hexadecimal numerical string;
- Lhex 216 unsigned 32-bit numeric value (dword) --> hexadecimal numerical string;
- Ftostr 211 real value --> numerical string;
- Len 215 gets the string length;
- Left 215 gets a left portion of a string;
- Mid 219 gets a middle portion of a string;
- Right 223 gets a right portion of a string;
- Insert 213 inserts a string into another string;
- Instr 214 finds a substring in a string;
- Strgen 226 generates a string using repeating substring;
- Strsum 227 calculates 16-bit (word) sum of string characters' ASCII codes;
- Ddstr 210 dot-decimal value --> dot-decimal string;
- Ddval 210 dot-decimal string --> dot-decimal value;
- Strand 225 logical AND on corresponding data bytes from two strings;
- Stror 227 logical OR on corresponding data bytes from two strings;
- Strxor 228 logical XOR on corresponding data bytes from two strings.

- Date and time serialization and de-serialization:
  - Daycount 209 given year, month, and date --> day number;
  - Mincount 220 given hours and minutes --> minute number;
  - Year 230 given day number --> year;
  - Month 221 given day number --> month;
  - Date 208 given day number --> date;
  - Weekday 230 given day number --> day of the week;
  - Hours 213 given minutes number --> hours;
  - Minutes 221 given minutes number --> minutes.
- Hash calculation, encryption, and related functions:
  - Aes128enc 206 encrypts data according to the AES128 algorithm (**not supported** on the EM500W 138 and DS1100 164 platforms);
  - Aes128dec 205 decrypts data according to the AES128 algorithm (**not supported** on the EM500W 138 and DS1100 164 platforms);
  - Rc4 222 encrypts/decrypts data according to the RC4 algorithm;
  - Md5 218 calculates MD5 hash of a string;
  - Sha1 223 calculates SHA-1 hash of a string;
  - Random 222 generates a random string;
- Miscellaneous
  - Cfloat 207 checks the validity of a real value.

GPIO Type 10.3

As for as GPIO lines go, Tibbo devices fall into two categories:

- Devices with unidirectional GPIO lines: such devices require explicit configuration of each GPIO line as input or output; and
- Devices with bidirectional GPIO lines: GPIO lines of these devices work as outputs and inputs at the same time.

To find out the type of GPIO lines on your device, refer to its platform documentation (for example, EM1000's is here 143).

Devices with unidirectional GPIO lines

On such devices you need to explicitly enable or disable the output driver of each I/O line (controlled by the io. 294 object). When the device boots up all pins are configured as inputs. If you want to use any particular I/O pin as the output, enable this pin's output driver first (set io.enabled 298 = 1-YES):

```
...
io.num= PL_IO_NUM_5      'select the line
io.enabled= YES          'enable output driver (you need to do this only
once)
io.state= LOW            'set the state
...
```

- Make sure that your external circuitry does not attempt to drive the I/O lines that have their output drivers enabled. Severe damage to the device and/or your circuitry may occur if this happens!

When the driver is enabled (io.enabled= 1-YES) and you read the state of the pin, you get back the state of your own output driver. To turn the line into an input, switch the output driver off (ser.enabled= 0- NO). This will allow you to sense the state of the external signal applied to the I/O line:

```
...
io.num= PL_IO_NUM_4      'just to select some line as an example
io.enabled= NO           'now the output driver is off
x=io.state               'read line state into x
...
```

There is the io.portenabled 302 property as well. It allows simultaneous configuration of all GPIO lines in the 8-bit I/O port.

Serial port 378 lines also require proper configuration. Depending on the mode of the serial port (see ser.mode 409) you need to set the following:

ser.mode	TX/ W1out/	RX/W1in/ din input	RTS/ W0out/	CTS/ W0&1in/

	dout output		cout output	cin input
0- PL_SER_MODE_UART 380	Will auto-configure as output(1)	Will auto-configure as input(1)	Requires configuration as output (2)	Requires configuration as input(2)
1- PL_SER_MODE_WIEGAND 383	Requires configuration as output	Requires configuration as input		
2- PL_SER_MODE_CLOCKDATA 386				

Notes:

1. When This line does not require configuration, it will be configured automatically as input or output when the port is opened. When the port is closed the line will return to the input/output and high/low state it had before the port was opened.
2. If RTS/CTS remapping 195 is supported, you need to configure the I/O pin to which this line of the serial port is *currently* mapped.

Devices with bidirectional GPIO lines

I/O lines of these devices do not require explicit configuration as inputs or outputs. All lines are "quasi-bidirectional" and can be viewed as open collector outputs with weak pull-up resistors. To "measure" an external signal applied to an I/O line, set this line to HIGH first, then read the state of the line. It is OK to drive the line LOW externally when the same line outputs HIGH internally. Io.enabled 298 and io.portenabled 302 properties exist, but only for compatibility with other platforms. Writing to these properties has no effect and reading them always returns 1- YES and 255 (all lines enabled) correspondingly.

To sense the state of the external signal applied to the I/O line, set the line to HIGH first:

```
...
io.num= PL_IO_NUM   'just to select some line as an example
io.state= HIGH      'now we can read the line
x=io.state          'read line state into x
...
```

Serial port lines of devices with bidirectional GPIO do not require configuration as well.

RTS/CTS Remapping

Remapping feature allows you to select which GPIO line of your device will function as the RTS line of a serial port 378. This is done through the ser.rtsmap 415 property. In the same manner, the CTS line of a serial port can be mapped to the interrupt line of your choice (see ser.ctsmap 403).

Not all devices support this feature. To find out if your device allows RTS/CTS

remapping, refer to its platform documentation (for example, EM1000's is here[143]). This topic will also present the default mapping for CTS and RTS lines of your device.

Note that on some devices you need to explicitly configure[194] CTS lines as inputs and RTS lines as outputs.

Serial Port FIFOs

FIFO (first-in-first-out) is a type of hardware memory that stores the data in a queue fashion. FIFOs are characterized by "depth". For example, 16-byte FIFO can queue up to 16 bytes of data, so it has the depth of 16.

Serial ports usually have two independent FIFOs -- one for the outgoing (TX) data, and one for the incoming (RX) data. The TX FIFO allows the system to place a number of bytes into the queue, and then the serial port's hardware will take care of their orderly transmission. The RX FIFO temporarily stores the incoming data until the system gets a chance to process it.

To find out if your device's serial port(s) have FIFOs and what are the depths of these FIFOs, refer to the device's platform documentation (for example, EM1000's is here[143]).

When the serial port[378] is in the UART/full-duplex/flow control[380] mode (ser.mode[409]= 0- PL_SER_MODE_UART, ser.interface[408]= 0- PL_SER_SI_FULLDUPLEX, and ser.flowcontrol[407]= 1- ENABLED) the device is monitoring its CTS input to see if attached serial device is ready to receive more data. If the CTS state changes to "cannot transmit" the device will stop sending out data immediately. However, the data that has already entered the FIFO will still be sent out. Therefore, after the CTS state becomes "cannot transmit" the device can still send out a number of characters that already went into the transmit FIFO.

Clock Frequency (PLL) Control

Some devices support clock frequency (PLL) control. When the PLL is on, the system is running at the highest possible frequency. When the PLL is off, the system is running at a reduced frequency. For example, the EM1000[143] can run at 88.4736MHz (PLL on) or 11.0592MHz (PLL off).

To find out if your device allows frequency control and what particular frequencies it can operate on, refer to this device's platform documentation (for example, EM1000's is here[143]).

The clock frequency affects all aspects of device operation that rely on this clock. Naturally, program execution speed, too, depends on the clock frequency. Serial port[378] baudrates (see ser.baudrate[402]) in the UART[380] mode (ser.mode[409]= 0- PL_SER_MODE_UART) also depend on the main clock. Finally, the frequency of the square wave generated by the beep.[232] object depends on the main clock as well. The period of the on_sys_timer[533] event of the sys.[526] is not affected by the PLL mode.

There is a way to set the baudrate in the clock-independent (and, actually, platform-independent) way -- see ser.div9600[404] property for details (example of use can be found in the Serial Settings[390] topic). For the beep object, you just have to set the beep.divider[232] correctly depending on the current PLL mode.

For PLL control, the sys.[526] object has a sys.currentpll[531] read-only property and

sys.newpll 532 method. See PLL Management 529 topic- it explains how to switch the PLL on and off.

After the external reset (see sys.resettype 535), devices with PLL control may boot with the PLL on or off. Refer to the "Platform-dependent Programming Information" topic inside your device's platform specifications section for more information.

Special Configuration Section of the EEPROM

All Tibbo devices have an EEPROM memory (accessible through the stor. 522 object).

Bottom 28 bytes of the EEPROM form a so-called "special configuration section". This section includes:

- 8 bytes for storing the MAC address of the device;
- 20 bytes for storing the password of the device.

By default, the special configuration area is not accessible to your application- the stor.base 523 property takes care of that. Unless you change it, this property specifies that your application's storage area starts at address 29. To see the MAC or password data, set the stor.base to 1. Reminder: EEPROM addresses are counted from 1.

MAC address

Bytes 1-8 of the EEPROM store the MAC address of the device. On power-up, the MAC address is read out from the EEPROM and programmed into the Ethernet controller. You can always check the current MAC through the net.mac 360 read-only property of the net. 358 object but there is no direct way to change it. Instead, you can change the MAC address data in the EEPROM itself. Next time the device boots up, it will start using this new address.

MAC address data in the EEPROM has a special format -- you have to follow it if you want the MAC to be recognized by the firmware (TiOS). Here is this format:

addr1	addr2	addr3	addr4	addr5	addr6	addr7	addr8
6	MAC0	MAC1	MAC2	MAC3	MAC4	MAC5	Checksum

The byte at address 1 must be set to 6- this means that 6 bytes of data follow (MAC address consists of 6 bytes). Addresses from 2 to 7 carry the MAC address itself. Address 8 stores the checksum, which is calculated like this:

255-(modulo8_sum_of_addr_1_through_7)

Here is a sample code that stores new MAC address into the EEPROM and then reboots the device:

```
dim s as string
dim x as byte
...
s= "0.2.123.124.220.240"  'supposing, we want to set this MAC
```

```
...
...
s=chr(6)+ddval(s)          'added first byte (always 6) and converted
readable MAC into bytes
x=255-strsum(s)            'calculated checksum and assigned the result to a
BYTE variable ("modulo 8" checksum)
s=s+chr(x)                 'now our string is ready

stor.base(1)               'will access the EEPROM from the bottom
x=stor.set(s,1)            'save the data
if x<>len(s) then          'it is a good programming practive to check the
result
      'failed
else
      sys.reboot           'new MAC set, reboot!
end if

...
```

There are limitations on what MAC address you can set. When loading the MAC into the Ethernet controller, the device always resets the first byte of this address to 0. For example, if you set the MAC to 1.2.3.4.5.6 then the actual MAC used by the device will be 0.2.3.4.5.6.

- If you write incorrect MAC data (wrong "length" byte or erroneous checksum), the device will ignore the stored MAC and boot up with the default MAC, which is 0.1.2.3.4.100.

Password

Bytes 9-28 store device password 41. The password is stored in the following format:

addr9	up to 16 bytes	3 bytes
len	password	MD5 check data

The byte at address 9 indicates the length of the password (16 bytes max). This is followed by the password itself, followed by 3 check bytes. These bytes are first 3 bytes of the md5 218 hash calculated over the combined length and password data bytes.

TiOS itself can accept any binary password with 0-16 bytes length (zero means "no password protection"). TIDE software, however, always sets 16-byte passwords. These are MD5 hashes of the actual passwords you type into **Enter Device Password** and **Change Device Password** dialogs.

Here is a code snippet that verifies if the password set for the device is really DEVICE_PASSWORD ("Tibbo" in the example below). If the EEPROM data is incorrect, the code sets the right data and reboots the device. By using this sort of code in your project you can make your devices self-protect themselves. This will spare you from manually setting the password for each device through TIDE.

```
'--------------------------------------------------------------------
Const DEVICE_PASSWORD="Tibbo"

'====================================================================
Sub On_sys_init()
      Dim s As String(16)
      Dim s2 As String(20)
      Dim check As String(3)
      Dim x As Byte

      'take the hash of password (this is what is supposed to be stored)
      s=md5(DEVICE_PASSWORD,"",MD5_FINISH,Len(DEVICE_PASSWORD))
      check=md5(chr(16)+s,"",MD5_FINISH,17)

      'see if this password is already set
      x=stor.base   'will restore the original value later
      stor.base=9   'this is where password data recides
      s2=stor.getdata(1,20)
      If asc(mid(s2,1,1))<>16 Or mid(s2,2,16)<>s Or mid(s2,18,3)<>check Then
            'password data in the EEPROM is incorrect! -- set the password
again
            stor.setdata(chr(16)+s+check,1)
            sys.reboot
      End If
      stor.base=x
      ...
```

Device Serial Number

Some Tibbo devices carry a unique serial number, and some devices even allow a portion of this serial number to be altered once.

To find out if your device contains a unique serial number and all the particulars regarding this number, refer to this device's platform documentation (for example, EM1000's is here 143).

To read the serial number of your device, use the sys.serialnum 534 R/O property. On devices without the serial number, this will return an empty string. On devices with the serial number, this will return the entire serial number including, if any, the programmable portion thereof.

use the sys.setserialnum 535 method to set the programmable portion of the serial number. If your device does not contain a user-programmable serial number field, this method will return 1- NG.

Be careful -- you can only set the programmable portion of the serial number **once**. There is no way to correct a mistake!

Flash Memory Configuration

Tibbo devices differ in their flash memory size and configuration. As Sharing Flash Between your Application and Data 237 explains, your device may *share* a single flash IC between the firmware, compiled Tibbo BASIC application, and application's data (accessible through the fd. 236 object). Alternatively, there may be a *dedicated* flash IC for firmware/application, and another IC for the fd. object's data. This second IC may not be present on the device itself.

To find out what flash memory arrangement your device utilizes, refer to your device's platform documentation (for example, EM1000's is here 143).

Status LEDs

BASIC-programmable devices supplied by Tibbo have one or two pairs of LEDs (or lines to control them).

Devices with 2 pairs of LEDs

On such devices, the first LED pair is comprised of green (SG) and red (SR) status LEDs, the second pair -- of green (EG) and yellow (EY) Ethernet LEDs.

Status LEDs have multiple functions:

- When the device is in the firmware upgrade mode, these LEDs indicate the status of the firmware upload process.
- When the device is under TiOS firmware control and Tibbo BASIC application is not running, these LEDs show current Tibbo BASIC application status.
- When the Tibbo BASIC application is running, status LEDs are under the control of the pat. 363 object.

The following table summarizes predefined status LED patterns:

Firmware upgrade mode		
	Green LED blinks slowly	File upload completed successfully.
	One long and one short "blink" of red LED	Communications error encountered during the serial file transfer.
	One long and two short "blinks" of red LED	FLASH memory failure.
Normal operation, Tibbo BASIC application not running		
	Fast-blinking GRGRGR... pattern	TiOS firmware not loaded or corrupted.
	Fast-blinking BBBB... pattern (B= red and green together)	Tibbo BASIC application loaded but cannot run due to insufficient variable (RAM) memory
	Fast-blinking G-G-G-... pattern	Tibbo BASIC application loaded but not running.
	Fast-blinking R-R-R-... pattern	Tibbo BASIC application not loaded or corrupted.

Ethernet LEDs indicate the following:

- Link/Data LED (green) is turned on when live Ethernet cable is plugged into the device. The LED blinks whenever an Ethernet packet is received.
- 100BaseT LED (yellow) is turned on when the device links with the hub at 100Mb. The LED is off when the link is established at 10Mb.

Devices with a single LED pair

On such devices, there are green (SG) and red (SR) status LEDs (or lines to control them). They function just as described above, but with one caveat: the brightness of these LEDs is indicative of the current Ethernet link state. When live Ethernet cable is not plugged into the device, the LEDs "play" patterns at a reduced brightness. When live Ethernet cable is plugged into the device, the LEDs "play" patterns at full brightness.

On devices with a single LED pair, there is no indication for the 100BaseT/10BaseT connection mode.

Some such devices feature a single third LED (or line to control it). This LED indicates current Ethernet link status: it will be on when live Ethernet cable is plugged.

Setup (MD) Button (Line)

Tibbo boards* and external controllers (such as the DS1206 186) have a button called "setup" or "MD" button ("MD" abbreviation stands for "mode"). Tibbo modules (such as the EM1000 143) have an MD pin for connecting an external button.

The setup button (line) has three functions:

- When a Tibbo BASIC application is running, it can use the button for its own purposes (see the button. 234 object).
- When the device is password-protected 41, keeping the button pressed while accessing the device from TIDE allows to bypass the password. This is the way to reset the password on the device.
- When the device is powered up (exits from the hardware reset) with the button pressed (line pulled low), it enters a firmware upgrade mode in which new TiOS firmware, possibly with compiled Tibbo BASIC application attached, can be uploaded into the device. If the device is powered up with the setup button not pressed (line not pulled low), the device starts the execution of the TiOS firmware (if loaded).

** This only applies to boards that carry one of Tibbo modules or directly incorporate BASIC-programmable hardware (such as the T1000 ASIC).*

Connecting GA1000

GA1000 is an add-on Wi-Fi module implementing 802.11b/g protocols (for documentation see *Programmable Hardware Manual*). This add-on device cannot work by itself and requires "external brains" in the form of one of Tibbo devices.

GA1000 interface

The GA1000 communicates with Tibbo devices through an SPI interface. Your device will control the GA1000 through five GPIO lines:

- **CS** -- SPI bus, chip select (active low);
- **CLK** -- SPI bus, clock;
- **DI** -- SPI bus, data in (must be connected to the GA1000's DO);
- **DO** -- SPI bus, data out (must be connected to the GA1000's DI);
- **RST** -- reset (active low). This line can be eliminated -- see below for details.

On platforms with unidirectional GPIOs lines, do not forget to configure CS, CLK, DO, and RST as outputs. DI must be configured as input. The wln. object won't do this automatically.

See Configuring Interface Lines topic of the wln. object's manual for more information on proper selection and configuration of GPIO lines on various devices (platforms).

Providing hardware reset

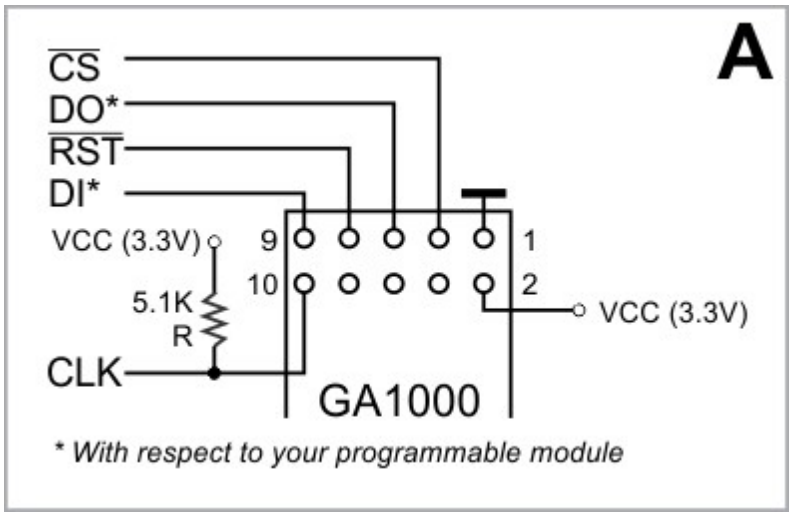
The wln. object directly controls CS, CLK, DI, and DO lines. Your application, however, must take care of the proper hardware reset for the GA1000. There are two methods for doing this:

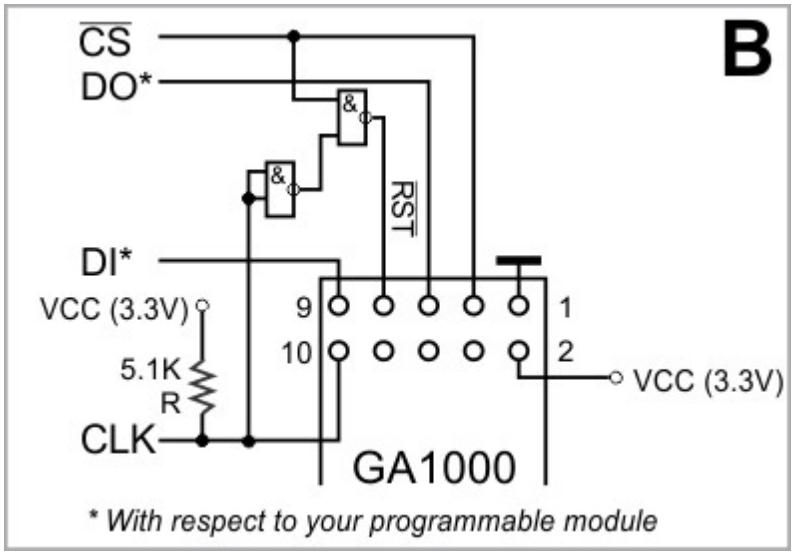
- Use a dedicated GPIO line to act as the RST line of the GA1000 interface (shown on diagram A below):

```
io.lineset(WLN_RST,LOW) 'apply reset (assuming that WLN_RST is a constant
that defines which GPIO line is connected to the GA1000's RST)
io.lineset(WLN_RST,HIGH) 'remove reset
```

- Use 2 NAND gates to combine CS and CLK signals and produce the reset signal for the GA1000 (shown on diagram B). This approach takes advantage of the fact that during SPI communications, CLK line will never be LOW while the CS line is HIGH. Schematic diagram on figure B generates reset when CS=HIGH and CLK=LOW. This way you save one GPIO line of your programmable module. Here is the code that will reset the GA1000:

```
io.lineset(WLN_CS,HIGH) 'set the CS line HIGH
io.lineset(WLN_CLK,LOW) 'apply reset
io.lineset(WLN_CLK,HIGH) 'remove reset
```

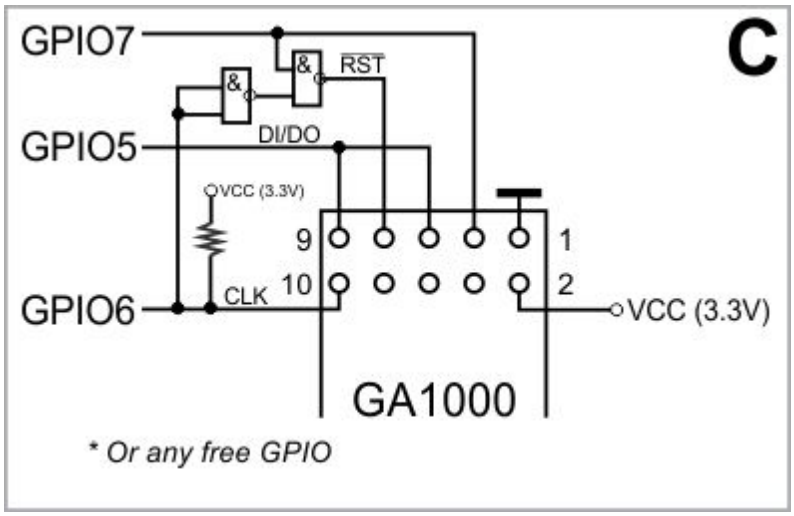




Tibbo devices differ in whether the CS, CLK, DI, and DO lines are remappable. On devices with remappable lines, you can choose any set of GPIOs to control the GA1000. On devices where remapping isn't provided, you just have to use "prescribed" GPIO lines. For information on this, see *Miscellaneous information* section of your platform's documentation (for example, EM1000W's is here 143).

Special case -- the EM500

Diagram C shows the recommended way of connecting the GA1000 to the EM500 138 module. GPIO lines are a precious commodity on the EM500 -- there are only eight of them available. As seen on the diagram C, you can get away with using only three lines to control the GA1000 (against the standard five lines). One line is saved by producing the reset out of CS and CLK lines. The second line is saved because EM500's bidirectional GPIOs 194 allow interconnecting DI and DO. The EM500 does not allow remapping of GA1000 lines, so GPIO line assignment shown below cannot be changed.



Debug Communications

When designing the debug communications for exchanging debug information between the TIDE and the target, we had two important goals:

- Allow maximum flexibility for your application and occupy minimum resources on the target.
- Allow for fast and efficient debug communications.

The resulting debug communications system:

- Does not even require a "proper" IP address on the target side.
- Allows you to freely change the IP address of the target while debugging.
- "Occupies" a single UDP port (65535) on the target, and even this port can be used by your program in most cases.

Your PC running TIDE sends debug messages as UDP datagrams, to target's port 65535. These messages include the MAC address of the target on which you are debugging your application.

The UDP port 65535 can still be used by your BASIC application. The target recognizes a datagram received on this port as a debug command only if this datagram starts with an underscore (_).

UDP datagrams received on UDP port 65535 that do not start with an underscore are not interpreted as debug commands by the target. Such datagrams are sent to your application for processing.

There are two slightly different communication modes for Tibbo devices. You can select them in the Project Settings 38 dialog, from the **transport** drop-down list.

UDP broadcast transport

With this selection, debug commands are sent through the Ethernet network as UDP broadcasts. They are, therefore, received by all devices on your local network

segment. Only the addressed Tibbo device will respond to the command. This is because each debug command contains a field specifying the MAC address of the target device. This method unnecessarily burdens all devices on the local network segment with having to "look at" all debug commands being transmitted. Also, some network equipment limits the amount of UDP broadcasts that can pass through it in one second. This may slow down your debug comms with the target.

WinPCap transport

With this selection, debug commands are sent through the Ethernet network as unicast UDP datagrams -- the target device is addressed directly. This method is the **preferred** way of communicating with Tibbo devices. The method relies on a WinPCap library that must be installed together with TIDE software. You must have seen the request for this during TIDE installation. With WinPCap transport, only the target device receives the debugging commands, and there is no issue with slow comms because of restrictive network equipment.

In most cases, debugging commands sent by TIDE cannot go across gateways (routers). This means the target and TIDE must reside on the same network segment -- remote debugging is not possible.

Also, debugging is not possible via Wi-Fi ports of Tibbo devices.

Serial Channels vs. Serial Ports

Enter topic text here.

Function Reference

Function reference section is a repository for all functions (syscalls) that have ever been created. The platform you are working with does not necessarily support every function. See your platform specifications 138 - you will find the list of support functions there.

Aes128dec Function

Function:	Encrypts data in 16-byte blocks according to the AES128 algorithm.
Syntax:	**aes128enc(byref** key **as string, byref** plain **as string) as string**
Returns:	Encrypted data (which will consist of complete 16-character blocks).
See Also:	aes128enc 206, rc4 222

Part	Description
key	Encryption key. Must be 16 characters long, or NULL string will be returned.
plain	Plain (unencrypted) data. Will be processed in 16-byte

blocks. Last incomplete block will be padded with zeroes.

Details

Aes128enc Function

Function:	Decrypts data in 16-byte blocks according to the AES128 algorithm.
Syntax:	**aes128dec(byref** key **as string, byref** cypher **as string) as string**
Returns:	Decrypted data (which will consist of complete 16-character blocks).
See Also:	aes128dec 205, rc4 222

Part	Description
key	Encryption key. Must be 16 characters long, or NULL string will be returned.
cypher	Encrypted data, must consist of one or more complete 16-character blocks, or NULL string will be returned.

Details

Asc Function

Function:	Returns the ASCII code of the leftmost character of the string.
Syntax:	**asc(byref** sourcestr **as string) as byte**
See Also:	Chr 208

Part	Description
sourcestr	Input string; the function will return ASCII code of the leftmost character of this string.

Details

Examples

```
x = asc("123") ' result will be 49 (ASCII code of '1')
```

Bin Function

Function: Converts unsigned 16-bit numeric value (word) into its binary string representation.

Syntax: **bin(**num **as integer) as string**

See Also: str 225, lstr 216, stri 226, lstri 217, lbin 214, hex 212, lhex 216, val 229, lval 218

Part	Description
num	Value to convert.

Details

Standard "&b" prefix is added at the beginning of the string.

Examples

```
dim s as string

s = bin(34) ' result will be '&b100010'
```

Cfloat Function

Function: Verifies whether the value of a floating-point variable is valid. Returns 0- VALID if the floating point value is OK, and 1- INVALID if the floating-point value is invalid.

Syntax: **cfloat(byref** num **as real) as valid_invalid**

See Also: strtof 228, ftostr 211

Part	Description
num	Variable to check.

Details

Floating-point calculations can lead to invalid result (#INF, -#INF errors, as per IEEE specification). When your application is in the debug 27 mode you will get a

FPERR exception 29 if such an error is encountered. In the release mode the Virtual Machine won't generate an exception, yet your application may need to know if a certain floating-point variable contains correct value. This is where cfloat function comes handy.

Examples

```
dim r1 as real
dim v as invalid_valid

dim r1=10E30
v=cfloat(r1) 'v will return 0- VALID
r1=r1*10E20 'at this point you will get FPERR exception if you are in the
debug mode
v=cfloat(r1) 'v will return 1- INVALID
```

Chr Function

Function: Returns the string that consists of a single character with ASCII code asciicode.

Syntax: **chr(**asciicode **as byte) as string**

See Also: Asc 206

Part	Description
asciicode	ASCII code of the character to return.

Details

It is also possible to use this function within string definitions, as shown in the example below.

Examples

```
dim x as byte
dim s as string

x=49
s = chr(x) ' result will be '1'
s = "FooBar" + chr(13) ' would add a carriage return to the end of the
string
```

Date Function

Function: Returns the date for a given day number.

Syntax: **date(**daycount **as word) as byte**

Returns:	Date in the 1-31 range.
See Also:	year 230, month 221, weekday 230, daycount 209, hours 213, minutes 221, mincount 220

Part	Description
daycount	Day number. Base date for the day count is 1-JAN-2000 (this is day #0).

Details

Examples

```
b = date(366) ' result will be 1 -- because day 366 is actually January 1st, 2001.
```

Daycount Function

Function:	Returns the day number for a given year, month, and date.
Syntax:	**daycount(**year **as byte,** month **as byte,** date **as byte) as word**
Returns:	Day number elapsed since 1-JAN-2000 (this is day #0). The range is 0-36524.
See Also:	year 230, month 221, date 208, weekday 230, hours 213, minutes 221, mincount 220

Part	Description
year	The year is supplied as offset from year 2000 (so, it is 6 for year 2006). Acceptable year range is 0-99 (2000-2099).
month	1-12 for January-December
date	Date of the month

Details

If any input parameter is illegal (year exceeds 99, month exceeds 12, etc.) this syscall will return 65535. This error value cannot be confused with an actual valid day number since the maximum day number recognized by this syscall is 12-DEC-2099 (day number 36524).

Examples

```
w = daycount(06, 10, 15) ' result will be 2479 (the serial day number for
October 15th, 2006)
```

Ddstr Function

Function: Converts "dot-decimal value" into "dot-decimal string".

Syntax: **ddstr(byref** str **as string) as string**

Returns: A dot-separated string consisting of decimal representations of all binary values in the input string. Each decimal value will be in the 0-255 range.

See Also: ddval 210

Part	Description
str	String of binary values to be converted into a dot-decimal string.

Details

This function is convenient for converting groups of bytes representing binary data (such as IP or MAC addresses) into their string representation.

Examples

```
dim s as string

s = chr(192)+chr(168)+chr(100)+chr(40) 'produce a string that contains these
values: 192,168,100,40
s = ddstr(s) 'now s will be equal to "192.168.100.40"
```

Ddval Function

Function: Converts "dot-decimal string" into "dot-decimal value".

Syntax: **ddval (byref** str **as string) as string**

Returns: A string of binary values.

See Also: ddstr 210

Part	Description
str	Dot-decimal string to be converted into a string of binary values. This string should comprise one or more dot-separated decimal values in the 0-255 range. Values that exceed 255 will produce an overflow, so result will be incorrect. If any other character other than "0"-"9" or "." is encountered then all digits after this character and up to

the next "." (if any) will be ignored. Leading spaces before each decimal value are allowed.

Details

This function is convenient for converting string representation of groups of bytes (such as IP or MAC addresses) into their binary form.

Examples

```
dim s as string

s = "192_3.1254.. 30" 'One value has invalid character in it ("_") and "3"
after this character will be ignored. Dot-decimal string. Another value --
1254 -- is out of range. Yet another value is missing and will be replaced
with 0.
s = ddstr(s) 'now s will contain these values: 192, 230, 0, 30.
```

Ftostr Function

Function: Converts real value into its string representation.

Syntax: **ftostr(byref** num **as real,** mode **as ftostr_mode,** rnd **as byte) as string**

See Also: strtof 228, cfloat 207, str 225, val 229

Part	Description
num	Real value to convert.
mode	Desired output format: 0- FTOSTR_MODE_AUTO: Choose between plain and mantissa/exponent format automatically. If mantissa/exponent format results in shorter string it will be used, otherwise plain format will be used. 1- FTOSTR_MODE_ME: Use mantissa/exponent format. 2- FTOSTR_MODE_PLAIN: Use plain format, not mantissa/exponent representation.
rnd	Number of digits to round the result to (total number of non-zero digits in the integer and fractional part of mantissa).

Details

Ftostr function offers much more control over the format of the output string compared to similar functions found on other systems. For starters, you can select whether you want to see mantissa/exponent, "regular" format, or let the function decide which format to use. Additionally, you control the rounding i.e. get to choose how many digits should be displayed -- and this influences the

representation both of the fractional and integer part of the value.

Examples below illustrate what you can do with ftostr. The ftostr has a counterpart -- fstr -- which is invoked implicitly whenever you assign a real to a string (string=real). Fstr is just like ftostr but mode and rnd parameters are fixed at 0-FTOSTR_MODE_AUTO and "maximum number of digits possible".

Examples

```
dim r1 as real
dim s as string

'demonstrate output formats
r1=10000000000.0 'notice '.0' -- it is necessary or compilier will generate
an error
s=ftostr(r1,FTOSTR_MODE_ME,11) 'result will be '1E+010'
s=ftostr(r1,FTOSTR_MODE_PLAIN,11) 'result will be '10000000000'
s=ftostr(r1,FTOSTR_MODE_AUTO,11) 'result will be '1E+010' because this
representation is more
compact

'demonstrate rounding
r1=1234567.125
s=ftostr(r1,FTOSTR_MODE_AUTO,15) 'result will be '1234567.125'
s=ftostr(r1,FTOSTR_MODE_AUTO,9) 'result will be '1234567.13'
s=ftostr(r1,FTOSTR_MODE_AUTO,2) 'result will be '1200000'

s=r1 'fstr will be used, result will be '1234567.125'
```

Hex Function

Function: Converts unsigned 16-bit numeric value (word) into its HEX string representation.

Syntax: **hex(**num **as integer) as string**

See Also: str 225, lstr 216, stri 226, lstri 217, bin 207, lbin 214, lhex 216, val 229, lval 218

Part	Description
num	Value to convert.

Details

Standard "&h" prefix is added at the beginning of the string.

Examples

```
dim s as string

s = hex(34) 'result will be '&h22'
```

Hours Function

Function:	Returns the hours value for a given minutes number.
Syntax:	**hours (**mincount **as word) as byte**
Returns:	Hours in the 0-23 range.
See Also:	year 230, month 221, date 208, weekday 230, daycount 209, minutes 221, mincount 220

Part	Description
mincount	The number of minutes elapsed since midnight (00:00 is minute #0). Maximum mincount number is 1439 (23:59).

Details

If a value higher than 1439 is supplied, this call will return 255. This error value cannot be confused with valid output since normal hours value cannot exceed 23.

Examples

```
b = hours(620) ' result will be 10 (because this minute number is falls between 10:00 and 11:00)
```

.Insert Function

Function:	Inserts insert_str string into the dest_str string at the insert position pos. Returns the new length of dest_str.
Syntax:	**insert(byref** dest_str **as string,** pos **as byte,byref** insert_str **as string) as byte**
See Also:	---

Part	Description
dest_str	The string to insert into.
pos	Insert position in the dest_str.
insert_str	The string to insert.

Details

This is an insert with overwrite, meaning that the insert_str will overwrite a portion of the dest_str.

Dest_str length can increase as a result of this operation (but not beyond declared string capacity). This will happen if the insertion position does not allow the

source_str to fit within the current length of the dest_string.

Examples

```
s = "FLIGHT XXX STATUS"
insert(s,8,"123") 's will now be 'FLIGHT 123 STATUS'
```

Instr Function

Function: Finds the Nth occurrence (defined by num, counting from 1) of a substring substr in a string sourcestr. Search is conducted from position frompos (leftmost character has position 1).

Syntax: **instr(**frompos **as byte,byref** sourcestr **as string, byref** substr **as string,** num **as byte) as byte**

See Also: ---

Part	Description
frompos	Position in the sourcestr from which to start searching. Leftmost character has position 1.
sourcestr	Source string in which the substring is to be found.
substr	Substring to search for within the source string.
num	Occurrence number of the substr, counting from 1.

Details

This function returns position in a string or zero if the Nth occurrence of the substring is not found.

Examples

```
x = instr(3,"ABCABCDEABC12","BC",2) ' result will be 10
```

Lbin Function

Function: Converts unsigned 32-bit numeric value (dword) into its binary string representation.

Syntax: **lbin(byref** num **as dword) as string**

See Also: str 225, lstr 216, stri 226, lstri 217, bin 207, hex 212, lhex 216, val 229, lval 218

Part	Description
num	Value to convert.

Details

Standard "&b" prefix is added at the beginning of the string.

Examples

```
dim s as string

s = lbin(2863311530) ' result will be '&b10101010101010101010101010101010'
```

Left Function

Function: Returns len leftmost characters of a string sourcestring.

Syntax: **left(byref** sourcestr **as string,** len **as byte) as string**

See Also: Right 223, Mid 219

Part	Description
sourcestr	String from which to take *len* leftmost characters.
len	Number of characters to take.

Details

Examples

```
ss = left("ABCDE",3) ' result will be 'ABC
```

Len Function

Function: Returns the length of (number of characters in) the string sourcestr.

Syntax: **len(byref** sourcestr **as string) as byte**

See Also: ---

Part	Description
sourcestr	String whose length needs to be calculated.

Details

Examples

```
x = len("ABC") ' result will be 3
```

Lhex Function

Function: Converts unsigned 32-bit numeric value (dword) into its HEX string representation.

Syntax: **lhex(byref** num **as dword) as string**

See Also: str 225, lstr 216, stri 226, lstri 217, bin 207, lbin 214, hex 212, val 229, lval 218

Part	Description
num	Value to convert.

Details

Standard "&h" prefix is added at the beginning of the string.

Examples

```
dim s as string

s = lhex(65536) 'result will be '&h10000'
```

Lstr Function

Function: Converts unsigned 32-bit numeric value (dword) into its decimal string representation.

Syntax: **lstr(byref** num **as dword) as string**

See Also: str 225, stri 226, lstri 217, bin 207, lbin 214, hex 212, lhex 216, val 229, lval 218

Part	Description
num	Value to be converted to string.

Details

Can be invoked implicitly, through the *string_var*= *dword_var* expression (see example below). Compiler is smart enough to pre-calculate constant-only expressions involving implicit use of lstr function.

Examples

```
dim d as dword
dim s as string

d1=123456
s = lstr(d) 'explicit invocation. Result will be '123456'
s = d 'implicit invokation
s = 666666 'will be calculated at compilation -- no actual lstr function
invokation will be here
```

Lstri Function

Function: Converts signed 32-bit numeric value (long) into its decimal string representation.

Syntax: **lstri(byref** num **as long) as string**

See Also: str 225, lstr 216, stri 226, bin 207, lbin 214, hex 212, lhex 216, val 229, lval 218

Part	Description
num	Value to be converted to string.

Details

Can be invoked implicitly, through the *string_var*= *long_var* expression (see example below). Compiler is smart enough to pre-calculate constant-only expressions involving implicit use of lstri function.

Examples

```
dim l as long
dim s as string

l= -5123
s = lstri(l) 'explicit invocation.
s = l 'implicit invocation
s = lstri(-20) 'will be calculated at compilation -- no actual lstri
function invokation will be here
```

Lval Function

Function: Converts string representation of a value into 32-bit value (dword or long).

Syntax: **lval(byref** sourcestr **as string) as dword**

See Also: str 225, lstr 216, stri 226, lstri 217, bin 207, lbin 214, hex 212, lhex 216, val 229

Part	Description
sourcestr	String to convert.

Details

Recognizes &b (binary) and &h (hexadecimal) prefixes. Can be invoked implicitly, through the d*word_var*= *string_var* expression (see example below). Compiler is smart enough to pre-calculate constant-only expressions involving implicit use of lval function.

Examples

```
dim d as word
dim l as long
dim s as string

s = "4294967295"
d = lval(s) 'explicit invocation, result will be 4294967295
l = lval(s) '
d = s 'implicit invocation, result will be -1 (4294967295 -> &hFFFFFFFF -> -
1 for signed variable)
d = "2402" 'be calculated at compilation -- no actual lval function
invokation will be here
```

Md5 Function

Function: Generates MD5 hash on the str string.

Syntax: **md5(byref** str **as string,byref** input_hash **as string,** md5_mode **as md5_modes,**total_len **as word) as string**

Returns: 16-character hash string; an empty string when invalid str or input_hash argument was detected

See Also: sha1 223

Part	Description
str	String containing (the next portion of) the input data to generate MD5 hash on. When md5_mode= 0- MD5_UPDATE,

	this string must be 64, 128, or 192 characters in length. Any other length will result in error and the function will return an empty string. When md5_mode= 1- MD5_FINISH, this string can have any length (up to 255 bytes).
input_hash	Hash obtained as a result of MD5 calculation on the previous data portion. Leave it empty for the first portion of data. Use the result of MD5 calculation on the previous data portion for the second and all subsequent portions of data. The result of MD5 is always 16 characters long, so passing the string of any other length (except 0 -- see above) will result in error and this function will return an empty string.
md5_mode	0- MD5_UPDATE : Set this mode for all data portions except the last one. 1- MD5_FINISH: Set this mode for the last data portion; also use this selection if you only have a single data portion.
total_len	Total length of processed data (in all data portions combined). Only relevant when md5_mode= 1- MD5_FINISH. That is, only relevant for the last or a single data portion.

Details

MD5 is a standard method of calculating hash codes on data of any size. The amount of input data can often exceed maximum capacity of string variables (255 characters). The md5 method can be invoked repeatedly in order to process the data of any size (see the example below).

Examples

```
Dim s, hash As String

'simple calculation on a short string
s="Sting to calculate MD5 on"
hash=md5(s,"",MD5_FINISH,Len(s))

'calculation on the entire contents of data in the 'text.txt 'file
romfile.open("text.txt")
hash=""
s=romfile.getdata(192) 'use max portions
While Len(s)=192
   hash=md5(s,hash,MD5_UPDATE,0)
   s=romfile.getdata(192)
Wend
hash=md5(s,hash,MD5_FINISH,romfile.size) 'last portion for whatever
unprocessed data is remaining in the file
```

Mid Function

Function:	Returns len characters from a string sourcestr starting from position pos.
Syntax:	**mid(byref** sourcestr **as string,** frompos **as byte,** len **as byte) as string**

See Also: Left 215, Right 223

Part	Description
sourcestr	String from which to take the middle section.
frompos	First character to take. The leftmost character is counted to be at position 1.
len	Number of characters to take.

Details

Examples

```
s = mid("ABCDE",2,3) ' result will be 'BCD'.
```

Mincount Function

Function: Returns the minutes number for a given hours and minutes.

Syntax: **mincount(**hours **as byte,** minutes **as byte) as word**

Returns: Minute number elapsed since midnight (00:00). This value is in the 0-1439 range.

See Also: year 230, month 221, date 208, weekday 230, daycount 209, hours 213, minutes 221

Part	Description
hours	An hour value, from 0 to 23.
minutes	A minute value, from 0 to 59.

Details

If any input parameter is illegal (hours exceeds 23, minutes exceeds 59, etc.) this syscall will return 65535. This error value cannot be confused with an actual valid minute number since the maximum minute number cannot exceed 1439.

Examples

```
w = mincount(14, 00) ' result will be 840
```

Minutes Function

Function:	Returns the minutes value for given minutes number.
Syntax:	**minutes(**mincount **as word) as byte**
Returns:	Minutes in the 0-59 range.
See Also:	year 230, month 221, date 208, weekday 230, daycount 209, hours 213, mincount 220

Part	Description
mincount	The number of minutes elapsed since midnight (00:00 is minute #0).

Details

If a value higher than 1439 is supplied, this call will return 255. This error value cannot be confused with valid output since normal minutes value cannot exceed 59.

Examples

```
b = minutes(61) ' result will be 1 - this is the time 01:01.
```

Month Function

Function:	Returns the month for a given day number.
Syntax:	**month(**daycount **as word) as pl_months**
Returns:	One of pl_months constants: 1- PL_MONTH_JANUARY: January. 2- PL_MONTH_FEBRUARY: February. 3- PL_MONTH_MARCH: March. 4- PL_MONTH_APRIL: April. 5- PL_MONTH_MAY: May. 6- PL_MONTH_JUNE: June. 7- PL_MONTH_JULY: July. 8- PL_MONTH_AUGUST: August. 9- PL_MONTH_SEPTEMBER: September. 10- PL_MONTH_OCTOBER: October. 11- PL_MONTH_NOVEMBER: November. 12- PL_MONTH_DECEMBER: December.
See Also:	year 230, date 208, weekday 230, daycount 209, hours 213,

minutes 221, mincount 220

Part	Description
daycount	Day number. Base date for the day count is 1-JAN-2000 (this is day #0).

Details

Examples

```
dim m as pl_months
m = month(32)   ' result will be PL_MONTH_FEBRUARY - day number 32 was in
February 2000.
```

Random Function

Function: Generates a string consisting of len random characters.

Syntax: **random(**len **as byte) as string**

See Also: ---

Part	Description
len	Length of the string to generate.

Details

Rc4 Function

Function: Encrypts/decrypts the data stream according to the RC4 algorithm.

Syntax: **rc4(byref** key **as string,** skip **as word, byref** data **as string) as string**

Returns: Processed data.

See Also: aes128enc 206, aes128dec 205

Part	Description
key	Encryption key, can have any length.

skip	The number of "skip" iterations. These are additional iterations added past the standard "key scheduling algorithm". Set this argument to 0 to obtain standard encryption results compatible with other systems.
data	Data to encrypt/decrypt.

Details

With RC4 algorithm, the same function is used both for encrypting and decrypting the data.

Right Function

Function: Returns len rightmost characters of a string sourcestr.

Syntax: **right(byref** sourcestr **as string,** len **as byte) as string**

See Also: Left 215, Mid 219

Part	Description
sourcestr	String from which to take *len* rightmost characters.
len	Number of characters to take.

Details

Examples

```
s = right("ABCDE",3) ' result will be 'CDE'
```

Sha1 Function

Function: Generates SHA1 hash on the str string.

Syntax: **sha1(byref** str **as string,byref** input_hash **as string,** sha1_mode **as sha1_modes,**total_len **as word) as string**

Returns: 20-character hash string; an empty string when invalid str or input_hash argument was detected

See Also: md5 218

Part	Description
str	String containing (the next portion of) the input data to

	generate SHA1 hash on. When sha1_mode= 0- SHA1_UPDATE, this string must be 64, 128, or 192 characters in length. Any other length will result in error and the function will return an empty string. When sha1_mode= 1- SHA1_FINISH, this string can have any length (up to 255 bytes).
input_hash	Hash obtained as a result of SHA1 calculation on the previous data portion. Leave it empty for the first portion of data. Use the result of SHA1 calculation on the previous data portion for the second and all subsequent portions of data. The result of sha1 is always 20 characters long, so passing the string of any other length (except 0 -- see above) will result in error and this function will return an empty string.
sha1_mode	0- SHA1_UPDATE : Set this mode for all data portions except the last one. 1- SHA1_FINISH: Set this mode for the last data portion; also use this selection if you only have a single data portion.
total_len	Total length of processed data (in all data portions combined). Only relevant when sha1_mode= 1- SHA1_FINISH. That is, only relevant for the last or a single data portion.

Details

SHA1 is a standard method of calculating hash codes on data of any size. The amount of input data can often exceed maximum capacity of string variables (255 characters). The sha1 method can be invoked repeatedly in order to process the data of any size (see the example below).

Examples

```
Dim s, hash As String

'simple calculation on a short string
s="Sting to calculate SHA1 on"
hash=sha1(s,"",SHA1_FINISH,Len(s))

'calculation on the entire contents of data in the 'text.txt 'file
romfile.open("text.txt")
hash=""
s=romfile.getdata(192) 'use max portions
While Len(s)=192
   hash=sha1(s,hash,SHA1_UPDATE,0)
   s=romfile.getdata(192)
Wend
hash=sha1(s,hash,SHA1_FINISH,romfile.size) 'last portion for whatever
unprocessed data is remaining in the file
```

Str Function

Function: Converts unsigned 16-bit numeric value (word) into its decimal string representation.

Syntax: **str(**num **as word) as string**

See Also: lstr 216, stri 226, lstri 217, bin 207, lbin 214, hex 212, lhex 216, val 229, lval 218

Part	Description
num	Value to be converted to string.

Details

Can be invoked implicitly, through the *string_var*= *word_var* expression (see example below). Compiler is smart enough to pre-calculate constant-only expressions involving implicit use of str function.

Examples

```
dim w as word
dim s as string

w=3400
s = str(w) 'explicit invocation.
s = w 'implicit invocation
s = 100 'will be calculated at compilation -- no actual str function
invokation will be here
```

Strand Function

Function: Calculates logical AND on data in str1 and str2 arguments.

Syntax: **strand(byref** str1 **as string, byref** str2 **as string) as string**

Returns: Result of logical AND operation.

See Also: stror 227, strxor 228

Part	Description
str1	Argument 1.
str2	Argument 2.

Details

This function treats data in str1 and str2 as two byte arrays. Logical AND operation

is performed on corresponding byte pairs (first byte of str1 AND first byte of str2, etc.).

If one of the arguments contains less bytes, then this argument is padded with zeroes prior to performing logical AND operation.

Strgen Function

Function:	Generates a string of len length consisting of repeating substrings substr.
Syntax:	**strgen(**len **as byte,byref** substr **as string) as string**
See Also:	---

Part	Description
len	Length of the string to generate
substr	Substring that will be used (repeatedly) to generate the string

Details

Notice that len parameter specifies total resulting string length in bytes so the last substring will be truncated if necessary to achieve exact required length. This function is an expanded version of the STRING$ function commonly found in other BASICs.

Examples

```
string1 = strgen(10,"ABC") ' result will be 'ABCABCABCA'.
```

Stri Function

Function:	Converts signed 16-bit numeric value (short) into its decimal string representation.
Syntax:	**stri(**num **as integer) as string**
See Also:	str 225, lstr 216, lstri 217, bin 207, lbin 214, hex 212, lhex 216, val 229, lval 218

Part	Description
num	Value to be converted to string.

Details

Can be invoked implicitly, through the *string_var= short_var* expression (see example below). Compiler is smart enough to pre-calculate constant-only expressions involving implicit use of stri function.

Examples

```
dim sh as short
dim s as string

sh= -3400
s = stri(sh) 'explicit invocation.
s = sh 'implicit invocation
s = stri(-100) 'will be calculated at compilation -- no actual stri function
invokation will be here
```

Stror Function

Function:	Calculates logical OR on data in str1 and str2 arguments.
Syntax:	**stror(byref** str1 **as string, byref** str2 **as string) as string**
Returns:	Result of logical OR operation.
See Also:	strand 225, strxor 228

Part	Description
str1	Argument 1.
str2	Argument 2.

Details

This function treats data in str1 and str2 as two byte arrays. Logical OR operation is performed on corresponding byte pairs (first byte of str1 OR first byte of str2, etc.).

If one of the arguments contains less bytes, then this argument is padded with zeroes prior to performing logical OR operation.

Strsum Function

Function:	Calculates 16-bit (word) sum of ASCII codes of all characters in a string.
Syntax:	**strsum(byref** sourcestr **as string) as word**
See Also:	---

Part	Description
sourcestr	String to work on

Details

This function is useful for checksum calculation.

Examples

```
w = strsum("012")  ' will return 147 (48+49+50).
```

Strtof Function

Function: Converts string representation of a real value into a real value.

Syntax: **strtof(byref** str **as string) as real**

See Also: ftostr 211, str 225, val 229

Part	Description
str	String to convert.

Details

You must keep in mind that floating-point calculations are inherently imprecise. Not every value can be converted into its exact floating-point representation. Also, strtof can be invoked implicitly. Examples below illustrates this.

Examples

```
dim r1 as real
dim s as string

s="456.125"
r1=strtof(s) 'r1 will be equal to 456.125. This conversion will be done
without errors.
s="123.200"
r1=strtof(s) '123.200 will be converted with errors. Actual result will be
123.25.
r1=s 'implicit invocation. Same result as for the line above.
```

Strxor Function

Function: Calculates exclusive OR (XOR) on data in str1 and str2 arguments.

Syntax: **strxor(byref** str1 **as string, byref** str2 **as string) as string**

Returns: Result of logical XOR operation.

See Also: strand 225, stror 227

Part	Description
str1	Argument 1.
str2	Argument 2.

Details

This function treats data in str1 and str2 as two byte arrays. Logical XOR operation is performed on corresponding byte pairs (first byte of str1 XOR first byte of str2, etc.).

If one of the arguments contains less bytes, then this argument is padded with zeroes prior to performing logical XOR operation.

Val Function

Function: Converts string representation of a value into 16-bit value (word or short).

Syntax: **val** (**byref** sourcestr **as string**) **as word**

See Also: str 225, lstr 216, stri 226, lstri 217, bin 207, lbin 214, hex 212, lhex 216, lval 218

Part	Description
sourcestr	String to convert.

Details

Recognizes &b (binary) and &h (hexadecimal) prefixes. Can be invoked implicitly, through the *word_var*= *string_var* expression (see example below). Compiler is smart enough to pre-calculate constant-only expressions involving implicit use of val function.

Beginning with release **2.0**, this function also plays the role of vali function, which has been removed.

Examples

```
dim w as word
dim sh as short
dim s as string

s="&hF222"
w = val(s) 'explicit invocation, result will be 61986
sh= val(s) 'explicit invocation, result will be -3550 (sh is a 16-bit signed
variable)
w = s 'implicit invocation
w = "2402" 'be calculated at compilation -- no actual val function
invokation will be here
```

Vali Function

Vali function is no longer available. Use val 229 function both for unsigned (word) and signed (short) conversions.

Weekday Function

Function:	Returns the day of the week for a given day number.
Syntax:	**weekday(**daycount **as word) as pl_days_of_week**
Returns:	One of pl_days_of_week constants: 1- PL_DOW_MONDAY: Monday. 2- PL_DOW_TUESDAY: Tuesday. 3- PL_DOW_WEDNESDAY: Wednesday. 4- PL_DOW_THURSDAY: Thursday. 5- PL_DOW_FRIDAY: Friday. 6- PL_DOW_SATURDAY: Saturday. 7- PL_DOW_SUNDAY: Sunday.
See Also:	year 230, month 221, date 208, daycount 209, hours 213, minutes 221, mincount 220

Part	Description
daycount	Day number. Base date for the day count is 1-JAN-2000 (this is day #0).

Details

Examples

```
dim w as pl_days_of_week
w = weekday(0)  ' result will be  PL_DOW_SATURDAY - the was the day of the
week for the 1st of January 2000.
```

Year Function

Function:	Returns the year for a given day number.
Syntax:	**year(**daycount **as word) as byte**
Returns:	Two last digits of the year (0 means 2000, 1 means 2001, and so on.)

See Also: month 221, date 208, weekday 230, daycount 209, hours 213, minutes 221, mincount 220

Part	Description
daycount	Day number. Base date for the day count is 1-JAN-2000 (this is day #0).

Details

Examples

```
b = year(366) ' result will be 1 (this day number is in year 2001).
```

Object Reference

Object reference section is a repository for all objects that have ever been created. The platform you are working with does not necessarily support every object. See your platform specifications 138 - you will find the list of supported objects there.

Objects are...

- Beep 232 — generates buzzer patterns.
- Button 234 — monitors MD line (setup button).
- Fd 236 — manages flash memory file system and direct sector access.
- Io 294 — handles I/O lines, ports, and interrupts.
- Kp 304 — scans keypads of matrix and "binary" types.
- Lcd 317 — controls graphical display panels (several types supported).
- Net 358 — controls Ethernet port.
- Pat 363 — "plays" patterns on up to five LED pairs.
- Ppp 366 — accesses the Internet over the modem.
- Pppoe 369 — accesses the Internet over the ADSL modem.
- Romfile 370 — facilitates access to resource files (fixed data).
- Rtc 375 — keeps track of date and time.
- Ser 378 — up to 4 serial ports (UART, Wiegand, and clock/data modes).
- Sock 421 — socket comms (up to 16 UDP, TCP, and HTTP sessions).
- Ssi 512 — up to four serial synchronous interface channels (for SPI, I2C, etc.).
- Stor 522 — provides access to the EEPROM.
- Sys 526 — in charge of general device functionality.
- Wln 536 — handles Wi-Fi interface (requires GA1000 add-on module).

Beep Object

The beep object allows you to generate "beep" patters using the beeper (buzzer) attached to the CO pin of your device. "CO" stands for "clock output" and its output can be actually used for anything, not just controlling beeper. Frequency available on the CO pin is defined by the beep.divider 232 property.

When the pattern starts playing, the CO line becomes an output automatically. Therefore, you do not need to use the io.enabled 298 property to configure this line as output. When the pattern stops playing, the line will return to the input/output and HIGH/LOW state that it had before the pattern started playing.

The pattern you play can be up to 16 steps long. Each "step" can be either "-" (buzzer off), or "B" (buzzer on). You can also define whether the pattern will only execute one or loop and play indefinitely. Additionally, you can make the pattern play at "normal" or double speed.

You load the new pattern to play with the beep.play 233 method. If the pattern is looped it will continue playing until you changed it. If the pattern is not looped it will play once and then the on_beep 233 event will be generated.

Buzzer patters offer a convenient way to tell the user what your system is doing. You can devise different patterns for different states of your device. You can use these patterns together with LED patterns generated by the pat 363 object.

Here is a simple example in which we generate three beeps on power up of the device.

```
sub on_sys_init
     beep.play("B-B-B",PL_PAT_CANINT)
end sub
```

To obtain a permanent, non-stop square wave on CO pin load the following pattern:

```
beep.play("B~",PL_PAT_CANINT)
```

.Divider Property

Function:	Sets the frequency of the square wave output on the CO line.
Type:	Word
Value Range:	0-65535, **default**= 1 (divide by 2)
See Also:	---

Details

Actual frequency can be calculated as base_frq/(2*beep.divider). Setting this property to 0 is equivalent to 65536 (i.e. actual frequency will be

base_frq/131072).

Base_frq depends on your platform -- you will find this information in your device's platform documentation (for example, EM1000's is here[143]).

Most low-cost buzzers have resonant frequency at which they emit the loudest sound. Check the specifications for your buzzer and set the divider accordingly.

On_beep Event

Function: Generated when buzzer pattern finishes "playing".

Declaration: **on_beep**

See Also: Beep Object[232], beep.play[233]

Details

This can only happen for "non-looped" patterns. Multiple on_beep events may be waiting in the event queue.

.Play Method

Function: Loads new beeper pattern to play.

Syntax: **pat.play(byref** pattern **as string,** patint **as pl_beep_int)**

Returns: ---

See Also: Beep Object[232]

Part	Description
pattern	Pattern string, can include the following characters: '**-**': the buzzer is off 'B' or 'b': the buzzer is on '~': looped pattern (can reside anywhere in the pattern string) '*': double-speed pattern (can reside anywhere in the pattern string). New in **V1.1**: You can use this symbol twice and, thus, obtain x4 speed if necessary!
patint	Defines whether the beep.play method is allowed to interrupt another pattern that is already playing: 0- PL_BEEP_NOINT: cannot interrupt 1- PL_BEEP_CANINT: can interrupt)

Details

Maximum pattern length is 16 "steps". The on_pat[365] event is generated once the

pattern finishes playing (looped patterns never finish playing).

Button Object

All external devices and board offered by Tibbo feature a single button called "setup button". Our modules have a line to connect such a button externally. The setup button has a special system function: powering-up a Tibbo device with this button pressed (setup button line held LOW) causes the device to enter the serial upgrade mode. This mode is for uploading new TiOS 7 (firmware) file into the device through the serial port.

The button is not doing anything system-related at other times, so it can be used by your Tibbo BASIC application -- hence, the button object. The object offers the folloing:

The button.pressed 235 property returns current (immediate) state of the button.

The on_button_pressed 234 event is generated when the button is pressed. The on_button_released 235 event is generated when the button is released.

The button.time 235 read-only property returns the time (in 0.5 intervals) elapsed since the button was last pressed or released. You can use this property, for instance, to separate button pressing into "short" and "long":

```
sub on_button_released
'see how much time has elapsed

      if button.time>4 then
            'the button for pressed for a "long" time -- do one thing
      else
            'the button was pressed for a "short" time -- do another thing
      end if
end sub
```

The button does not requre any pre-configuration and works always.

Note that the on_button_pressed 234 and on_button_released 235 events, as well as the button.time 235 R/O property utilize "debouncing", which filters out very short transitions of the button state. The button.pressed 235 R/O property, however, does not rely on debouncing and returns the immediate state of the button at the very moment the property is read.

On_button_pressed Event

Function:	Generated when the button on your device is pressed.
Declaration:	**on_button_pressed**
See Also:	Button.pressed 235

Details

Multiple on_button_pressed events may be waiting in the event queue. You can check the time elapsed since the previous on_button_released 235 event by reading the value of the button.time 235 read-only property.

Note that the button object performs "debouncing" which rejects very brief transitions of the button state. This event will not be generated for such spurious transitions.

On_button_released Event

Function: Generated when the button on your device is released.

Declaration: **on_button_released**

See Also: Button.pressed 235

Details

Multiple on_button_released events may be waiting in the event queue. You can check the time elapsed since the previous on_button_pressed 234 event by reading the value of the button.time 235 read-only property.

Note that the button object performs "debouncing" which rejects very brief transitions of the button state. This event will not be generated for such spurious transitions.

.Pressed R/O Property

Function: Returns the current button state.

Type: Enum (no_yes, byte)

Value Range: 0- NO: the button is not pressed.
1- YES: the button is pressed.

See Also: ---

Details

This property reflects an immediate state of the hardware at the very moment the property is read -- no "debouncing" performed. This is different from the on_button_pressed 234 and on_button_released 235 events, as well as the button.time 235 R/O property, which all take debouncing into the account.

.Time R/O Property

Function: Returns the time (in 0.5 second intervals) elapsed since the button was last pressed or released (whichever happened) later.

Type: Byte

Value Range: 0-255

See Also: ---

Details

It only makes sense to read this property inside the on_button_pressed 234 or on_button_released 235 event handlers. Once the value of this property reaches 255 (127 seconds) it stays at 255 (there is no roll-over to 0). Elapsed time is not counted when the execution of your application is paused.

Fd Object

This is the flash disk (fd.) object, it allows you to use your device's flash memory for data storage. There are two methods of working with the flash memory:

- With direct sector access 240, you can write and read flash sectors directly, without burdening yourself with the file system.
- With file-based access 245, you create a formatted disk that stores files.

Both methods can be used concurrently 263 and complement each other whenever necessary.

Here is what the fd. object has to offer in terms of the file-based access:

- Ability to store up to 64 files located in a single root directory (subdirectories are not supported, but can be emulated 250).
- Flexible file attributes 249 -- define and store any attributes you like.
- Methods to work with the file directory 252.
- Ability to open 253 and work with several files at once.
- Methods to write to and read from 254 the file, also cut out a portion of the file from the beginning or end 255.
- Fast and flexible search 256 to locate data within files. This also includes a record-style search!
- Automatic sector leveling 264.
- Transactions 259 to ensure disk integrity in the toughest of conditions (power failures, etc).
- A method for firmware/application self-upgrades 243 (not supported by all platforms).

Overview 3.1

In this section:

- Sharing Flash Between Your Application and Data 237
- Fd. Object's Status Codes 239
- Direct Sector Access 240
- File-based Access 245

- File-based and Direct Sector Access Coexistence 263
- Prolonging Flash Memory Life 264

Sharing Flash Between Your Application and Data

Discussed in this section: fd.availableflashspace 267.

The flash memory used by the fd. object consists of 264-byte sectors. It is customary to use 256 bytes of each sector to store actual data, and reserve the rest for "service" data (checksum 242, etc.).

Depending on what device you are using, the fd. object may be storing its data in one of the two ways (locations):

- On devices with *shared* flash memory, the firmware, compiled Tibbo Basic application, and the fd. object's data are stored on the same flash IC chip. All flash sectors that are not occupied by firmware/application are available for storing your data. We will refer to this area as a **data area** of the flash IC. The size of the data area, in sectors, can be obtained through the fd.availableflashspace 267 R/O property.
- On devices with *dedicated* flash memory, one flash IC is used for firmware and application storage, while a second flash IC is used for fd. object's data storage. On such devices, the data area occupies the entire second flash IC, and fd.availableflashspace 267 always returns the number of sectors in this second flash IC. On certain device, the second flash IC is not present onboard and must be connected externally.

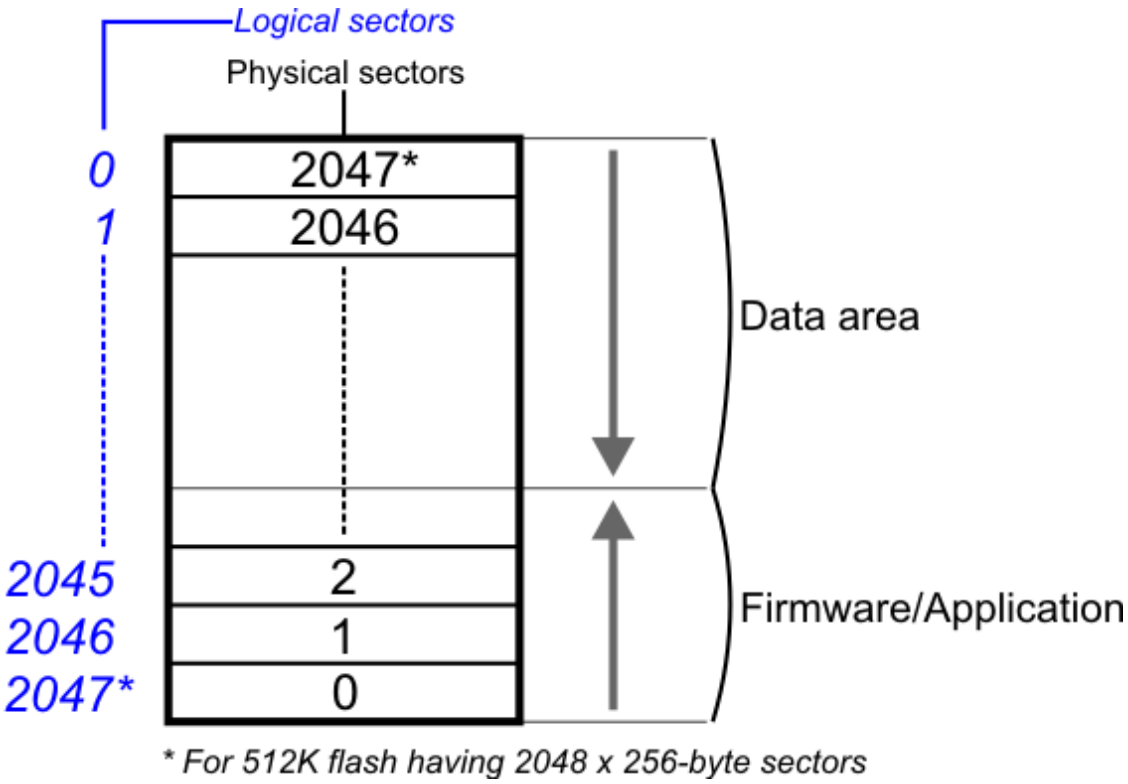
To find out what flash memory arrangement your device has, refer to your device's platform specification 138 (for example, EM1000's is here 143). You will find memory arrangement data under "Miscellaneous Information/ Flash memory configuration".

There are two ways in which you can store the data using the fd. object:

- The first way is to read and write flash sectors of the data area directly -- a so-called direct sector access 240. This method is simple and allows you to "do whatever you please" with the data area you have.
- The second way is file-based 245 -- you create a full-blown flash disk that maintains a simple file system. This introduces a certain overhead, since the file system needs a number of sectors for its own internal "housekeeping". At the same time, using files may be infinitely more convenient and flexible, as many complex operations happen automatically, thus simplifying your application. In addition, disk transactions 259 bulletproof your file operations.

Both the direct sector access and the file based access can be used at the same time 263 -- your flash disk can occupy only a portion of the data area, and the rest of the space can be used for direct sector access.

The fd. object refers to physical sectors of the data area in reverse (refer to the drawing below, which shows memory allocation on the shared flash IC). While the firmware and application are loaded into the flash memory starting from the physical sector 0, the allocation for the data area starts from the topmost physical sector. For convenience, related methods and properties of the .fd object refer to this topmost physical sector as the (logical) sector 0, the sector before the topmost sector -- as 1, and so on.



This approach was chosen to minimize the influence of the changing size of your Tibbo Basic application on the data you keep in the flash memory. Typically, the size of your application keeps changing as you develop and debug it. At the same time, you often need to keep certain data in the data area permanently, and don't want to recreate this data every time you upload a new iteration of your application. If the beginning of the data area was right after the end of the firmware/application area, any change in the application size would move the boundary of the data area, corrupt your data, and force you to recreate the data again. If, on the other hand, your data area starts from the top of the flash IC and continues downwards, then the chance of the data area corruption will be a lot smaller.

Naturally, the above applies to devices with *shared* flash memory. Devices with dedicated flash memory store firmware/application and fd. object's data in separate ICs, so there is no chance of corrupting the data area by loading a larger firmware/application.

- When debugging your application on devices with the shared flash memory, use only a portion of the data area and leave some part of this area unoccupied. This will create a gap of unused sectors between the data and the firmware/application areas of the flash. This way, your application will (mostly) be able to grow without corrupting the data area.

Fd. Object's Status Codes

Discussed in this topic: fd.laststatus 282.

Many things can go wrong when working with the flash memory. This is why methods of the fd. object return a status code. Good Tibbo Basic application doesn't just assume that all will be well and always checks the result of method execution.

Listed below are possible status codes retuned by the flash disk object. Of course, not every code can be generated by every method:

0- PL_FD_STATUS_OK: Completed successfully.

1- PL_FD_STATUS_FAIL : Physical flash memory failure (fatal: disk dismounted, must be reformatted).

2- PL_FD_STATUS_CHECKSUM_ERR: Checksum error has been detected in one of the disk sectors (fatal: disk dismounted, mast be reformatted).

3- PL_FD_STATUS_FORMAT_ERR: Disk formatting error has been detected (fatal: disk dismounted, mast be reformatted).

4- PL_FD_STATUS_INV_PARAM: Invalid argument have been provided for the invoked method.

5- PL_FD_STATUS_DUPLICATE_NAME: File with this name already exists.

6- PL_FD_STATUS_FILE_TABLE_FULL: Maximum number of files that can be stored on the disk has been reached, new file cannot be created.

7- PL_FD_STATUS_DATA_FULL: The disk is full, new data cannot be added.

8- PL_FD_STATUS_NOT_READY: The disk is not mounted.

9- PL_FD_STATUS_NOT_FOUND: File not found.

10- PL_FD_STATUS_NOT_OPENED: No file is currently opened "on" the current fd.filenum 273.

11- PL_FD_STATUS_ALREADY_OPENED: This file is already opened on some other file number.

12- PL_FD_STATUS_TRANSACTION_ALREADY_STARTED: Disk transaction has already been started (and cannot be restarted).

13- PL_FD_STATUS_TRANSACTION_NOT_YET_STARTED: Disk transaction hasn't been started yet.

14- PL_FD_STATUS_TRANSACTION_CAPACITY_EXCEEDED: Too many disk sectors have been modified in the cause of the current transaction (fatal: disk dismounted).

15- PL_FD_STATUS_TRANSACTIONS_NOT_SUPPORTED: The disk wasn't formatted to support transactions (use fd.formatj 277 with maxjournalsectors>1 to enable transactions).

16- PL_FD_STATUS_FLASH_NOT_DETECTED: Flash IC wasn't detected during boot, fd. object cannot operate normally.

The status code generated by the most recently invoked method is always kept by the fd.laststatus 282 R/O property. Some methods also return the status code directly:

```
If fd.create("File1.dat")<>PL_FD_STATUS_OK Then
   'some problem
End If
```

Other methods return data, so the only way to check the result of their execution is through the fd.laststatus:

```
s=fd.getdata(50) 'returns the data from the file, not the status code
If fd.laststatus<>PL_FD_STATUS_OK Then
   'some problem
End If
```

Most status codes are non-fatal, that is, they allow the disk to continue working.

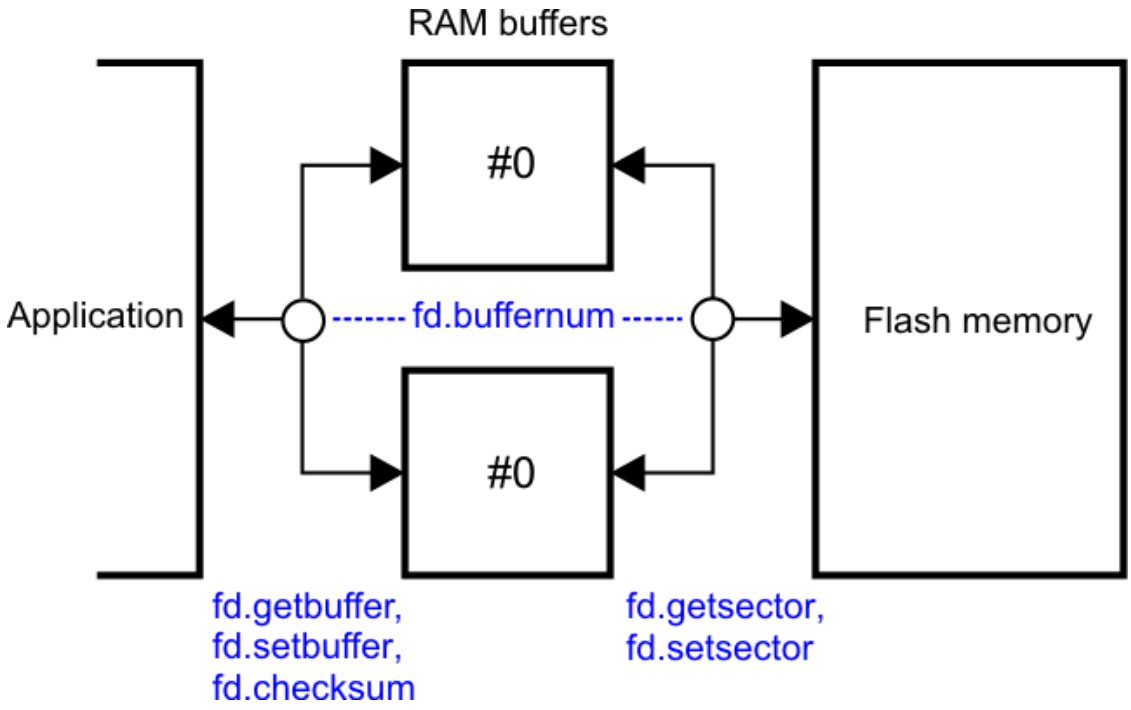
Selected status conditions result in the automatic dismounting[249] of the flash disk[245] (fd.ready[285] R/O property becomes 0- NO). Among these status codes, some (but not all) conditions also indicate that the disk is permanently damaged and must be reformatted[245].

Direct Sector Access

Discussed in this section: fd.buffernum[267], fd.getbuffer[278], fd.setbuffer[288], fd.getsector[281], fd.setsector[290], fd.checksum[268], fd.copyfirmware[269], fd.copyfirmwarelzo[270].

Direct sector access allows you to work with sectors in the data area[237] of your device's flash memory directly, without the need to create and manage any files. You work with sectors through two identical 264-byte RAM buffers numbered #0 and #1. The fd.buffernum[267] property selects one of the buffers as the source/destination for the data (see the drawing below).

All file-based[245] operations of the flash disk also rely on the RAM buffers and the selected buffer number may change as a result of their execution. When using direct sector access and file-based access concurrently[263], switch to the RAM buffer #0 *each time* before performing direct sector access -- this will guarantee that you won't corrupt the files and/or the file system and cause disk dismounting (fd.ready[285] becoming 0- NO). There is more on this in the File-based and Direct Sector Coexistence[263] topic.



Reading data from a sector is a two-step process. First, you use fd.getsector 281 to load all 264 bytes from the desired sector into the currently selected RAM buffer. Next, you use fd.getbuffer 278 to read the data from the selected buffer. This method allows you to read up to 255 bytes beginning from any offset in the buffer (offsets are counted from 0):

```
Dim s As String
...
'we want bytes 20-29 from sector #3
fd.buffernum=0 'select RAM buffer #0
If fd.getsector(3)<>PL_FD_STATUS_OK Then
   'flash failure
End If
s=fd.getbuffer(20,10) 'now s contains the desired data
```

Since the sector and (RAM buffer size) exceeds 255 bytes (maximum length of string variables), you can't actually read the whole sector contents in one portion. At least two fd.getbuffer reads are necessary for that.

To modify the data in the selected RAM buffer, use the fd.setbuffer 288 method. The fd.setbuffer allows you to write new data at any offset of the selected RAM buffer. To store the contents of the RAM buffer back to the flash memory, use fd.setsector 290:

```
Dim s As String
...
'modify first 3 bytes of sector #5
If fd.getsector(3)<>PL_FD_STATUS_OK Then
   'flash failure
End If
```

```
fd.setbuffer("ABC",0) 'write "ABC" to the buffer at offset 0
If fd.setsector(3)<>PL_FD_STATUS_OK Then
   'flash failure
End If
```

Since there are two identical RAM buffers, you can load the contents of two different sectors and work with the data concurrently, by switching between the buffers.

As covered under Sharing Flash Between Your Application and Data 237, logical sector numbers for fd.getsector and fd.setsector are not actual physical sector numbers of the flash IC. Logical sector #0 corresponds to the topmost physical sector of the flash IC, so logical numbering is "in reverse".

You can only write to sectors that reside within the data area of your flash chip (that is, logical sector numbers from 0 to fd.availableflashspace 267 - 1). For devices with shared flash memory, this prevents your application from inadvertently damaging its own code or firmware. Trespassing the data area boundary will result in the 4- PL_FD_STATUS_INV_PARAM status code 239. If you want to alter the data in the firmware/application area, see Upgrading the Firmware/Application 243 topic.

If you are using direct sector access and file-based access 245 at the same time, be sure to read about ensuring their proper coexistence 263.

Note also that fd.getsector and fd.setsector always access the actual specified target sector and not its cached copy even if the disk transaction 259 is in progress (fd.transactionstarted 294 = 1- YES) and the target sector has been cached already.

Using Checksums

When dealing with the flash memory, it is very often desirable to make sure that the data stored in flash sectors is not corrupted. One way to do so is by calculating the checksum on the sector data and storing this checksum together with the data. Each sector of the flash memory conveniently stores 8 extra bytes on top of the "regular" 256 bytes usually storing the actual data.

The fd. object uses two bytes at offsets 262 and 263 to store the checksum of the first 262 bytes of data residing in the same sector. The checksum is a 16-bit value. When the checksum is correct, modulo 16 sum of the entire sector's data is zero.

16-bit values for these calculations are little-endian. That is, offset 0 of the sector is presumed to be the high byte of the first 16-bit value, offset 1 -- low byte of the first 16-bit value, offset 2 -- high byte of the second 16-bit value, and so on.

When you are using the file-based access 245, the fd. object automatically calculates and/or verifies the checksum on all sectors it accesses. Should any sector turn out to contain an invalid checksum, the 2- PL_FD_STATUS_CHECKSUM_ERR status code 239 is generated.

Direct sector access is more primitive and it is your responsibility to maintain and verify the integrity of data you store in the flash memory. To aid you in this, the fd.checksum 268 method can be used to verify the checksum, or calculate and store it into the selected RAM buffer (where the sector's data is supposed to be already loaded):

```
'Load sector #10, verify its checksum, alter some data, recalculate the
checksum, and save new data
Dim i As word
...
'load the sector
If fd.getsector(10)<>PL_FD_STATUS_OK Then
```

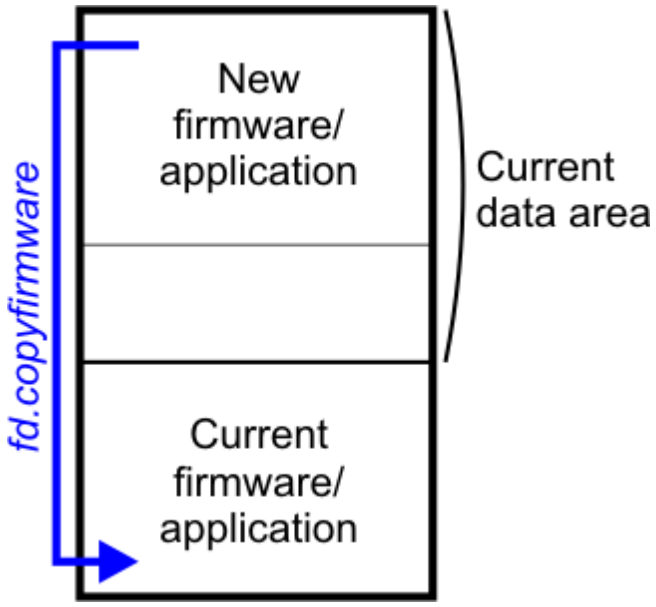
```
   'flash failure
End If

'verify the checksum
If fd.checksum(PL_FD_CSUM_MODE_VERIFY,i)<>OK Then
   'checksum error detected
End If

fd.setbuffer("ABC",20) 'alter data at offset 20

fd.checksum(PL_FD_CSUM_MODE_CALCULATE,i) 'recalculate the checksum, save it
back into the RAM buffer

'save back
If fd.setsector(10)<>PL_FD_STATUS_OK Then
   'flash failure
End If
```

Upgrading the Firmware/Application

As was explained under Direct Sector Access 240, you can't use fd.setsector 290 to write to the flash area occupied by the firmware and Tibbo Basic application. This is done to prevent your application from inadvertently damaging its own code or the firmware.

A special fd.copyfirmware 269 method is provided for changing the data in the firmware/application area of the flash. The method copies the specified number of sectors (starting from the logical sector 0) from the data area 237 and into the firmware/application area of the flash memory (starting from the physical sector 0), then reboots your device.

Before invoking this method, store the new "binary image" of the firmware/application in the data area of the flash memory starting from logical sector #0. Logical numbers are assigned to the sectors in reverse, so using fd.setsector 290 to write to logical sector #0 actually means storing to the topmost physical sector of the flash memory. It goes without saying that your data area *has* to have enough capacity to store the new binary image.



The diagram above presents the case of the shared flash memory. With dedicated memory, the fd. object uses a separate flash IC to store its data. The size of this flash IC must be large enough to store the new firmware/application that you want to "activate" with the fd.copyfirmware method.

Exactly how you receive the new binary image is immaterial to this discussion. You can use any suitable transmission method, such as TCIP/IP, FTP, your own proprietary protocol, whatever! The important part is that before invoking fd.copyfirmware you must have the binary image stored in the data area of the flash, starting from the logical sector #0, and you must know how many sectors this occupies. After that, invoke fd.copyfirmware and hope that you prepared the right data!

Another method -- fd.copyfirmwarelzo -- does the same job but assumes that the data prepared in the flash memory is not the binary image itself, but an LZO-compressed version of the same. This method will unpack the binary and copy it into the firmware/application area of the flash memory. Because the compressed data is not sector-aligned, this method takes the compressed data size length in bytes, not sectors.

Use "compressed" firmware upgrades whenever you are dealing with a sizeable Tibbo BASIC project. On devices with shared flash memory, such projects will leave you with a small amount of free flash space, so the uncompressed "upgrade binary" will probably not fit there. In such cases LZO compression may be your way out, as it can achieve 30-50% reduction in file size.

The decompression algorithm accepts files compressed with **lzo1x-999.exe** utility.

- **BE VERY CAREFUL!** Using fd.copyfirmware or fd.copyfirmwarelzo on incorrect data will "incapacitate" your device and further remote upgrades will become impossible. You will need to physically go to your device and upload correct firmware and/or application, possibly through its serial port. Scary, huh?

File-based Access

The following topics will explain how to work with the flash disk:

Formatting the Flash Disk

Discussed in this section: fd.formatj 277, fd.format 276.

Before the flash disk can be used it must be formatted. Formatting allocates and initializes the "housekeeping" area of the disk. This consists of two boot sectors plus a certain number of sectors for the file record table (FRT), the file allocation table (FAT), and possibly the transaction journal 259 if you wanted one. For details see Disk Area Allocation Details 246.

Formatting is performed using the fd.format 276 or fd.formatj 277 method. These methods accept, as arguments, two parameters: total number of sectors to be occupied by the disk, and the maximum number of files that you wish to be able to store on the disk. Fd.formatj additionally has the third argument: the number of sectors that you want to allocate to the transaction journal. Fd.format does not give any sectors to the latter, so it is like setting the journal size to 0.

- Be generous when allocating the journal area. A size of 50-100 sectors would be recommended. Journal sectors get written to all the time, and having many of them prolongs 264 the life of your flash memory.

The total number of sectors cannot exceed the size of the data area 237. That is, the maximum "gross" disk size is fd.availableflashspace 267 sectors. The flash disk needs a number of sectors for its internal housekeeping, so actual useful capacity of the disk will be less. Checking Disk Vitals 249 topic explains how to find out current disk capacity, as well as get other useful info. There is also a minimum limit that will be accepted for the total disk size. The minimum exists because the number of sectors occupied by the disk must at least be enough for the "housekeeping" data.

- When debugging your application on devices with the shared 237 flash memory, use only a portion of the data area and leave some part of this area unoccupied. This will create a gap of unused sectors between the data and

the firmware/application areas of the flash. This way, your application will (mostly) be able to grow without corrupting the data area.

The fd. object stores up to 64 files on the disk, in a single "root" directory. You can choose to have less files if you don't need so many. This will reduce the number of "housekeeping" sectors required for the disk. The economy is not dramatic, but you can still save some space. Each file record occupies 64 bytes, so one sector of the FRT table can store 4 file records. For this reason, the number of files you specify will always be adjusted to be the multiple of 4. For example:

```
'format a disk to occupy 3/4th of available space and store at least 10
files
If fd.formatj((fd.availableflashspace/4)*3,10,100)<>PL_FD_STATUS_OK Then
   'some problem...
End If
'actual maximum number of files will be 12 (adjusted up from 10 to be a
multiple of 4)
```

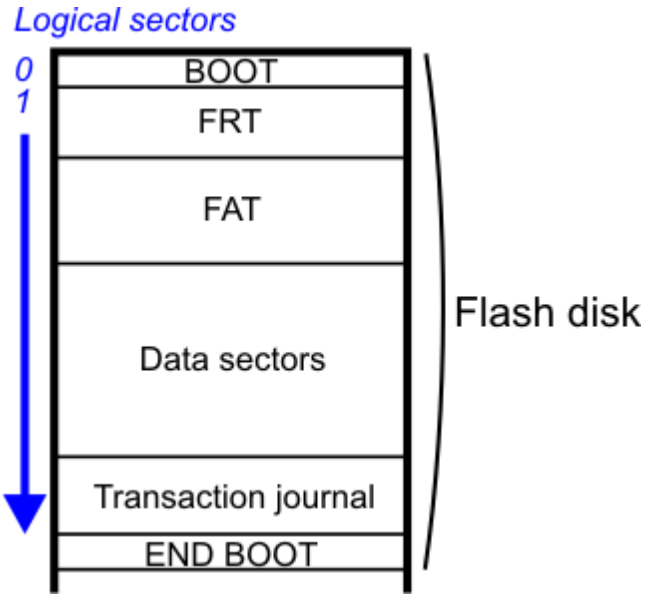
After the formatting the disk will be in the dismounted state and will need to be mounted 249 before any disk-related activity can be successfully performed.

This topic provides details on the internal structure of the flash disk.

The flash disk has a number of areas. Each area includes one or more flash sectors. Each sector has a physical size of 264 bytes. Of those, only 256 bytes are used to store the data. Two last bytes of each sector store the checksum 242. Six remaining bytes are mostly left unused, except in the FRT/FAT areas (see below), where they do serve a useful function.

The flash disk has the following areas:

- **BOOT** sector, always at logical sector #0 (so it is the topmost physical sector of the flash -- this was explained in Sharing Flash Between Your Application and Data 237). The boot sector contains the information about the sizes of other areas of the disk.
- File record table (**FRT**) area.
- File allocation table (**FAT**) area.
- **Data area** -- actual data sectors of files.
- **Transaction journal area** -- caches modified sectors during the disk transaction 259. This area only exists if you format the disk with the fd.formatj 277 method.
- **END BOOT** sector -- keeps the same data as the BOOT sector, but is located past the data area.



FRT area

Each file record occupies 64 bytes. Therefore, each sector of the FRT area can fit 4 file records. Therefore, if you specify (during formatting) that you would like to have 15 different files, the fd.format 276 (or fd.formatj 277) method will round this up to 16 files. This will require 4 sectors to fit. Actual amount of allocated sectors is always *double* that. This is done for sector leveling 264, which the fd. object takes care of internally. Also, the number of allocated sectors is never less than 8, again, for sector leveling reasons.

The maximum number of files stored by the .fd object is 64. Thefore, the maximum size of the FRT area is (64/4)*2= 32 sectors. Hence, the size of the FRT area is always between 8 and 32 sectors depending on the maximum number of files you need to store on the disk.

FAT area

Each FAT sector consists of 128 FAT entries, 2 bytes per entry. Therefore, each FAT sector can fit the allocation data for 128 sectors from the data area of the disk. To improve sector leveling, the number of sectors allocated for the FAT area is, again, double against what's necessary. So, for every 128 sectors in the data area of the disk there are 2 sectors in the FAT area. At least 16 sectors are always allocated to FAT.

An example; supposing the data area has 1100 sectors. Therefore, 1100/128= 9 FAT sectors are needed. The fd. object will allocate double this required minimum, so 18 FAT sectors will be prepared. Let's suppose now that we only have 500 data sectors. This requires 4 FAT sectors, 8 after we double this amount. This is less than the minimal 16 FAT sectors that are always provided, so the fd. object will still allocate 16 sectors.

Allocation and capacity calculation example

The fd.format method takes, as an input parameter, the total number of sectors that you wish the disk to occupy. From that, and the supplied desired max number of files, the fd.format will work out the size of each area of the disk. Let's see this on a real example:

Supposing, you call **fd.formatj(1300,41,100) --** you want the disk to occupy 1300 sectors and store 41 file. You want the journal area to have 100 free sectors. So, what will be the size of the FRT, FAT, and data areas?

First, the fd.format deducts 2 sectors to account for the BOOT and the END BOOT sectors that must always be present. This leaves us with 1298 sectors.

Next, decuct the journal area -- now only 1198 sectors are left.

Next, the fd.format determines the size of the FRT area: 41 is rounded to 44, this works out to hat (44/4)*2= 22 sectors are needed (this exceeds the minimum of 8, so we do not need to correct this number). We have 1198-22= 1176 sectors left.

Now, how many FAT sectors we need? 1176/128= 9.18, i.e. 10. Actual amount will be doubled, so we need 20 sectors (exceeds the minimum of 16, so we do not need to correct this number). Therefore, we are left with 1176-20= 1156 data sectors. This is what you will get from the fd.capacity 268 R/O property (see Checking Disk Vitals 249).

One small caveat...

Now let us show you one trickier example. Say, you do **fd.format(1301,4)**. You can quickly work out that you have 1301 sector for the FAT and data areas combined. Now, how should this be divided?

If you had 1300 sectors, then there would be no problem. Of them, 1280 would go to the data area, and 20 will belong to the FAT. 10 "necessary" FAT sectors will together hold 10*128= 1280 entries -- just right for the size of the data area. The number of FAT sectors is always doubled, so FAT will get 20 sectors. Everything fits perfectly!

Now, there is no good solution for 1301 sectors. If you give this extra one sector to the data area, then you will need two more sectors for the FAT area -- and you don't have those sectors. Allocating this extra sector to the FAT is useless, because you are not increasing the number of data sectors, so you don't really need an extra FAT sector. Hence, the value of 1301 is actually not suitable to us. Same goes for 1302. 1303 is good again, because three extra sectors allow us to give one sector to the data area, and add two sectors to the FAT area (FAT always increased in "doubles", remember?).

Fd.format avoid falling into the trap by correcting downwards the total disk size you request if it turns out to be one of the "problematic" values. So, if you do **fd.format(1302,4)**, then the result would be as if you did **fd.format(1300,4)**. The fd.totalsize 292 R/O property reflect this.

Why is there the END BOOT sector?

The END BOOT sector exists for one purpose -- to detect whether your application has encroached on the flash disk. As was already explained 237, on devices with the shared flash memory, the firmware/application and the data area reside in the same flash IC. Data area sectors are counted from the topmost physical sector of the flash memory. Therefore, the END BOOT sector is the one "closest" to your application. Should the application become larger and take some of the space that was previously occupied by the flash disk, the END BOOT sector will be overwritten. The fd. object will detect this during mounting 249 and return 3-PL_FD_STATUS_FORMAT_ERR status code 239.

Mounting the Flash Disk

Discussed in this topic: fd.mount 283, fd.ready 285.

The flash disk will not be accessible unless it is mounted using fd.mount 283. It can only be mounted after the flash memory has been successfully formatted using fd.formatj 277 (recommended) or fd.format 276. The disk has to be mounted after every reboot of your device (if you need to access it, of course). After the disk is mounted successfully, the fd.ready 285 R/O property will read 1- YES.

During mounting, the fd. object accesses the BOOT sectors 246 of the disk, reads the size of all disk areas 246, and determines if the flash disk is healthy. This does not include detailed checks of each file or data sector of the disk, but is enough to "catch" gross problems, such as disk corruption due to an increase in the application size.

Once the disk has been mounted, you can check its vitals 249, create and delete files 250, and perform other disk-related operations.

There is no way to explicitly dismount the disk, nor it is necessary. The disk will be dismounted (and fd.ready 285 become 0- NO) if:

- Any fatal condition is detected when working with it. This condition will be reflected by the fd.laststatus 282 R/O property. Not every error indicated by the fd.laststatus is fatal (see Fd. Object's Status Codes 239 topic).
- The disk is formatted using fd.formatj 277 (fd.format 276).
- Your application invokes fd.setbuffer 288/fd.setsector 290 inappropriately 263.

Checking Disk Vitals

Once the disk is mounted 249, you can check several important flash disk parameters:

- Fd.capacity 268 will tell you the number of usable data sectors on the disk (this excludes housekeeping sectors maintained by the disk).
- Fd.numservicesectors 284 will tell you how many sectors are used for internal "housekeeping" of the flash disk (see Disk Area Allocation Details 246 for... details).
- Fd.totalsize 292 will indicate how many sectors the disk occupies in the flash memory. Fd.totalsize=fd.capacity+fd.numservicesectors.
- Fd.getfreespace 280 method will return the number of free data sectors on the disk (those not yet occupied for file data storage).
- Fd.maxstoredfiles 283 will report the maximum number of files that can be stored on the disk (this is defined at the time of disk formatting 245).
- Fd.getnumfiles 281 fetches the number of files currently stored on the disk.

Keep in mind that currently existing disk might not be the largest one that could fit in the current data area (fd.availableflashspace 267).

File Names and Attributes

Every file you create on the flash disk has a file name and attributes. Both share a single text string that can be up to 56 characters long. The attributes (if any) are separated from the file name by a space. In other words, everything up to the first space is the file name, and everything else -- attributes.

The file names are case-sensitive and can include any characters, except, obviously, the space. This includes "/" and "\". The flash disk does not support subdirectories, but it is possible to emulate them by including "/" or "\" characters in the file name. The "." character can be used too, so you can have any extensions you like. The attributes portion of the string may contain any characters whatsoever, including spaces.

It is quite common for file systems to define attributes for the files. Typically, however, these attributes are preset. That is, you have a fixed list of things you can define about the file, such as the creating time, read-only flag, etc.

The fd. object uses a different approach. In a system that only runs a single application at any given time, it makes no sense to have, say, a fixed read-only flag. There is only one application running, after all. This application probably knows what files not to touch anyway! Or how about the date and time of the file creation? Does it make sense to keep this on a system without a real-time clock? Quite obviously, no.

For the above reasons, the fd. object allows you to store any attribute data and interpret it in any way you want. There is an attribute string and you can fill it with any data of your choosing and in accordance with the needs of your application. And yes, you can implement the read-only flag and record the creation date and time -- but only if you need to.

Here is an example where we create a file 250 and set its attributes too:

```
...
fd.create("File1.dat R 25-JUL-2008") ' <File1.dat> is the file name, <R 25-
JUL-2008> -- attributes, it is up to our program how to interpret this!
```

For more info on file attributes see Reading and Writing File Attributes 251.

Creating, Deleting, and Renaming Files

Discussed in this topic: fd.create 271, fd.rename 285, fd.delete 272, fd.getnumfiles 281, fd.maxstoredfiles 283.

You can't really do anything useful with the flash disk unless you create at least one file. The fd.create 271 method is used for this. The string you supply as an argument must include a file name and may also contain the attributes 249. Some examples:

```
If fd.create("File1.dat R 25-JUL-2008")<>PL_FD_STATUS_OK Then ' <File1.dat>
is the file name, <R 25-JUL-2008> -- attributes.
   'some problem
End If
If fd.create("    File2")<>PL_FD_STATUS_OK Then 'no attributes defined for
this file. Notice leading spaces -- they will be removed.
   'some problem
End If
If fd.create("Database/users.dat")<>PL_FD_STATUS_OK Then 'and here we
emulate a directory
   'some problem
End If
If fd.create("file 3")<>PL_FD_STATUS_OK Then 'if the idea was to create a
<file 3> file, then this won't work! "3" will be interpreted as attributes!
   'some problem
```

```
End If
```

Naturally, each file on the flash disk must have a unique name, or the 5- PL_FD_STATUS_DUPLICATE_NAME error[239] will be generated. Every existing file always has at least one data sector allocated to it. This is how the 7- PL_FD_STATUS_DATA_FULL error may be generated when you are creating a new file. Finally, the total number of files stored on the flash disk is limited to what you defined when formatting[245] your disk. This maximum can be checked through the fd.maxstoredfiles[283] R/O property. Try to exceed this number and you will get the 6- PL_FD_STATUS_FILE_TABLE_FULL error code. Current file count can be obtained through fd.getnumfiles[281].

To delete a file, use the fd.delete[272] method:

```
If fd.delete("File1.dat")<>PL_FD_STATUS_OK Then
   'some problem
End If
If fd.delete("File2 abc")<>PL_FD_STATUS_OK Then 'the "abc" part will be
ignored -- everything after the space is NOT a part of the file name
   'some problem
End If
If fd.delete("   Database/users.dat")<>PL_FD_STATUS_OK Then 'leading spaces
will be ignored
   'some problem
End If
```

The file you are deleting must, of course, still exist, or you will get the 9- PL_FD_STATUS_NOT_FOUND error[239]. You can also rename a file using the fd.rename[285] method (this will preserve file attributes[251]). It is OK to delete or rename a file which is currently opened[253].

Fd.create[271], fd.rename[285], and fd.delete[272] make changes to the sectors of the flash disk. For the highest possible reliability, use disk transactions[259] when invoking these methods.

Reading and Writing File Attributes

Discussed in this topic: fd.getattributes[278], fd.setattributes[287].

You can set initial file attributes[249] right when creating[250] a new file. To read existing file attributes, use the fd.getattributes[278] method:

```
'this example ignores potential error conditions
Dim s As String
...
s=fd.getattributes("File1.dat") 'get the attributes for this file
If instr(1,s,"R",1)=0 Then 'delete the file only if there is no 'R' in the
attributes
fd.delete("File1.dat")
End If
```

You can set new (change) attributes with the fd.setattributes[287] method:

```
If fd.setattributes("File1.dat", "D")<>PL_FD_STATUS_OK Then 'the attribute
string will now consist of a single 'D'
   'some problem
End If
```

Remember that the file name and the attributes share the same 56-byte string in the file record table of the flash disk. Therefore, the maximum length of the attributes is limited to 56-length_of_the_file_name-1. This "-1" accounts for the space character that separates the file name from the attributes.

Attempting to perform the fd.getattributes or fd.setattributes on a file that does not exist will generate the 9- PL_FD_STATUS_NOT_FOUND error[239].

Fd.setattributes[287] makes changes to the sectors of the flash disk. For the highest possible reliability, use disk transactions[259] when invoking it.

Walking Through File Directory

Discussed in this topic: fd.resetdirpointer[286], fd.getnextdirmember[280], fd.getnumfiles[281].

At times it is necessary to get the list of all files stored on the flash disk. Most operating systems provide this feature in the form of a DIR command. The fd. object offers two methods -- fd.resetdirpointer[286] and fd.getnextdirmember[280] -- that allow you to get the names of all stored files one by one.

An imaginary "directory pointer" walks through the file record table[246] (FRT) of the flash disk. The fd.resetdirpointer method resets the directory pointer to 0, i.e. to the very first directory record. The fd.getnextdirmember returns the next file name and advances the pointer. Only file names are returned, attributes[251] are excluded.

Calling the fd.getnextdirmember repeatedly will get you all the file names. When there are no more names left to go through, the method will return an empty string. You can use this to end your directory walk:

```
'walk through the file directory -- method #1
Dim s As String

fd.resetdirpointer
Do
   s=fd.getnextdirmember
   If fd.laststatus<>PL_FD_STATUS_OK Then
         'some problem
   End If
   ... 'process the file name in s
Loop While s<>"" 'we exit on empty string
```

Alternatively, you can use a for-next cycle, since the number of currently stored files can be checked through the fd.getnumfiles[281] method:

```
'walk through the file directory -- method #2
Dim s As String
Dim f As Byte

fd.resetdirpointer
For f=1 To fd.getnumfiles
   s=fd.getnextdirmember
   If fd.laststatus<>PL_FD_STATUS_OK Then
         'some problem
   End If
   ... 'process the file name in s
Next f
```

Opening Files

Discussed in this topic: fd.filenum 273, fd.open 284, fd.fileopened 273, fd.maxopenedfiles 282.

You must first open a file in order to work with its data. The fd.open 284 method opens a file with a specified name and "on" a file number currently selected by the fd.filenum 273 property. All operations related to the file data are then performed by referring to the file number, not the file name. These operations include writing to and reading from files 254, removing data from files 255, searching within files 256, and closing files 258.

The concept of file numbers is not new -- other operating systems, too, assign a number to the file when the file is opened. In Tibbo Basic, you select the number you want to open the file on yourself. You do this by selecting a desired value for the fd.filenum property:

```
'will open two files 'on' numbers 3 and 5

fd.filenum=3
If fd.open("File1")<>PL_FD_STATUS_OK Then
   'some problem
End If
fd.filenum=5
If fd.open("TrestFile")<>PL_FD_STATUS_OK Then
   'some problem
End If
```

Whenever you want to work with one of the currently opened files, just set the fd.filenum to the number on which this file was opened (naturally, you need to somehow remember this number). And how many files can be opened concurrently? The fd.maxopenedfiles 282 R/O property will tell you that. This value is platform-dependent. Your fd.filenum value can move between 0 and fd.maxopenedfiles-1.

When the file is opened "on" a certain file number, the fd.fileopened 273 R/O property returns 1- YES when this file number is selected in the fd.filenum.

Any leading spaces in the file name you supply for fd.open are removed. After that, only the part up to the first space is processed -- the rest of the string is ignored. The following three code lines all open the same file:

```
fd.open("File1")
fd.open("        File1") 'leading spaces will be removed
fd.open("File1 some more stuff") 'everything after the first space will be
ignored too
```

Naturally, the file with the specified name must exist, or you get the 9-PL_FD_STATUS_NOT_FOUND error. You may not open the same file "on" two different file numbers -- this will generate the 11-PL_FD_STATUS_ALREADY_OPENED error. You may reopen the same or another file "on" the same file number, but this can lead to the loss of (some) changes made to the previously opened file. To avoid this, close the file or use the fd.flush method before opening it again. Note that the fd.flush method does not depend on the current fd.filenum value and works globally on any most recently changed file.

Writing To and Reading From Files

Discussed in this topic: fd.getdata, fd.setdata, fd.pointer, fd.setpointer, fd.filesize.

Writing to and reading from files are two most important file operations. Having a flash disk would be pointless without them. Writing and reading is done using two methods -- fd.setdata and fd.getdata respectively. For these to work, the file must first be opened. Fd.setdata and fd.getdata work on the currently selected file, and the selection is made through the fd.filenum property.

The pointer

Both reading and writing operations are always performed from the *current pointer position*. This position can be checked through the fd.pointer R/O property and changed through the fd.setpointer method. Executing the fd.setdata or fd.getdata moves the pointer forward by the number of file positions written or read.

The pointer always points at the position (offset) in the file from which the next reading or writing will be done. File offsets are counted from one, not zero. The very first byte in the file is at offset one, the next byte -- at offset two, and so on. The very last byte in the file is at offset equal to the size of the file, which is indicated by the fd.filesize R/O property. The maximum pointer value, however, is fd.filesize+1! When the pointer is at maximum, it effectively points at the file position that doesn't yet exist. So, writing to the file at this position will append new data to the end of the file. Writing to the file when the pointer is not at the maximum will overwrite existing data.

When the file is opened, the pointer is set to 1 if the file has any data in it (fd.filesize<>0), or 0 if the file is empty (fd.filesize=0). It is not possible to set the pointer to zero if the file is not empty. The pointer will be moved automatically if the file becomes smaller and will be set to zero if the file becomes empty.

Writing and reading

In the following example, we append the data to the end of the file, then read the data from the beginning of the file:

```
Dim s As String

'open a file 'on' file number 3
fd.filenum=3
fd.open("SomeFile")
fd.setpointer(fd.filesize+1) 'works always -- no matter whether the file is
empty or not
fd.setdata("Append this to the file")
fd.setpointer(1) 'go back to the beginning of the file
s=fd.getdata(10) 'read 10 bytes from the beginning of the file
```

If fd.setdata is executed when the pointer isn't at the end of the file (at fd.filesize 274+1 position) then some of the existing file data will be partially overwritten. If the pointer moves past the current file size (see.filesize 274), the size will be increased automatically. As you append new data to the file, the file will grow larger. New data sectors will be allocated and added to the file automatically as needed. You will get the 7- PL_FD_STATUS_DATA_FULL error 239 if the disk runs out of free sectors. When the fd.setdata is executed and the disk full condition is detected, the entire data string supplied with fd.setdata is not saved into the file (and not just the part that couldn't fit).

Fd.setdata 289 makes changes to the sectors of the flash disk. For the highest possible reliability, use disk transactions 259 when invoking this method.

File data caching

The fd. object uses the RAM buffer #1 240 as an intermediary storage for the sector's data. When you access a certain data sector, the contents of this sector are loaded into the RAM buffer #1. When you change the data in the sector, it is the data in the RAM buffer that gets changed. The changed contents of the RAM buffer will not be saved back to the flash memory until the contents of *another* sector must be loaded into the buffer. So, for example, if you do this...

```
fd.setdata("Write this to a file")
```

... and then leave the disk alone, then the new data may stay in the RAM buffer indefinitely. It may get lost -- for example, if you reboot the device. To prevent this, close 258 the file or use the fd.flush 275 method -- either one will cause the changed data in the RAM buffer to be saved back to the flash memory. Note that the fd.flush method does not depend on the current fd.filenum value and works globally on any most recently changed file.

It is not necessary to use fd.flush if your disk operations are performed within a disk transaction 259.

Removing Data From Files

Discussed in this topic: fd.setfilesize 289, fd.cutfromtop 271.

Using the fd.setdata 289 method can only increase the size of your file. To reduce the file size, i.e. remove some data from it, use one of the two following methods. Both methods work on a currently selected file, and the selection is made through

the fd.filenum 273 property.

The fd.setfilesize 289 method cuts the end portion of your file and preserves a specified amount of bytes in the beginning of the file:

```
'open a file 'on' file number 4
fd.filenum=4
fd.open("SomeFile")
fd.setfilesize(fd.filesize/2) 'cut the file size in half
```

The fd.setfilesize can't be used to enlarge your file, only to make it smaller. Data sectors previously allocated to the file will be "released" (marked unused) if they become unnecessary due to the reduction in the file size. The first data sector of the file, however, will always remain allocated, even when the file size is set to 0.

The size of the file, indicated by the fd.filesize 274 R/O property, will be corrected downwards to reflect the amount of data left in the file.

The pointer position 254 will be affected by this method. If the file becomes empty, the pointer will be set to 0. If the new file size is not zero, but the new size makes current pointer position invalid (that is, fd.pointer > fd.filesize+1) then the pointer will be set to fd.filesize+1.

The second method -- fd.cutfromtop 271 -- removes a specified number of *sectors* (not bytes) from the beginning of the file and leaves the end portion of the file intact:

```
'open a file 'on' file number 2
fd.filenum=2
fd.open("SomeFile")
fd.cutfromtop(3) 'remove three front sectors from the file (that is, remove
up to 3*256= 768 bytes of data)
```

The size of the file will be corrected downwards in accordance with the amount of removed data. For example, performing fd.cutfromtop(2) on a file occupying 3 data sectors will reduce its size by 512 bytes (amount of data in 2 sectors removed). Performing fd.cutfromtop(3) will set the file size to 0.

The pointer position is always reset as a result of this method execution. If the new file size is 0, then the pointer will be set to 0 as well. If the file size is not 0, then the pointer will be set to 1.

Fd.setfilesize 289 and fd.cutfromtop 271 make changes to the sectors of the flash disk. For the highest possible reliability, use disk transactions 259 when invoking these methods.

Searching Within Files

Discussed in this topic: fd.find 274.

The fd.find 274 method allows you to quickly search through the file from a specified starting position, either in forward or back direction, and locate the Nth instance of a search substring. The method also allows you to specify the search "increment" (step). The search is performed on a currently selected file, and the

selection is made through the fd.filenum 273 property.

The search returns the position within the file, counting from 1, where the target occurrence of the substring has been encountered, or 0 if the target occurrence of the substring was not found.

The increment parameter is very important as it allows you to perform two fundamentally different classes of search.

Full text search

Set the increment to 1 and you will search through the entire contents of the file:

```
Dim dw As dword

'try to find the 2nd occurrence of 'ABC', search forward starting at the
beginning of the file
dw=fd.find(1,"ABC",2,FORWARD,1)
If fd.laststatus<>PL_FD_STATUS_OK Then
   'some disk-related error
Else

If dw<>0 Then
   'found! process this...
Else
   'not found...
End If
```

Since the search substring in the above example -- "ABC" -- does not have repeating fragments, you can actually set the search increment to 3 (the length of the substring). This will improve the search speed:

```
fd.find(1,"ABC",2,FORWARD,3) 'no repeating fragments in the substring, use
its length as increment
```

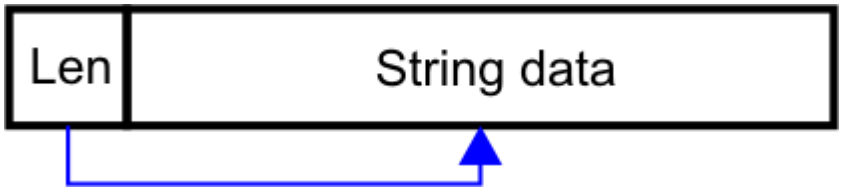
Record-style search

If the file in question is a data table consisting of individual records, then you can arrange for a very efficient search for the record with desired field value. All you need to achieve this is to have record fields occupy fixed offsets within records. For example, supposing you have the data table consisting of records with the following structure:

Field name	Offset in bytes*	Length in bytes
Category	+0	1
ID-code	+1	11
Last name	+12	21
First name	+43	21

** with respect to the beginning of the record*

Each record of this data table occupies 54 bytes, so this will be our step. Three fields -- "ID-code", "last name", and first name" are strings which, of course, can have a variable length. To facilitate the use of the fd.find method, each field must reside at a fixed offset relative to the beginning of the record. That is, even if the "ID-code" for a particular record is shorter than the maximum possible field length, the "Last name" field will still be at offset +12. To reflect actual length of the field data, each field of string type starts with a byte that denotes the length of the string, followed by the string data itself:



Now let's suppose you want to find a record whose "last name" is "Smith". For this we search starting from file offset 21, which is the offset of the "Last name" field within the record (this assumes that the first record starts right from the beginning of the file, which is usually the case). The search step will be 54 -- the size of the record:

```
Dim dw As dword

'try to find the record with the "Last name" set to "Smith"
dw=fd.find(21,chr(5)+"Smith",1,FORWARD,54) 'notice how we supply the string
length
If fd.laststatus<>PL_FD_STATUS_OK Then
   'some disk-related error
Else

If dw<>0 Then
   'found- convert into the record number and process...
   dw=dw/54
Else
   'not found...
End If
```

You can also search back, but remember that this is less efficient (takes ~ 50% longer) compared to forward searches.

Closing Files

Discussed in this topic: fd.close 269, fd.flush 275.

Files are closed using the fd.close 269 method. The method is performed on a currently selected file (selection is made through the fd.filenum 273 property). Attempting to invoke this method "on" a file number that did not have any opened file associated with it generates no error.

Proper file closing is only required if you made changes to the file. If you only read

data from the file you don't have to bother closing it -- you can open another file right "on" the same file number. Doing so on a file that was changed may cause some recent changes to be lost. To prevent this, use the fd.close method or fd.flush method. Note that fd.flush does not depend on the current fd.filenum value and works globally on any most recently changed file.

It is not necessary to use fd.flush if your disk operations are performed within a disk transaction.

Using Disk Transactions

Discussed in this topic: fd.transactionstart, fd.transactioncommit, fd.transactionstarted, fd.transactioncapacityremaining.

Flash disk is a sensitive media. Many things can go wrong -- especially if a disk operation is abruptly interrupted due to the power failure or for some other reason. Try appending data to a file in a loop while power-cycling the device at random. It won't be long before the file, and even the disk itself become corrupted.

Many systems require reliable disk operation, even during random power cuts. For such systems, disk transactions are the answer!

Transactions help strengthen your application on two levels:

- **Disk level:** seemingly simple operations of the disk often involve complex internal changes to the disk data. Losing power while executing such operations may corrupt your files and the disk itself. Transactions make sure it won't happen.
- **Logical level:** there are often group operations that must be performed together (in synchronicity), or not at all. For example, you make have the data table file and its index file. Change the first one, and the other one must be changed too. Transactions allow you to effect such changes in unison and make sure that both files are changed or no files are changed at all.

Transactions only make sense for disk data changing operations. These are fd.create, fd.rename, fd.delete, fd.setattributes, fd.setdata, fd.setfilesize, and fd.cutfromtop. Using transaction with fd. object's methods that only read the data is pointless.

Performing disk operations within transactions

Transactions are only possible when the flash disk is formatted with the fd.formatj method, and the maxjournalsectors argument was >1. For good disk leveling, choose this parameter to be in the 50-100 range, and definitely not below 17 (see the explanation here).

To start a disk transaction, use the fd.transactionstart method. To finish with the transaction and commit (save) all changes to the disk, invoke fd.transactioncommit. Between these two statements, add any operations that lead to the changing of the disk data.

With transaction in progress, no changes are made to the disk data. Turn the power off before executing fd.transactioncommit, turn the power back on, mount the disk, and it will look as if the disk operations within the unfinished transaction have never happened. This is because all changed sectors are temporarily cached in the journal area of the disk. Every sector that gets changed in the cause of a transaction is saved to the journal, and not back to its original place. Invoking fd.transactioncommit starts the process of copying changed sectors from the

journal memory and into their original locations. This work can be safely interrupted. If the device loses power while committing changes, next fd.mount 283 will finish this job!

Here is a simple example:

```
fd.transactionstart
fd.setdata("write some data to a file")
fd.transactioncommit
```

It may seem that fd.setdata is a trivial operation, so why put it inside the transaction? In reality, performing this simple task may lead to changing of as many as eight 261 (!) sectors on the disk. If your device experiences a power failure while in the middle of writing to a file, then this file, and even the disk itself can become corrupted!

Here is another example, where we change two files. Encasing this within the transaction makes sure that both files are changed, or no files are changed at all:

```
'we presume that there are two files opened 'on' file numbers 0 and 1
fd.transactionstart
fd.filenum=0
fd.setdata("write some data to the first file")
fd.filenum=1
fd.setdata("write some more to the second file")
fd.transactioncommit
```

Fd.transactionstarted 294 exists so you can check if any disk transaction is already in progress. Executing fd.transactioncommit while fd.transactionstarted= 0- NO generals a non-fatal 13- PL_FD_STATUS_TRANSACTION_NOT_YET_STARTED error 239. Executing fd.transactionstart when fd.transactionstarted= 1- YES generates 12- PL_FD_STATUS_TRANSACTION_ALREADY_STARTED non-fatal error code. 15- PL_FD_STATUS_TRANSACTIONS_NOT_SUPPORTED non-fatal error is generated if the disk was formatted with fd.format 276, or maxjournalsectors argument of fd.formatj 277 was set to 0 or 1.

You cannot abort the transaction that has already been started. If you started it, you must follow through with it. There is no need to execute fd.flush 275 when using disk transactions.

Transaction journal has limited capacity

You can't just pile up a large number of disk write operations within a single transaction. This is because transactions have a limit: the number of sectors that can be changed within a single transaction. Understanding Transaction Capacity 261 provides further details.

Direct sector access is not affected by transactions

Fd.setsector 290 and fd.getsector 281 always access the specified target sector and

not its cached copy even if the disk transaction is in progress the target sector has been cached already.

Discussed in this topic: fd.transactioncapacityremaining 292.

Fd.transactioncapacityremaining 292 can tell you how many changed sectors the transaction journal can still accommodate. The initial value of the fd.transactioncapacityremaining (just after fd.transactionstart 293) depends on two factors, whichever is smaller:

- The value of 16 (this is the absolute maximum).
- Maxjournalsectors-1, where maxjournalsectors is the argument of the fd.formatj 277 method.

The maxjournalsectors allocates the journal area of the flash disk. In this journal area, one sector is always needed for internal housekeeping, while other sectors cache the data sectors changed during the transaction. Therefore, if you want to achieve the maximum journal capacity (16), the journal must have at least 17 sectors in it. It is very important to use the maximum transaction memory capacity. The value of 16 is not random -- it guarantees that you will always be able, within a single transaction, to change the data in two different files.

In practice, do set a mach higher journal size. This won't increase transaction memory, but it will prolong 264 the life of the flash IC. Values in the 50-100 range are recommended.

14- PL_FD_STATUS_TRANSACTION_CAPACITY_EXCEEDED error 239 will be generated if transaction capacity is exceeded by performing too many disk write operations within a single transaction. The disk will be dismounted (but not damaged) and all changes to the disk made within this failed transaction will be forgotten.

Disk operations and the number of sectors they may affect

So how many sectors may be changed during various disk operations? The table below provides the worst-case numbers.

Fd. object's method that changes disk data	Maximum number of journal entries used
fd.create 271	FRT*2 + FAT*2 + DATA = **5**
fd.rename 285, fd.setattributes 287	FRT*2 = **2**
fd.delete 272, fd.setfilesize 289, fd.cutfromtop 271*	FRT*2+ FAT*X*2 = **?**
fd.setdata 289	FRT*2 + FAT*2*2 + DATA*2 = **8**

** Executing these methods can potentially overflow the transaction journal.*

And now, the explanation for the table data. It is based on the material from the Disk Area Allocation Details 246 topic.

- "FRT" means "one sector from the file records table". File records are changed when files are created, renamed, or deleted. They are also changed when the file

size changes, or when your application sets new file attributes.

- "FAT" -- "one sector from the file allocation table". File allocation sectors change when files are created and when file sectors are allocated or released in accordance with changing file sizes.
- "DATA" -- "one sector from the data sectors area of the disk". Data sectors hold actual file data. They change when you write new data to files.
- "FRT*2" and "FAT*2" are caused by sector leveling 264: changing one FAT or FRT sector actually means changing two sectors.
- "DATA*2" means that two data sectors can change when you write some data to the disk. This is because a portion of this data may reside in one data sectors, and the rest -- in another data sector.
- "X" parameter represents the number of *active* FAT sectors that can potentially be affected by invoking the corresponding method.

Here is how to calculate the X parameter:

Supposing, the flash disk capacity (fd.capacity 268) is 1000 sectors. The FAT, then, must hold 1000 entries. Each FAT sector can hold 128 entries, so there will be 8 *active* FAT sectors (X = 8). Do not confuse this with the *total* number of sectors in the FAT area -- it is at least twice this amount (because of sector leveling 264 measures). "8" here represents the number of FAT sectors that are needed at *any given time*.

Fd.delete 272, fd.setfilesize 289, and fd.cutfromtop 271 top have one thing in common: they can potentially cut a huge file (occupying the entire disk) to zero. In the process, X FAT sectors will be altered!

Therefore, for the disk with fd.capacity= 1000, executing these methods can potentially change 2+8*2=18 sectors. This is above the maximum journal capacity, which is 16! So, how to avoid 14-PL_FD_STATUS_TRANSACTION_CAPACITY_EXCEEDED error 239 and still use transactions?

And the answer is: do it in stages. The following example deletes a very large file in a "safe" manner:

```
dim i as byte

i=(fd.transactioncapacityremaining-1)/2 'this is how many FAT sectors we can
change at once
i=i*128 'and this is how many data sectors we can cut from the file at once
(one FAT sector holds 128 entries)

fd.open("FILE1")
while fd.filesize>0
      fd.transactionstart
      fd.cutfromtop(i)
      fd.transactioncommit
wend
fd.transactionstart
fd.delete("FILE1")
fd.transactioncommit
```

Not a direct sum total

The worst-case number of sectors that may be changed during a transaction is *not* the sum total of maximum numbers for each fd. object's method. In the example below, the maximum number of changed sectors is 9, not 10.

```
fd.transactionstart
fd.setdata("write some data to a file") 'up to 8 sectors changed, and this
includes 2 change FRT sectors
fd.setattributes("ABC") 'FRT changes again, but for the same FRT record
(this will use only 1 entry in the journal)
fd.transactioncommit 'therefore, the total is up to 9
```

In this example, the worst-case number is 10, because we are working with 2 different files and their FRT records could be in different FRT sectors:

```
fd.transactionstart
fd.finenum=0
fd.setdata("write some data to a file") 'up to 8 sectors changed for file #0
fd.filenum=1
fd.setattributes("ABC") '2 FRT sectors changed for file #1
fd.transactioncommit 'therefore, the total is up to 10
```

File-based and Direct Sector Access Coexistence

File-based 245 and direct sector 240 access methods can be used at the same time, as long as you understand how direct access may affect (and possibly *screw up*) the flash disk.

Direct Sector Access 240 explains that working with flash sectors is done through two RAM buffers numbered #0 and #1. Flash disk operation depends on these buffers as well.

Buffer #0 is used for processing housekeeping data and stores no valuable content that must be preserved past the end of a single disk-related method execution. That is, once the method such as the fd.setdata 289 has executed, buffer #0 has no valuable data in it.

Buffer #1 is used to load and store the contents of the most recently loaded (and possibly changed) data sector. So, if fd.setdata was recently called, this buffer may still hold new data, and this data may not be saved to the flash memory yet. Corrupting this unsaved data would have unpleasant consequences for the file. To prevent this, the fd. object automatically dismounts the disk (sets fd.ready 285 = 0-NO) if your application does fd.setbuffer 288 or fd.getsector 281 while the fd.buffernum 267 = 1 and while this buffer was loaded with *new and as yet unsaved* sector data.

Preventing disk dismounting is easy. You can opt to work on the RAM buffer #0 (set fd.buffernum= 0), or flush the unsaved data by invoking fd.flush 275 or fd.close 269.

Fd.setsector 290 method will cause disk dismounting if the destination was within the disk area (sectors 0 through fd.totalsize 292 -1). Writing outside the disk area will not interfere with the flash disk operation.

The fd.getbuffer 278 method does not cause any interference with the flash disk, so

it can be used freely.

The fd.checksum[268] will write to the RAM buffer (when in the "PL_FD_CSUM_MODE_CALCULATE" mode), but the checksum calculation method is the same as the one used by the disk itself, so setting this checksum will never be wrong.

Note, finally, that any file-related method will affect the value of the fd.buffernum. Never assume that you know what this property is set to. Always set it explicitly and each time before performing direct sector access.

Prolonging Flash Memory Life

Flash memory has a huge limitation -- the number of times you can rewrite each of its sectors is limited to around 100'000 times. At first this may seem like a virtually unreachable limit, but in reality you can "get there" quite fast, which is not a good thing. Once the flash wears down, you will start having data errors, data corruption, failed sectors, etc. This topic explains what you can do to prolong the life of the flash memory used in your device. This is generally achieved by (1) reducing the number of writes to the flash memory, and (2) using sector leveling, i.e. spreading sector writes, as evenly as possible, between the sectors.

File-based access

For file-based access[245], the fd. object does the bulk of work for you. We have already explained[246] that the file record table (FRT) and the file allocation table (FAT) -- the most heavily written to areas of the disk -- occupy at least double the space needed to store their data. They also have the minimum number of sectors they will take, regardless of what is necessary. This ensures that at any given time, the FRT and the FAT have spare sectors. These spare sectors, as well as occupied sectors, are in constant rotation. For example, every time you create a file, a change is made to one of the sectors of the FRT. This change requires writing data to one of the physical sectors. In the process, the fd. object will take the previously used FRT sector and "release it" into the pool of spare FRT sectors. At the same time, one of the spare FRT sectors will become active and store changed data. The FAT operates in the same manner. While being fully transparent to your application, the process greatly prolongs useful flash memory life.

You can, and are advised to, further reduce the wear of the FAT and FRT by decreasing the number of writes that will be required. One way to do so is to create all necessary files and allocate space for them once -- typically when your application "initializes" your device.

Say, you have a log file, which stores events collected by your application. An obvious approach would be to simply append each new event's data to the file. This way, the file will grow with each event added. But wait a second, this means that the FRT area, which keeps current file size, will be changed each time you add to the file! The FAT area will be stressed too!

An alternative approach would have us create a file of desired maximum size once and fill it up with "blank" data (such as &hFF codes). We will then overwrite this blank data with actual event data is events are generated. This time around, our actions will be causing no changes in the FRT and FAT areas, thus prolonging the life of the flash IC.

Not only this approach prolongs the life of the flash IC, but disk writing to the log will be faster (less sectors to change), and, in case you use transactions[259], less journal entries[261] will be used (a single transaction can fit more operations).

When using transactions, do allow a generous space for the journal memory[261]. Recommended maxjournalsectors value for fd.formatj[277] is 50-100. Journal memory

is a high-traffic disk area, so having more sectors there will improve flash memory life.

The data area of the disk has limited leveling that results in spreading unused sector utilization. The fd. object makes sure that when your file needs a new data sector, this data sector will be selected from a pool of available data sectors in a random fashion. Once the data sector has been allocated to a file, however, it stays with that file for as long as necessary. So, if you are writing at a certain file offset over and over again, you are stressing the same physical sector of the flash IC.

On large files, you rarely write at the same offset all the time. For example, if you have a log file that has 1000 data sectors, then it is unlikely you will be writing to the same sector over and over again. For smaller files the probability is higher. Your solution is to erase the file and recreate it again from time to time. This will randomly allocate new sectors for the file.

Direct sector access

Direct sector access 240 is a low-level form of working with the flash. You are your own master, the fd. object does not help you with anything, and it is up to you to make sure that the flash IC is not being worn out unevenly. Generally speaking, limit the number of times you are writing to the flash and/or implement some form of leveling where a large number of sectors are used to share the same task and each sector gets its fair share of work.

Properties and Methods

The following classification groups properties and methods of the fd. object by their logical function.

Properties and methods under fd.filenum 273 are indented to reflect the fact that they are performed on the file currently selected by the fd.filenum.

Member	Type	Where to read about it
Direct sector access		
fd.buffernum 267	Property	Direct Sector Access 240
fd.getsector 281	Method	Direct Sector Access 240
fd.setsector 290	Method	Direct Sector Access 240
fd.getbuffer 278	Method	Direct Sector Access 240
fd.setbuffer 288	Method	Direct Sector Access 240
fd.checksum 268	Method	Using Checksums 242
fd.copyfirmware 269	Method	Upgrading the Firmware/Application 243
fd.copyfirmwarelzo 270	Method	Upgrading the Firmware/Application 243
General disk info and operations		
fd.availableflashspace 267	R/O property	
fd.maxopenedfiles 282	R/O property	Opening Files 253

fd.formatj 277	Method	Formatting the Flash Disk 245
fd.format 276	Method	Formatting the Flash Disk 245
fd.mount 283	Method	Mounting the Flash Disk 249
fd.ready 285	R/O property	Mounting the Flash Disk 249
fd.capacity 268	R/O property	Checking Disk Vitals 249
fd.numservicesectors 284	R/O property	Checking Disk Vitals 249
fd.totalsize 292	R/O property	Checking Disk Vitals 249
fd.getfreespace 280	Method	Checking Disk Vitals 249
fd.laststatus 282	R/O property	Fd. Object's Status Codes 239
File directory		
fd.maxstoredfiles 283	R/O property	Checking Disk Vitals 249, Creating, Deleting, and Renaming Files 250
fd.getnumfiles 281	Method	Checking Disk Vitals 249, Creating, Deleting, and Renaming Files 250, Walking Through File Directory 252
fd.resetdirpointer 286	Method	Walking Through File Directory 252
fd.getnextfirmember 280	Method	Walking Through File Directory 252
fd.getattributes 278	Method	Reading and Writing File Attributes 251
fd.setattributes 287	Method	Reading and Writing File Attributes 251
fd.create 271	Method	Creating, Deleting, and Renaming Files 250
fd.rename 285	Method	Creating, Deleting, and Renaming Files 250
fd.delete 272	Method	Creating, Deleting, and Renaming Files 250
File access		
fd.flush 275	Method	Closing Files 258
fd.filenum 273	Property	Opening Files 253
fd.open 284	Method	Opening Files 253
fd.close 269	Method	Closing Files 258
fd.fileopened 273	R/O property	Opening Files 253
fd.filesize 274	R/O property	Writing To and Reading From Files 254
fd.pointer 285	R/O	Writing To and Reading From Files 254

	property	
fd.setpointer 291	Method	Writing To and Reading From Files 254
fd.getdata 279	Method	Writing To and Reading From Files 254
fd.setdata 289	Method	Writing To and Reading From Files 254
fd.cutfromtop 271	Method	Removing Data From Files 255
fd.setfilesize 289	Method	Removing Data From Files 255
fd.find 274	Method	Searching Within Files 256
fd.sector 287	R/O property	Writing To and Reading From Files 254
Transactions		
fd.transactionstart 293	Method	Using Disk Transactions 259
fd.transactioncommit 293	Method	Using Disk Transactions 259
fd.transactionstarted 294	R/O property	Using Disk Transactions 259
fd.transactioncapacityremaining 292	R/O property	Using Disk Transactions 259

.Availableflashspace R/O Property

Function: Returns the total number of sectors available to store application's data.

Type: Word

Value Range: The value depends on the flash capacity and the flash memory arrangement of your device.

See Also: Sharing Flash Between Your Application and Data 237

Details

.Buffernum Property

Function: Sets/returns the number of the RAM buffer that will be used for direct sector access.

Type: Byte

Value Range: 0 or 1. **Default**= 0 (RAM buffer #0 selected).

See Also: Direct Sector Access 240,

fd.getbuffer 278, fd.setbuffer 288, fd.getsector 281, fd.setsector 290, fd.checksum 268, fd.copyfirmware 269, fd.copyfirmwarelzo 270

Details

All file-based 245 operations of the flash disk also load data into the RAM buffers. Switch to the RAM buffer #0 *each time* before performing direct sector access 240 with this or other related methods -- this will guarantee that you won't corrupt the files and/or the file system and cause disk dismounting (fd.ready 285 becoming 0-NO).

.Capacity R/O Property

Function: Returns the capacity, in sectors, of the currently existing flash disk.

Type: Word

Value Range: 0-65535

See Also: Checking Disk Vitals 249,

fd.numservicesectors 284, fd.totalsize 292, fd.getfreespace 280, fd.maxstoredfiles 283, fd.getnumfiles 281

Details

The disk must be mounted (see fd.mount 283) for this property to return a meaningful value.

.Checksum Method

Function: Calculates and writes into the RAM buffer, or verifies the checksum for the data in the currently selected RAM buffer of the flash memory (selection is made through the fd.buffernum 267 property).

Syntax: **fd.checksum(**mode **as pl_fd_csum_mode, byref** csum **as word) as ok_ng**

Returns: 0- OK: Completed successfully (always the case when the mode= 1- PL_FD_CSUM_MODE_CALCULATE).

1- NG : The checksum was found to be invalid (can only be generated when the mode= 0-PL_FD_CSUM_MODE_VERIFY).

Also returns the calculation result indirectly, through the csum argument.

See Also: Using Checksums 242, Direct Sector Access 240,

fd.buffernum 267, fd.getbuffer 278, fd.setbuffer 288, fd.getsector 281, fd.setsector 290, fd.copyfirmware 269, fd.copyfirmwarelzo 270

Part	Description

mode	0- PL_FD_CSUM_MODE_VERIFY: verify the checksum. 1- PL_FD_CSUM_MODE_CALCULATE: calculate the checksum and store it into the selected RAM buffer.
csum	Indirectly returns calculated value. When the mode= 0- PL_FD_CSUM_MODE_VERIFY, this value returned will be 0 if the checksum was found to be correct, or some other value if the checksum was found to be wrong. When the mode= 1- PL_FD_CSUM_MODE_CALCULATE, the method will return a newly calculated checksum and store the same checksum into the RAM buffer.

Details

.Close Method

Function: Closes the file opened "on" a currently selected file number (selection is made through fd.filenum 273).

Syntax: **fd.close() as pl_fd_status_codes**

Returns: One of the following pl_fd_status_codes 282, also affects fd.laststatus 282:

0- PL_FD_STATUS_OK: Completed successfully.

1- PL_FD_STATUS_FAIL: Physical flash memory failure (fatal).

See Also: Closing Files 258,

fd.flush 275

Details

Invoking the method also does the job performed by the fd.flush 275 method (and this is why the 1- PL_FD_STATUS_FAIL status code may be returned here).

Attempting to invoke this method "on" a file number that did not have any opened file associated with it generates no error.

.Copyfirmware Method

Function: Copies the specified number of sectors (starting from logical sector 0) from the data area, and into the firmware/application area (starting from physical sector 0) of the flash memory, then reboots the device to make it run the new firmware/application.

Syntax: **fd.copyfirmware(**numsectors **as word)**

Returns: ---

See Also: Upgrading the Firmware/Application 243, Direct Sector Access 240

fd.buffernum 267, fd.getbuffer 278, fd.setbuffer 288, fd.getsector 281, fd.setsector 290, fd.checksum 268,

fd.copyfirmwarelzo 270

Part	Description
num sectors	Number of sectors to copy.

Details

Certain platforms do not support this method. Refer to your platform documentation 138 for details.

- **BE VERY CAREFUL!** Using the fd.copyfirmware on incorrect data will "incapacitate" your device and further remote upgrades will become impossible. You will need to physically go to your device and upload correct firmware and/or application, possibly through its serial port. Scary, huh?

.Copyfirmwarelzo Method

Function: Assumes that there is LZO-compressed firmware/application file of length bytes stored at logical sector 0 of the flash memory. Decompresses the file into the firmware/application area (starting from physical sector 0), then reboots the device to make it run the new firmware/application.

Syntax: **fd.copyfirmware(byref** length **as dword)**

Returns: ---

See Also: Upgrading the Firmware/Application 243, Direct Sector Access 240

fd.buffernum 267, fd.getbuffer 278, fd.setbuffer 288, fd.getsector 281, fd.setsector 290, fd.checksum 268, fd.copyfirmware 269

Part	Description
length	Compressed data length in bytes.

Details

Certain platforms do not support this method. Refer to your platform documentation 138 for details.

The decompression algorithm accepts files compressed with **lzo1x-999.exe** utility.

- **BE VERY CAREFUL!** Using the fd.copyfirmwarelzo on incorrect data will

"incapacitate" your device and further remote upgrades will become impossible. You will need to physically go to your device and upload correct firmware and/or application, possibly through its serial port. Scary, huh?

.Cutfromtop Method

Function: Removes a specified number of sectors from the beginning of a file opened "on" a currently selected file number (selection is made through fd.filenum 273).

Syntax: **fd.cutfromtop(**numsectors **as dword) as pl_fd_status_codes**

Returns: One of the following pl_fd_status_codes 282, also affects fd.laststatus 282:

0- PL_FD_STATUS_OK: Completed successfully.

1- PL_FD_STATUS_FAIL : Physical flash memory failure (fatal).

2- PL_FD_STATUS_CHECKSUM_ERR: Checksum error has been detected in one of the disk sectors (fatal).

3- PL_FD_STATUS_FORMAT_ERR: Disk formatting error has been detected (fatal).

8- PL_FD_STATUS_NOT_READY: The disk is not mounted.

14- PL_FD_STATUS_TRANSACTION_CAPACITY_EXCEEDED: Too many disk sectors have been modified in the cause of the current transaction (fatal).

See Also: Removing Data From Files 255,

fd.setfilesize 289

Part	Description
numsectors	Number of sectors to remove from the beginning of the file. Supplied value will be corrected downwards if exceeded the total number of sectors allocated to this file.

Details

As a result of this method invocation, the pointer will be set to 0 if the file becomes empty, or 1 if the file still has some data in it.

This method makes changes to the sectors of the flash disk. For the highest possible reliability, use disk transactions 259 when invoking this method.

.Create Method

Function: Creates a new file with the specified name and attributes.

Syntax: **fd.create(byref** name_attr **as string) as pl_fd_status_codes**

Returns: One of the following pl_fd_status_codes 282, also affects fd.laststatus 282:

0- PL_FD_STATUS_OK: Completed successfully.

1- PL_FD_STATUS_FAIL : Physical flash memory failure (fatal).

2- PL_FD_STATUS_CHECKSUM_ERR: Checksum error has been detected in one of the disk sectors (fatal).

4- PL_FD_STATUS_INV_PARAM: Invalid argument have been provided.

5- PL_FD_STATUS_DUPLICATE_NAME: File with this name already exists.

6- PL_FD_STATUS_FILE_TABLE_FULL: Maximum number of files that can be stored on the disk has been reached, new file cannot be created.

7- PL_FD_STATUS_DATA_FULL: The disk is full, new data cannot be added.

8- PL_FD_STATUS_NOT_READY: The disk is not mounted.

14- PL_FD_STATUS_TRANSACTION_CAPACITY_EXCEEDED: Too many disk sectors have been modified in the cause of the current transaction (fatal).

See Also: Creating, Deleting, and Renaming Files 250, File Names and Attributes 249,

fd.rename 285, fd.delete 272, fd.getnumfiles 281, fd.maxstoredfiles 283

Part	Description
name_attr	A string (1-56 characters), must contain a file name and, optionally, attributes separated from the file name by a space. File names are case-sensitive.

Details

This method makes changes to the sectors of the flash disk. For the highest possible reliability, use disk transactions 259 when invoking it.

.Delete Method

Function: Deletes a file with the specified file name from the flash disk.

Syntax: **fd.delete(byref** name **as string) as pl_fd_status_codes**

Returns: One of the following pl_fd_status_codes 282, also affects fd.laststatus 282:

0- PL_FD_STATUS_OK: Completed successfully.

1- PL_FD_STATUS_FAIL : Physical flash memory failure (fatal).

2- PL_FD_STATUS_CHECKSUM_ERR: Checksum error has been detected in one of the disk sectors (fatal).

3- PL_FD_STATUS_FORMAT_ERR: Disk formatting error has been detected (fatal).

8- PL_FD_STATUS_NOT_READY: The disk is not mounted.

9- PL_FD_STATUS_NOT_FOUND: File not found.

14- PL_FD_STATUS_TRANSACTION_CAPACITY_EXCEEDED: Too many disk sectors have been modified in the cause of the current transaction (fatal).

See Also: Creating, Deleting, and Renaming Files 250, , File Names and Attributes 249,

fd.create 271, fd.rename 285, fd.getnumfiles 281, fd.maxstoredfiles 283

Part	Description
name	A string (1-56 characters) with the file name. All characters after the first space encountered (excluding leading spaces) will be ignored. File names are case-sensitive.

Details

This method makes changes to the sectors of the flash disk. For the highest possible reliability, use disk transactions 259 when invoking it.

.Filenum Property

Function: Sets/returns the number of the currently selected file.

Type: Byte

Value Range: 0 to fd.maxopenedfiles 282-1. **Default**= 0 (file #0 selected).

See Also: Opening Files 253,

fd.open 284

Details

.Fileopened R/O Property

Function: Reports if any file is currently opened "on" the selected file number (selection is made through fd.filenum 273).

Type: Enum (no_yes, byte)

Value Range: 0- NO: No file is currently opened on this file number **(default)**.

1- YES: The file is currently opened on this file number.

See Also: Opening Files 253,

fd.filenum 273, fd.open 284, fd.maxopenedfiles 282

Details

.Filesize R/O Property

Function: Returns the size, in bytes, of the file opened "on" the currently selected file number (selection is made through the fd.filenum) 273. Returns zero if no file is currently opened.

Type: Dword

Value Range: 0 to "whatever is physically possible for the currently existing flash disk".

See Also: Writing To and Reading From Files 254,

fd.getdata 279, fd.setdata 289, fd.pointer 285, fd.setpointer 291

Details

.Find Method

Function: Finds the Nth instance of data satisfying selected criteria in a file opened "on" a currently selected file number (selection is made through fd.filenum 273).

Syntax: **fd.find(**frompos **as dword, byref** substr **as string,** instance **as word,** dir **as forward_back,** incr **as word,** mode **as pl_fd_find_modes) as dword**

Returns: File position (counting from one) at which the target occurrence of the substr was discovered, or 0 if the target occurrence of the substr was not found.

The method also affects the state of the fd.laststatus 282. The following status codes are possible:

0- PL_FD_STATUS_OK: Completed successfully.

1- PL_FD_STATUS_FAIL : Physical flash memory failure (fatal).

2- PL_FD_STATUS_CHECKSUM_ERR: Checksum error has been detected in one of the disk sectors (fatal).

3- PL_FD_STATUS_FORMAT_ERR: Disk formatting error has been detected (fatal).

4- PL_FD_STATUS_INV_PARAM: Invalid argument have been provided.

8- PL_FD_STATUS_NOT_READY: The disk is not mounted.

See Also: Searching Files 256

Part	Description
frompos	Starting position in a file from which the search will be conducted. File positions are counted from 1. Will be corrected automatically if out of range.
substr	The string to search for.
instance	Instance (occurrence) number to find.
dir	Search direction: 0- FORWARD: the search will be conducted from the frompos position and towards the end of the file. 1- BACK: the search will be conducted from the frompos position and towards the beginning of the file.
incr	Search position increment (or decrement for BACK searches).
mode	Search mode: 0- PL_FD_FIND_EQUAL: Find data that is equal to the substr. 1- PL_FD_FIND_NOT_EQUAL: Find data that is not equal to the substr. 2-PL_FD_FIND_GREATER: Find data with value greater than the value of the substr. 3- FIND_GREATER_EQUAL: Find data with value greater than or equal to the value of the substr. 4- PL_FD_FIND_LESSER: Find data with value less than the value of the substr. 4- PL_FD_FIND_LESSER_EQUAL: Find data with value less than or equal to the value of the substr.

Details

.Flush Method

Function: Saves back to the flash memory ("flushes") the changes made to the most recently edited file.

Syntax: **fd.close() as pl_fd_status_codes**

Returns: One of the following pl_fd_status_codes 282, also affects the status of fd.laststatus 282:

0- PL_FD_STATUS_OK: Completed successfully.

1- PL_FD_STATUS_FAIL: Physical flash memory failure (fatal).

See Also: Writing To and Reading From Files 254 (see "File Data Caching"!), Closing Files 258,

fd.close 269

Details

The fd.flush method does not depend on the current fd.filenum 273 value and works globally on any most recently changed file.

Executing fd.close 269 also does the job of fd.flush.

It is not necessary to use fd.flush if your disk operations are performed within a disk transaction 259.

.Format Method

Function:	Formats the flash memory to create a flash disk; no transaction journal 259 sectors will be allocated.
Syntax:	**fd.format(**totalsize **as word,** numstoredfiles **as byte) as pl_fd_status_codes**
Returns:	One of the following pl_fd_status_codes 282, also affects fd.laststatus 282: 0- PL_FD_STATUS_OK: Completed successfully. 1- PL_FD_STATUS_FAIL : Physical flash memory failure (fatal). 4- PL_FD_STATUS_INV_PARAM: Invalid argument have been provided for the invoked method. 16- PL_FD_STATUS_FLASH_NOT_DETECTED: Flash IC wasn't detected during boot, fd. object cannot operate normally.
See Also:	Formatting the Flash Disk 245, Checking Disk Vitals 249, fd.mount 283, fd.ready 285

Part	Description
totalsize	Desired number of sectors occupied by the disk in flash memory. Cannot exceed available space (fd.availableflashspace 267) and may be slightly corrected downwards for internal housekeeping reasons. Actual total size can be checked through the fd.totalsize 292.
maxstoredfiles	Desired maximum number of files that the disk will allow to create. Actual maximum number of files will be adjusted automatically to be a multiple of four and not exceed 64 (for example, specifying 6 will result in the actual value of 8). If you specify 0 you will get 4 files.

Details

Fd.format will not allocate and transaction journal sectors, so disk transactions 259 will be impossible. Use fd.formatj 277 (recommended) to create the disk that will support transaction.

After formatting the disk will be in the dismounted state and will need to be mounted (see fd.mount 283) before any disk-related activity can be successfully performed.

.Formatj Method

Function: Formats the flash memory to create a flash disk.

Syntax: **fd.format(**totalsize **as word,** numstoredfiles **as byte,** maxjournalsectors **as byte) as pl_fd_status_codes**

Returns: One of the following pl_fd_status_codes 282, also affects fd.laststatus 282:

0- PL_FD_STATUS_OK: Completed successfully.

1- PL_FD_STATUS_FAIL : Physical flash memory failure (fatal).

4- PL_FD_STATUS_INV_PARAM: Invalid argument have been provided for the invoked method.

16- PL_FD_STATUS_FLASH_NOT_DETECTED: Flash IC wasn't detected during boot, fd. object cannot operate normally.

See Also: Formatting the Flash Disk 245, Checking Disk Vitals 249, fd.mount 283, fd.ready 285

Part	Description
totalsize	Desired number of sectors occupied by the disk in flash memory. Cannot exceed available space (fd.availableflashspace 267) and may be slightly corrected downwards for internal housekeeping reasons. Actual total size can be checked through the fd.totalsize 292.
maxstoredfiles	Desired maximum number of files that the disk will allow to create. Actual maximum number of files will be adjusted automatically to be a multiple of four and not exceed 64 (for example, specifying 6 will result in the actual value of 8). If you specify 0 you will get 4 files.
maxjournalsectors	Number of sectors to allocate for the transaction journal 259. Give this disk area a generous number of sectors, possibly 50-100. Do not set below 17 -- this will limit the transaction capacity of the journal. Setting this parameter to 0 or 1 disables disk transactions completely.

Details

After formatting the disk will be in the dismounted state and will need to be mounted (see fd.mount 283) before any disk-related activity can be successfully performed.

.Getattributes Method

Function: Returns the attributes string for a file with the specified file name.

Syntax: **fd.getattributes(byref** name **as string) as string**

Returns: The string with attributes (may be up to 54 characters long) or an empty string if no attributes were set for this file. The method also affects the state of fd.laststatus 282. The following status codes are possible:

0- PL_FD_STATUS_OK: Completed successfully.

1- PL_FD_STATUS_FAIL : Physical flash memory failure (fatal).

2- PL_FD_STATUS_CHECKSUM_ERR: Checksum error has been detected in one of the disk sectors (fatal).

3- PL_FD_STATUS_FORMAT_ERR: Disk formatting error has been detected (fatal).

8- PL_FD_STATUS_NOT_READY: The disk is not mounted.

9- PL_FD_STATUS_NOT_FOUND: File not found.

See Also: Reading and Writing File Attributes 251, Creating, Deleting, and Renaming Files 250, File Names and Attributes 249,

fd.setattributes 287, fd.create 271

Part	Description
name	A string (1-56 characters) with the file name. All characters after the first space encountered (excluding leading spaces) will be ignored. File names are case-sensitive.

Details

.Getbuffer Method

Function: Reads the specified number of bytes from the currently selected RAM buffer of the flash memory (selection is made through the fd.buffernum 267 property).

Syntax: **fd.getbuffer(**offset **as word,** len **as word) as string**

Returns: The string with the data from the buffer.

See Also: Direct Sector Access 240,

fd.buffernum 267, fd.setbuffer 288, fd.getsector 281, fd.setsector 290, fd.checksum 268, fd.copyfirmware 269, fd.copyfirmwarelzo 270

Part	Description
offset	Starting offset in the buffer. Possible value range is 0-263 (the buffer stores 264 bytes of data, offset is counted from 0).
len	Number of bytes to read. The length of returned data will depend on one of three factors, whichever is smaller: *len* argument, amount of data still available in the buffer counting from the *offset* position, and the capacity of receiving string variable.

Details

.Getdata Method

Function: Reads a specified number of bytes from the file opened "on" a currently selected file number (selection is made through fd.filenum 273). The data is read starting at the fd.pointer 285 position.

Syntax: **fd.getdata(**maxinplen **as byte) as string**

Returns: The string with the data read from the file. The method also affects the state of fd.laststatus 282. The following status codes are possible:

0- PL_FD_STATUS_OK: Completed successfully.

1- PL_FD_STATUS_FAIL : Physical flash memory failure (fatal).

2- PL_FD_STATUS_CHECKSUM_ERR: Checksum error has been detected in one of the disk sectors (fatal).

3- PL_FD_STATUS_FORMAT_ERR: Disk formatting error has been detected (fatal).

8- PL_FD_STATUS_NOT_READY: The disk is not mounted.

10- PL_FD_STATUS_NOT_OPENED: No file is currently opened "on" the current value of the fd.filenum 273 property.

See Also: Writing To and Reading From Files 254, fd.setdata 289, fd.pointer 285, fd.setpointer 291, fd.filesize 274

Part	Description
maxinplen	Maximum number of bytes to read from the file. The length of returned data will depend on one of three factors, whichever is smaller: maxinplen argument, amount of data still available in the file counting from the current pointer position, and the capacity of receiving string variable.

Details

As a result of this method invocation, the pointer will be advanced forward by the

number of bytes actually read from the file.

.Getfreespace Method

Function: Returns the total number of free data sectors available on the flash disk.

Syntax: **fd.getfreespace() as word**

Returns: 0-65535, also affects the state of fd.laststatus[282]. The following status codes are possible:

0- PL_FD_STATUS_OK: Completed successfully.

1- PL_FD_STATUS_FAIL : Physical flash memory failure (fatal).

2- PL_FD_STATUS_CHECKSUM_ERR: Checksum error has been detected in one of the disk sectors (fatal).

3- PL_FD_STATUS_FORMAT_ERR: Disk formatting error has been detected (fatal).

8- PL_FD_STATUS_NOT_READY: The disk is not mounted.

See Also: Checking Disk Vitals[249],

fd.capacity[268], fd.numservicesectors[284], fd.totalsize[292], fd.maxstoredfiles[283], fd.getnumfiles[281]

Details

One data sectors carries 256 bytes of file data. Each existing file occupies at least one data sector, even if the file is empty.

The disk must be mounted (see fd.mount[283]) for this property to return a meaningful value.

.Getnextdirmember Method

Function: Returns the next filename (if any) found in the disk directory.

Syntax: **fd.getnextdirmember() as string**

Returns: The string containing the file name of the next directory member or an empty string if all file names have already been returned. The method also affects the state of fd.laststatus[282]. The following status codes are possible:

0- PL_FD_STATUS_OK: Completed successfully.

1- PL_FD_STATUS_FAIL : Physical flash memory failure (fatal).

2- PL_FD_STATUS_CHECKSUM_ERR: Checksum error has been detected in one of the disk sectors (fatal).

3- PL_FD_STATUS_FORMAT_ERR: Disk formatting error has been detected (fatal).

8- PL_FD_STATUS_NOT_READY: The disk is not mounted.

See Also: Walking Through File Directory[252],

fd.resetdirpointer[286], fd.getnumfiles[281]

Details

.Getnumfiles Method

Function: Returns the total number of files currently stored on the disk.

Syntax: **fd.getfreespace() as word**

Returns: 0-65535, also affects the state of fd.laststatus 282. The following status codes are possible:

0- PL_FD_STATUS_OK: Completed successfully.

1- PL_FD_STATUS_FAIL : Physical flash memory failure (fatal).

2- PL_FD_STATUS_CHECKSUM_ERR: Checksum error has been detected in one of the disk sectors (fatal).

3- PL_FD_STATUS_FORMAT_ERR: Disk formatting error has been detected (fatal).

8- PL_FD_STATUS_NOT_READY: The disk is not mounted.

See Also: Creating, Deleting, and Renaming Files 250, Walking Through File Directory 252, Checking Disk Vitals 249,

fd.capacity 268, fd.numservicesectors 284, fd.totalsize 292, fd.getfreespace 280, fd.maxstoredfiles 283

Details

The disk must be mounted (see fd.mount 283) for this property to return a meaningful value.

.Getsector Method

Function: Reads the entire 264 bytes from the specified sector into the currently selected RAM buffer of the flash memory (selection is made through the fd.buffernum 267 property).

Syntax: **fd.getsector(**num **as word) as pl_fd_status_codes**

Returns: One of the following pl_fd_status_codes 282, also affects the state of fd.laststatus 282:

0- PL_FD_STATUS_OK: Completed successfully.

1- PL_FD_STATUS_FAIL : Physical flash memory failure (fatal).

16- PL_FD_STATUS_FLASH_NOT_DETECTED: Flash IC wasn't detected during boot, fd. object cannot operate normally.

See Also: Direct Sector Access 240, File-based and Direct Sector Access Coexistence 263,

fd.buffernum 267, fd.getbuffer 278, fd.setbuffer 288, fd.setsector 290, fd.checksum 268, fd.copyfirmware 269, fd.copyfirmwarelzo 270

Part	Description
num	*Logical* number of the sector to read from (logical numbers are in reverse: reading from the logical sector 0 actually means reading from the last physical sector of the flash IC).

Details

All file-based 245 operations of the flash disk also load data into the RAM buffers. Switch to the RAM buffer #0 *each time* before performing direct sector access 240 with this or other related methods -- this will guarantee that you won't corrupt the files and/or the file system and cause disk dismounting (fd.ready 285 becoming 0-NO).

This method always accesses the actual specified target sector and not its cached copy even if the disk transaction 259 is in progress (fd.transactionstarted 294 = 1-YES) and the target sector has been cached already.

.Laststatus R/O Property

Function: Returns the execution result for the most recent disk-related method execution.

Type: Enum (pl_fd_status_code, byte)

Value Range: Many different fatal and non-fatal conditions are returned -- see Fd. Object's status codes 239 for details.

See Also: Fd. Object's Status Codes 239

Details

Some methods, such as fd.create 271, return execution status directly. For those, the fd.laststatus will contain the same status as the one directly returned.

Other methods return some other data. For example, fd.getdata 279 returns the data requested (or an empty string if something went wrong). The execution result for such methods can only be verified through this R/O property.

Note that some errors are fatal and the disk is dismounted (fd.ready 285 is set to 0-NO) immediately upon the detection of any such fatal error.

.Maxopenedfiles R/O Property

Function: Returns the total number of files that can be simultaneously opened by your application.

Type: Byte

Value Range: Platform-dependent.

See Also: Opening Files 253,

fd.filenum 273, fd.open 284, fd.fileopened 273

Details

The value of this property depends on the hardware (selected platform 138) and has nothing to do with the formatting of your flash disk.

.Maxstoredfiles R/O Property

Function: Returns the total number of files that can be simultaneously stored on the currently existing flash disk.

Type: Byte

Value Range: Value depends on the current disk formatting.

See Also: Formatting the Flash Disk 245, Creating, Deleting, and Renaming Files 250, Checking Disk Vitals 249,

fd.capacity 268, fd.numservicesectors 284, fd.totalsize 292, fd.getfreespace 280, fd.getnumfiles 281

Details

This number cannot be changed unless the disk is reformatted (see fd.formatj) 277.

The disk must be mounted (see fd.mount 283) for this property to return a meaningful value.

.Mount Method

Function: Mounts (prepares for use) the flash disk already existing in the flash memory.

Syntax: **fd.format() as pl_fd_status_codes**

Returns: One of the following pl_fd_status_codes 282, also affects fd.laststatus 282:

0- PL_FD_STATUS_OK: Completed successfully.

1- PL_FD_STATUS_FAIL : Physical flash memory failure (fatal).

1- PL_FD_STATUS_FAIL : Physical flash memory failure (fatal).

2- PL_FD_STATUS_CHECKSUM_ERR: Checksum error has been detected in one of the disk sectors (fatal).

3- PL_FD_STATUS_FORMAT_ERR: Disk formatting error has been detected (fatal).

16- PL_FD_STATUS_FLASH_NOT_DETECTED: Flash IC wasn't detected during boot, fd. object cannot operate normally.

See Also: Mounting the Flash Disk 249,

fd.formatj 277, fd.ready 285

Details

This method also finishes the transaction commit job 259 if it was started with the fd.transactioncommit 293 method and wasn't completed properly due to the power failure or some other reason.

.Numservicesectors R/O Property

Function: Returns the total number of sectors occupied by the "housekeeping" data of the currently existing flash disk.

Type: Byte

Value Range: Value depends on the current disk formatting.

See Also: Checking Disk Vitals 249, Disk Area Allocation Details 246, fd.capacity 268, fd.totalsize 292, fd.getfreespace 280, fd.maxstoredfiles 283, fd.getnumfiles 281

Details

The disk must be mounted (see fd.mount 283) for this property to return a meaningful value.

.Open Method

Function: Opens a file with a specified name "on" a currently selected file number (selection is made through fd.filenum 273).

Syntax: **fd.open(byref** name **as string) as pl_fd_status_codes**

Returns: One of the following pl_fd_status_codes 282, also affects fd.laststatus 282:

0- PL_FD_STATUS_OK: Completed successfully.

1- PL_FD_STATUS_FAIL : Physical flash memory failure (fatal).

2- PL_FD_STATUS_CHECKSUM_ERR: Checksum error has been detected in one of the disk sectors (fatal).

3- PL_FD_STATUS_FORMAT_ERR: Disk formatting error has been detected (fatal).

8- PL_FD_STATUS_NOT_READY: The disk is not mounted.

9- PL_FD_STATUS_NOT_FOUND: File not found.

11- PL_FD_STATUS_ALREADY_OPENED: This file is already opened "on" some other file number.

See Also: Opening Files 253,

fd.filenum 273, fd.fileopened 273, fd.maxopenedfiles 282

Part	Description
name	A string (1-56 characters) with the file name. All characters after the first space encountered (excluding leading spaces) will be ignored. File names are case-sensitive.

Details

.Pointer R/O Property

Function: Returns the pointer position for the file opened "on" the currently selected file number (selection is made through fd.filenum 273). Returns zero if no file is currently opened or the file is empty.

Type: Dword

Value Range: 0 to fd.filesize 274+1 (except for fd.filesize= 0, in which case fd.pointer= 0 too).

See Also: Writing To and Reading From Files 254, fd.getdata 279, fd.setdata 289, fd.setpointer 291, fd.filesize 274

Details

.Ready R/O Property

Function: Informs whether the flash disk is mounted and ready for use.

Type: Enum (no_yes, byte)

Value Range: 0- NO: The disk is not mounted and not ready for use **(default)**.

1- YES: The disk is mounted and ready for use.

See Also: Mounting the Flash Disk 249, fd.formatj 277, fd.mount 283

Details

.Rename Method

Function: Renames a file specified by its name.

Syntax: **fd.create(byref** old_name **as string, byref** new_name **as**

string) as pl_fd_status_codes

Returns: One of the following pl_fd_status_codes 282, also affects fd.laststatus 282:

0- PL_FD_STATUS_OK: Completed successfully.

1- PL_FD_STATUS_FAIL : Physical flash memory failure (fatal).

2- PL_FD_STATUS_CHECKSUM_ERR: Checksum error has been detected in one of the disk sectors (fatal).

4- PL_FD_STATUS_INV_PARAM: Old_name is a NULL string, which is not allowed.

5- PL_FD_STATUS_DUPLICATE_NAME: File with the new_name already exists.

8- PL_FD_STATUS_NOT_READY: The disk is not mounted.

9- PL_FD_STATUS_NOT_FOUND: The old_name file is not found.

14- PL_FD_STATUS_TRANSACTION_CAPACITY_EXCEEDED: Too many disk sectors have been modified in the cause of the current transaction (fatal).

See Also: Creating, Deleting, and Renaming Files 250, File Names and Attributes 249,

fd.create 271, fd.delete 272, fd.getnumfiles 281, fd.maxstoredfiles 283

Part	Description
old_name	A string (1-56 characters) with the name of the file to be renamed. All characters after the first space encountered (excluding leading spaces) will be ignored. File names are case-sensitive.
new_name	A string (1-56 characters) with the new name for the file. All characters after the first space encountered (excluding leading spaces) will be ignored.

Details

Renaming a file preserves file attributes.

This method makes changes to the sectors of the flash disk. For the highest possible reliability, use disk transactions 259 when invoking it.

.Resetdirpointer Method

Function: Resets the directory pointer to zero.

Syntax: **fd.resetdirpointer()**

Returns: ---

See Also: Walking Through File Directory 252,

fd.getnextdirmember 280, fd.getnumfiles 281

Details

.Sector R/O Property

Function: Returns the physical sector number corresponding to the current position of the file pointer.

Type: Word

Value Range: 0-65535

See Also: Writing To and Reading From Files 254

Details

Because the sectors belonging to a given file may be scattered around the flash disk, there is no simple way to figure out the number of the physical sector corresponding to the current file pointer position (see fd.pointer 285). This property exists purely for informational purposes. There is no real need for you to know where the fd. object stores your data.

.Setattributes Method

Function: Sets the attributes string for a file with the specified file name.

Syntax: **fd.setattributes(byref** name **as string, byref** attr **as string) as pl_fd_status_codes**

Returns: One of the following pl_fd_status_codes 282, also affects fd.laststatus 282:

0- PL_FD_STATUS_OK: Completed successfully.

1- PL_FD_STATUS_FAIL : Physical flash memory failure (fatal).

2- PL_FD_STATUS_CHECKSUM_ERR: Checksum error has been detected in one of the disk sectors (fatal).

3- PL_FD_STATUS_FORMAT_ERR: Disk formatting error has been detected (fatal).

8- PL_FD_STATUS_NOT_READY: The disk is not mounted.

9- PL_FD_STATUS_NOT_FOUND: File not found.

14- PL_FD_STATUS_TRANSACTION_CAPACITY_EXCEEDED: Too many disk sectors have been modified in the cause of the current transaction (fatal).

See Also: Reading and Writing File Attributes 251, Creating, Deleting, and Renaming Files 250, File Names and Attributes 249, fd.getattributes 278, fd.create 271

Part	Description
name	A string (1-56 characters) with the file name. All characters after the first space encountered (excluding leading spaces) will be ignored. File names are case-sensitive.
attr	A string with attributes to be set. Attributes length cannot exceed 56-length_of_the_file_name-1. This "-1" accounts for the space character that separates the file name from the attributes.

Details

This method makes changes to the sectors of the flash disk. For the highest possible reliability, use disk transactions[259] when invoking it.

.Setbuffer Method

Function:	Writes a specified number of bytes into the currently selected RAM buffer of the flash memory (selection is made through the fd.buffernum[267] property).
Syntax:	**fd.setbuffer(byref** data **as string,** offset **as word) as word**
Returns:	Actual number of bytes written.
See Also:	Direct Sector Access[240], File-based and Direct Sector Access Coexistence[263] fd.buffernum[267], fd.getbuffer[278], fd.getsector[281], fd.setsector[290], fd.checksum[268], fd.copyfirmware[269], fd.copyfirmwarelzo[270]

Part	Description
data	A string with the data to be written to the buffer.
offset	Starting offset in the buffer. Possible value range is 0-263 (the buffer stores 264 bytes of data, offset is counted from 0).

Details

The length of data actually written into the buffer may be limited if all supplied data can't fit between the offset position in the buffer and the end of the buffer.

All file-based[245] operations of the flash disk also load data into the RAM buffers. Switch to the RAM buffer #0 *each time* before performing direct sector access[240] with this or other related methods -- this will guarantee that you won't corrupt the files and/or the file system and cause disk dismounting (fd.ready[285] becoming 0-NO).

.Setdata Method

Function: Writes data to a file opened "on" a currently selected file number (selection is made through fd.filenum[273]). The data is written starting at the fd.pointer[285] position.

Syntax: **fd.setdata(byref** data **as string) as pl_fd_status_codes**

Returns: One of the following pl_fd_status_codes[282], also affects fd.laststatus[282]:

0- PL_FD_STATUS_OK: Completed successfully.

1- PL_FD_STATUS_FAIL : Physical flash memory failure (fatal).

2- PL_FD_STATUS_CHECKSUM_ERR: Checksum error has been detected in one of the disk sectors (fatal).

3- PL_FD_STATUS_FORMAT_ERR: Disk formatting error has been detected (fatal).

7- PL_FD_STATUS_DATA_FULL: The disk is full, new data cannot be added.

8- PL_FD_STATUS_NOT_READY: The disk is not mounted.

10- PL_FD_STATUS_NOT_OPENED: No file is currently opened on the current value of the fd.filenum[273] property.

14- PL_FD_STATUS_TRANSACTION_CAPACITY_EXCEEDED: Too many disk sectors have been modified in the cause of the current transaction (fatal).

See Also: Writing To and Reading From Files[254], fd.getdata[279], fd.pointer[285], fd.setpointer[291], fd.filesize[274]

Part	Description
data	A string containing data to be written to the file. If the disk becomes full, then no data will be written (and not just the portion that could not fit).

Details

As a result of this method invocation, the pointer will be advanced forward by the number of bytes actually written to the file.

This method makes changes to the sectors of the flash disk. For the highest possible reliability, use disk transactions[259] when invoking this method.

.Setfilesize Method

Function: Sets (reduces) the file size of a file opened "on" a currently selected file number (selection is made through fd.filenum[273]).

Syntax: **fd.setfilesize(**newsize **as dword) as pl_fd_status_codes**

Returns: One of the following pl_fd_status_codes[282], also affects fd.laststatus[282]:

0- PL_FD_STATUS_OK: Completed successfully.

1- PL_FD_STATUS_FAIL : Physical flash memory failure (fatal).

2- PL_FD_STATUS_CHECKSUM_ERR: Checksum error has been detected in one of the disk sectors (fatal).

3- PL_FD_STATUS_FORMAT_ERR: Disk formatting error has been detected (fatal).

8- PL_FD_STATUS_NOT_READY: The disk is not mounted.

14- PL_FD_STATUS_TRANSACTION_CAPACITY_EXCEEDED: Too many disk sectors have been modified in the cause of the current transaction (fatal).

See Also: Removing Data From Files 255, fd.cutfromtop 271

Part	Description
new size	Desired new file size in bytes. Supplied value will be corrected downwards if exceeded previous file size.

Details

As a result of this method invocation, the pointer position may be corrected downwards.

This method makes changes to the sectors of the flash disk. For the highest possible reliability, use disk transactions 259 when invoking this method.

.Setsector Method

Function: Writes the entire 264 bytes of the specified sector with the data from the currently selected RAM buffer of the flash memory (selection is made through the fd.buffernum 267 property).

Syntax: **fd.getsector(**num **as word) as pl_fd_status_codes**

Returns: One of the following pl_fd_status_codes 282, also affects fd.laststatus 282:

0- PL_FD_STATUS_OK: Completed successfully.

1- PL_FD_STATUS_FAIL: Physical flash memory failure (fatal).

4- PL_FD_STATUS_INV_PARAM: specified sector number is in the firmware/application area of the flash memory; access has been denied.

16- PL_FD_STATUS_FLASH_NOT_DETECTED: Flash IC wasn't detected during boot, fd. object cannot operate normally.

See Also: Direct Sector Access 240, Sharing Flash Between Your Application and Data 237, File-based and Direct Sector Access Coexistence 263,

fd.buffernum 267, fd.getbuffer 278, fd.setbuffer 288, fd.getsector 281, fd.checksum 268, fd.copyfirmware 269, fd.copyfirmwarelzo 270

Part	Description
num	*Logical* number of the sector to write to (logical numbers are in reverse: writing to the logical sector 0 actually means writing to the last physical sector of the flash IC). Acceptable range is 0 - fd.availableflashspace 267 - 1.

Details

The data area of the flash memory may house a formatted flash disk 245. Writing to the sector that belongs to the flash disk when the disk is mounted will automatically dismount the disk (set fd.ready 285 = 0- NO) and may render the disk unusable.

This method always accesses the specified target sector and not its cached copy even if the disk transaction 259 is in progress (fd.transactionstarted 294 = 1- YES) and the target sector has been cached already.

.Setpointer Method

Function: Sets the new pointer position for a file opened "on" a currently selected file number (selection is made through fd.filenum 273).

Syntax: **fd.setpointer(**pos **as dword) as pl_fd_status_codes**

Returns: One of the following pl_fd_status_codes 282, also affects fd.laststatus 282:

0- PL_FD_STATUS_OK: Completed successfully.

1- PL_FD_STATUS_FAIL : Physical flash memory failure (fatal).

2- PL_FD_STATUS_CHECKSUM_ERR: Checksum error has been detected in one of the disk sectors (fatal).

3- PL_FD_STATUS_FORMAT_ERR: Disk formatting error has been detected (fatal).

8- PL_FD_STATUS_NOT_READY: The disk is not mounted.

See Also: Writing To and Reading From Files 254, fd.getdata 279, fd.setdata 289, fd.pointer 285, fd.filesize 274

Part	Description
pos	Desired new pointer position. Supplied value will be corrected if out of range. For the files of 0 size (see fd.filesize 274), the pointer may only have one value -- 0. If the file has non-zero size, the pointer can be between 1 and fd.filesize+1. "1" is the position of the first byte of the file. The last existing byte of the file is at position equal to the value of

fd.filesize. "Fd.filesize+1" is the position at which new data can be added to the file.

Details

.Totalsize R/O Property

Function: Returns the total number of sectors occupied by the currently existing flash disk.

Type: Word

Value Range: 0-65535

See Also: Checking Disk Vitals 249,
fd.capacity 268, fd.numservicesectors 284, fd.getfreespace 280, fd.maxstoredfiles 283, fd.getnumfiles 281

Details

For internal reasons 246, the total size of the disk as returned by this property may be less than the size that was requested when formatting the disk with the fd.formatj 277 (fd.format 276) method. Actual usable capacity (fd.capacity 268) of the disk is less because the disk also needs a number of sectors for its "housekeeping" data (see fd.numservicesectors 284).

The disk must be mounted (see fd.mount 283) for this property to return a meaningful value.

.Transactioncapacityremaining R/O Property

Function: Returns the number of sectors that can still be changed in the cause of the current disk transaction.

Type: Byte

Value Range: 0 to 16 or maxjournalsectors-1, whichever is smaller. Maxjournalsectors is the argument of the fd.formatj 277 method.

See Also: Using Disk Transactions 259, Understanding Transaction Capacity 261
fd.transactionstart 293, fd.transactioncommit 293, fd.transactionstarted 294

Details

.Transactioncommit Method

Function: Commits a disk transaction.

Syntax: **fd.transactioncommit() as pl_fd_status_codes**

Returns: One of the following pl_fd_status_codes 282, also affects fd.laststatus 282:

0- PL_FD_STATUS_OK: Completed successfully.

1- PL_FD_STATUS_FAIL : Physical flash memory failure (fatal).

8- PL_FD_STATUS_NOT_READY: The disk is not mounted.

13- PL_FD_STATUS_TRANSACTION_NOT_YET_STARTED: Disk transaction hasn't been started yet.

15- PL_FD_STATUS_TRANSACTIONS_NOT_SUPPORTED: The disk wasn't formatted to support transactions (use fd.formatj 277 with maxjournalsectors>1 to enable transactions).

See Also: Using Disk Transactions 259,

fd.transactionstart 293, fd.transactionstarted 294, fd.transactioncapacityremaining 292

Details

.Transactionstart Method

Function: Starts a disk transaction.

Syntax: **fd.transactionstart() as pl_fd_status_codes**

Returns: One of the following pl_fd_status_codes 282, also affects fd.laststatus 282:

0- PL_FD_STATUS_OK: Completed successfully.

8- PL_FD_STATUS_NOT_READY: The disk is not mounted.

12- PL_FD_STATUS_TRANSACTION_ALREADY_STARTED: Disk transaction has already been started (and cannot be restarted).

15- PL_FD_STATUS_TRANSACTIONS_NOT_SUPPORTED: The disk wasn't formatted to support transactions (use fd.formatj 277 with maxjournalsectors>1 to enable transactions).

See Also: Using Disk Transactions 259,

fd.transactioncommit 293, fd.transactionstarted 294, fd.transactioncapacityremaining 292

Details

.Transactionstarted R/O Property

Function:	Reports whether a disk transaction is currently in progress.
Type:	Enum (no_yes, byte)
Value Range:	0- NO: Disk transaction is not in progress **(default)**. 1- YES: Disk transaction is in progress.
See Also:	Using Disk Transactions 259, fd.transactionstart 293, fd.transactioncommit 293, fd.transactioncapacityremaining 292

Details

IO Object

The io object controls I/O lines, ports, and interrupt lines of your device.

Overview 1.4.1

The I/O object controls your device's individual I/O lines and ports. Each port groups eight I/O lines. You can control the state of each line or entire port.

The list of available I/O lines and ports is platform-specific, i.e. it depends on your device. Two enum constant sets -- pl_io_num and pl_io_port_num -- define the set of available lines and ports. You can find the declaration of these enums in your platform documentation (for example, EM1000's is here 143).

There are two different methods for controlling I/O lines (ports):

- Method with pre-selection 295;
- Method without pre-selection 295.

I/O lines of your device can work as both inputs and outputs -- Controlling Output Buffers 296 topic explains this.

Some I/O lines can work as interrupts -- see Working With Interrupts 297 for explanation. The list of available interrupt lines is platform-specific. The pl_int_num enum constant set defines the set of related variables. Again, please refer to your platform documentation.

On many devices, a number of I/O lines may be shared with inputs/outputs of special function blocks (serial ports, etc.). When a special function block is enabled, I/O lines it uses cannot (should not) be manipulated though the io object.

Line/Port Manipulation With Pre-selection

I/O line manipulation with pre-selection works like this: you first select the line you want to work with using the io.num 301 property. You can then read and set the state of this line using the io.state 303 property. On certain platforms, you can also enable/disable the output buffer of the line with the io.enabled 298 property -- more on this in the Controlling Output Buffers 296 topic. Here is a code example:

```
io.num=PL_IO_NUM_5 'select line #5
io.enabled=YES 'enable this line
io.state=HIGH 'set this line to HIGH
io.state=LOW 'now set this line to LOW
io.enabled=NO 'configure the line as input now
x=io.state 'read the state of the line
```

Same can be done with ports -- use io.portnum 302, io.portstate 303, and io.portenabled 302 properties to achieve this:

```
io.portnum=2 'select port #2
io.portenabled=&hFF 'this means that every port line's output buffer will be
enabled (&hFF=&b11111111)
'output &h55: port lines 0, 2, 4, and 6 will be at HIGH, 4 remaining lines
will be at LOW (&h55=&b01010101)
io.portstate=&h55
'output another value
op.portstate=0
'configure the port for input and read its state
io.portenabled=0
x=io.portstate
```

This way of controlling the lines/ports of your device is good when you are going to manipulate the same line/port repeatedly. Performance is improved because you select the line/port once and then address this line/port as many times as you need.

Note that I/O line and port names are platform-specific and are defined by pl_io_num and pl_io_port_num enums respectively. The declarations for these enums can be found in your device's platform documentation (for example, EM1000's is here 143).

On many devices, a number of I/O lines may be shared with inputs/outputs of special function blocks (serial ports, etc.). When a special function block is enabled, I/O lines it uses cannot (should not) be manipulated though the io object.

Line/Port Manipulation Without Pre-selection

I/O line manipulation without pre-selection is good when you need to deal with several I/O lines at once. For this, use io.lineset 300, io.lineget 300, and io.invert 300 methods. These methods require the line number to be supplied directly, as a parameter. Therefore, pre-selection with the io.num 301 property is not necessary.

Moreover, executing these methods leaves the io.num value intact.

Here is an example of a serialized clock/data output. The clock line is PL_IO_NUM_0 and the data line is PL_IO_NUM_1. Notice how this is implemented -- clock line is preselected once, then set LOW and HIGH using the io.state 303 property. Meanwhile, the data line is updated using the io.lineset method. The variable x supposedly carries a bit of data to be output (where x gets is data is not shown).

```
Dim f As Byte
Dim x As low_high

io.num=PL_IO_NUM_0 'pre-select the clock line
For f=0 To 7
   ... 'obtain the value of the next bit, put it into x (not shown)
   io.state=LOW 'set the clock line LOW (the clock line has already been
preselected)
   io.lineset(PL_IO_NUM_1,x) 'output the next data bit
   io.state=HIGH 'set the clock line HIGH (the clock line has already been
preselected)
Next f
```

Direct port manipulation is achieved using io.portset 303 and io.portget 302 methods.

Note that I/O line and port names are platform-specific and are defined by pl_io_num and pl_io_port_num enums respectively. The declarations for these enums can be found in your device's platform documentation (for example, EM1000's is here 143).

On many devices, a number of I/O lines may be shared with inputs/outputs of special function blocks (serial ports, etc.). When a special function block is enabled, I/O lines it uses cannot (should not) be manipulated though the io object.

Controlling Output Buffers

As far as the I/O line/port control goes, there are two kinds 194 of Tibbo devices and corresponding platforms -- those without output buffer control, and those with output buffer control. You can find this information in your device's platform documentation (for example, EM1000's is here 143).

On devices without the output buffer control each I/O line's output driver is always enabled. If you want to use this line as an input, set its state to HIGH. After that, you can read this line's state. If the line is left unconnected or is not being pulled low externally you will read HIGH. If the line is being pulled low externally you will read LOW. Pulling the line LOW externally while this line's output driver is at HIGH will do no damage to the line.

Here is a code example in which we wait for the line #1 to become LOW:

```
io.num=PL_IO_NUM_1 'select the line
io.state=HIGH 'set it to HIGH so we can use it as an input
While io.state=HIGH 'wait until the line is pulled LOW externally
Wend
```

On devices with explicit output buffer control, you need to define whether the line

is an output (set io.enabled 298 = 1- YES) or input (set io.enabled 298 = 0- NO). Trying to read the line while it is in the output mode will simply return the state of the line's own output driver. Forcing the line externally while it works as an output may cause a permanent damage to the device. For this kind of devices, the above code must be modified to look like this:

```
io.num=PL_IO_NUM_1 'select the line
io.enabled=NO 'disable the output driver (line will function as an input
now)
While io.state=HIGH 'wait until the line is pulled LOW externally
Wend
```

Since ports consist of individual lines, the same applies to ports as well. What needs to be understood is that each port line can be configured as input or output individually. Hence, a particular port doesn't have to be "all outputs" or "all inputs". Here is an example where the lower half of the port lines is configured for output, and the rest of the lines serve as inputs:

```
'This is an example for devices with explicit output buffer control
io.portnum=PL_IO_PORT_1
io.portenabled= &h0F 'configure lower 4 lines as outputs, the rest will be
used as inputs
```

Some I/O lines are shared with inputs/outputs of special function blocks (serial ports, etc.). When a special function block is enabled, certain (not all) I/O lines it uses may be configured for input or output automatically. For such lines, when the corresponding special function block is disabled, the state of the output buffer is restored automatically to what it used to be prior to enabling this function block.

Note that I/O line and port names are platform-specific and are defined by pl_io_num and pl_io_port_num enums respectively. The declarations for these enums can be found in your device's platform documentation (for example, EM1000's is here 143).

Working With Interrupts

Some platforms have a number of I/O lines that can work as interrupt inputs. To enable a particular interrupt line, select it using the io.intnum 299 property and enable the line with the io.intenabled 299 property:

```
'enable interrupt line #0
io.intnum=PL_INT_NUM_0
io.intenabled=YES
```

Once the line has been enabled, the change in this line's state will generate an

on_io_int 301 event. The linestate argument of this event is bit-encoded: each bit of the value represents one interrupt line. For the PL_INT_NUM_0, the corresponding bit is bit 0, for PL_INT_NUM_1 -- bit 1, and so on. A particular bit of the linestate argument is set when the state change (from LOW to HIGH, or from HIGH to LOW) has been detected on the related interrupt line. Event handler for the on_io_int 301 event can then determine what triggered the interrupt:

```
Sub On_io_int(linestate As Byte)
   'check if it is the interrupt line #0 that has caused the interrupt
   If linestate And &h01 <>0 Then
      'yes, interrupt has been triggered by the interrupt line #0
      ...
   End If
End Sub
```

Please, note that the word "interrupt" is used here is a somewhat loose sense. On traditional microcontrollers, interrupt line status change causes a near-instantaneous pause to the execution of the main code and a jump to an "interrupt routine". Hence, the term "interrupt" -- the execution of the main code gets interrupted.

With the io object, the interrupt line state change does not disrupt the execution of the any event handler or reordering of pending events in the event queue 71. The on_io_int 301 event is added to the end of the event queue and is not handled until all earlier events are processed. Therefore, nothing is actually interrupted. Note also that there is no guaranteed interrupt response speed here -- the time between line state change detection and the execution of the on_io_int 301 event handler depends on the number of prior events waiting in the queue and, hence, cannot be pre-determined with any certainty.

Further, there may be only one on_io_int event waiting in the event queue. Another such event will not be generated unless the previous one is processed (this prevents the event queue from getting overflown). Therefore, some short-lived state changes may remain undetected.

Bottom line: the "interrupts" of the io object should be viewed as a more convenient alternative to programmatic polling of the I/O lines.

Note that interrupt line names, such as "PL_INT_NUM_0" are defined by the pl_int_num enum, which is platform-specific. The declaration of this enum can be found in the your device's platform documentation (for example, EM1000's is here 143).

Properties, Events, Methods

This section provides an alphabetical list of all properties, methods, and events of the io object.

.Enabled Property (Selected Platforms Only)

Function:	Sets/returns the state of the output buffer for the currently selected I/O line (selection is made through the io.num 301 property).
Type:	Enum (no_yes, byte)

Value Range:	0- NO **(default)**: disabled, I/O line works as an input 1- YES: enabled, I/O line works as an output
See Also:	Controlling Output Buffers 296, io.state 303

Details

Depending on the platform, explicit configuration 194 of the output buffers may or may not be required. For information on your device, see its platform documentation (for example, EM1000's is here 143). The property is either not available or has no effect on platforms that do not require buffer configuration.

.Intenabled Property

Function:	Enabled/disables currently selected interrupt line (selection is made through the io.intnum 299 property).
Type:	Enum (no_yes, byte)
Value Range:	0- NO **(default)**: the interrupt line is disabled 1- YES: the interrupt line is enabled
See Also:	Working With Interrupts 297

Details

Change of state of any enabled interrupt line leads to the on_int_ev 301 event generation. State change of disabled interrupt lines produces no effect.

.Intnum Property

Function:	Sets/returns the number of currently selected interrupt line.
Type:	Enum (pl_int_num, byte)
Value Range:	Platform-specific. See the list of pl_int_num constants in the platform specifications.
See Also:	Working With Interrupts 297

Details

Selected interrupt line can be enabled or disabled using the io.intenabled 299 property. State change on enabled interrupt lines causes on_io_int 301 event generation.

In order to work correctly as an interrupt on certain platforms, the line may need to be configured as an input -- see Controlling Output Buffers 296 for details.

.Invert Method

Function: Inverts the state of the I/O line specified by the num argument.

Syntax: **io.invert(**num **as pl_io_num)**

Returns: ---

See Also: Line/Port Manipulation Without Pre-selection 295, io.lineget 300, io.lineset 300

Part	Description
num	Platform-specific. See the list of pl_io_num constants in the platform specifications.

Details

No line pre-selection with the io.num 301 property is required and the value of the io.num will not be changed.

.Lineget Method

Function: Returns the state of the I/O line specified by the num argument.

Syntax: **io.lineget(**num **as pl_io_num) as low_high**

Returns: Current line state as LOW or HIGH (low_high enum values)

See Also: Line/Port Manipulation Without Pre-selection 295, io.lineset 300, io.invert 300

Part	Description
num	Platform-specific. See the list of pl_io_num constants in the platform specifications.

Details

No line pre-selection with the io.num 301 property is required and the value of the io.num will not be changed.

.Lineset Method

Function: Sets the I/O line specified by the num argument HIGH or LOW as specified by the state argument.

Syntax: **io.lineset(**num **as pl_io_num,** state **as low_high)**

Returns: ---

See Also: Line/Port Manipulation Without Pre-selection 295, io.lineget 300, io.invert 300

Part	Description
num	Platform-specific. See the list of pl_io_num constants in the platform specifications.
state	LOW or HIGH (low_high enum values).

Details

No line pre-selection with the io.num 301 property is required and the value of the io.num will not be changed.

.Num Property

Function:	Sets/returns the number of currently selected I/O line.
Type:	Enum (pl_io_num, byte)
Value Range:	Platform-specific. See the list of pl_io_num constants in the platform specifications.
See Also:	Line/Port Manipulation with Pre-selection 295

Details

Selects a particular I/O line to be manipulated through the io.state 303 and io.enabled 298 properties (the latter may or not be available on your platform). The list of available I/O lines is defined by the pl_io_num constant.

On_io_int Event

Function:	Generated when the change of state on one of the enabled interrupt lines is detected.
Declaration:	**on_io_int(**linestate **as byte)**
See Also:	Working With Interrupts 297

Part	Description
linestate	0-255. Each bit of this value corresponds to one interrupt line in the order that these lines are declared in the pl_int_num enum.

Details

Interrupt lines are enabled/disabled through the io.intenabled 299 property. Another on_io_int event is never generated until the previous one is processed.

.Portenabled Property (Selected Platforms Only)

Function: Sets/returns the state of the output buffers for the currently selected I/O port (selection is made through the io.portnum 302 property).

Type: Byte

Value Range: **Default**= 0 (all eight I/O lines of the port are configured as inputs)

See Also: Controlling Output Buffers 296, io.portstate 303

Details

Each I/O port groups eight I/O lines. Each bit of this property's byte value corresponds to one "member" I/O line. Setting the bit to 0 keeps the output buffer turned off, while setting the bit to 1 enables the output buffer.

Depending on the platform, explicit configuration 194 of the output buffers may or may not be required. For information on your device, see its platform documentation (for example, EM1000's is here 143). The property is either not available or has no effect on platforms that do not require buffer configuration.

.Portget Method

Function: Returns the state of the I/O port specified by the num argument.

Syntax: **io.portget(**num **as pl_io_port_num) as byte**

Returns: 0-255. Each bit of this value corresponds to one member I/O line of the 8-bit port.

See Also: Line/Port Manipulation Without Pre-selection 295, io.portset 303

Part	Description
num	Platform-specific. See the list of pl_io_port_num constants in the platform specifications.

Details

No line pre-selection with the io.portnum 302 property is required and the value of the io.portnum will not be changed.

.Portnum Property

Function: Sets/returns the number of currently selected I/O port.

Type: Enum (pl_io_port_num, byte)

Value Range: Platform-specific. See the list of pl_io_port_num constants in the platform specifications.

See Also: Line/Port Manipulation With Pre-selection 295

Details

Selects a particular I/O port to be manipulated through the io.portstate 303 and io.portenabled 302 properties (the latter may or not be available on your platform). Each port groups eight I/O lines. The list of available I/O ports is defined by the pl_io_port_num constant.

.Portset Method

Function: Sets the I/O port specified by the num argument to the state specified by the state argument.

Syntax: **io.portset(**num **as pl_io_port_num,** state **as byte)**

Returns: ---

See Also: Line/Port Manipulation Without Pre-selection 295, io.portget 302

Part	Description
num	Platform-specific. See the list of pl_io_port_num constants in the platform specifications.
state	0-255. Each bit of this value corresponds to one member I/O line of the 8-bit port.

Details

No line pre-selection with the io.portnum 302 property is required and the value of the io.portnum will not be changed.

.Portstate property

Function: Sets/returns the state of the currently selected I/O port (selection is made through the io.portnum 302 property).

Type: Byte

Value Range: 0-255. **Default** value is hardware-dependent.

See Also: Line/Port Manipulation With Pre-selection 295, io.portenabled 302

Details

Each I/O port groups eight I/O lines. Each bit of this property's byte value corresponds to one "member" I/O line.

.State Property

Function: Sets/returns the state of the currently selected I/O line (selection is made through the io.num 301 property).

Type:	Enum (low_high, byte)
Value Range:	0- LOW 1- HIGH (**Default** value is hardware-dependent)
See Also:	Line/Port Manipulation with Pre-selection 295, io.enabled 298

Details

Kp Object

This is the keypad (kp.) object, it allows you to work with a "matrix" keypad of up to 64 keys (8 scan lines by 8 return lines).

Features:

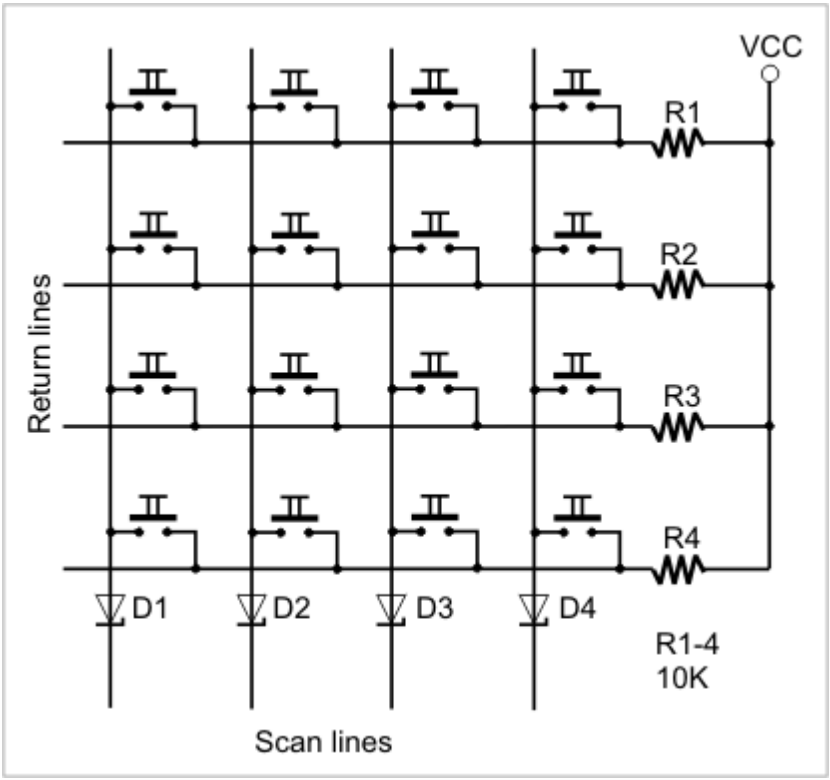
- Flexible keypad arrangement 304. Scan lines can double as LED control lines too.
- Five distinctive states 306 for each key allow you to create sophisticated keypad input. Individual programming of delay times 307 for each state transition.
- Flexible mapping 307 for scan and return lines -- use any I/O lines, in any order.
- Ability to auto-disable 307 the keypad when a certain key event/code combination is encountered.

Possible Keypad Configurations

The kp object works with a "matrix" keypad formed by scan and return lines. The keypad object supports up to 8 scan and 8 return lines, which means that you can build a keypad with up to 64 keys. A sample schematic diagram for a typical keypad is shown below.

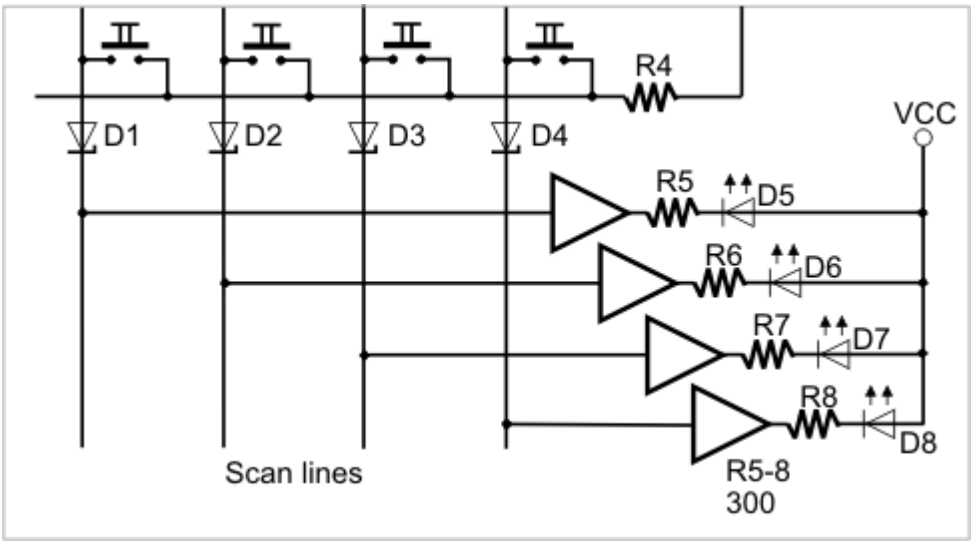
During the scanning process, the kp object "activates" one scan line after another. The line is activated by setting it LOW, while keeping all other scan lines HIGH. For each scan line, the kp object samples the state of return lines. If any return line is at LOW, this means the key located at the intersection of this return line and the currently active scan line is pressed.

A detailed discussion of the schematics falls outside the scope of this manual. We will only notice, in passing, that the diodes D1-D4 are necessary and should not be omitted. For best result, use Schottky diodes -- they have low drop voltage. Pull-up resistors R1-R4 prevent the return lines from floating and should be present as well.



On platforms with output buffer control 296, all intended scan lines should be configured as outputs, and all return lines -- as inputs (see io.num 301, io.enabled 298).

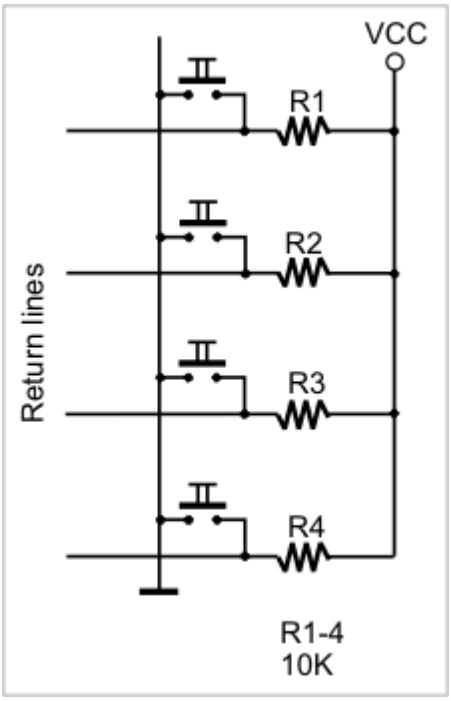
Scan lines can optionally perform a double duty and drive LEDs. One such LED can be connected to each scan line, preferably through a buffer, as shown on the drawing below. These LEDs can be used for any purpose you desire -- and this purpose can be completely unrelated to the keypad itself.



If the LED is connected as shown on the drawing, you need to set the corresponding I/O line LOW in order to turn this LED on. Each time the kp. object is to scan the keypad for pressed keys -- and this happens every 10ms -- it will first

set all scan lines to HIGH. This is necessary for correct keypad operation. Before doing so, however, the kp object will memorize the state of each scan line. This state will be restored after the scanning is complete. To your eye, this will look like that LED connected to the scan line was on all the time (of course, it it was on in the first place).

To build a functioning keypad you will need to have at least one return line. A sensible count of scan lines, however, starts from two! Having a single scan line is like having no scan lines whatsoever -- you might just as well ground this single scan line of the keypad, i.e. keep it active permanently. This arrangement is shown on the drawing below.



Preparing the Keypad for Operation 307 topic explains how to setup the kp. object properly for the keypad hardware you are using.

Key States and Transitions

Each key on your keypad can be in five different states, as defined by the pl_kp_event_codes constant. Here are those states:

0- PL_KP_EVENT_LONGRELEASED: The key has been released "for a while".

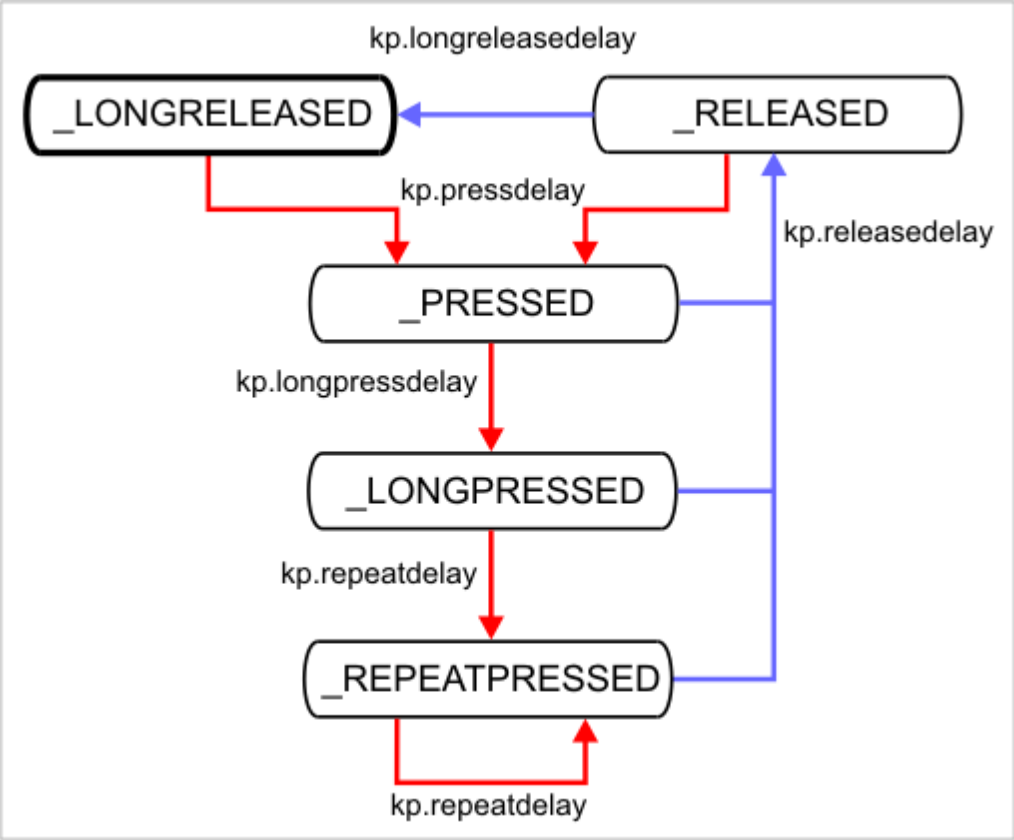
1- PL_KP_EVENT_RELEASED: The key has been released (just now).

2- PL_KP_EVENT_PRESSED: The key has been pressed (just now).

3- PL_KP_EVENT_LONGPRESSED: The key has been pressed "for a while".

4- PL_KP_EVENT_REPEATPRESSED: Auto-repeat for the key.

The diagram below shows all key states and possible state transitions.



Possible state transitions are indicated by arrows. Red arrows show transitions for when the key is pressed (or remains pressed), blue -- for when the key is released (or remains released). The time it takes for the key to transition from one state to another is quantified in 10ms intervals. This is the time period at which the kp. object will perform keypad scans. Each transition delay, expressed in 10ms intervals, is defined by a dedicated property, so you can decide for yourself what "just now" and "for a while" mean. Each time the key transitions to the next state, the on_kp 314 event is generated. See Servicing Keypad Events 310 for how to work with events. The keypad object has a buffer that can hold up to 8 keypad events.

When the keypad is enabled (kp.enabled 312 is set to 1- YES), the keypad buffer is cleared and each key's state is set to "longreleased". Press the key long enough, and the key will go into the "pressed" state. From there, the key can go to "released" and, later, "longreleased" if you let go of that key, or into "longpressed" and then "repeatpressed" if you keep the key pressed.

Five available key states, along with adjustable state transition times, allow you to create sophisticated keypad input. For example, it is possible to have a mixed alphanumerical input, like the one used on mobile phones for SMS entry. "Longpressed" events can be assigned to add a digit, for example, "1" if you press the "1ABC" key. "Pressed" events rotate between letters of the key ("A"->"B"->"C"->"A", etc.), unless another key is pressed or rotation times out on "longreleased" event. In both cases, input advances to the next character.

Preparing the Keypad for Operation

This topic explains what you need to do to property set up the kp. object. All preparations should be made with the keypad disabled (kp.enabled 312 = 0- NO), or this setup won't work.

Mapping scan and return lines

First, you need to define the list of scan and return lines. One great feature of the kp. object is that you can assign ("map") any I/O line of your device to be a scan or return line. I/O lines serving as scan or return lines don't even have to be "together" (have consecutive numbers). Scan lines are assigned through the kp.scanlinesmapping 317 property, return lines -- through the kp.returnlinesmapping 316 property. For example, here is how you select lines 24, 20, and 27 to serve as scan lines, and 28, 21, and 25 to serve as return lines:

```
kp.scanlinesmapping="24,20,27"
kp.returnlinesmapping="28,21,25"

io.num=PL_IO_NUM_24
io.enabled=YES
io.num=PL_IO_NUM_20_INT4
io.enabled=YES
io.num=PL_IO_NUM_27
io.enabled=YES
```

On platforms with output buffer control 296 each scan line must be configured as output, each return line -- as input (shown in the example above).

Notice how we did not have any order in specifying return and scan lines -- this simply does not matter. Select any lines, order then in any way you want. The only limitations are:

- You can't have more than 8 scan lines and 8 return lines;
- You must have at least 1 return line;
- Any given I/O line can only serve as a scan or return line, not both.

Line numbers are platform-dependent. They come from pl_io_num -- see its declaration in the "Platform-dependent Constants" section of your device's platform documentation (for example, EM1000's is here 143). The pl_io_num is a list of constants like "PL_IO_NUM_24". You cannot just drop "PL_IO_NUM_24" into kp.scanlinesmapping (or kp.returnlinesmapping). The correct way is to write "24" or use str(PL_IO_NUM_24). So, another way to do the above setup would look like this:

```
kp.scanlinesmapping=str(PL_IO_NUM_24)+","+str(PL_IO_NUM_20_INT4)+","+str
(PL_IO_NUM_27)
kp.returnlinesmapping=str(PL_IO_NUM_28)+","+str(PL_IO_NUM_21_INT5)+","+str
(PL_IO_NUM_25)
...
s=kp.returnlinesmapping 's will be equal to '28,21,25' (full constant names
like PL_IO_NUM_28 are not preserved)
```

Notice that no matter how you set kp.scanlinesmapping and kp.returnlinesmapping, reading them will always return a simple list of numbers (shown in the example above).

Defining state transition delays

Your second step is to set proper delay times for key state transitions 306. Five different properties are responsible for that: kp.pressdelay 315, kp.longpressdelay 313, kp.repeatdelay 315, kp.releasedelay 315, and kp.longreleasedelay 313. Each property sets the transition delay time in 10ms increments. Setting a property to 0 means that the corresponding transition will never happen. Note that the maximum value for each property is 254, not 255. All five properties already have sensible default values, so you only need to change them if you don't like what we have chosen for you:

```
kp.pressdelay=4 '40ms (4 successive keypad scans) to cinfirm that the key is
pressed
kp.longpressdelay=150 '1.5 seconds
kp.repeatdelay=0 'we do not want auto-repeat to work
kp.releasedelay=4 '40ms
kp.longreleasedelay=200 '2 seconds
```

Keypad auto-disable

Finally, you can select several event/code combinations that will automatically disable the keypad. This is done through the kp.autodisablecodes 312 property. Each time one of the pre-sent combinations of the key state and key code is detected, the kp.enabled property will be set to 0- NO, thus preventing further input until you re-enable the keypad.

This behavior can be very useful. Supposing, you have an application where you need to enter a certain code, then press <ENTER>. After you press <ENTER>, your application processes the input, which may take some time. What you often need is to prevent any further keypad input while the code is being processed. If you do not do this, the user might continue punching away and creating garbage input that your system does not need. This might even overwhelm the keypad buffer (see Servicing Keypad Events 310). The kp.autodisablecodes makes sure that the input stops at a certain event/code combination. Up to four such combinations can be defined.

Here is an example of how this property could be set:

```
kp.autodisablecodes=str(PL_KP_EVENT_PRESSED)+",49" 'assuming that <ENTER>
key has the code of 49
kp.autodisablecodes="2,49" 'this is because the PL_KP_EVENT_PRESSED constant
is equal to 2
```

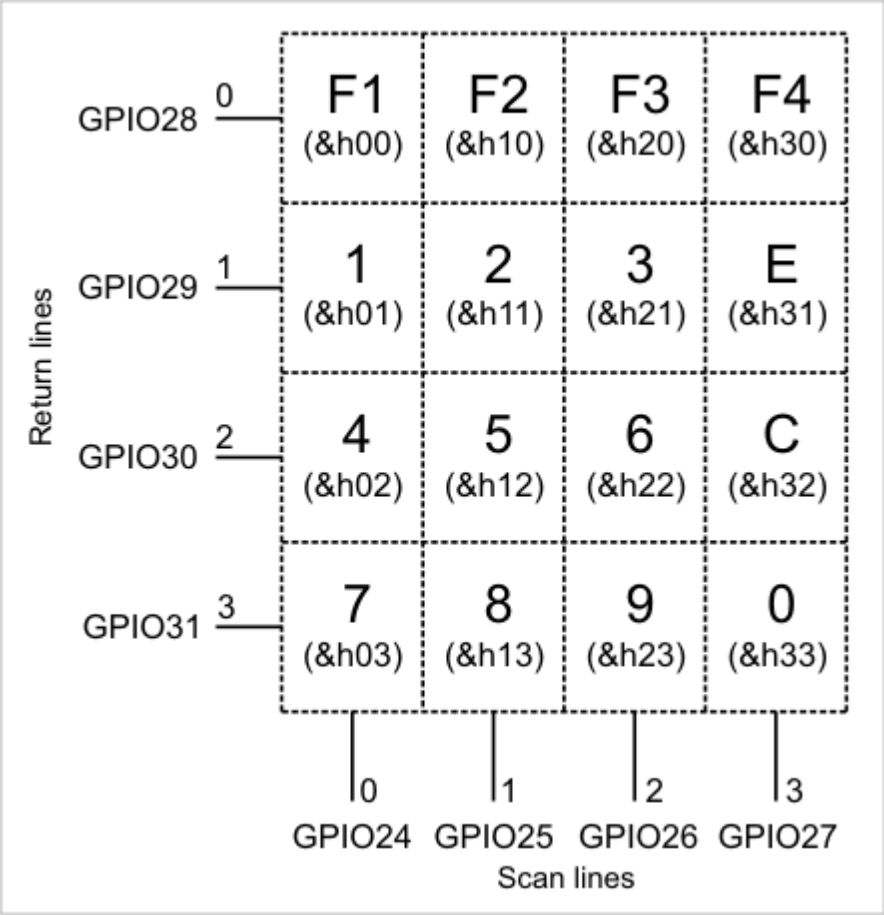
Once all the properties have been preset, enable the kp. object (kp.enabled= 1- YES), and start processing keypad events 310.

Servicing Keypad Events

Once you have correctly preset and enabled 307 the kp. object, you only need to process keypad events.

The on_kp 314 is the main event that is generated each time a key transitions to a new key state 306. The key_event argument will tell you what that new state is, while the key_code will tell you the code of the key. Event codes are defined in the on_kp_event_codes enum. The key code is composed of the scan line number (bits 7-4 of the key code), and the return line number (bits 3-0). Scan and return lines are numbered in the same order they are listed in the kp.scanlinesmapping 317 and kp.returnlinesmapping 316 properties.

For example, supposing you have a 4x4 keypad with <0> - <9> keys, also <F1> - <F4>, <E> (enter) and <C> (cancel). Notice how key codes in their hex representation reflect the number of the scan line (high digit) and return line (low digit):



Set up the kp. object to work correctly with your hardware:

```
kp.scanlinesmapping="24,25,26,27"
kp.returnlinesmapping="28,29,30,31"

io.num=PL_IO_NUM_24
io.enabled=YES
io.num=PL_IO_NUM_25
```

```
io.enabled=YES
io.num=PL_IO_NUM_26
io.enabled=YES
io.num=PL_IO_NUM_27
io.enabled=YES

'we are not going to change default delay values -- we like them as they are
(we came up with them, after all)

kp.autodisablecodes=str(PL_KP_EVENT_PRESSED)+",49" '<ENTER> will disable
further input

kp.enabled=YES
```

Here is an example of the event handler that adds your input to the inp_str string (global variable), clears the string when the <CANCEL> key is pressed, and launches the mysterious process_it procedure when the <ENTER> key is pressed:

```
Sub On_kp(key_event As pl_kp_event_codes, key_code As Byte)
   Dim x As Byte
   If key_event=PL_KP_EVENT_PRESSED Then
      Select Case key_code
         Case &h1:  x=1
         Case &h11: x=2
         Case &h21: x=3
         Case &h2:  x=4
         Case &h12: x=5
         Case &h22: x=6
         Case &h3:  x=7
         Case &h13: x=8
         Case &h23: x=9
         Case &h33: x=0
         Case &h32: '<CANCEL>
            inp_str=""
            Exit Sub
         Case &h31: '<ENTER> will disable the keypad...
            process_it()
            inp_str=""
            kp.enabled=YES '... so we re-enable it
            Exit Sub
      End Select
      inp_str=inp_str+chr(x) 'we will be here only when a numerical key is
pressed (see 'exit sub' under CANCEL and ENTER keys)
   End If
End Sub
```

Handling keypad buffer overflows

Another event -- on_kp_overflow 314 -- tells you that the input buffer of the keypad has been overwhelmed with frantic user input and the kp. object is not disabled. You respond appropriately:

```
Sub On_kp_overflow
```

```
      lcd.print("WHAT'S THE RUSH? SLOW DOWN!",0,0) 'tell the user
      inp_str="" 'clear the string
      kp.enabled=YES 're-enable the keypad
End Sub
```

Properties, Methods, Events

Properties, methods, and events of the kp object.

.Autodisablecodes Property

Function:	Defines which key event/code combinations disable the keypad.
Type:	String
Value Range:	Up to four comma-separated event/code pairs. **Default**= "".
See Also:	Preparing the Keypad for Operation 307, Key States and Transitions 306

Details

This property should contain a comma-separated list of event codes and key codes, for example: "2,15,0,20". In this example, two event/code pairs are set: "2,15" and "0,20". Event "2" is 2- PL_KP_EVENT_PRESSED, and event "0" is 0- PL_KP_EVENT_LONGRELEASED (see on_kp 314 event for a full list of codes). "15" and "20" are key codes. So, the keypad will be disabled (kp.enabled 312 set to 0- NO) when the key with code 15 is detected to be "pressed", or the key with code 20 is detected to be "longreleased".

The kp.autodisablecodes string should only contain a list of decimal numbers. That is, use "2" and not "2- PL_KP_EVENT_PRESSED". Only numerical characters are processed anyway -- writing "2- PL_KP_EVENT_PRESSED,15,0- PL_KP_EVENT_LONGRELEASED,20" will set this property to "2,15,0,20" anyway. You can, of course, write str(PL_KP_EVENT_PRESSED)+","+"15"+","+str (PL_KP_EVENT_LONGRELEASED)+","+"20" instead of "2,15,0,20".

This property can only be changed when the keypad is disabled (kp.enabled= 0- NO). Setting the property to "" means that no event and key combination will disable the keypad automatically.

.Enabled Property

Function:	Enables or disables the keypad.
Type:	Enum (no_yes, byte)
Value Range:	0- NO **(default)**: The keypad is disabled. 1- YES: The keypad is enabled.
See Also:	Preparing the Keypad for Operation 307

Details

The keypad matrix is being scanned and your application receives the on_kp 314 and on_kp_overflow 314 events only when the keypad is enabled (kp.enabled= 1- YES).

The following properties can be changed only when the keypad is disabled (kp.enabled= 0- NO): kp.autodisablecodes 312, kp.longreleasedelay 313, kp.longpressdelay 313, kp.pressdelay 315, kp.repeatdelay 315, kp.returnlinesmapping 316, kp.scanlinesmapping 317.

The keypad will be auto-disabled if an overflow is detected (see on_kp_overflow 314 event), or if one of the conditions for automatic keypad disablement is met (see kp.autodisablecodes 312).

Every time the keypad is enabled, each key's state is set to 0- PL_KP_EVENT_LONGRELEASED and the keypad event buffer is cleared.

.Longpressdelay Property

Function:	Defines (in 10ms increments) the amount of time a key should remain pressed for the key state to transition from "pressed" into "longpressed".
Type:	Byte
Value Range:	0-254. **Default**= 100 (1000ms).
See Also:	Key States and Transitions 306, Preparing the Keypad for Operation 307

Details

The on_kp 314 event with 3- PL_KP_EVENT_LONGPRESSED event code will be generated once the key transitions into the "longpressed" state.

This property can only be changed when the keypad is disabled (kp.enabled 312= 0- NO). Setting the property to 0 means that the key will never transition into the "longpressed" state.

.Longreleasedelay Property

Function:	Defines (in 10ms increments) the amount of time a key should remain released for the key state to transition from "released" into "longreleased".
Type:	Byte
Value Range:	0-254. **Default**= 100 (1000ms).
See Also:	Key States and Transitions 306, Preparing the Keypad for Operation 307

Details

The on_kp 314 event with 0- PL_KP_EVENT_LONGRELEASED event code will be generated once the key transitions into the "longreleased" state.

This property can only be changed when the keypad is disabled (kp.enabled 312 = 0-NO). Setting the property to 0 means that the key will never transition into the "longreleased" state.

On_kp Event

Function: Generated whenever a key transitions to another state.

Declaration: **on_kp(**key_event **as pl_kp_event_codes,** key_code **as byte)**

See Also: Servicing Keypad Events 310, on_kp_overflow 314

Part	Description
key_event	0- PL_KP_EVENT_LONGRELEASED: The key has transitioned into the "longreleased" state. 1- PL_KP_EVENT_RELEASED: The key has transitioned into the "released" state. 2- PL_KP_EVENT_PRESSED: The key has transitioned into the "pressed" state. 3- PL_KP_EVENT_LONGPRESSED: The key has transitioned into the "longpressed" state. 4- PL_KP_EVENT_REPEATPRESSED: Auto-repeat for the key.
key_code	Key code (byte). Bits 7-4 of this code represent scan line number, bits 3-0 -- return line number.

Details

Pressing and releasing any key on the keypad can generate up to five different events, as explained in Key States and Transitions 306. Scan lines and return lines are numbered as they are listed in kp.returnlinesmapping 316 and kp.scanlinesmapping 317.

This event can only be generated when the keypad is enabled (kp.enabled 312 = 1-YES).

On_kp_overflow Event

Function: Indicates that the keypad buffer has overflown and some key events may have been lost.

Declaration: **on_kp_overflow**

See Also: Servicing Keypad Events 310

Details

The keypad buffer stores up to 16 keypad events. Each such event causes the on_kp 314 generation. If your application is slow to process the keypad events, it is possible to overflow the keypad by pressing the keys in rapid succession. Once the

buffer overflows, the keypad is disabled automatically (kp.enabled 312 is set to 0- NO). You can re-enable the keypad by setting kp.enabled= 1- YES (this will clear the keypad buffer).

.Pressdelay Property

Function: Defines (in 10ms increments) the amount of time a key should remain pressed for the key state to transition from "released" into "pressed".

Type: Byte

Value Range: 0-254. **Default**= 3 (30ms).

See Also: Key States and Transitions 306, Preparing the Keypad for Operation 307

Details

The on_kp 314 event with 2- PL_KP_EVENT_PRESSED event code will be generated once the key transitions into the "pressed" state.

This property can only be changed when the keypad is disabled (kp.enabled 312 = 0- NO). Setting the property to 0 means that the key will never transition into the "pressed" state.

.Releasedelay Property

Function: Defines (in 10ms increments) the amount of time a key should remain released for the key state to transition from "pressed" or "longpressed" into "released".

Type: Byte

Value Range: 0-254. **Default**= 3 (30ms).

See Also: Key States and Transitions 306, Preparing the Keypad for Operation 307

Details

The on_kp 314 event with 1- PL_KP_EVENT_RELEASED event code will be generated once the key transitions into the "released" state.

This property can only be changed when the keypad is disabled (kp.enabled 312 = 0- NO). Setting the property to 0 means that the key will never transition into the "released" state.

.Repeatdelay Property

Function: Defines (in 10ms increments) the time period at which the on_kp 314 event with 4- PL_KP_EVENT_REPEATPRESSED event code will be generated once the key reaches the "longpressed" state and remains pressed.

Type:	Byte
Value Range:	0-254. **Default**= 50 (500ms).
See Also:	Key States and Transitions 306, Preparing the Keypad for Operation 307

Details

This property can only be changed when the keypad is disabled (kp.enabled 312= 0-NO). Setting the property to 0 means that the on_kp event with 4-PL_KP_EVENT_REPEATPRESSED event code will never be generated.

.Returnlinesmapping Property

Function:	Defines the list of up to 8 I/O lines that will serve as return lines of the keypad matrix.
Type:	String
Value Range:	Up to eight comma-separated I/O line numbers can be listed. **Default**= "".
See Also:	Possible Keypad Configurations 304, Preparing the Keypad for Operation 307, kp.scanlinesmapping 317

Details

This property should contain a comma-separated list of I/O lines numbers, for example: "24, 26, 27". Line numbers correspond to those of the pl_io_num enum. This enum is platform-specific. The declarations for the pl_io_num can be found in the "Platform-dependent Constants" section of your device's platform documentation (for example, EM1000's is here 143).

The kp.returnlinesmapping string should only contain a list of decimal numbers. That is, use "24" and not "24- PL_IO_NUM_24". Only numerical characters are processed anyway -- writing "24- PL_IO_NUM_24,25-PL_IO_NUM_25" will set this property to "24,25". You can, of course, write str(PL_IO_NUM_24)+","+str(PL_IO_NUM_25) as well.

The order in which you list the return lines *does matter* -- this is the order in which the lines will be numbered. All keys connected to the first return line will have their return field (bits 4-0) of the key code set to 0. For keys connected to the second line this field will contain 1, third line -- 2, and so on.

On platforms with output buffer control 296, all intended return lines should be configured as inputs by your application (see io.num 301, io.enabled 298).

This property can only be changed when the keypad is disabled (kp.enabled 312= 0-NO). Setting the property to "" means that the keypad will have no return lines. A keypad must have at least one return line to be able to work.

Return lines of the keypad should be separate from the scan lines (see kp.scanlinesmapping 317). The keypad will not work properly if you designate any I/O line as both a scan and return line.

.Scanlinesmapping Property

Function:	Defines the list of up to 8 I/O lines that will serve as scan lines of the keypad matrix.
Type:	String
Value Range:	Up to eight comma-separated I/O line numbers can be listed. **Default**= "".
See Also:	Possible Keypad Configurations 304, Preparing the Keypad for Operation 307, kp.scanlinesmapping 317

Details

This property should contain a comma-separated list of I/O lines numbers, for example: "28, 30, 31". Line numbers correspond to those of the pl_io_num enum. This enum is platform-specific. The declarations for the pl_io_num can be found in the "Platform-dependent Constants" section of your device's platform documentation (for example, EM1000's is here 143).

The kp.scanlinesmapping string should only contain a list of decimal numbers. That is, use "28" and not "28- PL_IO_NUM_28". Only numerical characters are processed anyway -- writing "28- PL_IO_NUM_28,30-PL_IO_NUM_30" will set this property to "28,30". You can, of course, write str(28- PL_IO_NUM_28)+","+str(30-PL_IO_NUM_30) as well.

The order in which you list the scan lines *does matter* -- this is the order in which the lines will be numbered. All keys connected to the first scan line will have their scan field (bits 7-4) of the key code set to 0. For keys connected to the second line this field will contain 1, third line -- 2, and so on.

On platforms with output buffer control 296, all intended scan lines should be configured as outputs by your application (see io.num 301, io.enabled 298).

This property can only be changed when the keypad is disabled (kp.enabled 312= 0-NO). Setting the property to "" means that the keypad will have no scan lines, which is also a *valid* keypad configuration.

Scan lines of the keypad should be separate from the return lines (see kp.returnlinesmapping 316). The keypad will not work properly if you designate any I/O line as both the scan line and return line.

LCD Object

This is the display (lcd.) object, it allows you to operate a display panel. With the lcd. object you can:

- Use any display from a growing list of supported controllers/panels 333.
- Inquire about the properties 318 of selected controller/panel.
- Draw pixels 320, lines 321, and rectangles 321; fill areas 321.
- Print text 322 -- non-aligned, aligned, rotated, etc.
- Display BMP images 329.

Overview 6.1

In this section:

Understanging Controller Properties

Display panels come in all types, shapes, and sizes. Tibbo supports a limited but growing number of controllers/panels 333. Each supported controller requires its own "mini-driver". Your application cannot specify the desired controller. Instead, controller type is selected through the Customize Platform dialog, accessible through the Project Settings dialog. Once this is done, a proper "mini-driver" is added to the executable binary during compilation.

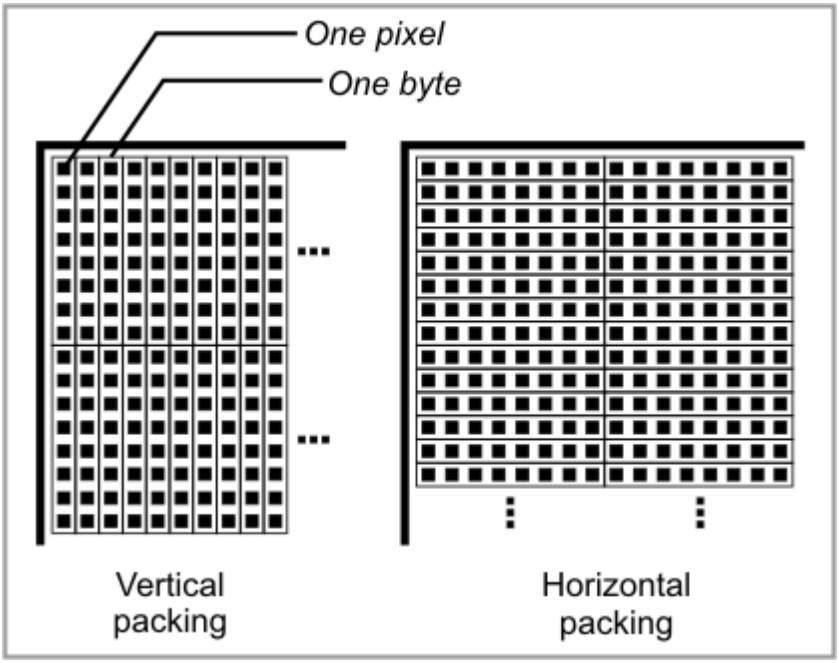
The lcd. object is ready to work with a wide range of displays, some of which are not actually of LCD type. So, the proper name for this object should be "display object". Nevertheless, we have decided to keep this name -- typing "lcd" is faster and the name has been in use for a while.

A number of R/O properties inform your application about the type and other vital parameters of the currently selected display:

Lcd.paneltype 349 indicates whether this panel is of monochrome/grayscale or color type.

Lcd.bitsperpixel 339 tells you how many bits in the display controller's memory are allocated for each display pixel. Of course, the more bits/pixel you get, the higher is your display quality.

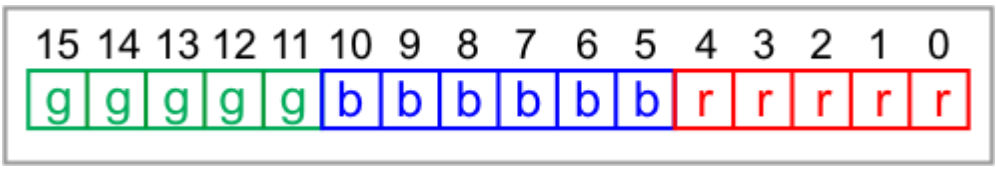
For displays with 1, 2, or 4 bits/pixel, a single byte of memory packs the data for 8, 4, or 2 pixels. Lcd.pixelpacking 350 will inform you how pixels are packed into memory bytes: vertically, or horizontally (see the drawing below). The reason you may want to know this is to achieve faster text output 322.



Color displays

For color displays (lcd.paneltyle= 1- PL_LCD_PANELTYPE_COLOR), three additional R/O properties -- lcd.redbits 352, lcd.greenbits 345, and lcd.bluebits 340 -- will indicate how many bits are available in each color channel, and also how three color fields are combined into a word describing the overall color of the pixel. You need to know this for setting pixels, as well as defining the foreground/background color used in drawing lines and rectangles, filling areas, and printing text (see Working With Pixels and Colors 320).

These three properties are of word type. Each 16-bit value packs two 8-bit parameters: number of bits per pixel for this color channel (high byte) and the bit position of the field in the color word (low byte). Supposing, lcd.redbits=&h0500, lcd.bluebits=&h0605, and lcd.greenbits=&h050B. You reconstruct the composition of the red, green, and blue bits in a word:



In this example, the red field is the first one on the right, followed by the blue field (this field starts from bit 5), then green field (starts from bit 11 or 7hB). You now also know that there are $5^2 = 32$ brightness levels for red and green, and 64 brightness levels for blue.

You can use this detailed information to select color values that will work correctly on all color displays, even those you haven't tested yet. Here is a useful example where you work out three constants -- color_red, color_green, and color_blue -- that will universally work for any color display.

```
Dim color_red, color_green, color_blue As word
...
```

```
lcd_red=val("&b"+strgen(lcd.redbits/&hFF,"1")+strgen(lcd.redbits And
&hFF,"0"))
lcd_green=val("&b"+strgen(lcd.greenbits/&hFF,"1")+strgen(lcd.greenbits And
&hFF,"0"))
lcd_blue=val("&b"+strgen(lcd.bluebits/&hFF,"1")+strgen(lcd.bluebits And
&hFF,"0"))
```

Now you can scientifically work out the constant for the white color:

```
...
lcd_white=lcd_red+lcd_green+lcd_blue
```

- In reality, you don't have to bother calculating color_white like this. Just select the highest possible value (&hFFFF) and this will be your white.

Preparing the Display for Operation

Several steps need to be taken before the display will become operational. Some of these steps are display-specific. Supported Controllers/Panels 333 section provides examples of startup code for each supported display.

Generally speaking, you need to take the following steps:

- Define which I/O lines and ports of your device control the display. This is done through the lcd.iomapping 347 property.
- Configure some I/O lines/ports as outputs, as required for controlling your particular display. This is only necessary on platforms with explicit output buffer control 194. For more details see your device's platform documentation (for example, EM1000's is here 143).
- Set the resolution of the display (lcd.width 357, lcd.height 346). These values depend on the panel, not the controller, so they cannot be detected automatically. Your application needs to set them "manually".
- Set lcd.rotated 353 = 1- YES if you wish the display image to be rotated 180 degrees (that is, the display of your device is installed up side down).
- Set lcd.inverted 347 = 1- YES if you need to invert the image on the display (may be required for certain panels).
- Enable the display by setting lcd.enabled 341 = 1- YES. This step will only work if your display is properly connected, correct display type is selected in your project, lcd.iomapping is set property, and necessary I/O lines are configured as outputs. The lcd.error 342 R/O property will indicate 1- YES if there was a problem enabling the display.
- Enable, the backlight, if needed -- this is not related to the controller/panel itself, but is still a necessary step. Light it up, don't linger in the dark!

Working With Pixels and Colors

The lcd.setpixel 354 method allows you to set the color of a single pixel. The method accepts a 16-bit color word, and the interpretation of this word is controller/panel-specific (see Understanding Controller Properties 318). In the following example, we determine the proper value for the green color and put a single green dot in the middle of the display:

```
Dim color_green As word
...
lcd_green=val("&b"+strgen(lcd.greenbits/&hFF,"1")+strgen(lcd.greenbits And
&hFF,"0"))
lcd.setpixel(lcd_green,lcd.width/2,lcd.height/2)
```

Lcd.setpixel is the only method that accepts the color directly. All other methods rely on two color properties: lcd.forecolor 344 and lcd.backcolor 339. Each property is a 16-bit value, just like the one used by the lcd.setpixel. The forecolor is the color of the "drawing pen", and the backcolor is the color of the background. In the following example, we set the lcd.forecolor to the brightest color available, and the lcd.backcolor to the darkest color available:

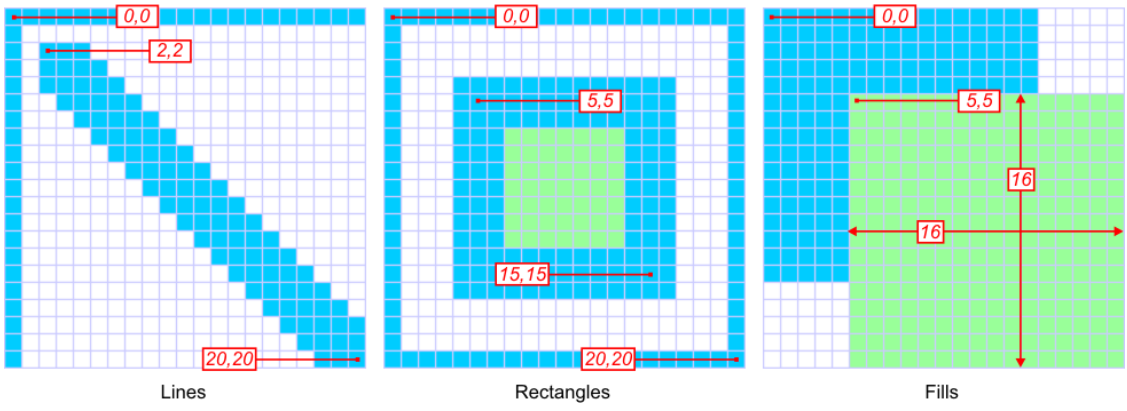
```
lcd.forecolor=&hFFFF
lcd.backcolor=0
```

The following methods are provided to output data onto the screen, and they all use the lcd.forecolor and, where necessary, the lcd.backcolor:

- Lcd.line 348, lcd.verline 357, lcd.horline 346 -- the line is drawn using the forecolor.
- Lcd.rectangle 352 -- the border is drawn using the forecolor, the internal area of the rectangle is not filled.
- Lcd.filledrectangle 343 -- the border is drawn using the forecolor, the internal area is filled with the backcolor.
- Lcd.fill 342 -- the entire area is filled with the forecolor.
- Lcd.print 351 and lcd.printaligned 351 -- print text, where each "on" dot of each character is displayed in forecolor, while each "off" dot is displayed in backcolor.

The following two sections -- Lines, Rectangles, and Fills 321, and Working With Text 322 -- discuss the above methods in more details.

Lines, Rectangles, and Fills



Lines

The lcd.line 348 method draws a line between any points. Lcd.verline 357 and lcd.horline 346 draw vertical and horizontal lines correspondingly. Use the last two methods whenever possible because they work faster than generic lcd.line. The line is drawn using the color set in the lcd.forecolor 344 property (see Working With Pixels/Lines 320), and the line width is defined by the lcd.linewidth 348 property. In the following example, we draw a picture as shown above, on the left:

```
lcd.forecolor=color_blue 'we assume we have already set color_blue
lcd.verline(0,0,20) 'vertical line, width=1 (default)
lcd.horline(0,20,0) 'horizontal line, width=1 (default)
lcd.linewidth=3 'change the width
lcd.line(2,2,20,20) 'line at 45 degrees, width=3
```

Defining lcd.linewidth>1 (3 for one of the lines in the above example) creates "fatter" lines. Notice how two points of the line are drawn and where each specified coordinate actually is.

Rectangles

Lcd.rectangle 352 draws an unfilled rectangle using lcd.forecolor as "pen" color, and pen width defined by the lcd.linewidth property. Lcd.filledrectangle 343 will additionally paint the internal area using lcd.backcolor. Example and its result:

```
lcd.forecolor=color_blue 'we assume we have already set color_blue
lcd.backcolor=color_green 'we assume we have already set color_green
lcd.rectangle(0,0,20,20) 'width=1 (default)
lcd.linewidth=3
lcd.filledrectangle(5,5,15,15) 'width=3, filled with background color
```

Fills

Lcd.fill 342 paints specified area with the lcd.forecolor:

```
lcd.forecolor=color_blue 'we assume we have already set color_blue
lcd.fill(0,0,16,16)
lcd.forecolor=color_green 'we assume we have already set color_green
lcd.fill(5,5,16,16)
```

Working With Text

Lcd.print 351 and lcd.printaligned 351 display text. The text is printed using the selected font. This means you need to have at least one font file in your project (see how to add 132 a file), and have this font selected before you can print anything.

Lcd.setfont 353 is used to select the font:

```
romfile.open("Tibbo-5x7(VP).bin")
lcd.setfont(romfile.offset)
```

Note that lcd.setfont will return 1- NG if you try to feed it a wrong file!

Once the font has been successfully selected, the lcd.fontheight 343 and lcd.fontpixelpacking 344 R/O properties will tell you the maximum height of characters in this font and how pixel data is packed within the font file. The meaning of the first one is obvious. The meaning of the second one will become apparent after (and if) you read Raster Font File (TRF) Format 325. If you don't want to read this, it's OK too, we can just go straight to the summary:

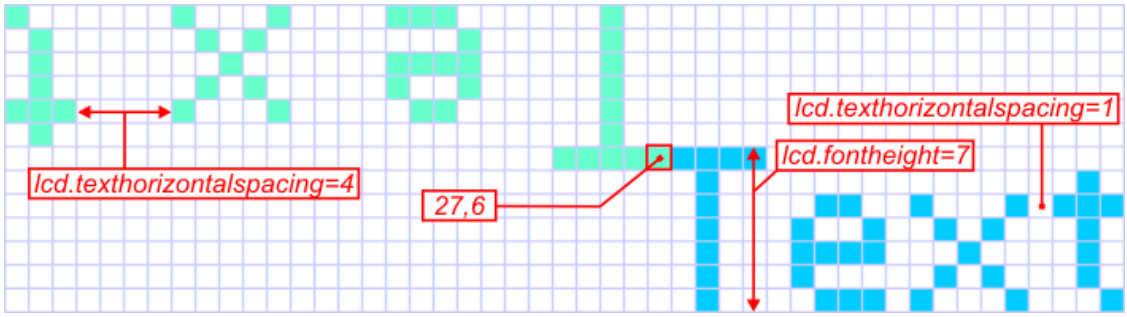
Fonts can be encoded horizontally or vertically, and the lcd.fontpixelpacking will tell you what type of font you have selected. You should only care about this for displays with lcd.bitsperpixel 339<8. if your display has lcd.bitsperpixel>8 then it doesn't matter what kind of font you use. If it is <8, then you are better off selecting a font for which lcd.fontpixelpacking is equal to lcd.pixelpacking 350 (depends on selected controller/panel). You achieve better performance when these two properties are "aligned". It will also work if you select a font which is "perpendicular" to your display, but text printing might slow down considerably.

We typically offer two versions for each font, for example, "Tibbo-5x7(V).bin" and "Tibbo-5x7(H)". V stands for "vertical" and "H" stands "horizontal".

Non-aligned text

Once the font has been selected, you can start printing. You do this with the lcd.print 351 method. This method always produces a single-line output. Two properties -- lcd.textorientation 356 and lcd.texthorizontalspacing 355 -- affect how your text is printed. The reference point of your text is at the top-left pixel of the output. X and Y arguments of lcd.print specify this corner, and the text is rotated "around" this pixel as well. Example:

```
lcd.print("Text",30,10)
lcd.textorientation=PL_LCD_TEXT_ORIENTATION_180
lcd.texthorizontalspacing=4
lcd.print("Text",30,10)
```



Lcd.print returns the total width of produced textual output in pixels. This can be very useful. Here is an example where you draw a frame around the text, and you want the frame size to be "just right":

```
x=lcd.print("Text",30,10)
lcd.rectangle(28,8,30+x+1,10+lcd.fontheight+1)
```

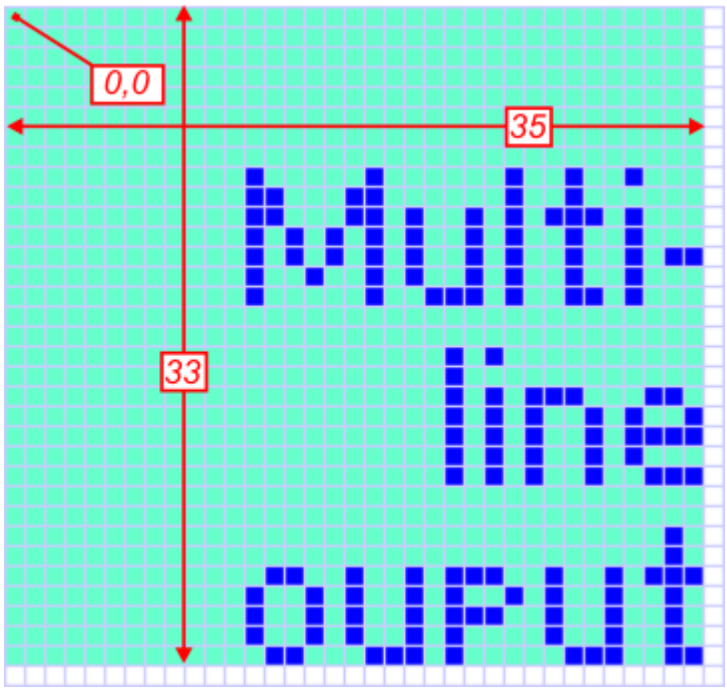
Sometimes, it is desirable to know the width of the text output before actual printing. Lcd.getprintwidth 345 will tell you how many pixels will be taken by your text horizontally (remember, text height is equal to lcd.fontheight 343). In the following example, we align the text output along the right side of the screen. This requires us to know how wide this output will be:

```
x=lcd.getprintwidth("Text")
x=lcd.print("Text",lcd.width-x,10)
```

Aligned text

The lcd.printaligned 351 method outputs your text within a specified rectangular area. Four properties -- -- lcd.textorientation 356, lcd.textalignment 355, lcd.texthorizontalspacing 355, and lcd.textverticalspacing 356 -- define how your text will be printed. To fit within the target area, the lcd. object will split the text into several lines as necessary. Only the text that can fit within the area will be displayed. You can add your own line breaks by using the ` character (ASCII code 96). Example:

```
lcd.textverticalspacing=2
lcd.textalignment=PL_LCD_TEXT_ALIGNMENT_BOTTOM_RIGHT
lcd.printaligned("Multi-line text",0,0,35,33)
```



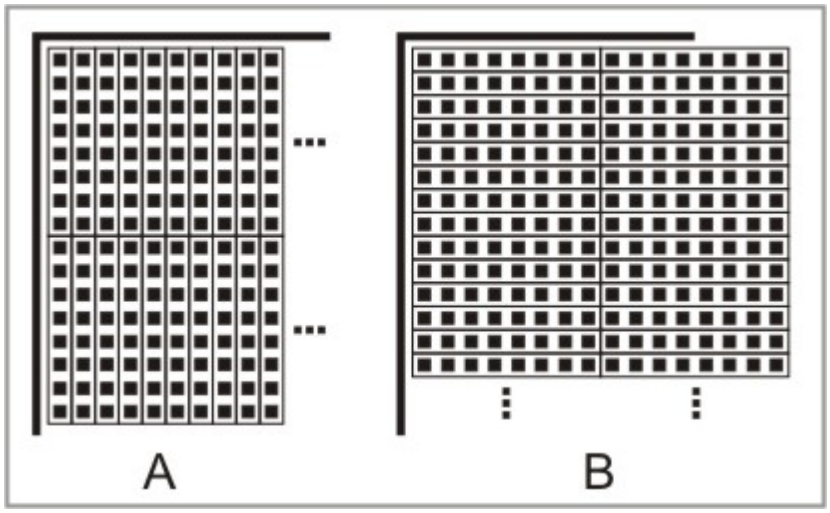
Note that lcd.printaligned returns the number of text lines that were produced.

Raster Font File Format

This topic details font file format used by the LCD object. The font file is a resource file that is added to your Tibbo Basic project. Like all other resource files "attached" to your project, font files are accessible through the romfile 370 object. Use and interpretation of font file data, however, is the responsibility of the lcd.object. The Romfile object merely stores these files.

Tibbo font files have "TRF" (Tibbo Raster Font) extension. TRF file format was designed with the following considerations in mind:

- Ability to handle large character sets (such as those required for Chinese language support). Hence, the use of 16-bit character codes.
- 16-bit character sets usually have large "gaps" (i.e. areas of unused codes). The TRF format offers an efficient way to define which characters are included into the font file and allows to conduct very efficient character search within the file.
- Support for proportional fonts. Hence, each character's width is individually defined.
- Support for fonts with anti-aliasing. Anti-aliasing is achieved by adjusting the "intensity" (brightness) of individual pixels. In an anti-aliased font, each pixel of character bitmap is represented by 2 or more bits of data. Fonts without anti-aliasing just need 1 data bit/pixel because each pixel can only be ON or OFF. At the time of writing, TiOS supported only fonts with 1 bit/pixel.
- Support for vertical and horizontal character bitmap encoding. Displays with lcd.bitsperpixel 339 = 1, 2, or 4 pack 8, 4, or 2 pixels into a single byte of display memory. Problem is, some displays combine the pixels vertically (see drawing A below), and some -- horizontally (drawing B). Text output on such displays is more efficient if character bitmaps of the TRF file use the same direction of packing.



TRF file format

The TRF file consists of four data areas:

Data area	Description
Header	Contains various information such as the total number of characters in this font, character height, etc. Also contains the number of code groups in the code groups table (see below).
Code groups table	Contains descriptors of "code groups". Each code group contains information about a range of codes that has no "gaps" (i.e. unused codes in the middle). The font file can have as many code groups as necessary.
Bitmap offset table	Contains addresses (offsets) of all character bitmaps in the TRF file. In combination with the code groups table provides a way to find the bitmap of any specific character.
Bitmaps	Contains all bitmaps of each character included into the font file. The width of each bitmap is defined individually and is stored together with the bitmap.

Header format

The header has a fixed length of 16 bytes and stores the following information:

Offset*	Bytes	Data	Comment
+0	2	**Num_of_chars**	Total number of characters in this font file
+2	1	**Pixels_per_byte**	0- eight pixels/byte (1 bit/pixel, no anti-aliasing); 1- four pixels/byte, 2- two pixels/byte, 3- one pixel/byte. Modes 1, 2, and 3 are currently not supported; these modes are for anti-aliasing and will be supported in the future.
+3	1	**Orientation**	0- pixels are grouped vertically; 1- pixels are grouped horizontally (irrelevant when **pixels_per_byte**=3)
+4	1	**Height**	Maximum character height in this font, in pixels.
+5	9	*<Reserved>*	Reserved for future use
+14	2	**Num_of_groups**	Number of entries in the code groups table

**With respect to the beginning of the file*

Code groups table

This table has variable number of entries. This number is stored in the **num_of_groups** field of the header. Each code group represents a range of codes that contains no gaps (no unused character codes in the middle). For example, supposing that we have a font that only contains characters '0'-'9' and 'A'-'Z'. This means that this font file will contain two groups of codes: 0030H through 0039H ('0'-'9') and 0041H through 005AH ('A'-'Z').

Each entry in the code groups table is 8 bytes long and has the following format:

Offset *	Bytes	Data	Comment
+0	2	**Start_code**	The first code in the group
+2	2	**Num_codes**	Number of individual character codes in this group
+4	4	**Bitmap_addr_offset**	Address (that falls within the bitmap offset table and is given with respect to the beginning of the file) at which the address of the bitmap of the first character in the code group is stored.

** With respect to the beginning of a particular table entry*

For the above example the code groups table will have two entries:

Start_code	Num_codes	Bitmap_offset
0030H	000AH	00000020H
0041H	001AH	00000048H

Here is how the above data was calculated. Start codes are obvious. Group one starts with code 0030H because this is the character code of '0'. The second group starts with the character code of 'A'. It is also easy to fill out the number of codes in each group: 10 (000AH) for '0'-'9' and 26 (001AH) for 'A'-'Z'.
Bitmap_addr_offset calculation is explained in the next section.

Bitmap offset table

This table has the same number of entries as the total number of characters included in the font file. Each entry consists of one field -- a 32-bit offset of a particular bitmap with respect to the beginning of the font file. Now you can see how we were able to calculate the data for the **bitmap_addr_offset** field of the code groups table. The header of the font file has a fixed length of 16 bytes. There are two code groups in our example, so the code groups table occupies 8x2=16 bytes. This means that the bitmap offset table starts from address 16+16=32 (0020H). Hence, the first entry in the code groups table points at address 0020H. The first code group contains 10 characters ('0'-'9'). These will "occupy" 10 entries in the bitmap offset table, which results in 10x4=40 bytes. Hence, the **bitmap_addr_offset** field for the second code group is set to 32+40=72 (0048H).

Bitmaps

Each bitmap starts with a single byte that encodes the width of the bitmap in pixels, followed by the necessary number of bytes representing this bitmap. Depending on the **pixels_per_byte** field of the header, each byte of data may

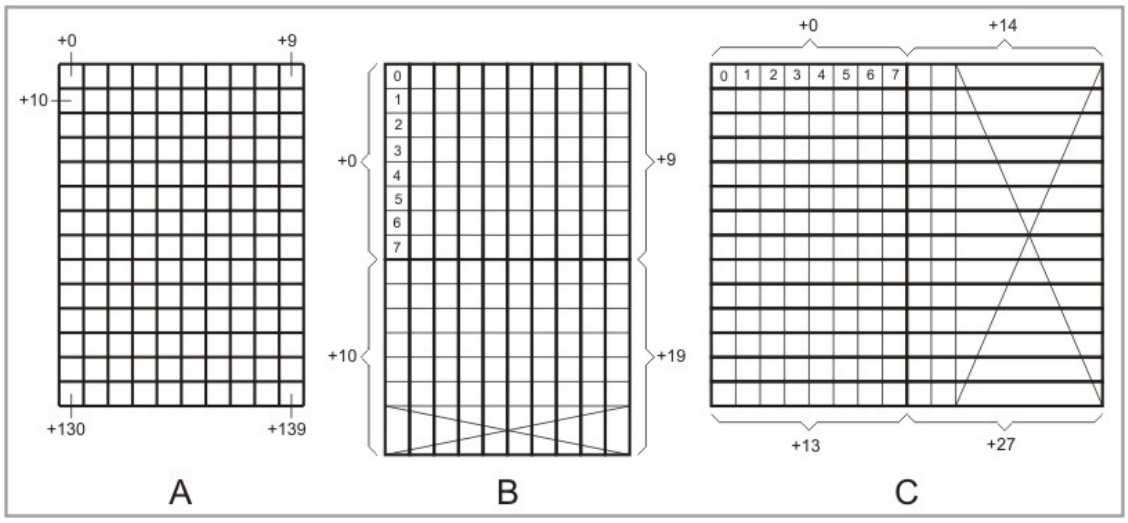
encode just one or several pixels. Additionally, when using more than 1 pixel-per-byte encoding, the **orientation** field of the header defines whether pixels are combined horizontally or vertically.

The drawings below illustrate how character bitmaps are stored in the font file. As an example, characters of 10x14 size (in pixels) are used. Drawing A is for one pixel/byte encoding, drawing B -- for 8 pixels/byte with vertical orientation, C -- for 8 pixels/byte with horizontal orientation. Notice that for cases B and C a portion of some bytes used to store the bitmaps is unused. Offsets of bytes relative to the beginning of the bitmap data are shown with a '+' sign.

Bitmap A takes 140 bytes. The first byte (+0) represents the pixel at the top left corner of the bitmap. Subsequent bytes represent all other pixels and the order is "left-to-right, top-to-bottom".

Bitmap B takes 20 bytes. The first byte encodes 8 vertically arranged pixels at the top left corner of the bitmap. Subsequent bytes represent all other pixel groups and the order is "left-to-right, top-to-bottom". There are 2 rows of bytes, and bits 6 and 7 of each byte in the second row are unused.

Bitmap C takes 28 bytes. The first byte encodes 8 horizontally arranged pixels at the top left corner of the bitmap. Subsequent bytes represent all other pixel groups and the order is "top-to-bottom, left-to-right". There are 2 columns of bytes, and bits 2-7 of each byte in the second column are unused.



Searching for a character bitmap

Here is how a target character bitmap is found within the font file. Again, we are using the example of the font file that contains characters '0'-'9' and 'A'-'Z'.

Supposing, we need to find the bitmap of character 'C' (code 0043H).

First, we need to see which code group code 004AH belongs to. We read the **num_of_groups** field of the header to find out how many code groups are contained in the font file. The field tells us that there are two groups.

Next, we start reading the code groups table (located at file offset +00000010H), entry by entry, in order to determine which code group the target character belong to. The first group starts from code 0030H and contains 10 character. Therefore, target character doesn't belong to it. The second group starts from code 0041H and contains 26 characters. The target code is 0043H. Therefore, target character belongs to this second group.

Next, we find the corresponding entry in the bitmap offset table. For this, we do a simple calculation:

bitmap_offset + (desired_code - **start_code**)*4: 00000048H + (0043H-0041H) *4= 00000050H.

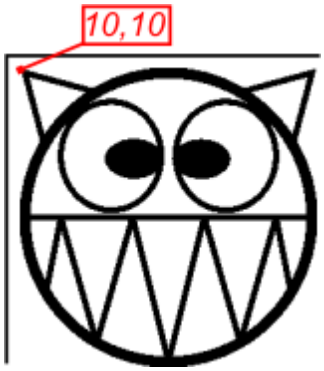
Next, we read a 32-bit value at file offset 00000050H. This will tell us the file offset at which the target bitmap is stored.

At this file offset, the first byte will be the width of the bitmap in pixels. Based on this width and also **height**, **pixels_per_byte** and **orientation** fields of the header we can calculate the number of bytes in the bitmap. For example, supposing that **height** = 14, **pixels_per_byte** = 0 (8 pixels/byte), and **orientation** = 0 (pixels are grouped vertically). Also, let's suppose that the width of the target character is 10 pixels. In this case the bitmap will occupy 20 bytes, as shown on the drawing B above. Two bits of each byte in the second byte row will be unused.

Displaying Images

Lcd.bmp 340 outputs a full image, or a portion of the image from a BMP file. Naturally, this file must be present in your project (see how to add 132 a file). Here is a simple example of how to display an image:

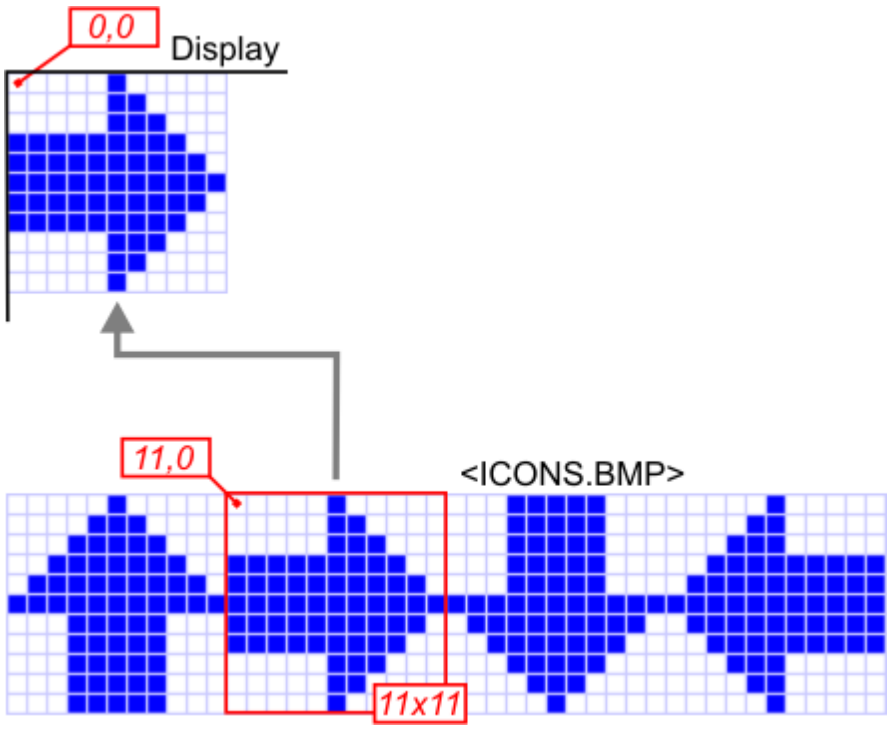
```
romfile.open("mad_happiness.bmp")
lcd.bmp(romfile.offset,0,0,0,0,65535,65535) 'see how we set maxwidth and
maxheight to 65535 (max value). This way we specify that the entire image
should be displayed (if only it can fit on the screen)
```



One powerful feature of the lcd.bmp method is that you can selectively display a portion of the image stored in a BMP file. This opens up two interesting possibilities. First, you can scroll around a large file image by specifying which part you want to see. Second, you can combine several images into one large file and display required portions when needed. For example, if your project requires several icons you can put them all into a single "strip". This improves performance because you

don't have to open a separate file to display each icon:

```
romfile.open("icon_strip.bmp")
lcd.bmp(romfile.offset,0,0,11,0,11,11) 'display the second icon (right
arrow)
```



Note that only 2-, 16-, and 256-color modes are currently supported and the lcd.bmp will return 1- NG if you try to display any other type of BMP file. Compressed BMP files will be rejected too.

The method takes into account the type of the currently selected controller/panel. The method will check the values of lcd.paneltype 349, lcd.bitsperpixel 339, lcd.redbits 352, lcd.greenbits 345, and lcd.bluebits 340 (explained in Understanding Controller Properties 318) and produce the best output possible for the selected display.

Improving Graphical Performance

Nobody likes sluggish products, and the way you work with your display can greatly influence *perceived* performance of your product. We say "perceived" because very often this has nothing to do with the actual speed at which your device completes required operations.

Group display-related operations together

The most important aspect of your application's performance (related to the display) is how fast the data on the display appears to have changed. Interestingly, what matters most is not the total time it takes to prepare and show the new data, but the "togetherness" at which all new displayed items pop up on the screen.

Let's illustrate this point. Here are two code examples that do the same and take roughly the same time to execute. We calculate and print two values. In the first

example, we calculate and print value 1, then calculate and print value 2:

```
Dim dw1,dw2 As dword
...
'Note so good! The user will see a noticeable delay between the first and
the second print.
lcd.print(str(dw1*100/200),0,0)
lcd.print(str(dw2*100/200),0,10)
```

In the second example we calculate values 1 and 2 first, then print them together:

```
Dim dw1,dw2 As dword
Dim s1,s2 As String
...
'Much better. The user will have a feeling that both values were calculated
and printed instantly!
s1=str(dw1*100/200)
s2=str(dw2*100/200)
lcd.print(s1,0,0)
lcd.print(s2,0,10)
```

Testing both examples shows that the perceived performance of the second code snippet is much better, while, in fact, the total working time of the processor was roughly the same. Why is there a difference? Because the output of the two values in the second example was spaced closer!

Bottom line: keep all display output as close together as possible. Pre-calculate everything first, then display all your items "at once".

Use display locking

No matter how hard you try to group all display-related output together, executing lcd. object's methods one after another may still take considerable time. Perceived performance can be improved on displays that allow you to "lock" display picture, change display data, then unlock the display again. With this approach, the user will see all changes appear instantly! Not all displays are suitable for this. Typically, this works well for TFT panels which continue to display the image for several seconds after the "refresh" was disabled. Other display types are not suitable for locking. We have provided locking-related info for each supported controller/panel 333.

The display is locked/unlocked using the Lcd.lock 348 and lcd.unlock 356 methods. You can place lcd.lock before the block of code that changes display data, and put lcd.unlock at the end of this code block:

```
...
s1=str(dw1*100/200)
s2=str(dw2*100/200)
lcd.lock 'lock the display
lcd.print(s1,0,0)
lcd.print(s2,0,10)
```

```
lcd.unlock 'unlock the display
...
```

Display-related code is often nested, with one procedure calling another, and so on. If you are using display locking, you should ideally place locks/unlocks in each related routine. A complication arises with regards to when to unlock the display. For example, supposing you have two subs: lcd_sub1 and lcd_sub2:

```
'some main routine that invokes lcd_sub1 and lcd_sub2
....
....
lcd_sub1
...
...
lcd_sub2
...
End Sub

'-----------------------------------------
Sub lcd_sub1 'calls sub2 too
lcd.lock
...
lcd_sub2
...
lcd.unlock
End Sub

'-----------------------------------------
Sub lcd_sub2 'called by sub1
lcd.lock
...
lcd.unlock
End Sub
```

Lcd_sub2 gets executed when invoked directly by the main code, and also when the lcd_sub1 is called. In the second case, the display should not be unlocked at the end of lcd_sub2 because the output is to be continued in lcd_sub1! And you know what? The display won't be unlocked because with the lcd object it is possible to nest locks/unlocks! In the following example, we do three consecutive locks and the display is locked right on the first one. We then do three consecutive unlocks, and the display is not unlocked until after the third one is executed:

```
...
lcd.lock 'lcd.lockcount=1, display is now locked
lcd.lock 'lcd.lockcount=2
lcd.lock 'lcd.lockcount=3
lcd.unlock 'lcd.lockcount=2
lcd.unlock 'lcd.lockcount=1
lcd.unlock 'lcd.lockcount=0, display is now unlocked
...
```

The lock/unlock mechanism maintains a counter, which can actually be read through the lcd.lockcount 349 R/O property. Each invocation of lcd.lock increments the counter by 1, each lcd.unlock decrements it by 1. The display is only unlocked when the counter is at 0, and locked when the counter is >0. This allows you to nest display-related procedures and safely have lock/unlock in each one of them!

Supported Controllers/Panels

The following controllers/panels are currently supported:

- Samsung S6B0108 (Winstar WG12864F) 333.
- Solomon SSD1329 (Ritdisplay RGS13128096) 335.
- Himax HX8309 (Ampire AM176220) 336.

None of the above suits your project's needs? Tibbo can be contracted to develop a driver for another display you want to use. We offer reasonable development prices and ZNR (zero non-refundable) payment scheme, in which your initial payment for the driver development is gradually returned to you as you purchase Tibbo hardware.

Samsung S6B0108 (Winstar WG12864F)

Controller: Samsung S6B0108.

Panel: Winstar WG12864F and similar panels.

Type: LCD with blue backlight/white dots, monochrome (1 bit/pixel), vertical pixel packing 318.

Locking: Supported, but the display image is not visible when the display is locked 330. This causes a noticeable "glitch" on the display when the lock/unlock is performed.

Test hardware: TEV-LB0 test board. This board is a part of the EM1000-TEV development system. See Programmable Hardware Manual for details.

I/O mapping for WG12864F

This panel requires 6 I/O lines and an 8-bit data bus. Control lines are **RST**, **EN**, **CS1**, **CS2**, **D/I**, and **R/W**. Each control line can be connected to any I/O pin of your device, and each such I/O pin must be configured as an output (if your device requires explicit I/O line buffer configuration). The data bus can be connected to any 8-bit port. DO NOT configure this port for output.

The value of the lcd.iomapping 347 property must be set correctly for the display to work (see Preparing the Display for Operation 320). For this particular display, the mapping string consists of 7 comma-separated decimal values:

1. Number of the I/O line connected to **RST**.
2. Number of the I/O line connected to **EN**.

3. Number of the I/O line connected to **CS1.**
4. Number of the I/O line connected to **CS2**.
5. Number of the I/O line connected to **D/I**.
6. Number of the I/O line connected to **R/W**.
7. Number of the I/O *port* connected to **DATA7-0**.

I/O line numbers are from the pl_io_num enum. Port number is from the pl_io_port_num enum. Line and port numbers are platform-specific. See the list of pl_io_num and pl_io_port_num constants in the platform specifications.

For the TEV-LB0 board, the lcd.iomapping property should be set to "44,46,40,41,43,42,4".

Code example -- TEV-LB0

This code will properly setup and enable this controller/panel (we assume that the testing is done using the TEV-LB0 board):

```
lcd.iomapping="44,46,40,41,43,42,4" 'RST,EN,CS1,CS2,D/I,R/W,DATA7-0

'configure control lines as outputs
io.num=PL_IO_NUM_46 'EN
io.enabled=YES
io.num=PL_IO_NUM_44 'RST
io.enabled=YES
io.num=PL_IO_NUM_40 'CS1
io.enabled=YES
io.num=PL_IO_NUM_41 'CS2
io.enabled=YES
io.num=PL_IO_NUM_42 'R/W
io.enabled=YES
io.num=PL_IO_NUM_43 'D/I
io.enabled=YES

'set resolution
lcd.width=128
lcd.height=64

'optionally set lcd.rotated here

lcd.enabled=YES 'done!

'turn on the backlight (strictly speaking, this is not related to the LCD
control, but we still show it here)
io.num=PL_IO_NUM_47
io.enabled=YES
io.state=HIGH

set_lcd_contrast(11) 'this is for external contrast control circuit
```

Contrast control

This is not a part of the panel itself, but we are still providing the code that will work on the TEV-LB0:

```
Sub set_lcd_contrast(level As Byte)
   'enable port, output data
   io.portnum=PL_IO_PORT_NUM_4
   io.portenabled=255
   io.portstate=level

   'generate strobe for the data register (on the LCD PCB)
   io.num=PL_IO_NUM_48
   io.enabled=YES
   io.state=HIGH
   io.state=LOW

   'disable port
   io.portenabled=0
End Sub
```

You can design your own contrast control circuit, of course.

Solomon SSD1329 (Ritdisplay RGS13128096)

Controller: Solomon SSD1329.

Panel: Ritdisplay RGS13128096 and similar panels.

Type: OLED, green, 16 levels (4 bits/pixel), horizontal pixel packing 318.

Locking: Supported, but the display image is not visible when the display is locked 330. This causes a noticeable "glitch" on the display when the lock/unlock is performed.

Test hardware: TEV-LB1 test board. This board is a part of the EM1000-TEV development system. See Programmable Hardware Manual for details.

I/O mapping for RGS13128096

This panel requires 5 I/O lines and an 8-bit data bus. Control lines are **RES**, **D/C**, **R/W**, **E**, and **CS**. Each control line can be connected to any I/O pin of your device, and each such I/O pin must be configured as an output (if your device requires explicit I/O line buffer configuration). The data bus can be connected to any 8-bit port. DO NOT configure this port for output.

The controller also has **BS1** and **BS2** interface type selection pin. For proper operation, tie these pins to Vcc.

The value of the lcd.iomapping 347 property must be set correctly for the display to work (see Preparing the Display for Operation 320). For this particular display, the mapping string consists of 6 comma-separated decimal values:

1. Number of the I/O line connected to **RES**.
2. Number of the I/O line connected to **D/C**.
3. Number of the I/O line connected to **R/W.**
4. Number of the I/O line connected to **E**.
5. Number of the I/O line connected to **CS**.
6. Number of the I/O *port* connected to **D7-0**.

I/O line numbers are from the pl_io_num enum. Port number is from the pl_io_port_num enum. Line and port numbers are platform-specific. See the list of pl_io_num and pl_io_port_num constants in your platform specifications.

For the TEV-LB0 board, lcd.iomapping should be set to "44,43,42,41,40,4".

Code example -- TEV-LB1

This code will properly setup and enable this controller/panel (we assume that the testing is done using the TEV-LB1 board):

```
lcd.iomapping="44,43,42,41,40,4" 'RST,D/C,R/W,E,CS,D7-0

'configure control lines as outputs
io.num=PL_IO_NUM_44 'RST
io.enabled=YES
io.num=PL_IO_NUM_43 'D/C
io.enabled=YES
io.num=PL_IO_NUM_42 'R/W
io.enabled=YES
io.num=PL_IO_NUM_41 'E
io.enabled=YES
io.num=PL_IO_NUM_40 'CS
io.enabled=YES

'set resolution
lcd.width=128
lcd.height=96

'optionally set lcd.rotated here

lcd.enabled=YES 'done!
```

Himax HX8309 (Ampire AM176220)

Controller: Himax HX8309.

Panel: Ampire AM176220 and similar panels.

Type: TFT, color, 16 bits/pixel.

Locking: Supported, display picture stays stable for at least 1 second after the display is locked 336. Unfortunately, there appears to be a hardware bug in the HX8309. This bug causes one of the horizontal lines of the panel (usually, the top or bottom line) to display random garbage while the display is locked. We have some ideas for workaround -- contact us if you encounter this problem.

Test hardware: TEV-LB2 test board. This board is a part of the EM1000-TEV development system. See Programmable Hardware Manual for details.

I/O mapping for AM176220

The panel requires 5 I/O lines and a 17-bit data bus, of which only bits DB7-0 are used (8-bit interface mode). Control lines are **RESET**, **RS**, **WR**, **RD**, and **CS**. Each control line can be connected to any I/O pin of your device, and each such I/O pin must be configured as an output (if your device requires explicit I/O line buffer configuration). The data bus can be connected to any 8-bit port. DO NOT configure this port for output.

The controller also has **IM0** and **IM3** interface type selection pin. For proper

operation, tie IM0 to Vcc, **IM3** - to the ground.

The value of the lcd.iomapping property must be set correctly for the display to work (see Preparing the Display for Operation). For this particular display, the mapping string consists of 6 comma-separated decimal values:

1. Number of the I/O line connected to **RESET**.
2. Number of the I/O line connected to **RS**.
3. Number of the I/O line connected to **WR.**
4. Number of the I/O line connected to **RD**.
5. Number of the I/O line connected to **CS**.
6. Number of the I/O *port* connected to **DB7-0**.

I/O line numbers are from the pl_io_num enum. Port number is from the pl_io_port_num enum. Line and port numbers are platform-specific. See the list of pl_io_num and pl_io_port_num constants in the platform specifications.

For the TEV-LB0 board, the lcd.iomapping should be set to "44,43,42,41,40,4".

Code example -- TEV-LB2

This code will properly setup and enable this controller/panel (we assume that the testing is done using the TEV-LB2 board):

```
lcd.iomapping="44,43,42,41,40,4" 'RESET,RS,WR,RD,CS,DB7-0

io.num=PL_IO_NUM_44 'RESET
io.enabled=YES
io.num=PL_IO_NUM_43 'RS
io.enabled=YES
io.num=PL_IO_NUM_42 'WR
io.enabled=YES
io.num=PL_IO_NUM_41 'RD
io.enabled=YES
io.num=PL_IO_NUM_40 'CS
io.enabled=YES

'set resolution
lcd.width=176
lcd.height=220

'optionally set lcd.rotated here

lcd.enabled=YES 'done!

'turn on the backlight (strictly speaking, this is not related to the LCD
control, but we still show it here)
io.num=PL_IO_NUM_47
io.enabled=YES
io.state=HIGH
```

Properties and Methods

The following classification groups properties and methods of the lcd. object by their logical function.

LCD panel characteristics:

- lcd.paneltype 349 [R/O Property]
- lcd.pixelpacking 350 [R/O Property]
- lcd.bitsperpixel 339 [R/O Property]
- lcd.redbits 352 [R/O Property]
- lcd.greenbits 345 [R/O Property]
- lcd.bluebits 340 [R/O Property]

Preparing to work:

- lcd.iomapping 347 [Property]
- lcd.width 357 [Property]
- lcd.height 346 [Property]
- lcd.inverted 347 [Property]
- lcd.rotated 353 [Property]
- lcd.enabled 341 [Property]

Graphical operations:

- lcd.setpixel 354 [Method]
- lcd.forecolor 344 [Property]
- lcd.backcolor 339 [Property]
- lcd.linewidth 348 [Property]
  - lcd.line 348 [Method]
  - lcd.horline 346 [Method]
  - lcd.verline 357 [Method]
  - lcd.rectangle 352 [Method]
  - lcd.filledrectangle 343 [Method]
  - lcd.fill 342 [Method]
- lcd.setfont 353 [Method]
- lcd.fontheight 343 [R/O Property]
- lcd.fontpixelpacking 344 [R/O Property]
- lcd.textalignment 355 [Property]
- lcd.textorientation 356 [Property]
- lcd.texthorizontalspacing 355 [Property]
- lcd.textverticalspacing 356 [Property]
  - lcd.print 351 [Method]
  - lcd.printaligned 351 [Method]
  - lcd.getprintwidth 345 [Method]
- lcd.bmp 340 [Method]

Miscellaneous:

- lcd.error 342 [R/O Property]
- lcd.lock 348 [Method]
- lcd.unlock 356 [Method]
- lcd.lockcount 349 [R/O Property]

.Backcolor Property

Function:	Specifies current background color.
Type:	Word
Value Range:	0-65535. **Default**= 0.
See Also:	Understanding Controller Properties 318, Working With Pixels and Colors 320, lcd.forecolor 344, lcd.linewidth 348

Details

The background color is used when drawing filled rectangles (lcd.filledrectangle 343) and performing fills (lcd.fill 342).

Property value interpretation depends on the currently selected controller/panel. Selection is made through the Customize Platform dialog, accessible through the Project Settings dialog.

The property is of word type, but only the lcd.bitsperpixel 339 lower bits of this value will be relevant. All higher bits will be ignored.

For monochrome and grayscale controllers/panels (lcd.paneltype 349 = 0- PL_LCD_PANELTYPE_GRAYSCALE), this value will relate to the brightness of the pixel. For color panels/controllers (lcd.paneltype= 1- PL_LCD_PANELTYPE_COLOR) the value is composed of three fields -- one each for the red, green, and blue "channels". Check lcd.redbits 352, lcd.greenbits 345, and lcd.bluebits 340 properties to see how the fields are combined into the color word.

.Bitsperpixel R/O Property

Function:	Returns the number of bits available for each pixel of the currently selected controller/panel.
Type:	Byte
Value Range:	Value depends on the currently selected controller/panel
See Also:	Understanding Controller Properties 318, Working With Pixels and Colors 320

Details

Controller/panel selection is made through the Customize Platform dialog, accessible through the Project Settings dialog.

For monochrome controllers/panels (see lcd.paneltype 349) the lcd.bitsperpixel will return 1, that is, the pixel can only be on or off. For grayscale panels, this value will

be >1, which indicates that each pixel can be set to a number of brightness levels. For example, if the lcd.bitsperpixel=4, then each pixel's brightness can be adjusted in 16 steps.

For color panels, this property reflects the combined number of red, green, and blue bits available for each pixel (see lcd.redbits 352, lcd.greenbits 345, and lcd.bluebits 340).

The number of bits per pixel affects how lcd.forecolor 344, lcd.backcolor 339, and lcd.setpixel 354 are interpreted. Also, the output produced by lcd.bmp 340 depends on this property.

.Bluebits R/O Property

Function: A 16-bit value packing two 8-bit parameters: number of "blue" bits per pixel (high byte) and the position of the least significant blue bit within the color word (low byte).

Type: Word

Value Range: Value depends on the currently selected controller/panel.

See Also: Understanding Controller Properties 318, Working With Pixels and Colors 320

Details

The value of this property depends on the currently selected controller/panel. Selection is made through the Customize Platform dialog, accessible through the Project Settings dialog. This property is only relevant for color panels (lcd.paneltype 349 = 1- PL_LCD_PANELTYPE_COLOR).

By taking the value of the high byte you can determine the number of the steps in which the brightness of the blue "channel" can be adjusted. For example, if the high byte is equal to 6, then there are 64 levels for blue.

This property also tells you the bit position and length of the blue field in color values used by lcd.forecolor 344, lcd.backcolor 339, and lcd.setpixel 354. If, for example, the lcd.redbits 352 =&h0500, lcd.bluebits=&h0605, and lcd.greenbits 345 =&h050B, then you can reconstruct the composition of the red, green, and blue bits in a word: bit 15 -> gggggbbbbbbrrrrr <- bit 0. In this example, the blue field is in the middle and occupies 6 bits (10-5).

.Bmp Method

Function: Displays a portion of or full image stored in a BMP file.

Syntax: **lcd.bmp(**offset **as dword,** x **as word,** y **as word,** x_offset **as word,** y_offset **as word,** maxwidth **as word,** maxheight **as word) as ok_ng**

Returns: 0- OK: Processed successfully.

1- NG: Unsupported or invalid file format.

See Also: Displaying Images 329

Part	Description
offset	Offset within the compiled binary of your application at which the BMP file is stored.
x	X coordinate of the top-left point of the image position on the screen. Value range is 0 to lcd.width 357 - 1.
y	Y coordinate of the top-left point of the image position on the screen. Value range is 0 to lcd.height 346 - 1.
x_offset	Horizontal offset within the BMP file marking the top-left corner of the image portion to be displayed.
y_offset	Vertical offset within the BMP file marking the top-left corner of the image portion to be displayed.
maxwidth	Maximum width of the image portion to be displayed. Actual width of the output will be defined by the total width of the image and specified x_offset.
maxheight	Maximum height of the image portion to be displayed. Actual height of the output will be defined by the total height of the image and specified y_offset.

Details

To obtain the offset, open the BMP file with romfile.open 374, then read the offset of this file from the romfile.offset 373 R/O property. Naturally, the BMP file must be present in your project for this to work (see how to add 132 a file).

Note that only 2-, 16-, and 256-color modes are currently supported and the lcd.bmp will return 1- NG if you try to display any other type of BMP file. Compressed BMP files will be rejected too.

The method takes into account the type of the currently selected controller/panel (selection is made through the Customize Platform dialog, accessible through the Project Settings dialog). It will check the values of lcd.paneltype 349, lcd.bitsperpixel 339, lcd.redbits 352, lcd.greenbits 345, and lcd.bluebits 340 and produce the best output possible for the selected display.

X_offset, y_offset, maxwidth, and maxheight arguments allow you to display only a portion of the BMP image. This way it is possible to scroll around a large image that does not fit on the screen. Another use is to combine several separate images into a single file and display selected portions. This improves efficiency because romfile.open 374 needs only be done once.

.Enabled Property

Function: Specifies whether the display panel is enabled.

Type: Enum (no_yes, byte)

Value Range: 0- NO: Disabled **(default)**.
1- YES: Enabled.

See Also: Preparing the Display for Operation 320

Details

Several properties -- lcd.iomapping 347, lcd.width 357, lcd.height 346, lcd.inverted 347, lcd.rotated 353 -- can only be changed when the display panel is disabled.

When you set this property to 1- YES, the controller of the panel is initialized and enabled. This will only work if your display is properly connected, correct display type is selected in your project, lcd.iomapping is set property, and necessary I/O lines are configured as outputs. The lcd.error 342 R/O property will indicate 1- YES if there was a problem enabling the display.

Setting the property to 0- NO disables the controller/panel.

.Error R/O Property

Function:	Indicates whether controller/panel I/O error has been detected.
Type:	Enum (no_yes, byte)
Value Range:	0- NO: No error detected. 1- YES: I/O error.
See Also:	Preparing the Display for Operation 320

Details

The lcd. object will detect a malfunction (or absence) of the controller/panel that is expected to be connected. If the display is not properly connected, or the lcd. object is not set up property to work with this display, the lcd.error will be set to 1- YES on attempt to enable the display (set lcd.enabled 341 = 1- YES).

.Fill Method

Function:	Paints the area with the "pen" color (lcd.forecolor 344).
Syntax:	**lcd.fill(**x **as word,**y **as word,** width **as word,** height **as word)**
Returns:	---
See Also:	Lines, Rectangles, and Fills 321, Working With Pixels and Colors 320, lcd.line 348, lcd.verline 357, lcd.horline 346, lcd.rectangle 352, lcd.filledrectangle 343

Part	Description
x	X coordinate of the top-left point of the area to be painted. Value range is 0 to lcd.width 357 - 1.
y	Y coordinate of the top-left point of the area to be painted. Value range is 0 to lcd.height 346 - 1.

width	Width of the paint area in pixels.
height	Height of the paint area in pixels.

Details

The display panel must be enabled (lcd.enabled 341 = 1- YES) for this method to work.

.Filledrectangle Method

Function: Draws a filled rectangle.

Syntax: **lcd.filledrectangle(**x1 **as word,**y1 **as word,** x2 **as word,** y2 **as word)**

Returns: ---

See Also: Lines, Rectangles, and Fills 321, Working With Pixels and Colors 320, lcd.line 348, lcd.verline 357, lcd.horline 346, lcd.rectangle 352, lcd.fill 342

Part	Description
x1	X coordinate of the first point. Value range is 0 to lcd.width 357 - 1.
y1	Y coordinate of the first point. Value range is 0 to lcd.height 346 - 1.
x2	X coordinate of the second point. Value range is 0 to lcd.width 357 - 1.
y2	Y coordinate of the second point. Value range is 0 to lcd.height 346 - 1.

Details

The border is drawn with the specified line width (lcd.linewidth 348) and "pen" color (lcd.forecolor 344). The rectangle is then filled using the background color (lcd.backcolor 339). Setting the lcd.linewidth to 0 will create a rectangle with no border -- basically, a filled area.

The display panel must be enabled (lcd.enabled 341 = 1- YES) for this method to work.

.Fontheight R/O Property

Function: Returns the maximum height, in pixels, of characters in the currently selected font.

Type: Byte

Value Range: 0-255. **Default**= 0.

See Also: Working With Text 320, lcd.fontpixelpacking 344

Details

This property will only return meaningful data after you select a font using the lcd.setfont 353 method.

.Fontpixelpacking R/O Property

Function:	Indicates how pixels are packed into bytes in a currently selected font.
Type:	Enum (ver_hor, byte)
Value Range:	0- PL_VERTICAL: Several vertically adjacent pixels are packed into each byte of character bitmaps. 1- PL_HORIZONTAL: Several horizontally adjacent pixels are packed into each byte of character bitmaps.
See Also:	Working With Text 322, lcd.fontheight 343

Details

Display controllers/panels can have vertical or horizontal pixel packing (see lcd.pixelpacking 350). The speed at which you can output the text onto the screen is improved when the lcd.pixelpacking and lcd.fontpixelpacking have the same value, i.e. controller memory pixels and font encoding are "aligned". Our font files are typically available both in vertical and horizontal pixel packing. Pick the right file for your controller/panel.

This property will only return meaningful data after you select a font using the lcd.setfont 353 method.

.Forecolor Property

Function:	Specifies current "pen" (drawing) color.
Type:	Word
Value Range:	0-65535. **Default**= 65535 (&hFFFF).
See Also:	Understanding Controller Properties 318, Working With Pixels and Colors 320, lcd.backcolor 339, lcd.linewidth 348

Details

Pen color is used when drawing lines (lcd.line 348, lcd.verline 357, lcd.horline 346) and rectangles (lcd.rectangle 352, lcd.filledrectangle 343), as well as displaying text (lcd.print 351, lcd.printaligned 351).

Property value interpretation depends on the currently selected controller/panel. Selection is made through the Customize Platform dialog, accessible through the Project Settings dialog.

The property is of word type, but only lcd.bitsperpixel 339 lower bits of this value will be relevant. All higher bits will be ignored.

For monochrome and grayscale controllers/panels (lcd.paneltype 349 = 0-

PL_LCD_PANELTYPE_GRAYSCALE), this value will relate to the brightness of the pixel. For color panels/controllers (lcd.paneltype= 1- PL_LCD_PANELTYPE_COLOR) the value is composed of three fields -- one for the red, green, and blue "channels". Check lcd.redbits 352, lcd.greenbits 345, and lcd.bluebits 340 properties to see how the fields are combined into the color word.

.Getprintwidth Method

Function: Returns the width, in pixels, of the text output that will be produced if the same line is actually printed with the lcd.print 351 method.

Syntax: **lcd.getprintwidth(byref** str **as string) as word**

Returns: Total width of text output in pixels.

See Also: Working With Text 322

Part	Description
str	Text to estimate the output width for.

Details

This method does not produce any output on the display, it merely estimates the width of the text *if* it was to be printed. Lcd.print also returns the width of the text in pixels, but this data comes *after* the printing. Sometimes it is desirable to know the output width for the line of text before printing it, and this method allows you to do so.

The width calculation will be affected by the value of the lcd.texthorizontalspacing 355 property.

.Greenbits R/O Property

Function: A 16-bit value packing two 8-bit parameters: number of "green" bits per pixel (high byte) and the position of the least significant green bit within the color word (low byte).

Type: Word

Value Range: Value depends on the currently selected controller/panel.

See Also: Understanding Controller Properties 318, Working With Pixels and Colors 320

Details

The value of this property depends on the currently selected controller/panel. Selection is made through the Customize Platform dialog, accessible through the Project Settings dialog. This property is only relevant for color panels (lcd.paneltype 349= 1- PL_LCD_PANELTYPE_COLOR).

By taking the value of the high byte you can determine the number of the steps in which the brightness of the green "channel" can be adjusted. For example, if the

high byte is equal to 5, then there are 32 levels for green.

This property also tells you the bit position and length of the green field in values used by lcd.forecolor 344, lcd.backcolor 339, and lcd.setpixel 354. If, for example, the lcd.redbits 352=&h0500, lcd.bluebits 340=&h0605, and lcd.greenbits=&h050B, then you can reconstruct the composition of the red, green, and blue bits in a word: bit 15 -> gggggbbbbbbrrrrr <- bit 0. In this example, the green field is the last field and occupies 5 bits (15-11).

.Height Property

Function: Sets the vertical resolution of the display panel in pixels.

Type: Word

Value Range: Appropriate value depends on the panel. **Default**= 0.

See Also: Preparing the Display for Operation 320, lcd.width 357

Details

Set this property according to the characteristics of your display panel.

This value is not set automatically when you select a certain controller because the capability of the controller may exceed the actual resolution of the panel, i.e. only "part" of the controller may be utilized.

This property can only be changed when the lcd is disabled (lcd.enabled 341= 0- NO) .

.Horline Method

Function: Draws a horizontal line.

Syntax: **lcd.horline(**x1 **as word,**x2 **as word,**y **as word)**

Returns: ---

See Also: Lines, Rectangles, and Fills 321, lcd.rectangle 352, lcd.filledrectangle 343, lcd.fill 342

Part	Description
x1	X coordinate of the first point. Value range is 0 to lcd.width 357-1.
x2	X coordinate of the second point. Value range is 0 to lcd.width 357-1.
y	Y coordinates of the first and second points. Value range is 0 to lcd.height 346-1.

Details

The line is drawn with the specified line width (lcd.linewidth 348) and "pen" color (lcd.forecolor 344). Drawing horizontal or vertical (lcd.verline 357) lines is more efficient than drawing generic lines (lcd.line 348), and should be used whenever possible.

The display panel must be enabled (lcd.enabled 341 = 1- YES) for this method to work.

.Inverted Property

Function:	Specifies whether the image on the display panel has to be inverted.
Type:	Enum (no_yes, byte)
Value Range:	0- NO: Not inverted, higher memory value of the pixel corresponds to higher brightness **(default)**. 1- YES: Inverted, higher memory value of the pixel corresponds to lower brightness.
See Also:	Preparing the Display for Operation 320

Details

Set this property according to the characteristics of your display panel.

This value is not set automatically when you select a certain controller because the display characteristics cannot be detected automatically, as they depend on the panel and its backlight arrangement.

This property can only be changed when the display is disabled (lcd.enabled 341 = 0- NO).

.Iomapping Property

Function:	Defines the list of I/O lines to interface with the currently selected controller/panel.
Type:	String
Value Range:	Value depends on the currently selected controller/panel. **Default**= "".
See Also:	Preparing the Display for Operation 320

Details

Controller/panel selection is made through the Customize Platform dialog, accessible through the Project Settings dialog.

Different controllers/panels require a different set of interface lines, and even the number of lines depends on the hardware. This property should contain a comma-separated list of decimal numbers that indicate which I/O lines and ports are used to connect the controller/panel to your device. The meaning of each number in the list is controller- and panel-specific. See the Supported Controllers 333 section for details.

This property can only be changed when the display is disabled (lcd.enabled 341 = 0- NO).

.Line Method

Function: Draws a line.

Syntax: **lcd.line(**x1 **as word,**y1 **as word,** x2 **as word,** y2 **as word)**

Returns: ---

See Also: Lines, Rectangles, and Fills 321, lcd.rectangle 352, lcd.filledrectangle 343, lcd.fill 342

Part	Description
x1	X coordinate of the first point. Value range is 0 to lcd.width 357 - 1.
y1	Y coordinate of the first point. Value range is 0 to lcd.height 346 - 1.
x2	X coordinate of the second point. Value range is 0 to lcd.width 357 - 1.
y2	Y coordinate of the second point. Value range is 0 to lcd.height 346 - 1.

Details

The line is drawn with the specified line width (lcd.linewidth 348) and "pen" color (lcd.forecolor 344). Drawing horizontal (lcd.horline 346) or vertical (lcd.verline 357) lines is more efficient than drawing generic lines, and should be used whenever possible.

The display panel must be enabled (lcd.enabled 341 = 1- YES) for this method to work.

.Linewidth Property

Function: Specifies current "pen" width in pixels.

Type: Byte

Value Range: 1-255. **Default**= 1 (1 pixel).

See Also: Lines, Rectangles, and Fills 321, lcd.forecolor 344, lcd.backcolor 339

Details

Pen width is used when drawing lines (lcd.line 348, lcd.verline 357, lcd.horline 346) and rectangles (lcd.rectangle 352, lcd.filledrectangle 343).

.Lock Method

Function: Freezes display output (on controllers/panels that support this feature).

Syntax: **lcd.lock**

Returns: ---

See Also: ---

Details

When the display is locked, you can make changes to the display data without showing these changes on the screen. You can then unlock the display (lcd.unlock) and show all the changes made at once. This usually greatly improves the display agility *perception* by the user (see Improving Graphical Performance 330).

Locking the display also prolongs the life of some displays, notably the displays of the OLED type.

When you execute this method for the first time, the display gets locked and the lcd.lockcount 349 R/O property changes from 0 to 1. You can invoke lcd.lock again and again, and the lcd.lockcount will increase with each call to the lcd.lock. This allows you to nest locks/unlocks (again, see Improving Graphical Performance 330). Of course, the display is not locked "any harder" at lcd.lockcount=2 compared to lcd.lockcount=1. The display is simply locked for all lcd.lockcount values other than 0.

Not all controllers/panels support this feature. See the Supported Controllers/Panels 333 section for details on the display you are using. If your display does not support locking, executing lcd.lock will have no effect and lcd.lockcount 349 will always stay at 0.

.Lockcount R/O Property

Function:	Indicates the current nesting level of the display lock.
Type:	Byte
Value Range:	0-255. **Default**= 0 (display unlocked).
See Also:	---

Details

Invoking lcd.lock 348 increases the value of this property by 1. If 255 is reached, the value does not roll over to 0 and stays at 255. Invoking lcd.unlock 356 decreases the value of this property by 1. When 0 is reached, the value does not roll over to 255 and stays at 0. The display is locked when lcd.lockcount is not at 0. Of course, the display does not get locked any "harder" with every increment of the lcd.lockcount.

When the display is locked, you can make changes to the display data without showing these changes on the screen. You can then unlock the display and show all the changes made at once. This usually greatly improves the display agility *perception* (see Improving Graphical Performance 330).

Not all controllers/panels support this feature. See the Supported Controllers/Panels 333 section for details on the display you are using. If your display does not support locking, executing lcd.lock will have no effect and lcd.lockcount 349 will always stay at 0.

.Paneltype R/O Property

Function:	Returns the type of the currently selected controller/ panel.
Type:	Enum (pl_lcd_paneltype, byte)

Value Range: 0- PL_LCD_PANELTYPE_GRAYSCALE: This is a monochrome or grayscale panel/controller.

1- PL_LCD_PANELTYPE_COLOR: This is a color panel/controller.

See Also: Understanding Controller Properties 318, Working With Pixels and Colors 320

Details

Controller/panel selection is made through the Customize Platform dialog, accessible through the Project Settings dialog.

Monochrome panels/controllers only allow you to turn pixels on and off. Grayscale panels/controllers allow you to set the brightness of pixels in steps. The number of available steps is defined by the number of bits assigned to each pixel (see lcd.bitsperpixel 339 property). Finally, color panels/controllers allow you to set the brightness separately for the red, green, and blue components of each pixel. Lcd.redbits 352, lcd.greenbits 345, and lcd.bluebits 340 R/O properties will tell you how many bits there are for each color "channel".

Panel/controller type affects how lcd.forecolor 344, lcd.backcolor 339, and lcd.setpixel 354 are interpreted. Also, the output produced by lcd.bmp 340 is affected by this.

.Pixelpacking R/O Property

Function: Indicates how pixels are packed into controller memory for the currently selected controller/panel.

Type: Enum (ver_hor, byte)

Value Range: 0- PL_VERTICAL: Several vertically adjacent pixels are packed into each byte of controller memory.

1- PL_HORIZONTAL: Several horizontally adjacent pixels are packed into each byte of controller memory.

See Also: Understanding Controller Properties 318

Details

Controller/panel selection is made through the Customize Platform dialog, accessible through the Project Settings dialog.

This property is only relevant for controllers/panels whose lcd.bitsperpixel 339 value is less than 8. In this case, 2, 4, or 8 pixels are packed into a single byte of controller memory.

This property is purely informational and largely has no influence over how you write your application. The only exception is related to working with text 322. Fonts 325 can also have vertical or horizontal packing and the speed at which you can output the text onto the screen is improved when the lcd.pixelpacking and lcd.fontpixelpacking 344 have the same value, i.e. controller memory pixels and font encoding are "aligned".

.Print Method

Function: Prints a line of text.

Syntax: **lcd.print(byref** str **as string,**x **as word,**y **as word) as word**

Returns: Total width of created output in pixels.

See Also: Working With Text 322, lcd.getprintwidth 345

Part	Description
str	Text to print.
x	X coordinate of the top-left corner of the text output. Value range is 0 to lcd.width 357 - 1.
y	Y coordinate of the top-left corner of the text output. Value range is 0 to lcd.height 346 - 1.

Details

For this method to work, a font must first be selected with the lcd.setfont 353 method. The lcd.textorientation 356 and lcd.texthorizontalspacing 355 properties affect how the text is printed.

This method always produces a single-line text output. Use lcd.printaligned 351 if you want to print several lines of text at once.

The display panel must be enabled (lcd.enabled 341 = 1- YES) for this method to work.

.Printaligned Method

Function: Print texts, on several lines if necessary, within a specified rectangular area.

Syntax: **lcd.printaligned(byref** str **as string,**x **as word,**y **as word,**width **as word,**height **as word) as byte**

Returns: Total number of text lines produced.

See Also: Working With Text 322

Part	Description
str	Text to print. Inserting ` character will create a line break.
x	X coordinate of the top-left point of the print area. Value range is 0 to lcd.width 357 - 1.
y	Y coordinate of the top-left point of the print area. Value range is 0 to lcd.height 346 - 1.
width	Width of the print area in pixels.

height	Height of the print area in pixels.

Details

For this method to work, a font must first be selected with the lcd.setfont[353] method. The lcd.textalignment[355], lcd.textorientation[356], lcd.texthorizontalspacing[355], and lcd.textverticalspacing[356] properties will affect how the text is printed.

This method breaks the text into lines to stay within the specified rectangular output area. Whenever possible, text is split without breaking up the words. A word will be split if it is wider than the width of the print area. You can add arbitrary line brakes by inserting ` (ASCII code 96).

The display panel must be enabled (lcd.enabled[341] = 1- YES) for this method to work.

.Rectangle Method

Function: Draws an unfilled rectangle.

Syntax: **lcd.rectangle(**x1 **as word,**y1 **as word,** x2 **as word,** y2 **as word)**

Returns: ---

See Also: Lines, Rectangles, and Fills[321], lcd.line[348], lcd.verline[357], lcd.horline[346], lcd.filledrectangle[343], lcd.fill[342]

Part	Description
x1	X coordinate of the first point. Value range is 0 to lcd.width[357]-1.
y1	Y coordinate of the first point. Value range is 0 to lcd.height[346]-1.
x2	X coordinate of the second point. Value range is 0 to lcd.width[357]-1.
y2	Y coordinate of the second point. Value range is 0 to lcd.height[346]-1.

Details

The rectangle is drawn with the specified line width (lcd.linewidth[348]) and "pen" color (lcd.forecolor[344]).

The display panel must be enabled (lcd.enabled[341] = 1- YES) for this method to work.

.Redbits R/O Property

Function: A 16-bit value packing two 8-bit parameters: number of "red" bits per pixel (high byte) and the position of the least significant red bit within the color word (low byte).

Type: Word

Value Range: Value depends on the currently selected controller/panel.

See Also: Understanding Controller Properties 318, Working With Pixels and Colors 320

Details

The value of this property depends on the currently selected controller/panel. Selection is made through the Customize Platform dialog, accessible through the Project Settings dialog. This property is only relevant for color panels (lcd.paneltype 349 = 1- PL_LCD_PANELTYPE_COLOR).

By taking the value of the high byte you can determine the number of the steps in which the brightness of the red "channel" can be adjusted. For example, if the high byte is equal to 5, then there are 32 levels for red.

This property also tells you the bit position and length of the red field in values used by lcd.forecolor 344, lcd.backcolor 339, and lcd.setpixel 354. If, for example, the lcd.redbits=&h0500, lcd.bluebits 340=&h0605, and lcd.greenbits 345=&h050B, then you can reconstruct the composition of the red, green, and blue bits in a word: bit 15 -> gggggbbbbbbrrrrr <- bit 0. In this example, the red field is the first field and occupies 5 bits (4-0).

.Rotated Property

Function: Specifies whether the image on the display panel is to be rotated 180 degrees.

Type: Enum (no_yes, byte)

Value Range: 0- NO: Not rotated **(default)**.

1- YES: Rotated 180 degrees.

See Also: Preparing the Display for Operation 320

Details

Set this property according to the orientation of the display panel in your device.

This property can only be changed when the display is disabled (lcd.enabled 341 = 0- NO).

.Setfont Method

Function: Selects a font to use for printing text.

Syntax: **lcd.setfont(**offset **as dword) as ok_ng**

Returns: 0- OK: The font was found and the data appears to be valid.

1- NG: There is no valid font data at specified offset.

See Also: Working with Text 322

Par	Description

t	
offset	Offset within the compiled binary of your application at which the font file is stored.

Details

A valid font file must be selected before you can use the lcd.print 351, lcd.printaligned 351, or lcd.getprintwidth 345 methods. Naturally, the font file must be present in your project for this to work (see how to add 132 a font file). To obtain correct offset, open the file using the romfile.open 374 method, then read the offset of this file from the romfile.offset 373 R/O property.

When the font file is successfully selected, the lcd.fontheight 343 and lcd.fontpixelpacking 344 R/O properties will be updated to reflect actual font parameters.

.Setpixel Method

Function:	Directly writes pixel data for a single pixel into the controller's memory.
Syntax:	**lcd.setpixel(**dt **as word,**x **as word,**y **as word)**
Returns:	---
See Also:	Working With Pixels and Colors 320, Understanding Controller Properties 318

Part	Description
dt	Pixel data to write.
x	X coordinate of the pixel. Value range is 0 to lcd.width 357 - 1.
y	Y coordinate of the pixel. Value range is 0 to lcd.height 346 - 1.

Details

Interpretation of the dt argument depends on the selected controller/panel. Selection is made through the Customize Platform dialog, accessible through the Project Settings dialog.

The dt argument is of word type, but only lcd.bitsperpixel 339 lower bits of this value will be relevant. All higher bits will be ignored.

For monochrome and grayscale controllers/panels (lcd.paneltype 349 = 0- PL_LCD_PANELTYPE_GRAYSCALE), the value of the dt argument sets the brightness of the pixel. For color panels/controllers (lcd.paneltype= 1- PL_LCD_PANELTYPE_COLOR) the value is composed of three fields -- one for the red, green, and blue "channels". Check lcd.redbits 352, lcd.greenbits 345, and lcd.bluebits 340 properties to see how the fields are combined into the dt word.

The display panel must be enabled (lcd.enabled 341 = 1- YES) for this method to work.

.Textalignment Property

Function: Specifies the alignment for text output produced by the lcd.printaligned 351 method.

Type: Enum (pl_lcd_text_alignment, byte)

Value Range: 0- PL_LCD_TEXT_ALIGNMENT_TOP_LEFT: Top, left **(default)**.

1- PL_LCD_TEXT_ALIGNMENT_TOP_CENTER: Top, center.

2- PL_LCD_TEXT_ALIGNMENT_TOP_RIGHT: Top, right.

3- PL_LCD_TEXT_ALIGNMENT_MIDDLE_LEFT: Middle, left.

4- PL_LCD_TEXT_ALIGNMENT_MIDDLE_CENTER: Middle, center.

5- PL_LCD_TEXT_ALIGNMENT_MIDDLE_RIGHT: Middle, right.

6- PL_LCD_TEXT_ALIGNMENT_BOTTOM_LEFT: Bottom, left.

7- PL_LCD_TEXT_ALIGNMENT_BOTTOM_CENTER: Bottom, center.

8- PL_LCD_TEXT_ALIGNMENT_BOTTOM_RIGHT: Bottom, right.

See Also: Working With Text 322, lcd.textorientation 356, lcd.texthorizontalspacing 355, lcd.textverticalspacing 356

Details

Lcd.printaligned 351 fits the text within a specified rectangular area. Lcd.textalignment defines how the text will be aligned within this area. The property has no bearing on the output produced by lcd.print 351.

.Texthorizontalspacing Property

Function: Specifies the gap, in pixels, between characters of text output produced by the lcd.print 351 and lcd.printaligned 351 methods.

Type: Byte

Value Range: 0- 255. **Default**= 1 (1 pixel).

See Also: Working With Text 322, lcd.textalignment 355, lcd.textorientation 356, lcd.textverticalspacing 356

Details

.Textorientation Property

Function: Specifies the print angle for text output produced by the lcd.print 351 and lcd.printaligned 351 methods.

Type: Enum (pl_lcd_text_orientation, byte)

Value Range: 0- PL_LCD_TEXT_ORIENTATION_0: At 0 degrees **(default)**.

1- PL_LCD_TEXT_ORIENTATION_90: At 90 degrees.

2- PL_LCD_TEXT_ORIENTATION_180: At 180 degrees.

3- PL_LCD_TEXT_ORIENTATION_270: At 270 degrees.

See Also: Working With Text 322, lcd.textalignment 355, lcd.texthorizontalspacing 355, lcd.textverticalspacing 356

Details

.Textverticalspacing Property

Function: Specifies the gap, in pixels, between the lines of text output produced by the lcd.printaligned 351 method.

Type: Byte

Value Range: 0- 255. **Default**= 1 (1 pixel).

See Also: Working With Text 322, lcd.textalignment 355, lcd.textorientation 356, lcd.texthorizontalspacing 355

Details

The property has no bearing on the output produced by lcd.print 351, because this method always creates a single-line output.

.Unlock Method

Function: Unfreezes display output (on controllers/panels that support this feature).

Syntax: **lcd.unlock**

Returns: ---

See Also: ---

Details

When the display is locked (see lcd.lock 348), you can make changes to the display data without showing these changes on the screen. You can then unlock the display with lcd.unlock and show all the changes made at once. This usually greatly improves the display agility perception by the user, even if your application is not

working any faster (see Improving Graphical Performance 330).

Each time you execute this method on a previously locked display, the value of the lcd.lockcount 349 R/O property decreases by 1. Once this value reaches 0, the display is unlocked and the user sees updated display data. The lcd.lockcount allows you to nest locks/unlocks (again, see Improving Graphical Performance 330).

Not all controllers/panels support this feature. See the Supported Controllers/Panels 333 section for details on the display you are using. If your display does not support locking, executing lcd.lock will have no effect and lcd.lockcount 349 will always stay at 0.

.Verline Method

Function: Draws a vertical line.

Syntax: **lcd.verline(**x **as word,**y1 **as word,**y2 **as word)**

Returns: ---

See Also: Lines, Rectangles, and Fills 321, lcd.rectangle 352, lcd.filledrectangle 343, lcd.fill 342

Part	Description
x	X coordinates of the first and second points. Value range is 0 to lcd.width 357 - 1.
y1	Y coordinate of the first point. Value range is 0 to lcd.height 346 - 1.
y2	Y coordinate of the second point. Value range is 0 to lcd.height 346 - 1.

Details

The line is drawn with the specified line widht (lcd.linewidth 348) and "pen" color (lcd.forecolor 344). Drawing horizontal (lcd.horline 346) or vertical lines is more efficient than drawing generic lines (lcd.line 348) and should be used whenever possible.

The display panel must be enabled (lcd.enabled 341 = 1- YES) for this method to work.

.Width Property

Function: Sets the horizontal resolution of the display panel in pixels.

Type: Word

Value Range: Appropriate value depends on the panel. **Default**= 0.

See Also: Preparing the Display for Operation 320, lcd.height 346

Details

Set this property according to the characteristics of your display panel.

The reason why this value is not set automatically when you select a certain

controller is because the capability of the controller may exceed the actual resolution of the panel, i.e. only "part" of the controller may be utilized.

This property can only be changed when the display is disabled (lcd.enabled 341 = 0-NO).

Net Object

The net object represents the Ethernet interface of your device. This object only specifies various parameters related to the Ethernet interface and is not responsible for sending/transmitting network data. The latter is the job of the sock 421 object.

Here is what you can do with the net object:

- Check if the Network Interface Controller (NIC) IC is functioning properly.
- Check your device's Ethernet MAC address.
- Set the IP-address of the Ethernet Interface.
- Set default gateway IP and the netmask.
- Be notified when the Ethernet cable is plugged or unplugged and check current link status.
- Be notified when data overflow occurs in the NIC.

Overview 7.1

Here you will find:

- Main Parameters 358 (IP, gateway IP, netmask, MAC).
- Checking Ethernet status 359 (link status change, failure, overflows).

Main Parameters

To enable Ethernet communications, you need to set the net.ip 360, net.gatewayip 361, and the net.netmask 360 properties. Actually, net.gatewayip and net.netmask are only needed when your device will be establishing outgoing connections to other hosts on the network (perform active opens). If your device will only be accepting incoming connections then you *do not have to* set the net.gatewayip and the net.netmask.

Strangely, a lot of people hold a passionate belief that default gateway IP and the netmask are necessary always, even for incoming connections. This is not true!

The net object is usually initialized once on startup, like this:

```
sub on_sys_init
      ... some other stuff

      net.ip = "192.168.1.95" 'just an example! May not work on your
```

```
network!
      net.gatewayip = "192.168.1.1" 'just an example! May not work on your
network!
      net.netmask= "255.255.255.0" 'just an example! May not work on your
network!

      ...some other stuff
end sub
```

On a lot of networks the IP, gateway IP, and the netmask parameters of the hosts are configured automatically, through the use of a special protocol called "DHCP". The net object does not support dhcp directly by we provide a BASIC library that implements DHCP functionality.

One additional read-only property- the net.mac 360- can be used to extract the MAC address of your device. Your program cannot change the MAC address directly. The MAC is stored in the special configuration area of the EEPROM. Access to the EEPROM is provided by the stor 522 object. To change the MAC address you need to rewrite the data in the EEPROM. For more details, see the Stor Object 522 and Stor.Base 523 Property topics.

Checking Ethernet Status

The net.failure 361 read-only property tells you if the NIC is functioning properly.

The net.linkstate 361 read-only property tells you if there is a live Ethernet cable plugged into the Ethernet port of your device, and, if yes, whether this is a 10BaseT or 100BaseT connection. The on_net_link_change 362 event is generated each time the link status changes:

```
sub on_net_link_change

      if net.linkstate= PL_NET_LINKSTATE_NOLINK then
            'switch the RED LED on (just an example of what you could do)
            pat.play("RRRRRRRRRRRRRRR", YES)
      else
            'switch the GREEN LED on
            pat.play("GGGGGGGGGGGGGGG", YES)
      end if

end sub
```

Notice, that the net.linkstate always reflects current link status, not the link at the time of event generation.

Finally, there is a on_net_overrun 362 event that is generated when internal RX buffer of NIC overflows.

Properties, Methods, Events

This section provides an alphabetical list of all properties, methods, and events of the net object.

.Mac R/O Property

Function:	Returns the MAC address of the Ethernet interface.
Type:	Dot-decimal string
Value Range:	Any valid MAC address, i.e. "0.1.2.3.4.5". Each device is preset with individual MAC address during production.
See Also:	---

Details

BASIC application cannot change MAC address directly. The MAC is stored in the EEPROM memory. This is the same memory used by the stor 522 object. On power-up the MAC address is loaded from the EEPROM and programmed into the Ethernet controller of the device. Stor object provides a way to change MAC address in the EEPROM- see Stor Object 522 for details.

.Ip Property

Function:	Sets/returns the IP address of the Ethernet interface of your device.
Type:	Dot-decimal string
Value Range:	Any IP address, such as "192.168.100.40". **Default**= "1.0.0.1"
See Also:	net.gatewayip 361, net.netmask 360

Details

This property can only be written to when no socket is engaged in communications through the Ethernet interface, i.e. there is no socket for which sock.statesimple 505 <> 0- PL_SSTS_CLOSED and sock.currentinterface 478 = 1- PL_INTERFACE_ETHERNET.

.Netmask Property

Function:	Sets/returns the netmask of the Ethernet interface of your device.
Type:	String
Value Range:	Any valid netmask, such as "255.255.255.0". **Default**= "0.0.0.0"
See Also:	net.ip 360, net.gatewayip 361

Details

This property can only be written to when no socket is engaged in communications through the Ethernet interface, i.e. there is no socket for which sock.statesimple 505 <> 0- PL_SSTS_CLOSED and sock.currentinterface 478 = 1- PL_INTERFACE_ETHERNET.

.Gatewayip Property

Function:	Sets/returns the IP address of the default gateway for the Ethernet interface of your device.
Type:	String
Value Range:	Any valid IP address, such as "192.168.100.40". **Default**= "127.0.0.1"
See Also:	net.ip 360, net.netmask 360

Details

This property can only be written to when no socket is engaged in communications through the Ethernet interface, i.e. there is no socket for which sock.statesimple 505 <> 0- PL_SSTS_CLOSED and sock.currentinterface 478 = 1- PL_INTERFACE_ETHERNET.

.Failure R/O Property

Function:	Reports whether the Network Interface Controller (NIC) IC has failed.
Type:	Enum (no_yes, byte)
Value Range:	0- NO **(default)**: No failure 1- YES: NIC failure
See Also:	---

Details

.Linkstate R/O Property

Function:	Returns current link status of the Ethernet port of the device.
Type:	Enum (pl_net_linkstate, byte)
Value Range:	0- PL_NET_LINKSTAT_NOLINK: No physical Ethernet link exists at the moment (the Ethernet port of the device is not connected to a hub).

1- PL_NET_LINKSTAT_10BASET: The Ethernet port of the device is linked to a hub (or directly to another device) at 10Mbit/sec.

2- PL_NET_LINKSTAT_100BASET: The Ethernet port of the device is linked to a hub (or directly to another device) at 100Mbit/sec.

See Also: on_net_link_change 362

Details

On_net_link_change Event

Function: Generated when the state of the physical link of Ethernet port changes.

Declaration: **on_net_link_change**

See Also: ---

Details

Multiple on_net_link_change events may be waiting in the event queue. This event does not "bring" with it new link state at the time of event generation. Current link state can be queried through the net.linkstate 361 property.

On_net_overrun Event

Function: Generated when overflow occurs on the internal RX buffer of the Network Interface Controller (NIC) IC.

Declaration: **on_net_overrun**

See Also: ---

Details

Another on_net_overrun event is never generated until the previous one is processed. Notice, that this event signifies the overrun of the hardware RX buffer of the NIC itself. This has nothing to do with the overrun of RX buffers of individual sockets (see on_sock_overrun 491 event).

Pat Object

The pat object allows you to "play" signal patters on up to five LED pairs, each pair consisting of a green and red LED.

The channel to work with is selected through the pat.channel 364 property. The first channel (channel 0) is the primary channel of your system. It utilizes green and red status LEDs that are present on all external devices, boards, and some modules offered by Tibbo. All modules have SG and SR I/O lines that are meant for controlling external status LEDs. Note that when the Tibbo BASIC application is not running, green and red status LEDs are used to display various status information 200 .

The remaining four channels (channel 1-4) are identical in function, but use regular I/O lines of Tibbo devices. Moreover, pat.greenmap 364 and pat.redmap 366 properties allow you to flexibly map the green and red LED control lines of each channel to any I/O lines of the device.

The pattern you play can be up to 16 steps long. Each "step" can be either "-" (both LEDs off), "R" (red LED on), "G" (green LED on), or "B" (both LEDs on). You can also define whether the pattern will only execute once or loop and play indefinitely. Additionally, you can make the pattern play at a normal, double, or quadruple speed.

You load the new pattern to play with the pat.play 365 method. If the pattern is looped it will continue playing until you change it. If the pattern is not looped it will play once and then the on_pat 365 event will be generated. When the event handler is entered, the pat.channel property will be automatically set to the channel number for which the event was generated.

LED patterns offer a convenient way to tell the user what your system is doing. You can devise different patterns for different states of your device.

Here is a simple example in which we keep the green LED on at all times, except when the button is pressed, after which the green LED is turned off and the red LED blinks three times *fast*. Additionally, both green and red LEDs blink 4 times on startup. In this example we work on channel 0:

```
Sub On_sys_init
   pat.channel=0 'not really necessary since 0 is the default value for this
property
   pat.play("B-B-B-B-",PL_PAT_CANINT)
End Sub

Sub On_button_pressed
   pat.play("*R-R-R-",PL_PAT_CANINT)
End Sub

Sub On_pat
      If pat.channel=0 Then 'not really necessary since we are not using any
other channels
      pat.play("~G",PL_PAT_CANINT)
   End If
End Sub
```

In the above example, the power-up pattern is loaded inside the on_sys_init 533 event handler. This is not a looped pattern, so once it finishes playing the on_pat 365 event is generated and the "permanent" pattern "green LED on" is loaded inside

this event's handler. This new pattern is looped (notice "~"). When the button is pressed, a fast pattern (notice "*") is loaded. This one makes the red LED blink three times. Again, this is not a looped pattern, so after it finishes playing the on_pat event is generated and the "permanent green" pattern is loaded again.

.Channel Property

Function: Selects/returns the LED channel (status LED pair) to work with.

Type: Byte

Value Range: The value of this property won't exceed 4 (even if you attempt to set higher value). **Default**= 0 (channel 0 selected).

See Also: ---

Details

Channels are enumerated from 0. All other properties, methods, and an event of this object relate to the currently selected channel. Note that this property's value will be set automatically when the event handle for the on_pat 365 event is entered.

.Greenmap Property

Function: For the selected LED channel (selection is made through the pat.channel 364 property), sets/returns the number of the I/O line that will act as a green LED control line.

Type: Enum (pl_int_num, byte)

Value Range: Platform-specific, see the list of pl_int_num constants in the platform specifications.

Default values:

Channel 0: **(-1)** (no mapping, read-only): the green status LED (control line) of Tibbo device is always used by this channel;

Channels 1-4: **PL_IO_NULL** (non-existent line).

See Also: Pat.redmap 366

Details

Channel 0 is special -- its LED control lines can't be remapped. This is because channel 0 uses standard green and red status LEDs (they are called SG and SR). For channel 0, reading the property always returns (-1), and writing has no effect.

All other channels use regular I/O lines of Tibbo devices. Any I/O line can be selected to be the green control line of the selected channel. By default, all control lines are mapped to the non-existent line PL_IO_NULL. Remap as needed and don't forget to configure the selected I/O line as an output -- this won't happen automatically.

On_pat Event

Function: Generated when an LED pattern finishes playing.

Declaration: **on_pat**

See Also: Pat.play 365

Details

This can only happen for "non-looped" patterns. Multiple on_pat events may be waiting in the event queue. When the event handler for this event is entered the pat.channel 364 property is automatically set to the channel for which this event was generated.

.Play Method

Function: Loads a new LED pattern to play on the currently selected LED channel (selection is made through the pat.channel 364 property).

Syntax: **pat.play(byref** pattern **as string,** patint **as pl_pat_int)**

Returns: ---

See Also: ---

Part	Description
pattern	Pattern string, can include the following characters: '-': both LEDs off 'R' or 'r': red LED on 'G' or 'g': green LED on 'B' or 'b': both LEDs on '~': looped pattern (can reside anywhere in the pattern string) '*': double-speed pattern (can reside anywhere in the pattern string). You can use this symbol twice to x4 speed. Adding even more '*' will not have any effect.
patint	Defines whether the pat.play method is allowed to interrupt another pattern that is already playing: 0- PL_PAT_NOINT: cannot interrupt 1- PL_PAT_CANINT: can interrupt)

Details

Maximum pattern length is 16 "steps". The on_pat 365 event is generated once the pattern finishes playing. Looped patterns never finish playing and thus the event is never generated for them.

Note that channels 1-4 require you to map LED control lines. See pat.greenmap 364 and pat.redmap 366 properties for details.

.Redmap Property

Function:	For the selected LED channel (selection is made through the pat.channel 364 property), sets/returns the number of the I/O line that will act as a red LED control line.
Type:	Enum (pl_int_num, byte)
Value Range:	Platform-specific, see the list of pl_int_num constants in the platform specifications. **Default values:** Channel 0: **(-1)** (no mapping, read-only): the green status LED (control line) of Tibbo device is always used by this channel; Channels 1-4: **PL_IO_NULL** (non-existent line).
See Also:	Pat.greenmap 364

Details

Channel 0 is special -- its LED control lines can't be remapped. This is because channel 0 uses standard green and red status LEDs (they are called SG and SR). For channel 0, reading the property always returns (-1), and writing has no effect.

All other channels use regular I/O lines of Tibbo devices. Any I/O line can be selected to be the red control line of the selected channel. By default, all control lines are mapped to the non-existent line PL_IO_NULL. Remap as needed and don't forget to configure the selected I/O line as an output -- this won't happen automatically.

Ppp Object

The ppp. object represents the PPP interface of your device. This is the interface that uses one of the serial ports of your device and works with traditional landline or GPRS modems.

The ppp. object itself does not handle the PPP link negotiation, which is quite complex. This needs to be done in Tibbo BASIC code. For GPRS modems there is a very convenient GPRS library 645 that implements the entire link negotiation process.

The ppp. object only specifies parameters related to the PPP interface and is not responsible for sending/transmitting network data. The latter is the job of the sock.

421 object. You will find PPP interface listed or available on the following sock. object's properties:

- Sock.availableinterfaces 475
- Sock.allowedinterfaces 474
- Sock.currentinterface 478
- Sock.targetinterface 506

.Buffrq Method

Function:	Pre-requests "numpages" number of buffer pages (1 page= 256 bytes) for the TX buffer of the ppp. object.
Syntax:	**ppp.buffrq(**numpages **as byte) as byte**
Returns:	Actual number of pages that can be allocated (byte).
See Also:	ppp.buffsize 367

Part	Description
numpages	Requested numbers of buffer pages to allocate (recommended value is **5**).

Details

Actual allocation happens when the sys.buffalloc 530 method is used.

The PPP object will be unable to operate properly if its buffer has inadequate capacity. Recommended buffer size is 5 pages.

The buffer can only be allocated when the PPP channel is not enabled (ppp.enabled 368= 0- NO). Executing sys.buffalloc while ppp.enabled= 1- YES will leave the buffer size unchanged.

The actual current buffer size can be verified through the ppp.buffsize 367 read-only property.

.Buffsize R/O Property

Function:	Returns the current capacity (in bytes) of the ppp. object's buffer.
Type:	Word
Value Range:	0-65535, **default**= 0 (0 bytes).
See Also:	wln.buffrq 367

Details

Buffer capacity can be changed through the ppp.buffrq 367 method followed by the sys.buffalloc 530 method invocation.

The PPP object will be unable to operate properly if its buffer has inadequate capacity. Recommended buffer size is **5 pages**.

.Enabled Property

Function: Enables/disables PPP interface on the serial port specified by the ppp.portnum 368 property.

Type: Enum (no_yes, byte)

Value Range: 0- NO **(default)**: not enabled
1- YES: enabled

See Also: ---

Details

Once this property is set to 1- YES, the selected serial port seizes to be under the control of your application and works exclusively for the ppp. object.

PPP channel setup (ppp.buffrq 367, ppp.ip 368, ppp.portnum 368) can only be altered when the ppp. object is disabled.

.Ip Property .4

Function: Sets/returns the IP address of the PPP interface of your device.

Type: Dot-decimal string

Value Range: Any IP address, such as "192.168.100.40". **Default**= "1.0.0.1".

See Also: ---

Details

Typically, the IP address of the PPP interface is negotiated with the ISP. Our (very convenient) GPRS library 645 implements all necessary steps of PPP link negotiation for GPRS modems.

This property can only be written to when the ppp. object is disabled (ppp.enabled 368 = 0- NO).

.Portnum Property

Function: Sets/returns the number of the serial port that will be used by the ppp. object.

Type: Byte

Value Range: 0 to ser.numofports 411 - 1

See Also: ---

Details

Once the PPP interface is enabled (ppp.enabled 368 = 1- YES), the selected serial port seizes to be under the control of your application and works exclusively for the

ppp. object.

The value of this property won't exceed ser.numofports 411 -1 (even if you attempt to set a higher value).

You can only change this property when the PPP channel is disabled (ppp.enabled= 0- NO).

Pppoe Object

The pppoe. object represents the PPPoE interface of your device. This is the interface that works over the Ethernet (net. 358) interface to carry the data between the device and the ADSL modem, the kind that needs user name and login to connect to the Internet. More about PPPoE here: http://en.wikipedia.org/wiki/Pppoe.

It is important to understand that net. and pppoe. represent two different network interfaces of your system, even though they both send and receive data through the same Ethernet wire. For example, PPPoE interface has its own IP address (pppoe.ip 370), which is different from net.ip 360.

The pppoe. object itself does not handle the ADSL login, which is quite complex. This task is performed by the PPPOE library 655. This library's code will perform all necessary login and configuration steps, and then set the only three properties that are needed for pppoe. object's operation:

- The IP address of your device on the PPPoE interface (pppoe.ip 370);
- The MAC address of the ADSL modem, a.k.a. "access concentrator" (pppoe.acmac 369);
- Session ID (pppoe.sessionid 370).

It follows from the above that the PPPoE interface is not automatically ready after the device boot. It has to be properly configured by the PPPoE library.

The pppoe. object only specifies parameters related to the PPPoE interface and is not responsible for sending/transmitting network data. The latter is the job of the sock. 421 object. You will find PPPoE interface listed or available on the following sock. object's properties:

- Sock.availableinterfaces 475
- Sock.allowedinterfaces 474
- Sock.currentinterface 478
- Sock.targetinterface 506

.Acmac Property

Function:	Sets/returns the MAC address of the ADSL modem (a.k.a. "access concentrator").
Type:	Dot-decimal string
Value Range:	Any MAC address, i.e. "0.1.2.3.4.5". **Default**= "0.0.0.0.0.0"
See Also:	---

Details

This property uniquely identifies the ADSL modem (access concentrator) that your device will use to access the Internet.

.Ip Property 0.2

Function:	Sets/returns the IP address of the PPPoE interface of your device.
Type:	Dot-decimal string
Value Range:	Any IP address, such as "192.168.100.40". **Default**= "0.0.0.0"
See Also:	---

Details

This property can only be written to when no socket is engaged in communications through the PPPoE interface, i.e. there is no socket for which sock.statesimple 505 <> 0- PL_SSTS_CLOSED and sock.currentinterface 478 = 1- PL_INTERFACE_PPPOE.

.Sessionid Property

Function:	Sets/returns the ID of the current PPPoE session.
Type:	Word
Value Range:	0-65535. **Default**= 0
See Also:	---

Details

Session ID is required for correct interaction between your device and ADSL modem (access concentrator). Use the PPPOE library 655 and let it take care of this and (almost) everything else.

Romfile Object

The romfile object allows you to access resource (ROM) files of your project. Resource files appear in the "Resource Files" folder of your project tree. These are files that are not processed by the compiler in any way -- they are just added to the compiled binary. Resource files are ideal for storing permanent data that never changes.

Data bytes in resource files are counted from 1.

Calling romfile.open 374 opens a resource file. Only one file can be opened at any given time and there is no need to close it. The size of the file you have opened can be checked through the romfile.size 375 read-only property. If you attempt to open a non-existent file its size will be returned as 0. This is how you know that the file does not exist!

There is a file pointer. When you open or reopen the file the pointer is set to 1 (the first byte in the file), unless the file does not exist or empty, in which case the pointer will be at 0.

You read the data from the file using the romfile.getdata 373 method. As you read from the file, the pointer moves - each time by the number of characters you've just read. The pointer can't exceed romfile.size+1 (unless the file does not exist or is empty, in which case the pointer will always be at 0).

You can read and set the pointer position directly using two properties -- romfile.pointer 374 and romfile.pointer32 374. The difference between them is that romfile.pointer is of the word type 48 and, hence, can only be used to set the pointer in the 1-65535 range. If the pointer is already above 65535, romfile.pointer will still return 65535.

The romfile.pointer32 property is of the dword type 48 and can handle all pointer positions. Use it if you access resource files that are larger than 65535 bytes. The disadvantage of romfile.pointer32 is that using it will result in reduced code performance. For this reason, rely on romfile.pointer whenever possible.

The romfile.find 372 and romfile.find32 372 methods search through the currently opened resource file. For example, supposing you have the following data in the file <parameters.txt>:

```
IP=192.168.1.40
PORT=1001
...
```

This sample data represents the list of parameters that your program uses. Now, supposing you need to extract the value of the "PORT" parameter. Here is the code:

```
dim dw,dw2 as dword
dim s as string(10)

'look for "PORT" first
romfile.open("resource2.txt")
dw=romfile.find(1,"PORT",1)
if dw=0 then sys.halt
dw=dw+len("PORT=")

'OK, now loop for CR after "PORT"
dw2=romfile.find(dw,chr(13),1)
if dw2=0 then sys.halt

'extract the value
romfile.pointer=dw
s=romfile.getdata(dw2-dw)
's now contains the port number
```

The difference between romfile.find 372 and romfile.find32 372 is that the former is of the word type 48 and the latter is of the dword type 48. Romfile.find can only "report back" file positions that are in the 1-65535 range. If the target instance of the substring was found beyond the 65535 position, romfile.find will still return 65535. Romfile.find32 doesn't have this limitation, but it will slow down your code, so use romfile.find whenever possible.

Sometimes you will need to open a resource file not for the purpose of accessing it, but for the purpose of "passing" this file to another object. Such reference is provided by the romfile.offset 373 read-only property. For illustration of use, see lcd.setfont 353 and wln.boot 558.

.Find Method .1

Function: Locates the Nth occurrence of a substring within the currently opened resource (ROM) file.

Syntax: **romfile.find(**frompos **as dword, byref** substr **as string,** num **as word) as word**

Returns: 16-bit value indicating the file position at which the specified occurrence of the substring was found or 0 if the specified occurrence wasn't found. If the specified occurrence was found at file position 65535 or higher, the value of 65535 will be returned.

See Also: Romfile Object 370

Part	Description
frompos	Starting search position in the file.
substr	Substring to search for.
num	Substring occurrence to search for.

Details

Bytes in the resource files are counted from 1. Use romfile.find32 372 if you are searching within a file larger than 65535 bytes.

.Find32 Method

Function: Locates the Nth occurrence of a substring within the currently opened resource (ROM) file.

Syntax: **romfile.find32(**frompos **as dword, byref** substr **as string,** num **as word) as dword**

Returns: 32-bit value indicating the file position at which the specified occurrence of the substring was found or 0 if the specified occurrence wasn't found.

See Also: Romfile Object 370

Part	Description
frompos	Starting search position in the file.
substr	Substring to search for.
num	Substring occurrence to search for.

Details

Bytes in the resource files are counted from 1. When searching inside files that do not exceed 65535 bytes, use romfile.find 372 instead -- this will speed up your application.

.Getdata Method

Function:	Reads data from a currently opened resource (ROM) file.
Syntax:	**romfile.getdata(**maxinplen **as byte) as string**
Returns:	String containing a part of the ROM file's data
See Also:	Romfile Object 370

Part	Description
maxinplen	Maximum number of characters to read from the file.

Details

The actual return string length is limited by three factors, whichever is smaller: the capacity of the receiving string, the amount of remaining data in the file (romfile.size 375+1-romfile.pointer 374), and the maxinplen argument.

Invoking this method moves the current pointer position forward by the actual number of bytes read.

.Offset R/O Property

Function:	For the currently opened resource (ROM) file returns the absolute file offset in the compiled project binary.
Type:	Dword
Value Range:	---
See Also:	Romfile Object 370

Details

The property is used to pass the file data to some other object that may need this data. This way, the "separation of labor" is maintained between the objects. The romfile object opens the ROM file (romfile.open 374) and passes the pointer to this file (through romfile.offset) to another object that requires this information.

For illustration of use, see lcd.setfont 353 and wln.boot 558.

.Open Method5

Function: Opens or re-opens a resource (ROM) file.

Syntax: **romfile.open(byref** filename **as string)**

Returns: ---

See Also: Romfile Object 370

Part	Description
filename	Name of the resource file to open.

Details

'If the file exists and is not empty, the romfile.pointer 374 and romfile.pointer32 374 properties are set to 1 (each time you (re)open the file). If the file doesn't exist or is empty, these two properties are set to 0.

There is no method (or need) to explicitly close resource files. Only one resource file can be opened at any given time.

.Pointer Property

Function: Sets/returns the current pointer position in the resource (ROM) file.

Type: Word

Value Range: See the explanation below.

See Also: Romfile Object 370

Details

When the file is (re)opened with the romfile.open 374 method, the pointer is reset to the first character of the file (position 1). If the file is not found or contains no data, the pointer is set to 0.

Pointer position cannot exceed romfile.size 375+1. When you read from the file with romfile.getdata 373, the pointer is automatically moved forward by the number of bytes that have been read out.

Since this property is of the word type, the maximum pointer value you can set with it is 65535. Reading the current pointer position with this property will return 65535 for all pointer positions from 65535 and up. To navigate within the files that exceed 65535 bytes, use romfile.pointer32 374 instead.

.Pointer32 Property

Function: Sets/returns the current pointer position in the resource (ROM) file.

Type:	Dword
Value Range:	See the explanation below.
See Also:	Romfile Object 370

Details

When the file is (re)opened with the romfile.open 374 method, the pointer is reset to the first character of the file (position 1). If the file is not found or contains no data, the pointer is set to 0.

Pointer position cannot exceed romfile.size 375+1. When you read from the file with romfile.getdata 373, the pointer is automatically moved forward by the number of bytes that have been read out.

To navigate within files that do not exceed 65535 bytes, use romfile.pointer 374 instead -- this will speed up your application.

.Size R/O Property

Function:	Returns the size of the currently opened resource (ROM) file.
Type:	Dword
Value Range:	---
See Also:	romfile.open 374

Details

Zero size is returned when the file does not exist or the file is empty.

RTC Object

The RTC object facilitates access to the real-time counter (RTC) of the device. The RTC is an independent hardware counter that has its own power input. When the backup battery is installed, the RTC will continue running even when the rest of the device is powered off. The RTC keeps track of elapsed days, minutes, and seconds, not actual date and time. This is why it is called the "counter", not "clock". This platform includes a set of convenient date and time conversion syscalls that can be used to transform RTC data into current date/time and back (see year 230, month 221, date 208, weekday 230, daycount 209, hours 213, minutes 221, and mincount 220).

Two methods of the RTC object- rtc.getdata 376 and rtc.setdata 377 are used to read and set the counter value.

Supposing that your project has curr_year, curr_month, curr_date, curr_hours, curr_minutes, and curr_seconds variables, here is how you set and get the time:

```
dim curr_year, curr_month, curr_date, curr_hours, curr_minutes,
curr_seconds as byte
dim x,y as word
...
...
'set RTC now (data is supposed to be prepared in curr_ variables)
rtc.set(daycount(curr_year,curr_month,curr_date),mincount(curr_hours,
curr_minutes),curr_seconds)
...
...
'get RTC now- first get the daycount and mincout into x and y
rtc.getdata(x,y,curr_seconds)
'now convert daycount and mincount into date and time
curr_year=year(x)
curr_month=month(x)
curr_date=date(x)
curr_hours=hours(y)
curr_minutes=minutes(y)
```

There is also rtc.running 377 read-only property that tells you whether the RTC is working.

- Notice, that after the device powers up and provided that the backup power was not present at all times the RTC may be in the undetermined state and not work properly until the rtc.set method is executed at least once. Incorrect behavior may include failure to increment or incrementing at an abnormal rate. Rtc.running cannot be used to reliably check RTC state in this situation.

  It is not necessary to use rtc.set if the backup power was present at all times while the device was off.

.Getdata Method (Previously .Get)

Function:	Returns current RTC data as the number of elapsed days, minutes and seconds.
Syntax:	**rtc.getdata(byref** daycount **as word, byref** mincount **as word, byref** seconds **as byte)**
Returns:	Number of elapsed days, minutes, and seconds. Notice that this is done indirectly, through byref arguments.
See Also:	RTC Object 375

Part	Description
daycount	Number of elapsed days.
mincount	Number of elapsed minutes.
seconds	Number of elapsed seconds.

Details

Some platforms include a set of convenient date and time conversion functions that can be used to transform "elapsed time" values into current weekday, date, and time (see weekday 230, year 230, month 221, date 208, hours 213, minutes 221).

- When the RTC is powered up after being off (that is, device had not power AND no backup power for a period of time), it may not work correctly until you set it using rtc.set 377 method. Incorrect behavior may include failure to increment or incrementing at an abnormal rate. Rtc.running cannot be used to reliably check RTC state in this situation.

  It is not necessary to use rtc.set if the backup power was present while the device was off.

With Tibbo Basic release **V2**, this method had to be renamed from .get to .getdata. This is because the period (".") separating "rtc" from "getdata" is now a "true" part of the language, i.e. it is recognized as a syntax unit, not just part of identifier. Hence, Tibbo Basic sees "rtc" and "get" as separate entities and "get" is a reserved word that can't be used.

.Running R/O Property

Function:	Returns current RTC state.
Type:	Enum (no_yes, byte)
Value Range:	0- NO: RTC is not running. 1- YES: RTC is running.
See Also:	RTC Object 375

Details

When this property returns 0- NO this typically is the sign of a hardware malfunction (for instance, RTC crystal failure).

- When the RTC is powered up after being off (that is, device had not power AND no backup power for a period of time), it may not work correctly until you set it using rtc.set 377 method. Rtc.running cannot be used to reliably check RTC state in this situation.

.Setdata Method (Previously .Set)

Function:	Presets the RTC with a number of elapsed days , minutes , and seconds.
Syntax:	**rtc.setdata(**daycount **as word,** mincount **as word,** seconds **as byte)**
Returns:	---
See Also:	RTC Object 375

Part	Description
daycount	Number of elapsed days.
mincount	Number of elapsed minutes.
seconds	Number of elapsed seconds.

Details

Some platforms includes a set of convenient date and time conversion functions that can be used to transform actual date into time into "elapsed time" values (see daycount 209 and mincount 220).

Notice, that after the device powers up and provided that the backup power was not present at all times the RTC may be in the undetermined state and not work properly until the rtc.set method is executed at least once. Incorrect behavior may include failure to increment or incrementing at an abnormal rate. Rtc.running cannot be used to reliably check RTC state in this situation.

It is not necessary to use rtc.set if the backup power was present at all times while the device was off.

With Tibbo Basic release **V2**, this method had to be renamed from .get to .getdata. This is because the period (".") separating "stor" from "setdata" is now a "true" part of the language, i.e. it is recognized as a syntax unit, not just part of identifier. Hence, Tibbo Basic sees "stor" and "set" as separate entities and "set" is a reserved word that can't be used.

Ser Object

This is the *serial port* object. This object encompasses all serial ports available on a particular platform. Each serial port can function as a standard UART 380, Wiegand 383 device, or clock/data 386 device. Direct support for Wiegand and clock/data interfaces is a unique feature of the serial port object.

Each serial port has a set of standard programmable parameters that you would expect to find, such as baudrate and parity. In addition, it has some unique features like the ability to automatically filter out so-called 'escape sequences' or select UART, Wiegand, or clock/data operating mode.

Follows is the list of features offered by the serial port object:

- Ability to work in UART, Wiegand, or clock/data mode
- Fully asynchronous operation with separate "data arrival" and "data sent" events.
- Automatic data overrun detection on the RX buffer.
- Adjustable receive (RX) and transmit (TX) buffer sizes for optimal RAM utilization.
- Data "grouping" in the RX buffer based on "intercharacter" delay (gap between two consecutive arriving characters) and amount of data in the buffer.

- Optional automatic port disabling when a data group has been received.
- Buffer shorting feature for fast data exchange between the ser object and other objects (such as the sock 421 object) that support standard Tibbo Basic data buffers.

For the UART mode:

- Ability to set any baudrate (that is physically possible on a particular platform) by specifying "divider value" instead of a "baudrate from a list" as is usually done on other products
- Standard parity selection: parity off, even, odd, mark, or space.
- Choice of word length- 7 or 8 data bits/word.
- Full duplex or half duplex operation.
- Optional automatic CTS/RTS flow control in the full-duplex mode.
- Automatic direction control via RTS line in the half-duplex mode with direction control polarity selection.
- Automatic detection of "escape sequences", active even when the buffer shorting is enabled (see below).

Overview 13.1

This section covers the serial port object in detail. Here you will find:

Anatomy of a Serial Port

A serial port is composed of actual hardware which controls serial port lines, and of buffers that store incoming data (which is to be processed by your application) and outgoing data (which has not yet left the port).

The serial port object contains properties, methods and events which relate both to the buffers and the UART itself (see Serial Settings 390).

The *buffers* available are:

- The **TX buffer**, which contains data due to be sent out of the port (i.e, it's the transmit *of your device!*). Your Tibbo Basic application puts the data into the TX buffer.
- The **RX buffer**, which contains incoming data received by the port. This data is to be processed by your application.

The *logical lines* available are:

- The **TX/W1out/dout output** line.
- The **RX/W1in/din input** line.
- The **RTS/W0out/cout output** line.
- The **CTS/W0&1in/cin input** line.

TX/W1out/dout and RX/W1in/din lines always have fixed "position" in the device i.e. they cannot be re-mapped to a different I/O pin. RTS/W0out/cout and CTS/W0&1in/cin lines can be re-mapped on select devices. Also, depending on the device and the serial port mode you may or may not require to explicitly configure

the lines of the serial port as inputs or outputs. Sometimes it will happen automatically, and sometimes you need to take care of this in your application through the io 294 object. You will find this information in the "Platform-related Programming Information" topic inside your platform specifications section.

Details of I/O line usage in each of the three operating modes of the port can be found here 380.

Three Modes of the Serial Port

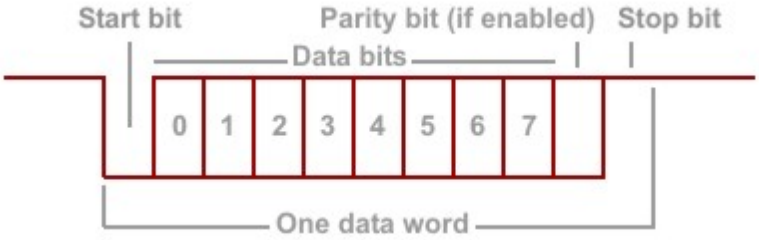
Each port of the serial object can operate in one of the three modes:

- UART Mode 380
- Wiegand Mode 383
- Clock/data Mode 386

The following subsections detail the signals that the serial port sends and expects to receive in each mode.

UART Mode

In the UART mode the serial port has standard UART functionality. You can select the baudrate, parity, number of bits in each character, and full- or half-duplex operation (see Serial Settings 390). The UART works with signals of "TTL-serial" polarity -- RX and TX lines are at logical HIGH when no data transmission is taking place, the start bit is LOW, stop bit is HIGH (shown below for the case of 8 bits/character and enabled parity).



UART data is sent and received via **TX** output and **RX** input lines. Two additional lines- **RTS** output and **CTS** input may also be used depending on the serial port setup (see below).

Please, remember that on your platform you may be required to correctly configure 194 some of your serial port's lines as inputs or outputs through the io.enabled 298 property of the io 294 object. Additionally, you may have the freedom of re-mapping certain serial port lines to different I/O pins of the device if required. For more information, refer to your device's platform documentation (for example, EM1000's is here 143).

How the serial port sends and receives UART data

The serial port can send and receive the data with no parity, or even, odd, mark, or space parity configuration. Additionally, you can specify the number of bits in each

character (7 or 8). The serial port takes care of parity calculation automatically. When parity bit is enabled, it will automatically calculate parity bit value for each character it transmits. When receiving, the serial port will correctly process incoming characters basing on the specified number of bits and parity mode. Actual parity check is not done. The serial port will receive the parity bit but won't actually check if it is correct.

Actual parity check is not done. The serial port will receive the parity bit but won't actually check if it is correct. Parity is mostly kept for compatibility with older devices, so the serial port transmits it correctly.

How the UART data is stored in the RX and TX buffers of the serial port

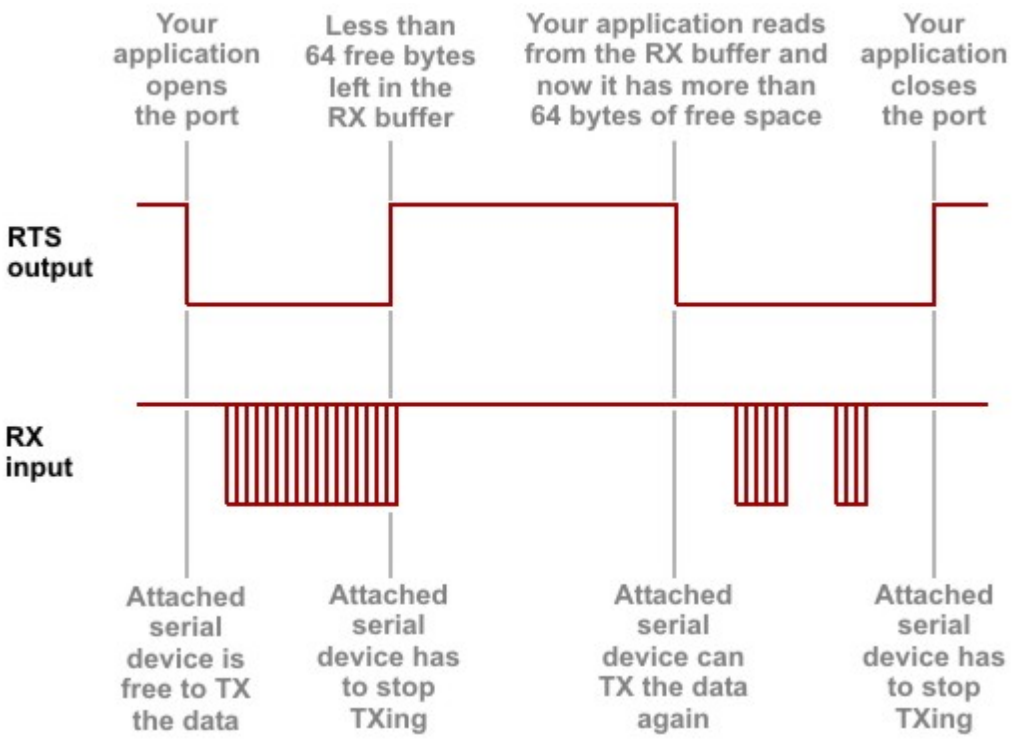
When in the UART mode, each data byte in the TX or RX buffer of the serial port represents one character. When the serial port is configured to send and receive 7-bit characters, the most significant bit of each byte in the RX buffer will be 0, and the most significant bit of each byte in the TX buffer will be ignored. Parity bits are not stored in the buffers. For the outgoing data stream, the serial port will calculate and append the parity bit automatically. For the incoming serial data, the serial port will discard the parity bit so the RX buffer will only get "pure" data.

Full-duplex operation

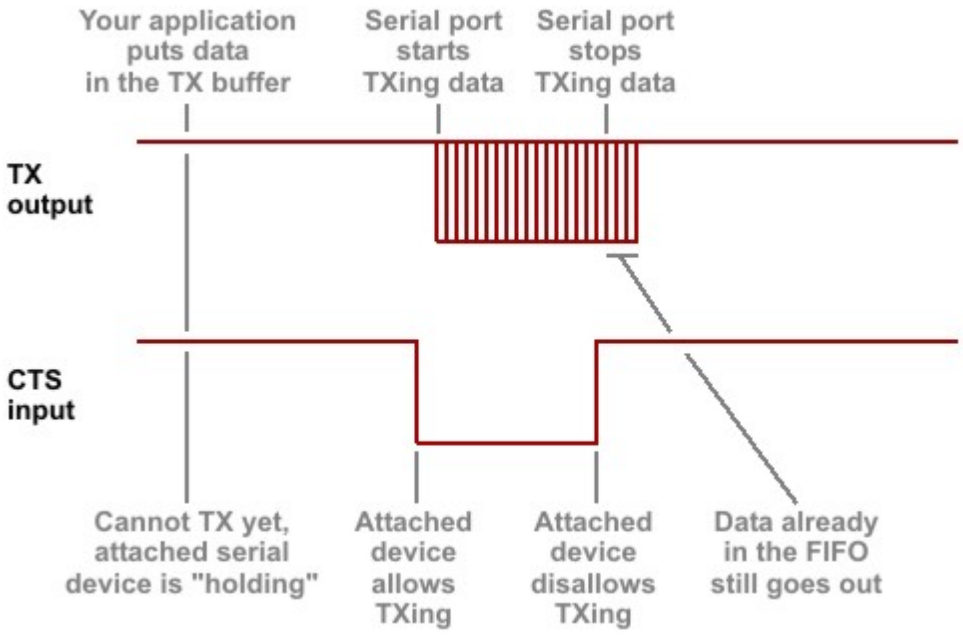
The full-duplex mode of operation is suitable for communicating with RS232, RS422 devices, 4-line RS485 devices, and most TTL-serial devices. Naturally, external transceiver IC is needed for RS232, RS4222, or RS485. TTL serial devices can be connected to the serial port lines directly in most cases.

In the full-duplex mode, the RTS output and CTS input lines can be used for optional flow control. The RTS output is low whenever the serial port is ready to receive the data from "attached" serial device (port is opened and the RX buffer has at least 64 bytes of free space). The RTS line is high whenever the serial port is closed or the RX buffer has less than 64 bytes of free space left. Figure below illustrates RTS operation.

All diagrams show TTL-serial signals. If you are dealing with the lines of the RS232 port you will see all signals in reverse!



The CTS input is used by the serial port to check if attached serial device is able to accept the data. The serial port will only start to send the data when the CTS input is low. The serial port will stop sending the data once the line goes high. Note, that some Tibbo devices have a hardware buffer called "FIFO[196]" ("fist-in-first-out" if you really need to know ;-). Once the TX data is in the FIFO it will be sent out even if the CTS line goes low. Therefore, after the attached serial device switches the CTS line to low the serial port may still output the number of bytes not exceeding the capacity of the FIFO. For more information, refer to your device's platform documentation (for example, EM1000's is here[143]).



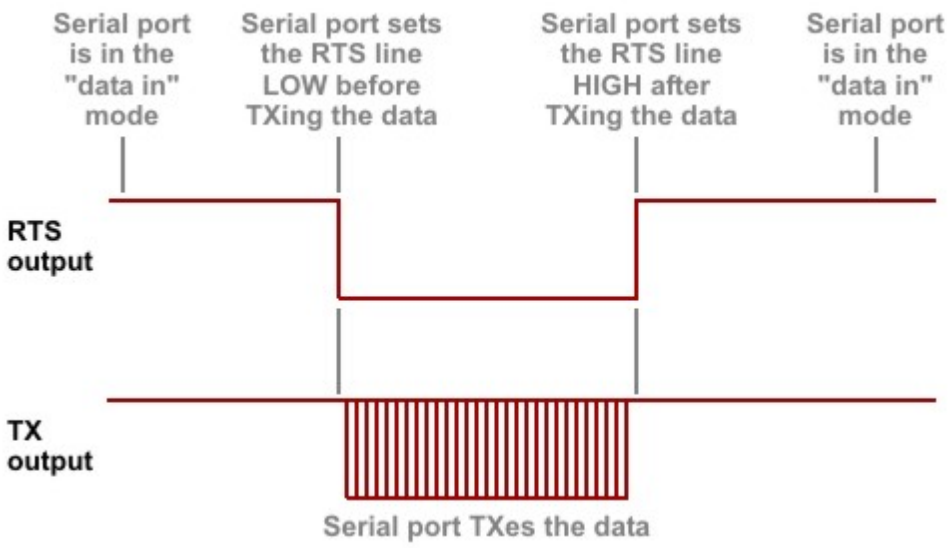
When the flow control is enabled, the RTS and CTS lines are controlled by the serial port and cannot be manipulated by your application. When the flow control is off, your application can set and read the state of these lines through the io 294 object.

Half-duplex operation

The half-duplex mode of operation is suitable for communicating with 2-wire RS485 devices. Again, appropriate interface transceiver IC is required to be connected to the serial port.

In the half-duplex mode the RTS output is used to control data transmission direction. You can select the polarity of the direction control signal, i.e. which state will serve as "data in" direction, and which- as "data out" direction (see Serial Settings 390).

When the serial port has no data to transmit (the TX buffer is empty), it is always ready to receive the data, so the RTS line is in the "data in" state. When the serial port needs to send out some data (the TX buffer is not empty) it switches the RTS line into the "data out" state, transmits the data, then switches the RTS back into the "data in" state. Assuming "LOWFORINPUT" direction control polarity, direction control looks like this:



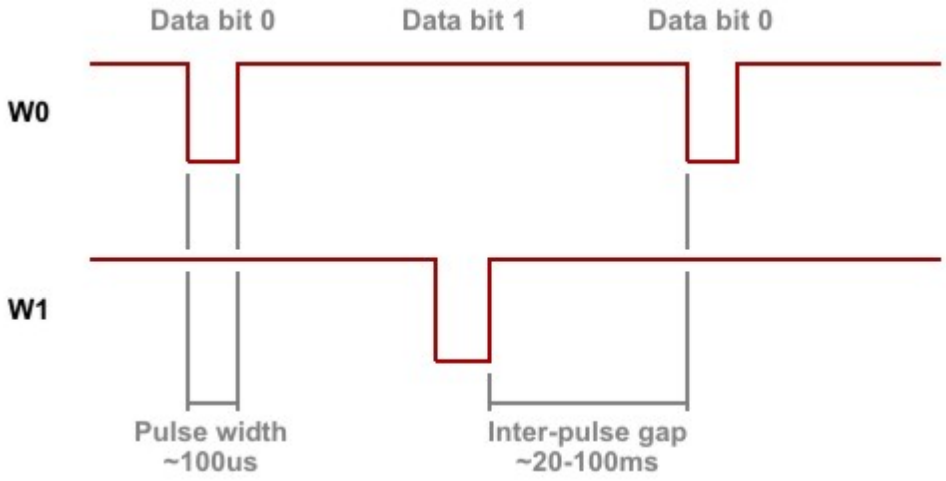
The CTS line is not controlled by the serial port when in the half-duplex mode. Your application can manipulate this line through the io 294 object.

Wiegand Mode

In the Wiegand mode the serial port is able to receive the data directly from any Wiegand device, such as card reader and also output the data in the Wiegand format, as if it was a card reader itself. Wiegand interface is popular in the security, access control, and automation industry.

Standard Wiegand data transmission is shown below. There are two data lines- W0 and W1. Negative pulse on the W0 line represents data bit 0. Negative pulse on the W1 line represents data bit 1. There is no standard Wiegand timing, so pulse widths as well as inter-gap widths wary greatly between devices. Averagely, pulse width is usually in the vicinity of 100uS (microseconds), while the inter-pulse gap is usually around 20-100ms (milliseconds).

There is no explicit way to indicate the end of transmission. Receiving device either counts received bits (if it knows how many to expect) or assumes the transmission to be over when the time since the last pulse on the W0 or W1 line exceeds certain threshold, for example, ten times the expected inter-pulse gap.



The serial port outputs Wiegand data through **W0out** and **W1out** lines and receives the data via **W0&1in** and **W1in** lines. "W0&1in" means that the signal on this input must be a logical AND of W0 and W1 output lines of attached Wiegand device (see below for details). Your application *should not* attempt to work with W0out and W1out outputs directly through the io 294 object when the serial port is in the Wiegand mode.

Please, remember that on your platform you may be required to correctly configure 194 some of your serial port's lines as inputs or outputs through the io.enabled 298 property of the io 294 object. Additionally, you may have the freedom of re-mapping certain serial port lines to different I/O pins of the device if required. For more information, refer to your device's platform documentation (for example, EM1000's is here 143).

How the serial port sends and receives raw Wiegand data

There are many Wiegand data formats currently in use. These formats define how "raw" data bits are processed and converted into actual data. Typically, there are 2 parity bits- one at the beginning, and another one at the end of Wiegand data. Parity calculation, however, varies from format to format. Additionally, the length of Wiegand output is not standardized.

All this makes it impossible for the serial port object to verify incoming Wiegand data, i.e. check the data length and calculate the checksum. Instead, this task is delegated to your application while the serial object only receives raw data. Similarly, before sending out Wiegand data your application needs to prepare this data in the desired format- the serial object itself will output any data stream.

How the Wiegand data is stored in the RX and TX buffers of the serial port

When in the Wiegand mode, each data byte in the TX or RX buffer of the serial port represents one bit of Wiegand data. This bit is recorded in the least significant bit position of each data byte in the buffer. For your application's convenience, when the serial port receives Wiegand bit stream, it adds an offset of 30Hex to each data bit. Therefore, the data recorded into the RX buffer can only consist of bytes 30H

and 31H. These correspond to ASCII characters '0' and '1'. This way, when your application reads RX buffer contents into a string variable the data will be "readable" without any additional conversion (ASCII characters with codes 0 and 1 would not be "readable").

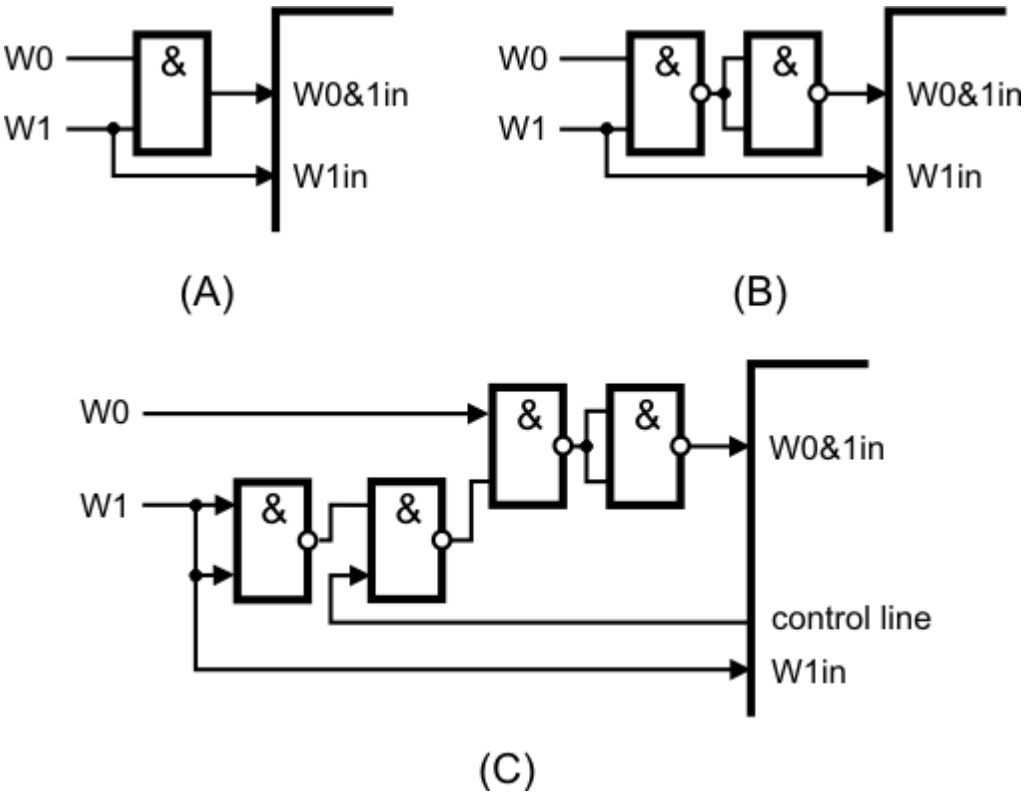
When the serial port outputs Wiegand data, it only takes bit 0 of each byte in the TX buffer. Other bits can contain any data. You can, for instance, put a string of ASCII characters '0' and '1' into the TX buffer and these will be correctly interpreted as data bits 0 and 1. This, again, is convenient for your BASIC application.

How the serial port transmits Wiegand data

Wiegand data output timing is fixed and your application cannot change it. Data pulses are 100uS wide and inter-pulse gaps are 2mS wide.

How the serial port receives Wiegand data

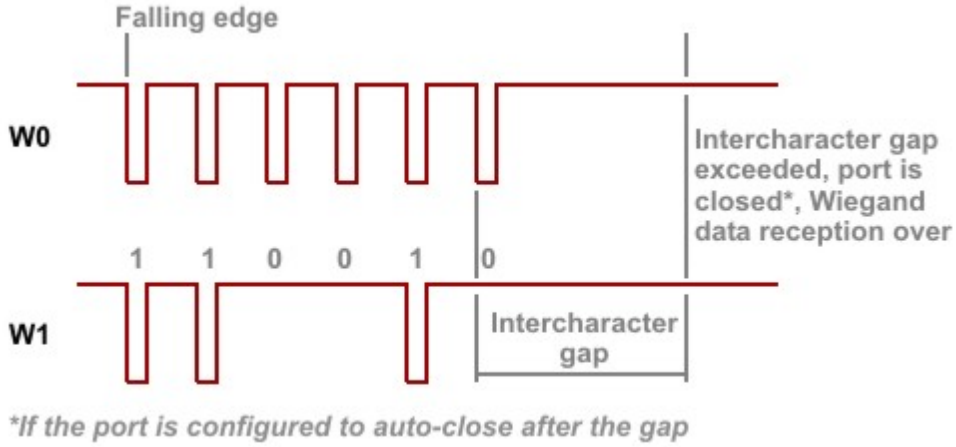
You already know that W0&1in input of the serial port must receive a logical AND of W0 and W1 output of attached Wiegand device. A simple AND gate will do the job (figure A below). Actually, NOR-AND gates are more popular and these can be used too (figure B). In case you are building a product that will also accept clock/data 386 input, you may need to control whether the W0&1in input should receive a logical AND of two lines, or just one of the lines. Schematic diagram C uses an additional I/O line of the device to control this. When the control line is HIGH the W0&1in input receives a logical AND of both W0 and W1 lines, when the control line is LOW, the W0&1in input receives just the signal from the W0 line. Four gates are required for this, so you will get away with using a single 74HC00 IC.



The serial port does not require an incoming Wiegand data stream to adhere to any strict timing. The port is simply registering high-to-low transitions on the W0&1in

line. When such transition is detected, the port checks the state of W1 line. If the line is HIGH, data bit 0 is registered, when the line is low, data bit 1 is registered.

The end of Wiegand transmission is identified by timeout- the serial port has a special property for that, called "intercharacter delay" (see Serial Settings 390). Another property- "auto-close"- can be used to disable the serial port after the delay has been encountered. This way, when the Wiegand output is over the port will be disabled and no further data will enter the port until you re-enable it.



Clock/Data Mode

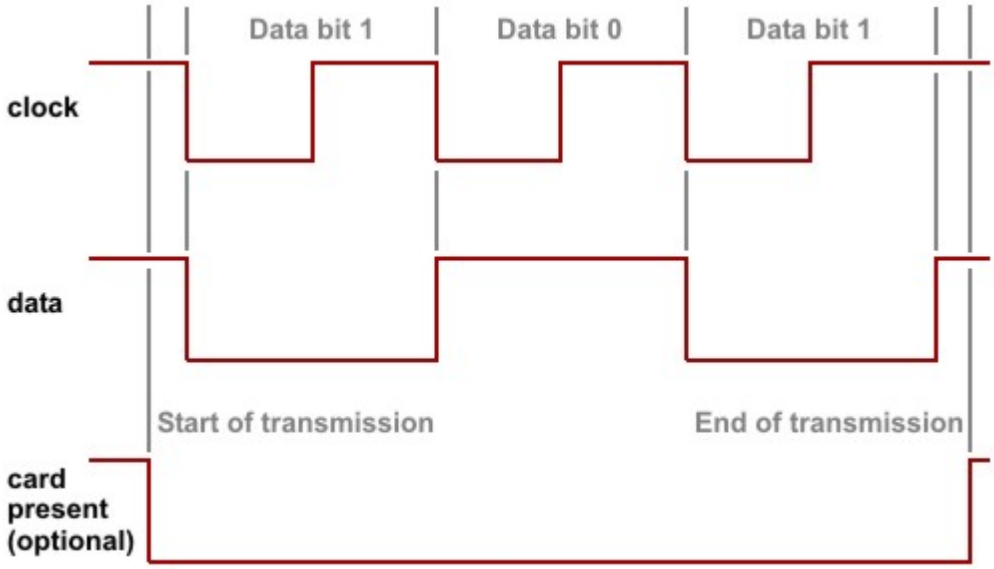
In the clock/data mode the serial port is able to receive the data directly from any clock/data or "magstripe interface" device, such as a card reader. The serial port is also able to output the data in the clock/data format, as if it was the card reader itself. Magstripe and clock/data interfaces are popular in the security, access control, automation, and banking industry.

Standard clock/data transmission is shown below. There are two data lines- clock and data. Each negative transition on the clock line marks the beginning of the data bit. The data line carries actual data. When the state of the data line is LOW it means data bit 1, and vise versa. There is no standard clock/data timing and some devices, such as non-motorized magnetic card readers, output the data at variable speeds (depending on how fast the user actually swipes the card).

The magstripe interface only differs from the clock/data interface in that it has a third line- card present. This line goes LOW before the data transmission and goes back to HIGH after the transmission is over. The serial port does not require the card present line for data reception. Just like with Wiegand data 383, it identifies the end of incoming data by measuring the time since the last negative transition on the clock line. For data transmission, your application can easily use any regular I/O line to serve as card present line.

Compared to Wiegand interface, the data format of clock/data interface is very

standardized and its varieties include standard data formats for different "tracks" of the magnetic card. Most clock/data devices you will actually encounter have nothing to do with magnetic cards but terminology persists.



The serial port outputs clock/data signals through **cout** and **dout** lines and receives the data via **cin** and **din** lines. Your application *should not* attempt to work with cout and dout outputs directly through the io 294 object when the serial port is in the clock/data mode.

Please, remember that on your platform you may be required to correctly configure 194 some of your serial port's lines as inputs or outputs through the io.enabled 298 property of the io 294 object. Additionally, you may have the freedom of re-mapping certain serial port lines to different I/O pins of the device if required. For more information, refer to your device's platform documentation (for example, EM1000's is here 143).

How the serial port sends and receives raw clock/data data

Clock/data from different "tracks" has different encoding. Encoding defines how "raw" data bits are processed and converted into actual data. To allow maximum flexibility, and also to maintain the data processing style used by the Wiegand interface 383, the serial port leaves the task of converting between the raw and actual data to your application. The serial port only sends and receives raw data without checking or transforming its contents.

How the clock/data stream is stored in the RX and TX buffers of the serial port

When in the clock/data mode, each data byte in the TX or RX buffer of the serial port represents one bit of the clock/data stream. This bit is recorded in the least significant bit position of each data byte in the buffer. For your application's convenience, when the serial port receives clock/data bit stream, it adds an offset of 30Hex to each data bit. Therefore, the data recorded into the RX buffer can only consist of bytes 30H and 31H. These correspond to ASCII characters '0' and '1'. This way, when your application reads RX buffer contents into a string variable the

data will be "readable" without any additional conversion (ASCII characters with codes 0 and 1 would not be "readable").

When the serial port outputs clock/data stream, it only takes bit 0 of each byte in the TX buffer. Other bits can contain any data. You can, for instance, put a string of ASCII characters '0' and '1' into the TX buffer and these will be correctly interpreted as data bits 0 and 1. This, again, is convenient for your BASIC application.

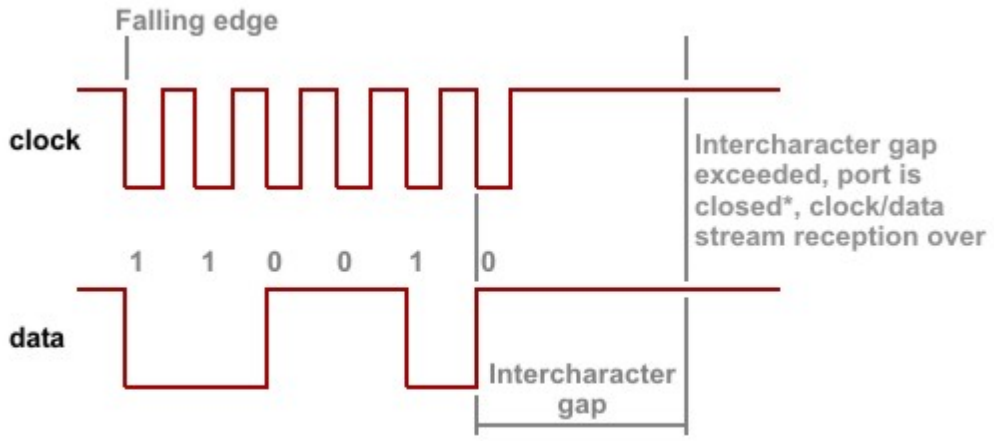
How the serial port transmits clock/data stream

Clock/data output timing is fixed and your application cannot change it. The data is output at a rate of 400us/bit (both LOW and HIGH phases of the clock signal are 200us in length).

How the serial port receives clock/data stream

The serial port does not require an incoming clock/data stream to adhere to any strict timing. The port is simply registering high-to-low transitions on the clock line. When such transition is detected, the port checks the state of the data line. If the line is HIGH, data bit 0 is registered, when the line is low, data bit 1 is registered.

The end of clock/data transmission is identified by timeout- the serial port has a special property for that, called "intercharacter delay" (see Serial Settings 390). Another property- "auto-close"- can be used to disable the serial port after the delay has been encountered. This way, when the clock/data output is over the port will be disabled and no further data will enter the port until you re-enable it.



If the port is configured to auto-close after the gap

Port Selection

There may be platforms with more than one serial port. You can obtain the number of serial ports available for your platform using the ser.numofports 411 property.

Since there can be multiple ports, you must state which port you are referring to when changing properties or invoking methods. This is done using the ser.num 411 property. For example:

```
ser.mode = PL_SER_MODE_WIEGAND
```

Can you tell what serial port the statement above applies to? Neither can the platform. Thus, the correct syntax would be:

```
ser.num = 0
ser.mode = PL_SER_MODE_WIEGAND
```

Now the platform knows what port you're working with. Once you have set the *port selector* (using ser.num), every method and property after that point is taken to refer to that port. Thus:

```
ser.num = 0
ser.enabled = 1
ser.baudrate = 1
ser.bits = 1 ' etc
```

The events generated by the ser object are not separate for each port. An event such as on_ser_data_arrival 412 serves all serial ports on your platform. Thus, when an event handler for the serial port object is entered, the port selector is automatically switched to the port number on which the event has occurred:

```
sub on_ser_data_arrival

     dim s as string
     s = ser.getdata(255) ' Note that you did not have use ser.num before
this statement.

end sub
```

As a result of this automatic switching, when an event handler for a serial port event terminates, the ser.num property retains its new value (nothing changes it back). You must take this into account when processing other event handlers which make use of the serial port (and are not serial port events). In other words, you should explicitly set the ser.num property whenever entering such an event handler, because the property might have been automatically changed prior to this event. To illustrate:

```
sub on_sys_init ' This is always the first event executed.

ser.num = 0  ' Supposedly, this would make all subsequent properties and
methods refer to this port.

end sub


sub on_ser_data_arrival ' Then, supposing this event executes.

dim s as string
s = ser.getdata(255) ' However, this event happens on the second port. So
```

```
now ser.num = 1.

end sub


sub on_sock_data_arrival ' And then this socket event executes.

ser.txclear ' You meant to do this for ser.num = 0 (as specified at
on_sys_init). But now port.num was changed to 1. You did not explicitly
specify a ser.num here, and now the tx buffer for the wrong port is cleared.
Oops.

end sub
```

Same precautions should be taken when using doevents 73. This is because doevents will let other events execute and so serial object events will potentially execute and cause the ser.num to change.

To recap, only one of two things may change the current ser.num: **(1)** Manual change or **(2)** a serial port event. And you cannot assume the number has remained unchanged if you set it somewhere else (because a serial port event might have happened since).

Specifying a port number for a single-port platform may seem redundant, but it makes your program portable. You will have an easier time migrating your program to a multi-port platform in the future.

Serial Settings

This topic briefly outlines the range of configurable options available on the serial port (this does not include the bulk of data on RX and TX buffers, which are described here 393).

Opening and closing the serial port

The ser.enabled 405 property defines whether the port is opened or closed. The port is closed by default, so you need to open it explicitly. The serial port won't receive or transmit the data when it is closed, but you will still be able to access its RX and TX buffers even at that time.

Serial port lines remapping

Depending on your platform, you may or may not be allowed to remap 195 **RTS/W0out/cout** output and **CTS/W0&1in/cin** input, i.e. choose what I/O pins of the device these lines should be on. For more information, refer to your device's platform documentation (for example, EM1000's is here 143). If your device supports remapping, you can use ser.ctsmap 403 and ser.rtsmap 415 properties to select required mapping. If your device does not support remapping then its platform will not have these properties. You can only perform remapping when the serial port is closed.

UART, Wiegand, or clock/data mode selection

The ser.mode 409 property selects UART 380, Wiegand 383, or clock/data 386 mode for the serial port. You can only change the mode when the serial port is closed.

How the incoming data is committed

One important concept you will need to understand is "RX data committing". You already know that the serial port records all incoming data into the RX buffer. New information is that the data recorded into the buffer is not immediately "reported" to your application. Instead, this buffer remains "uncommitted" until certain conditions are met. Uncommitted data is effectively invisible to your application, as if it is not there at all.

For UART 380 mode, the data is committed either when the total amount of uncommitted and committed data in the RX buffer exceeds 1/2 of this buffer's capacity or when the intercharacter gap, defined by the ser.interchardelay 408 property is exceeded. For Wiegand 383 and clock/data 386 modes, the first condition is not monitored, so only the intercharacter gap can commit the data.

The intercharacter gap is the time elapsed since the start of the most recent UART character reception in the UART mode or the most recent falling edge on the W0&1in/cin line in the Wiegand and clock/data modes. The idea is that once the data stops coming in, the serial port starts counting the delay. Once the delay exceeds the time set by the ser.interchardelay property, the data is committed and becomes visible to your application.

Another property -- ser.autoclose 402 -- defines whether the port will be closed (ser.enabled 405 = NO) once the intercharacter gap reaches ser.interchardelay value.

For the UART mode, the intercharacter delay allows your application to process the data more efficiently. By keeping the data invisible for a while the serial port can accumulate a large chunk of data that your application will be able to process at once. Imagine, for instance, that the data is flowing into the serial port character by character and your application has to also process this stream character by character. The overhead may be significant and overall performance of your application greatly reduced! Now, if this incoming data is combined into sizeable portions your application won't have to handle it in small chunks, and this will improve performance. Of course, you need to strike a balance here -- attempting to combine the data into blocks that are too large may reduce your application's responsiveness and make your program appear sluggish.

Notice, that the intercharacter gap is not counted when the new data is not being received because the serial port has set the RTS line to LOW (not ready). This could happen when flow control is enabled (more on flow control below).

For the Wiegand and clock/data modes, the intercharacter delay is the way to detect the end of incoming data stream. You are recommended to program the gap to about 10 times the data rate. For example, if you are receiving the Wiegand data at a rate of 1 bit per 20ms, then set the delay to 200 ms and it will serve as a reliable indicator of transmission end. This is when the ser.autoclose will come handy- once the gap is detected the port will be closed and this will prevent another Wiegand transmission from entering the RX buffer before your application processes the previous one.

How the outgoing data is committed

Outgoing data uses similar "data committing" concept as the incoming data. The objective is to be able to commit the data for sending once, even if the data was prepared bit by bit. This way your application can avoid sending out data in small chunks. Buffer Memory Status 394 topic details this.

UART mode settings

Several settings are unique to UART 380 mode. The serial port has all the usual

UART-related communication parameters: baudrate 402, parity 413, 7/8 bits 403. There is no property to select the number of stop bits. Second stop bit can be emulated by setting ser.parity= 3- PL_SER_PR_MARK.

The baudrate property actually keeps a divisor value. You can set any baudrate you want by providing the correct divisor. There is even a read-only ser.div9600 404 property that allows you to calculate the divisor value in a platform-independent way, by returning the required divisor value (for the current platform) to reach 9600bps.

The actual baudrate is calculated as follows: baudrate=(9600*ser.div9600)/ ser.baudrate. For example, if we need to achieve 38400bps and ser.div9600 returns 12, then we need to set ser.baudrate=3, because (9600*12)/38400=3. This serves as a platform-independent baudrate calculation, as ser.div9600 will return different values for different platforms.

For example:

```
function set_baud(baud as integer) as integer
'baud: 0- 1200, 1-2400, 2-4800, 3- 9600, 4-19200, other-38400

select case baud
      case 0: ser.baudrate=ser.div9600*8 '9600/1200=8
      case 1: ser.baudrate=ser.div9600*4 '9600/2400=4
      case 2: ser.baudrate=ser.div9600*2 '9600/4800=2
      case 3: ser.baudrate=ser.div9600   '9600/9600=1
      case 4: ser.baudrate=ser.div9600/2 '19200/9600=2
      case else: ser.baudrate=ser.div9600/4 '38400/9600=4
end function
```

The serial port can be used in full-duplex or half-duplex mode, as determined by the ser.interface 408 property (see UART Mode 380 for details). In the full-duplex mode, the serial port can optionally use the flow control 407. in the half-duplex mode, you can select the polarity 404 of the direction control signal.

In the UART mode, the serial port can recognize so-called 'escape sequences'. They are defined using the ser.esctype 405 and ser.escchar 405 properties.

When such a sequence is recognized, a special event (on_ser_esc 413) is generated and the port is disabled. Of course this can be achieved programmatically, but it would require you to parse all incoming data and really slow things down. Escape sequences were implemented with efficiency and speed in mind.

Escape sequences are rather arbitrary. They follow the style of escape sequences that Tibbo introduced before. However, they are useful for certain things. For example , if your application has a normal mode and serial 'setup' mode, you can use this sequence to switch into setup mode.

Escape sequence will work even if you are using buffer shorting 399, and that makes them especially powerful. If you are building a device which just routes the data between a serial interface and an Ethernet interface, you will use buffer shorting for performance, but you could still detect the escape sequences to switch into the serial programming mode or perform some other similar task.

Sending and Receiving Data (TX and RX buffers)

The serial port sends and receives data through TX (transmit) and RX (receive) buffers. Read on and you will know how to allocate memory for buffers, use them, handle overruns, and perform other tasks related to sending and receiving of data.

Allocating Memory for Buffers

Each buffer has a certain size, i.e, a memory capacity. This capacity is allocated upon request from your program. When the device initially boots, no memory is allocated to buffers at all.

Memory for buffers is allocated in pages. A *page* is 256 bytes of memory. Allocating memory for a buffer is a two-step process: First you have to request for a specific allocation (a number of pages) and then you have to perform the actual allocation. For the RX buffer, request memory using ser.rxbuffrq 415, and for the TX buffer, request it using ser.txbuffrq 419.

The allocation method (sys.buffalloc 530) applies to all buffers previously specified, in one fell swoop. Hence:

```
dim in, out as byte
out = ser.txbuffrq(10) ' Requesting 10 pages for the TX buffer. Out will
then contain how many can actually be allocated.

in = ser.rxbuffrq(7) ' Requesting 7 pages for the RX buffer. Will return
number of pages which can actually be allocated.

' ....  Allocation requests for buffers of other objects ....

sys.buffalloc ' Performs actual memory allocation, as per previous requests.
```

Actual memory allocation takes up to 100ms, so it is usually done just once, on boot, for all required buffers. If you do not require some buffer, you may choose not to allocate any memory to it. In effect, it will be disabled.

You may not always get the full amount of memory you have requested. Memory is not an infinite resource, and if you have already requested (and received) allocations for 95% of the memory for your platform, your next request will get up to 5% of memory, even if you requested for 10%.

There is a small overhead for each buffer. Meaning, not 100% of the memory allocated to a buffer is actually available for use. 16 bytes of each buffer are reserved for variables needed to administer this buffer, such as various pointers etc.

Thus, if we requested (and received) a buffer with 2 pages (256 * 2 = 512), we actually have 496 bytes in which to queue data (512 - 16).

- You can only change the size of buffers that belong to serial ports that are closed (ser.enabled 405 = 0) at the moment sys.buffalloc methods executes. If the port is opened at the time the sys.buffalloc method executes then buffer capacities for this port will remain unchanged, even if you requested changes through ser.rxbuffrq and ser.txbuffrq.

Using Buffers

Once you have allocated memory for the TX and RX buffers you can start sending and receiving data through them.

Sending Data (through TX buffer)

Sending data is a two-step process. First, you put the data in the TX buffer using the ser.setdata 418 method, and then you perform the actual sending (commit the data) using the ser.send 417 method. For example:

```
ser.setdata ("Foo") ' Placed our data in the TX buffer - not being sent out
yet.

' ... more code...

ser.setdata ("Bar") ' Added even more data to our buffer, waiting to be
sent.

ser.send ' Now data will actually start going out. Data sent will be
'FooBar'.
```

Since this is a two-step process, you may first prepare a large block of data in the TX buffer and only then commit this data (this may come handy in some applications).

TiOS features *non-blocking operation*. This means that on ser.send, for example, the program does not halt and wait until the data is completely sent. In fact, execution resumes immediately, even before the first byte goes out. Your program will not freeze just because you ordered it to send a large chunk of data.

Receiving Data (through RX buffer)

Receiving data is a one-step process. To extract the data from a buffer, use the ser.getdata 407 method. Data may only be extracted once from a buffer. Once extracted, it is no longer in the buffer. For example:

```
dim whatigot as string
whatigot = ser.getdata(255)
```

The variable whatigot now contains up to 255 bytes of data which came from the TX buffer of the serial port.

Buffer Memory Status

You cannot effectively use a buffer without knowing what its status is. Is it overflowing? Can you add more data? etc. Thus, each of the serial buffers has certain properties which allow you to monitor it:

The RX buffer

You can check the total capacity of the buffer with the ser.rxbuffsize 416 property. You can also find out how much *committed* data the RX buffer currently contains with the ser.rxlen 417 property (see Serial Settings 390 for explanation of what committed data is).

Sometimes you need to clear the RX buffer without actually extracting the data. In such cases the ser.rxclear 416 comes in handy.

The TX buffer

Similarly to the RX buffer, the TX buffer also has a ser.txbuffsize 419 property which lets you discover its capacity.

Unlike the RX buffer, the TX buffer has two "data length" properties: ser.txlen 420 and ser.newtxlen 410. The *txlen* property returns the amount of *committed* data waiting to be sent from the buffer (you commit the data by using the ser.send 417 method). The *new*txlen property returns the amount of data which has entered the buffer, but has not yet been committed for sending.

The TX buffer also has a ser.txfree 420 property, which directly tells you how much space is left in it. This does not take into account uncommitted data in the buffer -- actual free space is ser.txfree-ser.newtxlen!

ser.txlen + ser.txfree = ser.txbuffsize.

When you want to clear the TX buffer without sending anything, use the ser.txclear 420 method.

An example illustrating the difference between ser.txlen and ser.newtxlen:

```
sub on_sys_init

dim x,y as word ' declare variables

ser.rxbuffrq(1) ' Request one page for the rx buffer.
ser.txbuffrq(5) ' Request 5 pages for the tx buffer (which we will use).
sys.buffalloc ' Actually allocate the buffers.

ser.setdata("foofoo") ' Set some data to send.
ser.setdata("bar") ' Some more data to send.
ser.send ' Start sending the data (commit).
ser.setdata("baz") ' Some more data to send.
x = ser.txlen ' Check total amount of data in the tx buffer.
y = ser.newtxlen  ' Check length of data not yet committed. Should be 3.

end sub 'Set up a breakpoint HERE.
```

Don't step through the code. The sending is fast -- by the time you reach x and y by stepping one line at a time, the buffer will be empty and x and y will be 0. Set a breakpoint at the end of the code, and then check the values for the variables (by using the watch 33).

Receiving Data

In a typical system, there is a constant need to handle an inflow of data. A simple approach is to use polling. You just poll the buffer in a loop and see if it contains any data, and when fresh data is available, you do something with it. This would look like this:

```
sub on_sys_init

while ser.rxlen = 0
wend ' basically keeps executing again and again as long as ser.rxlen = 0
s = ser.getdata(255) ' once loop is exited, it means data has arrived. We
extract it.

end sub
```

This approach will work, but it will forever keep you in a specific event handler (such as on_sys_init 533) and other events will never get a chance to execute. This is an example of *blocking code* which could cause a system to freeze. Of course, you can use the doevents 87 statement, but generally we recommend you to avoid this approach.

Since our platform is event-driven, you should use events to tell you when new data is available. There is an on_ser_data_arrival 412 event which is generated whenever there is data in the rx buffer:

```
sub on_ser_data_arrival

dim s as string
s = ser.getdata(255) ' Extract the data -- but in a non-blocking way.
' .... code to process data ....
end sub
```

This on_ser_data_arrival 412 event is generated whenever there is data in the RX buffer, but only once. There are never two on_ser_data_arrival events waiting in the queue. The next event is only generated after the previous one has completed processing, if and when there is any data available in the RX buffer.

This means that when handling this event, you don't have to get *all* the data in the RX buffer. You can simply handle a chunk of data and once you leave the event handler, a new event of the same type will be generated if there is still unprocessed data left.

Here is a correct example of handling arriving serial data through the on_ser_data_arrival event. This example implements a data loopback -- whatever is received by the serial port is immediately sent back out.

```
sub on_ser_data_arrival
      ser.setdata(ser.getdata(ser.txfree))
      ser.send
end sub
```

We want to handle this loopback as efficiently as possible, but we must not overrun the TX buffer. Therefore, we cannot simply copy all arriving data from the RX buffer into the TX buffer. We need to check how much free space is available in the TX buffer. The first line of this code implements just that: Ser.getdata 407 method takes as much data from the RX buffer as possible, but not more than ser.txfree 420 (the available room in the TX buffer). The second line just sends the data.

Actually, this call will handle no more than 255 bytes in one pass. Even though we seemingly copy the data directly from the RX buffer to the TX buffer, this is done via a temporary string variable automatically created for this purpose. In this platform, string variables cannot exceed 255 bytes.

Polling method of data processing can sometimes be useful. See Generating Dynamic HTML Pages 466.

Sending Data

In the previous section, we explained how to handle an incoming stream of data. You could say it was incoming-data driven. Sometimes you need just the opposite -- you need to perform operations based on the sending of data.

For example, supposing that in a certain system, you need to send out a long string of data when a button is pressed. A simple code for this would look like this:

```
sub on_button_pressed
      ser.setdata("This is a long string waiting to be sent. Send me
already!")
      ser.send
end sub
```

The code above *would* work, but *only* if at the moment of code execution the necessary amount of free space was available in the TX buffer (otherwise the data would get truncated). So, obviously, you need to make sure that the TX buffer has the necessary amount of free space before sending. A simple polling solution would look like this:

```
sub on_button_pressed
      dim s as string
      s = "This is a long string waiting to be sent. Send me already!"
      while ser.txfree < len(s) 'we will wait for the necessary amount of
free space to become available
      wend
      ser.setdata(s)
      ser.send
end sub
```

Again, this is not so good, as it would block other event handlers. So, instead of doing that, we would employ a code that uses on_ser_data_sent 412:

```
dim s as string
s = "This is a long string waiting to be sent. Send me already!"

sub on_button_pressed
      ser.notifysent(ser.txbuffsize-len(s)) ' causes the on_ser_data_sent
event to fire when the tx buffer has space for our string
end sub

sub on_ser_data_sent
      ser.setdata(s)  ' put data in tx buffer
      ser.send  ' start sending it.
end sub
```

When we press the button, on_button_pressed 234 event is generated, so now the system knows we have a string to send. Using ser.notifysent 410 we make the system fire the on_ser_data_sent 412 event when the necessary amount of free space becomes available. This event will only be fired once -- and will be fired immediately if there is already enough available space.

Within the on_ser_data_sent event handler we put the data in the TX buffer and start sending it.

- Amount of data that will trigger on_ser_data_sent does not include uncommitted data in the TX buffer.

Handling Buffer Overruns

Handling RX buffer overruns

The on_ser_overrun event 413 is generated when an RX buffer overrun has occurred. It means the data has been arriving to the UART faster than you were handling it and that some data got lost.

This event is generated just once, no matter how much data is lost. A new event will be generated only after exiting the handler for the previous one. In the UART/ full-duplex mode (see UART Mode 380 for details) you can typically prevent this from happening by using the flow control (ser.flowcontrol 407).

Typically, the user of your system wants to know when an overrun has occurred. For example, you could blink a red LED when this happens.

```
sub on_ser_overrun
      pat.play("R-R-R-R")
end sub
```

Are TX buffer overruns possible?

TX buffer overruns are not possible. The serial port won't let you overload its TX buffer. If you try to add more data to the TX buffer than the free space in the buffer allows to store then the data you are adding will be truncated.

See Sending Data 397 for explanation on how to TX data correctly.

Redirecting Buffers

The following example appeared under Receiving Data 396:

```
sub on_ser_data_arrival
      ser.setdata(ser.getdata(ser.txfree))
      ser.send
end sub
```

This example shows how to send all data incoming to the RX buffer out from the TX buffer, in just two lines of code. However fast, this technique still passes all data through your BASIC code, even though you are not processing (altering, sampling) it in any way.

A much more efficient and advanced way to do this would be using a technique called *buffer redirection* (buffer shorting). With buffer shorting, instead of receiving the data into the RX buffer of your serial port, you are receiving it directly into the TX buffer of another object which is supposed to send out this data. This can be a serial object (the same port or a different one), a socket object, etc.

To use buffer shorting, you invoke the ser.redir 414 method and specify the buffer to which the data is to be redirected. Once this is done, the on_ser_data_arrival event won't be generated at all, because the data will be going directly to the TX buffer that you have specified. As soon as the data enters this buffer, it will be automatically committed for sending.

The ser.redir method will only work if the serial port is closed (ser.enabled 405 = 0-NO) at the time when this method is executed. Therefore, it makes sense to check the result of ser.enabled execution, as in the example below:

```
sub on_sys_init

      if ser.redir(PL_REDIR_SER)= PL_REDIR_SER then
            'redirection succeeded
      else
            'redirection failed (perhaps, the port is opened?)
      end if
end sub
```

The performance advantage of buffer shorting is enormous, due to two factors: first, you are not handling the data programmatically, so the VM isn't involved at all. And second, the data being received is received directly into the TX buffer from which it is transmitted, so there is less copying in memory.

Of course you cannot do anything at all with this data -- you are just pumping it through. However, very often this is precisely what is needed! Additionally, you *can* still catch escape sequences 405.

To stop redirection, you can use ser.redir(0), which means "receive data into the RX buffer in a normal fashion".

Sinking Data

Sometimes it is desirable to ignore all incoming data while still maintaining the serial port opened. The ser.sinkdata 418 property allows you to do just that.

Set the ser.sinkdata to 1- YES, and all incoming data will be automatically

discarded. This means that the on_ser_data_arrival 412 event will not be generated, reading ser.rxlen 417 will always be returning zero, and so on. No data will be reaching its destination even in case of buffer redirection 399. Escape characters 390, however, will still be detected in the incoming data stream.

Properties, Methods, Events

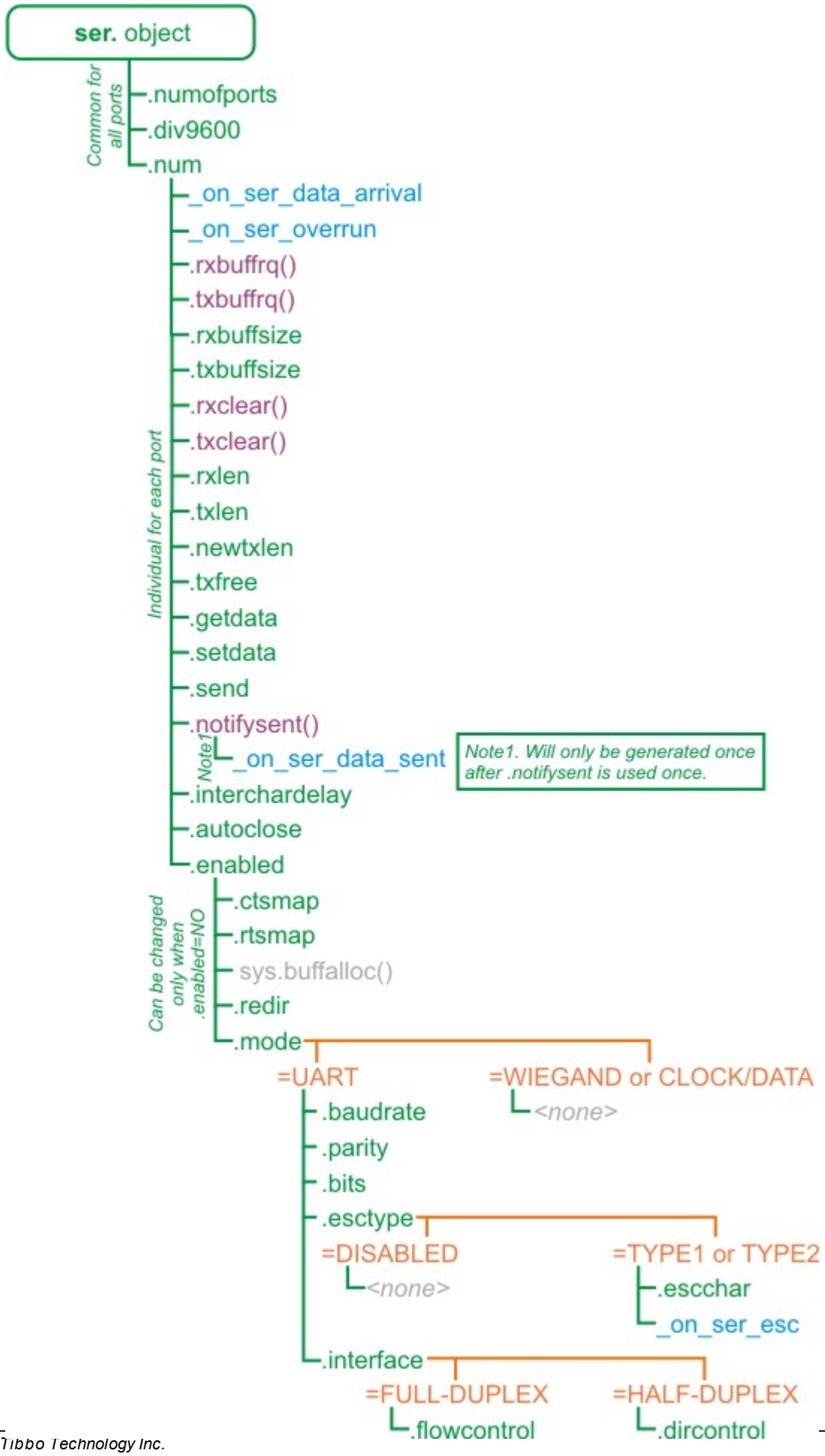
This section provides an alphabetical list of all properties, methods, and events of the ser object. For your convenience, here is a hierarchical map of the serial port's events, properties and methods.

All properties, methods, and events under ser.num 411 are shown as sub-nodes of this property because they refer to a serial port currently selected by the ser.num. The _on_ser_data_sent 412 event is subordinate to the set.notifysent 410 method because this method needs to be called each time you want to receive the _on_ser_data_sent.

The ser.ctsmap 403, ser.rtsmap 415, sys.buffalloc 530, ser.redir 414, and ser.mode 409 are subordinate to ser.enabled because they can be changed (or have effect) only when ser.enabled= 0- NO (the sys.buffalloc method is not a part of the ser object but its use is required for normal serial port operation- this is why it is listed here).

Ser.baudrate 402, ser.parity 413, ser.bits 403, ser.esctype 405, and ser.interface 408 are only relevant in the UART mode of operation.

Ser.escchar 405 is only relevant when ser.esctype is not DISABLED.



.Autoclose Property

Function: For currently selected serial port (selection is made through ser.num 411) specifies whether the port will be disabled once the intercharacter gap expires.

Type: dis_en (enum, byte)

Value Range: 0- DISABLED **(default)**
1- ENABLED

See Also: Serial Settings 390

Details

The serial port is disabled by setting ser.enabled 405 = 0- NO. Intercharacter gap duration is specified by the ser.interchardelay 408 property.

This property offers a way to make sure that no further data is received once the gap of certain length is encountered. This property is especially useful in Wiegand 383 or clock/data 386 mode (ser.mode 409 = 1- PL_SER_MODE_WIEGAND or 2- PL_SER_MODE_CLOCKDATA) where intercharacter gap is the only way to reliably identify the end of one data transmission.

.Baudrate Property

Function: Sets/returns the baudrate "divisor value" for the selected serial port (selection is made through ser.num 411).

Type: Word

Value Range: 0-65535, default value is platform dependent, results in **9600bps**.

See Also: UART Mode 380, Serial Settings 390

Details

Actual baudrade is calculated as follows: (9600*ser.div9600 404)/ser.baudrate. The ser.div9600 read-only property returns the value ser.baudrate must be set to in order to obtain 9600 bps on a particular platform.

For example, if we need to achieve 38400bps and ser.div9600 returns 12, then we need to set ser.baudrate=3, because (9600*12)/38400=3. This serves as a platform-independent baudrate calculation, as ser.div9600 will return different values for different platforms.

This property is only relevant when the serial port is in the UART mode (ser.mode 409 = 0- PL_SER_MODE_UART).

Technically speaking, this property should be called divisor, not baudrate. We called it *baudrate* so that you could easily find it.

.Bits Property

Function: Specifies the number of data bits in a word TXed/RXed by the serial port for the currently selected port (selection is made through ser.num 411)

Type: Enum (pl_ser_bits, byte)

Value Range: 0- PL_SER_BB_7: data word TXed/RXed by the serial port is to contain 7 data bits

1- PL_SER_BB_8 **(default)**: data word TXed/RXed by the serial port is to contain 8 data bits.

See Also: UART Mode 380, Serial Settings 390

Details

This property is only relevant when the serial port is in the UART mode (ser.mode 409 = 0- PL_SER_MODE_UART).

.Ctsmap property (Selected Platforms Only)

Function: Sets/returns the number of the I/O line that will act as **CTS/W0&1in/cin** input of currently selected serial port (selection is made through ser.num 411).

Type: Enum (pl_int_num, byte)

Value Range: Platform-specific, see the list of pl_int_num constants in the platform specifications.

See Also: Three modes of the Serial Port 380, Serial Settings 390

Details

This property is only available on selected platforms. For more information please refer to your device's platform documentation (for example, EM1000's is here 143).

Default value of this property is different for each serial port. See the list of pl_int_num constants in the platform specifications -- it shows default values as well.

Selection can be made only among interrupt lines. Regular, non-interrupt I/O lines cannot be selected. Property value can only be changed when the port is closed (ser.enabled 405=0- NO).

On certain platforms, you may need to configure 194 the line as input. This is done through the io.enabled 298 property of the io 294 object.

.Dircontrol Property

Function:	Sets/returns the polarity of the direction control line (RTS) for selected serial port (selection is made through ser.num 411).
Type:	Enum (pl_ser_dircontrol, byte)
Value Range:	0- PL_SER_DCP_LOWFORINPUT **(default)**: The RTS output will be LOW when the serial port is ready to RX data and HIGH when the serial port is TXing data. 1- PL_SER_SI_HIGHFORINPUT: The RTS output will be HIGH when the serial port is ready to rx data and LOW when the serial port is txing.
See Also:	UART Mode 380, Serial Settings 390

Details

This property is only relevant in the UART/half-duplex mode (ser.mode 409 = 0- PL_SER_MODE_UART and ser.interface 408 = 1- PL_SER_SI_HALFDUPLEX).

Note, that HIGH/LOW states specified above are for the TTL-serial interface of the MODULE-level products. If you are dealing with the RS232 port then the states will be in reverse (for example, 1- PL_SER_SI_HIGHFORINPUT will mean "RTS LOW for input, HIGH for output").

Depending on your platform, you may be allowed to remap 195 RTS line to other I/O pins of the device through the ser.rtsmap 415 and ser.ctsmap 403 properties. Also, you may be required to correctly configure 194 RTS line as an input through the io.enabled 298 property of the io 294 object. For more information, refer to your device's platform documentation (for example, EM1000's is here 143).

When the serial port is in the UART/half-duplex mode you can use the CTS line as a regular I/O line of your device.

.Div9600 R/O Property

Function:	Returns the value to which the ser.baudrate 402 property must be set in order to achieve the baudrate of 9600bps on the current device and under present conditions.
Type:	Word
Value Range:	---
See Also:	Serial Settings 390

Details

"Smart" applications will use this property to set baudrates in a platform-independent fashion. Even for the same device, the value required to achieve 9600bps may be different at different times. For example, some devices have PLLs (see sys.currentpll 531). Enabling and disabling PLL changes the clock frequency of the device and this affects the value returned by ser.div9600.

.Enabled Property

Function:	Enables/disables currently selected serial port (selection is made through ser.num 411).
Type:	Enum (no_yes, byte)
Value Range:	0- NO **(default)**: not enabled 1- YES: enabled
See Also:	Buffer Memory Status 394

Details

Enabling/disabling the serial port does not automatically clear its buffers, this is done via ser.rxclear 416 and ser.txclear 420. Notice that certain properties can only be changed and methods executed when the port is not enabled. These are ser.rtsmap 415, ser.ctsmap 403, ser.mode 409, ser.redir 414. You also cannot allocate buffer memory for the port (do sys.buffalloc 530) when the port is enabled.

.Escchar Property

Function:	For selected serial port (selection is made through ser.num 411) sets/retrieves ASCII code of the escape character used for type1 or type2 serial escape sequences.
Type:	Byte
Value Range:	0-255, **default=1** (SOH character)
See Also:	UART Mode 380, Serial Settings 390

Details

Which escape sequence is enabled is defined by the ser.esctype 405 property. This property is irrelevant when ser.esctype= 0- PL_SER_ET_DISABLED or when the serial port is in the Wiegand 383 or clock/data 386 mode (ser.mode= 1- PL_SER_MODE_WIEGAND or ser.mode= 2- PL_SER_MODE_CLOCKDATA) -- serial escape sequences are only recognized in the UART data.

.Esctype Property

Function:	Defines, for selected serial port (selection is made through ser.num 411), whether serial escape sequence recognition is enabled and, if yes, what type of escape sequence is to be recognised.
Type:	Enum (pl_ser_esctype, byte)
Value Range:	0- PL_SER_ET_DISABLED **(disabled)**: Recognition of serial escape sequences disabled.

1- PL_SER_ET_TYPE1: Escape sequences of type 1 are to be recognized.

2- PL_SER_ET_TYPE2: Escape sequences of type 2 are to be recognized.

See Also: UART Mode 380, Serial Settings 390

Details

Escape sequence is a special occurrence of characters in the incoming data received by the serial port. Escape sequences are only recognized in the UART 380 mode of operation (ser.mode 409 = 0- PL_SER_MODE_UART).

When escape sequence is detected the on_ser_esc 413 event is generated and the serial port is disabled (ser.enabled 405 = 0- NO). When enabled, serial escape sequence detection works even when the buffer shorting 399 is employed (see ser.redir 414 property).

Follows is the description of two escape sequence types:

Type 1

Type1 escape sequence consists of three consecutive escape characters (ASCII code of escape character is defined by the ser.escchar 405 property). For the escape sequence to be recognized each of the escape characters must be preceded by a time gap of at least 100ms:

...previous data <--100ms--> **E.C.** <--100ms--> **E.C.** <--100ms--> **E.C.**

If the time gap before a certain escape character exceeds 100ms then this character is considered to be a part of the escape sequence and is not recorded into the RX buffer. If the time gap before a certain escape character is less than 100ms than this character is considered to be a normal data character and is saved into the RX buffer. Additionally, escape character counter is reset and the escape sequence must be started again. The following example illustrates one important point (escape characters are shown as ■). Supposing the serial port receives the following string:

ABC<--100ms-->■ <--100ms-->■ ■ **DE**

First two escape characters is this example had correct time gap before them, so they were counted as a part of the escape sequence and not saved into the buffer. The third escape character did not have a correct time gap so it was interpreted as a data character and saved into the buffer. The following was recorded into the RX buffer:

ABC■DE

The side effect and the point this example illustrates is that first two escape characters were lost -- they neither became a part of a successful escape sequence (because this sequence wasn't completed), nor were saved into the buffer.

Type 2

Type 2 escape sequence is not based on any timing. Escape sequence consists of escape character (defined by the ser.escchar property) followed by any character other than escape character. To

receive a *data* character whose ASCII code matches that of escape character the serial port must get this character *twice*. This will result in a single character being recorded into the RX buffer.

The following sequence will be recognized as escape sequence (that is, if current escape character is not 'D'):

ABC■D

In the sequence below two consecutive escape characters will be interpreted as data (data recorded to the RX buffer will contain only one such character):

ABC■■

.Flowcontrol Property

Function:	Sets/returns flow control mode for currently selected serial port (selection is made through ser.num 411)
Type:	dis_en (enum, byte)
Value Range:	0- DISABLED **(default)** 1- ENABLED
See Also:	UART Mode 380, Serial Settings 390

Details

Only relevant when the ser.mode 409 = 0- PL_SER_MODE_UART and ser.interface= 408 0- PL_SER_SI_FULLDUPLEX (full-duplex). Flow control uses two serial port lines- RTS and CTS- to regulate the flow of data between the serial port of your device and another ("attached") serial device.

Depending on your platform, you may be allowed to remap 195 RTS and CTS lines to other I/O pins of the device through the ser.rtsmap 415 and ser.ctsmap 403 properties. Also, you may be required to correctly configure 194 RTS and CTS lines as an input and output through the io.enabled 298 property of the io 294 object. For more information, refer to your device's platform documentation (for example, EM1000's is here 143).

When the flow control is disabled both RTS and CTS lines are not used by the serial port and become regular I/O lines that can be controlled through the io 294 object.

.Getdata Method

Function:	For the selected serial port (selection is made through ser.num 411) returns the string that contains the data extracted from the RX buffer.
Syntax:	**ser.getdata(**maxinplen **as word) as string**
Returns:	String containing data extracted from the RX buffer
See Also:	Three Modes of the Serial Port 380, Receiving data 396

Part	Description
maxinplen	Maximum amount of data to return (word).

Details

Extracted data is permanently deleted from the buffer. Length of extracted data is limited by one of the three factors (whichever is smaller): amount of committed data in the RX buffer itself, capacity of the "receiving" string variable, and the limit set by the maxinplen argument.

In the UART 380 mode (ser.mode 409 = 0- PL_SER_MODE_UART) the data is extracted "as is". For Wiegand 383 and clock/data 386 mode (ser.mode= 1- PL_SER_MODE_WIEGAND and ser.mode= 2- PL_SER_MODE_CLOCKDATA) each character of extracted data represents one data bit and only two characters are possible: '0' or '1'.

.Interchardelay Property

Function: Sets/returns maximum intercharacter delay for the selected serial port (selection is made through ser.num 411) in 10ms steps.

Type: Byte

Value Range: 0-255, **default**= 0 (no delay)

See Also: Three Modes of the Serial Port 380, Serial Settings 390

Details

For UART mode (ser.mode 409 = 0- PL_SER_MODE_UART) specifies the time that needs to elapse since the arrival of the most recent serial character into the RX buffer to cause the data to be committed (and on_ser_data_arrival 412 event generated). For Wiegand and clock/data mode (ser.mode= 1- PL_SER_MODE_WIEGAND or 2- PL_SER_MODE_CLOCKDATA) the time since the most recent data bit (high-to-low transition on the **W0&1in/cin** line) is counted.

In the UART mode this property allows you to combine incoming serial data into larger "chunks", which typically improves performance. Notice, that the intercharacter gap is not counted when the new data is not being received because the serial port has set the RTS line to LOW (not ready). For this to happen, the serial port must be in the UART/full-duplex/flow control mode (ser.mode= 0- PL_SER_MODE_UART, ser.interface 408 = 0- 0- PL_SER_SI_FULLDUPLEX, and ser.flowcontrol 407 = 1- ENABLED) and the RX buffer must be getting nearly full (less than 64 bytes of free space left).

For Wiegand and clock/data modes, counting timeout since the last bit is the only way to determine the end of the data output. Suggested timeout is app. 10 times the bit period of the data output by attached Wiegand or clock/data device.

.Interface Property

Function: Chooses full-duplex or half-duplex operating mode for currently selected serial port (selection is made through ser.num 411).

Type:	Enum (pl_ser_interface, byte)
Value Range:	0- PL_SER_SI_FULLDUPLEX **(default)**: full-duplex mode. 1- PL_SER_SI_HALFDUPLEX: half-duplex mode.
See Also:	UART Mode 380, Serial Settings 390

Details

Full-duplex mode is suitable for RS232, RS422, or four-wire RS485 communications. RTS output (together with CTS input) can be used for optional hardware flow control (ser.flowcontrol 407).

Half-duplex mode is suitable for two-wire RS485 communications. RTS line is used for direction control. Hardware flow control is not possible, so ser.flowcontrol value is irrelevant. Direction control polarity can be set through ser.dircontrol 404.

This property is only relevant when the port is in the UART mode (ser.mode 409= 0-PL_SER_MODE_UART).

Depending on your platform, you may be allowed to remap 195 RTS and CTS lines to other I/O pins of the device through the ser.rtsmap 415 and ser.ctsmap 403 properties. Also, you may be required to correctly configure 194 RTS and CTS lines as an input and output through the io.enabled 298 property of the io 294 object. For more information, refer to your device's platform documentation (for example, EM1000's is here 143).

.Mode Property

Function:	Sets operating mode for the currently selected serial port (selection is made through ser.num 411).
Type:	Enum (pl_ser_mode, byte)
Value Range:	0- PL_SER_MODE_UART **(default)**: UART mode. 1- PL_SER_MODE_WIEGAND: Wiegand mode. 2- PL_SER_MODE_CLOCKDATA: clock/data (magstripe interface) mode.
See Also:	Three Modes of the Serial Port 380, Serial Settings 390

Details

Follows is a short introduction of three operating modes of the serial port:

UART mode Suitable for RS232, RS422, RS485, etc. communications in full-duplex or half-duplex mode (see ser.interface 408). Data is transmitted through the **TX** pin and received through the **RX** pin. Optionally, **RTS** (output) and **CTS** (input) lines are used for flow control (see ser.flowcontrol 407) in the full-duplex mode. Additionally, RTS can be used for direction control in the half-duplex mode.

Wiegand mode Suitable for sending to or receiving data from any standard Wiegand device. Data transmission is through pins **W0out** and **W1out**, reception- through **W0&1in** and **W1in**. "W0&1in" means that a logical AND of W0 and W1 signals must be applied to this input. Therefore, external logical gate is needed in order to receive Wiegand data.

Clock/data mode Suitable for sending to or receiving data from any standard clock/data (or magstripe) device. Data transmission is through pins **cout** and **dout**, reception- through **cin** and **din**. Third line of the magstripe interface- card present- is not required for data reception. For transmission, any I/O line can be used as card present output (under software control).

Changing port mode is only possible when the port is closed (ser.enabled 405 = 0-NO). Depending on your platform, you may be allowed to remap 195 RTS/W1out/cout and CTS/W0&1in/cin lines to other I/O pins of the device through the ser.rtsmap 415 and ser.ctsmap 403 properties. Also, depending on selected mode and your platform you may be required to correctly configure 194 some of your serial port's lines as inputs or outputs through the io.enabled 298 property of the io 294 object. For more information, refer to your device's platform documentation (for example, EM1000's is here 143).

We understand that it would be much more logical to call this property "interface", not "mode". Problem is, this property was added later, when ser.interface already came to mean something else 408. So, we had no choice but to choose unused term.

.Newtxlen R/O Property

Function: For the selected serial port (selection is made through ser.num 411) returns the amount of uncommited TX data in bytes.

Type: Word

Value Range: 0-65535, **default**= 0 (bytes)

See Also: Sending (Transmitting) Data 397, ser.txlen 420, ser.txfree 420

Details

Uncommitted data is the one that was added to the TX buffer with the ser.setdata 418 method but not yet committed using the ser.send 417 method.

.Notifysent Method

Function: Using this method for the selected serial port (selection is made through ser.num 411) will cause the on_ser_data_sent 412 event to be generated when the amount of committed data in the TX buffer is found to be below "threshold" number of bytes.

Syntax:	**notifysent(**threshold **as word)**
Returns:	---
See Also:	Sending (Transmitting) Data 397

Part	Description
threshold	Amount of bytes in the TX buffer below which the event it so be generated.

Details

Only one on_ser_data_sent event will be generated each time after the ser.notifysent method is invoked. This method, together with the on_ser_data_sent event provides a way to handle data sending asynchronously.

Just like with ser.txfree 420, the trigger you set won't take into account any uncommitted data in the TX buffer.

.Num Property

Function:	Sets/returns the number of currently selected serial port (ports are enumerated from 0).
Type:	Byte
Value Range:	The value of this property won't exceed ser.numofports 411 -1 (even if you attempt to set higher value). **Default**= 0 (port #0 selected)
See Also:	---

Details

All other properties and methods of this object relate to the serial port selected through this property. Note that serial port-related events such as on_ser_data_arrival 412 change currently selected port!

.Numofports R/O Property

Function:	Returns total number of serial ports found on the current platform.
Type:	Byte
Value Range:	platform-dependent
See Also:	ser.num 411

Details

On_ser_data_arrival Event

Function: Generated when at least one data byte is present in the RX buffer of the serial port (i.e. for this port the ser.rxlen 417 >0).

Declaration: **on_ser_data_arrival**

See Also: Buffer Memory Status 394

Details

When the event handler for this event is entered the ser.num 411 property is automatically switched to the port for which this event was generated. Another on_ser_data_arrival 412 event on a particular port is never generated until the previous one is processed. Use ser.getdata 407 method to extract the data from the RX buffer.

You don't have to process all data in the RX buffer at once. If you exit the on_ser_data_arrival event handler while there is still some unprocessed data in the RX buffer another on_ser_data_arrival event will be generated immediately.

This event is not generated for a particular port when buffer redirection is set for this port through the ser.redir 414 method.

On_ser_data_sent Event

Function: Generated after the total amount of *committed* data in the TX buffer of the serial port (ser.txlen 420) is found to be less than the threshold that was preset through the ser.notifysent 410 method.

Declaration: **on_ser_data_sent**

See Also: Sending (Transmitting) Data 397

Details

This event may be generated only after the ser.notifysent method was used. Your application needs to use the ser.notifysent method EACH TIME it wants to cause the on_ser_data_sent event generation for a particular port. When the event handler for this event is entered the ser.num 411 is automatically switched to the port on which this event was generated.

Please, remember that uncommitted data in the TX buffer is not taken into account for on_ser_data_sent event generation.

On_ser_esc Event

Function: Generated when currently enabled escape sequence is detected in the RX data stream.

Declaration: **on_ser_esc**

See Also: Serial Settings 390

Details

Once the serial escape sequence is detected on a certain serial port this port is automatically disabled (ser.enabled= 0- NO).

When event handler for this event is entered the ser.num 411 property is automatically switched to the port on which this event was generated.

Whether or not escape sequence detection is enabled and what kind of escape sequence is expected is defined by the ser.esctype 405 property. Escape sequence detection works even when buffer redirection is set for the serial port using the ser.redir 414 method.

On_ser_overrun Event

Function: Generated when data overrun has occurred in the RX buffer of the serial port.

Declaration: **on_ser_overrun**

See Also: Handling Buffer Overruns 398

Details

Another on_ser_overrun event for a particular port is never generated until the previous one is processed. When the event handler for this event is entered the ser.num 411 property is automatically switched to the port on which this event was generated.

Data overruns are a common occurrence on serial lines. The overrun happens when the serial data is arriving into the RX buffer faster than your application is able to extract it, the buffer runs out of space and "misses" some incoming data.

Data overruns are typically prevented through the use of RTS/CTS flow control (see the ser.flowcontrol 407 property).

.Parity Property

Function: Sets/returns parity mode for the selected serial port (selection is made through ser.num 411)

Type: Enum (pl_ser_parity, byte)

Value Range:

0- PL_SER_PR_NONE: no parity bit to be transmitted.

1- PL_SER_PR_EVEN: even parity.

2- PL_SER_PR_ODD: odd parity.

3- PL_SER_PR_MARK: parity bit always at "1".

4- PL_SER_PR_SPACE: parity bit always at "0".

See Also: UART Mode 380, Serial Settings 390

Details

Mark parity is equivalent to having a second stop-bit (there is no separate property to explicitly select the number of stop bits).

This property is only relevant when the serial port is in the UART mode (ser.mode 409 = 0- PL_SER_MODE_UART).

.Redir Method

Function: For the selected serial port (selection is made through ser.num 411) redirects the data being RXed to the TX buffer of the same serial port, different serial port, or another object that supports compatible buffers.

Syntax: **ser.redir(**redir **as pl_redir) as pl_redir**

Returns: Returns 0- PL_REDIR_NONE if redirection failed or the same value as was passed in the redir argument.

See Also: Redirecting Buffers (Shorting) 399

Part	Description
redir	Platform-specific. See the list of pl_redir constants in the platform specifications.

Details

Data redirection (sometimes referred to as "buffer shorting") allows to arrange efficient data exchange between ports, sockets, etc. in cases where no data alteration or parsing is necessary, while achieving maximum possible throughput is important.

The redir argument, as well as the value returned by this method are of "enum pl_redir" type. The pl_redir defines a set of platform inter-object constants that include all possible redirections for this platform. Specifying redir value of 0- PL_REDIR_NONE cancels redirection. When the redirection is enabled for a particular serial port, the on_ser_data_arrival 412 event is not generated for this port.

Once the RX buffer is redirected certain properties and methods related to the RX buffer will actually return the data for the TX buffer to which this RX buffer was redirected:

- Ser.rxbuffsize 416 will actually be returning the size of the TX buffer.
- Ser.rxclear 416 method will actually be clearing the TX buffer.
- Ser.rxlen 417 method will be showing the amount of data in the TX buffer.

If the redirection is being done on a serial port that is currently opened (ser.enabled 405 = 1- YES) then this port will be closed automatically.

.Rtsmap Property (Selected Platforms Only)

Function:	Sets/returns the number of the I/O line that will act as **RTS/W0out/cout** output of currently selected serial port (selection is made through ser.num 411).
Type:	Enum (pl_io_num, byte)
Value Range:	Platform-specific, see the list of pl_io_num constants in the platform specifications.
See Also:	Three modes of the Serial Port 380, Serial Settings 390

Details

This property is only available on selected platforms. For more information, refer to your device's platform documentation (for example, EM1000's is here 143).

Default value of this property is different for each serial port. See the list of pl_int_num constants in the platform specifications -- it shows default values as well.

Absolutely any I/O line can be selected by this property, as long as this line is not occupied by some other function. Property value can only be changed when the port is closed (ser.enabled 405 = 0- NO).

On certain platforms, you may need to configure 194 the line as output. This is done through the io.enabled 298 property of the io 294 object.

.Rxbuffrq Method

Function:	For the selected serial port (selection is made through ser.num 411) pre-requests "numpages" number of buffer pages (1 page= 256 bytes) for the RX buffer of the serial port.
Syntax:	**ser.rxbuffrq(**numpages **as byte) as byte**
Returns:	Actual number of pages that can be allocated (byte)
See Also:	Allocating Memory for Buffers 393, ser.txbuffrq 419

Part	Description
numpages	Requested numbers of buffer pages to allocate.

Details

Returns actual number of pages that can be allocated. Actual allocation happens when the sys.buffalloc 530 method is used. The serial port is unable to RX data if its RX buffer has 0 capacity. Actual current buffer capacity can be checked through the ser.rxbuffsize 416 which returns buffer capacity in bytes.

Relationship between the two is as follows: ser.rxbuffsize=num_pages*256-16 (or =0 when num_pages=0), where "num_pages" is the number of buffer pages that was GRANTED through the ser.rxbuffrq. "-16" is because 16 bytes are needed for internal buffer variables.

Buffer allocation will not work if the serial port to which this buffer belongs is opened (ser.enabled 405 = 1- YES) at the time when sys.buffalloc executes. You can only change buffer sizes of ports that are closed.

.Rxbuffsize R/O Property

Function:	For the selected serial port (selection is made through ser.num 411) returns current RX buffer capacity in bytes.
Type:	Word
Value Range:	0-65535
See Also:	Buffer Memory Status 394

Details

Buffer capacity can be changed through the ser.rxbuffrq 415 method followed by the sys.buffalloc 530 method.

The ser.rxbuffrq requests buffer size in 256-byte pages whereas this property returns buffer size in bytes. Relationship between the two is as follows: ser.rxbuffsize=num_pages*256-16 (or =0 when num_pages=0), where "num_pages" is the number of buffer pages that was GRANTED through the ser.rxbuffrq. "-16" is because 16 bytes are needed for internal buffer variables. The serial port cannot RX data when the RX buffer has zero capacity.

.Rxclear Method

Function:	For the selected serial port (selection is made through ser.num 411) clears (deletes all data from) the RX buffer.
Syntax:	**ser.rxclear**
Returns:	---
See Also:	Buffer Memory Status 394

Details

.Rxlen R/O Property

Function:	For the selected serial port (selection is made through ser.num 411) returns total number of committed bytes currently waiting in the RX buffer to be extracted and processed by your application.
Type:	Word
Value Range:	0-65535
See Also:	Serial Settings 390, Buffer Memory Status 394

Details

The on_ser_data_arrival 412 event is generated once the RX buffer is not empty, i.e. there is data to process. There may be only one on_ser_data_arrival event for each port waiting to be processed in the event queue. Another on_serial_data_arrival event for the same port may be generated only after the previous one is handled.

If, during the on_ser_data_arrival event handler execution, not all data is extracted from the RX buffer, another on_ser_data_arrival event is generated immediately after the on_ser_data_arrival event handler is exited.

Notice that the RX buffer of the serial port employes "data committing" based on the amount of data in the buffer and intercharacter delay (ser.interchardelay 408). Data in the RX buffer may not be committed yet. Uncommitted data is not visible to your application and is not included in the count returned by the ser.rxlen.

.Send Method

Function:	For the selected serial port (selection is made through ser.num 411) commits (allows sending) the data that was previously saved into the TX buffer using the ser.setdata 418 method.
Syntax:	**ser.send**
Returns:	---
See Also:	Serial Settings 390

Details

You can monitor the sending progress by checking the ser.txlen 420 property or using the ser.notifysent 410 method and the on_ser_data_sent 412 event.

.Setdata Method

Function: For the selected serial port (selection is made through ser.num 411) adds the data passed in the txdata argument to the contents of the TX buffer.

Syntax: **ser.setdata(byref** txdata **as string)**

Returns: ---

See Also: Three Modes of the Serial Port 380, Sending Data 397, ser.txlen 420, ser.txfree 420,

Part	Description
txdata	The data to send.

Details

In the UART 380 mode (ser.mode 409= 0- PL_SER_MODE_UART) the data is added "as is". For Wiegand 383 and clock/data 386 mode (ser.mode= 1- PL_SER_MODE_WIEGAND and ser.mode= 2- PL_SER_MODE_CLOCKDATA) each data character represents one data bit and only bit0 (least significant bit) of each character is relevant (therefore, adding "0101" will result in the 0101 sequence of data bits).

If the buffer doesn't have enough space to accommodate the data being added then this data will be truncated. Newly saved data is not sent out immediately. This only happens after the ser.send 417 method is used to commit the data. This allows your application to prepare large amounts of data before sending it out.

Total amount of newly added (uncommitted) data in the buffer can be checked through the ser.newtxlen 410 setting.

.Sinkdata Property

Function: For the currently selected serial port (selection is made through ser.num 411) specifies whether the incoming data should be discarded.

Type: Enum (yes_no, byte)

Value Range: 0- NO **(default)**: normal data processing.

1- YES: discard incoming data.

See Also: Sinking Data 399

Details

Setting this property to 1- YES causes the serial port to automatically discard all incoming data without passing it to your application. The on_ser_data_arrival 412 event will not be generated, reading ser.rxlen 417 will always return zero, and so on. No data will be reaching its destination even in case of buffer redirection 399. Escape characters 390, however, will still be detected in the incoming data stream.

.Txbuffrq Method

Function:	For the selected serial port (selection is made through ser.num 411) pre-requests "numpages" number of buffer pages (1 page= 256 bytes) for the TX buffer of the serial port.
Syntax:	**ser.txbuffrq(**numpages **as byte) as byte**
Returns:	Actual number of pages that can be allocated (byte).
See Also:	Allocating Memory for Buffers 393, ser.rxbuffrq 415

Part	Description
numpages	Requested numbers of buffer pages to allocate.

Details

Returns actual number of pages that can be allocated. Actual allocation happens when the sys.buffalloc 530 method is used. The serial port is unable to TX data if its TX buffer has 0 capacity. Actual current buffer capacity can be checked through the ser.txbuffsize 419 which returns buffer capacity in bytes.

Relationship between the two is as follows: ser.txbuffsize=num_pages*256-16 (or =0 when num_pages=0), where "num_pages" is the number of buffer pages that was GRANTED through the ser.txbuffrq. "-16" is because 16 bytes are needed for internal buffer variables.

Buffer allocation will not work if the serial port to which this buffer belongs is opened (ser.enabled 405= 1- YES) at the time when sys.buffalloc executes. You can only change buffer sizes of ports that are closed.

.Txbuffsize R/O Property

Function:	For the selected serial port (selection is made through ser.num 411) returns current TX buffer capacity in bytes.
Type:	Word
Value Range:	0-65535
See Also:	Buffer Memory Status 394

Details

Buffer capacity can be changed through the ser.txbuffrq 419 method followed by the sys.buffalloc 530 method.

The ser.txbuffrq requests buffer size in 256-byte pages whereas this property returns buffer size in bytes. Relationship between the two is as follows: ser.txbuffsize=num_pages*256-16 (or =0 when num_pages=0), where "num_pages" is the number of buffer pages that was GRANTED through the ser.txbuffrq. "-16" is

because 16 bytes are needed for internal buffer variables. The serial port cannot TX data when the TX buffer has zero capacity.

.Txclear Method

Function:	For the selected serial port (selection is made through ser.num 411) clears (deletes all data from) the TX buffer.
Syntax:	**ser.txclear**
Returns:	---
See Also:	Buffer Memory Status 394

Details

This method will **only** work when the port is closed (Ser.Enabled Property 405 = NO). You cannot clear the TX buffers while the port is open.

.Txfree R/O Property

Function:	For the selected serial port (selection is made through ser.num 411) returns the amount of free space in the TX buffer in bytes.
Type:	Word
Value Range:	0-65535
See Also:	Buffer Memory Status 394

Details

Notice, that the amount of free space returned by this property does not take into account any uncommitted data that might reside in the buffer (this can be checked via ser.newtxlen 410). Therefore, actual free space in the buffer is ser.txfree-ser.newtxlen. Your application will not be able to store more data than this amount.

To achieve asynchronous data processing, use the ser.notifysent 410 method to get on_ser_data_sent 412 event once the TX buffer gains required amount of free space.

.Txlen R/O Property

Function:	For the selected serial port (selection is made through ser.num 411) returns total number of *committed* bytes currently found in the TX buffer.

Type:	Word
Value Range:	0-65535
See Also:	Serial Settings 390, Buffer Memory Status 394, ser.newtxlen 410

Details

The data in the TX buffer does not become committed until you use the ser.send 417 method.

Your application may use the ser.notifysent 410 method to get on_ser_data_sent 412 event once the total number of committed bytes in the TX buffer drops below the level defined by the ser.notifysent method.

Sock Object

This is the sockets object. It allows you to maintain up to 16 simultaneous UDP or TCP ("normal" or HTTP) connections (actual number supported by the platform may be lower, due to memory constraints).

Very commonly, each connection is called a "socket". This is the term we will use as well. On other programming systems, sockets are often dynamic, created and destroyed as needed. With TiOS, you receive a preset number of sockets which have already been created for you, and just use them. A socket may be idle, but it will still be there.

Individual sockets have all the traditional settings you would expect to find, such as destination port number, protocol, etc. At the same time, their functionality goes significantly beyond what you usually find, and includes a lot of additional features that significantly lower the amount of code you need to write. For example, you can restrict incoming connections to your device, automatically filter out certain messages within the TCP data stream, etc.

The sockets object also implements webserver (HTTP) functionality. Each socket can carry a "normal" data connection or be in the HTTP mode.

- Currently, the socket object can only access first 65534 bytes of each HTML file, even if the actual file is larger! Make sure that all HTML files in your project are not larger than 65534 bytes. This is not to be confused with the size of HTTP output generated by the file. A very large output can be generated by a small HTML file (due to dynamic data)- and that is OK. What's important is that the size of each HTML file in your project does not exceed 65534 bytes.

The sock object should not be confused with objects used to represent actual network interfaces, such as the net 358 object which represents the Ethernet interface. The socket object is responsible for actual IP (TCP or UDP) communications -- it doesn't matter which interface these communications are

effected through. Therefore, this is not the right place to look for a property such as 'IP address'. This is an attribute of a particular network interface.

Follows is the list of features offered by each socket of the sock object:

- Support for UDP, TCP and HTTP protocols (this is a submode of TCP).
- An extensive set of properties that define which hosts can connect to the socket, whether broadcasts are supported, which listening ports are associated with the socket, etc.
- Support for automatic processing of inband commands-- messages that are passed within the TCP data stream.
- Detailed socket state reporting with 30 different states supported!
- Fully asynchronous operation with separate "data arrival" and "data sent" events.
- Automatic data overrun detection on the RX buffer.
- Adjustable receive (RX), transmit (TX), and other buffer sizes for optimal RAM utilization.
- Buffer shorting feature for fast data exchange between the sock object and other objects (such as the ser 378 object) that support standard Tibbo Basic data buffers.

Overview 14.1

This section covers the socket object in detail. Here you will find:

- Anatomy of a socket 422
- Socket selection 423
- Handling Network Connection 424
- Sendind and Receiving Data 444
- Working With Inband Commands 456
- Using HTTP 461

Anatomy of a Socket

A socket is composed of a send/receive logic that actually handles UDP, TCP (including HTTP) communications, and of 6 buffers. Each socket is capable of maintaining one connection with another node (host) on a network.

The socket object contains properties, methods and events which relate both to the buffers and the send/receive logic.

The *buffers* available are:

- The **RX** buffer, which stores data incoming from the host on the other side of a connection (this buffer doesn't have to be used for HTTP connections 461).
- The **TX** buffer, which stores data which is due for sending to the host on the other side of a connection (for HTTP connection, this buffer can store both the request and the reply).
- The **TX2** buffer, which is used internally, and only when inband commands 456 are enabled.

- The **CMD** buffer, which is used to store incoming inband commands (messages). It is used only when inband commands are enabled.
- The **RPL** buffer, which is used to store outgoing inband replies (messages). It is used only when inband commands are enabled.
- The **VAR** buffer, which is used to store HTTP 461 request string. It is needed only when the socket is in the HTTP mode.

Socket Selection

TiOS supports up to 16 sockets, but there may be platforms with less than 16 sockets available. You can obtain the number of sockets available for your platform using the sock.numofsock 488 property.

Since there can be multiple sockets, you must state which socket are you referring to when changing properties or invoking methods. This is done using the sock.num 488 property. For example:

```
sock.protocol = 1
```

Can you tell what socket the statement above applies to? Neither can the platform. Thus, the correct syntax would be:

```
sock.num = 0
sock.protocol = 1
```

Now the platform knows which socket you're working with. Once you have set the *socket selector* (using sock.num), every socket-specific method and property after that point is taken to refer to that socket. Thus:

```
sock.num = 0
sock.protocol = 1
sock.connectiontimeout = 10
sock.httpmode = 1 ' etc
```

The events for this object are not separate for each socket. An event such as on_sock_data_arrival 489 serves all sockets on your platform. Thus, when an event handler for the socket object is entered, the socket selector is automatically switched to the socket number on which the event occurred:

```
sub on_sock_data_arrival
      dim s as string
      s = sock.getdata(255) ' Note that you did not have to specify any
sock.num preceding this statement.
end sub
```

As a result of this automatic switching, when an event handler for a socket event terminates, the sock.num property retains its new value (nothing changes it back).

You must take this into account when processing other event handlers which make use of a socket (and are not socket events). In other words, you should explicitly set the sock.num property whenever entering such an event handler, because the property might have been automatically changed prior to this event. To illustrate:

```
sub on_sys_init ' This is always the first event executed.
      sock.num = 0  ' Supposedly, this would make all subsequent properties
and methods refer to this socket.
end sub

sub on_sock_data_arrival ' Then, supposing this event executes.
      dim s as string
      s = sock.getdata(255) ' However, this event happens on the second
socket. So now sock.num = 1.
end sub

sub on_ser_data_arrival ' And then this serial port event executes.
      sock.txclear ' You meant to do this for sock.num = 0 (as specified at
on_sys_init). But now sock.num was changed to 1! Oops...
end sub
```

To recap, only one of two things may change the current sock.num: **(1)** manual change or **(2)** a socket event. You cannot assume the number has remained unchanged if you set it somewhere else (because a socket event might have happened since).

Handling Network Connections

The whole purpose of socket's existence is to engage in network connections with other network hosts. Each socket can maintain a single connection using a UDP/IP or TCP/IP *transport* protocol. Which of the two is used is defined by the sock.protocol 492 property.

Sockets are also capable of working with HTTP. HTTP is not a transport protocol, rather, it is based on the TCP. Therefore, when your socket is using TCP it may be for "plain data transmission" or for HTTP.

TCP connection basics

What is TCP

The TCP is the most widely used transmission protocol. It is the backbone of all Internet traffic. The idea behind the TCP is to provide two communicating points (we can call them host A and host B) with a reliable, stream-oriented data link. "Stream-oriented" means that neither the host A, nor the host B have to worry about how the data travels across the connection. A just puts the stream of bytes in and B receives exactly the same stream of bytes on its side. It is the responsibility of the TCP to split the data into packets for transmission through the network, retransmit lost packets, make sure there are no data overruns, etc.

The TCP is strictly a "point-to-point" protocol: only two parties can engage in a connection and no third party can "join in".

TCP connections

Before any data can be transmitted one of the hosts has to establish a connection to another host. This is similar to placing a telephone call: one of the parties has to call the other end.

The host that takes initiative to establish a connection is said to be opening an "outgoing connection" or "performing an active open". This is like dialing a telephone number of the desired party, only the number is the IP address of another host.

The host that accepts the "call" is said to be accepting an "incoming connection" or "performing a passive open". This is similar to picking up the phone when it starts ringing.

Once connection has been established, both parties can "say something" (send data) at any time and the TCP will make sure that all data sent on one end arrives to the other end.

TCP connections are expected to be closed (terminated) properly- there is a special exchange of messages between the host to let each other know that connection is being terminated. This is called "graceful disconnect". There is also a "reset" (abort) which is much simpler and is akin to hanging up abruptly. Finally, there is a "discard" way to end the connection is which the host simply "forgets" that there was a connection.

The TCP connection can be closed purposefully, or it can timeout 437.

A TCP connection in progress is fully defined by 4 parameters: IP address and the port number on host A and the IP address and port number on host B. When the host is performing an active open, it has to "dial" not just the IP address of the target host, but also the port number on this host. Ports are not physical- they are just logical subdivisions of the IP address (65536 ports per IP). If the IP is a telephone number of the whole office then the port is an extension. The "calling" host is also calling not just from its IP address but also from specific port.

UDP "connection" basics

What is UDP

In many aspects, the UDP protocol is the opposite of the TCP protocol. Whereas the TCP provides a realiable, stream-oriented transport, the UDP offers a way to send data as separate packets or "UDP datagrams". This is similar to "paging" (as in, sending a message with a "pager" -- remember those?). What is sent is a packet containing some data. There is no guarantee that the other side will receive the packet and the sender won't know whether the packet was received or not.

Each UDP datagram lives its own life and you are responsible for dividing your data into chunks of reasonable size and sending it in separate datagrams. There is no connection establishment or termination on UDP- the datagram can be sent instantly, without any preparation.

Unline TCP, the UDP is not point-to-point. For example, several hosts can send the datagrams to the same socket of your device- and all of them can be accepted- a situation that is impossible with TCP. There is also an option to broadcast the datagram to all hosts connected to the same network segment- something that is also impossible with TCP.

And now to UDP "connections"...

This said, it should come as a bit of a surprise that we will now turn 180 degrees and start talking about UDP "connections". Didn't we just say that there is no such thing? Well, yes and no. On the physical network there is, indeed, no such thing. However, on our socket object level we have deliberately made UDP

communications look more like TCP connections. And, since the UDP "connection" is nothing more than our sock object's abstraction, we use the word "connection" in quotation marks.

We consider an active open to have been performed when our host has sent the first UDP datagram to the destination. A passive open is when we have received the first incoming UDP datagram from another host (provided that this datagram was accepted- more on that in Handling Incoming Connections 426).

There is no graceful disconnect for UDP connections. The connection can only be discarded (our host "forgets" about it). The UDP connection can also timeout 437.

Accepting Incoming Connections

Master switch for incoming connections

It is possible to globally enable or disable acceptance of incoming connections on all sockets, irregardless of the setup of individual sockets. This is done using the sock.inconenabledmaster 484 property. By default, this property is set to 1- YES.

Defining who can connect to your socket

The sock.inconmode 485 ("incoming connections mode") property allows you do define whether incoming connections will be accepted and, if yes, from who. By default (0- PL_SOCK_INCONMODE_NONE), incoming connections are not allowed at all. For TCP this means that incoming connection requests will be rejected. For UDP, incoming datagrams will simply be ignored.

If you don't mind to accept an incoming connection from any host/port then set the sock.inconmode 485 = 3- PL_SOCK_INCONMODE_ANY_IP_ANY_PORT. This way, whoever wants to connect to your socket will be able to do so as long as this party is using correct transport protocol (you define this through the sock.protocol 492 property) and is connecting to the right port number (more on this below).

If you are only interested in accepting connections from a particular host on the network then set sock.inconmode 485 = 2- PL_SOCK_INCONMODE_SPECIFIC_IP_ANY_PORT. This way, only the host with IP matching the one defined by the sock.targetip 506 property will be able to connect to your socket.

You can restrict things further and demand that not only the IP of the other host must match the one you set in sock.targetip 506 property, but also the port from which the connection is being originated must match the port defined by the sock.targetport 507 property. To achieve this, set the sock.inconmode 485 = 1- PL_SOCK_INCONMODE_SPECIFIC_IPPORT.

Here is an example of how to only accept incoming TCP connections from host 192.168.100.40 and port 1000:

```
sock.protocol= PL_SOCK_PROTOCOL_TCP
sock.inconmode= PL_SOCK_INCONMODE_SPECIFIC_IPPORT
sock.targetip= "192.168.1.40"
sock.targetport= 1000
```

The sock object rejects an incoming connection by sending out a reset TCP packet. This way, the other host is instantly notified of the rejection. There is an exception to this -- see Socket Behavior in the HTTP Mode 465.

Listening ports

Ports on which your socket will accept an incoming connection are called "listening ports". These are defined by two properties: the sock.localportlist 486 and the sock.httpportlist 483. Notice that both properties are of string type, so each one can accept a list of ports.

For example, to accept a normal data connection either on port 1001, port 2000, or port 3000, set the sock.localportlist= "1001,2000,3000". Once the connection is in progress, you can check which of the socket's local ports is actually engaged in this connection. This is done through the sock.localport 485 read-only property.

For UDP connections, the sock.localportlist is all there is. For TCP, which can be used for "plain vanilla data connections" or for HTTP, you have another property- sock.httpportlist. To be accepted by your socket, an incoming TCP connection has to target either one of the ports on the sock.localportlist, or one of the ports on the sock.httpportlist. The socket will automatically switch into the HTTP mode if the connection is accepted on one of the ports from the sock.httpportlist.

Here is an example:

```
sock.localportlist = "1001,2000"
sock.httpportlist = "80"
```

The above means that any incoming TCP connection that targets either port 1001 or port 2000 will be interpreted as a regular data connection. If connection target port 80 it will be accepted as an HTTP connection.

And what if the same port is listed both under the sock.localportlist and sock.httpportlist?

```
sock.localportlist = "1001,2000,80"
sock.httpportlist = "80"
```

In this example, if there is an incoming connection targeting "our" port 80 and the protocol is TCP then the mode will be HTTP- the sock.httpportlist has priority over sock.localportlist. Of course, for UDP the sock.httpportlist won't matter since the HTTP is only possible on TCP!

Setting allowed interfaces

The sock object is interface-independent and supports communications over more than one interface. The sock.allowedinterfaces 474 property defines the list of interfaces on which the current socket will accept incoming connections. The list is different and depends on the platform. To find out what interfaces are available, refer to your device's platform documentation (for example, EM1000's is here 143).

The sock.allowedinterfaces 474 property stored the string that lists all interfaces that the socket will listen on:

```
sock.allowedinterfaces = "NET,WLN" 'listen on Ethernet and Wi-Fi interfaces
```

Connections accepted even when the VM is paused

Once the socket has been setup it will accept an incoming connection even when the VM is paused (for example, has stopped on your breakpoint). All communications are handled by the master process 7, so the socket does not need the VM to accept an incoming connection (or, for that matter, receive and send the data).

Accepting UDP broadcasts

UDP datagrams can be sent as broadcasts. Broadcast, instead of specifying a particular network host as a destination, targets a group of hosts on the network.

The sock object supports link-level broadcasts. Such broadcast packets have their destination MAC address set to 255.255.255.255.255.255. Link-level broadcasts are received by all network hosts connected to the current network segment. Link-level broadcasts cannot penetrate routers, bridges, etc.

For the socket to accept incoming UDP broadcasts, the sock.acceptbcast 474 property must be set to 1- YES. In every other aspect working with incoming broadcast UDP datagrams is like working with regular incoming datagrams.

Understanding TCP Reconnects

The sock object has a unique feature- support for "reconnects". To simplify the explanation, let us start from a description of a real-life problem first.

When reconnects save the day

Supposing, you have a host on the network that is engaged in a TCP connection with one of the sockets of the socket object. The data is sent across this connection from the host to our socket. For a while, everything is fine and then the host momentarily loses power and reboots. Our socket doesn't know anything about this- from its point of view, the TCP connection is still OK. Just because no data is arriving from the host does not mean that there is a problem!

Meanwhile, the host reboots and attempts to establish a new connection to the socket- and gets rejected! This is because the socket thinks it is already engaged in a TCP connection- the one that has been left hanging since the host went down! This stagnant connection will remain in place until it times out- if timeouts are enabled at all through the sock.connectiontout 477 property.

Reconnects are a nifty way out of this situation. You enable reconnects through the sock.reconmode 492 (reconnection mode) property. For the above example, you typically set the sock.reconmode= 2- PL_SOCK_RECONMODE_2. This mode means, that if the socket is already engaged in a connection, and then there is an incoming connection attempt that originated from the same host (and any port), then the socket will forget everything about the original connection and accept the new one.

For our example case, this is the solution- after rebooting the host tries to establish

a new connection to our socket. The socket then "realizes" that this new connection is being attempted from the same host as the connection already in progress, discards the original connection and accepts the new one!

Reconnects must target the same port and interface!

Even when the socket has more than one listening port (i.e. sock.localportlist 486= "1000,1001") the reconnect will only be accepted if it targets the same local port of the socket as the one already engaged in the current connection being "replaced". In other words, to be successful, the reconnect must target the port that is currently returned by the sock.localport 485 read-only property.

The interface must also be the same. The host can't make an original connection through Ethernet, and then reconnect through Wi-Fi.

Which mode to choose?

As you can see, the sock.reconmode property gives you several "strictness levels" of dealing with reconnects. Which one to choose? Let us explain why the choice of sock.reconmode= 2- PL_SOCK_RECONMODE_2 is the most common. Typically, when the host is establishing an outgoing connection it does so from the ever changing port number. Basically, there is a "pool" of ports for this purpose and each new connection the host needs to establish will be made from the next port in the pool.

Each time the host connects to our socket the port on the host could be different. This is why it makes sense to accept reconnects from the same IP but any port. Disadvantage? Any connection originating from this host will essentially be treated as the same and the only connection!

Some programs (few!) establish connections from a specific, preset port. This may be done for a variety of reasons- no time to go into this here- just let us say, sometimes this is the case. If you are dealing with such a case then you can safely set sock.reconmode= 1- PL_SOCK_RECONMODE_1. This way, reconnects will only be accepted from the same IP and the same port as the original connection.

Total promiscuity -- mode 3!

Finally, there is a mode (3- PL_SOCK_RECONMODE_3) when the socket will accept a reconnect from any host or port. Basically, this means, that whatever connection is in progress, it will be interrupted and replaced by any other incoming connection.

Do not use reconnects for HTTP sockets!

Reconnects and HTTP do not play nicely together. When you request an HTML page, several simultaneous HTTP requests may be generated (one for the page itself, several -- for pictures on this page, etc.). All these requests will use a separate TCP socket, so multiple sockets will be opened (almost) at the same time. Now, what will happen if even just one of your application's "HTML" sockets has reconnects enabled? This single socket will intercept all HTML requests. So, if loading the HTML page needed 3 separate requests and TCP sessions, this socket will get them all -- and each next session opening will discard the previous one. Result won't be pretty!

Understanding UDP Reconnects and Port Switchover

For UDP "connections", there is also such a thing as reconnects. Due to a very different nature of UDP (compared to TCP), reconnects for UDP must be explained separately. Additionally, we need to introduce something called "port switchover".

Port switchover explained

With TCP, each side of the connection uses a single port both to send and receive data. With UDP, this doesn't have to be the case. The sock object, when it is engaged in a connection, can receive the data from one port but send the data to a different port!

When port switchover is disabled, the socket always addresses its outgoing UDP datagrams to the port, specified by the sock.targetport 507 property. When port switchover is enabled, the socket will address its outgoing datagrams to the port from which the most recent incoming datagram was received!

Notice, that we did not say anything about the IP switchover- so far we have only discussed ports.

UDP reconnects

Just like with TCP, the sock.reconmode 492 property defines, for UDP, what kind of incoming UDP datagram will be able to make the socket forget about its previous "connection" and switch to the new one. You have two choices: either define that reconnects are only accepted from a specific IP but any port or choose reconnects to be accepted from any IP and any port. Combine this with two options for port switchover and you have four combinations- four options for the sock.reconmode property. All this is best understood on the example.

Example: PL_SOCK_RECONMODE_0

Setup:

```
sock.protocol= PL_SOCK_PROTOCOL_UDP 'we are dealing with UDP
sock.inconmode= PL_SOCK_INCONMODE_ANY_IPPORT
sock.reconmode= PL_SOCK_RECONMODE_0 'reconnects accepted from the same IP,
any port, port switchover off
sock.localportlist= "3000"
sock.targetport= "900"
```

Two hosts are sending us datagrams and here is how the socket will react:

Incoming datagram	Socket reaction
192.168.100.40:1000 sends UDP datagram to the port 3000 of our device	Datagram accepted, UDP "connection" is now in progress, the socket will be sending all further outgoing datagrams to 192.168.100.40:900.
192.168.100.44:1001 sends UDP datagram to the port 3000 of our device	Datagram ignored- it came from a different IP. The socket will still be sending its outgoing datagrams to 192.168.100.40:900.
192.168.100.40:1100 sends UDP datagram to the port 3000 of our device	Reconnection accepted (so datagram is accepted), but the socket will still be sending all further outgoing datagrams to 192.168.100.40:900 (because port switchover is off).

Example: PL_SOCK_RECONMODE_1

Setup:

```
sock.protocol= PL_SOCK_PROTOCOL_UDP 'we are dealing with UDP
sock.inconmode= PL_SOCK_INCONMODE_ANY_IPPORT
sock.reconmode= PL_SOCK_RECONMODE_1 'reconnects accepted from any IP/port,
port switchover off
sock.localportlist= "3000"
sock.targetport= "900"
```

Two hosts are sending us datagrams and here is how the socket will react:

Incoming datagram	Socket reaction
192.168.100.40:1000 sends UDP datagram to the port 3000 of our device	Datagram accepted, UDP "connection" is now in progress, the socket will be sending all further outgoing datagrams to 192.168.100.40:900.
192.168.100.44:1001 sends UDP datagram to the port 3000 of our device	Reconnection accepted, the socket will be sending all further outgoing datagrams to 192.168.100.44:900.

Example: PL_SOCK_RECONMODE_2

Setup:

```
sock.protocol= PL_SOCK_PROTOCOL_UDP 'we are dealing with UDP
sock.inconmode= PL_SOCK_INCONMODE_ANY_IPPORT
sock.reconmode= PL_SOCK_RECONMODE_2 'reconnects accepted from the same IP,
any port, port switchover on
sock.localportlist= "3000"
sock.targetport= "900"
```

Two hosts are sending us datagrams and here is how the socket will react:

Incoming datagram	Socket reaction
192.168.100.40:1000 sends UDP datagram to the port 3000 of our device	Datagram accepted, UDP "connection" is now in progress, the socket will be sending all further outgoing datagrams to 192.168.100.40:1000.
192.168.100.44:1001 sends UDP datagram to the port 3000 of our device	Datagram ignored- it came from a different IP. The socket will still be sending its outgoing datagrams to 192.168.100.40:1000.
192.168.100.40:1100 sends UDP datagram to the port 3000 of our device	Reconnection accepted, the socket will be sending all further outgoing datagrams to 192.168.100.40:1100.

Example: PL_SOCK_RECONMODE_3

Setup:

```
sock.protocol= PL_SOCK_PROTOCOL_UDP 'we are dealing with UDP
sock.inconmode= PL_SOCK_INCONMODE_ANY_IPPORT
sock.reconmode= PL_SOCK_RECONMODE_3 'reconnects accepted from any IP/port,
port switchover on
sock.localportlist= "3000"
sock.targetport= "900"
```

Now, two hosts are sending us datagrams and here is how the socket will react:

Incoming datagram	Socket reaction
192.168.100.40:1000 sends UDP datagram to the port 3000 of our device	Datagram accepted, UDP "connection" is now in progress, the socket will be sending all further outgoing datagrams to 192.168.100.40:1000.
192.168.100.44:1001 sends UDP datagram to the port 3000 of our device	Reconnection accepted, the socket will be sending all further outgoing datagrams to 192.168.100.44:1001.

Incoming Connections on Multiple Sockets

So far, we have been talking about all kinds of incoming connections applied to a single socket. The fact is, the sock object supports up to 16 sockets which all can have a different setup. Just because an incoming connection is rejected or ignored on one socket does not mean that it won't be connected on another!

When the network packet is received by the sock object the latter attempts to find a socket for which this newly arrived packet is acceptable.

First, the sock object checks if the packet can be considered a part of any existing connection or a reconnection attempt for an existing connection. All sockets that are currently engaged in a connection are checked, starting from socket 0, then sock 1, 2, and up to sock.numofsock 488 - 1.

If it turns out that some socket can accept the packet as a part of current connection or an acceptable reconnection attempt then the search is over and the packet is "pronounced" to belong to this socket.

If it turns out that no socket currently engaged in a connection can accept the packet then the socket object checks all currently idle sockets to see if any of these sockets can accept this packet as a new incoming connection. Again, all idle

sockets are checked, starting from socket 0, and up to sock.numofsock-1.

For TCP, a packet that can start such a new connection is a special "SYN" packet. For UDP, any incoming datagram that can be accepted by the socket (which depends on the socket setup) can start the new connection.

If the packet cannot be construed as a part of any existing connection, re-connect, or new incoming connection then this packet is discarded.

Example

Supposing, we have the following setup:

```
sock.num= 0
sock.protocol= PL_SOCK_PROTOCOL_TCP
sock.inconmode= PL_SOCK_INCONMODE_SPECIFIC_IP_ANY_PORT
sock.targetip="192.168.100.40"
sock.reconmode= PL_SOCK_RECONMODE_2
sock.localportlist= "1001"

sock.num=1
sock.protocol= PL_SOCK_PROTOCOL_TCP
sock.inconmode= PL_SOCK_INCONMODE_ANY_IPPORT
sock.localportlist= "1001"

sock.num=2
sock.protocol= PL_SOCK_PROTOCOL_UDP
sock.inconmode= PL_SOCK_INCONMODE_ANY_IPPORT
sock.localportlist= "2000"
```

Here is a sample sequence of incoming connections and how the setup above would handle them:

Incoming datagram	Socket reaction
Incoming TCP connection to port 1001 from host 192.168.100.40:29600	Connection will be accepted on socket 0 since this socket lists 1001 as one of the listening ports and this incoming connection request is from "correct" host.
Incoming TCP connection to port 1001 from host 192.168.100.40:29601	This will be taken as a reconnect on socket 0: this incoming connection is from the same host as the previous one and it targets the same port 1001.
Incoming TCP connection to port 1001 from host 192.168.100.41:900	Connection will be accepted on socket 1, because socket 0 is already engaged in a connection and this new connection request cannot be intepreted as a reconnect (different host).
Incoming UDP datagram to port 2000 from host 192.168.100.40:320	This datagram will be accepted on socket 2 and "connection" will be opened.

Establishing Outgoing Connections

Performing active opens

Now that we know how to setup the sockets to accept incoming TCP connections and UDP "connections" we move on to learning about establishing connections of our own, or, as is often said, performing "active opens".

Establishing an outgoing connection is always an explicit action- you use the sock.connect 477 method *to attempt* to do this. Once you do this the socket will try to perform an active open to the IP and port specified by the sock.targetip 506 and sock.targetport 507 properties. There is also a sock.targetinterface 506 property -- this one defines which network interface the new connection will be passing through.

As you can see, the sock.targetip and sock.targetport properties perform a double-duty: for incoming connections they define (if required by the sock.inconmode 485) who will be able to connect to the socket. For outgoing connections this pair defines IP and host to which the socket will attempt to connect.

Notice, that your sock.connect invocation will only work if you do this while the socket is in the "closed" state (see Checking Connection Status 439).

Active opens for TCP

Once you tell a "TCP" socket to connect, the socket will do the following:

- Resolve the IP address of the target using the ARP protocol.
- Attempt to engage the target in a standard TCP connection sequence (SYN-SYN-ACK).
- Connection will be either established, or this will fail. Your program has a way to monitor this- see Checking Connection Status 439.

If what we've just said doesn't ring any bell for you, don't worry. These, indeed, are very technical things and you don't have to understand them fully to be able to use the sock object.

"Active opens" for UDP

When you tell an "UDP" socket to connect, the latter will just resolve the IP address of the target and consider "connection" established.

Do not forget: connection handling is fully asynchronous!

Keep in mind that the sock object handles communications asynchronously. When the VM executes the sock.connect 477 method it does not mean that the connection is established by the time the VM gets to the next program statement. Executing sock.connect merely instructs the master process to start establishing the connection (more on master process and the VM in the System Components 7 topic). Connection establishment can take some time and your application doesn't have to wait for that to complete. Checking Connection Status 439 topic explains how to find out actual connection status at any time (see sock.state 502 and

sock.statesimple 505 R/O properties).

- Asynchronous nature of the sock object has some interesting implications. More On the Socket's Asynchronous Nature 441 topic contains important information on the subject, so make sure you read it!

Sending UDP broadcasts

How to send UDP broadcasts

UDP datagrams can be sent as broadcasts. Broadcast, instead of specifying a particular network host as a destination, targets a group of hosts on the network.

The sock object supports link-level broadcasts. Such broadcast packets have their destination MAC address set to 255.255.255.255.255.255. Link-level broadcasts are received by all network hosts connected to the current network segment. Link-level broadcasts cannot penetrate routers, bridges, etc.

To make the socket send its outgoing UDP datagrams as broadcasts, set the sock.targetbcast 505 property to 1- YES:

```
...
sock.targetbcast= YES
sock.setdata("ABC") 'this is explained in 'Working With Buffers' section
sock.send 'this is explained in 'Working With Buffers' section
sock.connect 'broadcast UDP datagram with string 'ABC' will be sent out at
this point
...
```

There is one difference to grasp regarding the socket that is sending its outgoing packets as broadcasts: no incoming UDP packet that would have normally be interpreted as a re-connect will cause the socket to "switchover" to the source IP-address of the packet.

Let's evaluate two examples.

Example 1: regular UDP communications

Here is a sample setup for the UDP socket:

```
sock.protocol= PL_SOCK_PROTOCOL_UDP
sock.inconmode= PL_SOCK_INCONMODE_ANY_IP_ANY_PORT
sock.reconmode= PL_SOCK_RECONMODE_3
sock.targetip= "192.168.100.40"
sock.targetport= 1000
sock.localportlist= "2000"
sock.setdata("ABC")
sock.send
```

```
sock.connect
```

And here is a hypothetical sequence of events:

Incoming/outgoing datagram	Comment
Datagram with contents "ABC" sent to 192.168.100.40:1000 as soon as sock.connect method is invoked	We now have the UDP "connection" with 192.168.100.40:1000
Incoming UDP datagram from 192.168.100.41:20.	This will be taken as a reconnect- the socket is now engaged in a connection with 192.168.100.41:20.
Socket sends out another datagram- this time to 192.168.100.41:20!	Complete switchover happened- the socket is now transmitting data to the IP and port of the sender of the previous incoming datagram

Example 2: regular UDP communications

Here is another setup for the UDP socket:

```
sock.protocol= PL_SOCK_PROTOCOL_UDP
sock.inconmode= PL_SOCK_INCONMODE_ANY_IP_ANY_PORT
sock.reconmode= PL_SOCK_RECONMODE_3
sock.targetbcast= YES
sock.targetport=1000
sock.localportlist= "2000"
sock.setdata("ABC")
sock.send
sock.connect
```

Sequence of events:

Incoming/outgoing datagram	Comment
Datagram with contents "ABC" sent as broadcast to all stations on the network segment	We now have the UDP "connection" with... err... everybody on the segment
Incoming UDP datagram from 192.168.100.41:20.	This is still a reconnect, but the socket won't switch over to the IP-address of the sender. Only port switchover will take place!
Socket sends out another datagram- still as a broadcast but this time to port 20.	Port switchover happened because it is allowed by the sock.reconmode 492.

Closing Connections

Passive TCP connection termination

When your socket is engaged in a TCP connection with another host on the network, this host may choose to terminate the connection. This can be done through a "graceful disconnect" sequence ("FIN-ACK-FIN-ACK") or through a reset ("RST" packet).

In both cases the socket will handle connection closing automatically, without any help from your Tibbo BASIC program or the VM. In fact, the socket will accept connection termination even when the VM is stopped (for example, it was paused on your breakpoint). All communications are handled by the master process 7, so the socket does not need the VM to terminate the connection.

There is one intricate detail in the connection termination process that you will have to understand clearly. When the VM is not running, the socket can accept an incoming connection 426 (active open) and can accept an active close. However, the socket won't be able to accept another (next) incoming connection until the VM has had a chance to run and execute the on_sock_event 490 event handler (see Checking Connection Status 439).

Here is why. In many cases there is a need to perform certain actions (like, maybe, clear some buffers, initialize variables, etc.) after the previous connection ends and before the new one begins. Not letting the socket accept next connection before your program has a chance to respond to connection termination is a way to achieve this!

Actively closing TCP connections

A TCP connection can be closed, reset (aborted), or discarded using three different methods- sock.close 475, sock.reset 495, and sock.discard 478.

The recommended (and polite!) way of closing a TCP connection is through the sock.close method. This will initiate a proper closing sequence ("FIN-ACK-FIN-ACK"), known as a "graceful disconnect".

Connection reset is slightly more "rude"- your socket will simply tell the other end that "it is all over" (by sending the "RST" packet).

Finally, the sock.discard method simply makes your socket forget about the connection- the other side is not notified at all.

Just like with connection establishment, you can monitor the progress of connection termination- see Checking Connection Status 439.

Notice, that depending on the socket state at the moment of sock.close method invocation, the socket may need to resort to a simple connection closing option- reset or discard. Similarly, when you use the sock.reset, it may sometimes result in discard. For more information read the sock.close 475, sock.reset 495, and sock.discard 478 topics.

Actively closing UDP connections

For UDP, no matter what method you use to close the "connection", result will still be as if the sock.discard was invoked. This is because in reality there is no such thing as a proper UDP connection termination so simply "forgetting" about the connection is the only option the socket has.

Do not forget: connection handling is fully asynchronous!

Keep in mind that the sock object handles communications asynchronously. When the VM executes the sock.close 475 (sock.reset 495, sock.discard 478) method it does not mean that the connection is done with by the time the VM gets to the next program statement. Executing these methods merely instructs the master process to terminate the connection. Connection termination can take some time and your application doesn't have to wait for that to complete. Checking Connection Status 439 topic explains how to find out actual connection status at any time (see sock.state 502 and sock.statesimple 505 read-only properties).

- Asynchronous nature of the sock object has some interesting implications. More On the Socket's Asynchronous Nature 441 topic contains important information on the subject, so make sure you read it!

Socket re-use after connection closing

When the socket connection terminates, the socket is ready to accept another incoming connection or establish a new outgoing connection (if so configured) -- with one little caveat! There is a special built-in mechanism that ensures that your application has a chance to react after the previous connection terminates and before the next one is established.

For example, you might need to clean some buffers before each new incoming TCP connection. Naturally, you want this to happen before the new connection is actually accepted.

Typically, your program achieves this by executing code placed in the on_sock_event 490 event handler (this event is explained in Checking Connection Status 439 topic). The socket will not be able to engage in the new connection until the on_sock_event has a chance to execute. There is an interesting example on this in the More On the Socket's Asynchronous Nature 441 topic.

Connection timeouts

The sock.connectiontout 477 provides a way to automatically terminate a connection across which no data was exchanged for a predefined period of time. For TCP, reset (abort) is used, while UDP "connections" are simply discarded. Connection timeout is a useful way to exit "hanged" connections (this happens a lot with TCP on large networks).

The sock.toutcounter 507 R/O property informs your application of the time passed since the data was last exchanged across the connection. Each time there is some data sent or received the sock.toutcounter is reset to zero. The property increments at 0.5 second intervals while no data is moving through this socket.

If the sock.connectiontout is not at 0, the sock.toutcounter increments until it reaches the value of the sock.connectiontout and the connection is terminated. The sock.toutcounter then stays at the value of sock.connectiontout.

If the sock.connectiontout is at 0, the maximum value that the sock.toutcounter can reach is 1. That is, the sock.toutcounter will be at 0 after the data exchange, and at 1 if at least 0.5 seconds have passed since the last data exchange.

Normally, HTTP connections close automatically

There is one case where your socket will perform an active graceful disconnect without you using the sock.close method. This is the case when the socket is running in the TCP-HTTP mode.

In Accepting Incoming Connections 426 we have already explained that the socket automatically switches into the HTTP mode if a TCP connection is accepted on one of the ports from the sock.httpportlist 483 list. The socket can also be switched into the HTTP mode programmatically, through the sock.httpmode 481 property.

Default HTTP functionality requires that the TCP connection is closed once the HTTP server has finished sending out its "response" (i.e. HTML page or another file that has been requested). In this situation the socket won't need the sock.close from your program- the connection will be terminated automatically. In fact, when the socket is in the HTTP mode, your sock.close 475, sock.reset 495, and sock.discard 478 will simply be ignored. There is a sock.httpnoclose 482 property alters the standard socket behavior in the HTTP mode. Set this property to 1- YES and the connection will be kept opened even after the socket has sent all of the HTTP reply out.

Just like in all other cases, a new connection on the socket won't be accepted until your program has had a chance to respond- this was explained above.

Checking Connection Status

Simplified and detailed socket states

The sock object features several properties that provide complete information on what the socket is doing, which IP and port it is engaged in the connection with, etc.

The most important of all this data is the socket state. Two state groups are supported- "simplified" and "detailed". Simplified state tells you, generally, what condition the socket is in. Detailed state additionally tells you how this condition came to be.

For example, the "simplified" PL_SSTS_CLOSED state means that connection is closed (socket is idle). It doesn't tell you, however, why it is closed. "Detailed" state PL_SST_CL_ARESET_CMD tells you that connection is closed because there was an active close as a result of the sock.close 475 method invocation from your program!

On_sock_event and sock.event, sock.eventsimple read-only properties

Each time the socket state changes an on_sock_event 490 event is generated. Unlike many other events, this one can be generated again and again before the on_sock_event handler is invoked, so event queue can contain multiple such events. A separate event is generated for each new state the socket enters.

The on_sock_event "brings" two arguments with it -- one is called "newstate", another one -- "newstatesimple". These arguments contains the state of the socket at the moment when the on_sock_event was generated.

Newstate and newstatesimple arguments have been introduced in Tibbo Basic **V2.0**. Before, their role was played by two read-only properties -- sock.event 480 and sock.eventsimple 480. These properties are no longer available.

Here is one example of how on_sock_event can be used. Supposing, when connection is established we need to open the serial port and when it is no longer established close the serial port and clear the data from its TX and RX buffers:

```
sub on_sock_event(newstate as pl_sock_state,newstatesimple as
pl_sock_state_simple)
   if newstatesimple= PL_SSTS_EST then
      ser.enabled= YES 'connection has been established- open port
```

```
   else
      if ser.enabled= YES then 'connection is NOT established? Close and
clear the port if it is opened!
         ser.enabled= N0
         ser.txclear
         ser.rxclear
      end if
   end if
end sub
```

Sock.state and sock.statesimple read-only properties

There is also a pair of read-only properties- sock.state[502] and sock.statesimple[505]- that allows you to check current socket state (as opposed to the state that was at the moment of on_sock_event generation).

Here is an example of how you can use this. Supposing, you want to "play" an LED pattern that depends on the current state of the socket. Here is how you do this:

```
sub on_pat
'this event is generated each time a pattern finishes playing
      select case sock.statesimple
      case PL_SSTS_EST: 'connection is established- Green LED on
            pat.set("GGGGGGGGGGGGGGGG",NO)
      case PL_SSTS_ARP: 'ARP in progress- Green LED blinks
            pat.set("G-G-G-G-G-G-G-G-",NO)
      case PL_SSTS_PO: 'connection is being established- green LED goes
'blink-blink-blink'
            pat.set("----------G-G-G-",NO)
      case PL_SSTS_AO: 'connection is being established- green LED goes
'blink-blink-blink'
            pat.set("----------G-G-G-",NO)
      case else: 'connection is closed or being closed- green LED is 'blink-
blink'
            pat.set("------------G-G-",NO)
      end select
end sub
```

Understanding who you are talking to

Whenever the socket is engaged in the connection you can check the parameters of the other side through three read-only properties- sock.remotemac[495], sock.remoteip[494], and sock.remoteport[495]. For UDP, you can also check if the datagram you have received was sent to your device exclusively or it was a broadcast- the sock.bcast[475] will tell you that. For TCP, you can additionally check if the socket is in the "regular data" or HTTP mode- just check the sock.httpmode[481] property.

There is an intricate detail to understand about the sock.remotemac, sock.remoteip, sock.remoteport, and sock.bcast properties when you are using the UDP protocol.

With UDP, your socket may be accepting datagrams from several different hosts. As will be explained in Receiving Data in UDP Mode[450], the most common way to handle the incoming data is through the on_sock_data_arrival[489] event. You will get one such event for each UDP datagram that the socket will receive. If you check the sock.remotemac, sock.remoteip, sock.remoteport, or sock.bcast from within the

on_sock_data_arrival event handler you will get the sender's data for the UDP datagram *currently* being processed.

On the contrary, using sock.remotemac, sock.remoteip, sock.remoteport, or sock.bcast property outside of the on_sock_data_arrival event handler will give you the data for the *most recent* UDP datagram received into the RX buffer of the socket. This is not the same as the *next* UDP datagram to be extracted from the RX buffer and processed by your application!

Checking current interface

Starting from **V1.2**, the sock object provides an additional property, sock.currentinterface 478, which tells you which network interface the network connection is going through.

More On the Socket's Asynchronous Nature

In Establishing Outgoing Connections 434 and Closing Connections 437 topics we have already touched on the subject of the sock object's asynchronous nature. This topic offers further details on what that means for your application.

Executing sock.connect 477, sock.close 475, sock.reset 495, or sock.discard 478 method does not mean that your connection gets established or terminated by the time your program reaches the next statement. Executing these statements merely instructs the Master Process what to do with the connection (more on the Master Process in the System Components 7 topic). Connection establishment/termination can take some time and your application doesn't have to wait for that to complete. Checking Connection Status 439 topic explained how to find out actual connection status at any time (see sock.state 502 and sock.statesimple 505 read-only properties) .

There are certain situations when your program has to take the above into account. Here is one example. Supposing, we want to know the MAC address of a remote device to which we are establishing an outgoing connection. Naturally, we can do it this way:

```
'Correct code -- on startup we order the connection to be established and in
the on_sock_event event handler we record the MAC  address of the 'other
side'

Sub On_sys_init
   ...
   sock.targetip="192.168.100.40"   'we prepare for the connection
   sock.targetport=2000
   sock.connect             'and now we connect
End Sub

'-------------------------------------------------------------------------
---------
Sub On_sock_event(newstate As pl_sock_state,newstatesimple As
pl_sock_state_simple)
   Dim s As String
   If newstatesimple=PL_SSTS_EST Then
```

```
      'OK, so connection is established, let's get this MAC!
      s=sock.remotemac
   End If
End Sub
```

The above is a good example of event-driven programming. Sometimes, however, you need to establish a connection and "follow-up" on it in the same event handler. So, how do we do this? Here is a simple, and WRONG code:

```
'!!! BAD EXAMPLE !!!

Dim s As String
...
...

sock.targetip="192.168.100.40"   'we prepare for the connection
sock.targetport=2000
sock.connect             'and now we connect
s=sock.remotemac         'and now we try to check the MAC. WRONG! Connection
may not be established yet!

...
```

And here is the correct way to handle this. For clarity, this example assumes that connection will definitely be established.

```
'Correct, but simplified example (we do not handle possible connection
failure).

Dim s As String
...
...

sock.targetip="192.168.100.40"   'we prepare for the connection
sock.targetport=2000
sock.connect             'and now we connect
While sock.statesimple<>PL_SSTS_EST 'we wait here until the connection is
actually established
Wend
s=sock.remotemac         'Get the MAC!

...
```

Here is even more interesting example. Supposing, you want to close and reestablish a TCP connection right within the same event handler. Here is a wrong way of doing this:

```
'!!! BAD EXAMPLE !!! -- this just won't work!
...
...

sock.close
sock.connect
```

```
...
```

You see, executing sock.close doesn't really close the connection -- it only issues the instruction (to the Master Process) to close the connection. So, by the time program execution gets to the sock.connect method your previous connection is still on!

Correct way is to wait for the connection to actually be closed before executing sock.connect. Here is another example -- not quite correct either -- but closer to the truth.

```
'!!! 'BETTER' CODING !!! -- still not totally OK!
...
...

sock.close
   While sock.statesimple<>PL_SSTS_CLOSED 'here we wait for the connection
to be closed
   Wend
sock.connect
...
```

OK, this is better. One final correction and the code is complete. In the Closing Connections 437 topic ("Socket re-use after connection closing" section) we have already explained that the OS makes sure that the on_sock_event has a chance to execute after the old connection is closed and before the new one is established. In the above example both sock.close and sock.connect are in the same event handler -- the on_sock_event won't squeeze in between them unless you use the doevents 73 statement! Here is the correct code:

```
'100% CORRECT!
...
...

sock.close
   While sock.statesimple<>PL_SSTS_CLOSED 'here we wait for the connection
to be closed
   Wend
   doevents ' Absolutely essential for this particular case!
sock.connect
...
```

Notice how doevents is placed after the while-wend loop. It is absolutely essential that you do it this way! Of course, now that you have at least one doevents in the event handler you might as well add doevents in all "places of waiting" -- just to let other events execute sooner.

```
'Even better code!
...
...

sock.close
   While sock.statesimple<>PL_SSTS_CLOSED 'here we wait for the connection
```

```
to be closed
      doevents 'Not necessary but useful -- lets other events execute
   Wend
   doevents ' Absolutely essential for this particular case!
sock.connect
...
```

- You have to place doevents after the while-wend loop and you have to do this even if you don't actually have a handler for the on_sock_event 490 event in your application!

Sending and Receiving data

Once a network connection has been established the socket is ready to send and receive the data. This is done through two buffers- the TX buffer and the RX buffer. Read on and you will know how to allocate memory for buffers, use them, handle overruns, and perform other tasks related to sending and receiving of data.

Allocating Memory for Buffers

Each buffer has a certain size, i.e, a memory capacity. This capacity is allocated upon request from your program. When the device initially boots, no memory is allocated to buffers at all.

Memory for buffers is allocated in pages. A *page* is 256 bytes of memory. Allocating memory for a buffer is a two-step process: first you have to request for a specific allocation (a number of pages) and then you have to perform the actual allocation.

For the socket object to be able to send and receive the data, you have to give its TX and RX buffers some memory. This is done through the sock.txbuffrq 509 and sock.rxbuffrq 497 methods.

The allocation method (sys.buffalloc 530) applies to all buffers previously specified, in one fell swoop.

Hence:

```
dim in, out as byte
out = sock.txbuffrq(10) ' Requesting 10 pages for the TX buffer. Out will
then contain how many can actually be allocated.

in = sock.rxbuffrq(7) ' Requesting 7 pages for the RX buffer. Will return
number of pages which can actually be allocated.

' ...  Allocation requests for other buffers...

sys.buffalloc ' Performs actual memory allocation, as per previous requests.
```

Actual memory allocation takes up to 100ms, so it is usually done just once, on boot, for all required buffers. If you do not require some buffer, you may choose not to allocate any memory to it. In effect, it will be disabled.

You may not always get the full amount of memory you have requested. Memory is not an infinite resource, and if you have already requested (and received) allocations for 95% of the memory for your platform, your next request will get up to 5% of memory, even if you requested for 10%.

There is a small overhead for each buffer. Meaning, not 100% of the memory

allocated to a buffer is actually available for use. 16 bytes of each buffer are reserved for variables needed to administer this buffer, such as various pointers etc.

Thus, if we requested (and received) a buffer with 2 pages (256 * 2 = 512), we actually have 496 bytes in which to store data (512 - 16).

- If you are *changing* the size of any buffer for a socket using sys.buffalloc, and this socket is not closed (sock.statesimple 505 is not PL_SSTS_CLOSED), the socket will be automatically closed. Whatever connection you had in progress will be discarded. The socket will not be closed if its buffer sizes remain unchanged.

Using Buffers in TCP Mode

Once you have allocated memory for the TX and RX buffers you can start sending and receiving data through them. Since TCP is a stream-oriented protocol this is what buffers store- a stream of data being sent and received, without any separation into individual packets. Even for the outgoing data, you have no control over how it will be split into packets for transmission over the network.

Sending Data

Sending data a two-step process. First, you put the data in the TX buffer using the sock.setdata 500 method, and then you perform the actual sending (commit the data) using the sock.send 500 method. For example:

```
sock.setdata ("Foo") ' Placed our data in the TX buffer - not being sent out
yet.
' ... more code...
sock.setdata ("Bar") ' Added even more data to the TX buffer, waiting to be
sent.
sock.send ' Now data will actually start going out. Data sent will be
'FooBar'.
```

Since this is a two-step process, you may gradually fill the buffer to capacity, and only then send its contents.

TiOS features *non-blocking operation*. This means that on sock.send, for example, the program does not halt and wait until the data is completely sent. In fact, execution resumes immediately, even before the first byte goes out. Your program will not freeze just because you ordered it to send a large chunk of data.

The data can be stored in the TX buffer at any time but it will only be sent out if and when the network connection 424 is established. Storing the data in the TX buffer won't cause the socket to establish any connection automatically.

Receiving Data

Receiving data is a one-step process. To extract the data from the RX buffer, use the sock.getdata 480 method. Data may only be extracted once from the buffer. Once extracted, it is no longer in the buffer. For example:

```
dim whatigot as string
whatigot = sock.getdata(255)
```

The string whatigot now contains up to 255 bytes of data which came from the RX buffer of the socket.

Discussion of TCP data RXing continues in Receiving Data in TCP mode 448.

Using Buffers in UDP Mode

UDP is a packet-based protocol (as opposed to TCP, which is stream-based). In UDP you usually care what data belongs to what datagram (packet). The sockets object gives you complete control over the individual datagrams you receive and transmit.

Sending and receiving UDP data is still effected through the TX and RX buffers. The difference is that the subdivision into datagrams is preserved within the buffers.

Each datagram in the buffer has its own header:

- For the TX buffer, headers contain datagram length as well as the destination MAC, IP, port, and broadcast flag (indicating whether to send the datagram as a broadcast).
- For the RX buffer, headers contain datagram length plus the sender's MAC, IP, port, and broadcast flag (indicating whether the datagram is a broadcast).

Sending Data

Just like with TCP, sending data through the TX buffer in UDP mode is a two-step process; first you put the data in the buffer using the sock.setdata 500 method, and then you close a datagram and perform the actual sending (commit the data) using the sock.send 500 method.

The datagrams will never be mixed with one another. Once you invoke sock.send, the datagram is closed and sent (as soon as possible). Any new data added to the TX buffer will belong to a new datagram. For example:

```
sock.setdata ("Foo") ' Placed our data in the tx buffer - not being sent out
yet.

' ... more code...

sock.setdata ("Bar") ' Added even more data to the same datagram, waiting to
be sent.
sock.send ' A datagram containing 'FooBar' will now be closed and committed
for sending.
sock.setdata ("Baf") ' This new data will go into a new datagram.
sock.send ' Closes the datagram with only 'Baf' in it and commits it for
sending.
sock.send ' sends an empty UDP datagram!
```

Notice that in the example above we were able to send out an empty datagram by using sock.send without sock.setdata!

Keep in mind that there is a limitation for the maximum length of data in the UDP

datagram- 1536 bytes.

Receiving Data

Receiving data in UDP mode requires you to be within an event handler for incoming socket data, or to explicitly move to the next UDP datagram in the buffer.

To extract the data from a buffer, use the sock.getdata 480 method. This method only accesses a single datagram on the buffer, unless you use the sock.nextpacket 487 method. If you have several incoming datagrams waiting, you will have to process them one by one, moving from one to the next. This is good because this way you know where one datagram ends and another one begins.

Here is an example:

```
sub on_sock_data_arrival
      dim whatigot as string
      whatigot = sock.getdata(255) 'will only extract the contents of a
single datagram. Reenter the on_sock_data_arrival to get the next datagram
end sub
```

Data may only be extracted once from a buffer. Once extracted, it is no longer in the buffer. Discussion of UDP data RXing continues in Receiving Data in UDP mode 450.

TX and RX Buffer Memory Status

You cannot effectively use a buffer without knowing what is its status. Is it overflowing? Can you add more data? etc. Thus, each of the socket buffers has certain properties which allow you to monitor it:

The RX buffer

You can check the total capacity of the buffer with the sock.rxbuffsize 498 property. You can also find out how much data the RX buffer currently contains with the sock.rxlen 499 property. From these two data, you can easily deduce how much free space you have in the RX buffer -- even though this isn't such a useful datum (that's one of the reasons there is no explicit property for it).

Note that sock.rxlen returns the *gross* current size of data in the RX buffer. In TCP mode, this is equivalent to the actual amount of data in the buffer. However, in UDP mode, this value includes the headers preceding each datagram within the RX buffer -- the amount of actual data in the buffer is smaller than that. A separate property -- sock.rxpacketlen 499 returns the length of actual data in the UDP datagram you are currently processing.

Sometimes you need to clear the RX buffer without actually extracting the data. In such cases the sock.rxclear 498 comes in handy.

The TX buffer

Similarly to the RX buffer, the TX buffer also has a sock.txbuffsize 509 property which lets you discover its capacity.

The TX buffer has two "data length" properties: sock.txlen 510 and sock.newtxlen 486. The *txlen* property returns the amount of *committed* data waiting to be sent from the buffer (you commit the data by using the sock.send 500 method). The *new*txlen property returns the amount of data which has entered the buffer, but has not yet

been committed for sending. For UDP, this is basically the length of UDP datagram being created.

The TX buffer also has a sock.txfree 510 property, which directly tells you how much space is left in it. This does not take into account uncommitted data in the buffer -- actual free space is sock.txfree-sock.newtxlen!

ser.txlen + ser.txfree = ser.txbuffsize.

When you want to clear the TX buffer without sending anything, use the sock.txclear 510 method. Notice, however, that this will only work when the network connection is closed (sock.statesimple 505 = PL_SSTS_CLOSED).

An example illustrating the difference between sock.txlen and sock.newtxlen:

```
sub on_sys_init

dim x,y as word ' declare variables

sock.rxbuffrq(1) ' Request one page for the RX buffer.
sock.txbuffrq(5) ' Request 5 pages for the TX buffer (which we will use).
sys.buffalloc ' Actually allocate the buffers.

sock.setdata("foofoo") ' Set some data to send.
sock.setdata("bar") ' Some more data to send.
sock.send ' Start sending the data (commit).
sock.setdata("baz") ' Some more data to send.
x = sock.txlen ' Check total amount of data in the TX buffer.
y = sock.newtxlen  ' Check length of data not yet committed.

end sub 'Set up a breakpoint HERE.
```

Don't step through the code. The sending is fast -- by the time you reach x and y by stepping one line at a time, the buffer will be empty and x and y will be 0. Set a breakpoint at the end of the code, and then check the values for the variables (by using the watch 33).

Receiving Data in TCP Mode

We have already explained that RX and TX buffers operate differently in the TCP 445 and UDP 446 mode. This section explains how to receive data when the TCP protocol is used by the socket.

In a typical system, there is a constant need to handle an inflow of data. A simple approach is to use polling. You just poll the buffer in a loop and see if it contains any fresh data, and when fresh data is available, you do something with it. This would look like this:

```
sub on_sys_init

while sock.rxlen = 0
wend ' basically keeps executing again and again as long as sock.rxlen = 0
s = sock.getdata(255) ' once loop is exited, it means data has arrived. We
extract it.

end sub
```

This approach will work, but it will forever keep you in a specific event handler (such as on_sys_init 533) and other events will never get a chance to execute. This is an example of *blocking code* which could cause a system to freeze. Of course, you can use the doevents 87 statement, but generally we recommend you to avoid this blocking approach.

Since our platform is event-driven, you should use events to tell you when new data is available. There is an on_sock_data_arrival 489 event which is generated whenever there is data in the RX buffer:

```
sub on_sock_data_arrival

dim s as string
s = sock.getdata(255) ' Extract the data -- but in a non-blocking way.
' .... code to process data ....
end sub
```

The on_sock_data_arrival event is generated whenever there is data in the RX buffer, but only once. There are never two on_sock_data_arrival events (for the same socket) waiting in the queue. The next event is only generated after the previous one has completed processing, if and when there is any data available in the RX buffer.

This means that when handling this event, you don't have to get *all* the data in the RX buffer. You can simply handle a chunk of data and once you leave the event handler, a new event of the same type will be generated.

Here is a correct example of handling arriving socket data through the on_sock_data_arrival event. This example implements a data loopback -- whatever is received by the socket is immediately sent back out.

```
dim rx_tcp_packet_len as word 'to keep the size of the next incoming TCP
packet

sub on_sys_init
end sub
```

We want to handle this loopback as efficiently as possible, but we must not overrun the TX buffer. Therefore, we cannot simply copy all arriving data from the RX buffer into the TX buffer. We need to check how much free space is available in the TX buffer. The first line of this code implements just that: sock.getdata takes as much data from the RX buffer as possible, but not more than sock.txfree (the available room in the TX buffer). The second line just sends the data.

Actually, this call will handle no more than 255 bytes in one pass. Even though we seemingly copy the data directly from the RX buffer to the TX buffer, this is done via a temporary string variable automatically created for this purpose. In this platform, string variables cannot exceed 255 bytes.

Receiving Data in UDP Mode

In the previous section[448] we have already explained how to handle data reception when the socket is in the TCP mode. This section provides explanation for receiving data with UDP.

Using on_sock_data_arrival event

With UDP, you still (typically) process incoming data basing on the on_sock_data_arrival[489] event. Two differences apply:

- Each time the on_sock_data_arrival event handler is entered you only get to process a single UDP datagram waiting in the RX buffer. Unless you use a special method (see below), you won't be able to get the data from the next datagram, even if this datagram is already available in the RX buffer.
- If, while within the on_sock_data_arrival event handler, you don't read out entire contents of the "current" datagram and exit the event handler, then the unread portion of the datagram is discarded.

Here is an example: supposing, two datagrams are waiting in the RX buffer and their contents are "ABC" and "123". The following code will then execute twice:

```
sub on_sock_data_arrival
dim s as string(2)
s = sock.getdata(255) ' will get 2 bytes as this is the capacity of s
end sub
```

Since s has a maximum capacity of 2 the first time s=sock.getdata(255) executes you will get "AB". When the event handler is exited the rest of the first UDP datagram will be discarded, so next time you will get "12"!

Using sock.nextpacket method

Using on_sock_data_arrival event is a preferred method of handling an incoming data stream. Still, in selected cases you may need to process RX data in a loop, without leaving the event handler.

The sock.nextpacket[487] method exists for just such a case. The result of this method execution is equivalent to exiting/(re)entering the on_sock_data_arrival event handler: the unread portion of the previous UDP datagram is discarded and we move to the next UDP datagram (if any).

Here is a code example where we handle all incoming UDP data without exiting event handler:

```
dim s as string

...

l1:
while sock.rxlen = 0
      doevents 'good practice: let other events execute while we are waiting
wend

sock.nextpacket 'Now we know that we do have data, 'move to' the next UDP
datagram. This is like entering the on_sock_data_arrival.
```

```
s = sock.getdata(255) 'get data
goto l1

...
```

Sending TCP and UDP Data

In the previous sections, we have explained how to handle an incoming stream of data. You could say it was incoming-data driven. Sometimes you need just the opposite -- you need to perform operations based on the sending of data.

Sending data with on_sock_data_sent event

Supposing that in a certain system, you need to send out a long string of data when a button is pressed. A simple code for this would look like this:

```
sub on_button_pressed
      sock.setdata("This is a long string waiting to be sent. Send me
already!")
      sock.send
end sub
```

The code above *would* work, but *only* if at the moment of code execution the necessary amount of free space was available in the TX buffer (otherwise the data would get truncated). So, obviously, you need to make sure that the TX buffer has the necessary amount of free space before sending. A simple polling solution would look like this:

```
sub on_button_pressed
      dim s as string
      s = "This is a long string waiting to be sent. Send me already!"
      while sock.txfree < len(s)
      wend
      sock.setdata(s)
      sock.send
end sub
```

Again, this is not so good, as it would block other event handlers. So, instead of doing that, we would employ a code that uses on_sock_data_sent 489:

```
dim s as string
s = "This is a long string waiting to be sent. Send me already!"

sub on_button_pressed
      sock.notifysent(sock.txbuffsize-len(s)) ' causes the on_sock_data_sent
event to fire when the TX buffer has space for our string
end sub

sub on_sock_data_sent
      sock.setdata(s)   ' put data in TX buffer
      sock.send  ' start sending it.
```

```
end sub
```

When we press the button, on_button_pressed 234 event is generated, so now the system knows we have a string to send. Using sock.notifysent 487 we make the system fire the on_ser_data_sent event when the necessary amount of free space becomes available. This event will only be fired once -- and will be fired immediately if there is already enough available space.

Within the event handler for this event, we put the data in the TX buffer and start sending it.

- Amount of data that will trigger on_sock_data_sent does not include uncommitted data in the TX buffer.

UDP datagrams are generated as you create them

In Using Buffers in TCP Mode 446 we have already explained that for UDP you have a complete control over how the data you are sending is divided into the UDP datagrams. Each time you use the sock.send method you draw the boundary between the datagrams- the previous one is "closed" and the new one "begins". You can even send out an empty UDP datagram by executing sock.send 500 without using the sock.setdata 500 first.

Correctly responding to the sender of each UDP datagram

With UDP, your socket may be receiving UDP datagrams from several different hosts. When the on_sock_data_arrival event handler is entered (see Receiving Data in UDP mode 450) the following properties automatically reflect the source of the current datagram- sock.remotemac 495, sock.remoteip 494, and sock.remoteport 495. Additionally, the sock.bcast 475 property will tell you whether the datagram was a regular or a broadcast one (this material has already been covered in Checking Connection Status 439).

Additionally, any datagram that was generated and sent from within the on_sock_data_arrival event handler will be send to the sender of the datagram for processing of which the on_sock_data_arrival event handler was entered.

The point is that if you are sending out a datagram from within the on_sock_data_arrival event handler you are automatically replying to the sender of the datagram being processed. The following example sends back "GOT DATAGRAM FROM *xxx.xxx.xxx.xxx*" string in response to any datagram received by the socket:

```
sub on_sock_data_arrival
      sock.setdata("GOT DATAGRAM FROM " + sock.remoteip)
      sock.send
end sub
```

For the above example, even if several hosts send the datagrams to the socket at the same time each one of these hosts will get a correct reply back!

Now consider this example: each time the button is pressed the same message is generated:

```

```

```
sub on_button_pressed
      sock.setdata("GOT DATAGRAM FROM " + sock.remoteip)
      sock.send
end sub
```

The difference is that when you press the button the datagram will be sent to the destination, from which the most recent incoming UDP datagram was received (and accepted) by the socket!

"Split Packet" Mode of TCP Data Processing

Though our customer's feedback we have learned that sometimes it may be necessary to know the size of individual TCP packets. For example, as it turns out, some data encryption methods work on a packet level. Erase the border between TCP packets -- and you can't decrypt the data.

Introduced in Tibbo Basic V2.0, new sock.splittcppackets 501 property and on_sock_tcp_packet_arrival 491 event allow you to process incoming TCP data packet by packet. To achieve this, set the sock.splittcppacket= 1- YES. After that, the on_sock_tcp_packet_arrival will be generated for each incoming TCP packet carrying any new data (i.e., not a retransmission data). Here is an example of use:

```
dim rx_tcp_packet_len as word 'to keep the size of the next incoming TCP
packet

sub on_sys_init
   ...
   rx_tcp_packet_len=0
   sock.splittcppackets=YES
end sub

sub on_sock_tcp_packet_arrival(len as word)
   rx_tcp_packet_len=len 'we get the size of the next packet we are going to
process
end sub

sub on_sock_data_arrival
   dim s as string
   'take exactly the contents of one packet (we assume that it can fit in
the string!)
   s=sock.getdata(rx_tcp_packet_len)
   rx_tcp_packet_len=0 'very important!
end sub
```

Naturally, you may also need to control the size of outgoing TCP packets. This is done in a different way. With sock.splittcppackets 501 = 1- YES, when you put some data into the TX buffer and execute sock.send 500, the socket won't send this data out unless entire contents of the TX buffer can be sent out in a single TCP packet. Here is an example of how you can use that:

```
sock.splittcppackets=YES
...
...
sock.setdata("ABCDEFGH1234567890")
sock.send
```

```
while sock.txlen>0
   doevents
wend
```

Notice that this method has its disadvantage -- your data throughput is diminished because your program is seeking an acknowledgement for each TCP packet being sent. On the other hand, this is a bulletproof way of making sure that the outgoing TCP packet contains exactly the data you intended.

Be careful not to try to send more data than the RX buffer size on the other end. Since in this mode the socket won't send that data unless it can send all of it, your TCP connection will get stuck!

Also, attempting to send the packet with size exceeding the "maximum segment size" (MSS) as specified by the other end will lead to data fragmentation! The socket will never send any TCP packet with the amount of data exceeding MSS.

Handling Buffer Overruns

Handling RX buffer overruns

The on_sock_overrun event 491 is generated when an RX buffer overrun has occurred. It means the data has been arriving to the RX buffer faster than you were handling it and that some data got lost.

This event is generated just once, no matter how much data is lost. A new event will be generated only after exiting the handler for the previous one.

Normally, data overruns will not occur when the TCP transport protocol is used. This is because the TCP is intelligent enough to regulate the flow of data between the sender and the receiver and, hence, avoid overruns. The UDP protocol does not have a "flow control" mechanism and RX buffer overruns can and will happen.

Typically, the user of your system wants to know when an overrun has occurred. For example, you could blink a red LED when this happens.

```
sub on_ser_overrun
      pat.play("R-R-R-R")
end sub
```

Are TX buffer overruns possible?

TX buffer overruns are not possible. The socket won't let you overload its TX buffer. If you try to add more data to the TX buffer than the free space in the buffer allows to store then the data you are adding will be truncated.

See Sending Data 451 for explanation on how to TX data correctly.

Redirecting Buffers

The following example appeared under Receiving Data in TCP Mode 448:

```
sub on_sock_data_arrival
      sock.setdata(sock.getdata(sock.txfree))
      sock.send
end sub
```

This example shows how to send all data incoming to the RX buffer out from the TX buffer, in just two lines of code. However fast, this technique still passes all data through your BASIC code, even though you are not processing (altering, sampling) it in any way.

A much more efficient and advanced way to do this would be using a technique called *buffer redirection* (buffer shorting). With buffer shorting, instead of receiving the data into the RX buffer of your socket, you are receiving it directly into the TX buffer of another object which is supposed to send out this data. This can be a socket (same or different one), a serial port object, etc.

To use buffer shorting, you invoke the sock.redir 493 method and specify the buffer to which the data is to be redirected. Once this is done, the on_sock_data_arrival event won't be generated at all, because the data will be going directly to the TX buffer that you have specified. As soon as the data enters this buffer, it will be automatically committed for sending. Here is an example:

```
sub on_sys_init
      sock.num=0
      sock.redir(PL_REDIR_SOCK0) ' This is a loopback for socket 0.
end sub
```

The performance advantage here is enormous, due to two factors: first, you are not handling the data programmatically, so the VM isn't involved at all. And second, the data being received is received directly into the TX buffer from which it is transmitted, so there is less copying in memory.

Of course you cannot do anything at all with this data -- you are just pumping it through. However, very often this is precisely what is needed! Additionally, you *can* still process inband commands/replies 456 (messages).

To stop redirection, you can use sock.redir(0), which means "receive data into the RX buffer of the socket in a normal fashion".

Redirection and UDP

Using Buffers in UDP Mode 446 explained that the sock object preserves buffer boundaries when storing UDP datagrams in RX and TX buffer. To achieve this, the sock object uses special datagram headers that are also stored in these buffers. This means that the buffers contain not only data, but also an additional "service" information.

When an RX buffer of a "UDP socket" (i.e. a socket running in the UDP mode) is redirected to a TX buffer of another UDP socket, datagram boundaries are preserved. The receiving socket will still send out the data subdivided into the original datagrams.

When an RX buffer of a UDP socket is redirected to a TX buffer of a TCP socket or serial port, the service information is removed and datagram boundaries are dissolved. To the receiving TCP socket or serial port, the data appears to be a stream.

Sinking Data

Sometimes it is desirable to ignore all incoming data while still maintaining the connection opened. The sock.sinkdata 501 property allows you to do just that.

Set the sock.sinkdata to 1- YES, and all incoming data will be automatically discarded. This means that the on_sock_data_arrival 489 event will not be generated, reading sock.rxlen 499 will always be returning zero, and so on. No data will be reaching its destination even in case of buffer redirection 454. Inband commands 456, however, will still be extracted from the incoming data stream and processed. Sock.connectiontout 477 and sock.toutcounter 507 will work correctly as well.

Working With Inband Commands

Inband commands (replies) are messages embedded within the TCP data stream. Each inband message has a specific formatting that allows the socket to recognize it in the data stream being received.

Inband messages have many uses. For example, in a network-to-serial converter, you typically pass all serial data through a single TCP connection. At the same time, you often need to send some control commands to the device, i.e. for setup, etc. This can be done in two ways: through a separate network connection ("out-of-band" respective to the main data connection) or by embedding those commands inside the serial data stream ("inband" way). The second method is sometimes better, since very often you want to avoid or cannot have another network connection to your device.

Of course, you could just write a BASIC code that would be separating inband commands from the data but this would affect the performance (data throughput) of your device considerably. The sock object natively supports inband commands to avoid this performance penalty. In fact, inband commands will work even when the buffer redirection 454 is enabled!

Inband messages are only possible with TCP transport protocol. You cannot use inband messages with UDP!

Inband Message Format

Inband message passing is enabled through the sock.inbandcommands 484 property.

Each inband message has to start with a special escape character whose ASCII code is specified by the sock.escchar 479 property. The next character after the escape character can have any ASCII code *except* for the code of the escape character itself.

Following that is the body of the inband message. The last character in the message is a so-called end character, specified by the sock.endchar 479 property. It signals the end of the inband message and return to the "regular" data.

There are no specific limitations on how long the inband message can be. The length is only limited by how much space you allocate for the CMD and RPL buffers 457 that store incoming and outgoing inband messages.

And what if the data stream itself contains a character(s) with the ASCII code of the escape character you have set? Wouldn't this confuse the socket into thinking that this is the beginning of inband command? To avoid this situation, the *data* character with code of escape character is transmitted as two identical characters with the same ASCII code.

Example: supposing you have the following setup:

```
...
sock.inbandcommands= YES
sock.escchar=`$`
sock.endchar=`%`
...
```

Now, you have the following data stream coming into the socket:

ABCD$#inband commqand%EFG$$123

The socket will interpret this stream as including one inband command: "$#inband command%". Regular data, placed into the RX buffer of the socket will be "ABCDEFG$123". The first '$' character is interpreted as the beginning of the inband message because this character is followed by some other character ('#'). The second occurrence of the '$' character is interpreted as data, since this character is followed by another 'S' character. Resulting data stream contains only a single '$' character- the socket takes care of removing the second one automatically.

When sending data from the TX buffer, the socket also automatically doubles all data characters with the ASCII code of the escape character. So, if you want to send this string: "Outbound$!" what will actually be sent is:

Outbound$$!

Inband messages are not our invention. Many programs, such as the HyperTerminal, treat the character with code 255 as an escape character.

Inband-related Buffers (CMD, RPL, and TX2)

Three buffers are required for inband message processing. On startup, these buffers are not allocated any memory, so you have to do it if you are planning to send and receive inband messages.

- The CMD buffer- used to store incoming inband messages (we call them "inband commands"). Use the sock.cmdbuffrq 476 method to allocate memory for this buffer. Usually, inband commands are not very long so allocating a minimum space of 1 page is typically sufficient.
- The RPL buffer- used to store outgoing inband messages (we call them "inband replies"). Use the sock.rplbuffrq 496 method to allocate memory for this buffer. again, inband commands are usually not very long so allocating a minimum space of 1 page will probably be sufficient.
- The TX2 buffer- used internally by the socket when inband commands are enabled. You don't have to do anything with this buffer other than allocate memory for it. We recommend allocating as much space as you did for the TX buffer. Allocation is requested through the sock.tx2buffrq 508 method.

Actual memory allocation is done through the sys.buffalloc 530 method which applies to all buffers previously specified. Here is an example:

```
dim b1, b2, b3, b4, b5 as byte

...
'setup for other objects, sockets, etc.

b1= sock.txbuffrq(5) 'you need this buffer to send regular data
b2= sock.rxbuffrq(5) 'you need this buffer to receive regular data
b3= sock.cmdbuffrq(1) 'buffer for incoming inband commands
b4= sock.rplbuffrq(1) 'buffer for outgoing inband replies
b5= sock.tx2buffrq(5) 'same buffer size as for the TX buffer

....

sys.buffalloc ' Performs actual memory allocation, as per previous requests
```

Actual memory allocation takes up to 100ms, so it is usually done just once, on boot, for all required buffers.

You may not always get the full amount of memory you have requested. Memory is not an infinite resource, and if you have already requested (and received) allocations for 95% of the memory for your platform, your next request will get up to 5% of memory, even if you requested for 10%.

There is a small overhead for each buffer. Meaning, not 100% of the memory allocated to a buffer is actually available for use. 16 bytes of each buffer are reserved for variables needed to administer this buffer, such as various pointers etc.

Thus, if we requested (and received) a buffer with 2 pages (256 * 2 = 512), we actually have 496 bytes in which to store data (512 - 16).

- If you are *changing* the size of any buffer for a socket using sys.buffalloc, and this socket is not closed (sock.statesimple 505 is not PL_SSTS_CLOSED), the socket will be automatically closed. Whatever connection you had in progress will be discarded. The socket will not be closed if its buffer sizes remain unchanged.

Processing Inband Commands

What goes into the CMD buffer

All incoming inband commands are stored into the CMD buffer. At any given time the buffer may contain more than one command. Each inband message in the buffer already has its escape character and the character after the escape character removed. The end character, however, is not removed and can be used by your program to separate inband messages from each other.

Here is an example. Supposing, you have the following setup:

```
...
sock.inbandcommands= YES
sock.escchar=`@`
sock.endchar=`&`
...
```

Here is a sample data stream:

Data@command1&Moredata@!command2&Evenmoredata

This incoming data stream will have the following effect:

- The RX buffer will receive this data: "DataMoredataEvenmoredata".
- The CMD buffer will receive: "ommand1&command2&"

Notice, that the first inband command is missing the first character- this is because when the inband command is being processed both its escape character and the character following the escape character are removed. End characters of both inband commands are preserved so you can tell where each one ends.

Extracting the data from the CMD buffer

Extracting data from the CMD buffer is similar to extracting data from the RX buffer. A dedicated method- sock.getinband 481- does the job. This method is just like the sock.getdata 480, minus the maxinplen argument. Total amount of data in the CMD buffer can be checked through the sock.cmdlen 477 property.

Once extracted, the data is no longer in the buffer. You can use the sock.cmdlen 477 property to check how much data is waiting in the CMD buffer. There is no dedicated property to tell you the buffer capacity- just remember what the sock.cmdbuffrq 476 method returned if you have to know this!

The on_sock_inband 490 event is generated whenever there is some data in the CMD buffer, but only once. There are never two on_sock_inband events waiting in the queue. The next event is only generated after the previous one has completed processing, if and when there is any data available in the CMD buffer.

Inband commands only appear in the CMD buffer in their entirety. That is, if the buffer was previously empty and you get the on_sock_inband event then you are guaranteed that the buffer will contain a full command (or several full commands).

Here is an example:

```
sub on_sock_inband
      dim s as string 'we will keep the data from the CMD buffer here
      dim s2 as string 'this will keep individual inband commands
      dim x as byte

      s=sock.getinband 'we get entire CMD buffer contents into the s
      x=instr(1,s,chr(sock.endchar))
      while x<>0
             s2=left(s,x-1)  's2 now contains a single inband command
             s=right(s,len(s)-x)  'cut out this command

             'process inband command in the s2 as needed
             ...
             ...

             'any more inband commands to process now?
             x=instr(1,s,chr(sock.endchar))
      wend
end sub
```

- For the above example to work well, the size of the CMD buffer must not exceed the capacity of string variable s. This way whatever is extracted from the CMD buffer will always fit in s. A slightly more complex processing is needed if the buffer is larger than the capacity of s.

Are CMD buffer overruns possible?

CMD buffer overruns are not possible. If the socket receives an inband command that cannot be saved into the CMD buffer in its entirety, then the socket will discard the whole command (your program won't be notified of this in any way). Therefore, you are guaranteed to always receive complete inband commands, or nothing at all.

Incomplete inband commads

Take a look at this datastream:

Data@!comma@!command2&Moredata

What we have here is an inband command that is incomplete- a new inband command starts before the previous one ends. Such incomplete commands are discarded and not recorded into the CMD buffer.

Sending Inband Replies

How to generate inband reply correctly

Inband replies are sent from the RPL buffer. Unlike the process of sending "regular" TX data that requires you to use sock.setdata 500 and sock.send 500 methods, the process of sending an inband reply only takes one step. You set and send (commit) the data with a single method- sock.setsendinband 500.

The sock.setsendinband method puts the data into the RPL buffer and immediately commits it for sending. The socket does not add necessary incapsulation automatically: it is the responsibility of your application to add the escape character, some other character after the escape, and the end character.

Your inband reply will only be stored into the RPL buffer if the latter has enough space to store your entire message. If there is not enough free space then *nothing* will be stored. This is different from the TX buffer, for which *whatever can fit* is stored. You can check the free space in the RPL buffer by using the sock.rplfree 497 property. Amount of unsent data in the RPL buffer can be checked through the sock.rpllen 497 property.

You must not split your inband reply- it must be placed in the RPL buffer with a single invocation of sock.setsendinband. In the example below we reply back "OK" to each inband command we receive:

```
sub on_sock_inband
      dim s as string 'we will keep the data from the CMD buffer here
      dim s2 as string 'this will keep individual inband commands
      dim x as byte
```

```
      s=sock.getinband 'we get entire CMD buffer contents into the s
      x=instr(1,s,chr(sock.endchar))
      while x<>0
            s2=left(s,x-1)  's2 now contains a single inband command
            s=right(s,len(s)-x)  'cut out this command

            'reply back with OK
            sock.setsendinband(chr(sock.escchar)+" OK"+chr(sock.endchar))

            'any more inband commands to process now?
            x=instr(1,s,chr(sock.endchar))
      wend
end sub
```

And this would be an incorrect way- *do not* split inband replies!

```
...
sock.setsendinband(chr(sock.escchar))     'WRONG!
sock.setsendinband(" OK")                 'WRONG!
sock.setsendinband(chr(sock.endchar))     'WRONG!
...
```

- For the above example to work well, the size of the CMD buffer must not exceed the capacity of string variable s. This way whatever is extracted from the CMD buffer will always fit in s. A slightly more complex processing is needed if the buffer is larger than the capacity of s.
- Notice how we have created a character after the escape character- by adding a space in front of our "OK" reply, like this: " OK". This will work fine as long as our escape character is not space!

Using HTTP

The sock object can function as a HTTP server. This means that when certain conditions are met, individual sockets will switch into the HTTP mode and output the data in a style, consistent with the HTTP server functionality.

Certain BASIC information about the HTTP server has already been provided in Working with HTML 79 and Embedding Code Within an HTML File 80.

When the socket is in the HTTP mode your program has no control over the received data (HTTP requests) and only *sometimes* has control over the transmitted data (HTTP reply).

In the simplest case the file returned to the web browser is static- a "fixed" HTML page, a graphic, or some other file. Processing of such a static file requires no intervention from your program whatsoever. Just setup the socket(s) to be able to accept HTTP requests and the sock object will take care of the rest.

More often than not, however, you have to create a dynamic HTML page. Dynamic pages include fragments of BASIC code. When the sock object encounters such a fragment in the file being sent to the browser, it executes the code. This code, in turn, performs some action, for example, generates and sends some dynamic data to the browser or jumps to the other place in the HTML file.

The HTTP server built into the sock object understands two request types- GET and POST. Both can carry "HTTP variables" that the server will extract and pass to the BASIC code.

At the moment, the following file types are explicitly supported:

- HTML- can be static or include BASIC code.
- TXT- plain text, no BASIC code can be included.
- JPG and GIF- graphic files.
- SWF flash files.

All files other than HTML files are static and are sent to the browser "as is". There is, however, a method of programmatic generation of such files -- see URL Substitution 468.

All other file types are handled as binary files.

Currently, the socket object can only access first 65534 bytes of each file, even if the actual file is larger! Make sure that all HTML files (and other files that will be returned by the built in server of your device) are not larger than 65534 bytes. This is not to be confused with the size of HTTP output generated by HTTP file. A very large output can be generated by a small HTML file (due to dynamic data generation)- and that is OK.

65534 is actually the size limitation for the *compiled* HTML file. When compiling your project, the TIDE will separate the static portion of the file from the Tibbo Basic code fragments. Only the compiled file size matters.

HTTP-related Buffers

For the HTTP to work, you need to allocate some memory to the following buffers:

- RX buffer. This buffer will be receiving HTTP requests from the client (browser). Buffer allocation request is done through the sock.rxbuffrq 497 method. It is possible to avoid spending memory on the RX buffer by redirecting 454 this buffer to the TX buffer of the same socket. This will work because the web server operation is strictly sequential -- receive a request, then generate a reply. Requests and replies do not have to be stored concurrently, so one buffer is sufficient and you will save some memory!
- TX buffer. This buffer will be handling web server replies. If you enable redirection it will also receive HTTP requests. Buffer allocation request is done through the sock.txbuffrq 509 method. Practical experience shows that allocating just one page for this buffer makes HTTP request handling somewhat slow. At three or four pages, there is a significant performance improvement. Additional buffer pages do not lead to any dramatic improvements in performance.
- VAR buffer. Dynamic HTML pages include snippets of BASIC code and this code may need to know variable passed to the HTTP server using GET or POST methods. The VAR buffer stores those variables. Do the allocation with the sock.varbuffrq 511 method. Buffer size of one page is usually OK. If the application is to handle large amounts of variable data, such as in the case of file uploads, you can improve the performance by allocating more pages.

Actual memory allocation is done through the sys.buffalloc 530 method which applies to all buffers previously specified. Here is an example:

```
Dim f As Byte
```

```
...
'setup for other objects, sockets, etc.

'setup buffers of sockets 8-15 (will be used for HTTP)
For f=8 To 15
   sock.num=f
   sock.txbuffrq(1) 'you need this buffer for HTTP requests and replies
   sock.varbuffrq(1) 'you need this buffer to get HTTP variables
   sock.redir(PL_REDIR_SOCK0 + sock.num) 'this will allow us to avoid
wasting memory on the RX buffer
Next f

....

sys.buffalloc ' Performs actual memory allocation, as per previous requests.
```

Memory allocation takes up to 100ms, so it is usually done just once, on boot, for all required buffers.

You may not always get the full amount of memory you have requested. Memory is not an infinite resource, and if you have already requested (and received) allocations for 95% of the memory for your platform, your next request will get up to 5% of memory, even if you requested for 10%.

There is a small overhead for each buffer. Meaning, not 100% of the memory allocated to a buffer is actually available for use. 16 bytes of each buffer are reserved for variables needed to administer this buffer, such as various pointers etc.

Thus, if we requested (and received) a buffer with 2 pages (256 * 2 = 512), we actually have 496 bytes in which to store data (512 - 16).

- If you are *changing* the size of any buffer for a socket using sys.buffalloc, and this socket is not closed (sock.statesimple 505 is not PL_SSTS_CLOSED), the socket will be automatically closed. Whatever connection you had in progress will be discarded. The socket will not be closed if its buffer sizes remain unchanged.
- Notice, that in most cases you will need to reserve more than one socket for HTTP. The HTTP server may need to service multiple requests from different computers at the same time. Even for a single computer and a single HTML page, more than one socket may be needed. For example, if your HTML page contains a picture, the browser will establish two parallel connections to the sock object- one to get the HTML page itself, another one- to get the picture. We recommend that you reserve 4-8 sockets for the HTTP. It is better to have less buffer memory for each HTTP sockets than to have fewer HTTP sockets!

Setting the Socket for HTTP

How to set the socket for HTTP

Apart from assigning some memory to the TX, RX, and VAR buffers, the following needs to be done to make the socket work in HTTP mode:

- The protocol must be TCP (sock.protocol 492 = 1- PL_SOCK_PROTOCOL_TCP).
- The socket must be open for incoming connections (typically, from anybody: sock.inconmode 485 = 3- PL_SOCK_INCONMODE_ANY_IP_ANY_PORT).

- Reconnects should not be enabled- this is counter-productive for HTTP (sock.reconmode 492 = 0- PL_SOCK_RECONMODE_0). Reconnects are disabled by default so just leave it this way.
- Correct listening HTTP port must be set. Default HTTP port on all servers is 80 (sock.httpportlist 483 ="80").

In the previous topic, we have already explained that your system should reserve *several* HTTP sockets. Here is a possible initialization example:

```
dim f as byte

...
'setup for other sockets, etc.

'setup sockets 8-15 for HTTP
for f=8 to 15
      sock.num=f
      sock.protocol= PL_SOCK_PROTOCOL_TCP
      sock.inconmode= PL_SOCK_INCONMODE_ANY_IP_ANY_PORT
      sock.httpportlist="80"
next f
....
```

To make sure that the HTTP is working you can create and add to the project a simple static HTTP file. Call this file <index.html>- this is a default file that will be called if no specific file is requested by GET or POST. Here is a static file example:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD W3 HTML//EN">
<HTML>
<BODY>
      HELLO WORLD!<br>
</BODY>
</HTML>
```

Launch the browser, type the IP-address of your device, for example: "http://192.168.1.95" and you will get the output "HELLO WORLD!".

Don't forget to give your device an IP address. For example, if you are working through a regular wired Ethernet, you should assign an IP address through the net.ip 360 property.

Double-duty: non-HTTP and HTTP processing on the same socket

Your HTTP sockets don't have to be *exclusively* HTTP. You can have them behave differently depending on which listening port the TCP connection is being made to. Here is an example: supposing the setup of your device needs to be effected in two ways- via TELNET or via HTTP. Standard TELNET port is 23, standard HTTP port is 80. Setup your socket like this:

```
sock.num=f
```

```
sock.protocol= PL_SOCK_PROTOCOL_TCP
sock.inconmode= PL_SOCK_INCONMODE_ANY_IP_ANY_PORT
sock.localportlist="23"
sock.httpportlist="80"
....
```

The socket will now be accepting connections both on port 23 and port 80. When connection is made to port 23, the socket will work as a regular data socket, as was described in previous sections. When connection is made to port 80, the socket will automatically switch into the HTTP mode!

There is a property- sock.httpmode 481- that tells you which mode the socket is in- regular data mode or HTTP. You can even "forcefully" switch any TCP connection into the HTTP mode by setting sock.httpmode= 1- YES. You cannot, however, switch this connection back to the data mode, it will remain in the HTTP mode until termination.

Socket Behavior in the HTTP Mode

When in the HTTP mode, the socket is behaving differently compared to the normal data mode.

Incoming connection rejection

As was explained in Accepting Incoming Connections 426, if your device "decides" to reject an incoming TCP connection, it will send out a reset TCP packet. This way, the other host is instantly notified of the rejection. Rules are different for HTTP sockets:

- If there is an incoming TCP connection to the web server of the device (incoming HTTP connection request), and if your application has one or more sockets that are configured to accept this connection, and if all such sockets are already occupied, then the system will not reply to the requesting host *at all*.
- If there is an incoming HTTP connection request, and if your application has no HTTP sockets configured to accept this connection, then the system will still respond with a reset.

This behavior allows your application to get away with fewer HTTP sockets. Here is why. If all HTTP sockets are busy and your application sends out a reset, the browser will show a "connection reset" message. If, however, your device does not reply at all, the browser will wait, and resend its request later. Browsers are "patient" -- they will typically try several times before giving up. If any HTTP sockets are freed-up during this wait, the repeat request will be accepted and the browser will get its page. Therefore, very few HTTP sockets can handle a large number of page requests in a sequential manner and with few rejections.

Other differences

- All incoming data is still stored in the TX buffer (yes, TX buffer). This data, however, is not passed to your program but, instead, is interpreted as HTTP request. This HTTP request must be properly formatted. The sock object supports GET and POST commands.
- The RX buffer is not used 462 at all and does not have be allocated any memory.
- GET and POST commands can optionally contain "request variables". These are stored into the VAR buffer from which your program can read them out later.
- No on_sock_data_arrival 489 event is generated when the HTTP request string is

received into the RX buffer.

- Once entire request has been received the socket prepares and starts to output the reply. Your program has no control over this output until a BASIC code fragment is encountered in the HTTP file. The on_sock_data_sent 489 event cannot be used as well.
- When code fragment is encountered in the HTTP file control is passed to it and then your program can perform desired action, i.e. generate some dynamic HTML content, etc. When this fragment is entered, the sock.num 488 is automatically set to the correct socket number.
- Once HTTP reply has been sent to the client the socket will automatically close the connection, as is a normal socket behavior for HTTP. A special property- sock.httpnoclose 482- allows you to change this default behavior and leave the connection opened.

Including BASIC Code in HTTP Files

To create dynamic HTML pages, you include BASIC statements directly into the HTML file. A fragment of BASIC code is included within "<? ?>" encapsulation, as shown in the example below (for more info see Working with HTML 79):

```
<!DOCTYPE HTML public "-//W3C//DTD W3 HTML//EN">
<HTML>
<BODY>
      HELLO HTML WORLD!<br>
      <?
            'This is a BASIC code snippet. We can, for instance, send
something to the serial port
            ser.num=0
            ser.setdata("HELLO SERIAL WORLD TOO!")
            ser.send
      ?>
</BODY>
</HTML>
```

Each time this HTML page will be requested by the browser, the "HELLO SERIAL WORLD TOO!" string will be sent out of the serial port.

Generating Dynamic HTML Pages

How to "print" dynamic HTML data

In most cases the BASIC code included into the HTML page is used to generate dynamic HTML content, i.e. to send some dynamically generated data to the browser. This is done in a way, similar to sending out data in a regular data mode: you first set the data you want to send with the sock.setdata 500 method, then commit this data for sending using sock.send 500 method.

Here is an example HTML page that checks the state of one of the inputs of the I/O object and reports this state on the HTML page:

```
<!DOCTYPE HTML public "-//W3C//DTD W3 HTML//EN">
<HTML>
<BODY>
      The state of line 0 is:!<br>
      <?
            'This is a BASIC code fragment. Get the state of the line.
            io.num=0
            if io.state= LOW then
                  sock.setdata("LOW")
            else
                  sock.setdata("HIGH")
            end if
            sock.send
      ?>
</BODY>
</HTML>
```

- Notice how we did not have to use sock.num 488 - it is set automatically when control is passed to BASIC!

Be careful not to get your data truncated!

The above example is seemingly correct and in this particular case will work fine- because there is not much static data preceding the point where we generate dynamic content and not much dynamic content is generated as well. Generally speaking, you cannot expect that the TX buffer will have enough space when you need to put some data into it! If you are not careful the dynamic data you want to generate may get truncated!

To avoid this situation always check if the necessary amount of free space is available before attempting to put it into the TX buffer. This statement is true for the normal data mode as well, not just for the HTTP processing- we have already touched on this subject in Sending Data 451. What is different for HTTP is that you cannot use the on_sock_data_sent 489 event to "call you back" when the TX buffer frees up!

The only solution in case of HTTP is to wait, in a loop, for the desired amount of space to become available. Naturally, we don't want to be blocking the whole device, so "polite" waiting shall include a doevents 87 statement:

```
<?
      dim s as string(4)
?>

<!DOCTYPE HTML public "-//W3C//DTD W3 HTML//EN">
<HTML>
<BODY>
      The state of line 0 is:!<br>
      <?
            'This is a BASIC code snippet. Get the state of the line.
            io.num=0
            if io.state= LOW then
                  s="LOW"
            else
                  s=HIGH
            end if
```

```
            'and now we wait till the TX buffer has enough free space
            while sock.txfree<len(s)
                  doevents
            wend

            'OK, so necessary space is now available!
            sock.setdata(s)
            sock.send
      ?>
</BODY>
</HTML>
```

Now we have a bullet-proof dynamic content generation *and* non-blocking operation! As you get more experience with HTML, you will see that the doevents statement has to be used quite often.

Special case: doevents and concurrent request of the same file

With HTTP it is entirely possible that two computers (browsers) will request the same HTML file. If you have allocated more than one socket for HTTP it is also possible that both of those requests will be processed at the same time. Of course, the BASIC VM is a single-process system, so when it comes to processing dynamic part of the files, one of the requests will be first.

It is also entirely possible, that when executing the BASIC procedure from the first "instance" of the file, execution arrives at doevents and the second instance of the same file will start to process. After all, doevents allows other events to execute, so this can include activity in other sockets.

Now, if the same BASIC procedure was allowed to execute again this would cause recursion: an attempt to execute a portion of code while it is already executing! Recursion is not allowed 66, so the VM will not re-enter the code immediately. Instead, the second instance of HTML page will be "on hold" until the first instance finishes running the procedure in question. After that, the same procedure will be executed for the second instance. This behavior has been introduced in **V1.2** of TiOS. In the previous release, procedure execution for the second instance of the HTML page was simply skipped and not executed at all, which is wrong!

Once again, for this to occur the following conditions must be met: the HTML procedure in question must use doevents statement, and the same file (HTML page) must be requested simultaneously from several browsers.

Avoiding illegal characters

HTTP restricts the use of some characters- you cannot freely send any code from the ASCII table. When you use the sock.setdata and sock.send, you are "printing" data directly to the browser. It is your program's responsibility to avoid illegal characters and use "%*xx*" instead.

URL Substitution

HTML files can include BASIC code 466 and thus be dynamic 466. Actual HTML contents received by the browser may partially be generated by your Tibbo BASIC application. Files of other types -- plain text, graphics, etc. -- cannot include executable code and are static in nature. These files are sent to the browser "as is".

So, what if you need to generate such a non-HTTP file dynamically? For example, what if you want to generate a BMP file? The answer is in using "URL substitution". The sock.urlsubstitutes 511 property allows you to do this.

The property stored a list of comma-separated file names (with extensions). When the web server receives a request for a non-HTML file from the browser (say, "pix.bmp"), it first tries to find this file among the HTML and resource files of your project. Failing this, the web server looks at defined substitutions. If the requested file is not on the list, the web server returns the "404 error".

If the file is on the list of defined substitutions, the browser looks for the file with the same name but ".html" extension ("pix.html"). If there is no such file, the "404 error" is, again, the answer. If this HTML file exists the server outputs this file, but makes it look like it was a file of the original type (server processes "pix.html", brower gets "pix.bmp").

Since the actual behind-the-scenes output is done for the HTML file you can put your own code into that file and generate content dynamically. The following example shows how we generate the file "pix.bmp" dynamically. For this to work, we need to set sock.urlsubstitutes="pix.bmp" somewhere in the initialization section of the project. Here is what we have in "pix.html" file:

```
<?
   Dim x As Byte
   Dim s As String

   romfile.open("source.bmp")
   Do
      x=sock.txfree
      s=romfile.getdata(x)
      sock.setdata(s)
      sock.send
   Loop While Len(s)=x
?>
```

In this example we simply read and output the contents of another bmp file called "source.bmp". No value is added -- we could just access that graphical file directly. Your code, however, is free to do anything and substitution opens up the way to create your picture on the fly, or dynamically generate the content for other "static" files.

Working with HTTP Variables

HTML pages often pass "variables" to each other. For example, if you have an HTML form to fill on page "form.html" you may want to have the data input by the user on page "result.html".

The explanation of HTTP variables is divided into three sections:

- Simple case 470 -- the amount of HTTP variable data does not exceed 255 bytes;
- Complex case 471 -- the amount of HTTP variable data exceeds 255 bytes (can easily happen, especially with file uploads);
- Details 473 on how the variable data actually looks to your application depending on the method used (HTTP GET or POST).

In most cases the client will send a fairly small amount of variable data, so it all will fit in 255 bytes. Use the sock.httprqstring 483 R/O property, as shown in the following example. Say, you have a login to perform. The user enters his/her name and password on page "index.html". Actual login is processed on page "login.html". The data is passed between these two pages "invisibly", using the HTTP POST method.

Here is the file "index.html":

```
<html> <body>
      <form action="form_get.html" method="post" enctype="multipart/form-
data">
            username: <Input Type="text" name="user"><br>
            password: <Input Type="password" name="pwd"><br>
            <Input Type="submit" value="send">
      </form>
</body> </html>
```

And here is "login.html":

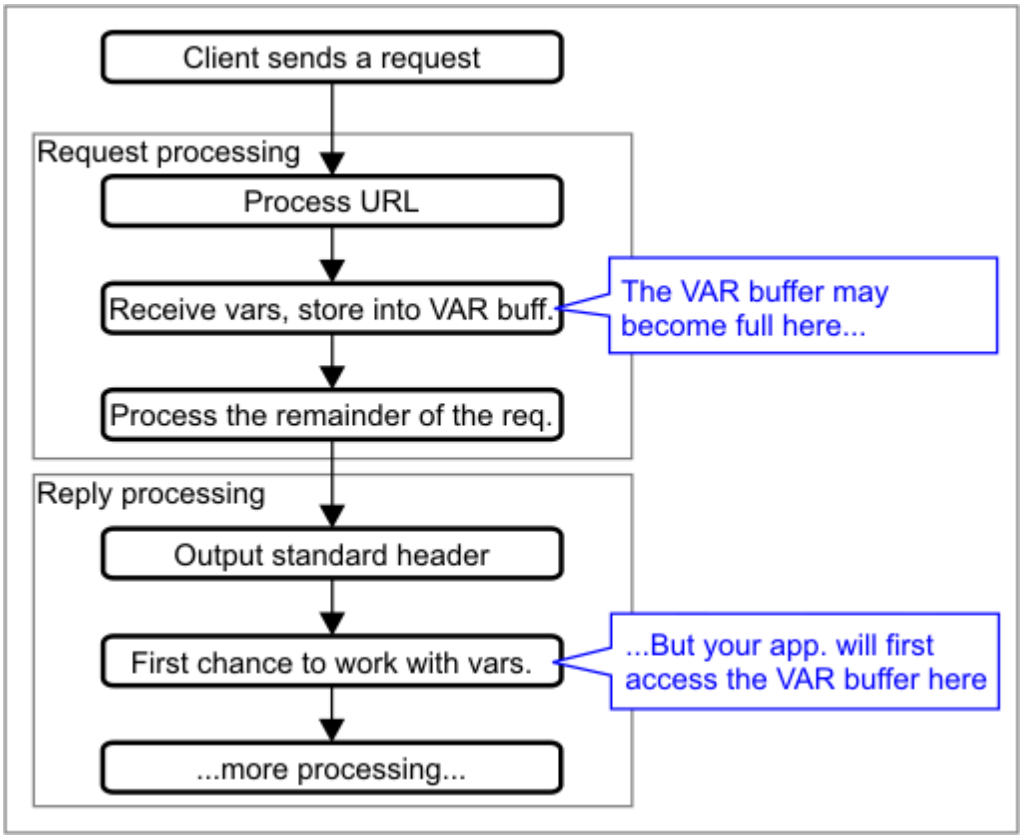
```
<html><body>
      <?
            dim s as string
            s=sock.httprqstring
            'actual processing here, s now contains the variable string
      ?>
</body></html>
```

Details on Variable Data 473 explains what exactly you get from the sock.httprqstring 483 property.

The advantage of using the sock.httprqstring is that the variable data is always there, no matter how many times you read it. Be forewarned, though...

- If you are using the sock.httprqstring, and if the client sends more data than can fit in the VAR buffer, the execution of the HTTP request processing will be stalled indefinitely. See Complex Case (Large Amount of Variable Data) 471 for a method of handling this correctly.

Let's talk a bit about why the socket may get stuck. The following diagram details the flow of HTTP request and response processing:

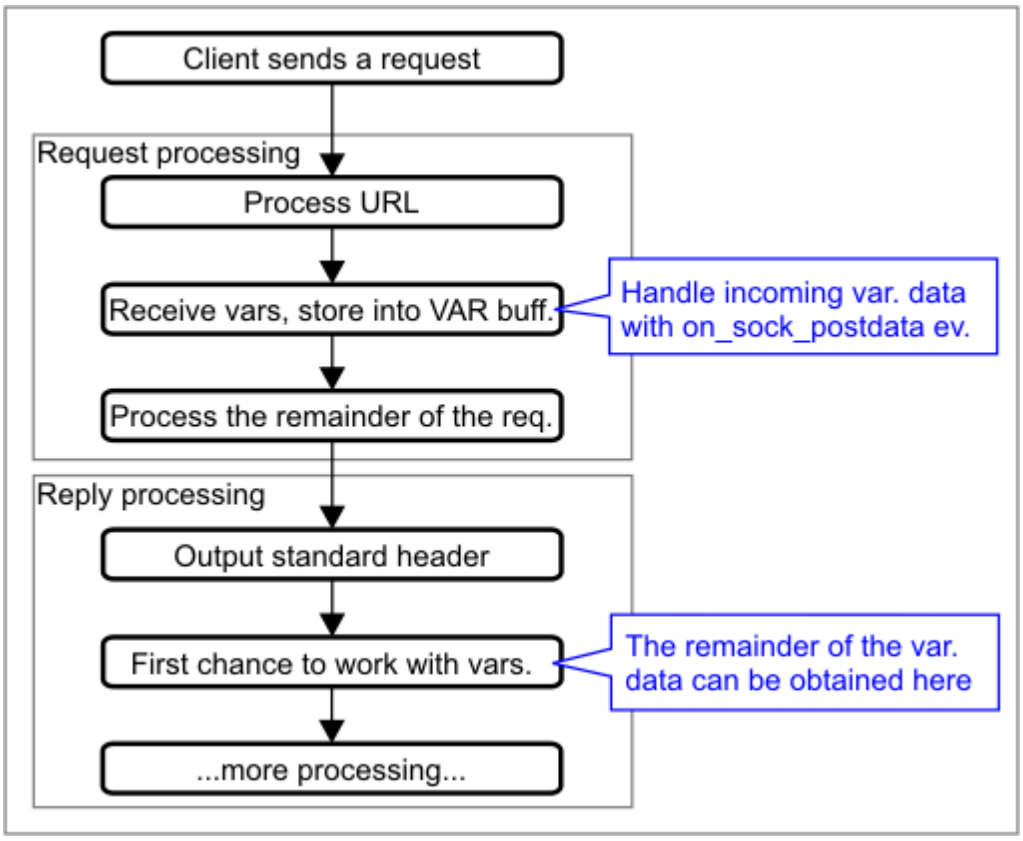


As you can see from the diagram, the socket will automatically extract the variable data and store it into the VAR buffer. The code example above will access the buffer during the reply phase -- from within the "login.html" page. Problem is, processing may never get that far. If the VAR buffer becomes full in the request processing phase, the socket will simply keep waiting, and the reply phase will never start! This is true no matter what HTTP method you use -- GET or POST. Read on 471 and we will tell you how to avoid this.

To avoid a problem situation described in the previous 470 topic use a more complex, but very reliable way of handling large HTTP variable data.

The on_sock_postdata 491 event is generated when there is data in the VAR buffer. In this regard, the event is similar in nature to the on_sock_data_arrival 489 event which is generated when the RX buffer of the socket has data. Unlike the on_sock_data_arrival event, the on_sock_postdata will first be generated only after the VAR buffer becomes full. That is, you won't be bothered by this event unless the HTTP request processing simply can't continue without it.

OK, so now you have a chance to access and process the HTTP variable data as it arrives and before the reply phase even starts. The sock.gethttprqstring 480 method, unlike the sock.httprqstring 483 R/O property, actually removes the data from the VAR buffer, thus freeing up the buffer space. Diagram below illustrates the process:



Here is a modified login example:

In the event handler for the on_sock_postdata event we extract available HTTP variable data and process it. The event will be called as many times as necessary. For example, we may save the data into a file:

```
Sub On_sock_postdata()
   fd.setdata(sock.gethttprqstring(255)) 'very simplified, but illustrates
the point
End Sub
```

Then, in the HTML file. It is necessary to remember that a portion of the HTTP variable data may still be unhandled by the time you get here:

```
<html><body>
   <?
      While sock.httprqstring<>"" 'extract the remaining part of the
variable data
         fd.setdata(sock.gethttprqstring(255))
      Wend
   ?>
</body></html>
```

It is unlikely that user login will require such careful handling. User name and

password will comfortably fit in 255 bytes, unless you users are paranoid and have humongous passwords. Still, there are many situation when you need to send large variable data. For example, HTTP POST methods are routinely used to upload files to the web server and your device will be able to handle this, too.

So far we haven't touched on the subject of actual data that you will read using sock.httprqstring 483 and sock.gethttprqstring 480. In other words, we haven't discussed the contents of the VAR buffer. It is time to clear this.

We have already explained that the above R/O property and method will return the same data, with the only difference that the sock.httprqstring will not actually remove the data from the VAR buffer and will not be able to access past the first 255 bytes of such data.

Data format in the VAR buffer depends on the method used.

VAR buffer contents -- HTTP GET method

Supposing the client has sent the following request:

```
GET /form_get.html?user=CHUCKY&pwd=THEEVILDOLL HTTP/1.1
Host: 127.0.0.1:1000
User-Agent: Mozilla/5.0 (and so on)
```

The VAR buffer will get everything past the URL and until the first CR/LF. The part is highlighted in **blue**. There are two variables -- user and pwd. They are equal to "CHUCKY" and "THEEVILDOLL". Your application can just ignore the "HTTP/1.1" part.

VAR buffer contents -- HTTP POST method

Here is a sample client request:

```
POST /form_get.html HTTP/1.1
Host: 127.0.0.1:1000
User-Agent: Mozilla/5.0
(lots of stuff skipped)
Content-Type: multipart/form-data; boundary=---------------------------
21724139663430
Content-Length: 257

-----------------------------21724139663430
Content-Disposition: form-data; name="user"

CHUCKY
-----------------------------21724139663430
Content-Disposition: form-data; name="pwd"

THEEVILDOLL
```

----------21724139663430--

You will get everything shown in **blue**.

Properties, Methods, and Events

This section provides an alphabetical list of all properties, methods, and events of the sock object.

.Acceptbcast Property

Function:	For the currently selected socket (selection is made through sock.num 488) specifies whether the socket will accept incoming broadcast UDP datagrams.
Type:	Enum (yes_no, byte)
Value Range:	0- NO **(default)**: will not accept broadcasts. 1- YES: will accept broadcasts.
See Also:	---

Details

This property is irrelevant for TCP communications (sock.protocol 492 =PL_SOCK_PROTOCOL_TCP).

.Allowedinterfaces Property

Function:	For the currently selected socket (selection is made through sock.num 488) defines the list of network interfaces on which this socket will accept incoming connections.
Type:	String
Value Range:	Platform-specific. Refer to your device's platform documentation (for example, EM1000's is here 143).
See Also:	Accepting Incoming Connections 426

Details

Interfaces that can be on the list are: "NET" (Ethernet), "WLN" (Wi-Fi), "PPP", and "PPPoE". The list of allowed interfaces is comma-delimited, i.e, "WLN,NET".

Note that reading back the value of this property will return the same list, but not necessarily in the same order. For example, the application may write "WLN,NET" into this property, yet read "NET,WLN" back. Unsupported interface names will be dropped from the list automatically.

The list of interfaces supported by your platform can be checked through sock.availableinterfaces 475. Only interfaces from this list can be specified as "allowed". Trying to allow an unsupported interface will not work.

The socket will not accept a connection on the interface which is not on the sock.allowedinterfaces list, even if all other connection parameters such as protocol, port, etc. are correct.

.Availableinterfaces R/O Property

Function:	Returns a comma-delimited list of network interfaces available on this platform.
Type:	String
Value Range:	Platform-specific. Refer to your device's platform documentation (for example, EM1000's is here 143).
See Also:	sock.targetinterface 506, sock.currentinterface 478, sock.allowedinterfaces 474

Details

.Bcast R/O Property

Function:	For the currently selected socket (selection is made through sock.num 488) reports whether the current or most recently received UDP datagram was a broadcast one.
Type:	Enum (no_yes, byte)
Value Range:	0- NO **(default)**: the UDP datagram is not a broadcast one. 1- YES: the UDP datagram is a broadcast one.
See Also:	sock.remotemac 495, sock.remoteip 494, sock.remoteport 495

Details

When the on_sock_data_arrival 489 event handler is entered, the sock.bcast will contain the broadcast status for the current datagram being processed. Outside of the on_sock_data_arrival event handler, the property will return the broadcast status of the most recent datagram received by the socket.

.Close Method

Function:	For the selected socket (selection is made through sock.num 488) causes the socket to close the connection with the other host.
Syntax:	**sock.close**
Returns:	---

See Also:

Details

For established TCP connections this will be a "graceful disconnect", if the TCP connection was in the "connection opening" or "connection closing" state this will be a reset (just like when the sock.reset 495 method is used). If connection was in the ARP phase or the transport protocol was UDP (sock.protocol 492 = 0- 0-PL_SOCK_PROTOCOL_UDP) the connection will be discarded (just like when the sock.discard 478 method is used). Method invocation will have NO effect if connection was closed at the time when the method was called (sock.state 502 in one of PL_SST_CLOSED states).

This method will be ignored when called from within an HTML page. HTML sockets are handled automatically and your application is not at freedom to close HTML sockets arbitrarily.

.Cmdbuffrq Method

Function:	For the selected socket (selection is made through sock.num 488) pre-requests "numpages" number of buffer pages (1 page= 256 bytes) for the CMD buffer of the socket.
Syntax:	**sock.cmdbuffrq(**numpages **as byte) as byte**
Returns:	Actual number of pages that can be allocated (byte)
See Also:	sock.rplbuffrq 496

Part	Description
numpages	Requested numbers of buffer pages to allocate.

Details

The CMD buffer is the buffer that accumulates incoming inband commands (messages). This method returns actual number of pages that can be allocated. Actual allocation happens when the sys.buffalloc method is used. The socket is unable to receive inband commands if its CMD buffer has 0 capacity. Unlike for TX or RX buffers there is no property to read out actual CMD buffer capacity in bytes. This capacity can be calculated as num_pages*256-16 (or =0 when num_pages=0), where "num_pages" is the number of buffer pages that was GRANTED through the sock.cmdbuffrq 476 method. "-16" is because 16 bytes are needed for internal buffer variables.

Buffer allocation will not work if the socket port to which this buffer belongs is not idle (sock.statesimple 505 is not at 0- PL_SSTS_CLOSED) at the time when sys.buffalloc executes. You can only change buffer sizes of sockets that are idle.

The CMD buffer is only required when inband commands are enabled (sock.inbandcommands 484 = 1-YES).

.Cmdlen R/O Property

Function:	For the selected socket (selection is made through sock.num 488) returns the length of data (in bytes) waiting to be processed in the CMD buffer.
Type:	Word
Value Range:	**Default**= 0 (0 bytes)
See Also:	sock.rpllen 497, sock.inbandcommands 484

Details

The CMD buffer accumulates incoming inband commands (messages) and may contain more than one such command. Use sock.getinband 481 method to extract the data from the CMD buffer.

.Connect Method

Function:	For the selected socket (selection is made through sock.num 488) causes the socket to attempt to connect to the target host.
Syntax:	**sock.connect**
Returns:	---
See Also:	

Details

The target is specified by the sock.targetport 507 and sock.targetip 506 (unless, for UDP, the socket is to broadcast the data- see the sock.targetbcast 505 property). Outgoing connection will be attempted through the network interface defined by the sock.targetinterface 506 property.

Method invocation will have effect only if connection was closed at the time when the method was called (sock.state 502 in one of PL_SST_CLOSED states).

.Connectiontout Property

Function:	For the currently selected socket (selection is made through sock.num 488) sets/returns connection timeout threshold for the socket in half-second increments.
Type:	Word
Value Range:	0-65535 (0-32767.5 seconds, 0 means "no timeout"), **default**= 0 (no timeout)
See Also:	Closing Connections 437

Details

When no data is exchanged across the connection for sock.connectiontout/2 number of seconds this connection is aborted (in the same way as if the sock.reset 495 method was used). Connection timeout of 0 means "no timeout".

Actual time elapsed since the last data exchange across the socket can be obtained through the sock.toutcounter 507 R/O property.

.Currentinterface R/O Property

Function:	For the currently selected socket (selection is made through sock.num 488) returns the network interface this socket is currently communicating through.
Type:	Enum (pl_sock_interfaces, byte)
Value Range:	Platform-specific. See the list of pl_sock_interfaces constants in the platform specifications (for example, EM1000's is here 143).
See Also:	Checking Connection Status 439

Details

The value of this property is only valid when the socket is not idle, i.e. sock.statesimple<> 0- PL_SSTS_CLOSED.

.Discard Method

Function:	For the selected socket (selection is made through sock.num 488) causes the socket to discard the connection with the other host.
Syntax:	**sock.discard**
Returns:	---
See Also:	sock.close 475, sock.reset 495

Details

Discarding the connection means simply forgetting about it without notifying the other side of the connection in any way.

This method will be ignored when called from within an HTML page. HTML sockets are handled automatically and your application is not at freedom to discard HTML sockets arbitrarily.

.Endchar Property

Function:	For the currently selected socket (selection is made through sock.num 488) specifies the ASCII code of the character that will end inband command (message).
Type:	Byte
Value Range:	0-255, **default**= 13 (CR)
See Also:	sock.esccchar 479

Details

Each inband message has to end with the end character, which will mark a return to the "regular" data stream of the TCP connection.

This property is irrelevant when inband commands are disabled (sock.inbandcommands 484 = 0- NO). The program won't be able to change the value of this property when the socket is not idle (sock.statesimple 505 <> 0- PL_SSTS_CLOSED).

.Escchar Property

Function:	For the currently selected socket (selection is made through sock.num 488) specifies the ASCII code of the character that will be used as an escape character for inband commands (messages).
Type:	Byte
Value Range:	0-255, **default**= 255
See Also:	sock.endchar 479

Details

Each inband message starts with "EC OC", where "EC" is the escape character defined by the sock.esccchar property and "OC" is any character other than "EC". With inband commands enabled, data characters with code matching that of the escape character is transmitted as "EC EC" (that is, two identical characters are needed to transmit a single data character with code matching that of escape character).

This property is irrelevant when inband commands are disabled (sock.inbandcommands 484 = 0- NO). The program won't be able to change the value of this property when the socket is not idle (sock.statesimple 505 <> 0- PL_SSTS_CLOSED).

.Event R/O Property (Obsolete)

This property is no longer available. Instead, the on_sock_event 490 property has a newstate argument that carries the state of the socket at the time of event generation.

.Eventsimple R/O Property (Obsolete)

This property is no longer available. Instead, the on_sock_event 490 property has a newstatesimple argument that carries the simplified state of the socket at the time of event generation.

.Getdata Method

Function:	For the selected socket (selection is made through sock.num 488) returns the string that contains the data extracted from the RX buffer.
Syntax:	**ser.getdata(**maxinplen **as word) as string**
Returns:	String containing data extracted from the RX buffer
See Also:	

Part	Description
maxinplen	Maximum amount of data to return (word).

Details

Extracted data is permanently deleted from the buffer. Length of extracted data is limited by one of the three factors (whichever is smaller): amount of data in the RX buffer itself, capacity of the "receiving" string variable, and the limit set by the maxinplen argument.

Additionally, if this socket uses UDP transport protocol (sock.protocol 492 = 1-PL_SOCK_PROTOCOL_TCP) the length of data that will be extracted is limited to the UDP datagram being processed. Additional conditions apply to UDP datagram processing; see on_sock_data_arrival 489 event and sock.nextpacket 487 method.

.Gethttprqstring Method

Function:	For the selected socket (selection is made through sock.num 488) extracts up to 255 bytes of the HTTP request string from theVAR buffer.
Syntax:	**ser.gethttprqstring(**maxinplen **as word) as string**
Returns:	String containing data extracted from the VAR buffer.
See Also:	Working with HTTP Variables 469, Sock.httprqstring 483

Part	Description
maxinplen	Maximum amount of data to return (word).

Details

Extracted data is permanently deleted from the VAR buffer. VAR buffer contents are explained in Details on Variable Data 473.

Length of extracted data is limited by one of the three factors (whichever is smaller): amount of data in the buffer itself, capacity of the "receiving" string variable, and the limit set by the maxinplen argument.

This method is only relevant when the socket is in the HTTP mode (sock.httpmode 481 = 1- YES). Use it from within an HTML page or on_sock_postdata 491 event handler.

.Getinband Method

Function: For the selected socket (selection is made through sock.num 488) returns the string that contains the data extracted from the CMD buffer.

Syntax: **sock.getinband as string**

Returns: String containing data from the CMD buffer

See Also:

Part	Description

Details

The CMD buffer is the buffer that accumulates inband commands. Extracted data is permanently deleted from the CMD buffer. Length of extracted data is limited by one of the two factors (whichever is smaller): amount of data in the CMD buffer itself, and the capacity of the "receiving" buffer variable. Several inband commands may be waiting in the CMD buffer. Each command will always be complete, i.e. there will be no situation when you will extract a portion of the command because the end of this command hasn't arrived yet. Inband commands stored in the CMD buffer will have escape character (see sock.escchar 479 property) and the character after the escape character already cut off, but the end character (see sock.endchar 479 property) will still be present. Therefore, your application can separate inband command from each other by finding the end characters.

.Httpmode Property

Function: For the currently selected socket (selection is made through sock.num 488) specifies whether this socket is in the HTTP mode.

Type:	Enum (no_yes, byte)
Value Range:	0- NO **(default)**: "regular" TCP connection. 1- YES: - HTTP connection.
See Also:	---

Details

This property is irrelavant when the sock.protocol[492] = PL_SOCK_PROTOCOL_UDP (UDP). If you do not set this property directly, it's value will be:

0- NO: for all outgoing connections (active opens) of the socket.

0- NO: for incoming connections received on one of the ports from the sock.localportlist[486] list.

1- YES: for incoming connections received on one of the ports from the sock.httpportlist[483] list.

You can manually switch any TCP connection at any time after it has been established from "regular" to HTTP by setting sock.httpmode= 1. However, this operation is "sticky"- once you have converted the TCP connection into the HTTP mode you cannot convert it back into the regular mode- trying to set sock.httpmode=0 won't have any effect- the TCP connection will remain in the HTTP mode until this connection is closed.

.Httpnoclose Property

Function:	For the currently selected socket (selection is made through sock.num[488]) sets/returns whether TCP HTTP connection will be kept opened after the HTTP request has been processed and the HTML page has been sent out
Type:	Enum (no_yes, byte)
Value Range:	0- NO **(default)**: connection will be closed (standard HTML server behavior) 1- YES: will be kept opened, special string will be used as a separator.
See Also:	---

Details

Normally, the end of HTML output is indicated by closing the TCP connection. When this property is set to 1- YES, connection is not closed at the end of HTML output. As a substitute, the end of HTML page output is marked by the following string: "<tibbo_linkserver_http_proxy_end_of_output>".

.Httpportlist Property

Function:	For the currently selected socket (selection is made through sock.num 488) sets/returns the list of listening ports on any of which this socket will accept an incoming HTTP connection.
Type:	String
Value Range:	0-32 characters, **default**= ""
See Also:	sock.localport 485, sock.httpmode 481

Details

This property is only relevant when incoming connections are allowed by the sock.inconmode 485 property and the the sock.protocol 492 = 1- PL_SOCK_PROTOCOL_TCP (HTML output cannot be done through the UDP).

This property is of string type and the list of ports is a comma-separated string, i.e. "80,81" (to accept HTTP connections on either port 80 or 81). Max string length for this property is 32 bytes.

Notice, that there is also the sock.localportlist 486 property that defines a list of listening ports for UDP and non-HTTP TCP connections. When a particular port is listed both under the sock.localportlist and the sock.httpportlist, and the protocol for this socket is TCP then sock.httpportlist has precedence (incoming TCP connection on the port in question will be interpreted as HTTP).

For example, if the sock.httpportlist= "80,81", the sock.localportlist="3000,80", the sock.protocol= 1- PL_SOCK_PROTOCOL_TCP, and there is an incoming TCP connection request on port 80 then this connection will be interpreted as HTTP one.

.Httprqstring R/O Property

Function:	For the selected socket (selection is made through sock.num 488) returns up to 255 bytes of the HTTP request string stored in the VAR buffer.
Type:	String
Value Range:	**Default**= ""
See Also:	Sock.varbuffrq 511

Details

The sock.httprqstring is a property; it can be invoked several times and will return the same data (when this property is used the data is not deleted from the VAR buffer). VAR buffer contents are explained in Details on Variable Data 473.

This property is only relevant when the socket is in the HTTP mode (sock.httpmode 481 = 1- YES). Use it from within an HTML page or on_sock_postdata 491 event handler. Maximum length of data that can be obtained through this property is 255

bytes, since this is the maximum possible capacity of a string variable that will accept the value of the sock.httprqstring.

HTTP requests can be much longer than 255 bytes and can even include entire files being uploaded from the client to your device. Rely on the on_sock_postdata event and the sock.gethttprqstring 480 method to handle large amounts of HTTP variable data correctly.

- If you are using the sock.httprqstring, and if the client sends more data than can fit in the VAR buffer, the execution of the HTTP request will be stalled indefinitely. To avoid this, reply on the on_sock_postdata event and the sock.gethttprqstring method, as explained in Complex Case (Large Amount of Variable Data) 471.

.Inbandcommands Property

Function:	For the currently selected socket (selection is made through sock.num 488) specifies whether inband command passing is allowed.
Type:	Enum (no_yes, byte)
Value Range:	0- NO **(default)**: inband commands are not allowed. 1- YES: inband commands are allowed.
See Also:	---

Details

Inband commands are messages passed within the TCP data stream. Each message has to be formatted in a specific way- see the sock.escchar 479 and sock.endchar 479 properties.

Inband commands are not possible for UDP communications so this setting is irrelevant when the sock.protocol 492 = 1- PL_SOCK_PROTOCOL_UDP. Inband messaging will work even when redirection (buffer shorting) is enabled for the socket (see the sock.redir 493 method). The program won't be able to change the value of this property when the socket is not idle (sock.statesimple 505 <> 0- PL_SSTS_CLOSED).

.Inconenabledmaster Property

Function:	A master switch that globally disables incoming connection acceptance on all sockets, irregardless of each socket's individual setup.
Type:	Enum (no_yes, byte)
Value Range:	0- NO: No socket will be allowed to accept an incoming connection. 1- YES **(default)**: Incoming connections are globally enabled. Individual socket's behavior and whether it will accept or reject a particular incoming connection depends

on the setup of this socket.

See Also: sock.inconmode 485, sock.localportlist 486, sock.httpportlist 483

Details

This property can be used to temporarily disable incoming connection acceptance on all sockets without changing individual setup of each socket.

.Inconmode Property

Function: For the currently selected socket (selection is made through sock.num 488) specifies whether incoming connections (passive opens) will be accepted and, if yes, from which sources.

Type: Enum (pl_sock_inconmode, byte)

Value Range: 0- PL_SOCK_INCONMODE_NONE **(default)**: The socket does not accept any incoming connections.

1- PL_SOCK_INCONMODE_SPECIFIC_IPPORT: The socket will only accept an incoming connection from specific IP (matching sock.targetip 506) and specific port (matching sock.targetport 507)

2- PL_SOCK_INCONMODE_SPECIFIC_IP_ANY_PORT: The socket will only accept an incoming connection from specific IP (matching sock.targetip), but any port.

3- PL_SOCK_INCONMODE_ANY_IP_ANY_PORT: The socket will accept an incoming connection from any IP and any port.

See Also: sock.reconmode 492, sock.localportlist 486, sock.httpportlist 483

Details

.Localport R/O Property

Function: For the currently selected socket (selection is made through sock.num 488) returns current local port of the socket.

Type: Word

Value Range: 0-65535, default= 0

See Also: ---

Details

Your application cannot set the local port directly. Instead, a list of ports on which the socket is allowed to accept an incoming connection (passive open) is supplied via the sock.localportlist 486 and sock.httpportlist 483 properties.

An incoming connection is accepted on any port from those two lists. The sock.localport property reflects current or the most recent local port on which connection was accepted.

.Localportlist Property

Function:	For the currently selected socket (selection is made through sock.num 488) sets/returns the list of listening ports on any of which this socket will accept an incoming UDP or TCP connection.
Type:	String
Value Range:	0-32 characters, **default**= ""
See Also:	sock.localport 485

Details

The socket will only accept an incoming connection when such connections are allowed by the sock.inconmode 485 property. Whether the socket will accept UDP or TCP connections is defined by the sock.protocol 492 property. Additionally, the sock.allowedinterfaces 474 property defines network interfaces on which the socket will accept an incoming connection.

This property is of string type and the list of ports is a comma-separated string, i.e. "1001,3000" (to accept connections on either port 1001 or 3000). Max string length for this property is 32 bytes. Notice, that there is also a sock.httpportlist 483 property that defines a list of listening ports for HTTP TCP connections.

.Newtxlen R/O Property

Function:	For the selected socket (selection is made through sock.num 488) returns the amount of uncommitted TX data in bytes.
Type:	Word
Value Range:	0-65535, **default**= 0 (0 bytes)
See Also:	---

Details

Uncommitted data is the one that was added to the TX buffer with the

sock.setdata 500 method but not yet committed using the sock.send 500 method.

.Nextpacket Method

Function:	For the selected socket (selection is made through sock.num 488) in the UDP mode (sock.protocol 492 = 0-PL_SOCK_PROTOCOL_UDP) closes processing of current UDP datagram and moves to the next datagram.
Syntax:	**sock.nextpacket**
Returns:	---
See Also:	---

Details

For UDP, the sock.getdata 480 method only extracts the data from a single UDP datagram even if several datagrams are stored in the RX buffer. When incoming UDP datagram processing is based on the on_sock_data_arrival 489 event the use of the sock.nextpacket method is not required since each invocation of the on_sock_data_arrival event handler "moves" processing to the next UDP datagram.

The method is useful when it is necessary to move to the next datagram without re-entering on_sock_data_arrival event handler. Therefore, sock.nextpacket is only necessary when the application needs to process several incoming UDP packets at once and within a single event handler.

.Notifysent Method

Function:	Using this method for the selected socket (selection is made through sock.num) will cause the on_sock_data_sent 489 event to be generated when the amount of committed data in the TX buffer is found to be below "threshold" number of bytes.
Syntax:	**notifysent(**threshold **as word)**
Returns:	---
See Also:	

Part	Description
threshold	Amount of bytes in the TX buffer below which the event it so be generated.

Details

Only one on_sock_data_sent event will be generated each time after the sock.notifysent is invoked. This method, together with the on_sock_data_sent event provides a way to handle data sending asynchronously.

Just like with sock.txfree 510, the trigger you set won't take into account any

uncommitted data in the TX buffer.

.Num Property

Function:	Sets/returns the number of currently selected socket.
Type:	Byte
Value Range:	The value of this property won't exceed sock.numofsock 488 -1 (even if you attempt to set higher value). **Default**= 0 (socket #0 selected).
See Also:	---

Details

Sockets are enumerated from 0. Most other properties and methods of this object relate to the socket selected through this property. Note that socket-related events such as on_sock_data_arrival 489 change currently selected socket!

.Numofsock R/O Property

Function:	Returns total number of sockets available on the current platform.
Type:	Byte
Value Range:	platform-dependent
See Also:	sock.num 488

Details

.Outport Property

Function:	For the currently selected socket (selection is made through sock.num 488) sets/returns the number of the port that will be used by the socket to establish outgoing connections.
Type:	Word
Value Range:	0-65535, **default**= 0
See Also:	---

Details

If this property is set to 0 then the socket will use "automatic" port numbers: for the first connection since the powerup the port number will be selected randomly, for all subsequent outgoing connections the port number will increase by one. Actual local port of a connection can be queried through the sock.localport 485 read-only property.

If this property is not at zero then the port it specifies will be used for all outgoing connections from this socket.

On_sock_data_arrival Event

Function: Generated when at least one data byte is present in the RX buffer of the socket (i.e. for this socket the sock.rxlen 499 >0).

Declaration: **on_sock_data_arrival**

See Also: sock.nextudpdatagram 487

Details

When the event handler for this event is entered the sock.num 488 property is automatically switched to the socket for which this event was generated. Another on_sock_data_arrival 489 event on a particular socket is never generated until the previous one is processed.

Use the sock.getdata 480 method to extract the data from the RX buffer.

For TCP protocol (sock.protocol 492 = 1- PL_SOCK_PROTOCOL_TCP), there is no separation into individual packets and you get all arriving data as a "stream". You don't have to process all data in the RX buffer at once. If you exit the on_sock_data_arrival event handler while there is still some unprocessed data in the RX buffer another on_sock_data_arrival event will be generated immediately.

For UDP protocol (sock.protocol= 0- PL_SOCK_PROTOCOL_UDP), the RX buffer preserves datagram boundaries. Each time you enter the on_sock_data_arrival event handler you get to process next UDP datagram. If you do not process entire datagram contents the unread portion of the datagram is discarded once you exit the event handler.

This event is not generated for a particular socket when buffer redirection is set for this socket through the sock.redir 493 method.

On_sock_data_sent Event

Function: Generated after the total amount of *committed* data in the TX buffer of the socket (sock.txlen) is found to be less than the threshold that was preset through the sock.notifysent 487 method.

Declaration: **on_sock_data_sent**

See Also: ---

Details

To cause generation of the on_sock_data_sent event, the application needs to use sock.notifysent each time. When the event handler is entered the sock.num 488 is automatically switched to the socket on which this event was generated.

Please, remember that uncommitted data in the TX buffer is not taken into account for on_sock_data_sent event generation.

On_sock_event Event

Function: Notifies your program that the socket state has changed.

Declaration: **on_sock_event(**newstate **as pl_sock_state,** newstatesimple **as pl_sock_state_simple)**

See Also: ---

Part	Description
newstate	"Detailed" state of the socket at the time of event generation.
newstatesimple	"Simplified" state of the socket at the time of event generation.

Details

The newstate and newstatesimple arguments carry the state as it was at the moment of event generation. This is different from sock.state 502 and sock.statesimple 505 R/O properties that return *current* socket state). The newstate argument uses the same set of constants as the sock.state. The newstatesimple argument uses the same set of constants as the sock.statesimple. See sock.state 502 and sock.statesimple 505 for detailed description of reported socket states.

Newstate and Newstatesimple arguments of the on_sock_event have been introduced in the Tibbo Basic release **2.0**. They replace sock.event 480 and sock.eventsimple 480 R/O properties which are no longer available.

On_sock_inband Event

Function: At least one data byte is present in the CMD buffer (sock.cmdlen 477>0).

Declaration: **on_sock_inband**

See Also: ---

Details

Use the sock.getinband 481 method to extract the data from the CMD buffer. Another on_inband_command event on a particular socket is never generated until

the previous one is processed. When the event handler is entered the sock.num is automatically switched to the socket on which this event was generated.

On_sock_overrun Event

Function: Data overrun has occurred in the RX buffer of the socket.

Declaration: **on_sock_overrun**

See Also: ---

Details

Normally, overruns can only happen for UDP communications as UDP has no "data flow control" and, hence, data overruns are normal. Another on_sock_overrun event on a particular socket is never generated until the previous one is processed. When the event handler for this event is entered the sock.num 488 is automatically switched to the socket on which this event was generated.

On_sock_postdata

Function: Generated when at least one data byte is present in the VAR buffer of the socket, but only after the VAR buffer has become full at least once in the cause of the current HTTP request processing.

Declaration: **on_sock_postdata**

See Also: Sock.varbuffrq 511

Details

HTTP requests can contain large amount of HTTP variable data, which is stored into the VAR buffer. The amount of such data can exceed the VAR buffer capacity. If this is not handled properly, the HTTP request execution may stall indefinitely -- see Working with HTTP Variables 469.

After the socket accepts an HTTP connection, this event is not generated (for this particular connection) until the VAR buffer becomes full. Once this happened, the event is generated even if there is a single byte waiting to be processed in the buffer. Two same-socket on_sock_postdata events never wait in the queue -- the next event can only be generated after the previous one is processed.

When the event handler for this event is entered the sock.num 488 property is automatically switched to the socket for which this event was generated.

Use the sock.gethttprqstring 480 method or sock.httprqstring 483 property to work with the VAR buffer's data.

On_sock_tcp_packet_arrival Event

Function: Notifies your program that the TCP packet of a certain size has arrived.

Syntax:	**on_sock_tcp_packet_arrival(**len **as word)**
See Also:	"Split Packet" Mode of TCP Data Processing 453

Part	Description
len	Length of the new data in the RXed TCP packet.

Details

This event is only generated when sock.splittcppackets 501 = 1- YES and sock.inbandcommands 484 = 0- DISABLED. Notice that only new data, never transmitted before, is counted. If the packet is a retransmission then this event won't be generated. Also, if some part of packet's data is a retransmission and some part is new then only the length of the new data will be reported. This way your program can maintain correct relationship between data lengths reported by this event and actual data in the RX buffer.

The on_sock_tcp_packet_arrival event is always generated before the on_sock_data_arrival 489 for any incoming TCP data (packet).

.Protocol Property

Function:	For the currently selected socket (selection is made through sock.num 488) selects the transport protocol.
Type:	Enum (pl_sock_protocol, byte)
Value Range:	0- PL_SOCK_PROTOCOL_UDP **(default)**: UDP transport protocol. 1- PL_SOCK_PROTOCOL_TCP: TCP transport protocol.
See Also:	---

Details

Notice, that there is no "HTTP" selection, as HTTP is not a transport protocol (TCP is the transport protocol required by the HTTP). You make the socket accept HTTP connections by specifying the list of HTTP listening ports using the sock.httpportlist 483 property or using the sock.httpmode 481 property.

The program won't be able to change the value of this property when the socket is not idle (sock.statesimple 505 <> 0- PL_SSTS_CLOSED).

.Reconmode Property

Function:	For the currently selected socket (selection is made through sock.num 488) whether the socket accepts reconnects, and, if yes, from which sources.
Type:	Enum (pl_sock_reconmode, byte)
Value Range:	0- PL_SOCK_RECONMODE_0 (default): For UDP: Reconnects accepted only from the same IP as the one

already engaged in the current connection with this socket, but any port; port switchover will not happen. TCP: reconnects are not accepted at all.

1- PL_SOCK_RECONMODE_1: For UDP: Reconnects accepted from any IP, any port; port switchover will not happen. TCP: reconnects accepted only from the same IP and port as the ones already engaged in the current connection with this socket.

2- PL_SOCK_RECONMODE_2: For UDP: Reconnects accepted only from the same IP as the one already engaged in the current connection with this socket, but any port; port switchover will happen. TCP: reconnects accepted only from the same IP as the one already engaged in the current connection with this socket, but any port.

3- PL_SOCK_RECONMODE_3: For UDP: Reconnects accepted from any IP, any port; port switchover will happen. TCP: reconnects accepted from any IP, any port.

See Also: ---

Details

Reconnect situation is when a passive open and resulting connection replace, for the same socket, the connection that was already in progress. For UDP, this property additionally defines whether a "port switchover" will occur as a result of an incoming connection (passive open) or a reconnect. Port switchover is when the socket starts sending its outgoing UDP datagrams to the port from which the most recent UDP datagram was received, rather than the port specified by the sock.targetport 507 property.

DO NOT enable reconnects on sockets that are used to handle HTML requests. This will interfere with HTML operation, as explained in the Understanding TCP Reconnects 428 topic.

.Redir Method

Function: For the selected socket (selection is made through sock.num 488) redirects the data being RXed to the TX buffer of the same socket, different socket, or another object that supports compatible buffers.

Syntax: **sock.redir(**redir **as pl_redir) as pl_redir**

Returns: Returns 0- PL_REDIR_NONE if redirection failed or the same value as was passed in the redir argument.

See Also:

Part	Description

redir	Platform-specific. See the list of pl_redir constants in the platform specifications.

Details

Data redirection (sometimes referred to as "buffer shorting") allows to arrange efficient data exchange between ports, sockets, etc. in cases where no data alteration or parsing is necessary, while achieving maximum possible throughput is important.

The redir argument, as well as the value returned by this method are of "enum pl_redir" type. The pl_redir defines a set of platform inter-object constants that include all possible redirections for this platform. Specifying redir value of 0-PL_REDIR_NONE cancels redirection. When the redirection is enabled for a particular socket, the on_sock_data_arrival 489 event is not generated for this port.

Once the RX buffer is redirected certain properties and methods related to the RX buffer will actually return the data for the TX buffer to which this RX buffer was redirected:

- sock.rxbuffsize 498 will actually be returning the size of the TX buffer.
- sock.rxclear 498 method will actually be clearing the TX buffer.
- sock.rxlen 499 method will be showing the amount of data in the TX buffer.

.Remoteip R/O Property

Function:	For the currently selected socket (selection is made through sock.num 488) returns the IP address of the host with which this socket had the most recent or currently has a connection.
Type:	String
Value Range:	**Default**= "0.0.0.0"
See Also:	sock.remotemac 495, sock.remoteport 495, sock.bcast 475

Details

The application cannot directly change this property, it can only specify the target IP address for active opens through the sock.targetip 506 property.

For UDP connections, when the on_sock_data_arrival 489 event handler is entered, the sock.remoteip will contain the IP address of the sender of the current datagram being processed. Outside of the on_sock_data_arrival event handler, the property will return the source IP address of the most recent datagram received by the socket.

.Remotemac R/O Property

Function:	For the currently selected socket (selection is made through sock.num 488) returns the MAC address of the host with which this socket had the most recent or currently has a connection.
Type:	String
Value Range:	**Default**= "0.0.0.0.0.0"
See Also:	sock.remoteip 494, sock.remoteport 495, sock.bcast 475

Details

For UDP connections, when the on_sock_data_arrival 489 event handler is entered, the sock.remotemac will contain the MAC address of the sender of the current UDP datagram being processed. Outside of the on_sock_data_arrival event handler, the property will return the source MAC address of the most recent datagram received by the socket.

.Remoteport R/O Property

Function:	For the currently selected socket (selection is made through sock.num 488) returns the port number of the host with which this socket had the most recent or currently has a connection.
Type:	Word
Value Range:	0-65535, **default**= 0
See Also:	sock.remotemac 495, sock.remoteip 494, sock.bcast 475

Details

The application cannot directly change this property, it can only specify the target port for active opens through the sock.targetport 507 property.

For UDP connections, when the on_sock_data_arrival 489 event handler is entered, the sock.remoteport will contain the port number of the sender of the current datagram being processed. Outside of the on_sock_data_arrival event handler, the property will return the source port of the most recent datagram received by the socket.

.Reset Method

Function:	For the selected socket (selection is made through sock.num 488) causes the socket to abort the connection with the other host.

Syntax:	**sock.reset**
Returns:	---
See Also:	sock.close 475

Details

For TCP connections that were established, being opened, or being closed this will be a reset (RST will be sent to the other end of the connection). If connection was in the ARP phase or the transport protocol was UDP (sock.protocol 492 = 0- PL_SOCK_PROTOCOL_UDP) the connection will be discarded (just like when the sock.discard 478 method is used).

Method invocation will have NO effect if connection was closed at the time when the method was called (sock.state 502 in one of PL_SST_CLOSED states).

This method will be ignored when called from within an HTML page. HTML sockets are handled automatically and your application is not at freedom to reset HTML sockets arbitrarily.

.Rplbuffrq Method

Function:	For the selected socket (selection is made through sock.num 488) pre-requests "numpages" number of buffer pages (1 page= 256 bytes) for the RPL buffer of the socket.
Syntax:	**sock.cmdbuffrq(**numpages **as byte) as byte**
Returns:	Actual number of pages that can be allocated (Byte).
See Also:	sock.cmdbuffrq 476

Part	Description
numpages	Requested numbers of buffer pages to allocate.

Details

The RPL buffer is the the buffer that stores outgoing inband replies (messages). Actual allocation happens when the sys.buffalloc 530 method is used. The socket is unable to send inband replies if its RPL buffer has 0 capacity.

Unlike for TX or RX buffers there is no property to read out actual RPL buffer capacity in bytes. This capacity can be calculated as num_pages*256-16 (or =0 when num_pages=0), where "num_pages" is the number of buffer pages that was GRANTED through the sock.rplbuffrq. "-16" is because 16 bytes are needed for internal buffer variables.

Buffer allocation will not work if the socket port to which this buffer belongs is not idle (sock.statesimple 505 is not at 0- PL_SSTS_CLOSED) at the time when sys.buffalloc executes. You can only change buffer sizes of sockets that are idle.

The RPL buffer is only required when inband commands are enabled (sock.inbandcommands 484 = 1- YES).

.Rplfree R/O Property

Function:	For the selected socket (selection is made through sock.num 488) returns the free space (in bytes) available in the RPL buffer
Type:	Word
Value Range:	0-65535, **default**= 0 (0 bytes)
See Also:	sock.cmdlen 477, sock.rpllen 497, sock.inbandcommands 484

Details

The RPL buffer is the buffer that keeps outgoing inband replies (messages). Your application adds inband replies to the RPL buffer with the sock.setsendinband 500 method. Several inband replies may be waiting in the RPL buffer.

.Rpllen R/O Property

Function:	For the selected socket (selection is made through sock.num 488) returns the length of data (in bytes) waiting to be send out from the RPL buffer.
Type:	Word
Value Range:	
See Also:	sock.cmdlen 477, sock.rplfree 497, sock.inbandcommands 484

Details

Your application adds inband replies to the RPL buffer with the sock.setsendinband 500 method. Several inband replies may be waiting in the RPL buffer.

.Rxbuffrq Method

Function:	For the selected socket (selection is made through sock.num) pre-requests "numpages" number of buffer pages (1 page= 256 bytes) for the RX buffer of the socket.
Syntax:	**sock.rxbuffrq(**numpages **as byte) as byte**
Returns:	Actual number of pages that can be allocated (Byte).
See Also:	sock.txbuffrq 509

Part	Description
numpages	Requested numbers of buffer pages to allocate.

Details

Returns actual number of pages that can be allocated. Actual allocation happens when the sys.buffalloc 530 method is used. The socket is unable to RX data if its RX buffer has 0 capacity. Actual current buffer capacity can be checked through the sock.rxbuffsize 498 which returns buffer capacity in bytes.

Relationship between the two is as follows: sock.rxbuffsize=num_pages*256-16 (or =0 when num_pages=0), where "num_pages" is the number of buffer pages that was GRANTED through the sock.rxbuffrq. "-16" is because 16 bytes are needed for internal buffer variables.

Buffer allocation will not work if the socket port to which this buffer belongs is not idle (sock.statesimple 505 is not at 0- PL_SSTS_CLOSED) at the time when sys.buffalloc executes. You can only change buffer sizes of sockets that are idle.

.Rxbuffsize R/O Property

Function:	For the currently selected socket (selection is made through sock.num 488) returns current RX buffer capacity in bytes.
Type:	Word
Value Range:	0-65535
See Also:	---

Details

Buffer capacity can be changed through the sock.rxbuffrq 497. The sock.rxbuffrq requests buffer size in 256-byte pages whereas this property returns buffer size in bytes.

Relationship between the two is as follows: sock.rxbuffsize=num_pages*256-16 (or =0 when num_pages=0), where "num_pages" is the number of buffer pages that was GRANTED through the sock.rxbuffrq. "-16" is because 16 bytes are needed for internal buffer variables. The socket cannot RX data when the RX buffer has zero capacity.

.Rxclear Method

Function:	For the selected socket (selection is made through sock.num 488) clears (deletes all data from) the RX buffer.
Syntax:	**sock.rxclear**
Returns:	---

See Also: ---

Details

Invoking this method will have no effect when the socket is in the HTTP mode (sock.httpmode 481 = 1- YES).

.Rxpacketlen R/O Property

Function: For the selected socket (selection is made through sock.num 488) returns the length (in bytes) of the UDP datagram being extracted from the RX buffer.

Type: Word

Value Range: 0-1536, **default**= 0

See Also: sock.rxlen 499

Details

This property is only relevant when the sock.protocol 492 = 1- PL_SOCK_PROTOCOL_TCP. Correct way of using this property is within the on_sock_data_arrival 489 event or in conjunction with the sock.nextpacket 487 method.

.Rxlen R/O Property

Function: For the selected socket (selection is made through sock.num 488) returns total number of bytes currently waiting in the RX buffer to be extracted and processed by your application.

Type: Word

Value Range: 0-65535, **default**= 0

See Also: ---

Details

The on_sock_data_arrival 489 event is generated once the RX buffer is not empty, i.e. there is data to process. There may be only one on_ser_data_arrival event for each socket waiting to be processed in the event queue. Another on_sock_data_arrival event for the same socket may be generated only after the previous one is handled.

If, during the on_sock_data_arrival event handler execution, not all data is extracted from the RX buffer, another on_sock_data_arrival event is generated immediately after the on_sock_data_arrival event handler is exited.

.Send Method

Function:	For the selected socket (selection is made through sock.num 488) commits (allows sending) the data that was previously saved into the TX buffer using the sock.setdata 500 method.
Syntax:	**sock.rxbuffrq**
Returns:	---
See Also:	---

Details

You can monitor the sending progress by checking the sock.txlen 510 property or using the sock.notifysent 487 method and the on_sock_data_sent 489 event.

.Setdata Method

Function:	For the selected socket (selection is made through sock.num 488) adds the data passed in the txdata argument to the contents of the TX buffer.
Syntax:	**ser.setdata(byref** txdata **as string)**
Returns:	---
See Also:	---

Part	Description
txdata	The data to send; this data will be added to the contents of the TX buffer.

Details

If the buffer doesn't have enough space to accommodate the data being added then this data will be truncated. Newly saved data is not sent out immediately. This only happens after the sock.send 500 method is used to commit the data. This allows your application to prepare large amounts of data before sending it out.

Total amount of newly added (uncommitted) data in the buffer can be checked through the sock.newtxlen 486 setting.

.Setsendinband Method

Function:	For the selected socket (selection is made through sock.num 488) puts the data into the RPL buffer. This method also commits the data.
Syntax:	**sock.setsendinband(byref** data **as string)**

Returns: ---

See Also: ---

Part	Description
data	The data to send (String).

Details

This method is different from the TX buffer for which two separate methods- sock.setdata 500 and sock.send 500- are used to store and commit the data. For the RPL buffer you store and commit the data with a single sock.setsendinband method.

It is the responsibility of your application to properly encapsulate outgoing messages with escape sequence ("EC OC", see the sock.escchar 479 property) and the end character (see the sock.endchar 479 property). When adding the data to the RPL buffer make sure you are adding entire inband message at once- you are not allowed to do this "in portions"!

Sinkdata Property

Function: For the currently selected socket (selection is made through sock.num 488) specifies whether the incoming data should be discarded.

Type: Enum (yes_no, byte)

Value Range: 0- NO **(default)**: normal data processing.

1- YES: discard incoming data.

See Also: Sinking Data 456

Details

Setting this property to 1- YES causes the socket to automatically discard all incoming data without passing it to your application. The on_sock_data_arrival 489 event will not be generated, reading sock.rxlen 499 will always return zero, and so on. No data will be reaching its destination even in case of buffer redirection 454. Inband commands 456, however, will still be extracted from the incoming data stream and processed. Sock.connectiontout 477 and sock.toutcounter 507 will work correctly as well.

.Splittcppackets Property

Function: For the currently selected socket (selection is made through sock.num 488) specifies whether the program will have additional control over the size of TCP packets being received and transmitted.

Type: Enum (no_yes, byte)

Value Range:

0- NO **(default)**: regular processing of TCP data in which your program does not care about the size of individual TCP data packets being transmitted and received.

1- YES: Additional features are enabled that allow your program to know the size of each incoming TCP packet and also control the size of each outgoing TCP packet.

See Also:

"Split Packet" Mode of TCP Data Processing 453

Details

When this property is set to 1- YES your program gets an additional degree of control over TCP. For incoming TCP data, the program can know the size of individual incoming packets (this will be reported by the on_sock_tcp_packet_arrival 491 event).

For outgoing TCP data, no packet will be sent out at all unless entire contents of the TX buffer can be sent. Therefore, by executing sock.send 500 and waiting for sock.txlen 510=0 your program can make sure that the packet sent will have exactly the size you needed.

The property is only relevant when the sock.inbandcommands 484= 0- NO. With inband commands enabled, the socket will always behave as if the sock.splittcppackets= 0- NO. The program won't be able to change the value of this property when the socket is not idle (sock.statesimple 505<> 0- PL_SSTS_CLOSED).

Notice, that sending out TCP data and waiting for the sock.txlen 510=0 significantly diminishes your TX data throughput. This is because each send will be waiting for the other end to confirm the reception of data.

Also, with sock.splittcppackets= 1= YES make sure that you are not sending more data than the size of the RX buffer on the other end. If this happens, no data will ever get through because your side will be waiting for the chance to send out all TX data at once, and the other end won't be able to receive this much data in one piece.

Also, attempting to send the packet with size exceeding the "maximum segment size" (MSS) as specified by the other end will lead to data fragmentation! The socket will never send any TCP packet with the amount of data exceeding MSS.

.State R/O Property

Function:

For the currently selected socket (selection is made through sock.num 488) returns "detailed" current socket state.

Type:

Enum (pl_sock_state, byte)

Value Range:

0- PL_SST_CLOSED **(default)**: Connection is closed (and haven't been opened yet, it is a post-powerup state). Applied both to UDP and TCP.

1- PL_SST_CL_PCLOSED: Connection is closed (it was a passive close). Applies only to TCP.

2- PL_SST_CL_ACLOSED: Connection is closed (it was an active close by the application). Applies only to TCP.

3- PL_SST_CL_PRESET_POPENING: Connection is closed (it was a passive reset during a passive open). Applies only to TCP.

4- PL_SST_CL_PRESET_AOPENING: Connection is closed (it was a passive reset during an active open). Applies only to TCP.

5- PL_SST_CL_PRESET_EST: Connection is closed (it was a passive reset while in "connection established" state). Applies only to TCP.

6- PL_SST_CL_PRESET_PCLOSING: Connection is closed (it was a passive reset while performing a passive close). Applies only to TCP.

7- PL_SST_CL_PRESET_ACLOSING: Connection is closed (it was a passive reset while performing an active close). Applies only to TCP.

8- PL_SST_CL_PRESET_STRANGE: Connection is closed (it was a passive reset, no further details available). Applies only to TCP.

9- PL_SST_CL_ARESET_CMD: Connection is closed (it was an active reset issued by the application). Applies only to TCP.

10- PL_SST_CL_ARESET_RE_PO: Connection is closed (it was an active reset issued because of excessive retransmission attempts during a passive open). Applies only to TCP.

11- PL_SST_CL_ARESET_RE_AO: Connection is closed (it was an active reset issued because of excessive retransmission attempts during an active open). Applies only to TCP.

12- PL_SST_CL_ARESET_RE_EST: Connection is closed (it was an active reset issued because of excessive retransmission attempts while in "connection established" state). Applies only to TCP.

13- PL_SST_CL_ARESET_RE_PCL: Connection is closed (it was an active reset issued because of excessive retransmission attempts during a passive close). Applies only to TCP.

14- PL_SST_CL_ARESET_RE_AC: Connection is closed (it was an active reset issued because of excessive retransmission attempts during a passive open). Applies only to TCP.

15- PL_SST_CL_ARESET_TOUT: Connection is closed (it was an active reset caused by connection timeout, i.e. no data was exchanged for sock.connectiontout number of seconds). Applies only to TCP.

16- PL_SST_CL_ARESET_DERR: Connection is closed (it was an active reset caused by a data exchange error). Applies only to TCP.

17- PL_SST_CL_DISCARDED_CMD: Connection is closed (it was discarded by the application). Applies both to UDP and TCP.

18- PL_SST_CL_DISCARDED_PO_WCS: Connection is closed (it was discarded because an error in connection sequence was detected during a passive open). Applies only to TCP.

19- PL_SST_CL_DISCARDED_AO_WCS: Connection is closed (it was discarded because an error in connection sequence was detected during an active open). Applies only to TCP.

20- PL_SST_CL_DISCARDED_ARPFL: Connection is closed (it was discarded because the device has failed to resolve the IP address of the destination during an active open, i.e. there was no reply to ARP requests). Applies both to UDP and TCP.

21- PL_SST_CL_DISCARDED_TOUT: Connection is closed (it was discarded because connection has timed out, i.e. no data was exchanged for sock.connectiontout number of seconds). Applies only to UDP.

32- PL_SST_ARP: ARP resolution is an progress (it is an active open). Applied both to UDP and TCP.

64- PL_SST_PO: Connection is being established (it is a passive open). Applies only to TCP.

96- PL_SST_AO: Connection is being established (it is an active open). Applies only to TCP.

128- PL_SST_EST: Connection is established (generic, includes both passive and active open). Applies both to UDP and TCP.

128- PL_SST_EST_POPENED: Connection is established (it was a passive open). Applies both to UDP and TCP.

129- PL_SST_EST_AOPENED: Connection is established (it was an active open). Applies both to UDP and TCP.

160- PL_SST_PC: Connection is being closed (it is a passive close). Applies only to TCP.

192- PL_SST_AC: Connection is being closed (it is an active close). Applies only to TCP.

See Also: ---

Details

This property tells the "detailed" current state of the socket, as opposed to the sock.event 480 property that returns the "detailed" state at the moment of the on_sock_event 490 event generation.

Another set of read-only properties- sock.eventsimple 480 and sock.statesimple 505- return "simplified" socket states.

.Statesimple R/O Property

Function: For the currently selected socket (selection is made through sock.num 488) returns "simplified" current socket state.

Type: Enum (pl_sock_state_simple, byte)

Value Range:

0- PL_SSTS_CLOSED **(default)**: Connection is closed. Applies both to UDP and TCP.

1- PL_SSTS_ARP: ARP resolution is an progress (it is an active open). Applies both to UDP and TCP.

2- PL_SSTS_PO: Connection is being established (it is a passive open). Applies only to TCP.

3- PL_SSTS_AO: Connection is being established (it is an active open). Applies only to TCP.

4- PL_SSTS_EST: Connection is established. Applies both to UDP and TCP.

5- PL_SSTS_PC: Connection is being closed (it is a passive close). Applies only to TCP.

6- PL_SSTS_AC: Connection is being closed (it is an active close). Applies only to TCP.

See Also: ---

Details

This property tells the current "simplified" state of the socket, as opposed to the sock.eventsimple 480 property that returns the "simplified" state at the moment of the on_sock_event 490 event generation.

Another set of read-only properties- sock.event 480 and sock.state 502- return "detailed" socket states.

.Targetbcast Property

Function: For the currently selected socket (selection is made through sock.num 488) specifies whether this port will be sending its outgoing UDP datagrams as link-level broadcasts.

Type: Enum (no_yes, byte)

Value Range:

0- NO: UDP datagrams will be sent as "normal" packets

1- YES: UDP datagrams will be sent out as link-level broadcast packets

See Also: ---

Details

This property is only relevant for UDP communications (sock.protocol 492= PL_SOCK_PROTOCOL_UDP). When this property is set to 1- YES the socket will be

sending out all UDP datagrams as broadcasts and incoming datagrams won't cause port switchover, even if the latter is enabled through the sock.reconmode 492 property.

.Targetinterface Property

Function:	For the currently selected socket (selection is made through sock.num 488) selects the network interface through which an outgoing network connection will be established.
Type:	Enum (pl_sock_interfaces, byte)
Value Range:	Platform-specific. See the list of pl_sock_interfaces constants in the platform specifications (for example, EM1000's is here 143).
See Also:	Establishing Outgoing Connections 434

Details

--

.Targetip Property

Function:	For active opens on the currently selected socket (selection is made through sock.num 488) specifies the target IP to which the socket will attempt to connect to. For passive opens specifies, in certain cases, the only IP address from which an incoming connection will be accepted.
Type:	String.
Value Range:	Any valid IP address, i.e. "192.168.100.40". Default= "0.0.0.0"
See Also:	sock.targetport 507, sock.remoteport 495

Details

For active opens, this property is only relevant at the moment of connection establishment.

For incoming connections, whether this property will matter or not is defined by the sock.inconmode 485 property. When the sock.inconmode= 1-PL_SOCK_INCONMODE_SPECIFIC_IPPORT or 2-PL_SOCK_INCONMODE_SPECIFIC_IP_ANY_PORT only the host with IP matching the one set in the sock.targetip property will be able to connect to the socket.

Current IP on the "other side" of the connection can always be checked through the sock.remoteip 494 read-only property.

.Targetport Property

Function:	For active opens on the currently selected socket (selection is made through sock.num 488) specifies the target port to which the socket will attempt to connect to. For passive opens specifies, in certain cases, the only port from which an incoming connection will be accepted.
Type:	Word
Value Range:	0-65535, **default**= 0
See Also:	sock.targetip 506, sock.remoteip 494

Details

For active opens, this property is only relevant at the moment of connection establishment.

For incoming connections, whether this property will matter or not is defined by the sock.inconmode 485 property. When the sock.inconmode= 1-PL_SOCK_INCONMODE_SPECIFIC_IPPORT an incoming connection will only be accepted from the port matching the one set in the sock.targetport property.

Current port on the "other side" of the connection can always be checked through the sock.remoteport 495 read-only property.

.Toutcounter R/O property

Function:	For the currently selected socket (selection is made through sock.num 488) returns the time, in 0.5 second intervals, elapsed since the data was last send or received on this socket.
Type:	Word
Value Range:	0-65535, **default**= 0
See Also:	Closing Connections 437

Details

This property is reset to 0 each time there is some data exchanged across the socket connection. The property increments at 0.5 second intervals while no data is moving through this socket.

If the sock.connectiontout 477 is not at 0, this property increments until it reaches the value of the sock.connectiontout and the connection is terminated. The sock.toutcounter then stays at the value of sock.connectiontout.

If the sock.connectiontout is at 0, the maximum value that the sock.toutcounter can reach is 1. That is, the sock.toutcounter will be at 0 after the data exchange, and at 1 if at least 0.5 seconds have passed since the last data exchange.

.Tx2buffrq Method

Function: For the selected socket (selection is made through sock.num 488) pre-requests "numpages" number of buffer pages (1 page= 256 bytes) for the TX2 buffer of the socket.

Syntax: **sock.tx2buffrq(**numpages **as byte) as byte**

Returns: Actual number of pages that can be allocated (Byte).

See Also: sock.txbuffrq 509

Part	Description
numpages	Requested numbers of buffer pages to allocate.

Details

The TX2 buffer is required when inband commands are enabled (sock.inbandcommands 484 = 1- YES), without it the socket won't be able to TX data. Returns actual number of pages that can be allocated. Actual allocation happens when the sys.buffalloc 530 method is used. Unlike for TX or RX buffers there is no property to read out actual TX2 buffer capacity in bytes. This capacity can be calculated as num_pages*256-16 (or =0 when num_pages=0), where "num_pages" is the number of buffer pages that was GRANTED through the sock.tx2buffrq. "-16" is because 16 bytes are needed for internal buffer variables.

Buffer allocation will not work if the socket port to which this buffer belongs is not idle (sock.statesimple 505 is not at 0- PL_SSTS_CLOSED) at the time when sys.buffalloc executes. You can only change buffer sizes of sockets that are idle.

.Tx2len R/O Property

Function: For the selected socket (selection is made through sock.num 488) returns the amount of data waiting to be sent out in the TX2 buffer.

Type: Word

Value Range: 0-65535, **default**= 0 (0 bytes)

See Also: sock.txlen 510

Details

The TX2 buffer is needed to transmit outgoing TCP data when inband commands (messages) are enabled (sock.inbandcommands 484 = 1- YES). If your application needs to make sure that all data is actually sent out from the socket then it must verify that both TX and TX2 buffers are empty.

.Txbuffrq Method

Function:	For the selected socket (selection is made through sock.num 488) pre-requests "numpages" number of buffer pages (1 page= 256 bytes) for the TX buffer of the socket.
Syntax:	**sock.txbuffrq(**numpages **as byte) as byte**
Returns:	Actual number of pages that can be allocated (Byte).
See Also:	sock.tx2buffrq 508

Part	Description
numpages	Requested numbers of buffer pages to allocate.

Details

Actual allocation happens when the sys.buffalloc 530 method is used. The socket is unable to TX data if its TX buffer has 0 capacity. Actual current buffer capacity can be checked through the sock.txbuffsize 509 which returns buffer capacity in bytes. Relationship between the two is as follows: sock.txbuffsize=num_pages*256-16 (or =0 when num_pages=0), where "num_pages" is the number of buffer pages that was GRANTED through the sock.txbuffrq. "-16" is because 16 bytes are needed for internal buffer variables.

Buffer allocation will not work if the socket port to which this buffer belongs is not idle (sock.statesimple 505 is not at 0- PL_SSTS_CLOSED) at the time when sys.buffalloc executes. You can only change buffer sizes of sockets that are idle.

.Txbuffsize R/O Property

Function:	For the currently selected socket (selection is made through sock.num 488) returns current TX buffer capacity in bytes.
Type:	Word
Value Range:	0-65535, **default**= 0 (0 bytes).
See Also:	---

Details

Buffer capacity can be changed through the sock.txbuffrq 509 method followed by the sys.buffalloc 530 method.

The sock.txbuffrq requests buffer size in 256-byte pages whereas this property returns buffer size in bytes. Relationship between the two is as follows: sock.txbuffsize=num_pages*256-16 (or =0 when num_pages=0), where "num_pages" is the number of buffer pages that was GRANTED through the sock.txbuffrq. "-16" is because 16 bytes are needed for internal buffer variables.

The socket cannot TX data when the TX buffer has zero capacity.

.Txclear Method

Function:	For the selected socket (selection is made through sock.num 488) clears (deletes all data from) the TX buffer.
Syntax:	**sock.rxclear**
Returns:	---
See Also:	---

Details

Invoking this method will have no effect when the socket is not closed (sock.statesimple 505 <> 0- PL_SSTS_CLOSED).

.Txfree R/O Property

Function:	For the selected socket (selection is made through sock.num 488) returns the amount of free space in the TX buffer in bytes.
Type:	Word
Value Range:	0-65535, **default**= 0 (0 bytes).
See Also:	---

Details

Notice, that the amount of free space returned by this property does not take into account any uncommitted data that might reside in the buffer (this can be checked via sock.newtxlen 486). Therefore, actual free space in the buffer is sock.txfree-sock.newtxlen. Your application will not be able to store more data than this amount.

To achieve asynchronous data processing, use the sock.notifysent 487 method to get on_sock_data_sent 489 event once the TX buffer gains required amount of free space.

.Txlen R/O Property

Function:	For the selected socket (selection is made through sock.num 488) returns total number of *committed* bytes currently found in the TX buffer.
Type:	Word
Value Range:	0-65535, **default**= 0 (0 bytes).
See Also:	---

Details

The data in the TX buffer does not become committed until you use the sock.send 500 method.

Your application may use the sock.notifysent 487 method to get on_sock_data_sent 489 event once the total number of committed bytes in the TX buffer drops below the level defined by the sock.notifysent method.

.Urlsubstitutes

Function:	A comma-separated list of filenames whose extensions will be automatically substituted for ".html" by the internal webserver of your device.
Type:	String
Value Range:	0-40 characters, **default**= ""
See Also:	URL Substitution 468

Details

The substitution will be used only if the resource file with the requested file name is not included in the project directly.

For example, setting this property to "pix1.bmp" will force the webserver to actually process "pix1.html", but only if the file "pix1.bmp" is not found. Data output by the webserver to the browser will still look like a ".bmp" file. For this to work, the "pix1.html" must exist in the project.

This property allows programmatic generation of non-HTML files. In the above example it is possible to generate the BMP file through a BASIC code. There is no other way to do this, since only HTML files are parsed for BASIC code inclusions.

.Varbuffrq Method

Function:	For the selected socket (selection is made through sock.num 488) pre-requests "numpages" number of buffer pages (1 page= 256 bytes) for the VAR buffer of the socket.
Syntax:	**sock.varbuffrq(**numpages **as byte) as byte**
Returns:	Actual number of pages that can be allocated (Byte).
See Also:	---

Part	Description
numpages	Requested numbers of buffer pages to allocate.

Details

The VAR buffer is the buffer that stores the HTTP request string. The socket is unable to process HTTP requests if its VAR buffer has 0 capacity.

Actual allocation happens when the sys.buffalloc 530 method is used. Unlike for TX or RX buffers there is no property to read out actual VAR buffer capacity in bytes. This capacity can be calculated as num_pages*256-16 (or =0 when num_pages=0), where "num_pages" is the number of buffer pages that was GRANTED through the sock.varbuffrq. "-16" is because 16 bytes are needed for internal buffer variables.

Buffer allocation will not work if the socket port to which this buffer belongs is not idle (sock.statesimple 505 is not at 0- PL_SSTS_CLOSED) at the time when sys.buffalloc executes. You can only change buffer sizes of sockets that are idle.

The VAR buffer is only required when you plan to use this socket in the HTTP mode- see sock.httpmode 481 property, also sock.httpportlist 483.

Ssi Object

The ssi. object implements up to four serial synchronous interface (SSI) channels on the general-purpose I/O lines of your BASIC-programmable device. Examples of such interfaces are:

- I2C 516 (http://en.wikipedia.org/wiki/I2C), and YES, ssi. object can handle the acknowledgement bit automatically.
- SPI (http://en.wikipedia.org/wiki/Serial_peripheral_interface).
- Clock/data (example: goto http://www.maxim-ic.com and search for MAX7219 display driver IC);
- Numerous variations on the above.

You can typically get all the necessary data from the IC you want to interface to. Note that some interface designs may not be free to use (i.e. they are patented, etc.).

The common denominator among these interfaces is the presence of the clock line that "paces" data transmission on the data line(s) -- with each clock cycle on the clock line, one bit of data can be transmitted over the data line(s). In contrast, UARTs (managed by the ser. 378 object) are not synchronous -- there is no clock line to synchronize two devices and this is why there is an "A" ("asynchronous") in "UART".

SSI interface links have masters and slaves. The master's role is to generate the clock pulses that slaves will use to synchronize themselves to the master. Only one device can be the master at any given time. **The ssi. object can only work as the master.** That is, it generates the clock for others to use.

Of course, it is possible to communicate with SSI devices by manipulating general-purpose I/O lines of your device directly. This is known as "bit-banging" or "bit-blasting". The advantage of the SSI object is in speed -- the same data exchange will often compete *hundreds of times faster* compared with bit-banging.

A particular channel to work with is selected through the ssi.channel 518 property.

All other properties and methods apply to the currently selected channel.

The ssi.enabled 520 property defines if the channel is disabled or enabled. Necessary configuration 513 of the communication channel can only be performed when ssi.enabled= 0- NO. Actual data exchange 515 is possible only when ssi.enabled= 1- YES.

If you are going to talk to an I2C device, be sure to check out More on I2C 516 topic.

Configuring SSI Channel

In this section:

CLK, DO, and DI Lines

Each communication channel comprises a clock line (CLK), data out (DO) line (a.k.a. "master out, slave in" -- MOSI), and data in (DI) line ("master in, slave out" -- MISO). Ssi.clkmap 518, ssi.dimap 519, and ssi.domap 519 properties define which three general-purpose I/O lines of your device are assigned to serve as the CLK, DI, and DO of the selected channel.

Some interfaces, notably, I2C 516, use a single physical line for moving data in both directions (slave-to-master and master-to-slave). With such interfaces, ssi.dimap and ssi.domap must point to the same physical I/O line. Furthermore, ssi.zmode 522 property must be set to 1- PL_SSI_ZMODE_ENABLED_ON_ZERO and there may be a need for a "pull-up resistor" on the data line (in the I2C world this joint line is called "SDA" and the pull-up resistor is directly specified by relevant datasheets).

Many interfaces require additional interface lines. For example, there is a chip select (CS) line in SPI. The ssi. object does not control such lines -- handle them directly using the io. 294 object.

On devices with unidirectional I/O lines 194, the I/O lines forming your SSI channels must be properly configured (see io.enabled 298): CLK and DO lines must be set as outputs, and the DI line must be set as input. To find out the type of GPIO lines on your device, refer to its platform documentation (for example, EM1000's is here 143).

Baudrate

The ssi.baudrate 517 property defines the speed at which the CLK line will be toggled. The term baudrate is a bit of a misnomer because it may lead one to believe that the baudrate in the ssi. object is something stable and precise. In reality, SSI baudrate only crudely defines the clock speed. The baudrate is also device-specific. Here is the formula for T1000-based devices such as the EM1000 143:

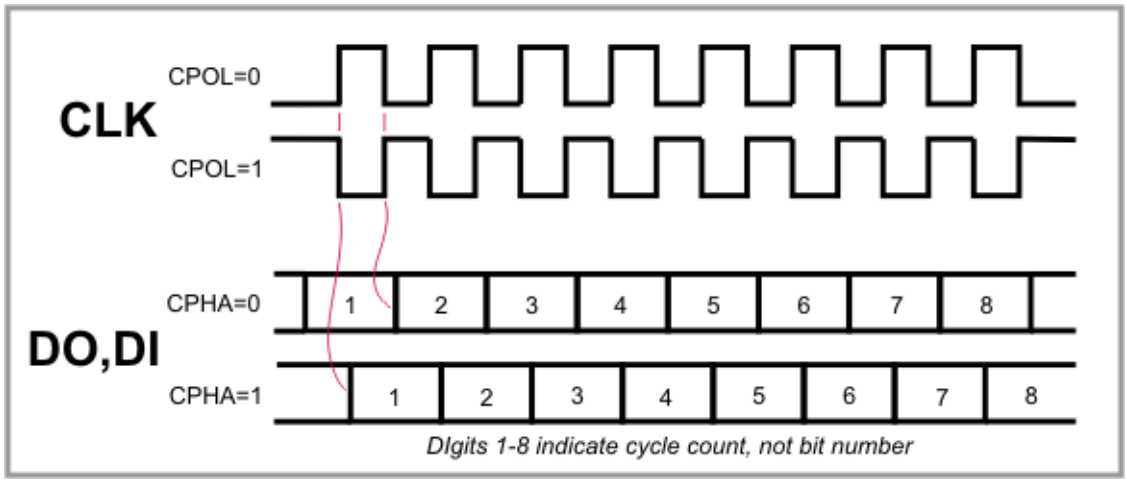
- When the PLL is enabled (sys.currentpll 531 = 1- PL_ON) the clock period can be calculated as 0.8us + ssi.baudrate * 0.112us.
- With the PLL disabled the period will be 8 times longer.

SSI Modes

The ssi.mode 520 property defines one of the four possible modes for the CLK line. These modes directly correspond to the SPI modes as described here: http://en.wikipedia.org/wiki/Serial_peripheral_interface.

SSI mode combines two independent parameters -- CPOL (clock polarity), and CPHA (clock phase):

- Mode 0: CPOL=0, CPHA=0;
- Mode 1: CPOL=0, CPHA=1;
- Mode 2: CPOL=1, CPHA=0;
- Mode 3: CPOL=1, CPHA=1.



Here is a brief description of modes -- some "preparedness" is required to comprehend this:

- CPOL=0: clock line is LOW when idle:

  - CPHA=0: data bits are captured on the CLK's rising edge (LOW-to-HIGH transition) and data bits are propagated on the CLK's falling edge (HIGH-to-LOW transition).

  - CPHA=1: data bits are captured on the CLK's falling edge and data bits are propagated on the CLK's rising edge.

- CPOL=1: clock line is HIGH when idle:

- CPHA=0: data bits are captured on the CLK's falling edge and data bits are propagated on the CLK's rising edge.

- CPHA=1: data bits are captured on the CLK's rising edge and data bits are propagated on the CLK's falling edge.

Note also that there is an additional difference:

- In SSI modes 1 and 3, the first data bit (in either direction) will be present on the DO (DI) line *after* the first transition on the CLK line.
- In SSI modes 0 and 2, the first data bit (in either direction) will be present on the DO (DI) line *before* transitions on the CLK line start.

Direction

The ssi.direction 519 property defines whether the SSI channel will output (input) the data least significant bit first (ssi.direction= 0- PL_SSI_DIRECTION_RIGHT) or most signification bit first (ssi.direction= 1- PL_SSI_DIRECTION_LEFT).

Sending and Receiving Data

Those familiar only with serial 378 ("UART") communications will find this surprising: there are no separate send and receive operations for the ssi. object. When you are sending something out you are also receiving something in at the same time (and vise versa). On the ssi. object's level, these two operations are simultaneous -- with each clock pulse generated on the CLK line, one data bit will be output onto the DO line, and one bit will be recorded from the DI line. The same is true when DO and DI lines are merged.

Of course, on the *operational* level, interactions with a slave SSI device usually consist of sending something (command or data) to the slave device first and then receiving something (data) from it later. Typically, when your device is sending a command, the slave side won't be sending anything meaningful back at the same time, but the SSI channel will still record the data on the DI line. It is your application that should know how to interpret the data received from the slave device.

In the similar matter, whenever you are receiving from the slave device, you are also sending something. Normally, when the slave device is sending you data it will ignore everything you are sending to it.

There are two ways to send and receive data: word-by-word, or as a "string".

For "word-by-word transactions", use ssi.value 521 method. It will simultaneously clock out and clock in up to 16 bits of data. If you are sending less than 16 bits, then the specified number of the rightmost bits will be sent and received.

```
Dim x As word
...
ssi.value(&hCA5A,14) 'When we want to send something out, we can ignore the
"data" we get back
```

```
x=ssi.value(&hFFFF,14) 'When we are expecting to receive data, we send out
'all 1s' (good practice)
x=ssi.value(&hCA5A,14) 'Of course we can do both at the same time
```

For "string transactions", use ssi.str 520 method. String transactions are more sophisticated: you can send (and receive) a string consisting of multiple bytes. These bytes may be handled as "pure" 8-bit words or 9-bit words consisting of eight data bits plus one acknowledgement bit, as required for I2C communications 515 (http://en.wikipedia.org/wiki/I2C). For each byte it outputs, the ssi. object will check for the slave's acknowledgement. Data output will not continue if there was no acknowledgement. You can detect this by comparing the length of the returned string and the length of the string you were trying to send -- if they do not match, then the data exchange ended prematurely.

```
Dim Input_String, Output_String As String
...
Output_String="\x40\x12\xFF"
Input_String=ssi.str(Output_String,1) 'output data, with acknowledgements
enabled
If Len(Output_String)<>Len(Input_String) Then
     'something went wrong
End If
...
```

If you are going to talk to an I2C device, be sure to check out More on I2C 516 topic.

More on I2C 5.3

This topic a continuation of the discussion started in Sending and Receiving Data 515 .

I2C (and many existing variations on it) relies on a single data line (called "SDA") for data transmission in both directions. DO and DI lines of the SSI channel must be joined together. On devices with unidirectional I/O lines 194, the ssi.zmode 522 property must be set to 1- PL_SSI_ZMODE_ENABLED_ON_ZERO (not required on devices with bidirectional I/O lines). The DO line will then operate in the following manner:

- To set the DO line HIGH, your device will disable the output buffer. The line will then float HIGH (because of "weak pull-ups" in the device). In this state, the slave SSI device can safely output its own data onto the SDA.
- To set the DO line LOW, your device will enable the output buffer and set the output to LOW.

It follows then, that if your DO and DI lines are joined together, and you want to receive data from the slave device, you should keep your own output at "all 1s" while the slave device is supposed to send data, like this:

```
...
ssi.str("\x40\x12\xFF",1) 'we anticipate that the slave device will reply
```

```
after we output two bytes of our own data
...
```

Each I2C transaction requires so-called start and stop sequences -- the ssi. object won't handle this so you need to implement this in code. Below is a snippet from a real application. Notice how necessary transitions on the SDA line are performed by setting the DO line LOW and then enabling/disabling its output buffer.

```
...
i2c_start()
ssi.str("\x40\x12\xFF",1)        'output data, with acknowledgements enabled
i2c_stop()
...

'-------------------------------------------------------------
Sub i2c_start()
      io.lineset(SSI_CLK,HIGH)
      io.num=SSI_DO              'set SDA to HIGH first so we can have HIGH-
>LOW transition
      io.state=LOW               'we are manipulating data line through the
OE register
      io.enabled=NO
      io.enabled=YES             'this will set the data output to LOW
End Sub

'-------------------------------------------------------------
Sub i2c_stop()
      io.lineset(SSI_CLK,LOW)    'this will remove the ack bit
      io.num=SSI_DO              'set SDA to LOW first so we can have LOW-
>HIGH transition
      io.state=LOW               'we are manipulating data line through the
OE register
      io.enabled=YES
      io.lineset(SSI_CLK,HIGH)
      io.enabled=NO              'this will set the data output to HIGH
End Sub
```

Properties, Methods

Properties, methods, and events of the ssi object.

.Baudrate Property

Function: For the currently selected SSI channel (see ssi.channel 518) sets/returns the clock rate on the CLK line.

Type: Byte

Value Range: 1-255, the fastest rate is achieved at 1, the slowest rate -- at 255. **Default**= 1 (the fastest rate possible).

See Also: ssi.direction 519, ssi.mode 520, ssi.zmode 522

Details

Actual clock rate is device-dependent. See Baudrate 514 topic for details.

This property can only be changed when ssi.enabled 520 = 0- NO.

It is actually permissible to set the property to 0 -- this will be like setting it to 256 (slowest possible clock rate).

.Channel Property

Function: Sets/returns the number of the currently selected SSI channel (channels are enumerated from 0).

Type: Byte

Value Range: 0-3. **Default** = 0 (channel #0 selected)

See Also: Ssi Object 512

Details

All other properties and methods of this object relate to the channel selected through this property.

.Clkmap Property

Function: For the currently selected SSI channel (see ssi.channel 518) sets/returns the number of the general-purpose I/O line to serve as the clock (CLK) line of this channel.

Type: Enum (pl_io_num, byte)

Value Range: Platform-specific. See the list of pl_io_num constants in the platform specifications. **Default** = PL_IO_NULL (NULL line).

See Also: CLK, DO, and DI Lines 513,

ssi.dimap 519, ssi.domap 519

Details

This property can only be changed when ssi.enabled 520 = 0- NO.

On devices with unidirectional I/O lines, the CLK line must be "manually" configured as output (see io.enabled 298 = 1- YES).

.Dimap Property

Function: For the currently selected SSI channel (see ssi.channel 518) sets/returns the number of the general-purpose I/O line to serve as the data in (DI) line of this channel.

Type: Enum (pl_io_num, byte)

Value Range: Platform-specific. See the list of pl_io_num constants in the platform specifications. **Default**= PL_IO_NULL (NULL line).

See Also: CLK, DO, and DI Lines 513,

ssi.clkmap 518, ssi.domap 519

Details

This property can only be changed when ssi.enabled 520 = 0- NO.

On devices with unidirectional I/O lines, the DI line must be "manually" configured as input (see io.enabled 298 = 0- NO).

.Direction Property

Function: For the currently selected SSI channel (see ssi.channel 518) sets/returns the direction of data input and output (least significant bit first or most significant bit first).

Type: Enum (pl_ssi_direction_options , byte)

Value Range: 0- PL_SSI_DIRECTION_RIGHT **(default)**: data input/ output least significant bit first

1- PL_SSI_DIRECTION_LEFT: data input/output most significant bit first

See Also: Direction 515

Details

This property can only be changed when ssi.enabled 520 = 0- NO.

.Domap Property

Function: For the currently selected SSI channel (see ssi.channel 518) sets/returns the number of the general-purpose I/O line to serve as the data out (DO) line of this channel.

Type: Enum (pl_io_num, byte)

Value Range: Platform-specific. See the list of pl_io_num constants in the platform specifications. **Default**= PL_IO_NULL (NULL line).

See Also: CLK, DO, and DI Lines 513,

ssi.clkmap 518, ssi.dimap 519

Details

This property can only be changed when ssi.enabled 520 = 0- NO.

On devices with unidirectional I/O lines, the DO line must be "manually" configured as output (see io.enabled 298 = 1- YES).

.Enabled Property

Function: Enables/disables the currently selected SSI channel (see ssi.channel 518).

Type: Enum (no_yes, byte)

Value Range: 0- NO **(default)**: disabled
1- YES: enabled

See Also: Ssi Object 512

Details

SSI channel's operating parameters (ssi.baudrate 517, ssi.mode 520, etc.) can only be changed when the channel is disabled.

You can only send and receive the data (ssi.value 521, ssi.str 520) when the channel is enabled.

.Mode Property

Function: For the currently selected SSI channel (see ssi.channel 518) sets/returns the clock mode.

Type: Enum (pl_ssi_modes , byte)

Value Range: 0- PL_SSI_MODE_0 **(default)**
1- PL_SSI_MODE_1
2- PL_SSI_MODE_2
3- PL_SSI_MODE_3

See Also: ssi.baudrate 517, ssi.direction 519, ssi.zmode 522

Details

This property can only be changed when ssi.enabled 520 = 0- NO. For detailed explanation of clock modes see SSI Modes 514.

.Str Method

Function: For the currently selected SSI channel (see ssi.channel 518) outputs a string of byte data to the slave device and simultaneously inputs the same amount of data from the

slave device.

Syntax: **ssi.str(byref** txdata **as string,** ack_bit **as no_yes) as string**

Returns: A string of the same length as txdata or less if the transmission ended prematurely due to the acknowledgement error.

See Also: Sending and Receiving Data 515, ssi.value 521

Part	Description
txdata	The string to send to the slave device.
ack_bit	0- NO: transmit/receive byte data as 8-bit words, without the use of the acknowledgement bit. 1- YES: transmit/receive byte data as 9-bit words comprising 8 bits of data and the acknowledgement bit.

Details

This method can be invoked only when ssi.enabled 520 = 1- YES.

.Value Method

Function: For the currently selected SSI channel (see ssi.channel 518) outputs a data word of up to 16 bits and simultaneously inputs a data word of the same length.

Syntax: **ssi.value(**txdata **as word,** len **as byte) as word**

Returns: 16-bit value containing the data received from the slave device, the number of bits received from the slave device will be equal to the len argument, and these data bits will be right-aligned within the returned 16-bit word.

See Also: Sending and Receiving Data 515, ssi.str 520

Part	Description
txdata	Data to output to the slave device. The number of rightmost bits equal to the len argument will be sent.
len	Number of data bits to send to and receive from the slave device.

Details

Data input/output direction (least significant bit first or most significant bit first) is defined by the ssi.direction 519 property.

This method can be invoked only when ssi.enabled 520 = 1- YES.

.Zmode Property

Function: For the currently selected SSI channel (see ssi.channel 518) sets/returns the mode of the data out (DO) line.

Type: Enum (pl_ssi_zmodes , byte)

Value Range: 0- PL_SSI_ZMODE_ALWAYS_ENABLED **(default)**: the DO line toggles normally by setting the output buffer to LOW or HIGH.

1- PL_SSI_ZMODE_ENABLED_ON_ZERO: for HIGH state, the output buffer of the DO line is turned off, for LOW state, the output buffer is turned on and the line is set to LOW.

See Also: ssi.baudrate 517, ssi.direction 519, ssi.mode 520

Details

This property is only useful on devices with unidirectional I/O lines 194 and in case the DO and DI lines are joined together, as necessary for the I2C and similar interfaces. See More on I2C 516 for more details.

This property can only be changed when ssi.enabled 520 = 0- NO.

Stor Object

The stor object provides access to the non-volatile (EEPROM) memory in which your application can store data that must not be lost when the device is switched off.

The stor.size 525 read-only property tells you the amount of EEPROM memory offered by the stor object. The stor.set 524 is used to write the data to the EEPROM, while the stor.get 523 method is used to read the data from the EEPROM.

Here is a simple example how to save the IP address of your device in the EEPROM:

```
dim s as string
dim x as byte

s=ddval(net.ip) 'this way it will take less space in the EEPROM (only 4
bytes needed)
x= stor.set(s,0) 'write EEPROM

'check result
if x<>len(s) then
      'EEPROM write failure- do something about this!
end if
```

And here is how you read this data back from the EEPROM:

```
net.ip=ddstr(stor.get(0,4))
```

Notice, that with the stor object addresses are counted from 1, not 0. That is, the first memory location has address 1.

Special configuration area

The EEPROM IC of the device is also used to store certain configuration information [197] required by the device. Memory available to your application equals the capacity of the IC minus the size of the special configuration area.

By default, when you access the first byte of the EEPROM you are actually accessing the first memory location above the special configuration area. One property -- stor.base [523] -- returns the size of this offset. On startup, stor.base is equal to the size of the special configuration area, so your program can only access the memory above this area.

You can change the stor.base and access configuration area when you need. For example, you can change the MAC address this way- next time the device boots up it will start using newly set address.

.Base Property

Function:	Sets/returns the base address of the EEPROM from which the area available to your application starts.
Type:	Word
Value Range:	1-<actual memory capacity>, **default**= <size of special configuration area>+1
See Also:	Stor Object [522], stor.size [525]

Details

By default, the value returned by this property is the address of the first EEPROM location just above the special configuration area [197]. For example, if the size of the special configuration area on your platform is 28 bytes then stor.base will return 29 by default.

This Default value makes sure that your application won't overwrite MAC or password. When you are accessing EEPROM memory using stor.set [524] or stor.get [523] methods, you specify the start address. Actual physical address you access is start_address+stor.base-1.

If your application needs to change some parameters in the configuration area you can set the stor.base to 1- this way you will have access to the entire memory.

.Getdata Method (previously .Get)

Function:	Reads data from the EEPROM.
Syntax:	**stor.getdata(**startaddr **as word,** len **as byte) as string**
Returns:	String contains the data read out from the EEPROM.

See Also: Stor Object 522, stor.set 524

Part	Description
startaddr	The starting address in the EEPROM memory (addresses are counted from 1, if you set this parameter to 0 it will be interpreted as 1).
len	Maximum number of bytes to read.

Details

The len parameter defines the *maximum* number of bytes to read from the EEPROM. Actual amount of extracted data is also limited by the capacity of the receiving variable and the starting address (in relation to the memory capacity of the EEPROM). Memory capacity can be checked through the stor.size 525 read-only property. Notice that when the stor.getdata executes, an offset equal to the value of stor.base 523 is added to the startaddr. For example, if the stor.base returns 9 and you do stor.getdata(1,3) then you will actually be reading the data starting from physical EEPROM location 9. If you set the stor.base to 1 you will be able to access the EEPROM right from the physical address 1.

By default, the stor.base is set in such a way as to allow access to the EEPROM starting from the address just above the special configuration area 197 of your device -- for details on what this area actually stores see your device's platform documentation (for example, EM1000's is here 143). By setting the stor.base to 1 you are allowing access to the special configuration area.

With Tibbo Basic release **V2**, this method had to be renamed from .get to .getdata. This is because the period (".") separating "stor" from "getdata" is now a "true" part of the language, i.e. it is recognized as a syntax unit, not just part of identifier. Hence, Tibbo Basic sees "stor" and "get" as separate entities and "get" is a reserved word that can't be used.

.Setdata Method (previously .Set)

Function: Writes data into the EEPROM.

Syntax: **stor.setdata(byref** datatoset **as string,** startaddr **as word) as byte**

Returns: Actual number of bytes written into the EEPROM.

See Also: Stor Object 522, stor.get 523, stor.size 525

Part	Description
datatoset	Data to write into the EEPROM.
startaddr	Starting address in the EEPROM from which the data will be stored (addresses are counted from 1, if you set this parameter to 0 it will be interpreted as 1).

Details

The operation has completed successfully if the value returned by this method equaled the length of the datatoset string. If this is not the case then the write has (partially) failed and there may be two reasons for this: physical EEPROM failure or invalid startaddr (too close to the end of memory to save the entire string).

Notice that when the stor.setdata executes, an offset equal to the value of stor.base 523 is added to the startaddr. For example, if the stor.base returns 9 and you do stor.setdata("ABC",1) then you will actually be reading the data starting from physical EEPROM location 9. If you set the stor.base to 1 you will be able to access the EEPROM right from the physical address 1.

By default, the stor.base is set in such a way as to allow access to the EEPROM starting from the address just above the special configuration area 197 of your device -- for details on what this area actually stores see your device's platform documentation (for example, EM1000's is here 143). By setting the stor.base to 1 you are allowing access to the special configuration area.

With Tibbo Basic release **V2**, this method had to be renamed from .set to .setdata. This is because the period (".") separating "stor" from "setdata" is now a "true" part of the language, i.e. it is recognized as a syntax unit, not just part of identifier. Hence, Tibbo Basic sees "stor" and "set" as separate entities and "set" is a reserved word that can't be used.

.Size R/O Property

Function:	Returns total EEPROM memory capacity (in bytes) for the current device.
Type:	Word
Value Range:	Platform-dependent
See Also:	Stor Object 522

Details

Certain amount of EEPROM memory is occupied by the special configuration section 197 -- for details on what this area actually stores see your device's platform documentation (for example, EM1000's is here 143).

By default, special configuration area is not accessible to the application and is excluded from memory capacity reported by the stor.size. For example, if the EEPROM IC used by this platform has 2048 bytes of data, and the size of the special configuration memory is 8 bytes, then the stor.size will return 2040 by default. At the same time, the default value of stor.base 523 property will be 9, which means that the EEPROM locations 1-8 are occupied by the special configuration area.

If you set the stor.base to 1 (for instance, to edit the MAC address), the stor.size will show the capacity of 2048. In other words, the number this property returns is actual_EEPROM_capacity-stor.base+1.

Sys Object

This is the system object that loosely combines "general system" stuff such as initialization (boot) event, buffer management, system timer, PLL mode, and some other miscellaneous properties and methods.

Overview

Here you will find:

- On_sys_init (boot) event 526
- Buffer Management 526
- System Timer 528
- PLL Management 529
- Serial Number 530
- Miscellaneous stuff 530 such as how to halt execution, reboot, check firmware version.

On_sys_init Event

The sys object provides a very important event- on_sys_init 533. This event is guaranteed to be generated first when your device starts running. Therefore, you should put all your initialization code for sockets, ports, .etc into the event handler for this event:

```
sub on_sys_init
      net.ip="192.168.1.95"
      sock.num=0
      sock.targetip= "192.168.1.96"
      '...and everything else that you may need!
end sub
```

Buffer Management

How to allocate buffer memory

Some objects, such as the sock 421 or ser 378, rely on buffers for storing the data being sent and received. For example, each serial port of the ser object has two buffers- RX and TX, while each socket of the sock object has 6 different buffers.

By default, all buffers have no memory allocated for them, which basically means that they cannot store any data. For related objects to work, these buffers need to be given memory.

Memory is allocated in pages. Each page equals 256 bytes. There is a small overhead for each buffer- 16 bytes are used for internal buffer houskeeping (variables, pointers, etc.). Therefore, if you have allocated 2 pages for a particular buffer, then this buffer's actual capacity will be 2*256-16= 496 bytes.

Buffer memory allocation is a two-step process. First, you request certain number of pages for each port, socket, etc. that you plan to use. This is done through

methods of corresponding objects. For example, the ser.rxbuffrq 415 method requests buffer memory for the RX buffer of one of the serial ports. Similar methods exist for all other buffers of all objects that require buffer memory. Note, that actual memory allocation does not happen at this stage- only your "requests" are collected.

After all requests have been made actual memory allocation is performed by using the sys.buffalloc 530 method. This allocates memory, as per previous requests, for all buffers of all objects. Here is an example:

```
'allocate memory for RX and TX buffers of serial port 0 and socket 0

'make requests
ser.num= 0
ser.rxbuffrq= 5
ser.txbuffrq= 5
sock.num=0
sock.rxbuffrq= 4
sock.txbuffrq= 4

'and now actual allocation
sys.buffalloc
```

Typically, buffer memory allocation is done in the on_sys_init event handler but you don't have to do it this way. In fact, your application can re-allocate buffer memory space according to the changes in the operating mode or other conditions.

Sys.buffalloc could take up to several hundred milliseconds to execute, so it makes sense to use it as little as possible. Hence, request all necessary buffer allocations first, then use the sys.buffalloc once to finish the job.

Buffer (re)allocation for a specific object will only work if the corresponding object or part of the object to which this buffer belongs is idle. "Part" refers to a particular serial port of the ser object, or particular socket of the sock object, etc. to which the buffer you are trying to change belongs. "Idle" means different things for different objects: ser.enabled 405 = 0- NO for the serial port, sock.statesimple 505 = 0- PL_SSTS_CLOSED for the socket, etc.

Intelligent memory allocation basing on what's available

Memory is not an infinite resource and you will not always get as much memory as you have requested. Methods such as the ser.rxbuffrq 415 actually return the amount of memory that can be allocated:

```
dim x as byte
x=ser.rxbuffrq(5) 'x could get less than 5, which means that you got less
memory than you have asked for
```

Two properties of the sock object were implemented to help you allocate memory intelligently, basing on what is available.

The sys.totalbuffpages 536 read-only property tells you how many buffer pages are there on your system in total. This is defined by the amount of physical variable memory (RAM) available minus memory required to store your project's variables

(so, as your project grows available buffer memory shrinks).

The sys.freebuffpages 531 read-only property tells you the amount of free (not yet allocated) buffer pages.

Here is an example in which we allocate memory equally between the TX and RX buffers of four serial ports and four sockets:

```
dim x,f as byte

x=sys.totalbuffpages/16 'will work correctly if we haven't allocated
anything yet

for f=0 to 3
      ser.num= f
      ser.rxbuffrq(x)
      ser.txbuffrq(x)
      sock.num= f
      sock.rxbuffrq(x)
      sock.txbuffrq(x)
next f
```

In the above example we do not check what individual buffer requests return because we have already calculated each buffer's size basing on the total number of buffer pages available.

System Timer

The system object provides a timer event- on_sys_timer 533- that is generated periodically. The period depends on the platform. If the platform doesn't have a sys.onsystimerperiod 533 property, then this period is fixed at 0.5 seconds. If the platform supports this property, then 0.5 second is the default period and the property allows your program to adjust it.

If your software has some periodic tasks you can put the code into the on_sys_timer event handler. Notice that when generated, this event goes into the queue and waits in line to be processed, just like all other events. Therefore, you cannot expect great accuracy in the period at which the on_sys_timer event handler is entered. You can just expect that *on average* you will be getting this event every 0.5 seconds.

There is also a sys.timercount 536 read-only property. Timer counter is a free-running counter that is initialized to 0 when your device is powered up and increments every 0.5 seconds. The sys.timercount is useful in determining elapsed time, for instance, when you are waiting for something.

Here is an example. Supposing, you are supposed to wait for the serial data to arrive, but you are not willing to wait more than 10 seconds:

```
dim w as word
...
...
```

```
w=sys.timecount 'memorize time count at the beginning of waiting for serial
data
while ser.rxlen=0
      'nope, no data yet, do we still have time?
      if sys.timercount-w>20 then goto enough_waiting 'we quit after 10
seconds
      doevents 'polite waiting includes this
wend

...
...
```

The above example does not represent best coding style. Generally speaking, this is not a great programming but sometimes you just have to wait in a cycle.

PLL Management

PLL is a module of the device that transforms the clock generated by onboard crystal into higher frequency (x8 of the base for Tibbo devices). When the PLL is on, the device runs at 8 times the base frequency, when the PLL is off, the device runs at a "native" frequency of the crystal. Naturally, the device is 8 times faster (and consumes almost as much more power) when the PLL is on.

Not all Tibbo devices have PLLs- to find out if yours does, refer to its platform documentation (for example, EM1000's is here 143).

When a certain platform supports PLL, it will have a sys.currentpll 531 read-only property and sys.newpll 532 method to control the PLL. Due to the nature of PLL operation it is impossible to switch it on and off while the CPU is executing the firmware. The PLL needs time to "stabilize" its output frequency and it is not safe to let this happen when the CPU is running. Instead, the PLL is toggled when the CPU is in the reset state.

To change PLL mode, request new state through the sys.newpll method, then self-reset the device through the sys.reboot 534 method. After the reboot the device emerges from reset with new PLL state (and PLL frequency already stabilized).

Here is a code example that makes sure that your device is running with PLL off:

```
sub on_sys_init
      if sys.currentpll=YES then
            sys.newpll(OFF)
            sys.reboot
      end if
end sub
```

External resets- power-up and RST pin reset (reset button reset)- set the PLL to default state (typically ON). On some devices there is a hardware jumper that defines the post-external reset state of the PLL.

Notice that PLL mode affects other objects- for example, baudrates of serial ports 378 (this is why there is a ser.div9600 404 property) and frequency generated by the beep 232 object.

Serial Number

Most Tibbo devices feature unique serial numbers. The length and contents of serial numbers are device-dependent. For more information refer to your device's platform documentation (for example, EM1000's is here 143).

Miscellaneous

The sys.version 536 property returns the version of the TiOS (firmware) running on your device.

The sys.halt 532 method can be used to halt the execution of your program. For example, you can use this to halt the program when the Ethernet interface failure is detected:

```
...
if net.failure= YES then
      sys.halt
end if
...
```

The sys.reboot 534 method causes the device to reboot. This may be needed, for instance, to change the mode of the PLL 529.

The sys.runmode 534 read-only property informs whether the device is running in the release or debug mode. This data can be used by the program to exhibit different behavior- for example report all errors in the debug mode and try to "manage on its own" in the release mode (for more info see Two Modes of Target Execution 27).

The sys.resettype 535 read-only property tells you what caused the most recent reset experienced by your device.

Properties, Methods, Events

This section provides an alphabetical list of all properties, methods, and events of the sys object.

.Buffalloc Method

Function:	Allocates buffer memory as previously requested by "buffrq" methods of individual objects (such as ser.rxbuffrq 415).
Syntax:	**sys.buffalloc**
Returns:	---
See Also:	sys.totalbuffpages 536, sys.freebuffpages 531

Details

Call this method after requesting all buffers you need through methods like

ser.txbuffrq 419 and sock.cmdbuffrq 476.

This method takes significant amount of time (100s of milliseconds) to execute, during which time the device cannot receive network packets, serial data, etc. For certain interfaces like serial ports some incoming data could be lost.

Buffer (re)allocation for a specific object will only work if the corresponding object or part of the object to which this buffer belongs is idle. "Part" refers to a particular serial port of the ser 378 object, or particular socket of the sock 421 object, etc. to which the buffer you are trying to change belongs. "Idle" means different things for different objects: ser.enabled 405 = 0- NO for the serial port, sock.statesimple 505 = 0- PL_SSTS_CLOSED for the socket, etc.

.Currentpll R/O Property (Selected Platforms Only)

Function:	Returns current PLL mode of the device
Type:	Enum (no_yes, byte)
Value Range:	0- NO: PLL is off, the device runs at low speed with reduced power consumption. 1- YES: PLL is on, the device runs at maximum speed, x8 faster than low speed.
See Also:	sys.newpll 532

Details

After the external reset the device typically boots with PLL on. You can switch PLL off and on programmatically by using the sys.newpll 532 method and then "self-resetting" the device using the sys.reboot 534 method.

Actual PLL mode change only takes place after you "self-reset" the device using sys.reboot 534 method or the device self-resets due to some other reason (for instance, there is a self reset after a new BASIC application upload, or when you hit "restart" button in TIDE). External resets- power-up and RST pin reset (reset button reset)- set the PLL to default state (typically ON). On some devices there is a hardware jumper that defines the post-external reset state of the PLL.

Not all Tibbo devices have PLL- to find out if yours does, refer to its platform documentation (for example, EM1000's is here 143).

.Freebuffpages R/O Property

Function:	Returns the number of free (not yet allocated) buffer pages (one page= 256 bytes).
Type:	Byte
Value Range:	0-255
See Also:	sys.totalbuffpages 536

Details

Only changes after the sys.buffalloc 530 method is used. "Preparatory" methods like ser.rxbuffrq 415 do not influence what this property returns.

.Halt Method

Function:	Stops your program execution (halts VM).
Syntax:	**sys.halt**
Returns:	---
See Also:	sys.reboot 534

Details

Causes the same result as when you press PAUSE in TIDE 15 during the debug session. Once this method has been used, there is no way for your device to resume execution without your help.

.Newpll Method (Selected Platforms Only)

Function:	Sets new state of the PLL.
Syntax:	**sys.newpll(**newpllstate **as off_on)**
Returns:	---
See Also:	sys.currentpll 531

Part	Description
newpllstate	Specifies what state the PLL will be in after the device emerges from internal reset: 0- OFF: PLL will be off, the device will emerge from reset at low speed with reduced power consumption 1- ON: PLL will be on, the device will emerge from reset at maximum speed, x8 faster than low speed

Details

Actual PLL mode change only takes place after you "self-reset" the device using sys.reboot 534 method or the device self-resets due to some other reason (for instance, there is a self reset after a new BASIC application upload, or when you hit "restart" button in TIDE). External resets- power-up and RST pin reset (reset button reset)- set the PLL to default state (typically ON). On some devices there is a hardware jumper that defines the post-external reset state of the PLL.

Not all Tibbo devices have PLL- to find out if yours does, refer to its platform documentation (for example, EM1000's is here 143).

Notice that PLL mode affects other objects- for example, baudrates of serial ports 378 (this is why there is a ser.div9600 404 property) and frequency generated by the beep 232 object.

On_sys_init Event

Function: First event to be generated when your device boots up.

Declaration: **on_sys_init**

See Also: ---

Details

Typically, initialization code for you application is placed here.

On_sys_timer Event

Function: Periodic event.

Declaration: **on_sys_timer**

See Also: sys.timercount 536

Details

Multiple on_sys_timer events may be waiting in the event queue. On_sys_timer event is not generated when the program execution is PAUSED (in debug mode 27).

New in **V1.2**. If the platform does not support the sys.onsystimerperiod 533 property, then the interval of this event generation is fixed at 0.5 seconds. If the platform does support this property, then the period is adjustable.

.Onsystimerperiod Property (Selected Platforms Only)

Function: New in **V1.2**. Sets/returns the period for the on_sys_timer 533 event generation expressed in 10ms intervals.

Type: Byte

Value Range: 0-255. **Default**= 50 (500ms).

See Also: ---

Details

Defines, in 10ms increments, the period at which the on_sys_timer 533 event will be generated. Platforms that do not support this property have the period fixed at 0.5

seconds.

.Reboot Method

Function: Causes your device to reboot.

Syntax: **sys.reboot**

Returns: ---

See Also: sys.currentpll 531, sys.runmode 534, sys.resettype 535, and sys.halt 532

Details

After the device reboots it will behave as after any other reboot: enter PAUSE mode if your program was compiled for debugging, or start execution if the program was compiled for release (see Two Modes of Target Execution 27).

The PLL mode will change after the reboot if you requested the changed through sys.newpll 532 method.

.Runmode R/O Property

Function: Returns current run (execution) mode.

Type: Enum (pl_sys_mode, byte)

Value Range: 0- PL_SYS_MODE_RELEASE: debugging is not possible, application execution starts immediately after device powers up. Severe errors such as "divizion by zero" are ignored and do not stop execution.

1- PL_SYS_MODE_DEBUG: debug mode in which it is possible to cross-debug the application (under the control of TIDE 15 software). Application execution is not started automatically after the power up. Severe errors such as "divizion by zero" halt execution.

See Also: sys.currentpll 531, sys.newpll 532, sys.resettype 535, and sys.halt 532

Details

For some programs, it may be useful to know if the program is currently executing in Debug Mode or Release Mode(see Two Modes of Target Execution 27).

Serialnum R/O Property (Selected Platforms Only)

Function: Returns a string containing the serial number of the device.

Type: String

Value Range:	Platform-dependent
See Also:	Serial Number 530

Details

To find out if there is a serial number, refer to your device's platform documentation (for example, EM1000's is here 143).

Setserialnum Method (Selected Platforms Only)

Function:	Sets the serial number of the device.
Syntax:	**sys.setserialnum(byref** str **as string) as ok_ng**
Returns:	0- OK: The serial number was set successfully. 1- NG: Serial number programming failed.
See Also:	Serial Number 535

Details

To find out if there is a serial number, refer to your device's platform documentation (for example, EM1000's is here 143).

.Resettype R/O Property

Function:	Returns the type of the most recent hardware reset.
Type:	Enum (pl_sys_reset_type, byte)
Value Range:	0- PL_SYS_RESET_TYPE_INTERNAL: The most recent reset was generated internally. 1- PL_SYS_RESET_TYPE_EXTERNAL: The most recent reset was generated externally.
See Also:	sys.currentpll 531, sys.runmode 534, sys.reboot 534, and sys.halt 532

Details

Internal resets are generated when the device self-reboots. This can be caused by the execution of sys.reboot 534 in your application, or command from TIDE.

External resets are the ones that are caused by power-cycling (turning the device off and back on) or applying a reset pulse to the RTS line of the device (pushing reset button).

.Timercount R/O Property

Function: Returns the time (in half-second intervals) elapsed since the device powered up.

Type: Word

Value Range: 0-65535

See Also: on_sys_timer 533

Details

Once this timer reaches 65535 it rolls over to 0.

.Totalbuffpages R/O Property

Function: Returns the total amount of memory pages available for buffers (one page= 256 bytes).

Type: Byte

Value Range: 0-255

See Also: sys.buffalloc 530, sys.freebuffpages 531

Details

Calculated as total available variable memory (RAM) in pages minus number of pages required to store variables of the current project.

.Version R/O Property

Function: Returns firmware (TiOS) version string.

Type: String

Value Range: Whatever is set in firmware, for example "<EM202-1.00.00>"

See Also: ---

Details

Wln Object

The wln. object represents the Wi-Fi interface of your device. It is through this

object that you find available Wi-Fi networks and select the one to associate with. You can also create an ad-hoc network of your own and have other stations connect to it.

The wln. object is not responsible for actual data communications over the Wi-Fi -- this is the job of the sock. 421 object.

On platforms with Wi-Fi support, you will find Wi-Fi interface listed or available on the following sock. object's properties:

- Sock.availableinterfaces 475
- Sock.allowedinterfaces 474
- Sock.currentinterface 478
- Sock.targetinterface 506

In the task it performs, the wln. object is similar to the net 358 object, which controls another interface -- the Ethernet. In comparison, the wln. object is *much* more complex.

You can avoid dealing with the complexity of Wi-Fi by using our WLN library 703. The library handles the tasks of bringing up the Wi-Fi unterface, finding the specified access point, and associating with it using selected security protocol. The library also offers other convenient "services".

The wln object allows you to:

- Scan for available networks and obtain their parameters such as name, channel, mode, etc. "Hidden networks" (those that do not broadcast their SSID) can also be found.
- Set WEP, WPA-PSK, or WPA2-PSK security modes and related keys. Part of required WPA/WPA2 functionality is implemented in the WLN library 703. That is, the wln. object and WLN library work together to achieve WPA-PSK and WPA2-PSK support.
- Associate with one of the networks (at a time) or form your own "ad-hoc" network on a desired channel.
- Monitor received signal strength.
- Detect disassociation from the network.
- Detect Wi-Fi interface power-down or malfunction.

The wln. object works with dedicated hardware -- the GA1000 add-on module. This add-on hardware is described in our Programmable Hardware Manual.

Overview 18.1

If you are new to Wi-Fi communications, then we recommend you to read the Wi-Fi Parlance Primer 538 that will introduce you to some important Wi-Fi lingo.

Wln Tasks 539 topic explains the basics of interaction with the wln. object.

The rest of the manual follows the natural sequence of steps that you usually take when working with and through the Wi-Fi interface (Wln State Transitions 542 topic expands on the subject). The steps are as follows:

- Brining up Wi-Fi interface 543
- Scanning for Wi-Fi networks 550
- Setting WEP or WPA security 552
- Associating with selected network 554 (or creating own ad-hoc network 555)
- Communicating via Wi-Fi interface 555
- Disassociating from the network 556 (or terminating own ad-hoc network 556)
- Detecting disassociation or offline state 556

Wi-Fi Parlance Primer

If you are new to Wi-Fi networking, this section will give you a bit of knowledge on the highly specialized jargon used in the Wi-Fi world. We also provided useful links that will lead you to a lot more info on the subject.

Here are the abbreviations and terms that are used in the Wi-Fi world:

- **BSS** stands for Basic Service Set (ah, now it is clear!). In simple words, this is a wireless network created by a single access point (http://en.wikipedia.org/wiki/Wireless_access_point), but not always. It all depends on the...
- **BSS mode**. Wi-Fi networks can operate in either the infrastructure mode, or ad-hoc mode. In the infrastructure mode, a wireless access point is used to create, control and regulate the network. In the ad-hoc mode, there is no access point and wireless devices communicate directly with each other. This is called "IBSS" (Independent Basic Service Set), more on this here: http://en.wikipedia.org/wiki/Independent_Basic_Service_Set.
- **SSID** stands for Service Set Identifier. In simple words, the SSID is a name of the wireless network. In case of the infrastructure network, this name is preset on the access point during its configuration. You can read more on SSIDs here: http://en.wikipedia.org/wiki/BSSID#Service_set_identifier_.28SSID.29.
- **BSSID** stands for Basic Service Set Identifier. This is the "MAC address of the wireless network". When your network is built on the access point, this is the MAC address of this access point. For ad-hoc networks the BSSID is selected with certain randomness built in. More on this here: http://en.wikipedia.org/wiki/BSSID#Basic_service_set_identifier.
- **Channel**. Wi-Fi devices operate on one of 14 preset frequencies. Channel refers to the channel number, not the actual frequency used. Depending on the locale, you can be restricted to fewer channels: http://en.wikipedia.org/wiki/List_of_WLAN_channels
- **RSSI**. Stands for Received Signal Strength Indication. This is a measure of the quality of RF signal received from the wireless network (or peer). More on this here: http://en.wikipedia.org/wiki/Rssi
- **WEP** stands for Wired Equivalent Privacy, a widely used method of protecting Wi-Fi networks from eavesdropping and unauthorized access. The name carries a bit of a wishful thinking, as it has been clearly demonstrated that WEP is rather weak

and can be easily defeated. Read on here: http://en.wikipedia.org/wiki/Wired_Equivalent_Privacy.

- **WPA** means Wi-Fi Protected Access. This is a security protocol that exists in two versions -- WPA and WPA2. WPA was developed in response to serious weaknesses discovered in the WEP standard. Tibbo devices support "personal" WPA protocols WPA-PSK and WPA2-PSK. Read about WPA here: http://en.wikipedia.org/wiki/Wi-Fi_Protected_Access.

Wln Tasks

True to the non-blocking operation philosophy 4 of the entire system, the wln object does not stall the entire Tibbo Basic application execution to wait for the Wi-Fi interface to complete required operation ("task"). You program gives the wln. object a task to perform, and then it is free to go and do other things.

There are nine wln. tasks:

- Setting TX power 550 (initiated by wln.settxpower 570 method). This is an "immediate" task.
- Passively Scanning for Wi-Fi networks 550 (initiated by wln.scan 567 method).
- Actively Scanning for Wi-Fi networks 550 (initiated by wln.activescan 556 method).
- Setting WEP mode and key 553 (initiated by wln.setwep 570 method). This is an "immediate" task.
- Setting WPA/WPA2 mode and key 553 (initiated by wln.setwpa 571 method). This is an "immediate" task.
- Associating with selected network 554 (initiated wln.associate 557 by method).
- Creating own ad-hoc network 555 (initiated by wln.networkstart 564 method).
- Disassociating from the network 556 (initiated by wln.disassociate 561 method).
- Terminating own ad-hoc network 556 (initiated by wln.networkstop 564 method).

Three tasks on the list -- wln.settxpower 570, wln.setwep 570, and wln.setwpa 571 -- are so-called immediate tasks. They complete as soon as they are started. If the execution advances to the next statement in the program then you know that these tasks are done with.

All remaining tasks take time to complete, and they complete asynchronously with respect to the program execution. The following example shows a wrong way of tasking:

```
'THIS WON'T WORK!
...
wln.scan("NET1")
wln.associate(wln.scanresultbssid,"NET1",wln.scanresultchannel,
wln.scanresultbssmode) 'this task will be skipped over!
```

Here is how you should do this: use the wln.task 572 read-only property and wait until the previous task is completed.

```
'A BETTER WAY
...
wln.scan("NET1")
```

```
while wln.task<>PL_WLN_TASK_IDLE 'waiting for the task to complete...
wend
...
wln.associate(wln.scanresultbssid,"NET1",wln.scanresultchannel,
wln.scanresultbssmode)
while wln.task<>PL_WLN_TASK_IDLE 'waiting for the task to complete...
wend
...
```

The above approach still needs some refinement. Just making sure that the previous task has competed will not guarantee that your next task will be accepted. This is because some tasks can only be accepted under certain conditions. For example, you can't associate while you are already associated. Try this, and wln.associate 557 will return 1- REJECTED.

```
'GOOD PROGRAMMING MEANS ANTICIPATING PROBLEMS
While wln.task<>PL_WLN_TASK_IDLE 'waiting for the previous task to
complete...
Wend
...
If wln.associate(wln.scanresultbssid,"NET1",wln.scanresultchannel,
wln.scanresultbssmode)<>ACCEPTED Then
   'We already made sure that the previous task was completed.
   'Hence, there is a more 'fundamental' reason for the rejection!
End If
```

Now, this is still not all. "Task completed" is not equal to "task completed successfully". In the above example, we were trying to associate with the "NET1" network. Now, have we actually succeeded? Find out by testing the value of wln.associationstate 558! For every task that may result in failure there is a way to know if the execution was successful or not.

```
'MORE CHECKING
While wln.task<>PL_WLN_TASK_IDLE 'waiting for the previous task to
complete...
Wend
...
If wln.associate(wln.scanresultbssid,"NET1",wln.scanresultchannel,
wln.scanresultbssmode)<>ACCEPTED Then
   'Handle this...
End If
While wln.task<>PL_WLN_TASK_IDLE 'waiting for association to complete...
Wend
'did we succeed?
If wln.associationstate<>PL_WLN_ASSOCIATED Then
   'something went wrong...
End If
...
```

One problem with the code in the above examples is that it is, essentially, blocking. Your application is not doing anything useful while the Wi-Fi interface is associating.

To take advantage of the event-driven nature of the system, you can base your execution flow on the on_wln_task_complete 565 event which is generated each time a task is completed. Completed_task argument of the event handler carries the code of the event that has been completed. Therefore, you can advance through steps in this manner:

```
'THIS CODE TAKES FULL ADVANTAGE OF THE EVENT-DRIVEN NATURE OF THE SYSTEM

'-----------------------------------------
Sub On_sys_init
   ...
   wln.scan("CISCO1") 'start the task and don't wait
End Sub

'-----------------------------------------
Sub On_wln_task_complete(completed_task As pl_wln_tasks)
   Select Case completed_task

   Case PL_WLN_TASK_SCAN:
      'wln.scan() completed
      If wln.scanresultssid="" Then
         'passive scan failed to reveal the network, let's try active
scanning
         wln.activescan("CISCO1")
      Else
         'network discovered
         goto associate_now
      End If

   Case PL_WLN_TASK_ACTIVESCAN:
      'wln.activescan() completed
      If wln.scanresultssid="" Then
         'network still not found -- handle this
         ...
      End If
associate_now:
      wln.associate(wln.scanresultbssid,"CISCO1",wln.scanresultchannel,
wln.scanresultbssmode)

   Case PL_WLN_TASK_ASSOCIATE:
      'wln.associate() completed, proceed in this manner...
      ...
   End Select
End Sub

'-----------------------------------------
Sub On_wln_event(wln_event As pl_wln_events)
   'here we catch hardware problems and disassociations -- also
asynchronously
End Sub
```

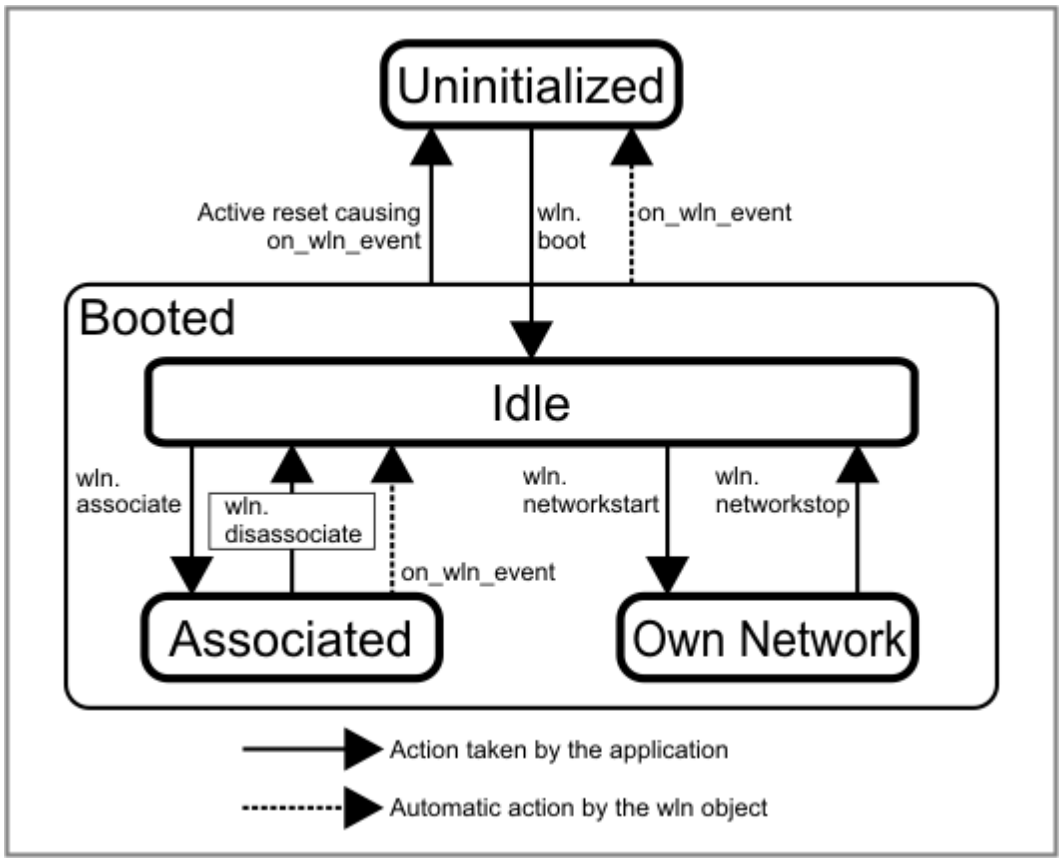
Notice the on_wln_event 565 in the code above. It allows us to catch "problems".

Wln State Transitions

The Wi-Fi hardware may be in one of the following states:

- Uninitialized state (wln.enabled 562 = 0- NO).
- Booted and idle (wln.enabled= 1- YES and wln.associationstate 558 = 0- PL_WLN_NOT_ASSOCIATED).
- Booted and associated with a network (wln.enabled= 1- YES and wln.associationstate= 1- PL_WLN_ASSOCIATED).
- Booted and running its own ad-hoc network (wln.enabled= 1- YES and wln.associationstate= 2- PL_WLN_OWN_NETWORK).

The following diagram details possible state transitions.



The only way to advance from the uninitialized state into the booted state is through a successful boot. The process is described in Bringing Up Wi-Fi Interface 543. The key method for the process is wln.boot 558.

There is no special method for powering down. Your application can only hardware-reset 546 the GA1000, after which the boot process can be repeated. The on_wln_event 565 is generated when the GA1000 goes offline, either as a result of a deliberate reset, or in case the Wi-Fi hardware malfunctions or gets disconnected.

Transition between the idle and associated states happens as a result of successful association. This is detailed in Associating With Selected Network 554. The key method is wln.associate 557.

The wln.disassociate 561 method can be used to force disassociation. The wln object also detects the loss of association automatically, i.e. when the network in

question "disappears". In both cases, the on_wln_event 565 event is generated.

The Wi-Fi interface can also create its own ad-hoc network 555, which is achieved through the wln.networkstart 564 method. Terminating Own Ad-hoc Network 556 explains how to end this (in short, use wln.networkstop 564).

Notice that you cannot be associated and run your own network at the same time. These states are mutually exclusive.

Brining Up Wi-Fi Interface

The easiest way to make the Wi-Fi interface work is by calling wln_start() 724 of the WLN library 703. This will save you a ton of effort, seriously!

If you can't or won't use the library, here is the sequence of steps that you just have to take in order to bring up the Wi-Fi interface:

- Configure interface lines 545
- Reset Wi-Fi module 546
- Select the domain 547
- Allocate buffer memory 547
- Set MAC address (optional) 548
- Boot up the GA1000 add-on module 549
- Set the IP, gateway IP, and netmask 549
- Set TX power (550 *really* 550 optional) 550

The following is a simplified sample code that demonstrates the process. Typically, it would be called from the on_sys_init 533 event handler, but you can actually call it from anywhere in your application. You can also call this code repeatedly and even after the Wi-Fi interface has already been running. We call the code simplified because it does not check for any error conditions.

```
'BRINGING UP THE WI-FI MODULE (SIMPLIFIED)

#If PLATFORM_ID=EM500W
      #define WLN_RESET_MODE 1 'reset is controlled by the combination of CS
and CLK
      'there is no need to map CS, DI, DO, and CLK lines because they are
fixed
#elif PLATFORM_ID=EM1206W
      #define WLN_RESET_MODE 0 'there is a dedicated reset line
      #define WLN_RST PL_IO_NUM_11
      #define WLN_CS PL_IO_NUM_15
      #define WLN_DI PL_IO_NUM_12
      #define WLN_DO PL_IO_NUM_13
      #define WLN_CLK PL_IO_NUM_14
#else
      #define WLN_RESET_MODE 0 'there is a dedicated reset line
      #define WLN_RST PL_IO_NUM_51
      #define WLN_CS PL_IO_NUM_49
      #define WLN_DI PL_IO_NUM_52
      #define WLN_Do PL_IO_NUM_50
      #define WLN_CLK PL_IO_NUM_53
#endif
```

```
'-------------------------------------------------------------------------
-

      '----- configure interface lines -----
      '(on platforms with fixed mapping this will have no effect and do no
harm)
      wln.csmap=WLN_CS
      io.num=WLN_CS
      io.enabled=YES
      wln.dimap=WLN_DI
      wln.domap=WLN_DO
      io.num=WLN_DO
      io.enabled=YES
      wln.clkmap=WLN_CLK
      io.num=WLN_CLK
      io.enabled=YES
      io.num=WLN_RST
      io.enabled=YES

      '----- reset Wi-Fi module -----
      #If WLN_RESET_MODE
            'reset is controlled by the combination of CS and CLK
            io.lineset(wln.csmap,HIGH)
            io.lineset(wln.clkmap,LOW)
            io.lineset(wln.clkmap,HIGH)
      #Else
            'there is a dedicated reset line
            io.num=WLN_RST
            io.state=LOW
            io.state=HIGH
      #endif

      'in case we called wln_init() after it has already been up and running
      While wln.enabled=YES
      Wend

      '----- set the domain -----
      wln.domain=PL_WLN_DOMAIN_FCC

      '----- allocate buffers -----
      wln.buffrq(5)
      sys.buffalloc

      '----- set MAC address (optional) -----
      wln.mac="0.100.110.120.130.140"

      '----- boot up the GA1000 -----
      romfile.open("ga1000fw.bin")
      wln.boot(romfile.offset)

      '----- setup the IP, gateway, netmask -----
      wln.ip="192.168.1.86"
      wln.gatewayip="192.168.1.1"
      wln.netmask="255.255.255.0"

      '----- set TX power (REALLY optional) -----
      wln.settxpower(15)
      while wln.task<>PL_WLN_TASK_IDLE
      wend
```

Configuring Interface Lines

The GA1000 add-on interacts with your BASIC-programmable device through an SPI interface. The SPI interface has four signals: chip select (CS), clock (CLK), data in (DI), and data out (DO). On the EM500 138, DI and DO are combined together (this saves one GPIO line on a module that has very few of them). Connecting GA1000 201 shows related schematic diagrams.

All Tibbo devices except the EM500 allow remapping of SPI lines. That is, any four GPIO lines of your device can be chosen to control the GA1000. Wln.csmap 560, wln.clkmap 560, wln.dimap 560, and wln.domap 562 properties exist for the purpose. There is no remapping on the EM500 and manipulating these properties makes no difference for it.

The GA1000 also has a reset (RST) line. As Connecting GA1000 201 explains, the RST line can be driven by a dedicated GPIO line of your device, or through a "joint effort" of CLK and DO, which, again, spares one GPIO pin of your device. There isn't any property to remap the RST line. This is because the RST is supposed to be controlled by your application directly. It exists outside of the wln. object's realm, so choose any GPIO to control it.

This said, some Tibbo hardware has all the choices made for you already. We are talking about the NB1000 board (part of DS100x) and the EM1206EV board. On these boards, all the necessary hardware connections to the GA1000 are already there, so your mapping just has to follow them. With the exception of the EM500, you have a complete freedom to map any way you want on all other devices, but why would you? We recommend you to follow this scheme:

```
#If PLATFORM_ID=EM500W
      #define WLN_RESET_MODE 1 'reset is controlled by the combination of CS
and CLK
      'there is no need to map CS, DI, DO, and CLK lines because they are
fixed
#elif PLATFORM_ID=EM1206W or PLATFORM_ID=DS1101W or PLATFORM_ID=DS1102W
'also applies to EM1206EV board
      #define WLN_RESET_MODE 0 'there is a dedicated reset line
      #define WLN_RST PL_IO_NUM_11
      #define WLN_CS PL_IO_NUM_15
      #define WLN_DI PL_IO_NUM_12
      #define WLN_Do PL_IO_NUM_13
      #define WLN_CLK PL_IO_NUM_14
#Else 'for all other devices including NB1010
      #define WLN_RESET_MODE 0 'there is a dedicated reset line
      #define WLN_RST PL_IO_NUM_51
      #define WLN_CS PL_IO_NUM_49
      #define WLN_DI PL_IO_NUM_52
      #define WLN_Do PL_IO_NUM_50
      #define WLN_CLK PL_IO_NUM_53
#endif
```

The above sets three schemes: one for the EM500 138 module and related devices, the second one for the EM1206 158, DS1101 168, DS1102 174, and related devices, and the third one for all other devices including the EM1000 143, EM1202 151, etc.

Here is the code snippet that prepares GPIO lines:

```
      '----- configure interface lines -----
```

```
      '(on platforms with fixed mapping this will have no effect and do no
harm)
      wln.csmap=WLN_CS
      io.num=WLN_CS
      io.enabled=YES
      wln.dimap=WLN_DI
      wln.domap=WLN_DO
      io.num=WLN_DO
      io.enabled=YES
      wln.clkmap=WLN_CLK
      io.num=WLN_CLK
      io.enabled=YES
      io.num=WLN_RST
      io.enabled=YES
```

Note that mapping can't be changed when the Wi-Fi hardware is already booted 549 (i.e. wln.enabled 562 = 1- YES).

Applying Reset

Once the RST line has been properly configured 545, you can hardware-reset the GA1000 at any time by executing the following simple code. Note that the GA1000 *has* to be reset after the power up of your device. Proper hardware reset is *not* optional!

If your design has no GPIO line to spare, the reset signal can be derived from the combination of CS and CLK line signals (check Connecting GA1000 201, diagram B). When there is a dedicated RST line, this line is manipulated directly.

Notice also the *while...wend* loop. It is there for proper GA1000 reset after it has already been up and running. When the hardware reset is applied, the GA1000 will go offline, and the wln. object will detect this in a matter of milliseconds. Your program should not try to work with the wln. object again until the wln.enabled 562 property goes to 0- NO in response to the hardware reset. Your code can stay and wait for this (as shown below) or rely on the on_wln_event 565. Once the event is triggered, you are free to repeat the whole process of bringing up Wi-Fi interface 543 .

```
      '----- reset Wi-Fi module -----
      #If WLN_RESET_MODE
            'reset is controlled by the combination of CS and CLK
            io.lineset(wln.csmap,HIGH)
            io.lineset(wln.clkmap,LOW)
            io.lineset(wln.clkmap,HIGH)
      #Else
            'there is a dedicated reset line
            io.num=WLN_RST
            io.state=LOW
            io.state=HIGH
      #endif

      'in case we called wln_init() after it has already been up and running
      While wln.enabled=YES
      Wend
```

Selecting Domain

Wireless communications and channels are tightly regulated in every country on Earth, and this applies to Wi-Fi networks as well. Not every one of 14 pre-defined Wi-Fi frequencies is allowed to be used in every country. It is your responsibility to set a correct "domain" for your Wi-Fi device. This is done through the wln.domain 561 property. Supported domains are US (FCC), EU, JAPAN, and "OTHER", which mimics US channel set.

Selected domain restricts available channels for wln.activescan 556 and wln.associate 557 methods. These are methods that cause the GA1000 to transmit data. Wln.scan 567 is not restricted by the selected domain in any way, because it only listens for incoming Wi-Fi signals, without actually transmitting anything.

One side effect of the above is that you may be able to discover the network with wln.scan, yet unable to associate with it because this network is operating on a channel which is disallowed in a currently selected domain!

Allocating Buffer Memory

The wln. object requires a single buffer. This buffer is used to form outgoing packets and is necessary for correct operation. You never have to deal with this buffer directly -- it is handled internally by the wln. object itself.

Buffer memory is allocated in pages. A page is 256 bytes of memory. Allocating memory for a buffer is a two-step process: First you have to request for a specific allocation (a number of pages) and then you have to perform the actual allocation. Request the size you need in pages using the wln.buffrq 559 method.

The allocation method (sys.buffalloc 530) applies to all buffers previously specified, in one fell swoop:

```
Dim x As Byte
x = wln.buffrq(5) ' request 5 pages for the wln buffer. X will then contain
how many can actually be allocated
' ....  Allocation requests for buffers of other objects...
sys.buffalloc 'perform actual memory allocation, as per previous requests
```

You may not always get the full amount of memory you have requested. Memory is not an infinite resource, and if you have already requested (and received) allocations for 95% of the memory for your platform, your next request will get up to 5% of memory, even if you requested for 10%.

Current wln. buffer size in bytes can always be checked with the wln.buffsize 559 read-only property.

Note that wln buffer size can't be changed when the Wi-Fi hardware is already booted 549 (wln.enabled 562 = 1- YES).

How many pages should the wln buffer get?

The size of the wln buffer directly dictates the maximum size of network packets that the wln object will be able to send (this buffer has nothing to do with incoming packets). Up to 100 bytes of the buffer space are required for various packet headers, and the rest is available to packet payload. For example, if you have allocated 2 pages for the buffer, then the buffer size is 512 bytes. Hence, maximum

payload size cannot exceed 412 bytes. That is, every TCP, UDP, etc. packet sent, including its protocol headers, will not exceed 412 bytes.

For TCP communications, the size of individual packets is not that critical. The beauty of TCP is that it can work with practically any buffer space available. in theory, the bigger the buffer, the better the TCP throughput is. In reality, you will stop feeling any improvement in TCP performance once your wln. buffer size exceeds ~3 pages. UDP is another matter entirely. If you want to be able to send UDP datagrams of a certain size, then you must make sure that you have created an adequate wln. buffer.

In any case, TiOS does not send out TCP or UDP packets with payloads exceeding 1024 bytes (4 pages). This is the internal limitation of TiOS iteself. Add to this TCP/UDP and Wi-Fi headers, and you are still within 5 pages. Therefore, there is no point in setting the wln. buffer to more than 5 pages.

Setting MAC Address (Optional)

Every network interface needs its own MAC, and the Wi-Fi port is no exception. The wln.mac 563 property exists for this purpose. Your Wi-Fi module already carries a MAC address onboard -- it is preset during manufacturing, so you don't actually have to take care of the MAC. Leave the wln.mac at its default pre-boot value of "0.0.0.0.0.0", boot up 549 the Wi-Fi interface (wln.boot 558), and the wln.mac will be updated with the pre-assigned address that is stored inside the Wi-Fi module.

You can use another MAC if you want, too. Set the desired MAC address *before* booting up the Wi-Fi, and this MAC will be used instead of the pre-assigned one. That is, if wln.mac is set to anything but "0.0.0.0.0.0" and then wln.boot is called, then "your" MAC will be used instead.

```
'set the mac address
wln.mac="0.1.2.3.100.200" 'override pre-assigned MAC with another address
...
romfile.open("ga1000fw.bin")
wln.boot(romfile.offset) 'the hardware will start using your MAC
```

The pre-assigned MAC inside the Wi-Fi module will not be altered. It is always there and can be called up by leaving the wln.mac at "all zeroes", then booting the Wi-Fi hardware.

A bit of info on MACs

The MAC address can be either "globally unique" or "locally administered". There is also a provision for "unicast" and "multicast" addressing. You can find more information on this here: http://en.wikipedia.org/wiki/Mac_address.

Your organization can purchase a block of globally unique addresses, or choose to assign random locally administered addresses. In the latter case, set the most significant byte of the address to 2, and choose random values for the remaining 5 bytes (random 222 function will help). Note that each device you are using should have a unique MAC address. It is a good idea to generate the MAC once, and store it in the EEPROM memory (see the stor. 522 object). The MAC can then be retrieved on each boot and written into the wln.mac 563 property.

Note that the MAC address of the Wi-Fi interface can't be set when the hardware is already booted (wln.enabled 562 = 1- YES).

Booting Up the Hardware

Booting up the GA1000 hardware is done through the wln.boot 558 method. The GA1000 does not have a ROM or flash memory and its internal processor executes the firmware from RAM. Before the GA1000 module can start working, you need to upload this firmware into it, and this is what wln.boot really does.

The firmware file is called "ga1000fw.bin" (the file can be downloaded from Tibbo website). The file must be added 132 to your Tibbo Basic project as a binary resource file 20.

Access to resource files is through the romfile. 370 object. First, you open the "ga1000fw.bin" file with the romfile.open 374 method, then pass the pointer to this file (value of the romfile.offset 373 R/O property) to the wln.boot method:

```
'boot it up
romfile.open("ga1000fw.bin")
If wln.boot(romfile.offset) Then
   'something is wrong, react to this
   ...
End If
```

The boot takes 1-3 seconds to complete. The method will return 0- OK if the boot was completed successfully. At that moment, wln.enabled 562 will become 1- YES.

The boot will fail (return 1- NG) if:

- The Wi-Fi hardware is not powered, not properly reset 546, mapped incorrectly 545, or malfunctions.
- The offset to the firmware file is incorrect or the file is not included in your project.
- The Wi-Fi hardware is already booted and operational.

Setting IP, Gateway, and Netmask

The Wi-Fi is a separate network interface and so it has its own IP address, which is set using the wln.ip 563 property. This address is different from the IP address of the Ethernet interface (see net.ip 360). We noticed that many people find it "unusual" that Tibbo hardware device would turn out to have two IP addresses. In fact, this is completely normal. On the PC, every network interface has the IP of its own as well.

Technically speaking, IP address configuration can be done at any time. This topic has been placed into this section to remind you that the IP of the wln. object has to be set, if not right after the boot 549, then at some later point. If your application uses a static IP, then setting it in the boot section of your code is a good idea. If the application obtains the IP address through DHCP, then the IP can only be set after communicating with the DHCP server, and this will only be possible after successful association 554. You may even need to set the IP address repeatedly if your product switches between different networks (access points).

There are also wln.gatewayip 562 and wln.netmask 564 properties that may need to

be set along with the IP address. This is optional and is only required if your device will have to establish outgoing connections to the network hosts outside of your LAN.

Note that IP, gateway IP, and netmask of the Wi-Fi interface can't be set when there is at least one open socket in your system (sock.statesimple 505 <> 0- PL_SSTS_CLOSED) that operates on the Wi-Fi interface (sock.currentinterface 478 = 2- PL_SOCK_INTERFACE_WLN).

A very useful DHCP library 618 can handle IP, gateway IP, and netmask configuration both for the Ethernet (net. 358) and Wi-Fi (wln.) interfaces of your device. Use it and save yourself a ton of work!

Setting TX Power (Optional)

The output power of the Wi-Fi hardware can be adjusted in 12 steps. The wln.settxpower 570 method is provided for that purpose. The power value roughly corresponds to dB. The lowest output power is set with wln.settxpower(4) and the highest power is set with wln.settxpower(15). Lower power reduces the current consumption of the Wi-Fi module, but not by much. We recommend that you do not touch this.

Note that setting TX power is an "immediate" wln task 539 and there is a certain correct way of handling tasks. "Immediate" means you don't have to wait for the task to complete -- it is finished as soon as wln.settxpower is done executing.

Scanning for Wi-Fi Networks

Scanning allows you to discover all networks in your device's range and also learn about their operating parameters, such as the name, RF channel, signal strength, etc. Two methods -- wln.scan 567 and wln.activescan 556 -- are provided for this purpose. Both methods are wln tasks 539 and there is a certain correct way of handling tasks.

Wln.scan performs a passive detection of networks. During the passive detection, the hardware listens for "beacons" transmitted by wireless networks and extracts network information from these beacon packets. This method can't collect the network information of "hidden" wireless networks that do not broadcast their SSIDs (names).

Passive scanning is performed on all 14 frequency channels, regardless of the value of wln.domain 561. This is not in violation of any regulations because the passive scanning does not involve transmitting any data out of the device (the transmitter stays silent).

Wln.activescan actively "probes" the environment around the device by sending -- you guessed it -- "probe" packets. This method can find all the wireless networks that wln.scan would find, plus it can discover a hidden network, but only if you knew this network's name in advance.

Active scanning (sending of "probe" packets) is only performed on "allowed" channels. The list of allowed channels depends on the selected domain (wln.domain 561).

Scanning can be performed at any time, even when the Wi-Fi interface is in the associated state (wln.associationstate= 558 1- PL_WLN_ASSOCIATED) or running its own ad-hoc network (wln.associationstate= 2- PL_WLN_OWN_NETWORK). Keep in mind, however, that scanning temporarily disrupts communications between the device and the access point. This is because scanning involves checking for available access points on all (allowed) frequencies. Obviously, the radio can't keep

communicating with the "current" access point while jumping from channel to channel.

Both wln.scan and wln.activescan accept a single argument of string type. Presence or absence of this argument defines the "operating mode" of these methods. You can either discover all wireless networks 551 in range, or try to collect information about the specific network 551.

Also read about what happens when several access points have the same name 552.

Discovering All Wireless Networks

When called with an empty string argument, wln.scan 567 and wln.activescan 556 will attempt to find all wireless networks in range. After the task completes, the wln.scanresultssid 569 R/O property will contain a comma-separated list of network names.

```
'scan for available network
Dim s As String
...
wln.scan("") 'you could use wln.activescan('') instead
While wln.task<>PL_WLN_TASK_IDLE
Wend
s=wln.scanresultssid 'the list of networks will be copied into s
```

After the execution of the above, the s string may contain something like this: "TIBBO,c1100_1,.....,WNET2". Notice "....." -- these are five bytes with ASCII code 0. They represent a hidden network (i.e. the network that does not broadcast its SSID). Naturally, the name of this network is not revealed to your device. This will be the case regardless of whether you used wln.scan or wln.activescan.

Collecting Data About Specific Network

When called with the argument set to the name of a particular network, the wln.scan 567 and wln.activescan 556 methods will attempt to find this wireless network and obtain its operational parameters:

```
'scan for a specific network
wln.activescan("Tibbo1") 'you could use wln.scan(''), unless you are
expecting to be dealing with a hidden network
While wln.task<>PL_WLN_TASK_IDLE
Wend
If wln.scanresultssid<>"" Then
     'network found!
     ...
End If
```

After the scan...

- Wln.scanresultssid 569 R/O property will contain the name of the specified network, or nothing if the network wasn't found. If the network was found, the following five properties will also be updated:
- Wln.scanresultbssid 567 R/O property will contain the BSSID ("MAC address") of the specified network.
- Wln.scanresultbssmode 568 R/O property will contain the BSS mode of the network (infrastructure or ad-hoc).
- Wln.scanresultchannel 568 R/O property will return the number of the RF channel on which the network operates.
- Wln.scanresultrssi 569 R/O property will contain the strength of the RF signal received from the specified network.
- Wln.scanresultwpainfo 569 R/O property will contain the data necessary for WPA/WPA2 protocols 553. This is not a human-readable data. Better let our WLN library 703 interpret it.

Both wln.scan and wln.activescan will collect exactly the same data about the specified network. The only difference is that wln.scan won't be able to do the job if this wireless network does not broadcast its SSID. Wln.activescan, on the contrary, will get this done. So, if you need to work with a hidden network, then you need to know this network's name, and you need to use wln.activescan to obtain this network's data.

Multiple Access Points With the Same Name

Many installations have several access points with the same SSID (name). These may be viewed as a single wireless network consisting of multiple access points, or several networks that use the same SSID. What happens when you do wln.scan 567 or wln.activescan 556 in this environment?

Let's assume you have five access points in range: "CISCO1", "Net10", and three access points named "tibbo_net".

If you are discovering all wireless networks 551 in range, i.e. do wln.scan("") or wln.activescan(""), then after the scanning task is complete, wln.scanresultssid 569 will contain several identical entries, like this: "CISCO1,tibbo_net,tibbo_net,Net10, tibbo_net". You can't really distinguish one "tibbo_net" from another!

If you are collecting data about specific network 551, i.e. do wln.scan("tibbo_net") or wln.activescan("tibbo_net"), then wln.scanresultssid will end up containing a single entry: "tibbo_net". Question is, which one of the three "tibbo_net" access points does this entry represent? The answer: it is the one with the strongest signal (the highest wln.scanresultrssi 569)! All other "wln.scanresult..." R/O properties will then contain values pertaining to this access point. Thus, when given a choice, the wln. object automatically selects the best access point to work with.

Setting Wi-Fi Security

The wln. object supports two types of Wi-Fi security:

- WEP64 and WEP128 553 -- a weak and old form of Wi-Fi protection.
- WPA-PSK and WPA2-PSK 553 -- two modern forms of Wi-Fi security (use WPA2 whenever possible). These security modes require the WLN library 703 (or a lot of coding effort) to work correctly.

WEP and WPA security must be set before associating 554 with the selected network or creating own ad-hoc network 555.

Setting WEP Mode and Key

The wln.setwep 570 method allows you to specify the WEP mode and key. Note that wln.setwep is an "immediate" wln task 539 and there is a certain correct way of handling tasks. "Immediate" means you don't have to wait for the task to complete -- it is finished as soon as wln.setwep is done executing.

The mode can be either DISABLED, WEP64, or WEP128. WEP key is entered as a HEX string, not ASCII string. Each character in a string represents one HEX digit: 0..9 or A..F (a..f). The key has a fixed length: 10 HEX digits for WEP-64 or 26 HEX digits for WEP-128. If your key is too short it will be padded with zeroes. If the key is too long it will be truncated.

Here is the code example that sets the Wi-Fi to WEP128 mode:

```
'set WEP128
wln.setwep("11111111111111111111111111",PL_WLN_WEP_MODE_128) 'we love to
choose difficult keys
```

Wi-Fi devices routinely define four WEP keys, but Tibbo hardware only uses a single key (key 1).

Note that the WEP mode and key can't be changed while the Wi-Fi interface is in the associated state (wln.associationstate= 558 1- PL_WLN_ASSOCIATED) or is running its own network (wln.associationstate= 2- PL_WLN_OWN_NETWORK).

You need to call wln.setwep *every time* before associating 554 with the access point or starting 555 own network. If you are switching from an access point using WEP security to another access point with no security or WPA security 553, you still need to execute wln.setwep("", PL_WLN_WEP_MODE_DISABLED)!

Setting WPA Mode and Key

The wln.setwpa 571 method allows you to specify the WEP mode and key. Looks easy, similar to WEP 553, but the reality is that doing WPA is a hard hard job! It is, in fact, so hard and involves so many steps that we are not even going to attempt to explain how this is done. Use the WLN library 703 whenever you have to deal with WPA/WPA2. This is the way of doing this, really!

Tibbo devices only support "personal" WPA modes WPA-PSK and WPA2-PSK.

If you are switching from an access point using WPA security to another access point with no security or WEP security 553, you still need to execute wln.setwpa(PL_WLN_WPA_DISABLED,0,"",0)!

Associating With Selected Network

Association is a process by which your Wi-Fi device establishes a link with an access point or another wireless station running an ad-hoc network.

The association process is initiated using the wln.associate 557 method. Association is a required step before you will be able to send and receive the data over the Wi-Fi. Note that wln.associate is a wln task 539 and there is a certain correct way of handling tasks.

Wln.associate will be rejected (return 1- REJECTED) if:

- Another task is currently in progress.
- GA1000 add-on module is not online (wln.enabled 562 = 0- NO).
- The Wi-Fi interface is already in the non-idle state (wln.associationstate 558 <>0- PL_WLN_NOT_ASSOCIATED). That is, you can't try to associate when you are already associated or running own ad-hoc network 555.

The method will return 0- ACCEPTED if the task is accepted for processing.

Prior to associating, you need to set the WEP 553 or WPA 553 security (if required). You also need to know several key parameters about the network you are associating with:

- SSID (the name) of this network;
- BSSID ("MAC") of this network;
- Channel on which this network operates;
- BSS mode of this network (whether the network is infrastructure or ad-hoc).

Normally, the SSID is known, but BSSID, channel, and BSS mode require some digging. The easiest way to sniff out correct values of these parameters is through scanning 551. Wln.scan 567 or wln.activescan 556 will fill out wln.scanresultbssid 567, wln.scanresultchannel 568, and wln.scanresultbssmode 568 R/O properties. You only need to put them to good use, as shown in the example below.

Before you look at the example, consider this: you may be able to discover the network with wln.scan, yet unable to associate with it because this network is operating on a channel which is disallowed in a currently selected domain 547.

```
'connect to the access point named TIBBO.
wln.scan("TIBBO") 'scanning for a specific network will give us necessary
parameters (you can also use wln.activescan)
While wln.task<>PL_WLN_TASK_IDLE
Wend
If wln.scanresultssid<>"" Then
   'wireless network not found
End If

'now can associate: 'wln.scanresult...' properties contain necessary data
after the scanning
wln.associate(wln.scanresultbssid, wln.scanresultssid,
wln.scanresultchannel, wln.scanresultbssmode)
While wln.task<>PL_WLN_TASK_IDLE
Wend
If wln.associationstate=PL_WLN_ASSOCIATED Then
   'successful association!
   ...
End If
```

After the association task is completed you have to check the association result. Mere task completion does not indicate success! The wln.associationstate 558 will provide the indication.

If you are running with no security or WEP security 553, you can start communicating 555 over the Wi-Fi interface as soon as you are done associating. If it is WPA you are dealing with, completing wln.associate is really just a first step. There is a lengthy and laborious message exchange to follow. It is best to delegate this to our WLN library 703!

When the wln. object is in the associated state, wln.rssi 566 read-only property is constantly updated with the strength of the signal coming from the access point. Do not confuse this with the wln.scanresultrssi 569 property which returns the signal strength of a particular network obtained during scanning 550.

Creating Own Ad-hoc Network

Rather than associating 554 with somebody else's network, the Wi-Fi interface can create an ad-hoc network of its own and have other devices associate with it. This is done by using wln.networkstart 564. Note that wln.networkstart is a wln task 539 and there is a certain correct way of handling tasks.

Wln.networkstart will be rejected (return 1- REJECTED) if:

- Another task is currently in progress.
- GA1000 add-on module is not online (wln.enabled 562 = 0- NO).
- The Wi-Fi interface is already in the non-idle state (wln.associationstate 558 <>0- PL_WLN_NOT_ASSOCIATED). That is, you can't start a network when you are already associated or running a network.

The method will return 0- ACCEPTED if the task is accepted for processing.

To start a network, you only need to make up your mind regarding its name and operating channel:

```
'take control and run your own network
wln.networkstart("VOICEOFREBELS",6)
While wln.task<>PL_WLN_TASK_IDLE
Wend
```

Communicating via Wln Interface

Actual data exchange over the Wi-Fi interface falls outside the responsibilities of the wln object. This is the task of the sock. 421 object. This object has a set of properties that define whether a particular socket will be listening on the Wi-Fi interface (sock.allowedinterfaces 474 property), establish an outgoing connection through the Wi-Fi interface (sock.targetinterface 506 property). See also: sock.currentinterface 478 read-only property.

Disassociating From the Network

To disassociate from the network, use the wln.disassociate 561 method. Note that wln.disassociate is a wln task 539 and there is a certain correct way of handling tasks.

```
'disassociate now
wln.disassociate
While wln.task<>PL_WLN_TASK_IDLE
Wend
```

Disassociation can be forced by the access point. Disassociation will also happen if the access point goes offline or out of range, or if the Wi-Fi hardware is powered down, reset, disconnected, or malfunctions. This will be detected 556 by the wln object.

Terminating Own Ad-hoc Network

Use wln.networkstop 564 to terminate your ad-hoc network. We are talking about the network your hardware has created -- you can't terminate an ad-hoc network of someone else. Note that wln.networkstop is a wln task 539 and there is a certain correct way of handling tasks.

Detecting Disassociation or Offline State

The wln. object automatically detects disassociation from the wireless network and powering-off of or reset of the GA1000. On_wln_event 565 event is fired up if either condition is detected. in response to this event, your application can re-initialize the Wi-Fi hardware and/or re-associate with the wireless network.

Properties, Methods, Events

Properties, methods, and events of the wln object.

.Activescan Method

Function:	Causes the Wi-Fi interface to commence either the active detection of available wireless networks or obtainment of an additional information about a particular network specified by its SSID (name).
Syntax:	**wln.activescan(byref** ssid **as string) as accepted_rejected**
Returns:	One of accepted_rejected constants: 0- ACCEPTED. 1- REJECTED.
See Also:	Scanning for Wi-Fi Networks 550, Wln Tasks 539, wln.scan 567

Part	Description

ssid	Network name. Leave empty to detect all available networks -- after the scan, the comma-delimited list of networks will be in wln.scanresultssid 569. Alternatively, specify the network name. If the network is detected, its parameters will be available through wln.scanresultssid 569, wln.scanresultbssid 567, wln.scanresultbssmode 568, wln.scanresultchannel 568, wln.scanresultrssi 569, wln.scanresultwpainfo 569.

Details

Active scanning process is a task and there is a certain correct way 539 of handling tasks.

Scanning while the Wi-Fi interface is in the associated state (wln.associationstate= 558 1- PL_WLN_ASSOCIATED) or running its own ad-hoc network (wln.associationstate= 2- PL_WLN_OWN_NETWORK) will temporarily disrupt communications between the device and the "current" access point.

Active detection of networks is performed only on allowed channels, as determined by the wln.domain 561 property. This is because an active "probe" signal is transmitted during the active scanning.

Active scanning can detect a networks that doesn't broadcast its SSID, but you still need to know this network's name in advance.

.Associate Method

Function:	Causes the Wi-Fi interface to attempt association with the specified wireless network.
Syntax:	**wln.associate(byref** bssid **as string, byref** ssid **as string,** channel **as byte,** bssmode **as pl_wln_bss_modes) as accepted_rejected**
Returns:	One of accepted_rejected constants: 0- ACCEPTED. 1- REJECTED.
See Also:	Associating With Selected Network 554, Setting Wi-Fi Security 552, Wln Tasks 539 wln.disassociate 561

Part	Description
bssid	The BSSID ("MAC address") of the network with which to associate.
ssid	The name of the target network with which to associate.
channel	Channel on which the target network is operating.
bssmode	Network mode: 0- PL_WLN_BSS_MODE_INFRASTRUCTURE: This is an infrastructure network (access point). 1- PL_WLN_BSS_MODE_ADHOC: This is an ad-hoc (device-to-device) network.

Details

The association process is a task and there is a certain correct way 539 of handling tasks. Task completion does not imply success -- association result has to be verified by reading the state of the wln.associationstate 558 read-only property after the task is completed.

.Associationstate R/O Property

Function: Indicates whether the Wi-Fi interface is idle, associated with another network, or running its own ad-hoc network.

Type: Enum (pl_wln_association_states, byte)

Value Range: 0- PL_WLN_NOT_ASSOCIATED **(default)**: The Wi-Fi interface is idle.

1- PL_WLN_ASSOCIATED: The Wi-Fi interface is associated with a wireless network.

2- PL_WLN_OWN_NETWORK: The Wi-Fi interface is running its own ad-hoc network.

See Also: Associating With Selected Network 554, Disassociating From the Network 556, Creating Own Ad-hoc Network 555, Terminating Own Ad-hoc Network 556, Detecting Disassociation or Offline State 556

Details

After the successful association, which is initiated through the wln.associate 557 method, the value of this property changes to 1- PL_WLN_ASSOCIATED. The value is reset back to 0- PL_WLN_NOT_ASSOCIATED if disassociation occurs (on_wln_event 565 will be generated too).

Disassociation can be induced through the wln.disassociate 561 method or forced by the access point. Disassociation will also happen if the access point goes offline or out of range, or if the GA1000 is powered down, reset, disconnected, or malfunctions.

After the Wi-Fi interface succeeds in creating its own ad-hoc network (see wln.networkstart 564), the value of this property becomes 2- PL_WLN_OWN_NETWORK. The value is reset back to 0- PL_WLN_NOT_ASSOCIATED when the ad-hoc network is terminated with wln.networkstop 564.

.Boot Method

Function: Boots up the Wi-Fi interface, which involves sending to the GA1000 hardware a firmware file for its embedded processor.

Syntax: **wln.boot(**offset **as dword) as ok_ng**

Returns: One of ok_ng constants:

0- OK: completed successfully.

1- NG: boot failed.

See Also: Booting Up the Hardware 549

Part	Description
offset	Offset of the "ga1000fw.bin" file within the compiled binary of your project. The offset is obtained using the romfile.offset 373 read-only property.

Details

For wln.boot to succeed, *ga1000fw.bin* file must be present in your project as a binary resource file.

.Buffrq Method

Function: Pre-requests "numpages" number of buffer pages (1 page= 256 bytes) for the TX buffer of the wln. object.

Syntax: **wln.buffrq(**numpages **as byte) as byte**

Returns: Actual number of pages that can be allocated (byte).

See Also: Allocating Buffer Memory 547
wln.buffsize 559

Part	Description
numpages	Requested numbers of buffer pages to allocate (recommended value is **5**).

Details

This method will not work if the GA1000 is already operational (wln.enabled 562 = 1-YES).

.Buffsize R/O Property

Function: Returns the current capacity (in bytes) of the wln. object's TX buffer.

Type: Word

Value Range: 0-65535, **default**= 0 (0 bytes).

See Also: Allocating Buffer Memory 547
wln.buffrq 559

Details

.Clkmap Property

Function: Sets/returns the number of the I/O line to serve as the clock (CLK) line of the GA1000's SPI interface.

Type: Enum (pl_io_num, byte)

Value Range: Platform-specific. See the list of pl_io_num constants in the platform specifications. **Default**= PL_IO_NULL (NULL line).

See Also: Configuring Interface Lines 545

wln.csmap 560, wln.dimap 560, wln.domap 562

Details

The selection cannot be changed once the Wi-Fi hardware is already operational (wln.enabled 562 = 1- YES).

This property has no effect on the EM500W 138 platform.

.Csmap Property

Function: Sets/returns the number of the I/O line to serve as the chip select (CS) line of the GA1000's SPI interface.

Type: Enum (pl_io_num, byte)

Value Range: Platform-specific. See the list of pl_io_num constants in the platform specifications. **Default**= PL_IO_NULL (NULL line).

See Also: Configuring Interface Lines 545

wln.clkmap 560, wln.dimap 560, wln.domap 562

Details

The selection cannot be changed once the Wi-Fi hardware is already operational (wln.enabled 562 = 1- YES).

This property has no effect on the EM500W 138 platform.

.Dimap Property

Function: Sets/returns the number of the I/O line to serve as the data in (DI) line of the GA1000's SPI interface.

Type: Enum (pl_io_num, byte)

Value Range: Platform-specific. See the list of pl_io_num constants in the platform specifications. **Default**= PL_IO_NULL (NULL line).

See Also: Configuring Interface Lines 545

wln.clkmap 560, wln.csmap 560, wln.domap 562

Details

The selection cannot be changed once the Wi-Fi hardware is already operational (wln.enabled 562 = 1- YES).

This DI line must be connected to the DO pin of the GA1000.

The property has no effect on the EM500W 138 platform.

.Disassociate Method

Function:	Causes the Wi-Fi interface to commence disassociation from the wireless network.
Syntax:	**wln.disassociate() as accepted_rejected**
Returns:	One of accepted_rejected constants: 0- ACCEPTED. 1- REJECTED.
See Also:	Disassociating From the Network 556, Wln Tasks 539 wln.associate 557, wln.associationstate 558

Details

The disassociation process is a task and there is a certain correct way 539 of handling tasks.

.Domain Property

Function:	Selects the domain (area of the world) in which this device is operating. This defines the list of channels on which the Wi-Fi interface will perform active scanning or associate with wireless networks.
Type:	Enum (pl_wln_domains, byte)
Value Range:	0- PL_WLN_DOMAIN_FCC **(default)**: FCC domain (US). Allowed channels: 1-11. 1- PL_WLN_DOMAIN_EU: European Union. Allowed channels: 1-13. 2- PL_WLN_DOMAIN_JAPAN: Japan. Allowed channels: 1-14. 3- PL_WLN_DOMAIN_OTHER: All other countries. Allowed channels: 1-11.
See Also:	Selecting Domain 547

Details

This property can't be changed while the Wi-Fi hardware is operational (wln.enabled 562 = 1- YES). Note that domain selection only affects active scanning (wln.activescan 556) and association (wln.associate 557). Passive scanning (wln.scan) 567 is not restricted by the selected domain in any way, because the GA1000 will only listen for incoming Wi-Fi signals, without actually transmitting

anything. Selected domain also doesn't limit the ability of the Wi-Fi interface to start its own ad-hoc network (wln.networkstart 564) on whatever channel you specify.

.Domap Property

Function: Sets/returns the number of the I/O line to serve as the data out (DO) line of the GA1000's SPI interface.

Type: Enum (pl_io_num, byte)

Value Range: Platform-specific. See the list of pl_io_num constants in the platform specifications. **Default**= PL_IO_NULL (NULL line).

See Also: Configuring Interface Lines 545

wln.clkmap 560, wln.csmap 560, wln.dimap 560

Details

The selection cannot be changed once the Wi-Fi hardware is already operational (wln.enabled 562= 1- YES).

The DO line must be connected to the DI pin of the GA1000.

This property has no effect on the EM500W 138 platform.

.Enabled R/O Property

Function: Indicates whether the Wi-Fi interface is operational.

Type: Enum (no_yes, byte)

Value Range: 0- NO **(default)**: The Wi-Fi interface is not operational.

1- YES: The Wi-Fi interface is operational.

See Also: Bringing Up the Hardware 549, Detecting Disassociation or Offline State 556

Details

The Wi-Fi hardware becomes operational after a successful boot using the wln.boot 558 method, at which time wln.enabled is set to 1- YES.

The Wi-Fi interface is disabled and the wln.enabled is reset to 0- NO if the Wi-Fi hardware is disconnected, powered down, malfunctioned, or was intentionally reset 546. When this happens, the on_wln_event 565 event is generated with its wln_event argument set to 0- PL_WLN_EVENT_DISABLED.

.Gatewayip Property

Function: Sets/returns the IP address of the default gateway for the Wi-Fi interface of your device.

Type: Dot-decimal string

Value Range: Any IP address, such as "192.168.1.1". **Default**= "0.0.0.0".

See Also: Setting IP, Gateway, and Netmask 549

wln.ip 563, wln.netmask 564

Details

This property can only be written to when no socket is engaged in communicating through the Wi-Fi interface, i.e. there is no socket for which sock.statesimple 505 <> 0- PL_SSTS_CLOSED and sock.currentinterface 478 = 2- PL_INTERFACE_WLN.

.Ip Property

Function: Sets/returns the IP address of the Wi-Fi interface of your device.

Type: Dot-decimal string

Value Range: Any IP address, such as "192.168.100.40". **Default**= "1.0.0.1".

See Also: Setting IP, Gateway, and Netmask 549

wln.gatewayip 562, wln.netmask 564

Details

This property can only be written to when no socket is engaged in communications through the Wi-Fi interface, i.e. there is no socket for which sock.statesimple 505 <> 0- PL_SSTS_CLOSED and sock.currentinterface 478 = 2- PL_INTERFACE_WLN.

.Mac Property

Function: Sets/returns the MAC address of the Wi-Fi interface.

Type: Dot-decimal string

Value Range: Any valid MAC address, i.e. "0.1.2.3.4.5". **Default**= "0.0.0.0.0.0".

See Also: Setting MAC Address 548

Details

This property can only be written to while the Wi-Fi hardware is not operational (wln.enabled 562 = 0- NO).

The GA1000 add-on board already has a proper MAC address internally. To use this MAC, leave the wln.mac at "0.0.0.0.0.0".

.Netmask Property

Function: Sets/returns the netmask of the Wi-Fi interface of your device.

Type: Dot-decimal string

Value Range: Any valid netmask, such as "255.255.255.0". **Default**= "0.0.0.0".

See Also: Setting IP, Gateway, and Netmask 549

wln.ip 563, wln.gatewayip 562

Details

This property can only be written to when no socket is engaged in communications through the Wi-Fi interface, i.e. there is no socket for which sock.statesimple 505 <> 0- PL_SSTS_CLOSED and sock.currentinterface 478 = 2- PL_INTERFACE_WLN.

.Networkstart Method

Function: Causes the Wi-Fi interface to commence creating its own ad-hoc network.

Syntax: **wln.networkstart(byref** ssid **as string,** channel **as byte) as accepted_rejected**

Returns: One of accepted_rejected constants:

0- ACCEPTED.

1- REJECTED.

See Also: Creating Own Ad-hoc network 555, Setting Wi-Fi Security 552, Wln Tasks 539

wln.networkstop 564

Part	Description
ssid	The name of the ad-hoc network to create.
channel	Channel on which the new ad-hoc network will operate.

Details

The network creation process is a task and there is a certain correct way 539 of handling tasks. Task completion does not imply success -- the result has to be verified by reading the state of the wln.associationstate 558 read-only property after the task is completed.

.Networkstop Method

Function: Causes the Wi-Fi interface to commence the termination of its own ad-hoc network.

Syntax: **wln.networkstop() as accepted_rejected**

Returns:	One of accepted_rejected constants: 0- ACCEPTED. 1- REJECTED.
See Also:	Terminating Own Ad-hoc Network 556, Wln Tasks 539 wln.networkstart 564

Details

The network termination process is a task and there is a certain correct way 539 of handling tasks.

On_wln_event Event

Function:	Generated when the wln object detects disassociation from the wireless network or the Wi-Fi hardware is disconnected, powered-down, reset, or is malfunctioning.
Declaration:	**on_wln_event(**wln_event **as pl_wln_events)**
See Also:	Detecting Disassociation or Offline State 556 on_wln_task_complete 565

Part	Description
wln_event	Registered event: 0- PL_WLN_EVENT_DISABLED: Wi-Fi hardware has been disconnected, powered down, or is malfunctioning. 1- PL_WLN_EVENT_DISASSOCIATED: Wi-Fi interface has been disassociated from the wireless network.

Details

Multiple on_wln_event events may be waiting in the event queue. For this reason the doevents statement will be skipped (not executed) if encountered within the event handler for this event or the body of any procedure in the related call chain.

On_wln_task_complete Event

Function:	Generated when the Wi-Fi interface completes executing a given task.
Declaration:	**on_wln_task_complete(**completed_task **as pl_wln_tasks)**
See Also:	Wln Tasks 539 on_wln_event 565

Part	Description
completed_task	The task just completed: 1- PL_WLN_TASK_SCAN: Scan task completed (this task is initiated by the wln.scan 567 method). 2- PL_WLN_TASK_ASSOCIATE: Association task completed (this task is initiated by the wln.associate 557 method). 3- PL_WLN_TASK_SETTXPOWER: TX power adjustment task completed (this task is initiated by the wln.settxpower 570 method). This is an "immediate" task. 4- PL_WLN_TASK_SETWEP: WEP mode and keys setup task completed (this task is initiated by the wln.setwep 570 method). This is an "immediate" task. 5- PL_WLN_TASK_DISASSOCIATE: Disassociation task completed (this task is initiated by the wln.disassociate 561 method). 6- PL_WLN_TASK_NETWORK_START: Ad-hoc network creation completed (this task is initiated by the wln.networkstart 564 method). 7- PL_WLN_TASK_NETWORK_STOP: Ad-hoc network termination completed (this task is initiated by the wln.networkstop 564 method). 8- PL_WLN_TASK_SETWEP: WPA mode and keys setup task completed (this task is initiated by the wln.setwpa 571 method). This is an "immediate" task.

Details

The wln.task 572 read-only property will change to 0- PL_WLN_TASK_IDLE along with this event generation.

Multiple on_wln_task_complete events may be waiting in the event queue. For this reason the doevents statement will be skipped (not executed) if encountered within the event handler for this event or the body of any procedure in the related call chain.

.Rssi R/O Property

Function: Indicates the strength of the signal being received from the wireless network that the Wi-Fi interface is currently associated with, or wireless peer in case of the ad-hoc network.

Type: Byte

Value Range: 0-255, **default= 0.**

See Also: Associating With Selected Network 554

wln.scanresultrssi 569

Details

The signal strength is expressed in 256 aribitrary levels that do not correspond to any standard measurement unit.

This property is only updated while the Wi-Fi interface is in the non-idle state

(wln.associationstate 558<>0- PL_WLN_NOT_ASSOCIATED).

.Scan Method

Function: Causes the Wi-Fi interface to commence either the passive detection of available wireless networks or obtainment of an additional information about a particular network specified by its SSID (name).

Syntax: **wln.scan(byref** ssid **as string) as accepted_rejected**

Returns: One of accepted_rejected constants:

0- ACCEPTED.

1- REJECTED.

See Also: Scanning for Wi-Fi Networks 550, Wln Tasks 539, wln.activescan 556

Part	Description
ssid	Network name. Leave empty to detect all available networks -- after the scan, the comma-delimited list of networks will be in wln.scanresultssid 569. Alternatively, specify the network name. If the network is detected, its parameters will be available through wln.scanresultssid 569, wln.scanresultbssid 567, wln.scanresultbssmode 568, wln.scanresultchannel 568, wln.scanresultrssi 569, wln.scanresultwpainfo 569.

Details

Passive scanning process is a task and there is a certain correct way 539 of handling tasks.

Scanning while the Wi-Fi interface is in the associated state (wln.associationstate= 558 1- PL_WLN_ASSOCIATED) or running its own ad-hoc network (wln.associationstate= 2- PL_WLN_OWN_NETWORK) will temporarily disrupt communications between the device and the "current" access point.

Passive detection of networks is performed on all channels, regardless of the value of the wln.domain 561 property. This is because no radio signal is transmitted during the passive scanning.

Passive scanning won't work with networks that have their SSID (name) hidden. To work with "hidden networks", use wln.activescan 556 method instead.

.Scanresultbssid R/O Property

Function: After a successful scan for a particular network (wln.scan 567 with the ssid specified) this property will contain the BSSID ("MAC address") of this network.

Type: Dot-decimal string

Value Range: Standard 6-byte MAC value

See Also: Scanning for Wi-Fi Networks 550

wln.scanresultbssmode 568, wln.scanresultchannel 568, wln.scanresultrssi 569, wln.scanresultssid 569,

wln.scanresultwpainfo 569

Details

This property will not be updated if the wln.scan 567 method is invoked with its ssid argument left empty ("search for all networks" mode).

.Scanresultbssmode R/O Property

Function: After a successful scan for a particular network (wln.scan 567 with the ssid specified) this property will contain the network mode of this network.

Type: Enum, byte

Value Range: 0- PL_WLN_BSS_MODE_INFRASTRUCTURE: wireless network with an access point.

1- PL_WLN_BSS_MODE_ADHOC: device-to-device network without an access point.

See Also: Scanning for Wi-Fi Networks 550

wln.scanresultbssid 567, wln.scanresultchannel 568, wln.scanresultrssi 569, wln.scanresultssid 569, wln.scanresultwpainfo 569

Details

This property will not be updated if the wln.scan 567 method is invoked with its ssid argument left empty ("search for all networks" mode).

.Scanresultchannel R/O Property

Function: After a successful scan for a particular network (wln.scan 567 with the ssid specified) this property will contain the number of the channel on which this network operates.

Type: Byte

Value Range: 1-14

See Also: Scanning for Wi-Fi Networks 550

wln.scanresultbssid 567, wln.scanresultbssmode 568, wln.scanresultrssi 569, wln.scanresultssid 569, wln.scanresultwpainfo 569

Details

This property will not be updated if the wln.scan 567 method is invoked with its ssid argument left empty ("search for all networks" mode).

.Scanresultrssi R/O Property

Function: After a successful scan for a particular network (wln.scan 567 with the ssid specified) this property will contain the strength of the signal received from this network.

Type: Byte

Value Range: 0-255

See Also: Scanning for Wi-Fi Networks 550

wln.scanresultbssid 567, wln.scanresultbssmode 568, wln.scanresultchannel 568, wln.scanresultssid 569, wln.scanresultwpainfo 569, wln.rssi 566

Details

This property will not be updated if the wln.scan 567 method is invoked with its ssid argument left empty ("search for all networks" mode).

.Scanresultssid R/O Property

Function: After the scan this property will contain a comma-delimited list of discovered networks or the name of a particular network depending on how the scan was performed.

Type: String

Value Range: Up to 95 characters of data

See Also: Scanning for Wi-Fi Networks 550

wln.scanresultbssid 567, wln.scanresultbssmode 568, wln.scanresultchannel 568, wln.scanresultrssi 569, wln.scanresultwpainfo 569

Details

If the wln.scan 567 method was invoked with its name argument left empty, this property will contain the list of all discovered networks. If the name argument specified a particular network and scanning found this network to be present, then this property will contain the name of this network.

.Scanresultwpainfo R/O Property

Function: After a successful scan for a particular network (wln.scan 567 with the ssid specified) this property will contain binary data required for WPA/WPA2 security protocol.

Type: Byte

Value Range: 0-255

See Also: Scanning for Wi-Fi Networks 550

wln.scanresultbssid 567, wln.scanresultbssmode 568,

wln.scanresultchannel[568], wln.scanresultrssi[569], wln.scanresultssid[569]

Details

This property will not be updated if the wln.scan[567] method is invoked with its ssid argument left empty ("search for all networks" mode).

This data is not intended for humans. The property exists to facilitate the operation of the WLN library[703].

.Settxpower Method

Function: Causes the Wi-Fi interface to commence the adjustment of TX power to the specified level.

Syntax: **wln.settxpower(**level **as byte) as accepted_rejected**

Returns: One of accepted_rejected constants:

0- ACCEPTED.

1- REJECTED.

See Also: Setting TX Power[550], Wln Tasks[539]

Details

Part	Description
level	Value between 4 and 15 that roughly corresponds to the transmitter's output power in dB. Attempting to specify the level < 4 results in level = 4; attempting to specify the level > 15 results in level = 15.

Adjusting TX power is an "immediate" task and there is a certain correct way[539] of handling tasks. "Immediate" means you don't have to wait for the task to complete -- it is finished as soon as wln.settxpower is done executing. The on_wln_task_complete[565] event is still generated.

.Setwep Method

Function: Causes the Wi-Fi interface to commence setting new WEP mode and key.

Syntax: **wln.setwep(byref** wepkey **as string,** wepmode **as pl_wln_wep_modes) as accepted_rejected**

Returns: One of accepted_rejected constants:

0- ACCEPTED.

1- REJECTED.

See Also: Setting WEP Mode and Key[553], Wln Tasks[539]

wln.setwpa[571]

Part	Description
wep key	A string containing new WEP key. This is a "HEX strings" -- each character in the string represents one HEX digit. The string must contain 10 HEX digits for WEP64 and 26 HEX digits for WEP128. Excessive digits are ignored. Missing digits are assumed to be 0.
wep mode	WEP mode to set: 0- PL_WLN_WEP_MODE_DISABLED: WEP is to be disabled. 1- PL_WLN_WEP_MODE_64: 64-bit WEP is to be used. 0- PL_WLN_WEP_MODE_128: 128-bit WEP is to be used.

Details

Changing WEP mode and keys is an immediate task and there is a certain correct way 539 of handling tasks. "Immediate" means you don't have to wait for the task to complete -- it is finished as soon as wln.setwep is done executing. The on_wln_task_complete 565 event is still generated.

.Setwpa Method

Function:	Causes the Wi-Fi interface to commence setting new WPA mode and key.
Syntax:	**wln.setwpa(**wpamode **as pl_wln_wpa_modes,** algorithm **as pl_wln_wpa_algorithms, byref** wpakey **as string,** cast **as pl_wln_wpa_unicast_multicast) as accepted_rejected**
Returns:	One of accepted_rejected constants: 0- ACCEPTED. 1- REJECTED.
See Also:	Setting WPA Mode and Key 553, Wln Tasks 539 wln.setwep 570

Part	Description
wep mode	WPA mode to set: 0- PL_WLN_WPA_DISABLED: WPA is to be disabled. 1- PL_WLN_WPA_WPA1_PSK: WPA1-PSK is to be used. 0- PL_WLN_WPA_WPA2_PSK: WPA2-PSK is to be used.
algorithm	Encryption algorithm to be used: 0- PL_WLN_WPA_ALGORITHM_TKIP: TKIP algorithm. 1- PL_WLN_WPA_ALGORITHM_AES: AES algorithm (also referred to as CCMP).
wpa key	A string containing new WPA key. Supposed to be 16 characters long.
cast	Type of key: 0- PL_WLN_WPA_CAST_UNICAST: unicast key.

1- PL_WLN_WPA_CAST_MULTICAST: multicast key.

Details

WPA/WPA2 security is very complex. We recommend using the WLN library 703.

Changing WPA mode and keys is an immediate task and there is a certain correct way 539 of handling tasks. "Immediate" means you don't have to wait for the task to complete -- it is finished as soon as wln.setwpa is done executing. The on_wln_task_complete 565 event is still generated.

.Task R/O Property

Function: Indicates current wln. task being executed.

Type: Enum (pl_wln_tasks, byte)

Value Range:

0- PL_WLN_TASK_IDLE **(default)**: No task is in progress.

1- PL_WLN_TASK_SCAN: Scan task is in progress (initiated by wln.scan 567).

2- PL_WLN_TASK_ASSOCIATE: Association task is in progress (initiated by wln.associate 557).

3- PL_WLN_TASK_SETTXPOWER: TX power adjustment task is in progress (initiated by wln.settxpower 570).

4- PL_WLN_TASK_SETWEP: WEP mode and keys setup task is in progress (initiated by wln.setwep 570).

5- PL_WLN_TASK_DISASSOCIATE: Disassociation task is in progress (initiated by wln.disassociate 561).

6- PL_WLN_TASK_NETWORK_START: Ad-hoc network creation task is in progress (initiated by wln.networkstart 564).

7- PL_WLN_TASK_NETWORK_STOP: Ad-hoc network termination task is in progress (initiated by wln.networkstop 564).

8- PL_WLN_TASK_SETWPA: WPA mode and keys setup task is in progress (initiated by wln.setwpa 571).

See Also: Wln Tasks 539

Details

The wln. object will only accept another task for execution after the previous task has been completed (wln.task= 0- PL_WLN_TASK_IDLE). Whenever a task completes, an on_wln_task_complete 565 event is generated.

Libraries

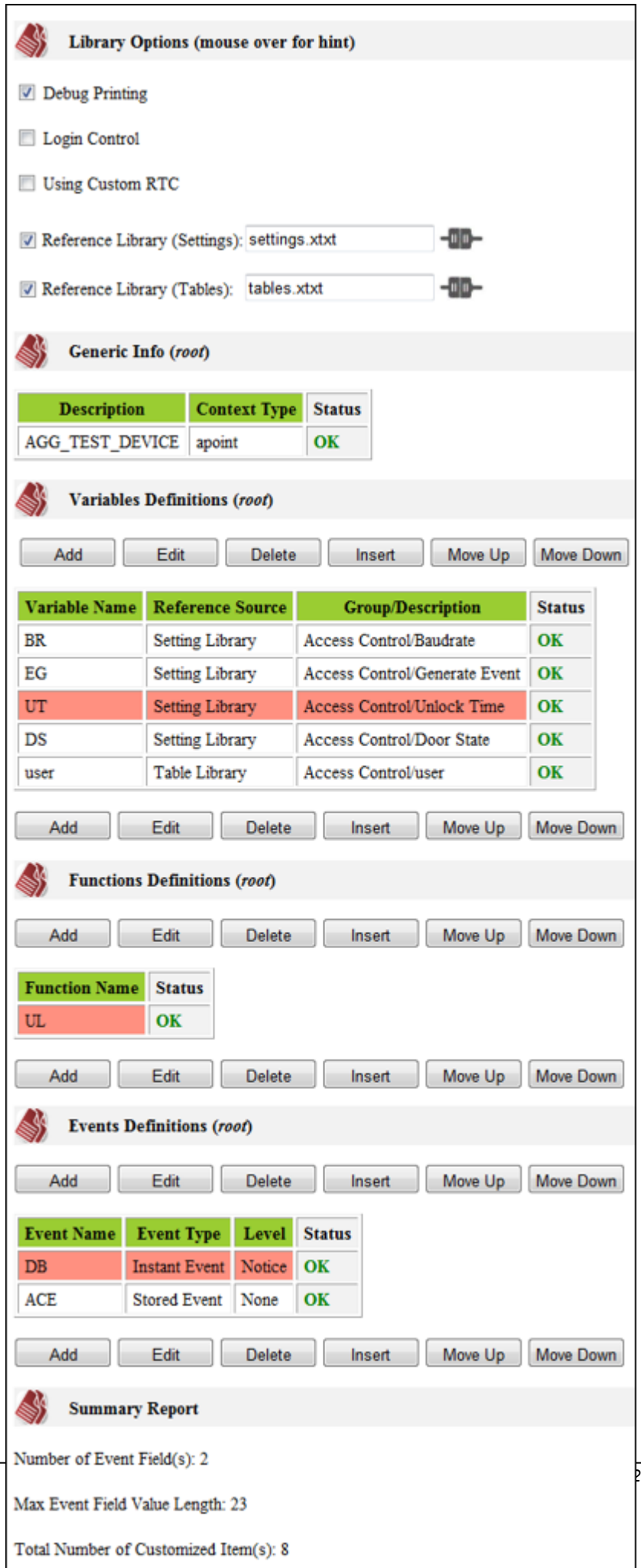
Powerful as they are, our platforms 138 (with their objects 231 and functions 205) still leave you with a fair amount of coding to do for many common tasks. Official Tibbo libraries aim to alleviate this burden by providing a framework of standard plug-and-play code modules that you can freely reuse in your projects.

In this section:

- Common Library Info 575 section explains how our libraries are organized.
- Library Reference 580 contains detailed documentation on all our libraries.

... and here is something to GRAB YOUR ATTENTION:

Many of our libraries now feature **configurators** 578 -- HTML/JavaScript-based editors for configuration files. Trust us, these really simplify your life. Instead of going on and on about why this is cool, we limit ourselves to a descriptive screenshot (it is from the AggreGate library 580):



Library Options (mouse over for hint)

- [x] Debug Printing
- [ ] Login Control
- [ ] Using Custom RTC
- [x] Reference Library (Settings): settings.xtxt
- [x] Reference Library (Tables): tables.xtxt

Generic Info (*root*)

Description	Context Type	Status
AGG_TEST_DEVICE	apoint	OK

Variables Definitions (*root*)

Add | Edit | Delete | Insert | Move Up | Move Down

Variable Name	Reference Source	Group/Description	Status
BR	Setting Library	Access Control/Baudrate	OK
EG	Setting Library	Access Control/Generate Event	OK
UT	Setting Library	Access Control/Unlock Time	OK
DS	Setting Library	Access Control/Door State	OK
user	Table Library	Access Control/user	OK

Add | Edit | Delete | Insert | Move Up | Move Down

Functions Definitions (*root*)

Add | Edit | Delete | Insert | Move Up | Move Down

Function Name	Status
UL	OK

Add | Edit | Delete | Insert | Move Up | Move Down

Events Definitions (*root*)

Add | Edit | Delete | Insert | Move Up | Move Down

Event Name	Event Type	Level	Status
DB	Instant Event	Notice	OK
ACE	Stored Event	None	OK

Add | Edit | Delete | Insert | Move Up | Move Down

Summary Report

Number of Event Field(s): 2

Max Event Field Value Length: 23

Total Number of Customized Item(s): 8

Common Library Info

Below you will find:

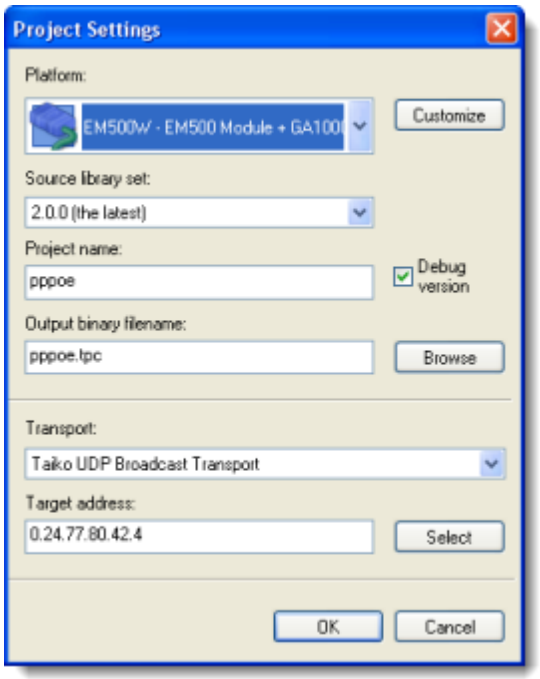
Library Sets

By default, libraries reside in the platforms folder, in the **\src** subfolder ("src" stands for "sources").

Go there and you will discover several internal folders, with names like **0_90** and **2_00**. Each one of those is a **library set**.

All libraries employed in your project always come from the same library set. The idea behind library sets is that you want to freeze the libraries for use in a particular project, while we need to continue expanding them. Once the library set is selected, it stays unchanged (or, rather, we don't change it), and so does your project. If we want to add something, we create a new set and make changes there. You can then use this new library set in your future projects.

The current library set is selected in the Project Settings 38 dialog. When starting a new project, choose the library set with the highest version number. This will be the latest one we've got.



Anatomy of Tibbo Libraries

Most Tibbo libraries are supplied as a pair of files -- a **.tbs** file with all the code, and a **.tbh** header file with declarations. Both must be added[577] to your project.

Tibbo libraries consist of a set of procedures (subs and functions) that typically fall into three categories:

- **API procedures** provide a way for your program to control and interact with the library. For example, the DHCP[668] library has a dhcp_start()[636] function that launches the DHCP client on the specified network interface. It is usually up to you to select when and how to invoke API procedures.
- **Event procedures** must be called from corresponding event handlers and you don't have any leeway in how you implement this. Each event procedure must be properly called, or the library will not work correctly. Typically, you only need to call the procedure without any additional conditional code surrounding the call. If several libraries "hook" onto the same event then you can call event procedures of different libraries in any order.
- **Callback procedures** have their bodies outside of their library. That is, the libraries will expect you to create the bodies for callback procedures elsewhere in your code (and may we suggest that you choose to have them in the **device.tbs** file of your project). Callback procedures are used by libraries to inform your "other code" of events happening within the library. For example, the DHCP[668] library calls the callback_dhcp_ok()[638] procedure whenever a DHCP process completes successfully. Arguments for this procedure carry the newly obtained IP and related parameters. You can then place necessary code within callback_dhcp_ok(). The is supposed to update your device's IP, etc. according to the data supplied by the DHCP library.

Virtually every library has a number of defines located in the **.tbh** (header) file of the library. Defines determine how the library operates -- they are library "options". Each define looks like this:

```
#ifndef DHCP_DEBUG_PRINT
      #define DHCP_DEBUG_PRINT 0
#endif
```

Libraries that come with configurators[578] take care of all defines automatically. Just edit the stuff in the configurator and you are done. Some libraries don't have configurators, mostly because there isn't much to configure. With such libraries you need to add your own #define statements as described in the documentation.

Do not edit library header files! Instead, put your #define statements into the global.tbh[576] file, like this:

global.tbh:

```
'DEFINES-----------------------------------------------------------------
#define DHCP_DEBUG_PRINT 1 'this overrides the default value, which is 0

'INCLUDES----------------------------------------------------------------
include "dhcp\trunk\dhcp.tbh"
```

Libraries and Platforms

Some libraries can't run on every platform 138 -- there may be conditions or requirements that the platform must satisfy in order for a certain library to run on it. For example, the WLN library 703 can only run on platforms that support the Wi-Fi (wln. 536) network interface.

Such limitations, if any, are described under "Supported platforms" in the library info table (found on the title page of each library manual).

Adding Library Files to Projects

Most Tibbo libraries are supplied as a pair of files -- a **.tbs** file with all the code, and a **.tbh** header file with declarations. Both must be added to your project.

Reminder: library files you are adding always come from the selected library set 575.

To add library files to your project, follow the instructions under Adding, Removing and Saving Files 18 (the section about adding existing files).

Library files are not copied into your project's directory. They are just "referenced". Notice also how they appear in the Libraries tree node of the project tree 134. Added library files are initially locked for editing. This is why you will see their file names in gray, with the "lib, lock" label printed next to them as well.

You can unlock the files (by right-clicking on the file in the tree and selecting "Unlock"), but this is not recommended.

Do not change the code in the library files. If you feel that a certain library doesn't fully answer your needs and requires modifications, copy the library files into your project's directory, add these files as the project files and then modify the code in the "derivative" files.

About _get_info() API Functions

Most libraries include **_get_info()** API functions (as an example, see dhcp_get_info 636). These are created with a purpose of providing a standard interface for obtaining various library information. "Questions" you application may want to "ask" will depend on the nature of the library.

Each **_get_info()** procedure accepts, as the main parameter, the **info_element** argument which specifies what kind of information is being requested. There is always a corresponding **_info_elements** enum, which lists all information elements available.

There is also a second argument -- **extra_data** (string) -- which is used for passing additional information when necessary.

For universality, **_get_info()** functions return data in a string form. It is your application's responsibility to handle the returned data correctly. For example, if you requested the number of buffer pages required for correct library operation, then your application should convert the returned string into a value.

Library Configurators

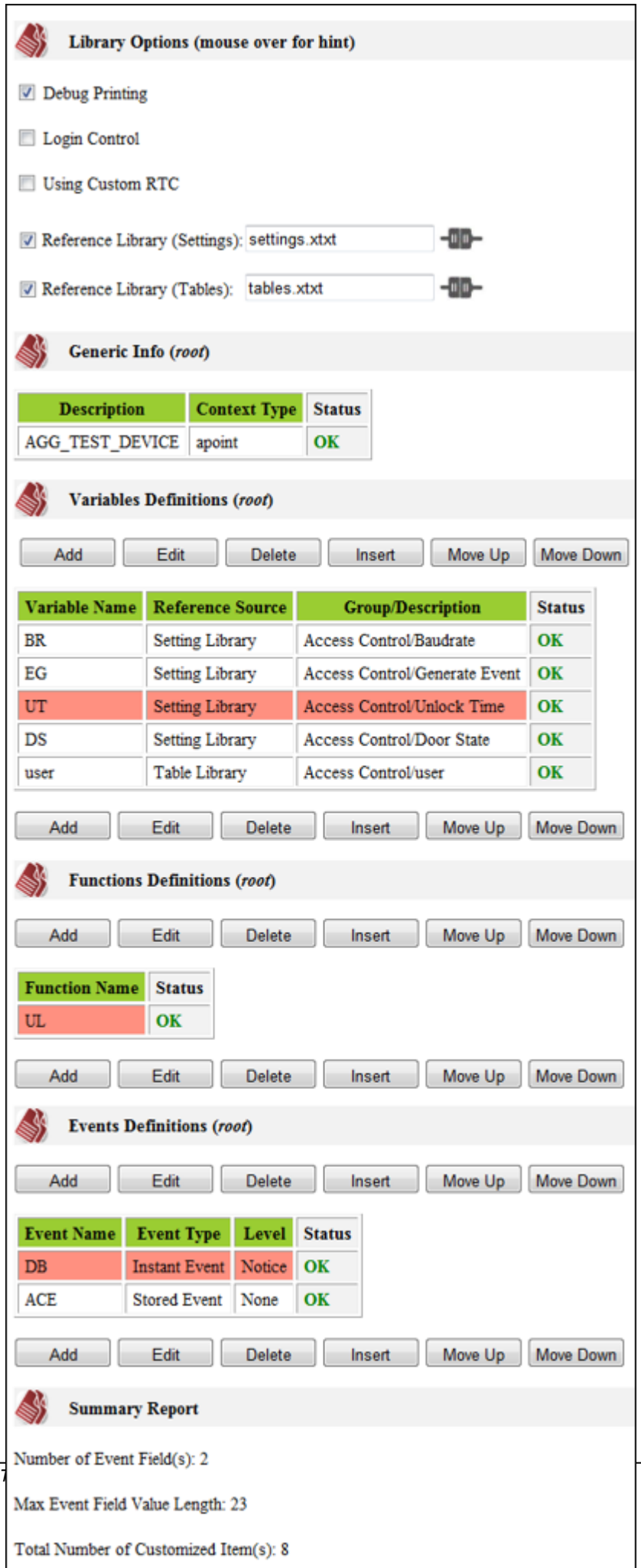
Configurators are HTML/JavaScrip-based editors for configuration files.

Internally, configuration files are text files that contain (define) necessary data (for the library to operate). For example, the configuration file for the AGG (AggreGate) library contains text describing AggreGate variables, functions, and events. Looking at the "unshielded" file reveals some really heavy stuff:

```
==info  root      <R=<AGG_TEST_DEVICE><apoint>>
==variables       root      <R=<version><<<version><S>> <M=1>
<X=1>><Version><1><0><^><remote|General><^>>
~~<R=<date><<<date><D>> <M=1> <X=1>><Date/Time><1><0><^><remote|General><^>>
~~<R=<modtime><<<variable><S>> <<modtime><D><F=N>>><Modification
time><1><0><^><^><^>>
>>BR  S       <R=<BR><<<BR><I><V=<L=0
13>><S=<1200=0><2400=1><4800=2><9600=3><19200=4><38400=5><57600=6><115200=7>
<150=8><300=9><600=10><28800=11><230400=12><460800=13>>><M=1><X=1>><Baudrate>
<1><1><^><remote|Access Control><^>>
>>EG  S       <R=<EG><<<EG><B>><M=1><X=1>><Generate Event><1><1><^><remote|Access
Control><^>>
>>UT  S       <R=<UT><<<UT><I><V=<L=0 255>>><M=1><X=1>><Unlock
Time><1><1><^><remote|Access Control><^>>
>>DS  S       <R=<DS><<<DS><I><S=<Closed=0><Opened=1>><V=<L=0
1>>><M=1><X=1>><Door State><1><0><^><remote|Access Control><^>>
>>user T      <R=<user><<<user_id><S><V=<L=1 14>><A=123456>><<name><S><V=<L=0
31>>><M=0><X=1024>><user><1><1><^><remote|Access Control><^>>
==functions     root
<R=<hash><<<context><S>><<variable><S>><M=1><X=1>><<<hash><I>><M=1><X=1>><Table
Hash><^><^>>
~~<R=<finishTable><<<context><S>><<variable><S>><M=1><X=1>><<<hash><I>><M=1><X=1>><
Finish Table><^><^>>
~~<R=<startTable><<<context><S>><<variable><S>><M=1><X=1>><<M=0><X=0>><Start
Table><^><^>>
~~<R=<addRecord><<<context><S>><<variable><S>><<record><S>><M=1><X=1>><<M=0><X=0>>
<Add Record><^><^>>
```

We are sure you'd rather look at this (see below).

Descriptor files are added like any other files (see Adding, Removing and Saving Files 181). The **Type** is set to *Configuration File*, and then the **Format** is set to the format of the library you are dealing with.



Library Options (mouse over for hint)

- [x] Debug Printing
- [ ] Login Control
- [ ] Using Custom RTC
- [x] Reference Library (Settings): settings.xtxt
- [x] Reference Library (Tables): tables.xtxt

Generic Info (*root*)

Description	Context Type	Status
AGG_TEST_DEVICE	apoint	OK

Variables Definitions (*root*)

Add | Edit | Delete | Insert | Move Up | Move Down

Variable Name	Reference Source	Group/Description	Status
BR	Setting Library	Access Control/Baudrate	OK
EG	Setting Library	Access Control/Generate Event	OK
UT	Setting Library	Access Control/Unlock Time	OK
DS	Setting Library	Access Control/Door State	OK
user	Table Library	Access Control/user	OK

Add | Edit | Delete | Insert | Move Up | Move Down

Functions Definitions (*root*)

Add | Edit | Delete | Insert | Move Up | Move Down

Function Name	Status
UL	OK

Add | Edit | Delete | Insert | Move Up | Move Down

Events Definitions (*root*)

Add | Edit | Delete | Insert | Move Up | Move Down

Event Name	Event Type	Level	Status
DB	Instant Event	Notice	OK
ACE	Stored Event	None	OK

Add | Edit | Delete | Insert | Move Up | Move Down

Summary Report

Number of Event Field(s): 2

Max Event Field Value Length: 23

Total Number of Customized Item(s): 8

Library Reference

This library reference documents the library set **V2.0.0**.

This is the library set which is recommended for use with all new projects.

Here is the breakdown of available libraries, by levels. The levels reflect library dependencies. For example, the SOCK 664 and FILENUM 641 libraries are on the lowest level because they do depend on any other libraries. The STG 668 (settings) library is one level higher because its operation requires the services of the FILENUM library (albeit optionally).

Dependencies may be "hard", soft, or conditional. Hard dependency is when "library A relies on library B". For example, the DHCP 618 library needs the SOCK library, or the project won't even compile.

Soft dependencies are caused by the anticipated logic of operation. For example, the DHCP library does not technically need the STG library. However, as demonstrated in the DHCP sample code 622, it is nice to have the STG library in the project because it can help us store the obtained IP. Hence, the DHCP library sits higher than the STG library.

Example of a conditional dependency: the DHCP library doesn't require the WLN 703 library, unless you want to run DHCP over the Wi-Fi interface, in which case the WLN library becomes necessary.

High (top)
AGG 580 (AggreGate)
DHCP 618
WLN 703 (Wi-Fi Association); GPRS 645 (PPP link negotiation); PPPOE 655 (PPPoE Login)
STG 668 (Settings); TBL (Tables) [not yet documented]
SOCK 664 (Socket numbers); FILENUM 641 (File numbers); TIME (Date/Time) [not yet documented]
Low (bottom)

The Library Reference section introduces libraries in the top-to-bottom order.

AGG (AggreGate) Library

The AGG library implements an interface to Tibbo AggreGate Server (http://aggregate.tibbo.com). It is the "agent" that runs on your Tibbo BASIC device and connects it to the server using Tibbo's proprietary AggreGate Communication Protocol, or ACP (http://aggregate.tibbo.com/docs/ap_protocol.htm). Among the six ways to connect a device to the AggreGate server (http://aggregate.tibbo.com/technology/connectivity.html), using the library fast-tracks you through either the "second way" or the "fourth way".

The second way -- "Connecting An Existing Device With AggreGate Agent" -- assumes that there is a "legacy" device which is already designed, and possibly in production. This device knows nothing about Tibbo's AggreGate technology, yet you want to connect it to the AggreGate server. In order to achieve this, you use a Tibbo BASIC module or controller (EM1000 143, DS1206 158, etc.) to act as an

"adaptor" that translates between the native protocol (or signals) of the legacy device and the AggreGate server. The fastest way to get this done is to write a Tibbo BASIC application that relies on the AGG library for the ACP implementation.

The fourth way -- "Designing A New Device Based On Programmable Module" -- is for when you are building a new device from scratch and you want this device to work with AggreGate. You take one of our very capable modules (such as the EM1000 143) and use it as the "CPU" of the new device. You then task this "CPU" with all kinds of things that make your device operational. One of the tasks -- communicating with the AggreGate server -- is handled by the AGG library.

The AggreGate Communication Protocol (ACP) is rather complex and would take you quite some time to read through, comprehend, and implement -- have you not had the AGG library. When you are using the library, all this complexity is hidden behind an easy-to-use configurator 583. Fill in all the necessary "items", and the server will know how to work with your device.

Without resorting to rewriting the AggreGate manual (http://aggregate.tibbo.com/docs/), let's remind ourselves that from the AggreGate's point of view, your device is an "object" which comprises items of three kinds: variables (properties), functions (methods), and events. To avoid confusion with the internal variables, methods, and events of Tibbo BASIC, we will refer to AggreGate ones as **A-variables**, **A-functions**, and **A-events**.

An A-variable can be a simple single-value affair, an array of values, or a full-blown data table (and it will still be a "variable" in the AggreGate's view). The AGG library relies on the STG (settings) 668 library for storing single-value and array A-variables. This provides for persistent and reliable storage of A-variables, and also -- through pre-gets and post-sets 683 -- for controlling and monitoring the device operation through A-variables (settings). Naturally, the latter will require the custom code to be written and placed into callback_stg_pre_get() 700 and callback_stg_post_set() 701.

Table A-variables are handled through the **TBL (tables)** library [not yet documented].

A-functions and A-events require custom code to be written, so it's not enough to just put necessary data into the configurator 583.

Library Info

Supported platforms:	Any platform. If the platform of your choice doesn't support the RTC 375 you will have to implement date/time-keeping in your own way (for example, by connecting an external RTC chip and writing necessary code).
Files to include:	aggregate.tbs, aggregate.tbh (from *current_library_set*\aggregate\trunk\).
Dependencies:	SOCK 664 library; TIME library [not yet documented]; WLN 703 library if your application will be connecting to the AggreGate server through Wi-Fi; GPRS 645 library if your application will be connecting to the AggreGate server through GPRS.
API procedures:	agg_start() 607 -- starts the AGG library, parses the configuration file, prepares all necessary variables.

Use API procedures 576 to interact with the library.

agg_stop() 608 -- stops the AGG library causing it to release occupied buffers.

agg_get_connection_state() 609 -- gets the current state of the connection to the AggreGate server

agg_record_decode() 609 -- extracts the value of the specified argument from the encoded string containing the values of all arguments for an A-function.

agg_record_encode() 610 -- appends the argument's value to the encoded A-function's argument string in preparation for sending this string to the AggreGate server.

agg_fire_instant_event() 610 -- generates an A-event and sends it to the AggreGate server without storing it into the log.

agg_stored_event_added() 611 -- must be called every time a stored A-event is added to the log file.

agg_proc_stored_events() 612 -- nudges the AGG library to process stored A-events.

Event procedures:

agg_proc_timer() 612 -- call this from the on_sys_timer() 533 event handler.

agg_proc_data() 612 -- call this from the on_sock_data_arrival() 489 event handler.

agg_proc_data_sent() 613 -- call this from the on_sock_data_sent() 489 event handler.

agg_proc_sock_event() 612 -- call this from the on_sock_event() 490 event handler.

Callback procedures:

Implement the bodies of callback procedures 576 elsewhere in your project.

callback_agg_get_firmware_version() 613 -- requests the version string for the current application.

callback_agg_device_function() 613 -- invoked when the device needs to execute an A-function.

callback_agg_synchronized() 614 -- informs of the completion of the synchronization process between the device and the AggreGate server.

callback_agg_pre_buffrq() 614 -- called when the library needs to allocate buffer space and the required space is not available.

callback_agg_buff_released() 615 -- called when the library no longer needs buffers and released them.

callback_agg_error() 615 -- informs of an error or condition detected within the library.

callback_agg_convert_setting() 616 -- invoked every time an A-variable is being read or written. Provides an opportunity to change the value or type before writing to a setting or after reading from a setting.

callback_agg_convert_event_field() 617 -- invoked every time a stored A-event is being extracted from the log and sent to the server. Provides an opportunity to change the value or type of A-event arguments before sending the stored A-event to the server.

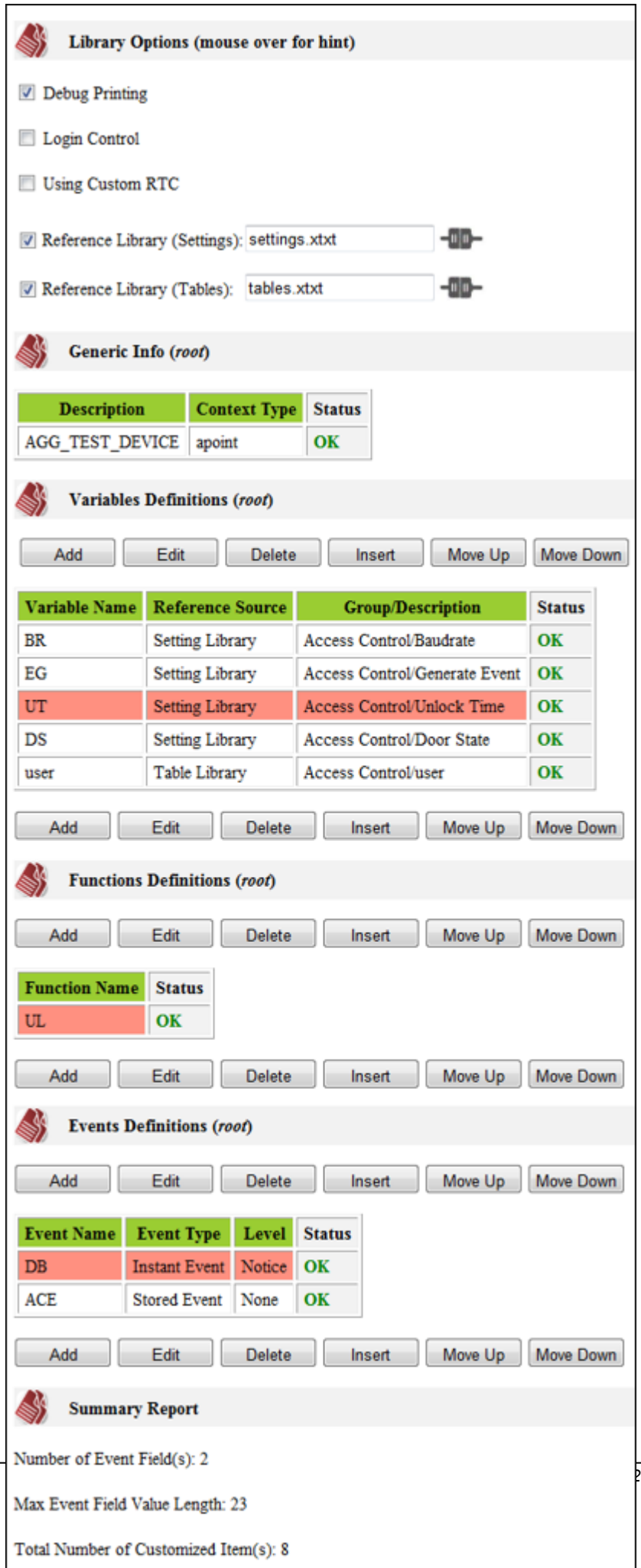
callback_agg_rtc_sg() 618 -- invoked when the library needs to get or set the device's date and time. Necessary only when "use custom RTC" option is selected in the configurator 583.

Required buffer space: 4 buffer pages

AggreGate Configurator

The AggreGate configurator is a JavaScript-based editor 578 for the AggreGate configuration file. Do not confuse the two. The AggreGate configuration file is a part of your project (it is a resource file). The configurator is a "representer" -- it provides a convenient interface for the editing of the configuration file. Along with providing a pleasant interface, the configurator masks the enormous complexity of the underlying configuration file.

The following access control demo 585 will teach you how to use the configurator. For now, enjoy this screenshot...



Library Options (mouse over for hint)

- [x] Debug Printing
- [ ] Login Control
- [ ] Using Custom RTC
- [x] Reference Library (Settings): settings.xtxt
- [x] Reference Library (Tables): tables.xtxt

Generic Info (*root*)

Description	Context Type	Status
AGG_TEST_DEVICE	apoint	OK

Variables Definitions (*root*)

Add | Edit | Delete | Insert | Move Up | Move Down

Variable Name	Reference Source	Group/Description	Status
BR	Setting Library	Access Control/Baudrate	OK
EG	Setting Library	Access Control/Generate Event	OK
UT	Setting Library	Access Control/Unlock Time	OK
DS	Setting Library	Access Control/Door State	OK
user	Table Library	Access Control/user	OK

Add | Edit | Delete | Insert | Move Up | Move Down

Functions Definitions (*root*)

Add | Edit | Delete | Insert | Move Up | Move Down

Function Name	Status
UL	OK

Add | Edit | Delete | Insert | Move Up | Move Down

Events Definitions (*root*)

Add | Edit | Delete | Insert | Move Up | Move Down

Event Name	Event Type	Level	Status
DB	Instant Event	Notice	OK
ACE	Stored Event	None	OK

Add | Edit | Delete | Insert | Move Up | Move Down

Summary Report

Number of Event Field(s): 2

Max Event Field Value Length: 23

Total Number of Customized Item(s): 8

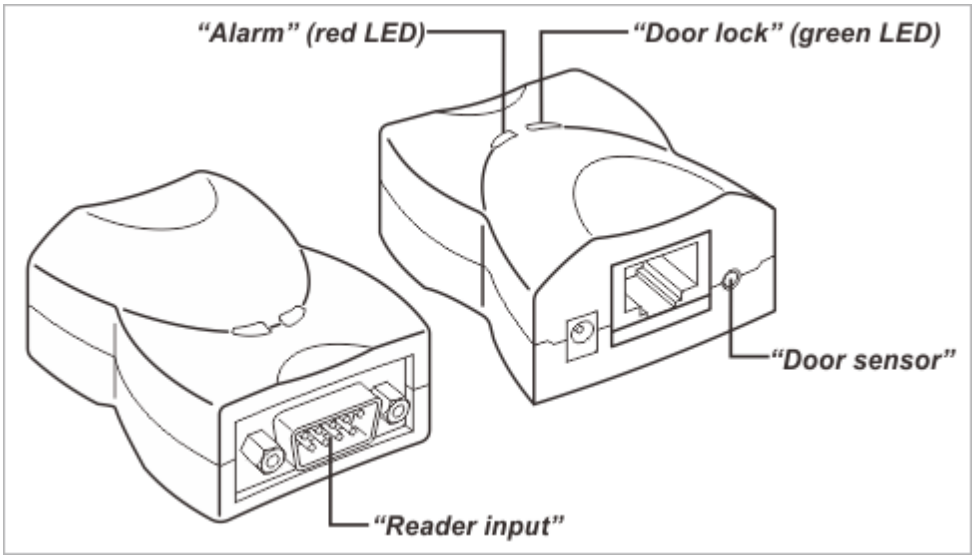
The Access Control Demo

We are going to teach the AGG library by creating a simple access control application. The application is super-minimalistic but it still implements the core of access control functionality:

- Monitoring one door;
- Maintaining a user table (a list of allowed users and their access codes) and granting access;
- Sounding alarm when the door is forced;
- Allowing for remote door opening by the AggreGate operator.

The plan is to allow running this access control demo on most of our BASIC-programmable devices. You will need a bit of a "make believe" to be able to see an access control system in a product like our DS1206:

- We will pretend that the green status LED 200 is the door lock. Green LED ON = unlocked.
- We will further pretend that the red status LED 200 is the alarm relay. Red LED ON = alarm (door forced).
- We will imagine that the MD button 201 is our door sensor. MD button pressed = door opened.
- Finally, we will use the serial port for receiving user codes. You can connect a real card reader, or use I/O NINJA to type in the codes.



The application will work on BASIC-programmable Tibbo devices with RTC. Examples

of such devices:

- DS1206 186 (shown above)
- DS1202 181
- DS1000 164
- EM1000 143 (EM1000EV, EM1000TEV)
- EM1206 158 (EM1206EV)

...

If you are designing a real access control system based on our module or device, you will have no difficulty converting this demo application to driving real relays, monitoring real sensors, etc.

The Steps.1.3

Here is our action plan for creating a simple access control application. To follow the steps, download **test_agg_lib.zip** archive from our website:

http://tibbo.com/basic/resources.html.

The archive contains all the implementation steps as listed below:

1. Prepare 586 (install, configure) the AggreGate server and client.
2. Embryonic project 587 -- the device connects to the server, does nothing else (**test_agg_lib_1**).
3. Add setting A-variables 588 for storing the device configuration (**test_agg_lib_2**).
4. Add table A-variable 593 for storing user codes (**test_agg_lib_3**).
5. Add an A-function 596 for remote door unlocking (**test_agg_lib_4**).
6. Generate instant A-event 598 on device boot (**test_agg_lib_5**).
7. Generate stored A-event 601 for access control activity reporting (**test_agg_lib_6**).
8. Add the glue code 605 that ties it all together (**test_agg_lib_6**).

Preparing the AggreGate Server

The AggreGate Server and AggreGate client software can be downloaded here:

http://aggregate.tibbo.com/downloads.html

Be careful to select correct files!

You can install both the Server and Client components on the same PC.

The demo application you are about to test expects the AggreGate server to have an account named **admin**.

Step 1: The Embryo

This step corresponds to **test_agg_lib_1**.

In this step we are going to create an embryonic application. It will be able to connect to the AggreGate server and transmit the minimally necessary identifying information.

The success of this step depends on many factors. You have to correctly install and prepare 586 the **AggreGate Server** and **AggreGate Client** software. You have to use matching account names on the server and on the device. Passwords must match. The IP address of your device must be correctly chosen. The IP address of the server must be specified correctly, and so on. In other words, pay attention to all the details!

The steps

We assume you are not going to type everything in from scratch and just open the **test_agg_lib_1** project. Notice that...

1. The project contains the SOCK 664 and TIME libraries [not yet documented].
2. **Aggregate.tbs** and **aggregate.tbh** are added 577 to the project as well (from *current_library_set*\aggregate\trunk\). There is also a necessary line in **global.tbh**: **include "aggregate\trunk\aggregate.tbh"**.
3. There is an **aggregate.xtxt** configuration file 583 with type= **configuration file**, and format= **AggreGate (AGG) library**. There is a line in global.tbh that is required for the configuration file to work correctly: **includepp** 96 **"aggregate.xtxt"**.
4. In the configuration file, **Debug Printing** is *enabled*. This allows you to "see what's going on". Don't forget to *disable* this later, after you've made sure that the library operates as expected.
5. Also, the **Description** field is set to *AGG_TEST_DEVICE*. The **Context Type** is *Apoint*.
6. AGG library event procedures are added to event handlers:

   - agg_proc_timer() 612 is in the on_sys_timer() 533 event handler (this library assumes that this event is generated twice per second);

   - agg_proc_data() 612 is in the on_sock_data_arrival() 489 event handler;

   - agg_proc_sock_event() 612 is in the on_sock_event() 490 event handler (the exact line is **agg_proc_sock_event(newstate,newstatesimple)**.
7. There are empty callback procedures in **device.tbs**:

   - callback_agg_get_firmware_version() 613;
   - callback_agg_pre_buffrq() 614;
   - callback_agg_buff_released() 615;
   - callback_agg_error() 615;
   - callback_agg_device_function() 613;
   - callback_agg_synchronized() 614.
8. Callback_agg_get_firmware_version() 613 contains the code that returns the firmware version string. There is a **#define FIRMWARE_VERSION** in **global.tbh**.
9. on_sys_init() 533 calls agg_start() 607 -- this is required to make the library

operational. Notice how we check the code returned by agg_start()!

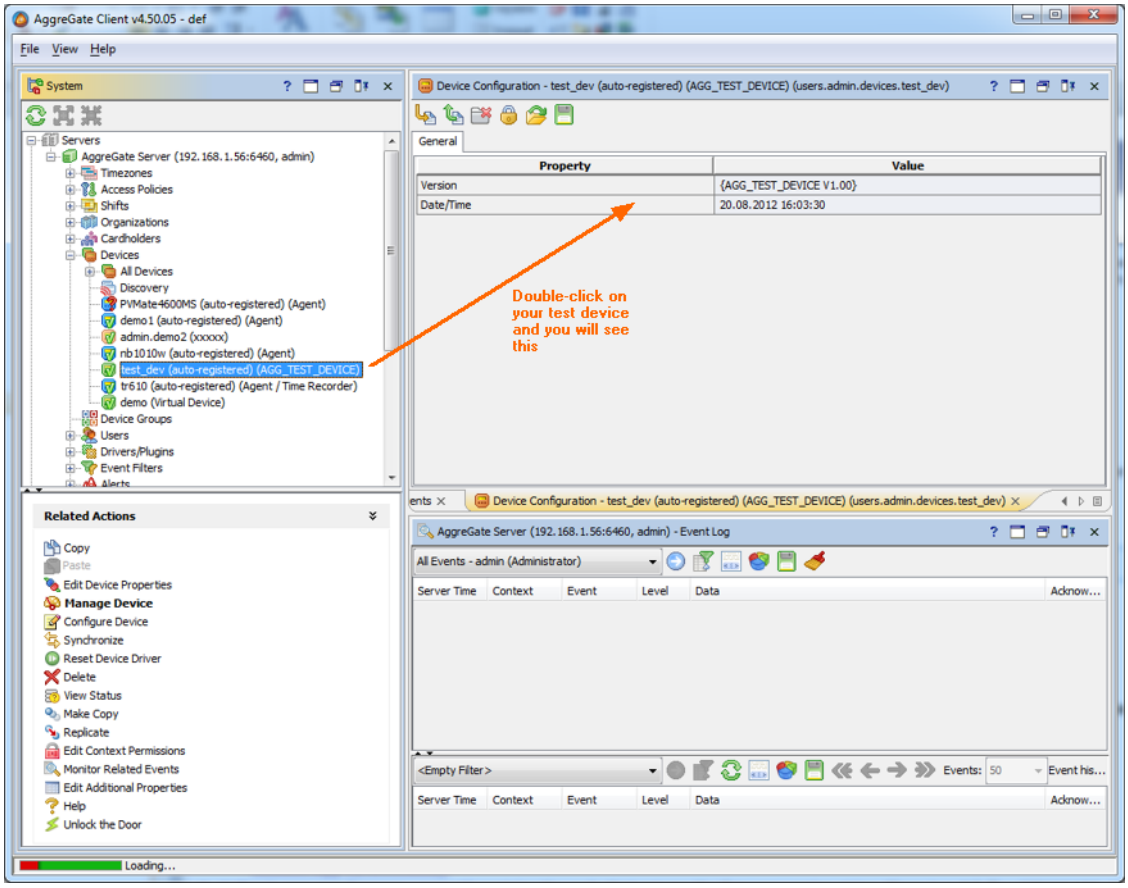
10. Ours is a simple demo and we are sure we won't have memory shortage. In large projects where a lot of things compete for memory you may get the callback_agg_pre_buffrq() 614 call. This will indicate that the AGG library doesn't have enough memory for its buffers and that you need to clear up this memory right inside **callback_agg_pre_buffrq()**. Failure to do so will result in the EN_AGG_STATUS_INSUFFICIENT_BUFFER_SPACE code returned by **agg_start()**.

The result

With debug printing enabled, you will see the following output in TIDE:

```
AGG()> ---START---
AGG()> connection established
AGG()> Device synchronized with the server
```

At the same time your device will appear in AggreGate. Nothing fancy yet, just the basics:



Step 2: Adding Setting A-variables

This step corresponds to **test_agg_lib_2**.

In this step we are going to add necessary setting A-variables. These will be used to store the few configuration and status parameters our application requires.

A-variables we are going to have are:

- The **BR** setting to store the baudrate of the serial port. This will allow us to select a standard baudrate from a list of baudrates (like 9600, 19200, etc.)
- The **EG** setting to enable/disable access control event generation.
- The **UT** setting to store the unlock time for the door. This is the number of *half-second intervals* that the door lock will stay unlocked for (once it is unlocked).
- The **DS** setting for *reading* the current door state (closed or opened).

The steps

Looking now at the **test_agg_lib_2** project, notice that...

1. The STG 668 library is now in the project (and all related steps were taken, including calling stg_start() 691 in on_sys_init() 533, etc.).
2. A way to initialize settings 677 is provided. After each boot, the device will start blinking its red and green status LEDs 200. This will continue for 5 seconds. If you press the MD button 201 within this time, stg_restore_multiple() 692 will be called. Reboot the device after the green status LED is turned on. If you do not press the button within the 5-second time window, the application will continue running. All related code is in the on_sys_init() 533 event handler.
3. **Settings.xtxt** defines required settings 590. Notice that the **Timestamp** option is enabled in the setting configurator 670. This is because the AGG library requires settings to carry timestamps.
4. **AggreGate.xtxt** lists required A-variables 591.
5. There is an additional callback procedure 576 in **device.tbs** -- for callback_agg_convert_setting() 616. Examine the code -- it "converts" between the UT setting (measuring time in half-second intervals) and the UT A-variable (measuring time in seconds). You can now see this procedure's usefulness -- it allows you to store values differently compared to how they are perceived in AggreGate.

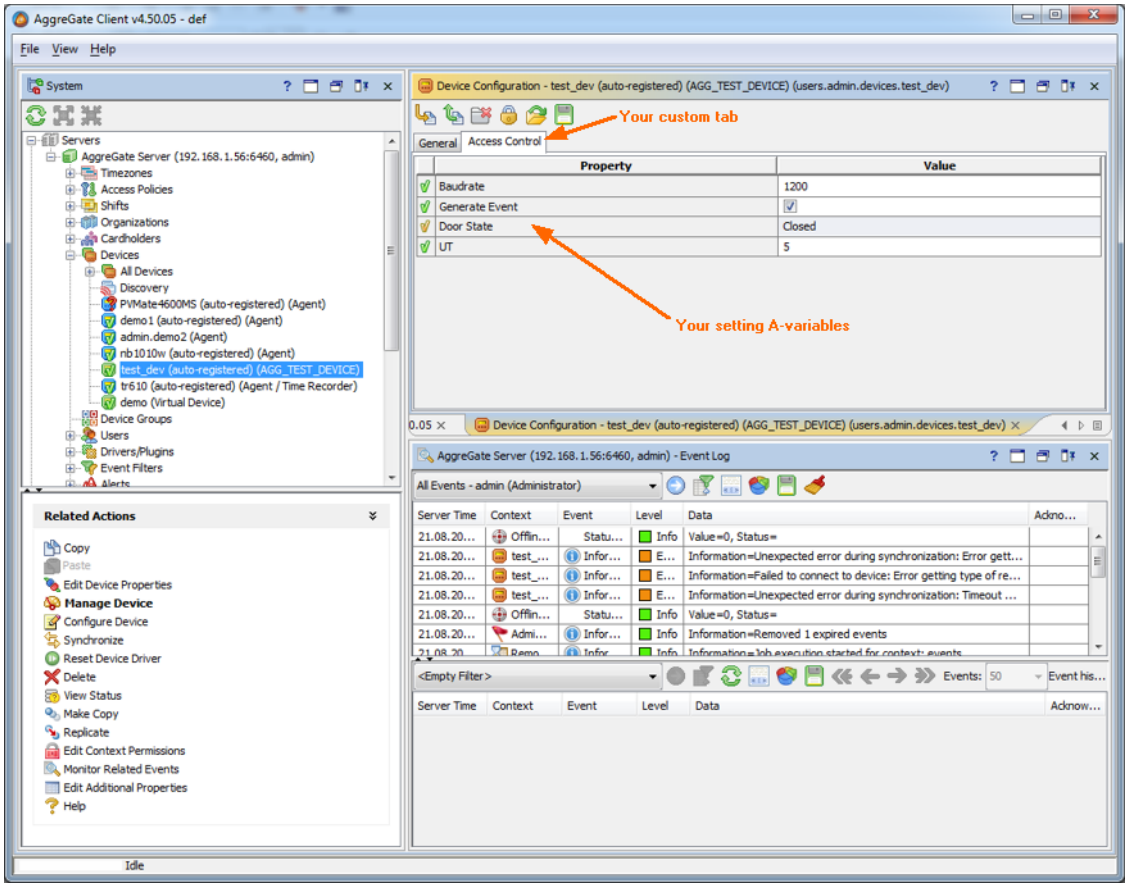
The result

You will now be able to see newly created A-variables in the AggreGate Client (as shown on the screenshot below). Notice how all four A-variables appear on the **Access Control** tab? This is defined 591 in the AggreGate configurator.

Notice also that each A-variable is displayed/edited differently:

- The Baudrate (**BR**) setting has a drop-down which shows all available values;
- The Generate Event (**GE**) setting as a checkbox because it is of the boolean type;
- The Unlock Time (**UT**) setting is just a plain value that is expressed in seconds (although the **UT** setting is in half-seconds).
- The Door State (**DS**) setting displays custom values "closed" and "opened" and is read-only.

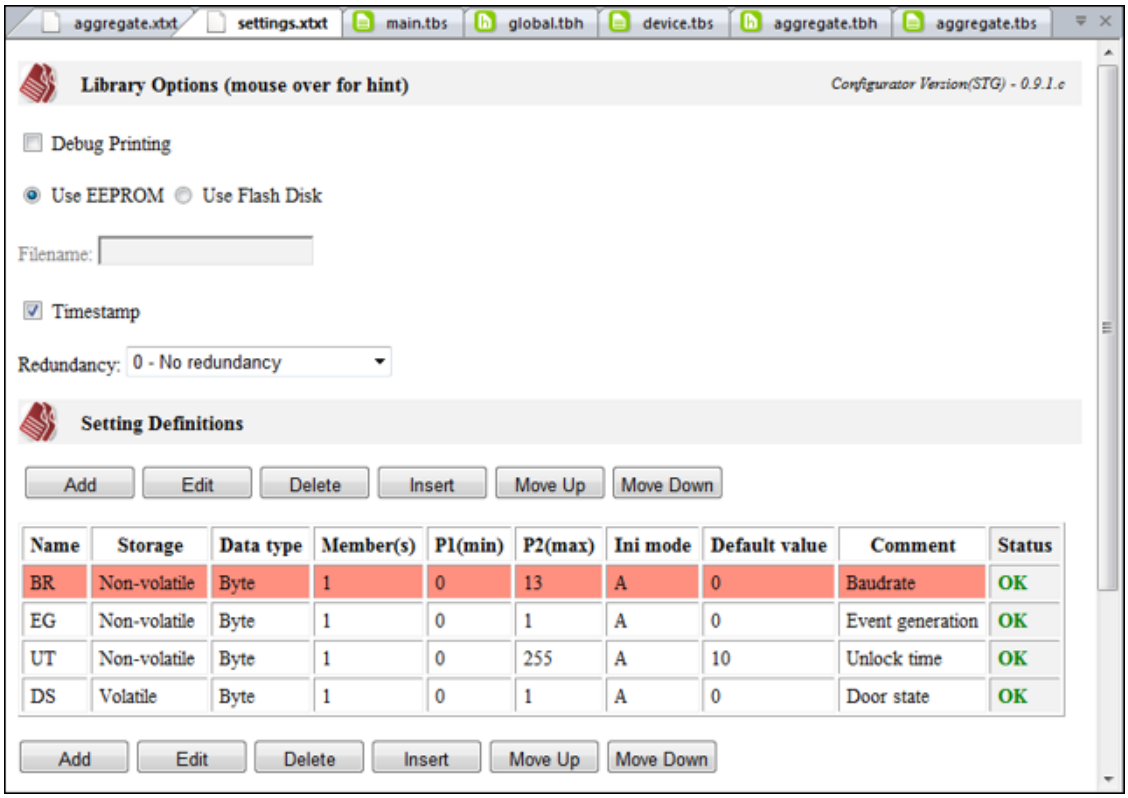
Setting A-variables as we have them now are not doing anything useful on the device just yet. They simply exist as values. We will glue it all together 605 later.



Define Required Settings

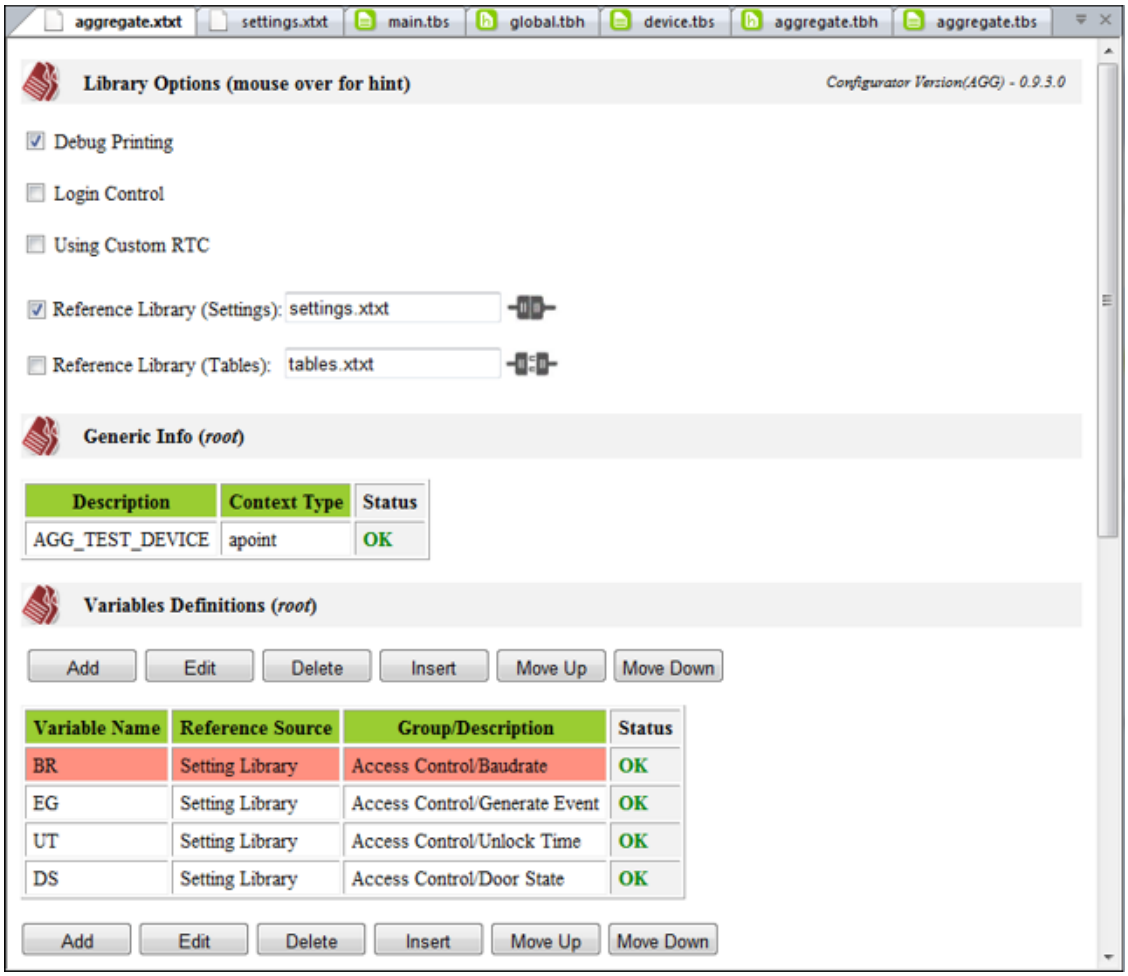
Add **BR**, **EG**, **UT**, and **DS** variables using STG library's configurator 670. You should end up with a setting table as shown below. Some things to notice:

- The **Timestamp** is enabled. This is *required* by the AggreGate library.
- The **BR** setting has the maximum value of 13. This is because we will support 14 different baudrates (0~13).
- The **EG** setting has the maximum value of 1. On the AggreGate side it will be boolean, hence there are only two possible values for it: 0 and 1.
- The **UT** setting is limited to 255 max. This is measure in half-second intervals (this is how often the on_sys_timer 533 event is generated).
- The **DS** setting resides in the volatile memory (RAM). This is a read-only parameter that simply reflects the door state, so it makes no sense to put it into the EEPROM.



Define Required A-variables

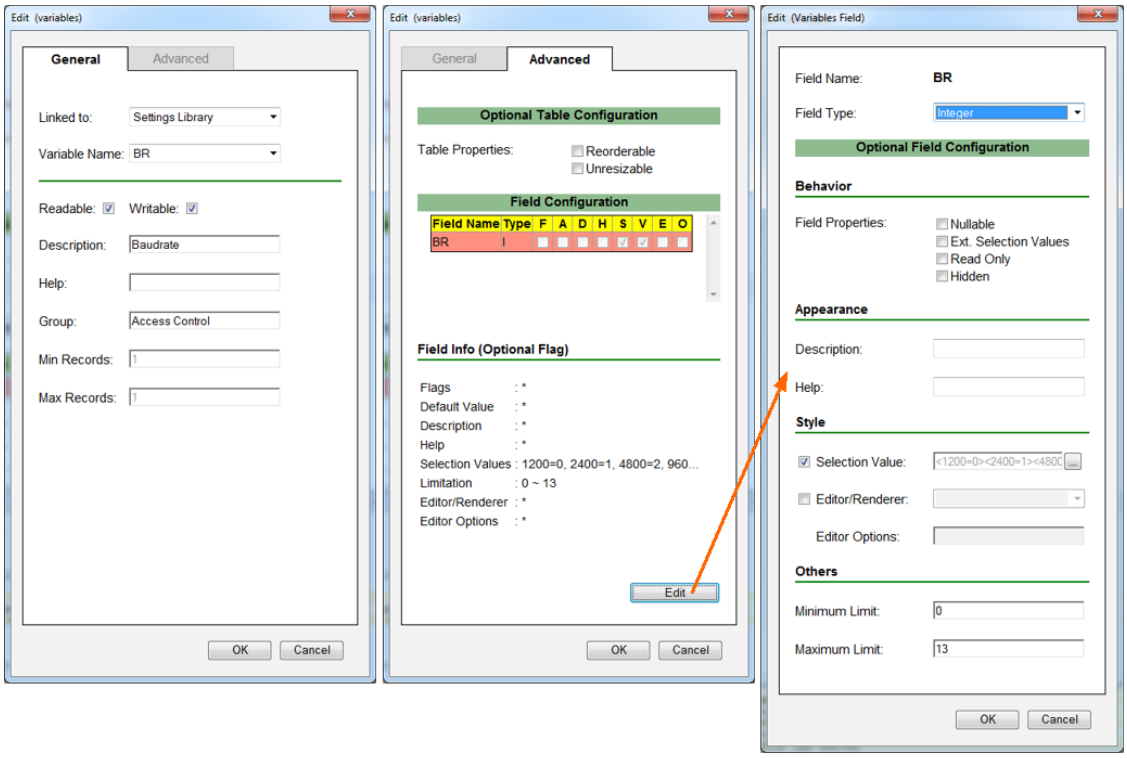
Add **BR**, **EG**, **UT**, and **DS** setting A-variables using AGG library's configurator 583. You should end up with a setting table as shown below:



Note that **Reference Library (Settings)** must be *enabled* and correct setting configurator 670 file must be specified (*settings.xtxt*). We are sure you will quickly find your way around the AggreGate configurator. Let's just briefly review some things worth noting. Double-click on the **BR** A-variable line -- the **Edit (A-variables)** dialog will open.

- On the **General** tab, notice that the A-variable is **Linked to** the *Settings Library*. The **Variable Name** drop-down contents reflect available settings. Your A-variable is basically a "propagated" setting! You can also link to tables -- this will be our next step's task 593.
- Notice also that the A-variable is **Readable** and **Writable**. All our A-variables are, except **DS**, which is only **Readable** (read-only).
- **Description** is set to *Baudrate*, and this is how this A-variable is visible in AggreGate.
- The **Group** is set to *Access Control*. This is where the **Access Control** tab you see in the AggreGate client comes from. The client will create separate tabs for all groups encountered. We only use one custom group in this project.
- Now switch to the **Advanced** tab and press **Edit**. Don't we have a lot here!
- Notice how the **Field Type** is *Integer*. The AggreGate server supports a different type set, and *byte* isn't on the list. The nearest suitable type is selected, which is *integer*.
- **Selection Value** is enabled and if you click **...** you will see the whole list. Now you know where the drop-down selector on the **Baudrate** A-variable comes from.

- **Maximum Limit** will not allow you to set anything above *13*.
- There are numerous other fields we haven't touched. Explore and you shall discover!



Moving now to the remaining A-variables, and only touching on the notable differences:

- The **EG** A-variable has its **Field Type** = *Boolean*.
- The **UT** A-variable is limited to 127.
- The **DS** A-variable has two custom **Selection Values** (*opened* and *closed*). it is only **Readable**, and not **Writable**.

Step 3: Adding Table A-variables

This step corresponds to **test_agg_lib_3**.

This is an access control project, hence it requires a user table which will keep user names and ID codes. In this step we create this **user** table.

The steps

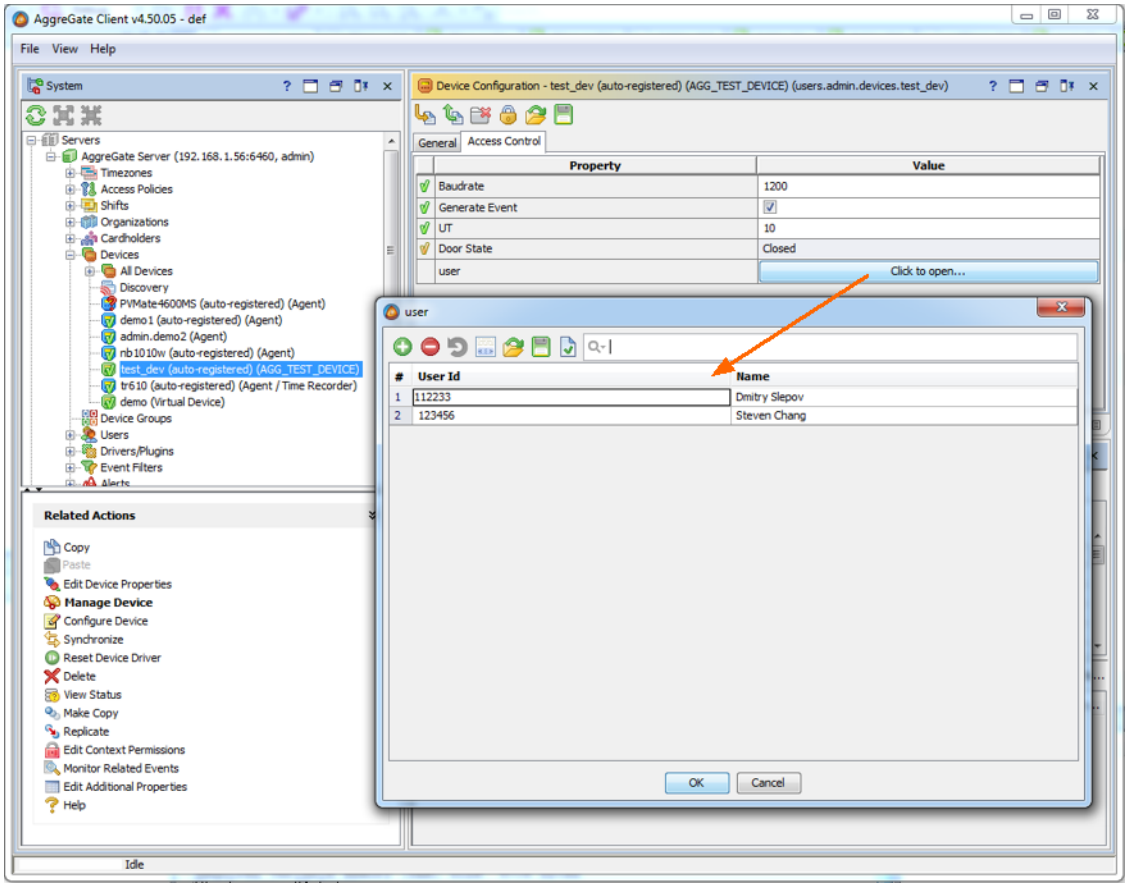
Notice that...

1. The **TBL library** [not yet documented] is now in the project. Inspect the on_sys_init() 533 event handler -- it contains several things related to the library's operation: the flash disk is mounted and formatted if necessary. The flash disk is also formatted during the initialization. **Tbl_start()** [not yet documented] is called.
2. The FILENUM 641 library is now in the project -- it is required by the TBL library.
3. **Tables.xtxt** defines the USER table 594.
4. **Aggregate.xtxt** contains new table A-variable 595.

5. Notice how the **AggreGate Hash** option is enabled in the table **configurator** [not yet documented]. This is because the AGG library requires this.

The result

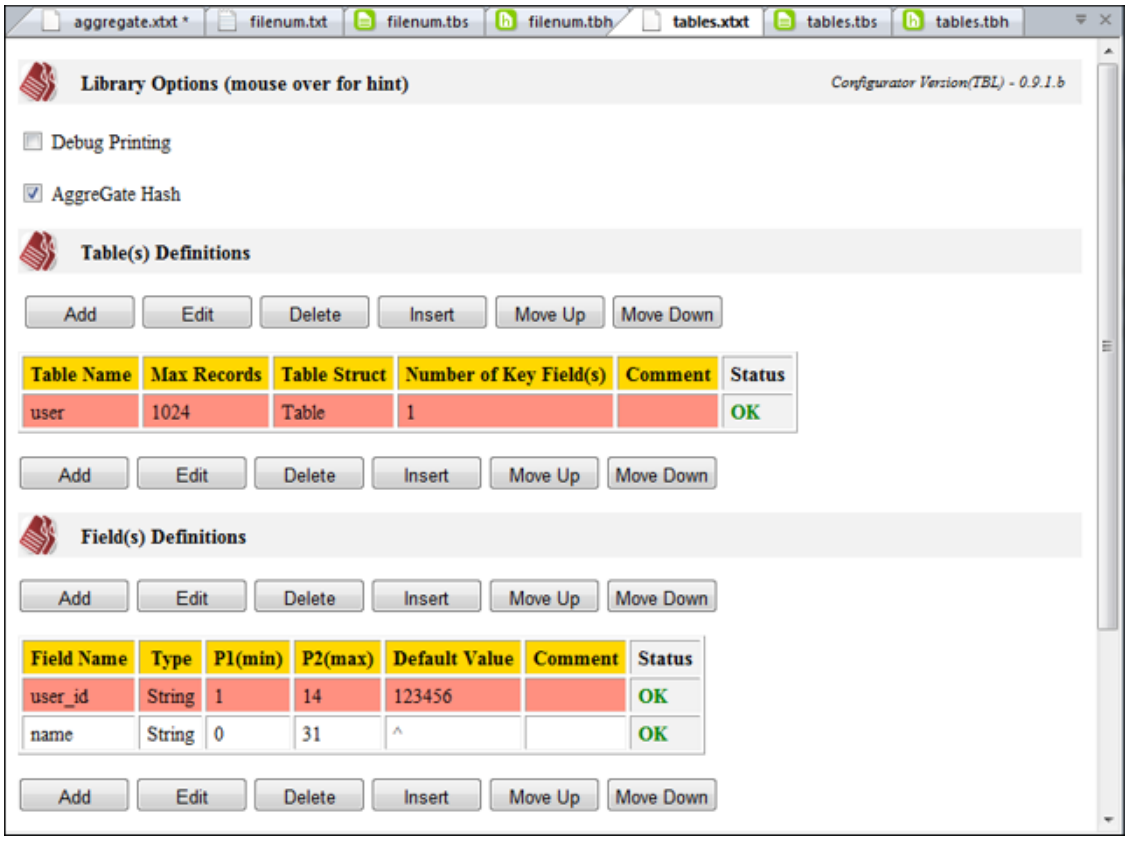
Now you have an editable **user** table! It isn't yet "connected" to anything in any useful way -- we will take care of this 605 later.



Define the User Table

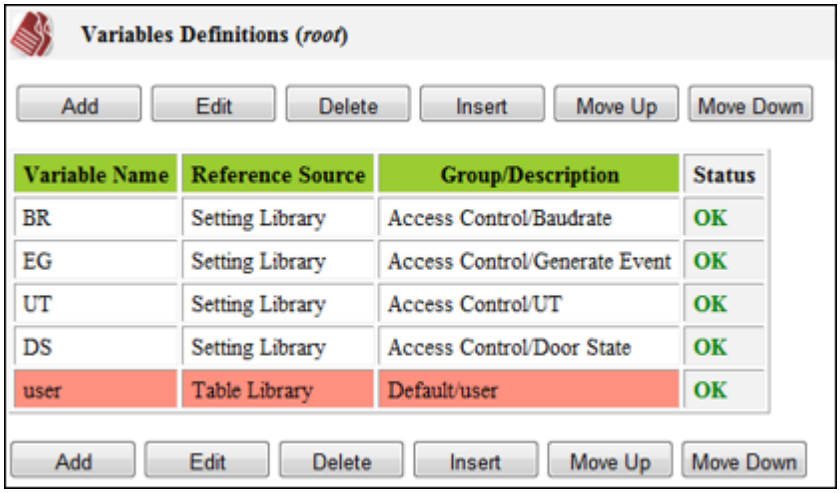
This table **configurator** [not yet documented] screenshot shows the added **user** table. Note that...

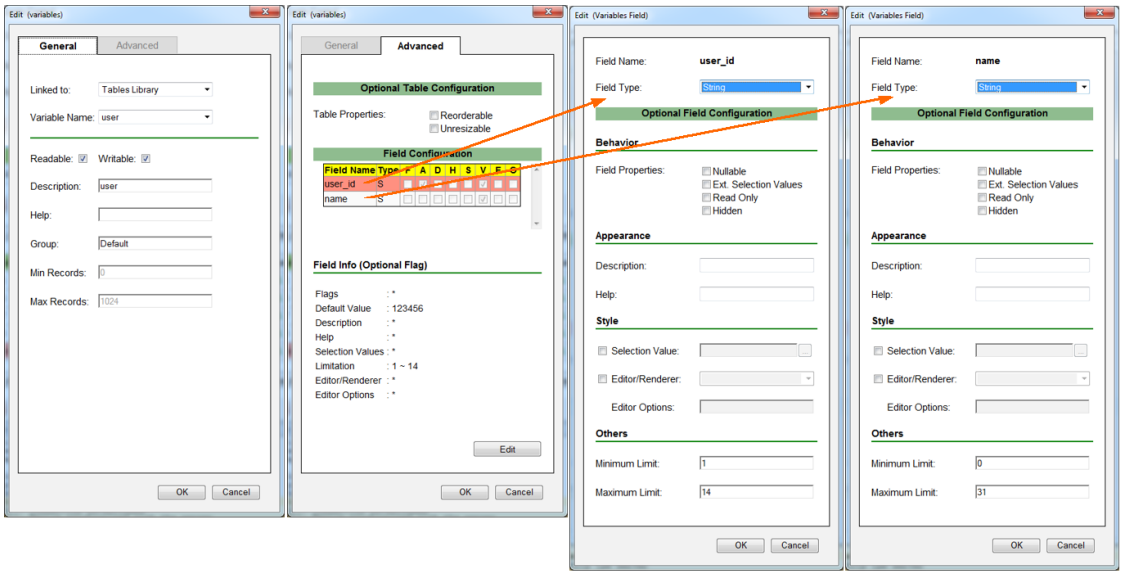
- **AggreGate Hash** is enabled. This is required by the AggreGate library.
- The **user** table has *one* **key field**. It is the first one in the list -- the **user_id** field.



Add the Table A-variable

These AggreGare configurator 583 screenshots show the added **user** table A-variable.





Step 4: Adding A-functions

This step corresponds to **test_agg_lib_4**.

A-functions are methods that the AggreGate server can execute on the device. In our access control project, we will add an "open door" A-function which will be used for remote unlocking of the door. This A-function will have an argument -- for how long the door will have to remain unlocked.

The A-function's argument will be of an overriding nature. When set to 0, it will have no effect and the door will be unlocked for the period of time specified by the UT A-variable 588. When not zero, the argument will override the **UT** A-variable *once*.

The steps

Notice that...

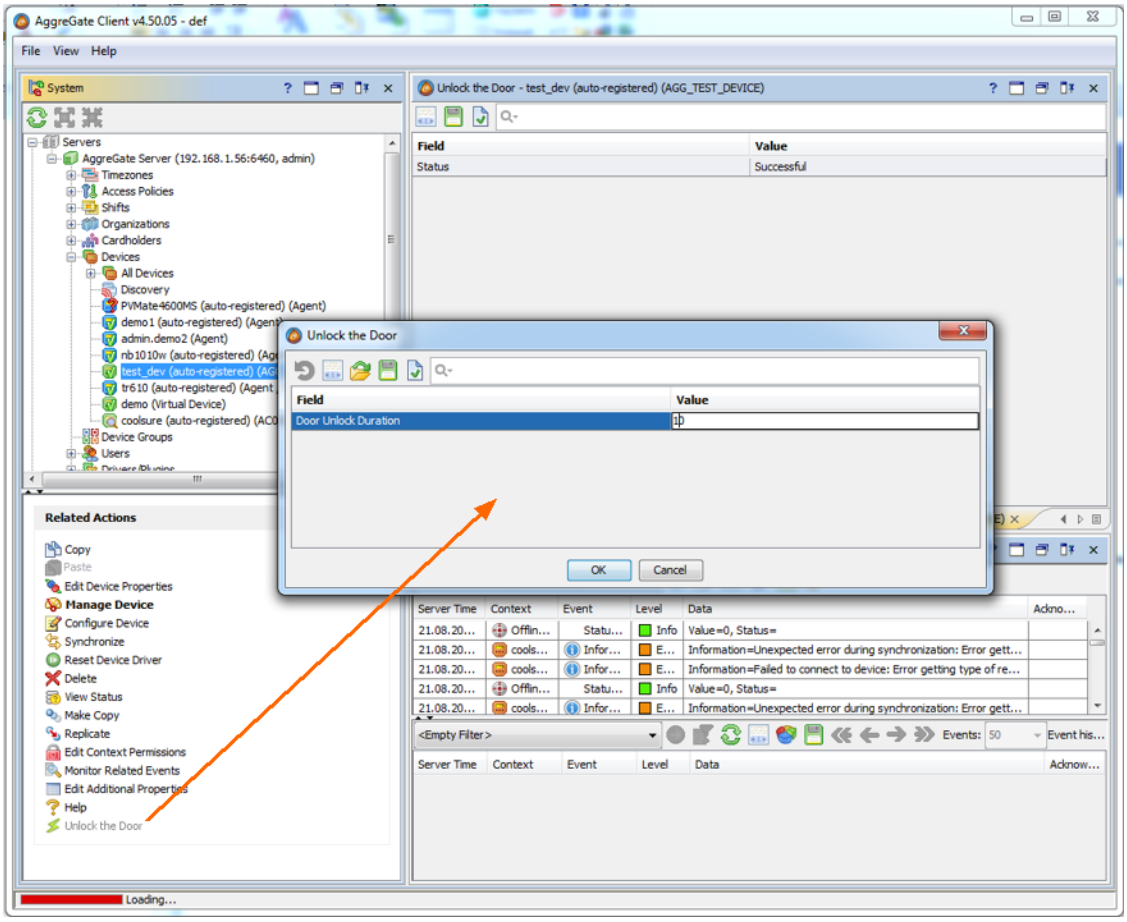
1. **Aggregate.xtxt** now contains the UL A-function 597.
2. Callback_agg_device_function() 613 implements necessary code for executing the **UL** A-function.
3. Notice how Agg_record_decode() 609 is used to extract the argument of the A-function.
4. Agg_record_encode() 610 is used to return A-function value to the AggreGate server. The value is always "success" in our case.

The result

Single-click on the device in the AggreGate tree, and locate **Unlock the Door** in the **Related Actions** pane below. Click **Unlock the Door**, and you will get a dialog requesting **Door Unlock Duration**. Input a value, press **OK**. The function will be executed and return *Success*.

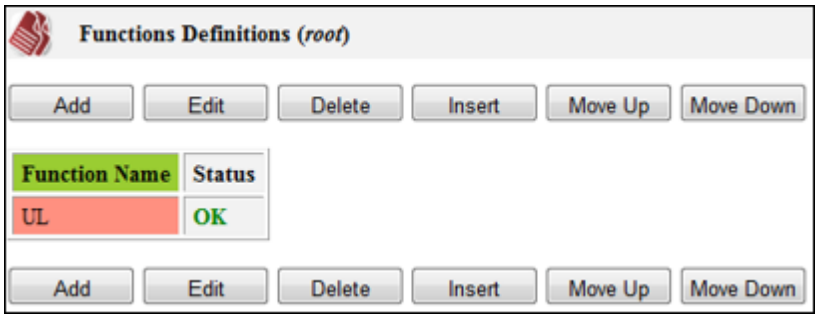
Of course, there is no code (yet) that actually unlocks the door. We will take care of this later 605.

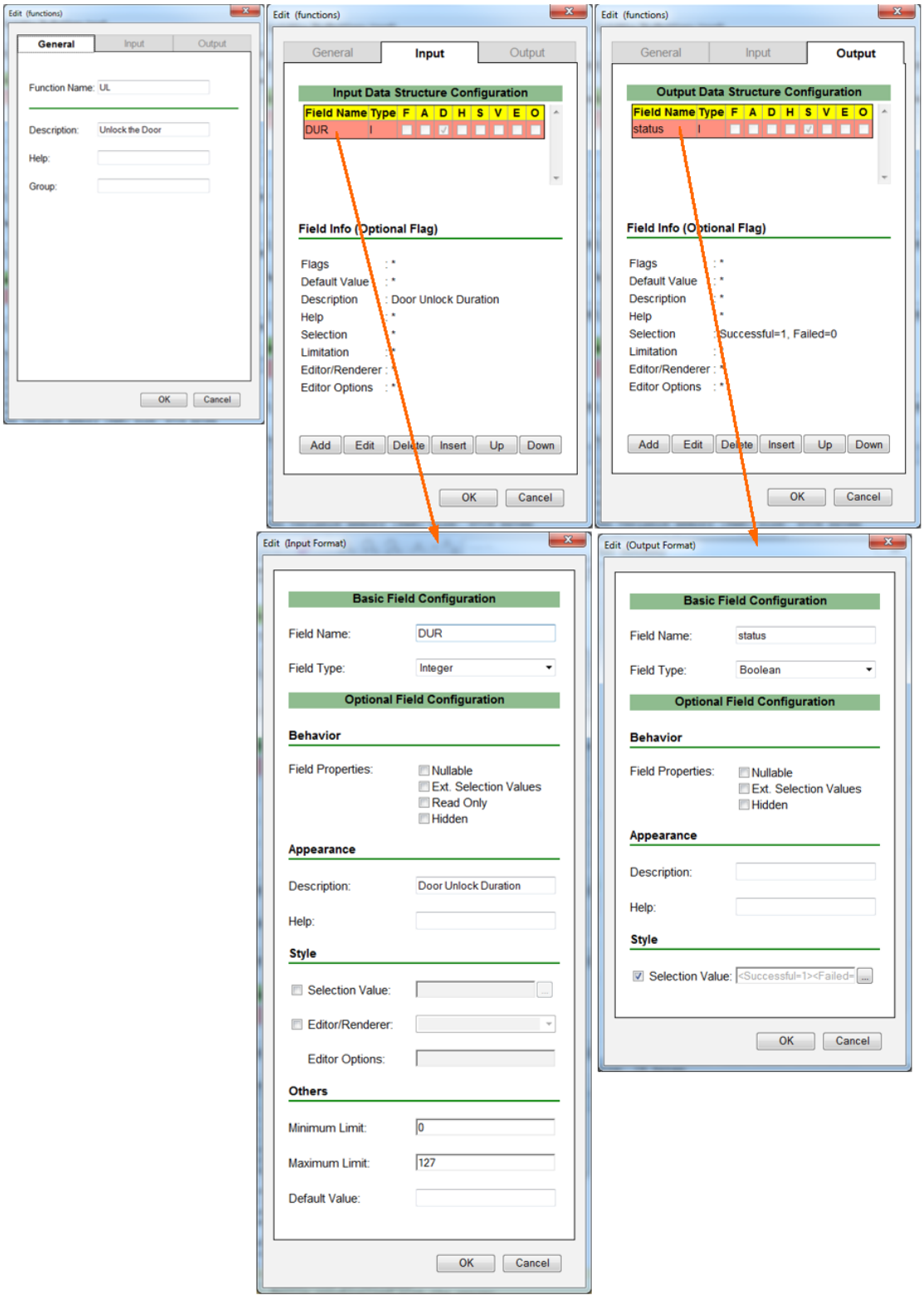
- Can't see **Unlock the Door** in the list? Navigate away from your test device, i.e. click on something else in the tree. Come back -- and you will see the newly added A-function.



Adding A-function

These AggreGare configurator 583 screenshots show the added **UL** A-function.





Step 5: Firing Instant A-events

This step corresponds to **test_agg_lib_5**.

Instant A-events are sent to the AggreGate server "on the spot". They are "lightweight" -- there is no dealing with storing them on the flash disk, etc. They are fast(er) -- just send! They are more reliable -- no flash disk complexity involved. They are also easier to be lost. Should your device lose power at a wrong

moment, the instant A-event will be forgotten. Another limitation: instant A-events can only be fired successfully when there is an active connection to the AggreGate server. Bottom line: use instant A-events for events that are not-so-important and/or numerous.

In this step we add an instant A-event for reporting the booting (powering up) of the device.

The steps

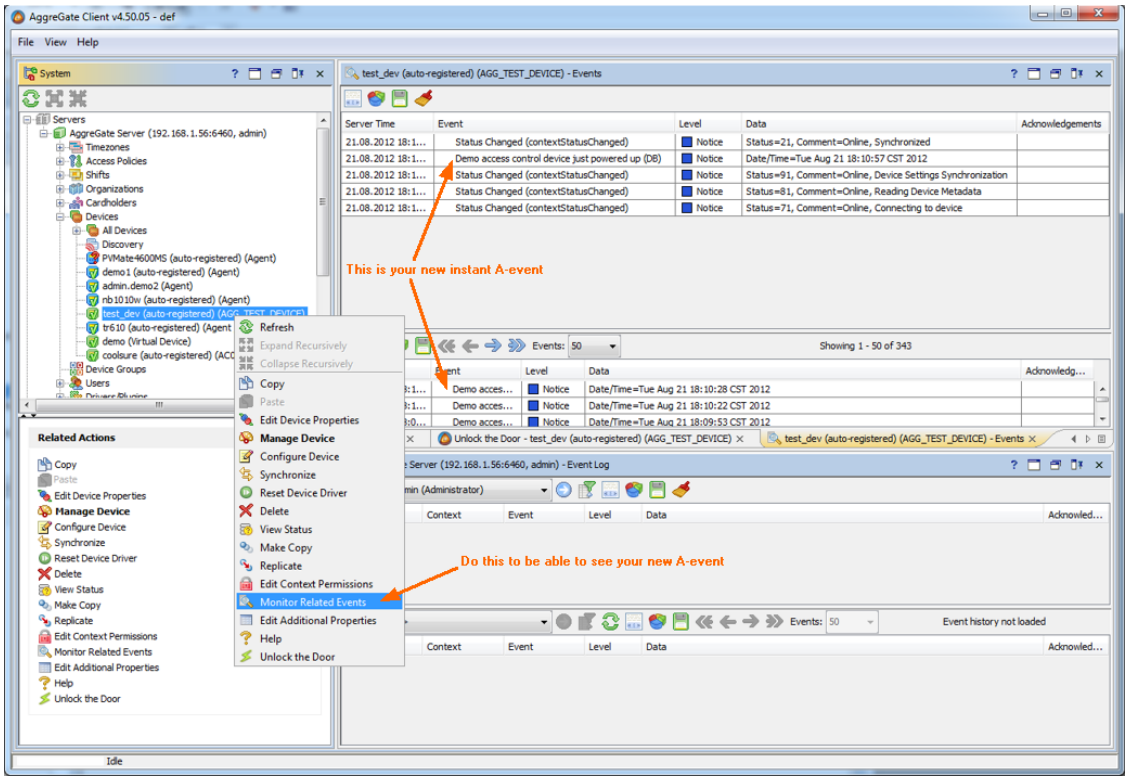
Notice that...

1. **Aggregate.xtxt** now contains the DB instant A-event 600. This event has only one field -- the date/time of the event. It is possible to have additional fields, of course.
2. There is a new **generate_boot_event()** sub is in **device.tbs**. It calls agg_fire_instant_event() 610 to send the instant A-event to the AggreGate server. We call **generate_boot_event()** from callback_agg_synchronized() 614, not on_sys_init() 533. As was explained above, you need to have an established connection to the AggreGate server in order to be able to fire instant A-events, and the connection is ready when callback_agg_synchronized() is called.
3. Notice how this procedure uses agg_record_encode() 610.
4. Notice also the use of the TIME library [not yet documented] for producing the date/time field of correct format.

Notice how the add_fire_instant_event() 610 call has the **event_level** argument. The level defined by this argument will override the default level set in the configurator 600, unless you set the **event_level**= *EN_AGG_EVENT_LEVEL_USE_DEFAULT,* in which case the default event level will be used.

The result

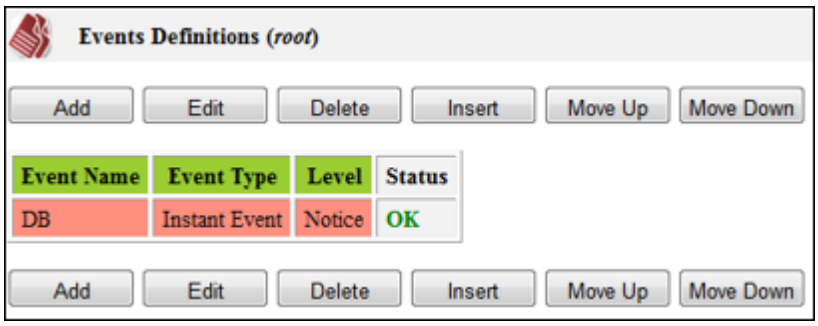
To be able to see *instances* of your new event, right-click on the device in the tree and choose **Monitor Related Events**.

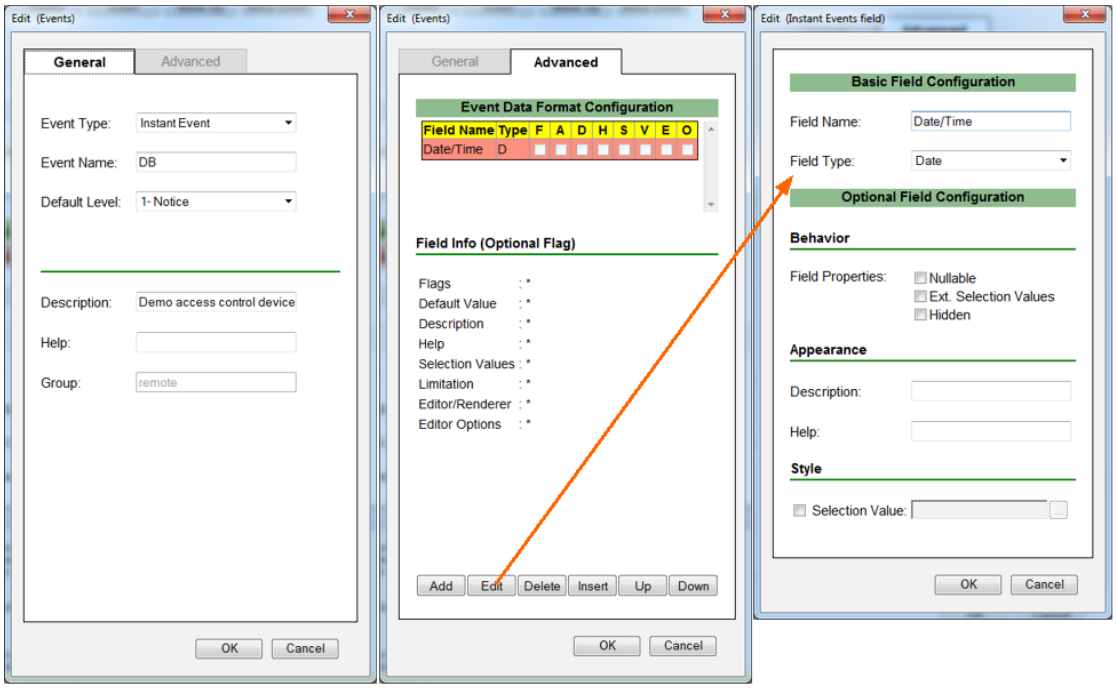


Adding Instant A-event

These AggreGare configurator 583 screenshots show the added instant A-event.

Notice the **Default Level** field. This is the event level that will be reported if agg_fire_instant_event() 610 is called with **event_level**= *EN_AGG_EVENT_LEVEL_USE_DEFAULT*.





Step 6: Handling Stored A-events

This step corresponds to **test_agg_lib_6**.

Stored A-events, as their name implies, are first stored on the device. Stored A-events are kept in the log table, and log tables are handled by the **TBL library** [not yet documented]. Each stored A-event is kept in the log until the AGG library has a chance to send it to the AggreGate server. Once recorded, these A-events won't be lost. Having an AggreGate server connection is *not* a precondition for the generation of stored A-events. The disadvantage is somewhat heavier implementation and slower event handling speed.

There will be a separate log table [not yet documented] (and a file on the flash disk 236) for each *type* of stored A-event in your application. This is because different A-event types can potentially have a different set of fields and, hence, the different storage format.

Another important point to discuss: if you come from a field like access control or IT, then you may be accustomed to a certain way of using the term "event". You probably dealt with events like "access granted", "access violation", "access denied", and so on. Each one of those is considered to be a separate "event".

With AggreGate, if it comes from the same log table, then it is the same A-event. The ACE A-event below is generated on "access granted", "access violation", etc., yet it is all the same single event called the "Access Control Event (ACE)". It is the *event description* that differentiates each ACE A-event *instance*. Keep this in mind when reviewing the code added in step 6.

The steps

Notice that...

1. The **TBL library** [not yet documented] is already in the project. We added it when we were creating the USER table 593.

2. **Tables.xtxt** now contains[603] the **ACE** table. Note how there are three fields:

   - The **DT** field for storing the date and time of the event.

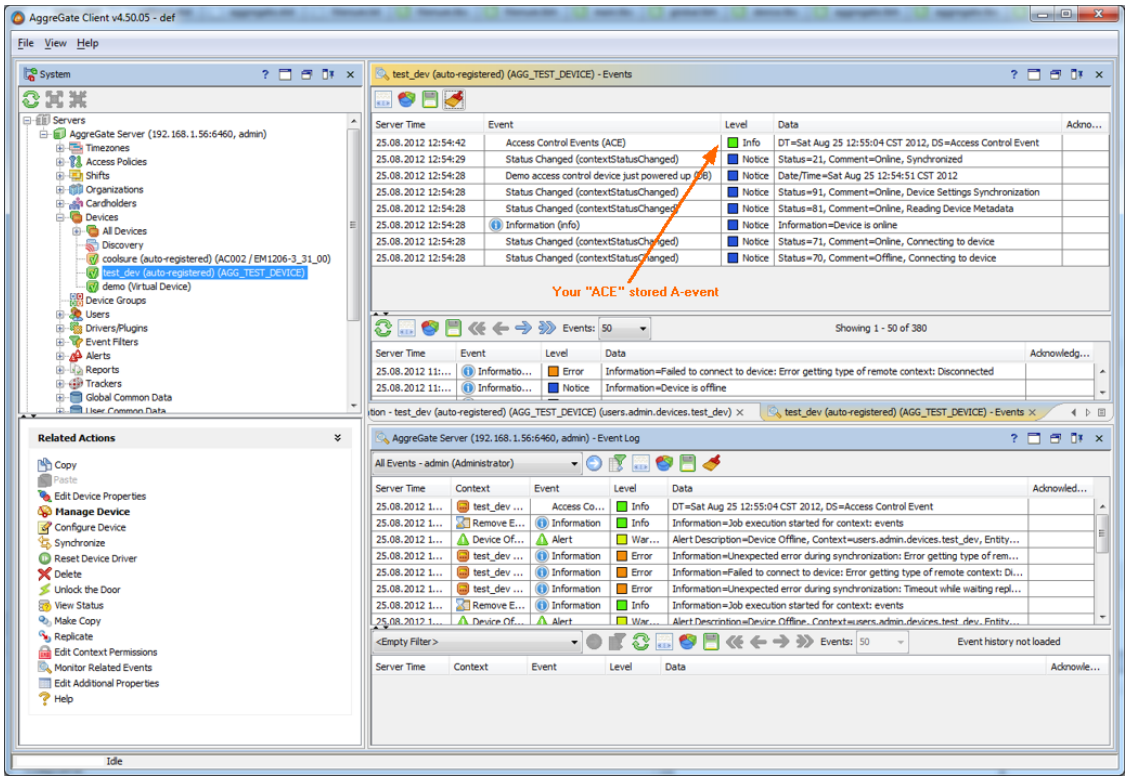
   - The **AEL** field for event the level. Stored A-events, like instant A-events[598], have the default event level[604] which can be overridden. To be able to override, have a special field named **AEL** (A-event level) in your log table[603]. This field must be of the byte type, with possible values from 0 to 5[606]. Once you have the **AEL** field in the table, the event level for each particular event *instance* will come from this field.

   - The **DS** field carrying the event description (like "access granted", "access violation", etc.). Notice how this field's size is only four characters -- how can we possible fit a meaningful description in four characters?.. Read on and you will know!

3. **Aggregate.xtxt** now contains[604] the **ACE** stored A-event[600]. It "links" to the **ACE** table.
4. Call to agg_proc_data_sent()[613] is in on_sock_data_sent()[489] event handler.
5. New **generate_door_event()** sub is in **device.tbs**. We call this from on_button_pressed()[234]. You don't have to be connected to the AggreGate server in order to be able to generate this event -- that's the beauty of stored A-events. Whenever you press the MD button[201], one ACE event is generated. The description it carries for now is always "ACE". This is temporary -- we will add real descriptions in the next step[605].
6. Notice how this procedure uses agg_stored_event_added()[611]. You have to use it every time you add a stored A-event. It "nudges" the AGG library, i.e. tells it that there are stored events that haven't been sent to the server yet.
7. Notice that agg_proc_stored_events()[612] is also called(). **This is completely optional** and serves to speed up the sending of the stored A-event to the AggreGate server. The AGG library already uses this procedure internally. Calling this "manually" just after doing agg_stored_event_added() may speed things up a bit, especially on big projects with heavy and complicated code.
8. Our sample code also makes use of callback_agg_convert_event_field()[617]. If you put a breakpoint there you will notice that this procedure is invoked every time you generate a stored A-event. The procedure is called separately for each even't field, except the **ACE** field. The idea is the same as for the callback_agg_convert_setting[616] call, which was discussed in Adding Setting A-variables[588]: you get a chance to convert or change a field's value before sending it to the AggreGate Server. In our code we use the call to convert short event descriptions (abbreviations, really) into comprehensible lines of text!

The result

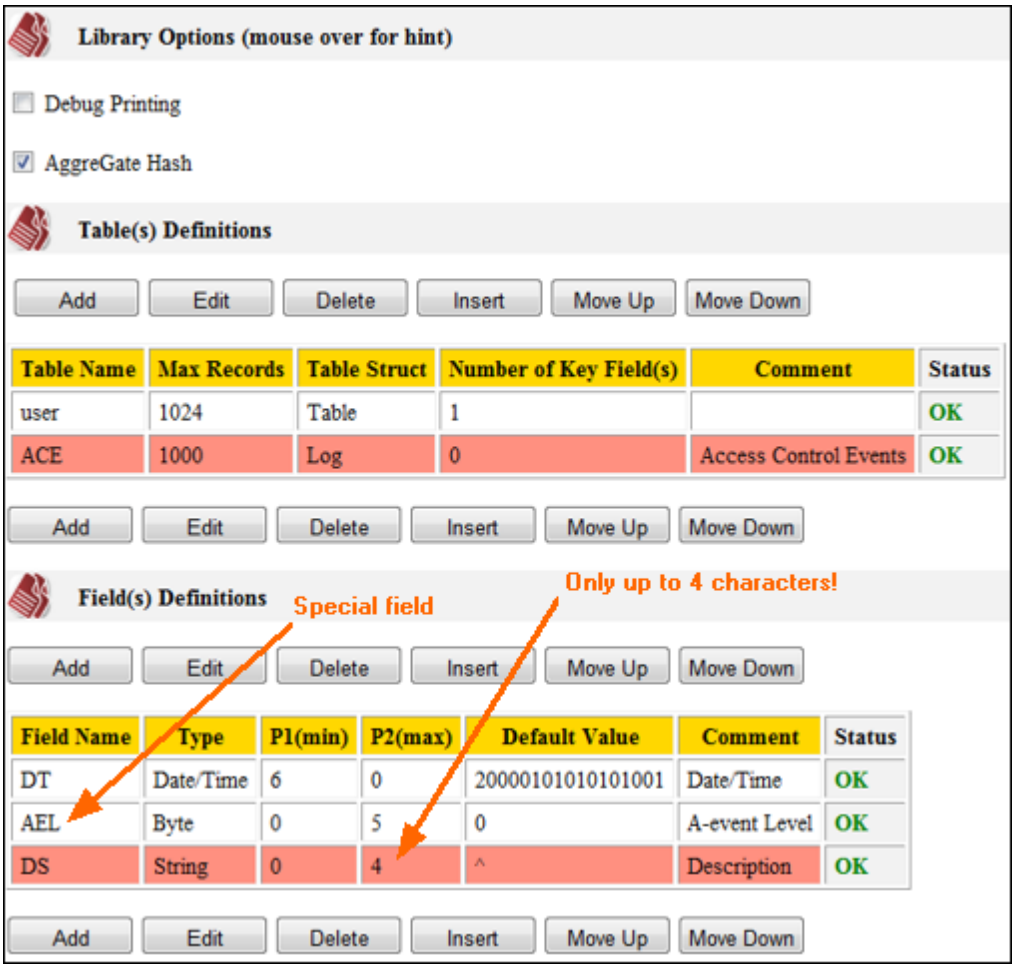
To be able to see *instances* of your new event, right-click on the device in the tree and choose **Monitor Related Events**.



Define the ACE Table

This table **configurator** [not yet documented] screenshot shows the added **ACE** log table.

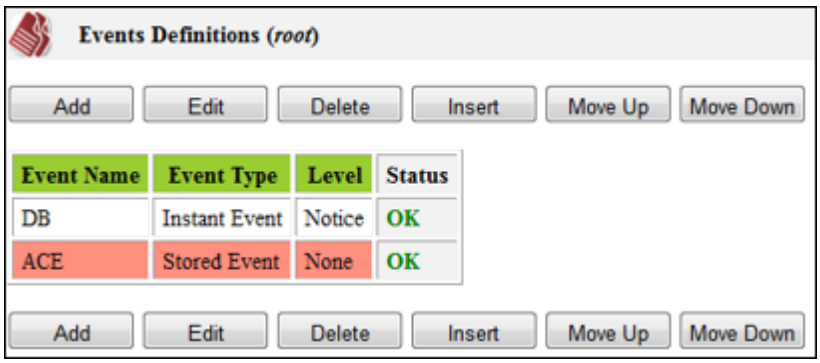
- Notice the **AEL** (A-Event level) field. Since the **ACE** log table has this field, the level of each ACE event *instance* will be taken from this field.
- Notice also that the **DS** (description) field accommodates up to 4 characters -- read about our processing technique for this field here 601.

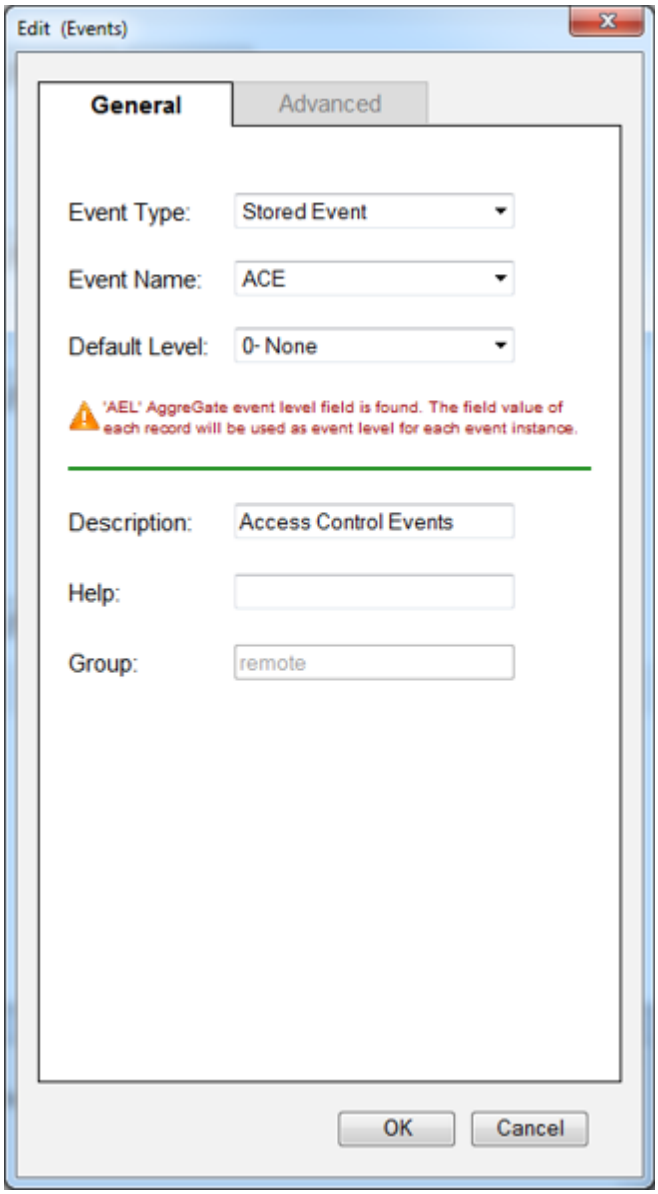


Define the ACE Stored Event

These AggreGare configurator screenshots show the added **ACE** stored A-event.

Notice the note in red under the **Default Level**. It means that the AEL (A-event level) field is found in the ACE table and the event level will be defined separately for each event *instance* in accordance with the AEL field's value.





Step 7: Gluing it All Together

This step corresponds to **test_agg_lib_7**.

And now, ladies and gentlemen, let's crown this application with some glue code. Observe the code and notice the following...

1. Handling of the **BR** 588 setting A-variable 588:

   - Every time we select new baudrate in AggreGate, callback_stg_post_set() 701 is invoked. Our code there calls **baudrate_set()**. Don't understand how this works? Read Using Pre-gets and Post-sets 683.

   - We also call **baudrate_get()** from callback_stg_pre_get() 700.

   - **Baudrate_get()** is also invoked from **setup_serial_port()**, which is called from the on_sys_init() 533.

2. Handling of the **UT** 588 setting A-variable 588: again, we just return the actual door status in callback_stg_pre_get() 700. If you may recall 585, the MD button 201

is used as a make-believe door sensor.

3. The **GE** setting A-variable now properly allows or prevents the events from generating.
4. Handling of user access codes is in the on_ser_data_arrival event handler.

Step 8: Adding Bells and Whistles

This step does not exist yet -- you can just go ahead and create it yourself.

You can enhance the access control demo in a myriad ways. We will just point out additional facilities offered by the AGG library.

Note that...

1. You can always stop the library and free up buffer memory by calling agg_stop(). When your application does this, it will get callback_agg_buff_released() invoked.
2. You can use agg_get_connection_state() to poll the AGG library for it connection state. You can then display the state on the LCD screen (if your device has it), using LEDs, or in some other way.
3. Callback_agg_error() provides a unified place to respond to errors generated by the AGG library.
4. Some hardware projects require precise timekeeping (don't we know this!). If you are not satisfied with the internal RTC of your device (such as the EM1000), or you are using a device that has no RTC (EM500) then you can always connect your own RTC chip and write your own RTC routines:
   - *Enable* (check) **Using Custom RTC** in the AggreGate configurator.
   - Add the Callback_agg_rtc_sg() procedure to your code.

En_agg_event_levels

Available event levels apply both to instant A-events and stored A-events.

0- EN_AGG_EVENT_LEVEL_NONE

1- EN_AGG_EVENT_LEVEL_NOTICE

2- EN_AGG_EVENT_LEVEL_INFO

3- EN_AGG_EVENT_LEVEL_WARNING

4- EN_AGG_EVENT_LEVEL_ERROR

5- EN_AGG_EVENT_LEVEL_FATAL

6- EN_AGG_EVENT_LEVEL_USE_DEFAULT

In the above list, the last value ("DEFAULT") can only be used as the value of the **event_level** argument of instant A-events and as the value of the **AEL** field for stored A-events.

En_agg_status_codes

Several procedures in the library utilize the en_agg_status_codes enum. This enum has the following members:

0- EN_AGG_STATUS_OK: success.

1- EN_AGG_STATUS_NOT_STARTED: agg_start() 607 was not used or failed.

2- EN_AGG_STATUS_WRONG_DESCRIPTOR: wrong descriptor 583 file data.

3- EN_AGG_STATUS_OUT_OF_SOCKETS: no free sockets available for the library to operate.

4- EN_AGG_STATUS_INVALID_INTERFACE: unsupported network interface specified.

5- EN_AGG_STATUS_INSUFFICIENT_BUFFER_SPACE: insufficient number of buffer pages available and the call to callback_agg_pre_buffrq() 614 failed to cure the problem.

6- EN_AGG_STATUS_CONNECTION_LOST: lost connection to the AggreGate server.

7- UNABLE TO CONNECT: unable to connect to the AggreGate server.

8- EN_AGG_STATUS_SETTING_ERROR: a problem with a setting A-variable 588.

9- EN_AGG_STATUS_TABLE_ERROR: a problem with a table A-variable 593.

10- EN_AGG_STATUS_FUNCTION_ERROR: a problem with an A-function 596.

11- EN_AGG_STATUS_ITEM_NOT_FOUND: unknown item specified.

Library Procedures

In this section:

Agg_start()

Description: API procedure, starts the AGG library, parses the configuration file 583, prepares the library for operation.

Syntax: **function agg_start(**interface **as pl_sock_interfaces, byref** owner_name **as string, byref** device_name **as string, byref** password **as string, byref** agg_server_ip **as string,** agg_server_port **as word,** agg_server_tout **as word,** auto_reg **as no_yes) as en_agg_status_codes**

Returns: One of the en_agg_status_codes 606

See Also: The Embryo 587

Part	Description
interface	The network interface through which the library will be connecting to the AggreGate server. The list of available interfaces is platform-dependent 138 (look for the Enum pl_sock_interfaces topic inside your platform's documentation).
owner_name	Must match your account on the AggreGate server ("admin" by default).
device_name	The name under which the device will appear in the list of devices under your account.
password	The password your device will use to login to the AggreGate server.
agg_server_ip	The IP address of the AggreGate server.
agg_server_port	The listening port on the AggreGate server (6480 by default).
agg_server_tout	Maximum waiting time for the AggreGate server interactions. The recommended value is 600.
auto_reg	Whether or not the device will be allowed to register on the AggreGate server if the server does not already have this device under the specified **owner_name** account

Details

MUST be called first, before any other procedure in this library is invoked, or EN_AGG_STATUS_NOT_STARTED 606 will be returned by every other procedure you call.

Agg_stop()

Description: API procedure, stops the AGG library causing it to release occupied buffers.

Syntax: **declare sub agg_stop()**

Returns: ---

See Also: Adding Bells and Whistles 606

callback_agg_buff_released() 615

Details

Agg_get_connection_state()

Description: API procedure, returns the current state of the device's connection to the AggreGate server as well as the synchronization status.

Syntax: **sub agg_get_connection_state(byref** link_state **as en_agg_link_states, byref** sock_state **as pl_sock_state_simple)**

Returns: TCP connection state and synchronization status

See Also: Adding Bells and Whistles 606

Part	Description
link_state	Indirectly returns the synchronization status: 0- EN_AGG_LINK_STATE_IDLE: no connection to the AggreGate server. 1- EN_AGG_LINK_STATE_CONNECTING: connecting now. 2- EN_AGG_LINK_STATE_SYNCRONIZING: synchronizing now. 3- EN_AGG_LINK_STATE_DISCONNECTED: disconnected (by the server). 4- EN_AGG_LINK_STATE_ESTABLISHED: connected and fully synchronized.
sock_state	Indirectly returns the state of the TCP link to the server. This is directly from the sock.statesimple 505 property of the socket used for connecting to the AggreGate server.

Details

Agg_record_decode()

Description: API procedure, extracts the value of the specified argument from the encoded string containing the values of all arguments for an A-function.

Syntax: **function agg_record_decode(byref** encode_string **as string,** field_index **as byte) as string**

Returns: The specified argument's value

See Also: Adding A-functions 609

Part	Description
encode_string	The argument string sent by the server.
field_index	The index (counting from 0) of the argument that needs to be extracted.

Details

Only works for argument strings containing simple single-value arguments. Need to pass complex arguments containing entire tables? Write your own decoder. String format is described here:

http://aggregate.tibbo.com/docs/en/ap_data_tables_encoding.htm

Agg_record_encode()

Description: API procedure, appends the argument's value to the encoded A-function's or A-event's argument string in preparation for sending this string to the AggreGate server.

Syntax: **sub agg_record_encode(byref** encode_string **as string, byref** field_value **as string)**

Returns: Updated encode_string

See Also: Adding A-functions 596, Firing Instant A-events 598

Part	Description
encode_string	The argument string being prepared.
field_index	New value to be appended to the string.

Details

Should be called repeatedly if the A-function has multiple arguments.

Can only handle simple single-value arguments. Need to pass complex arguments containing entire tables? Write your own encoder. String format is described here:

http://aggregate.tibbo.com/docs/en/ap_data_tables_encoding.htm

Agg_fire_instant_event()

Description: API procedure, generates an instant A-event and sends it directly to the AggreGate server.

Syntax: **sub agg_fire_instant_event(byref** context_name **as**

string, byref event_name **as string, byref** event_string **as string,** event_level **as en_agg_event_levels)**

Returns: ---

See Also: Firing Instant A-events 598

Part	Description
context_name	Always set this to "root".
event_name	Name of the instant A-event, must match one of the A-events you've defined through the configurator 583.
event_string	The encoded string containing the values of the A-event's arguments. Use agg_record_encode() 610 to prepare. Use agg_record_encode() 610 to prepare.
event_level	A-event level as defined by the en_agg_event_levels 606 enum. If you set this to EN_AGG_EVENT_LEVEL_USE_DEFAULT then the event level defined through the configurator 600 will be used. Specifying any other event level will override the default level specified in the configurator.

Details

Agg_stored_event_added()

Description: API procedure, must be called every time a stored A-event is added to the log file.

Syntax: **sub agg_stored_event_added(byref** event_name **as string)**

Returns: ---

See Also: Handling Stored A-events 601

Part	Description
event_name	Name of the A-event, must match the event that's been added.

Details

Agg_proc_stored_events()

Description: API procedure, nudges the AGG library to process stored A-events.

Syntax: **sub agg_proc_stored_events()**

Returns: ---

See Also: Handling Stored A-events 601

Details

The use of this procedure is completely optional.

Agg_proc_timer()

Function: Event procedure, call it from the on_sys_timer() 533 event handler.

Syntax: **sub agg_proc_timer()**

Returns: ---

See Also: The Embryo 587

Details

The AGG library expects that the system timer 528 is running at half-second intervals.

Agg_proc_data()

Function: Event procedure, call it from the on_sock_data_arrival() 489 event handler.

Syntax: **sub agg_proc_data()**

Returns: ---

See Also: The Embryo 587

Details

Agg_proc_sock_event()

Function: Event procedure, call it from the on_sock_event() 490 event handler.

Syntax: **sub agg_proc_sock_event(**sock_state **as pl_sock_state,** sock_state_simple **as pl_sock_state_simple)**

Returns: ---

See Also: The Embryo 587

Details

This procedure's arguments are the same as for on_sock_event() 490.

Agg_proc_data_sent()

Function: Event procedure, call it from the on_sock_data_sent() 489 event handler.

Syntax: **sub agg_proc_data_sent()**

Returns: ---

See Also: Handling Stored A-events 601

Details

Callback_agg_get_firmware_version()

Description: Callback procedure, requests the version string for the current application. Procedure body has to be created elsewhere in the project (externally with respect to the library).

Syntax: **function callback_agg_get_firmware_version() as string**

Returns: Version string

See Also: The Embryo 587

Details

This isn't required, but customarily we enclose version strings in braces ("{}"). For example: **{agg_demo.1.01.01}**.

Callback_agg_device_function()

Description: Callback procedure, invoked when the device needs to execute an AggreGate function. Procedure body has to be created elsewhere in the project (externally with respect to the library).

Syntax: **function callback_agg_device_function(byref** function_name **as string ,byref** context_name **as string, byref** function_input **as string, byref** function_output **as string) as ok_ng**

Returns: ---

See Also: Firing Instant A-events 598

Part	Description
function_name	AggreGate function name.
context_name	This library only supports the root context, so context_name should be ignored.
function_input	The encoded string containing the values of all arguments of the AggreGate function. Use agg_record_decode() 609 to process.
function_output	The encoded string containing the values of the function's output. Use agg_record_encode() 610 to process.

Details

Callback_agg_synchronized()

Description: Callback procedure, informs of the completion of the synchronization process between the device and the AggreGate server. Procedure body has to be created elsewhere in the project (externally with respect to the library).

Syntax: **sub callback_agg_synchronized()**

Returns: Version string

See Also: Firing Instant A-events 598

Details

Callback_agg_pre_buffrq()

Description: Callback procedure, informs of the insufficient number of free buffer pages available for use by the library. Procedure body has to be created elsewhere in the project (externally with respect to the library).

Syntax: **sub callback_agg_pre_buffrq(**required_buff_pages **as byte)**

Returns: ---

See Also: The Embryo 587

Part	Description
required_buffer_pages	The number of additional buffer pages that the AGG library needs to operate. Your application must free up at least this number of buffer pages within callback_agg_pre_buffrq() or agg_start() 607 will fail with the AGG_STATUS_INSUFFICIENT_BUFFER_SPACE code 606.

Details

This procedure will be called only if there are not enough buffer pages available.

Callback_agg_buff_released()

Description: Callback procedure, informs of the release of buffer pages by the library. Procedure body has to be created elsewhere in the project (externally with respect to the library).

Syntax: **sub callback_agg_buff_released()**

Returns: ---

See Also: Adding Bells and Whistles 606

agg_stop() 608

Details

The number of free buffer pages can be checked through the sys.freebuffpages 531 R/O property.

Callback_agg_error()

Description: Callback procedure, informs of an error or condition detected within the library. Procedure body has to be created elsewhere in the project (externally with respect to the library).

Syntax: **sub callback_agg_error(**failure_code **as en_agg_status_codes, byref** error_item **as string)**

Returns: ---

See Also: ---

Part	Description
failure_code	One of the en_agg_status_codes 606
error_item	Name of the item in question, for example, the name of a setting A-variable.

Details

Callback_agg_convert_setting()

Description: Callback procedure, invoked every time a setting A-variable is being read or written. Provides an opportunity to convert the value before writing to a setting or after reading from a setting.

Procedure body has to be created elsewhere in the project (externally with respect to the library).

Syntax: **sub callback_agg_convert_setting(byref** setting_name **as string,** index **as byte, byref** setting_value **as string,** op **as en_agg_rdwr)**

Returns: ---

See Also: Adding Setting A-variables 588

Callback_agg_convert_event_field() 617

Part	Description
setting_name	Name of the setting being read or written.
index	Setting index (zero for single-value settings).
setting_value	Value to be converted. This argument is also used for returning the converted value: take setting_value, modify as needed, and store back.
op	EN_AGG_GET- reading from the setting EN_AGG_SET- writing to the setting

Details

This procedure allows to store the value in a format or type that is different from the way the same is perceived in AggreGate.

Here is a simple example: Supposing, there is a setting that stores device temperature. The setting data originates from a temperature measuring IC, which outputs a byte value. Seven most significant bits are degrees. The least significant bit represent a 0.5-degree step. The output of 0 corresponds to -40 degrees C. Therefore, the output of 255 corresponds to +87.5 degrees C.

Now, the operator in AggreGare would surely prefer to see "natural" temperature values like -40 and +87.5, rather than 0 and 255. This procedure can provide a place to do the conversion.

The same possibility exists for stored event's fields (see Callback_agg_convert_event_field() 617) but not for table A-variables 593. Tables

must be stored in exactly the same way that the AggreGate server "believes" them to be stored. This is because table synchronization is based on hash strings.

Callback_agg_convert_event_field()

Description: Callback procedure, invoked every time a stored A-event is being extracted from the log and sent to the server. Provides an opportunity to change the value or type of A-event fields before sending the stored A-event to the server.

Procedure body has to be created elsewhere in the project (externally with respect to the library).

Syntax: **declare sub callback_agg_convert_event_field(byref** table_name **as string, byref** field_name **as string, byref** field_value **as string)**

Returns: ---

See Also: Handling Stored A-events 601

Callback_agg_convert_setting() 616

Part	Description
table_name	The name of the stored A-event (and the log table which stores it).
field_name	The name of the field.
field_value	The value of the field.

Details

For every stored A-event being sent to the server, this procedure will be called separately for each event's field.

Type/value conversion possibility also exists for setting A-variables (see callback_agg_convert_setting() 616) but not for table A-variables 593. Tables must be stored in exactly the same way that the AggreGate server "believes" them to be stored. This is because table synchronization is based on hash strings.

Callback_agg_rtc_sg()

Description: Callback procedure, invoked when the library needs to get or set the device's date and time. Procedure body has to be created elsewhere in the project (externally with respect to the library).

Syntax: **sub callback_agg_rtc_sg(byref** wdaycount **as word, byref** wmincount **as word, byref** bsec **as byte, byref** wmilsec **as word,** op **as en_agg_rdwr)**

Returns: ---

See Also: ---

Part	Description
wdaycount	Number of elapsed days.
wmincount	Number of elapsed minutes.
bsec	Number of elapsed seconds.
wmilsec	Number of elapsed milliseconds.
op	EN_AGG_GET or EN_AGG_SET

Details

Necessary only when **Use custom RTC** option is *enabled* in the configurator 583.

DHCP Library

The DHCP library implements DHCP client on Ethernet (net. 358) and Wi-Fi (wln. 536) network interfaces. The DHCP protocol is used for automatic configuration of your device's IP address, netmask, and gateway IP address. More on the protocol here: http://en.wikipedia.org/wiki/Dhcp.

The library is capable of running DHCP clients on both network interfaces concurrently. The library is event-based and non-blocking -- it quietly runs in the background and takes a minimal amount of CPU time.

Library Info

Supported platforms: Any platform with at least the Ethernet (net. 358) or Wi-Fi (wln. 536) interface except for EM202.

Files to include: Dhcp.tbs, dhcp.tbh (from *current_library_set*\dhcp\trunk\) .

Dependencies: SOCK 664 library;

You may also need WLN library 703

API procedures:

Use API procedures 576 to interact with

dhcp_get_info() -- returns library-specific information (such as required buffer space).

dhcp_start() 636 -- starts the DHCP client on the specified

the library.	network interface. dhcp_stop() 637 -- stops the DHCP client on the specified network interface.
Event procedures:	dhcp_proc_timer() 637 -- call this from the on_sys_timer() 533 event handler.
Call event procedures 576 from corresponding event handlers, as described here 619.	dhcp_proc_data() 638 -- call this from the on_sock_data_arrival() 489 event handler.
Callback procedures:	callback_dhcp_ok() 638 -- called when the library completes successfully on one of the interfaces.
Implement the bodies of callback procedures 576 elsewhere in your project.	callback_dhcp_failure() 639 -- called when the library fails on one of the interfaces after DHCP_MAX_RETRIES 634. callback_dhcp_pre_clear_ip() 639 -- called when the library is about to set the interface's IP to 0.0.0.0. callback_dchp_pre_buffrq() 640 -- called when the library needs to allocate buffer space and the required space is not available. callback_dhcp_buff_released() 640 -- called when the library no longer needs buffers and released them.
Required buffer space:	6 buffer pages. These are utilized when needed and released when the DHCP client completes its job. The amount of buffer space needed is constant and does not depend on the number of network interfaces involved.

Step-by-step Usage Instructions

Minimal steps

1. Make sure you have the SOCK 664 library in your project.
2. Add 577 **dhcp.tbs** and **dhcp.tbh** files to your project (they are in *current_library_set*\dhcp\trunk).
3. Add #define DHCP_DEBUG_PRINT_1 634 to the defines section of the global.tbh file of your project. This way you will "see what's going on". Don't forget to remove this later, after you've made sure that the library operates as expected.
4. Add **include "dhcp\trunk\dhcp.tbh"** to the includes section of the global.tbh file.
5. Add dhcp_proc_timer() 637 to the on_sys_timer() 533 event handler code (see "Further considerations" below).
6. Add dhcp_proc_data() 638 to the on_sock_data_arrival() 489 event handler code.
7. Create empty callback function 576 bodies (presumably in the device.tbs): callback_dhcp_ok() 638, callback_dhcp_failure() 639, callback_dhcp_pre_clear_ip() 639, callback_dhcp_pre_buffrq() 640, and callback_dhcp_buff_released() 640. Hint: copy from declarations in the dhcp.tbh or from our code example 622.
8. Call dhcp_start() 636 from somewhere -- once for every interface you want to

use DHCP on. How you will do this depends on your program's logic. The no-brainer decision is to call from the on_sys_init() 533 event handler. In the simplest case, leave the suggested_ip and host_name arguments empty. Note that dhcp_start() may fail, so it is wise to check the returned status code.

9. Implement meaningful code for the callback_dhcp_ok() 638 function. Typically, once the DHCP process completes, you should update the IP, netmask, and gateway IP of the corresponding interface.

All of the above is illustrated in the Code Examples 622.

Additional recommended steps

1. Store the obtained IP address in the EEPROM and provide it with the dhcp_start() 636. The STG 668 library comes handy for this -- see our example 631. This way your device will be (mostly) able to keep using the same IP.
2. Optionally provide the device (host) name with the dhcp_start() 636. Some DHCP servers are linked to DNS servers and sending the host name will automatically register your device with the DNS server. By default, device names are disabled. To enable names, add a like like this to the global.tbh: #define DHCP_MAX_HOST_NAME_LEN 32 634 (this will allow names of up to 32 chars).
3. The use of dhcp_stop() 637 is optional. In fact, under the normal circumstances you should let the DHCP client run continuously. This is because the library needs to take care of the "lease expiration" and renew the lease at proper times (see Operation Details 621). However, you may choose to terminate the DHCP client on a particular interface if the DHCP configuration fails (too many times).
4. The use of callback_dhcp_failure() 639 is optional. You may leave the procedure empty. Alternatively, once DHCP fails (too many times), you can set an alternative IP, netmask, and gateway IP for the corresponding network interface of your device.
5. Callback_dhcp_pre_buffrq() 640 is only called when there isn't enough free buffer space for the DHCP library's needs. Leave the procedure empty if you are supremely confident that there is always enough free buffer space. Alternatively, free up the required space when asked for it, or the DHCP will fail.
6. Callback_dhcp_buff_released() 640 is called when the DHCP library no longer needs buffer space. A smart application would seize on the opportunity to redistribute newly available buffer pages to other needy parts of your system. Then, again, you can just ignore this chance.

See Code Examples 622 for illustration of the points above.

Not enough buffer space to run DHCP? Take some space away from your socket buffers 422. Without properly configured IP, sockets are of no use, right? So, deallocate socket buffer space in the callback_dhcp_pre_buffalloc() and give buffers back to sockets in the callback_dhcp_buff_released().

Further considerations

The DHCP library expects the dhcp_proc_timer() 637 to be called at 0.5 sec intervals. This is the default value which can be changed through the sys.onsystimerperiod 533 property. If your project needs to have the on_sys_timer() 533 event at a different rate, please make sure that dhcp_proc_timer() is still called 2 times/second. For example, if the on_sys_timer() is set to trigger 4 times/second,

you need to add "divider" code that only calls dhcp_proc_timer() on every second invocation of the on_sys_timer().

Operation Details

The DHCP client, once started on a particular interface with the dhcp_start() 636, will run continuously until stopped through the dhcp_stop() 637. The operation of the library may be observed in the debug mode 27 by adding #define DHCP_DEBUG_PRINT 1 634 to the defines section of the global.tbh file of your project. A wealth of status information will then be printed in the console pane as the DHCP library operates.

Once dhcp_start() is called successfully (and it can fail -- observe returned code!), the library attempts to obtain the IP address, netmask, and the gateway IP address from the DHCP server. If the process fails, the library will retry. A total of DHCP_MAX_RETRIES 634 attempts are performed (with randomized waiting times between retries, but not exceeding DHCP_MAX_RETRY_DELAY 634 seconds), after which callback_dhcp_failure() 639 is called and the library goes into the waiting period of DHCP_POST_FAIL_DELAY 634 seconds. When the period is over, another batch of DHCP_MAX_RETRIES attempts is performed, and so on ad infinitum, until the dhcp_stop() 637 is called or DHCP configuration succeeds.

DHCP configuration requires that the initial IP address of the client device be 0.0.0.0. If the IP address of the specified interface is different, the library will set it to 0.0.0.0. Changing the IP address on a particular interface requires all socket connections operating on this interface to be closed first. The DHCP library will take care of that by calling sock.discard 478 for each involved socket. Before doing that, the DHCP library will invoke callback_dhcp_pre_clear_ip() 639. This way, your application may "prepare" for what is coming. For example, you can add code for gracefully closing TCP connections (instead of letting the DHCP library to do a rude discard).

A buffer space of 6 pages is required while a batch of DHCP_MAX_RETRIES attempts is performed on either network interface. The library will check if the required buffer space is available and call callback_dhcp_pre_buffrq() 640 if more space is needed. Once the batch of retry attempts finishes (either successfully or unsuccessfully), the buffer space is no longer needed and is released by the library. The callback_dhcp_buff_released() 640 is called to notify your system of the fact. Your system can then check the total number of free pages available (now that the DHCP library released some pages) through the sys.freebuffpages 531 R/O property.

The library can run DHCP configuration on both the Ethernet (net. 358) and Wi-Fi (wln. 536) interfaces concurrently. This does not increase buffer space requirements. The library will appropriate buffer pages when needed by either of the interfaces and release buffer pages when neither network interface requires them any longer.

Once the DHCP configuration completes successfully on a particular network interface, the callback_dhcp_ok() 638 is invoked. You can then program the new IP (and the netmask, and the gateway IP) into the net. 358 or wln. 536 object. Doing so is only possible when no sockets are active on the network interface whose properties you are about to change.

After the successful DHCP configuration, the library goes into counting lease time. Once 90% of the lease time passes, the library will attempt to renew the lease. Buffer pages will be needed for this again. The IP address does not have to be reset to 0.0.0.0 for the lease renewal, so callback_dhcp_pre_clear_ip() will not be called, either.

Note that even in case of the lease renewal the IP address offered by the DHCP server may change. Don't forget to take care of this! Actually, it may be easier to just reboot the device than deal with all the implications of the changed IP. Code

Examples topic shows how to deal with this.

The library also monitors the link state of network interfaces. If the Ethernet cable is unplugged and then re-plugged (this is monitored through the net.linstate R/O property), the DHCP process (if enabled) will restart automatically for the Ethernet interface. Similarly, if the access point association is lost and then reestablished (wln.associationstate), the DHCP process (if enabled) will restart automatically for the Wi-Fi interface.

Code Examples

Projects in the Code Examples section are published on our website under the name "test_wln_lib".

In this section:

Step 1: Code Example for the Ethernet Interface

This and other projects in the Code Examples section are published on our website under the name "test_dhcp_lib".

Here is a simple example of using DHCP on the Ethernet (net.) interface. Notice how the callback_dhcp_ok() procedure is implemented! This is a good illustration of the protracted explanation in the Operations Details topic (not that we think you will read it all).

Note also the use of the NET_AVAILABLE define. This comes from the platform. If the Ethernet interface is available on the currently selected platform, then NET_AVAILABLE is "1". Similarly, if the Wi-Fi interface is available, then WLN_AVAILABLE is "1". Using such defines you can create applications that will compile on any platform, as long as this platform has at least the Ethernet, or the Wi-Fi interface.

Here is a sample debug output we've got after running the code:

```
DHCP(net)> ---START---
DHCP(net)> TX discovery message
DHCP(net)> RX offer message
DHCP(net)> TX request message
DHCP(net)> RX confirmation message
DHCP(net)> ---OK(ip: 192.168.1.246, gateway: 192.168.1.1, netmask:
255.255.255.0, lease: 529200 sec.)---
```

- If you are compiling on one of the "W" platforms (that is, platforms with Wi-Fi), then you need to disable the wln. object, or this project won't compile. Go Project -> Settings -> Customize and set Wi-Fi (wln.) object to "Disabled".

And here is the code itself...

global.tbh:

```
'DEFINES-------------------------------------------------------------
#define DHCP_DEBUG_PRINT 1 'to see debug output by the DHCP lib

'INCLUDES------------------------------------------------------------
include "sock\trunk\sock.tbh" 'this lib is necessary for the DHCP lib's
operation
include "dhcp\trunk\dhcp.tbh"
```

main.tbs:

```
include "global.tbh"

'====================================================================
sub on_sys_init()
      #if NET_AVAILABLE
            dhcp_start(PL_SOCK_INTERFACE_NET,"","")
      #endif
end sub

'--------------------------------------------------------------------
sub on_sys_timer()
      dhcp_proc_timer()
end sub

'--------------------------------------------------------------------
sub on_sock_data_arrival()
      dhcp_proc_data()
end sub
```

device.tbs:

```
include "global.tbh"

'====================================================================
sub callback_dhcp_ok(renew as no_yes, interface as pl_sock_interfaces, byref
ip as string, byref gateway_ip as string, byref netmask as string,
lease_time as dword)
      dim f as byte

      #if NET_AVAILABLE
            if interface=PL_SOCK_INTERFACE_NET then
                  if renew=YES and net.ip<>ip then
                        'this is a lease renewal and the DHCP server has
issues new IP
                        'it is better to reboot than deal with the
implications of the changed IP
                        sys.reboot
                  end if

                  if net.ip<>ip then
                        'no need to close sockets -- this is definitely
NOT a renewal an the DHCP lib
```

```
                            'has already closed all socket connections on
this interface prior to setting
                            'the IP to 0.0.0.0
                            net.ip=ip
                            net.gatewayip=gateway_ip
                            net.netmask=netmask
                        end if
                end if
        #endif
end sub

'--------------------------------------------------------------------
sub callback_dhcp_failure(interface as pl_sock_interfaces,failure_code as
en_dhcp_status_codes)
end sub

'--------------------------------------------------------------------
sub callback_dhcp_pre_clear_ip(interface as pl_sock_interfaces)
end sub

'--------------------------------------------------------------------
sub callback_dhcp_pre_buffrq(required_buff_pages as byte)
end sub

'--------------------------------------------------------------------
sub callback_dhcp_buff_released()
end sub
```

Step 2: Code Example for the Wi-Fi Interface

This and other projects in the Code Examples section are published on our website under the name "test_dhcp_lib".

Now let's add DHCP for the Wi-Fi interface (wln.[536]) to the previous sample code[622] .

In order to even get to the DHCP stage, the Wi-Fi interface must first associate with the access point. We use our own handy WLN library[703] for that. Notice the use of the WLN_AVAILABLE define. This example does not address Wi-Fi WPA security, see Trying WPA[711] for inspiration.

Sample debug output follows. Notice some error messages there. This is because our device, although working with two interfaces, was actually connected to the same LAN with the same DHCP server. DHCP messages are sent as broadcasts and both interfaces will get the same message. This results in one interface getting messages intended for the other interface and rejecting them as erroneous. This does not affect the operation of the library.

```
DHCP(net)> ---START---
DHCP(wln)> ---START---
DHCP(net)> TX discovery message
DHCP(net)> RX offer message
DHCP(net)> INFO: RX unexpected message (wasn't expecting anything at the
moment)
DHCP(net)> INFO: RX unexpected, invalid, or unrelated message (it was
discarded)
DHCP(net)> TX request message
```

```
DHCP(net)> RX confirmation message
DHCP(net)> ---OK(ip: 192.168.1.215, gateway: 192.168.1.1, netmask:
255.255.255.0, lease: 529200 sec.)---
DHCP(wln)> Wi-Fi interface associated with the AP -- DHCP (re)started
DHCP(wln)> TX discovery message
DHCP(wln)> RX offer message
DHCP(wln)> INFO: RX unexpected message (wasn't expecting anything at the
moment)
DHCP(wln)> INFO: RX unexpected, invalid, or unrelated message (it was
discarded)
DHCP(wln)> TX request message
DHCP(wln)> RX confirmation message
DHCP(wln)> ---OK(ip: 192.168.1.216, gateway: 192.168.1.1, netmask:
255.255.255.0, lease: 529200 sec.)---
```

- If you disabled the wln. 536 object in the previous step 622, the you need to enable it now. Plus, you need to test on one of the "W" platforms (that is, platforms with Wi-Fi). Go Project -> Settings -> Customize and set Wi-Fi (wln.) object to "Enabled".

And now the code (plus don't forget to add ga1000fw.bin into your project)...

global.tbh:

```
'DEFINES---------------------------------------------------------------
#define DHCP_DEBUG_PRINT 1 'to see debug output by the DHCP lib

'this is for WLN library
#if PLATFORM_ID=EM500 or PLATFORM_ID=EM500W
      #define WLN_RESET_MODE 1 'there will be no dedicated reset, and all
other lines are fixed
#elif PLATFORM_ID=EM1206 or PLATFORM_ID=EM1206W
      #define WLN_CLK PL_IO_NUM_14
      #define WLN_CS PL_IO_NUM_15
      #define WLN_DI PL_IO_NUM_12
      #define WLN_DO PL_IO_NUM_13
      #define WLN_RST PL_IO_NUM_11
#else
      'EM1000, NB1010,...
      #define WLN_CLK PL_IO_NUM_53
      #define WLN_CS PL_IO_NUM_49
      #define WLN_DI PL_IO_NUM_52
      #define WLN_DO PL_IO_NUM_50
      #define WLN_RST PL_IO_NUM_51
#endif

'INCLUDES--------------------------------------------------------------
include "sock\trunk\sock.tbh" 'this lib is necessary for the DHCP lib's
operation
include "wln\trunk\wln.tbh" 'this lib is necessary for the DHCP on Wi-Fi
interface
include "dhcp\trunk\dhcp.tbh"
```

main.tbs:

```
include "global.tbh"

'====================================================================
sub on_sys_init()

      #if NET_AVAILABLE
            dhcp_start(PL_SOCK_INTERFACE_NET,"","")
      #endif

      #if WLN_AVAILABLE
            wln_start("TIBB1",WLN_SECURITY_MODE_WEP64,"1234567890",
PL_WLN_DOMAIN_FCC) '<-- set what you need here, see WLN lib docs
            dhcp_start(PL_SOCK_INTERFACE_WLN,"","")
      #endif
end sub

'--------------------------------------------------------------------
sub on_sys_timer()
      dhcp_proc_timer()

      #if WLN_AVAILABLE
            wln_proc_timer() 'see WLN lib docs
      #endif
end sub

'--------------------------------------------------------------------
sub on_sock_data_arrival()
      dhcp_proc_data()

      #if WLN_AVAILABLE
            wln_proc_data() 'see WLN lib docs
      #endif
end sub

'--------------------------------------------------------------------
#if WLN_AVAILABLE
sub on_wln_task_complete(completed_task as pl_wln_tasks)
      wln_proc_task_complete(completed_task) 'see WLN lib docs
end sub
#endif

'--------------------------------------------------------------------
#if WLN_AVAILABLE
sub on_wln_event(wln_event as pl_wln_events)
      wln_proc_event(wln_event)
end sub
#endif
```

device.tbs:

```
include "global.tbh"

'====================================================================
sub callback_dhcp_ok(renew as no_yes, interface as pl_sock_interfaces, byref
ip as string, byref gateway_ip as string, byref netmask as string,
lease_time as dword)
      dim f as byte

      #if NET_AVAILABLE
```

```
            if interface=PL_SOCK_INTERFACE_NET then
                  if renew=YES and net.ip<>ip then
                        'this is a lease renewal and the DHCP server has
issues new IP
                        'it is better to reboot than deal with the
implications of the changed IP
                        sys.reboot
                  end if

                  if net.ip<>ip then
                        'no need to close sockets -- this is definitely
NOT a renewal an the DHCP lib
                        'has already closed all socket connections on
this interface prior to setting
                        'the IP to 0.0.0.0
                        net.ip=ip
                        net.gatewayip=gateway_ip
                        net.netmask=netmask
                  end if
            end if
      #endif

      #if WLN_AVAILABLE
            if interface=PL_SOCK_INTERFACE_WLN then
                  if renew=YES and wln.ip<>ip then
                        'this is a lease renewal and the DHCP server has
issues new IP
                        'it is better to reboot than deal with the
implications of the changed IP
                        sys.reboot
                  end if

                  if wln.ip<>ip then
                        'no need to close sockets -- this is definitely
NOT a renewal an the DHCP lib
                        'has already closed all socket connections on
this interface prior to setting
                        'the IP to 0.0.0.0
                        wln.ip=ip
                        wln.gatewayip=gateway_ip
                        wln.netmask=netmask
                  end if
            end if
      #endif
end sub

'-------------------------------------------------------------------
sub callback_dhcp_failure(interface as pl_sock_interfaces,failure_code as
en_dhcp_status_codes)
end sub

'-------------------------------------------------------------------
sub callback_dhcp_pre_clear_ip(interface as pl_sock_interfaces)
end sub

'-------------------------------------------------------------------
sub callback_dhcp_pre_buffrq(required_buff_pages as byte)
end sub

'-------------------------------------------------------------------
sub callback_dhcp_buff_released()
```

```
end sub

'--------------------------------------------------------------------
#if WLN_AVAILABLE
sub callback_wln_ok()
end sub
#endif

'--------------------------------------------------------------------
#if WLN_AVAILABLE
sub callback_wln_failure(wln_state as en_wln_status_codes)
end sub
#endif

'--------------------------------------------------------------------
#if WLN_AVAILABLE
sub callback_wln_pre_buffrq(required_buff_pages as byte)
end sub
#endif

'--------------------------------------------------------------------
#if WLN_AVAILABLE
sub callback_wln_rescan_result(current_rssi as byte, scan_rssi as byte,
different_ap as no_yes)
end sub
#endif
```

Step 3: Adding Bells and Whistles

This and other projects in the Code Examples section are published on our website under the name "test_dhcp_lib".

Writing quality code means anticipating things to go wrong and not as planned. Here are some ideas on fortifying your code:

Check the result of dhcp_start()

This one returns one of the en_dhcp_status_codes() 635 and you should check what's returned! Decide for yourself what to do in each particular case or simply halt the execution if there is an error:

```
...
if dhcp_start(PL_SOCK_INTERFACE_NET)<>DHCP_STATUS_OK then sys.halt 'do the
same for the Wi-Fi interface
...
```

What to do when callback_dhcp_failure() is called

The DHCP client, once started, continues to run indefinitely. Still, at some point your application has to face the simple fact that the DHCP configuration has, indeed, failed, and it is time to do something about it. A set of alternative configuration parameters will come handy at that time. In the code below,

net_dhcp_fail_ctr and wln_dhcp_fail_ctr accumulate failures. Notice how dhcp_stop () 637 is called once the count reaches DHCP_FAIL_LIMIT:

device.tbs:

```
include "global.tbh"

'-----------------------------------------------------------------------
const DHCP_FAIL_LIMIT=1 'max number of DHCP retry "batches"
const ALT_NET_IP="192.168.1.40"
const ALT_NET_GATEWAY_IP="192.168.1.1"
const ALT_NET_NETMASK="255.255.255.0"
const ALT_WLN_IP="192.168.1.41"
const ALT_WLN_GATEWAY_IP="192.168.1.1"
const ALT_WLN_NETMASK="255.255.255.0"

'-----------------------------------------------------------------------
#if NET_AVAILABLE
      dim net_dhcp_fail_ctr as byte 'no need to init to 0 (this is automatic
for global vars)
#endif

#if WLN_AVAILABLE
      dim wln_dhcp_fail_ctr as byte 'no need to init to 0 (this is automatic
for global vars)
#endif

'=======================================================================
sub callback_dhcp_ok(renew as no_yes, interface as pl_sock_interfaces, byref
ip as string, byref gateway_ip as string, byref netmask as string,
lease_time as dword)
      dim f as byte

      #if NET_AVAILABLE
            if interface=PL_SOCK_INTERFACE_NET then
                  if renew=YES and net.ip<>ip then
                        'this is a lease renewal and the DHCP server has
issues new IP
                        'it is better to reboot than deal with the
implications of the changed IP
                        sys.reboot
                  end if

                  if net.ip<>ip then
                        'no need to close sockets -- this is definitely
NOT a renewal an the DHCP lib
                        'has already closed all socket connections on
this interface prior to setting
                        'the IP to 0.0.0.0
                        net.ip=ip
                        net.gatewayip=gateway_ip
                        net.netmask=netmask
                        net_dhcp_fail_ctr=0
                  end if
            end if
      #endif

      #if WLN_AVAILABLE
            if interface=PL_SOCK_INTERFACE_WLN then
                  if renew=YES and wln.ip<>ip then
```

```
                        'this is a lease renewal and the DHCP server has
issues new IP
                        'it is better to reboot than deal with the
implications of the changed IP
                        sys.reboot
                    end if

                    if wln.ip<>ip then
                        'no need to close sockets -- this is definitely
NOT a renewal an the DHCP lib
                        'has already closed all socket connections on
this interface prior to setting
                        'the IP to 0.0.0.0
                        wln.ip=ip
                        wln.gatewayip=gateway_ip
                        wln.netmask=netmask
                        wln_dhcp_fail_ctr=0
                    end if
                end if
        #endif
end sub

'-----------------------------------------------------------------------
sub callback_dhcp_failure(interface as pl_sock_interfaces,failure_code as
en_dhcp_status_codes)
        #if NET_AVAILABLE
                if interface=PL_SOCK_INTERFACE_NET then
                    if net_dhcp_fail_ctr>=DHCP_FAIL_LIMIT then
                        net.ip=ALT_NET_IP
                        net.gatewayip=ALT_NET_GATEWAY_IP
                        net.netmask=ALT_NET_NETMASK
                        dhcp_stop(PL_SOCK_INTERFACE_NET)
                    else
                        net_dhcp_fail_ctr=net_dhcp_fail_ctr+1
                    end if
                end if
        #endif

        #if WLN_AVAILABLE
                if interface=PL_SOCK_INTERFACE_WLN then
                    if wln_dhcp_fail_ctr>=DHCP_FAIL_LIMIT then
                        wln.ip=ALT_WLN_IP
                        wln.gatewayip=ALT_WLN_GATEWAY_IP
                        wln.netmask=ALT_WLN_NETMASK
                        dhcp_stop(PL_SOCK_INTERFACE_WLN)
                    else
                        wln_dhcp_fail_ctr=wln_dhcp_fail_ctr+1
                    end if
                end if
        #endif
end sub
...
```

What to do when callback_dhcp_pre_clear_ip() is called

This is very application-specific but we'd say, close your existing TCP connections nicely. Whatever you leave unattended will be brutally squashed by the DHCP library (using the sock.discard 478 hammer), so you may have to tend to your socket

connections in advance. Here is an example where you wait for the unsent data to be sent out on a "SOCK_DATA" socket:

```
...
      sock.num=SOCK_DATA
      while sock.txlen>0
            doevents 'optional, allows other things in your app to execute
      wend
...
```

What to do when callback_dhcp_pre_buffrq() is called

If it is called then you don't have enough buffer pages. OK, don't panic... you can always take some buffers away from somebody. Supposing you have a lot of buffers in the "SOCK_DATA" socket of your project. Sockets are useless without properly configured IP, so you could just temporary use this buffer space:

```
...
'-----------------------------------------------------------------------
sub callback_dhcp_pre_buffrq(required_buff_pages as byte)
      sock_release(SOCK_DATA) 'this deallocates buffers
end sub

'-----------------------------------------------------------------------
sub callback_dhcp_buff_released()
      sock.num=SOCK_DATA
      sock.rxbuffrq(sys.freebuffpages/2)
      sock.txbuffrq(sys.freebuffpages/2)
      sys.buffalloc
      'proceed with socket setup
end sub
...
```

Step 4: Adding More Bells and Whistles

This and other projects in the Code Examples section are published on our website under the name "test_dhcp_lib".

OK, DHCP now works but your device is probably getting a new IP each time it boots. This is because the device does not "suggest" an IP address when calling dhcp_start() 636. It is a good practice to do so -- memorize the previously obtained IP, then request the same from the DHCP server during the next boot.

Since the IP address must be preserved between boots, your device needs to store it in the EEPROM. Use the STG 668 (settings) library for the purpose (this is, by the way, a shining example of STG's immense usefulness). In the code below, two settings -- "IPN" and "IPW" -- remember the IP addresses for the Ethernet and Wi-Fi interfaces. Note that the STG library is integrated here in a rather sloppy way. Everything will work but don't take this as a golden standard for STG library use.

Suggested IP address is provided in the on_sys_init() 533 handler, when executing dhcp_start() 636, first for Ethernet, and then for Wi-Fi. When you run the code for the first time, settings won't have valid values, so stg_get() 696 will return default values of the settings, which are "0.0.0.0" (see settings.txt descriptor file). This means that no specific IP is being requested.

Notice also how device names ("tibbo_net" and "tibbo_wln") are supplied with dhcp_start().

global.tbh:

```
'DEFINES----------------------------------------------------------------
#define DHCP_DEBUG_PRINT 1 'to see debug output by the DHCP lib

'this is for WLN library
#if PLATFORM_ID=EM500 or PLATFORM_ID=EM500W
      #define WLN_RESET_MODE 1 'there will be no dedicated reset, and all
other lines are fixed
#elif PLATFORM_ID=EM1206 or PLATFORM_ID=EM1206W
      #define WLN_CLK PL_IO_NUM_14
      #define WLN_CS PL_IO_NUM_15
      #define WLN_DI PL_IO_NUM_12
      #define WLN_DO PL_IO_NUM_13
      #define WLN_RST PL_IO_NUM_11
#else
      'EM1000, NB1010,...
      #define WLN_CLK PL_IO_NUM_53
      #define WLN_CS PL_IO_NUM_49
      #define WLN_DI PL_IO_NUM_52
      #define WLN_DO PL_IO_NUM_50
      #define WLN_RST PL_IO_NUM_51
#endif

'INCLUDES---------------------------------------------------------------
include "sock\trunk\sock.tbh" 'this lib is necessary for the DHCP lib's
operation
include "settings\trunk\settings.tbh" 'this lib is for saving obtained IPs
includepp "settings.txt" 'for STG library
include "wln\trunk\wln.tbh" 'this lib is necessary for the DHCP on Wi-Fi
interface
include "dhcp\trunk\dhcp.tbh"
```

main.tbs:

```
...
sub on_sys_init()

      stg_start()

      #if NET_AVAILABLE
            dhcp_start(PL_SOCK_INTERFACE_NET,stg_get("IPN",0),"tibbo_net")
      #endif

      #if WLN_AVAILABLE
            wln_start("TIBB1",WLN_SECURITY_MODE_WEP64,"1234567890",
PL_WLN_DOMAIN_FCC) '<-- set what you need here, see WLN lib docs
            dhcp_start(PL_SOCK_INTERFACE_WLN,stg_get("IPW",0),"tibbo_wln")
      #endif
```

```
end sub
...
```

When DHCP configuration completes successfully, callback_dhcp_ok() is called and the obtained IP is saved into the corresponding setting using stg_set() 697. This also "repairs" the setting if its previous data was invalid. This way, when the device boots next time, stg_get() 696 will return the actual saved data.

device.tbs:

```
...
sub callback_dhcp_ok(renew as no_yes, interface as pl_sock_interfaces, byref
ip as string, byref gateway_ip as string, byref netmask as string,
lease_time as dword)
      dim f as byte

      #if NET_AVAILABLE
            if interface=PL_SOCK_INTERFACE_NET then
                  if renew=YES and net.ip<>ip then
                        'this is a lease renewal and the DHCP server has
issues new IP
                        'it is better to reboot than deal with the
implications of the changed IP
                        sys.reboot
                  end if

                  if net.ip<>ip then
                        'no need to close sockets -- this is definitely
NOT a renewal an the DHCP lib
                        'has already closed all socket connections on
this interface prior to setting
                        'the IP to 0.0.0.0
                        net.ip=ip
                        net.gatewayip=gateway_ip
                        net.netmask=netmask
                        net_dhcp_fail_ctr=0

                        'save this IP into the EEPROM for future use
                        if stg_get("IPN",0)<>ip then
                              stg_set("IPN",0,ip)
                        end if
                  end if
            end if
      #endif

      #if WLN_AVAILABLE
            if interface=PL_SOCK_INTERFACE_WLN then
                  if renew=YES and wln.ip<>ip then
                        'this is a lease renewal and the DHCP server has
issues new IP
                        'it is better to reboot than deal with the
implications of the changed IP
                        sys.reboot
                  end if

                  if wln.ip<>ip then
                        'no need to close sockets -- this is definitely
NOT a renewal an the DHCP lib
```

```
                            'has already closed all socket connections on
this interface prior to setting
                            'the IP to 0.0.0.0
                            wln.ip=ip
                            wln.gatewayip=gateway_ip
                            wln.netmask=netmask
                            wln_dhcp_fail_ctr=0

                            'save this IP into the EEPROM for future use
                            if stg_get("IPW",0)<>ip then
                                   stg_set("IPW",0,ip)
                            end if
                     end if
              end if
      #endif
end sub
...
'--------------------------------------------------------------------
sub callback_stg_error(byref stg_name_or_num as string,index as byte,status
as en_stg_status_codes)
end sub

'--------------------------------------------------------------------
sub callback_stg_pre_get(byref stg_name_or_num as string,index as byte,byref
stg_value as string)
end sub

'--------------------------------------------------------------------
sub callback_stg_post_set(byref stg_name_or_num as string, index as byte,
byref stg_value as string)
end sub
...
```

settings.txt (the underlying configuration file):

```
>>IPN E      D      1      4      4      A      0.0.0.0       Ethernet IP
>>IPW E      D      1      4      4      A      0.0.0.0       Wi-Fi IP

#define STG_DESCRIPTOR_FILE "settings.xtxt"
#define STG_MAX_NUM_SETTINGS 2
#define STG_MAX_SETTING_NAME_LEN 3
#define STG_MAX_SETTING_VALUE_LEN 15
```

Library Defines (Options)

Any of the options below look cryptic? Read the Operation Details 621 section.

DHCP_DEBUG_PRINT (default= 0)

0- no debug information.

1- print debug information into the output pane. Debug printing only works when the project is in the debug mode 27. However, still set this option to 0 for release, as this will save memory and code space.

DHCP_MAX_RETRIES (default= 3)

The number of retry attempts in one batch of DHCP configuration attempts.

DHCP_WAIT_TIME (default= 2)

Maximum waiting time, in seconds, for the DHCP server to respond to the device's request.

DHCP_MAX_RETRY_DELAY (default= 10)

Maximum waiting time, in seconds, between the retry attempts within one batch of retries. Actual waiting time is randomized between 1 and DHCP_RANDOM_RETRY_DELAY seconds.

DHCP_POST_FAIL_DELAY (default= 180)

Delay, in seconds, between the batches of retries.

DHCP_MAX_HOST_NAME_LEN (default=0)

Maximum device (host) name length.

En_dhcp_status_codes

Several procedures in the library utilize the en_dhcp_status_codes enum. This enum has the following members:

0- DHCP_STATUS_OK: success.

1- DHCP_STATUS_OUT_OF_SOCKETS: no free sockets available for the library to operate.

2- DHCP_STATUS_INVALID_INTERFACE: unsupported network interface specified (use PL_SOCK_INTERFACE_NET or PL_SOCK_INTERFACE_WLN only).

3- DHCP_STATUS_INSUFFICIENT_BUFFER_SPACE: insufficient number of buffer pages available and the call to callback_dhcp_pre_buffrq() 640 failed to cure the problem.

4- DHCP_STATUS_FAILURE: interaction with the DHCP server failed (because there was no reply, the reply was unrecognized, invalid, etc.).

Library Procedures

In this section:

- Dhcp_proc_timer() 637
- Dhcp_proc_data() 638
- Callback_dhcp_ok() 638
- Callback_dhcp_failure() 639
- Callback_dhcp_pre_clear_ip() 639
- Callback_dhcp_pre_buffrq() 640
- Callback_dhcp_buff_released() 640

Dhcp_get_info()

Description: API procedure, returns library-specific information according to the requested information element.

Syntax: **function dhcp_get_info(**info_element **as dhcp_info_elements, byref** extra_data **as string) as string**

Returns: Requested data in *string* form

See Also: About _get_info() API Functions 577

Part	Description
info_element	Information element being requested: 0- DHCP_INFO_ELEMENT_REQUIRED_BUFFERS: total number of buffer pages required for the library to operate.
extra_data	Leave this string argument empty.

Details

Dhcp_start()

Description: API procedure, starts DHCP client on the specified network interface.

Syntax: **function dhcp_start(**interface **as pl_sock_interfaces, byref** requested_ip **as string, byref** host_name **as string) as en_dhcp_status_codes**

Returns: One of these en_dhcp_status_codes 635: DHCP_STATUS_OK, DHCP_STATUS_OUT_OF_SOCKETS, DHCP_STATUS_INVALID_INTERFACE

See Also: Step-by-step Usage Instructions 619, Operation Details 621

Part	Description

interface	Network interface to start the DHCP client on: 0- PL_SOCK_INTERFACE_NULL: do not choose this one, it is for a "non-existing interface". 1- PL_SOCK_INTERFACE_NET: Ethernet (net. 358) interface. 2- PL_SOCK_INTERFACE_WLN: Wi-Fi (wln. 536) interface.
requested_ip	Optionally provide an IP address that the device wishes to use or continue using. This is typically an IP address that was previously obtained. Leave this argument empty if no IP is being requested specifically. This argument will be ignored if the data format is wrong (for example, "192.168.1").
host_name	The device name to register with the DHCP server. Add DHCP_MAX_HOST_NAME_LEN 634 option to global.tbh for this to work. Leave this argument empty if you don't wish to send the host name. Name length can't exceed the value set by DHCP_MAX_HOST_NAME_LEN.

Details

The interface option 0- PL_SOCK_INTERFACE_NULL exists because the interface argument is of the pl_sock_interfaces type, which defines available network interfaces for the sock. 421 object (see, for example, sock.targetinterface 506). Obviously, there is no point selecting this option for dhcp_start().

Dhcp_stop()

Description:	API procedure, stops DHCP client on the specified network interface.
Syntax:	**function dhcp_stop(**interface **as pl_sock_interfaces) as en_dhcp_status_codes**
Returns:	---
See Also:	Step-by-step Usage Instructions 619, Operation Details 621

Details

Dhcp_proc_timer()

Function:	Event procedure, call it from the on_sys_timer() 533 event handler.
Syntax:	**sub dhcp_proc_timer()**
Returns:	---
See Also:	Step-by-step Usage Instructions 619, Operation Details 621 (see "Additional Considerations"!)

Details

Dhcp_proc_data()

Function: Event procedure, call it from the on_sock_data_arrival() 489 event handler.

Syntax: **sub dhcp_proc_data()**

Returns: ---

See Also: Step-by-step Usage Instructions 619, Operation Details 621

Details

Callback_dhcp_ok()

Description: Callback procedure, informs of the successful procurement of configuration parameters from the DHCP server. Procedure body has to be created elsewhere in the project (externally with respect to the library).

Syntax: **sub callback_dhcp_ok(**renew **as no_yes,** interface **as pl_sock_interfaces, byref** dhcp_obtained_ip **as string, byref** dhcp_obtained_gateway **as string, byref** dhcp_obtained_netmask **as string,** dhcp_obtained_lease_time **as dword)**

Returns: ---

See Also: Step-by-step Usage Instructions 619, Operation Details 621

Part	Description
renew	0- NO: this is an initial configuration of the IP, etc. 1- YES: this is a lease renewal (watch out for IP changes!).
interface	Network interface on which the DHCP configuration has completed successfully: 1- PL_SOCK_INTERFACE_NET: Ethernet (net. 358) interface. 2- PL_SOCK_INTERFACE_WLN: Wi-Fi (wln. 536) interface.
dhcp_obtained_ip	Obtained IP address in the human-readable form, i.e. "192.168.1.40".
dhcp_obtained_gateway	Obtained gateway IP address in the human-readable form, i.e. "192.168.1.1".
dhcp_obtained_ne	Obtained netmask in the human-readable form, i.e.

tmask	"255.255.255.0".
dhcp_obtained_lease_time	Obtained lease time expressed as the number of seconds.

Details

Callback_dhcp_failure()

Description: Callback procedure, informs of the failure to procure configuration parameters from the DHCP server after DHCP_MAX_RETRIES 634. Procedure body has to be created elsewhere in the project (externally with respect to the library).

Syntax: **sub callback_dhcp_failure(**interface **as pl_sock_interfaces,** failure_code **as en_dhcp_status_codes)**

Returns: ---

See Also: Step-by-step Usage Instructions 619, Operation Details 621

Part	Description
interface	Network interface on which the DHCP client has failed: 1- PL_SOCK_INTERFACE_NET: Ethernet (net. 358) interface. 2- PL_SOCK_INTERFACE_WLN: Wi-Fi (wln. 536) interface.
failure_code	One of the en_dhcp_status_codes 635

Details

Callback_dhcp_pre_clear_ip()

Description: Callback procedure, informs that the DHCP library will set the specified interface's IP address to 0.0.0.0. Procedure body has to be created elsewhere in the project (externally with respect to the library).

Syntax: **sub callback_dhcp_pre_clear_ip(**interface **as pl_sock_interfaces)**

Returns: ---

See Also: Step-by-step Usage Instructions 619, Operation Details 621

Part	Description
interface	Network interface on which the library will set the IP to 0.0.0.0: 1- PL_SOCK_INTERFACE_NET: Ethernet (net.358) interface. 2- PL_SOCK_INTERFACE_WLN: Wi-Fi (wln.536) interface.

Details

This procedure will only be called if the IP address is not already set to 0.0.0.0.

Callback_dhcp_pre_buffrq()

Description: Callback procedure, informs of the insufficient number of free buffer pages available for use by the library. Procedure body has to be created elsewhere in the project (externally with respect to the library).

Syntax: **sub callback_dhcp_pre_buffrq(**required_buff_pages **as byte)**

Returns: ---

See Also: Step-by-step Usage Instructions619, Operation Details621

Part	Description
required_buffer_pages	The number of additional buffer pages that the DHCP library needs to operate. Your application must free up at least this number of buffer pages within the callback_dhcp_pre_buffrq () or DHCP configuration will fail with the DHCP_STATUS_INSUFFICIENT_BUFFER_SPACE code635.

Details

This procedure will be called only if there are not enough buffer pages available.

Callback_dhcp_buff_released()

Description: Callback procedure, informs of the release of buffer pages by the library. Procedure body has to be created elsewhere in the project (externally with respect to the library).

Syntax: **sub callback_dhcp_buff_released()**

Returns: ---

See Also: Step-by-step Usage Instructions619, Operation Details621

Details

The number of free buffer pages can be checked through the sys.freebuffpages 531 R/O property.

FILENUM (File Numbers) Library

The FILENUM library automates file number assignment. The fd. 236 object supports up to fd.maxopenedfiles 282 opened files. Using the FILENUM library, your code can get an unused file number, work with the file "on" this number, then release the file number into a pool of free file numbers.

Library Info

Supported platforms:	Any platform with the fd. 236 object.
Files to include:	Filenum.tbs, filenum.tbh (from *current_library_set*\filenum\trunk\).
Dependencies:	---
API procedures: Use API procedures 576 to interact with the library.	filenum_get() 644 -- returns a free file number or 255 if no free file numbers left. filenum_who_uses() 644 -- returns the signature of the specified file number's user. filenum_release(645) -- releases the file number.
Event procedures:	---
Callback procedures:	---
Required buffer space:	---

Step-by-step Usage Instructions

Minimal steps

1. Add 577 **filenum.tbs** and **filenum.tbh** files to your project (they are in *current_library_set*\filenum\trunk).
2. Add **include "filenum\trunk\filenum.tbh"** to the includes section of the **global.tbh** file.
3. Add #define FILENUM_MAX_FILENAME_LEN 643 to the defines section of the **global.tbh** file; set it to the value that reflects the longest filename you are going to use.
4. Call filenum_get() 644 whenever you need a free file number. Observe the result -- if it is 255 then there are no free file numbers available!
5. Call filenum_release() 645 when you no longer need a particular file number. This way, the file number can be reused by something else in your code.

Additional recommended steps

1. The use of filenum_who_uses() 644 is optional. This call will return a string signature left by the caller when obtaining the file number. FILENUM_MAX_SIGNATURE_LEN 643 must be >0 for signatures to be saved.

We have provided a code snippet 642 illustrating the use of the FILENUM library.

Using the FILENUM library? Then, use it everywhere in your code! Do not "appropriate" file numbers without going through this library.

Operation Details

The FILENUM library maintains a list of free and occupied file numbers. This is kept in the filenum_in_use array. You can see this array's contents under the debugger. To get a free file number, call filenum_get() 644 and it will return a file number, which will now be marked as used. The function will return 255 if there are no free file numbers left, so watch out for this number!

When calling filenum_get(), you can supply a meaningful short name of the caller. For example, if the file number is to be used for accessing the log file, set the signature to "LOG". The filenum_user_signature array keeps the signatures left by filenum_get() callers. Observing this array's contents under the debugger will provide a clear picture of who is using which file number. The signature of free file numbers is "-". Note that FILENUM_MAX_SIGNATURE_LEN defines maximum signature length and the default value is 0. Set this to anything above 0 to be able to observe the signatures. You can set it back to 0 when the debugging process is over.

Another way to observe the operation of the library in the debug mode 27 is by adding #define FILENUM_DEBUG_PRINT 1 643 to the defines section of the global.tbh file of your project. A wealth of status information will be printed in the console pane as the FILENUM library operates.

To release the file niumber, call filenum_release() 645. This will mark the file number as free and the next caller of the filenum_get() will have a chance to reuse this file number.

Filenum_who_uses() 644 returns a signature left by the specified file number's user. This function doesn't have to be used and exists only for convenience.

A Code Snippet

Here is a small code snippet illustrating the process of getting and releasing file numbers. We sacrificed robustness for simplicity, so don't take the code below for a shining example of fd. 236 object's usage. Just see the parts related to the FILENUM library -- that's the point right now.

```
...
'set "#define FILENUM_MAX_SIGNATURE_LEN 3" and "#define FILENUM_DEBUG_PRINT
1" before running this test

'prepare the disk and create a file
fd.format(fd.availableflashspace,16)
fd.mount()
fd.create("log.txt")
```

```
'get a file number...
log_filenum=filenum_get("LOG")
if log_filenum=255 then
      sys.halt '...uh-uh, out of sockets!
end if

'add data to the file, then close
fd.filenum=log_filenum
fd.open("LOG")
fd.setdata(filenum_who_uses(log_filenum))
fd.close

'release the file number
filenum_release(log_filenum)
...
```

And here is what appeared in the output pane of TIDE as the code executed:

```
FILENUM> 'LOG' got file #0
FILENUM> 'LOG' released file #0
```

The first message corresponds to filenum_get() 644, the second one -- to filenum_release() 645. "LOG" is the signature left by us, because this code deals with a log file.

Notice that the data saved into the log file consists of the file number's signature. As a result, the file will contain the string "LOG".

Library Defines (Options)

Any of the options below look cryptic? Read the Operation Details 642 section.

FILENUM_DEBUG_PRINT (default= 0)

0- no debug information.

1- print debug information into the output pane. Debug printing only works when the project is in the debug mode 27. However, still set this option to 0 for release, as this will save memory and code space.

FILENUM_MAX_SIGNATURE_LEN (default= 0)

Size of members of the filenum_user_signature array. Set to >0 to be able to observe the signatures left by the callers of the filenum_get() 644.

FILENUM_MAX_FILENAME_LEN (default= 0)

Maximum filename length. This must be set to at least the length of the longest filename you are planning to use in your project.

Library Procedures

In this section:

Filenum_get()

Description: API procedure, returns a free file number or 255 if no free file numbers left.

Syntax: **function filenum_get(byref** signature **as string) as byte**

Returns: The number of the unused file number or 255 if all file numbers are in use.

See Also: Step-by-step Usage Instructions 641, Operation Details 642

Part	Description
signature	A string with a short description of the file number's purpose. You may leave the signature empty, too.

Details

Set FILENUM_MAX_SIGNATURE_LEN 643 to anything above 0 in order to be able to use signatures.

Filenum_who_uses()

Description: API procedure, returns the signature of the specified file number's user.

Syntax: **function filenum_who_uses(**file_num **as byte) as string**

Returns: The signature left by the file number's user on filenum_get() 644 invocation.

See Also: Step-by-step Usage Instructions 641, Operation Details 642

Part	Description
file_num	File number.

Details

Set FILENUM_MAX_SIGNATURE_LEN 643 to anything above 0 in order to be able to use signatures.

Filenum_release()

Description: API procedure, releases the file number.

Syntax: **sub filenum_release(**file_num **as byte)**

Returns: ---

See Also: Step-by-step Usage Instructions 641, Operation Details 642

Part	Description
file_num	The file number to be released.

Details

GPRS (PPP) Library

The GPRS library handles PPP link establishment and is primarily targeted for GPRS modems. Together with the ppp. 366 object, it forms a complete GPRS/PPP solution for your device.

Although the library can theoretically work with any GPRS modem, it has been specifically tested with GC864 GPRS module manufactured by TELIT (www.telit.com). This module can be installed on the NB1010 board offered by Tibbo.

The library is event-based and non-blocking -- it quietly runs in the background and takes a minimal amount of CPU time.

Library Info

Supported platforms:	Any platform with a serial port.
Files to include:	Gprs.tbs, gprs.tbh (from *current_library_set*\gprs\trunk\).
Dependencies:	SOCK 664 library.
API procedures:	gprs_start() 652 -- starts the PPP login/configuration process.
Use API procedures 576 to interact with the library.	gprs_stop() 653 -- stops (aborts) PPP link establishment or session.
Event procedures:	gprs_proc_timer() 653 -- call this from the on_sys_timer() 533 event handler.
Call event procedures 576 from corresponding event handlers, as described here 619.	gprs_proc_sock_data() 653 -- call this from the on_sock_data_arrival() 489 event handler.
	gprs_proc_ser_data() 654 -- call this from the on_ser_data_arrival() 412 event handler.
Callback procedures:	callback_gprs_ok() 654 -- called when the library completes PPP login/configuration.

Implement the bodies of callback procedures elsewhere in your project.	callback_gprs_failure() -- called when PPP login/ configuration or established link fails. callback_gprs_pre_buffrq() -- called when the library needs to allocate buffer space and the required space is not available.
Required buffer space:	10 buffer pages minimum with GPRS_PAYLOAD_SIZE of 1. Released when gprs_stop() is called.

Step-by-step Usage Instructions

Steps below assume that you are using the NB1010 board (DS101x device) for testing.

1. Make sure you have the SOCK library in your project.
2. Add **gprs.tbs** and **gprs.tbh** files to your project (they are in *current_library_set*\gprs\trunk).
3. Add #define GPRS_DEBUG_PRINT 1 to the defines section of the global.tbh file of your project. This way you will "see what's going on". Don't forget to remove this later, after you've made sure that the library operates as expected.
4. Add **include "gprs\trunk\gprs.tbh"** to the includes section of the global.tbh file.
5. Add gprs_proc_timer() to the on_sys_timer() event handler code (see "further considerations" below).
6. Add pppoe_proc_sock_data() to the on_sock_data_arrival() event handler code.
7. Add pppoe_proc_ser_data() to the on_ser_data_arrival() event handler code.
8. Create empty callback function bodies (presumably in the device.tbs): callback_gprs_ok(), callback_gprs_failure(), callback_gprs_pre_buffrq(). Hint: copy from declarations in the pppoe.tbh or from our code example.
9. Call gprs_start() from somewhere. The no-brainer decision is to call from the on_sys_init() event handler. Supply correct ATDT line and APN. In our tests on *Taiwan Cellular* network, we have to use ***99#** and **INTERNET**. Your network may require a different string. Note that pppoe_start() may fail, so it is wise to check the returned status code.
10. Implement meaningful code for the callback_gprs_ok(). Once it is called, you know that your PPP link is up.

All of the above is illustrated in the code example.

Further considerations

The GPRS library expects the gprs_proc_timer() to be called at 0.5 sec intervals. This is the default value which can be changed through the sys.onsystimerperiod property. If your project needs to have the on_sys_timer() event at a different rate, please make sure that gprs_proc_timer() is still called 2 times/second. For example, if the on_sys_timer() is set to trigger 4 times/second, you need to add "divider" code that only calls gprs_proc_timer() on every second invocation of the on_sys_timer().

If you are using an external GPRS modem, you need to add #define

GPRS_MODULE_EXTERNAL 1 650 to global.tbh.

Depending on how you connect the GPRS modem, you may also need to add the following custom defines:

- GPRS_SER_PORT 650
- GPRS_SER_PORT_RTSMAP 650
- GPRS_SER_PORT_CTSMAP 650
- GPRS_SWITCH 650
- GPRS_RESET 650

Operation Details

Enter topic text here.

Operation Details

GPRS library's operation is persistent: once you start it by calling gprs_start() 652, it will "persist" to keep PPP link established. No matter how many times the link fails, the library will keep trying. If the successfully established PPP link fails, the library will attempt to reestablish the link. This can only be halted by calling gprs_stop() 653 .

GPRS library is buffer-hungry. Even if you set GPRS_PAYLOAD_SIZE 650 to 1, you will need 10 buffer pages for the GPRS library to operate. With default payload size of 4, this increases to 25 buffer pages. GPRS_PAYLOAD_SIZE defines the maximum size of TCP and UDP packets that the PPP interface will be able to handle. Unless you specifically want to send out large UDP packets, it is actually OK to set the payload size to the minimum -- GPRS links are slow and you won't feel any performance decrease just because you limit the size of packets.

Buffer space is allocated when gprs_start() 652 is called. The library will check if the required buffer space is available and call callback_ppp_pre_buffrq() 663 if more space is needed. Buffer pages allocated to the PPP library will only be released if gprs_stop() 661 is called.

Successul PPP link establishment is a multi-step process and involves link configuration using LCP protocol, followed by the login and the configuration of device's IP address. To observe the process, add #define GPRS_DEBUG_PRINT 1 650 to the defines section of the global.tbh file of your project (and don't forget to remove this later).

Once the PPP link is successfully established, the library will set ppp.ip 368 and call callback_gprs_ok() 654. The library will then start monitoring PPP link's "health". The library will periodically send echo request packets and expect echo replies from the other end. If this times out, the library will assume that the PPP link has failed, call gprs_failure() 654, and attempt to reestablish the link.

If your application calls pppoe_stop() 661, the library will apply the hardware reset to the GPRS module (provided that GPRS_MODULE_EXTERNAL 650 is 0), release all buffers, and go into the idle state until gprs_start() 652 is called again.

Code Example

global.tbh:

```
'DEFINES---------------------------------------------------------------
#define SUPPORTS_GPRS 1
#define GPRS_DEBUG_PRINT 1
#define GPRS_PAYLOAD_SIZE 1
```

```
'INCLUDES--------------------------------------------------------------
include "sock\trunk\sock.tbh"
include "gprs\trunk\gprs.tbh"

'DECLARATIONS----------------------------------------------------------
declare tcp_sock_o as byte
```

main.tbs:

```
include "global.tbh"
'----------------------------------------------------------------------

const AT_DT_COMMAND="*99#"
const REMOTE_IP="124.155.161.141"                    '<----- CHANGE THIS AS
NEEDED
const REMOTE_PORT=1001                               '<----- CHANGE THIS AS
NEEDED
'----------------------------------------------------------------------
dim tcp_sock_o as byte

'======================================================================
sub on_sys_init()
      dim res as en_gprs_status_codes

      '----- this is for the outgoing test connection
      tcp_sock_o=sock_get("TCPA")
      sock.num=tcp_sock_o
      sock.txbuffrq(1)
      sock.rxbuffrq(1)
      sys.buffalloc
      sock.protocol=PL_SOCK_PROTOCOL_TCP
      sock.targetip=REMOTE_IP
      sock.targetport=REMOTE_PORT
      sock.targetinterface=PL_SOCK_INTERFACE_PPP
      '----- end

      res=gprs_start(AT_DT_COMMAND)
end sub

'----------------------------------------------------------------------
sub on_ser_data_arrival()
      gprs_proc_ser_data()
end sub

'----------------------------------------------------------------------
sub on_sock_data_arrival()
      dim s as string(32)

      gprs_proc_sock_data()

      '----- this is for the outgoing test connection
      if sock.num=tcp_sock_o then
            s=sock.getdata(255)
            pat.play("-***",PL_PAT_CANINT)
      end if
      '----- end
end sub
```

```
'-------------------------------------------------------------------
sub on_sys_timer()
      gprs_proc_timer()

      '----- this is for the outgoing test connection
      sock.num=tcp_sock_o
      if sock.statesimple=PL_SSTS_EST then
            sock.setdata("ABC")
            sock.send
      end if
      '----- end
end sub

'-------------------------------------------------------------------
sub on_sock_event(newstate as pl_sock_state, newstatesimple as
pl_sock_state_simple)
      '----- this is for the outgoing test connection
      if sock.num=tcp_sock_o then
            pat.play("-***",PL_PAT_CANINT)
      end if
      '----- end
end sub

'-------------------------------------------------------------------
sub on_pat()
      '----- this is for the outgoing test connection
      sock.num=tcp_sock_o
      if sock.statesimple=PL_SSTS_EST then
            pat.play("G~",PL_PAT_CANINT)
      end if
      '----- end
end sub
```

device.tbs:

```
include "global.tbh"

'===================================================================
sub callback_gprs_ok()
      sock.num=tcp_sock_o
      sock.connect
end sub

'-------------------------------------------------------------------
sub callback_gprs_failure()
      sock.num=tcp_sock_o
      sock.discard
end sub

'-------------------------------------------------------------------
sub callback_gprs_pre_buffrq(required_buff_pages as byte)
end sub
```

Library Defines (Options)

Any of the options below look cryptic? Read Operation Details 647 section.

SUPPORTS_GPRS (default= 0)

0- GPRS disabled (library code won't be compiled into the project).

1- GPRS enabled.

GPRS_DEBUG_PRINT (default= 0)

0- no debug information.

1- print debug information into the output pane. Debug printing only works when the project is in the debug mode 27. However, still set this option to 0 for release, as this will save memory and code space.

GPRS_SEND_PING (default= 0)

0- suppress pinging.

1- ping GPRS_PING_IP in the end of PPP link establishment (as a proof of success).

GPRS_PING_IP (default= 0.0.0.0)

This is the IP address that will be pinged when GPRS_SEND_PING is set.

Only relevant when GPRS_SEND_PING is 1.

GPRS_SER_PORT (default= 3)

The serial port to which the GPRS modem is connected.

Like with ser.num 411, serial ports are numbered from 0. Keep at default value for NB1010 and DS101x devices.

GPRS_SER_PORT_RTSMAP (default depends on GPRS_SER_PORT)

The mapping of the RTS line connected to the GPRS modem.

If GPRS_SER_PORT is 0, the default value of GPRS_SER_PORT_RTSMAP is 0.

If GPRS_SER_PORT is 1, the default value of GPRS_SER_PORT_RTSMAP is 1.

If GPRS_SER_PORT is 2, the default value of GPRS_SER_PORT_RTSMAP is 2.

If GPRS_SER_PORT is 3, the default value of GPRS_SER_PORT_RTSMAP is 3.

Keep at default value for NB1010 and DS101x devices.

GPRS_SER_PORT_CTSMAP (default depends on GPRS_SER_PORT)

The mapping of the CTS interrupt line connected to the GPRS modem.

If GPRS_SER_PORT is 0, the default value of GPRS_SER_PORT_CTSMAP is 0.

If GPRS_SER_PORT is 1, the default value of GPRS_SER_PORT_CTSMAP is 1.

If GPRS_SER_PORT is 2, the default value of GPRS_SER_PORT_CTSMAP is 2.

If GPRS_SER_PORT is 3, the default value of GPRS_SER_PORT_CTSMAP is 3.

Keep at default value for NB1010 and DS101x devices.

GPRS_MODULE_EXTERNAL (default= 0)

0- the GPRS module is internal (like on the NB1010 and DS101x), GPRS_SWITCH and GPRS_RESET lines are used to control the module; there is a proper reset and startup delay when gprs_start() 652 is called.

1- the GPRS module is external, GPRS_SWITCH and GPRS_RESET GPIO lines are not in use, there is no startup delay- the GPRS module is supposed to be up and running by the time gprs_start() 652 is called.

Keep at default value for NB1010 and DS101x devices.

GPRS_SWITCH (default= 54)

GPIO line that, when set LOW, diverts traffic on GPRS_SER_PORT to the GPRS module. When the line is HIGH or disabled, serial port traffic goes elsewhere in the device.

Only relevant when GPRS_MODULE_EXTERNAL is 0.

Keep at default value for NB1010 and DS101x devices.

GPRS_RESET (default= 55)

GPIO line that, when set HIGH, applies hardware reset to the GPRS module. The line is set to LOW to release the GPRS module from reset.

'Only relevant when GPRS_MODULE_EXTERNAL is 0.

Keep at default value for NB1010 and DS101x devices.

GPRS_PAYLOAD_SIZE (default= 4)

Size of TCP and UDP packets that the GPRS interface will be able to carry, expressed in 256-byte units.

'Do not set >4 or <1. Smaller value reduces the total number of buffer pages required by the library (see gprs_get_info() 652).

En_gprs_status_codes

Several procedures in the library utilize the en_gprs_status_codes enum. This enum has the following members:

0- GPRS_STATUS_OK: success.

1- GPRS_STATUS_INSUFFICIENT_BUFFER_SPACE: insufficient number of buffer pages available and the call to callback_gprs_pre_buffrq() 655 failed to cure the problem.

Library Procedures

In this section:

Gprs_get_info()

Description: API procedure, returns library-specific information according to the requested information element.

Syntax: **function gprs_get_info(**info_element **as gprs_info_elements, byref** extra_data **as string) as string**

Returns: Requested data in *string* form

See Also: About _get_info() API Functions 577

Part	Description
info_element	Information element being requested: 0- GPRS_INFO_ELEMENT_REQUIRED_BUFFERS: total number of buffer pages required for the library to operate.
extra_data	Leave this string argument empty.

Details

Gprs_start()

Description: API procedure, starts the PPP login/configuration process.

Syntax: **function gprs_start(byref** at_dt_command_param **as string, byref** apn **as string) as en_gprs_status_codes**

Returns: One of these en_pppoe_status_codes 660:
PPP_STATUS_OK,
PPP_STATUS_INSUFFICIENT_BUFFER_SPACE

See Also: Step-by-step Usage Instructions 646, Operation Details 647

Part	Description
at_dt_command_param	This depends on the mobile network. Here is an actual line to be used with *Taiwan Cellular*: ***99#**.
apn	Again, the APN depends on the network. In here we have to use **INTERNET**.

Details

GPRS library operation is persistent -- once you start it with this call, the library will "persist" to keep the PPP link established. No matter how many times this fails, the library will keep trying. If the successfully established PPP link fails, the library will attempt to reestablish the link.

Gprs_stop()

Description: API procedure, stops (aborts) PPP link establishment or session.

Syntax: **sub gprs_stop()**

Returns: ---

See Also: Step-by-step Usage Instructions 646, Operation Details 647

Details

Gprs_proc_timer()

Function: Event procedure, call it from the on_sys_timer() 533 event handler.

Syntax: **sub gprs_proc_timer()**

Returns: ---

See Also: Step-by-step Usage Instructions 646, Operation Details 647

Details

Gprs_proc_sock_data()

Function: Event procedure, call it from the on_sock_data_arrival() 489 event handler.

Syntax: **sub gprs_proc_sock_data()**

Returns: ---

See Also: Step-by-step Usage Instructions 646, Operation Details 647

Details

Gprs_proc_ser_data()

Function: Event procedure, call it from the on_ser_data_arrival() 412 event handler.

Syntax: **sub gprs_proc_ser_data()**

Returns: ---

See Also: Step-by-step Usage Instructions 646, Operation Details 647

Details

Callback_gprs_ok()

Description: Callback procedure, informs of the successful establishment of the PPP link. Procedure body has to be created elsewhere in the project (externally with respect to the library).

Syntax: **sub callback_gprs_ok()**

Returns: ---

See Also: Step-by-step Usage Instructions 646, Operation Details 647

Details

By the time this procedure is called, the GPRS library will have already set ppp.ip 368.

Callback_gprs_failure()

Description: Callback procedure, informs that PPP login/configuration or established link failed. Procedure body has to be created elsewhere in the project (externally with respect to the library).

Syntax: **sub callback_gprs_failure()**

Returns: ---

See Also: Step-by-step Usage Instructions 646, Operation Details 647

Details

Callback_gprs_pre_buffrq()

Description: Callback procedure, informs of the insufficient number of free buffer pages available for use by the library. Procedure body has to be created elsewhere in the project (externally with respect to the library).

Syntax: **sub callback_gprs_pre_buffrq(**required_buff_pages **as byte)**

Returns: ---

See Also: Step-by-step Usage Instructions 646, Operation Details 647

Part	Description
required_buffer_pages	The number of additional buffer pages that the GPRS library needs to operate. Your application must free up at least this number of buffer pages within callback_gprs_pre_buffrq() or PPP login/configuration will fail with the GPRS_STATUS_INSUFFICIENT_BUFFER_SPACE code 651.

Details

This procedure will be called only if there are not enough buffer pages available.

PPPOE Library

The PPPOE library handles PPPoE login and configuration. Together with the pppoe. 369 object, it forms a complete PPPoE solution for your device.

The library is event-based and non-blocking -- it quietly runs in the background and takes a minimal amount of CPU time.

Library Info

Supported platforms: Any platform with the Ethernet (net. 358) interface.

Files to include: Pppoe.tbs, pppoe.tbh (from *current_library_set*\pppoe\trunk\).

Dependencies: SOCK 664 library.

API procedures: pppoe_start() 661 -- starts the PPPoE login/configuration process.

Use API procedures 576 to interact with the library.

pppoe_stop() 661 -- stops (aborts) PPPOE login/ configuration or session.

Event procedures: pppoe_proc_timer() 662 -- call this from the on_sys_timer() 533 event handler.

Call event procedures 576 from corresponding event handlers, as

pppoe_proc_data() 662 -- call this from the on_sock_data_arrival() 489 event handler.

described here.

Callback procedures:	callback_pppoe_ok() -- called when the library completes PPPoE login/configuration.
Implement the bodies of callback procedures elsewhere in your project.	callback_pppoe_failure() -- called when PPPoE login/configuration or established link fails.
	callback_pppoe_pre_buffrq() -- called when the library needs to allocate buffer space and the required space is not available.
Required buffer space:	2 buffer pages. These are never released, even when you do pppoe_stop().

Step-by-step Usage Instructions

1. Make sure you have the SOCK library in your project.
2. Add **pppoe.tbs** and **pppoe.tbh** files to your project (they are in *current_library_set*\pppoe\trunk).
3. Add #define PPPOE_DEBUG_PRINT 1 to the defines section of the global.tbh file of your project. This way you will "see what's going on". Don't forget to remove this later, after you've made sure that the library operates as expected.
4. Add **include "pppoe\trunk\pppoe.tbh"** to the includes section of the global.tbh file.
5. Add pppoe_proc_timer() to the on_sys_timer() event handler code (see "further considerations" below).
6. Add pppoe_proc_data() to the on_sock_data_arrival() event handler code.
7. Create empty callback function bodies (presumably in the device.tbs): callback_pppoe_ok(), callback_pppoe_failure(), callback_pppoe_pre_buffrq(). Hint: copy from declarations in the pppoe.tbh or from our code example.
8. Call pppoe_start() from somewhere. The no-brainer decision is to call from the on_sys_init() event handler. Supply correct ADSL login and password (suggestion: use STG library to keep them). Note that pppoe_start() may fail, so it is wise to check the returned status code.
9. Implement meaningful code for the callback_pppoe_ok(). Once it is called, you know that your PPPoE interface is up.

All of the above is illustrated in the code example.

Further considerations

The PPPOE library expects the pppoe_proc_timer() to be called at 0.5 sec intervals. This is the default value which can be changed through the sys.onsystimerperiod property. If your project needs to have the on_sys_timer() event at a different rate, please make sure that pppoe_proc_timer() is still called 2 times/second. For example, if the on_sys_timer() is set to trigger 4 times/second, you need to add "divider" code that only calls pppoe_proc_timer() on every second invocation of the on_sys_timer().

Operation Details

PPPoE library's operation is persistent: once you start it by calling pppoe_start() 661, it will "persist" to connect to the ADSL modem. No matter how many times the connection fails, the library will keep trying. If the successfully established PPPOE link fails, the library will attempt to reestablish the link. This can only be halted by calling pppoe_stop() 661.

The PPPoE specification envisions the use of more than one ADSL modem (a.k.a. "access concentrator") on the network. The protocol provides the means for the device to choose which access concentrator it wants to communicate through. In reality, most networks only have a single ADSL modem. Since this is a norm, the PPPOE library is designed to attempt to use whatever access concentrator responds to the device's "discovery" packets first.

A buffer space of 2 pages is required for the library operation. These are allocated when pppoe_start() 661 is called. The library will check if the required buffer space is available and call callback_pppoe_pre_buffrq() 663 if more space is needed. Buffer pages allocated to the PPPOE library are never released, even when successful PPPoE link is established or pppoe_stop() 661 is called.

Successul PPPoE link establishment is a multi-step process and involves link configuration using LCP protocol, followed by the login and the configuration of device's IP address. To observe the process, add #define PPPOE_DEBUG_PRINT 1 659 to the defines section of the global.tbh file of your project (and don't forget to remove this later).

Once the PPPoE link is successfully established, the library will set the properties of the pppoe. 369 object and call callback_pppoe_ok() 662. The library will then start monitoring PPPoE link's "health". The library will periodically send echo request packets and expect echo replies from the ADSL modem. If this times out, the library will assume that the PPPoE link has failed, call pppoe_failure() 662, and attempt to reestablish the link.

If your application calls pppoe_stop() 661, the library will property terminate the PPPoE link (if it was established) and go into the idle state until pppoe_start() 661 is called again.

You can learn more about PPPoE here: http://en.wikipedia.org/wiki/Pppoe.

Code Example

Here is a simple example of using the PPPOE library. Pppoe_start() is called on boot (). Once the PPPoE link is established, the device will open an outgoing TCP connection to a certain remote IP and port.

Green status LED is used to indicate the TCP connection status: it will be on when the TCP connection is established.

To test the TCP connection, the code will periodically send a small string of data to the remote end of the connection. If any reply is received, the green status LED will momentarily turn off.

Before running the code, do not forget to change ADSL_NAME, ADSL_LOGIN, REMOTE_IP, and REMOTE_PORT constants.

global.tbh:

```
'DEFINES-----------------------------------------------------------------
#define PPPOE_DEBUG_PRINT 1

'INCLUDES----------------------------------------------------------------
include "sock\trunk\sock.tbh"
```

```
include "pppoe\trunk\pppoe.tbh"

'DECLARATIONS----------------------------------------------------------
declare tcp_sock_o as byte
```

main.tbs:

```
include "global.tbh"

'-----------------------------------------------------------------------
const ADSL_NAME="correct_name"                  '<----- CHANGE THIS AS NEEDED
const ADSL_PASSWORD="correct_password"  '<----- CHANGE THIS AS NEEDED
const REMOTE_IP="59.120.32.27"                  '<----- CHANGE THIS AS NEEDED
const REMOTE_PORT=40000                                 '<----- CHANGE THIS AS
NEEDED

'-----------------------------------------------------------------------
dim tcp_sock_o as byte

'=======================================================================
sub on_sys_init()
      dim res as en_pppoe_status_codes

      '----- this is for the outgoing test connection
      tcp_sock_o=sock_get("TCPA")
      sock.num=tcp_sock_o
      sock.txbuffrq(1)
      sock.rxbuffrq(1)
      sys.buffalloc
      sock.protocol=PL_SOCK_PROTOCOL_TCP
      sock.targetip=REMOTE_IP
      sock.targetport=REMOTE_PORT
      sock.targetinterface=PL_SOCK_INTERFACE_PPPOE
      '----- end

      res=pppoe_start(ADSL_NAME,ADSL_PASSWORD)
end sub

'-----------------------------------------------------------------------
sub on_sock_data_arrival()
      dim s as string(32)

      pppoe_proc_data()

      '----- this is for the outgoing test connection
      if sock.num=tcp_sock_o then
            s=sock.getdata(255)
            pat.play("-***",PL_PAT_CANINT)
      end if
      '----- end
end sub

'-----------------------------------------------------------------------
sub on_sys_timer()
      pppoe_proc_timer()

      '----- this is for the outgoing test connection
      sock.num=tcp_sock_o
      if sock.statesimple=PL_SSTS_EST then
```

```
            sock.setdata("ABC")
            sock.send
      end if
      '----- end
end sub

'-------------------------------------------------------------------
sub on_sock_event(newstate as pl_sock_state, newstatesimple as
pl_sock_state_simple)
      '----- this is for the outgoing test connection
      if sock.num=tcp_sock_o then
            pat.play("-***",PL_PAT_CANINT)
      end if
      '----- end
end sub

'-------------------------------------------------------------------
sub on_pat()
      '----- this is for the outgoing test connection
      sock.num=tcp_sock_o
      if sock.statesimple=PL_SSTS_EST then
            pat.play("G~",PL_PAT_CANINT)
      end if
      '----- end
end sub
```

device.tbs:

```
include "global.tbh"

'===================================================================
sub callback_pppoe_ok()
      sock.num=tcp_sock_o
      sock.connect
end sub

'-------------------------------------------------------------------
sub callback_pppoe_failure(pppoe_code as en_pppoe_status_codes)
      sock.num=tcp_sock_o
      sock.discard
end sub

'-------------------------------------------------------------------
sub callback_pppoe_pre_buffrq(required_buff_pages as byte)
end sub
```

Library Defines (Options)

There is only one define for this library:

PPPOE_DEBUG_PRINT (default= 0)

0- no debug information.

1- print debug information into the output pane. Debug printing only works when the project is in the debug mode[27]. However, still set this option to 0 for release,

as this will save memory and code space.

En_pppoe_status_codes

Several procedures in the library utilize the en_pppoe_status_codes enum. This enum has the following members:

0- PPPOE_STATUS_OK: success.

1- PPPOE_STATUS_OUT_OF_SOCKETS: no free sockets available for the library to operate.

2- PPPOE_STATUS_INSUFFICIENT_BUFFER_SPACE: insufficient number of buffer pages available and the call to callback_pppoe_pre_buffrq() 663 failed to cure the problem.

3- PPPOE_TIMEOUT: PPPoE login/configuration or established link failed -- timeout while waiting for the reply from the ADSL modem (access concentrator).

4- PPPOE_CONFIGURATION_ERROR: PPPoE login/configuration failed -- something went wrong... and on the PPPoE many things can go wrong. Perhaps, login name or password should be re-checked?

5- PPPOE_STOPPED: pppoe_stop() 661 was called.

Library Procedures

In this section:

Pppoe_get_info()

Description: API procedure, returns library-specific information according to the requested information element.

Syntax: **function pppoe_get_info(**info_element **as pppoe_info_elements, byref** extra_data **as string) as string**

Returns: Requested data in *string* form

See Also: About _get_info() API Functions 577

Part	Description
info_element	Information element being requested: 0- PPPOE_INFO_ELEMENT_REQUIRED_BUFFERS: total number

	of buffer pages required for the library to operate.
extra_data	Leave this string argument empty.

Details

Pppoe_start()

Description: API procedure, starts the PPPoE login/configuration process.

Syntax: **function pppoe_start(byref** login **as string, byref** password **as string) as en_pppoe_status_codes**

Returns: One of these en_pppoe_status_codes 660: PPPOE_STATUS_OK, PPPOE_STATUS_OUT_OF_SOCKETS, PPPOE_STATUS_INSUFFICIENT_BUFFER_SPACE

See Also: Step-by-step Usage Instructions 656, Operation Details 657

Part	Description
login	ADSL login name.
password	ADSL login password.

Details

PPPOE library operation is persistent -- once you start it with this call, the library will "persist" to connect to the ADSL modem. No matter how many times the connection fails, the library will keep trying. If the successfully established PPPOE link fails, the library will attempt to reestablish the link.

Pppoe_stop()

Description: API procedure, stops (aborts) PPPOE login/configuration or session.

Syntax: **sub pppoe_stop()**

Returns: ---

See Also: Step-by-step Usage Instructions 656, Operation Details 657

Details

Pppoe_proc_timer()

Function: Event procedure, call it from the on_sys_timer() 533 event handler.

Syntax: **sub pppoe_proc_timer()**

Returns: ---

See Also: Step-by-step Usage Instructions 656, Operation Details 657

Details

Pppoe_proc_data()

Function: Event procedure, call it from the on_sock_data_arrival() 489 event handler.

Syntax: **sub pppoe_proc_data()**

Returns: ---

See Also: Step-by-step Usage Instructions 656, Operation Details 657

Details

Callback_pppoe_ok()

Description: Callback procedure, informs of the successful establishment of the PPPoE link. Procedure body has to be created elsewhere in the project (externally with respect to the library).

Syntax: **sub callback_pppoe_ok()**

Returns: ---

See Also: Step-by-step Usage Instructions 656, Operation Details 657

Details

By the time this procedure is called, the PPPOE library will have already set pppoe.ip 370, pppoe.acmac 369, and pppoe.sessionid 370.

Callback_pppoe_failure()

Description: Callback procedure, informs that PPPoE login/configuration or established link failed. Procedure body has to be created elsewhere in the project (externally with respect to the library).

Syntax: **sub callback_pppoe_failure(**pppoe_code **as**

en_pppoe_status_codescodes)

Returns: ---

See Also: Step-by-step Usage Instructions 656, Operation Details 657

Part	Description
pppoe_state	Reason for failure: 3- PPPOE_TIMEOUT: timeout while waiting for the reply from the ADSL modem (access concentrator). 4- PPPOE_CONFIGURATION_ERROR: something went wrong... and on the PPPoE many things can go wrong. Perhaps, login name or password should be re-checked?. 5- PPPOE_STOPPED: pppoe_stop() 661 was called.

Details

As follows from the above, this procedure is called even if your application "manually" aborts PPPoE by calling pppoe_stop() 661.

Callback_pppoe_pre_buffrq()

Description: Callback procedure, informs of the insufficient number of free buffer pages available for use by the library. Procedure body has to be created elsewhere in the project (externally with respect to the library).

Syntax: **sub callback_pppoe_pre_buffrq(**required_buff_pages **as byte)**

Returns: ---

See Also: Step-by-step Usage Instructions 656, Operation Details 657

Part	Description
required_buffer_pages	The number of additional buffer pages that the PPPOE library needs to operate. Your application must free up at least this number of buffer pages within the callback_pppoe_pre_buffrq() or PPPoE login/configuration will fail with the PPPOE_STATUS_INSUFFICIENT_BUFFER_SPACE code 660.

Details

This procedure will be called only if there are not enough buffer pages available.

SOCK (Socket Numbers) Library

The SOCK library automates socket number assignment. The sock. 421 object supports up to sock.numofsock 488 sockets. Using the SOCK library, your code can get an unused socket, use the socket as needed, then release the socket into a pool of free sockets.

Library Info

Supported platforms:	Any platform with the sock. 421 object.
Files to include:	Sock.tbs, sock.tbh (from *current_library_set*\sock\trunk\).
Dependencies:	---
API procedures: Use API procedures 576 to interact with the library.	sock_get() 667 -- returns a free socket number or 255 if no free sockets left. sock_who_uses() 668 -- returns the signature of the specified socket's user. sock_release() 668 -- releases the socket (number), discards socket connection, restores socket's properties to their default states.
Event procedures:	---
Callback procedures:	---
Required buffer space:	---

Step-by-step Usage Instructions

Minimal steps

1. Add 577 **sock.tbs** and **sock.tbh** files to your project (they are in *current_library_set*\sock\trunk).
2. Add **include "sock\trunk\sock.tbh"** to the includes section of the global.tbh file.
3. Call sock_get() 667 whenever you need a free socket. Observe the result -- if it is 255 then there are no free sockets available!
4. Call sock_release() 668 when you no longer need a particular socket. This way, the socket can be reused by something else in your code.

Additional recommended steps

1. The use of sock_who_uses() 668 is optional. This call will return a string signature left by the caller when obtaining the socket. SOCK_MAX_SIGNATURE_LEN 667 must be >0 for signatures to be saved.

We have provided a code snippet 665 illustrating the use of the SOCK library.

Using the SOCK library? Then, use it everywhere in your code! Do not "appropriate" sockets without going through this library.

Operation Details

The SOCK library maintains a list of free and occupied sockets. This is kept in the sock_in_use array. You can see this array's contents under the debugger. To get a free socket (number), call sock_get() 667 and it will return a socket number, which will now be marked as used. The function will return 255 if there are no free sockets left, so watch out for this number!

When calling sock_get(), you can supply a meaningful short name of the caller. For example, if the socket is to be used for TELNET-style communications with the device, set the signature to "TLNT". The sock_user_signature array keeps the signatures left by sock_get() callers. Observing this array's contents under the debugger will provide a clear picture of who is using which socket. The signature of free sockets is "-". Note that SOCK_MAX_SIGNATURE_LEN 667 defines maximum signature length and the default value is 0. Set this to anything above 0 to be able to observe the signatures. You can set it back to 0 when the debugging process is over.

Another way to observe the operation of the library in the debug mode 27 is by adding #define SOCK_DEBUG_PRINT 1 667 to the defines section of the global.tbh file of your project. A wealth of status information will be printed in the console pane as the SOCK library operates.

To release the socket, call sock_release() 668. This will mark the socket as free and the next caller of the sock_get() will have a chance to reuse this socket number. Additionally, the socket will be "cleaned up for reuse". Any socket connection in progress will be discarded (with sock.discard 478) and each property will be returned to its original, post-reset value.

Sock_who_uses() 668 returns a signature left by the specified socket's user. This function doesn't have to be used and exists only for convenience.

A Code Snippet

Here is a small code snippet illustrating the process of getting and releasing sockets. We sacrificed robustness for simplicity, so don't take the code below for a shining example of sock. 421 object's usage. Just see the parts related to the SOCK library -- that's the point right now.

```
...
'set "#SOCK_MAX_SIGNATURE_LEN 4" and "#define SOCK_DEBUG_PRINT 1" before
running this test

dim tcp_sock as byte 'to remember the socket number we've got

net.ip="192.168.1.93" '<--- edit as needed

'get a socket
```

```
tcp_sock=sock_get("TCPS")
'gotta check if there was a free socket for us...
if tcp_sock=255 then
      sys.halt '...uh-huh, out of sockets!
end if

'setup the socket
sock.num=tcp_sock
sock.rxbuffrq(1)
sock.txbuffrq(1)
sys.buffalloc
sock.protocol=PL_SOCK_PROTOCOL_TCP
sock.targetip="192.168.1.67" '<--- edit as needed
sock.targetport=10000 '<--- edit as needed
sock.connect
while sock.statesimple<>PL_SSTS_EST
wend

'send data -- we use the signature of the socket's user
sock.setdata(sock_who_uses(tcp_sock))
sock.send
while sock.txlen>0
wend

'close the connection
sock.close
while sock.statesimple<>PL_SSTS_CLOSED
wend

'release the socket
sock_release(tcp_sock)
...
```

And here is what appeared in the output pane of TIDE as the code executed:

```
SOCK> 'TCPS' got socket #2
SOCK> 'TCPS' released socket #2
```

The first message corresponds to sock_get() 667, the second one -- to sock_release() 668. "TCPS" is the signature left by us, it stands for "TCP socket". The socket number obtained is 2 because we ran this code within a large program that uses a lot of other sockets for a lot of other things.

The example above establishes an outgoing connection to 192.168.1.67:10000. In our test, this was a PC running I/O Ninja, our sniffer/terminal software (you can get it at ninja.tibbo.com). We opened a "listener socket" on I/O Ninja, and here is what we saw:

```
Accepted TCP connection from 192.168.1.93:10254
TCPS
Remote node 192.168.1.93:10254 has closed TCP connection
```

This "TCPS" is the signature our code provided when calling sock_get(). The reason it appeared in Ninja is because of the sock.setdata(sock_who_uses(tcp_sock)) line in the code.

Library Defines (Options)

Any of the options below look cryptic? Read the Operation Details 665 section.

SOCK_DEBUG_PRINT (default= 0)

0- no debug information.

1- print debug information into the output pane. Debug printing only works when the project is in the debug mode 27. However, still set this option to 0 for release, as this will save memory and code space.

SOCK_MAX_SIGNATURE_LEN (default= 0)

Size of members of the sock_user_signature array. Set to >0 to be able to observe the signatures left by the callers of the sock_get() 667.

Library Procedures

In this section:

- Sock_get() 667
- Sock_who_uses() 668
- Sock_release() 668

Sock_get()

Description:	API procedure, returns a free socket number or 255 if no free sockets left.
Syntax:	**function sock_get(byref** signature **as string) as byte**
Returns:	The number of the unused socket or 255 if all sockets are in use.
See Also:	Step-by-step Usage Instructions 664, Operation Details 665

Part	Description
signature	A string with a short description of the socket's purpose. You may leave the signature empty, too.

Details

Set SOCK_MAX_SIGNATURE_LEN 667 to anything above 0 in order to be able to use signatures.

Sock_who_uses()

Description: API procedure, returns the signature of the specified socket's user.

Syntax: **function sock_who_uses(**sock_num **as byte) as string**

Returns: The signature left by the socket's user on sock_get() 667 invocation.

See Also: Step-by-step Usage Instructions 664, Operation Details 665

Part	Description
sock_num	Socket number.

Details

Set SOCK_MAX_SIGNATURE_LEN 667 to anything above 0 in order to be able to use signatures.

Sock_release()

Description: API procedure, releases the socket (number), discards socket connection, restores socket's properties to their default states.

Syntax: **sub sock_release(**sock_num **as byte)**

Returns: ---

See Also: Step-by-step Usage Instructions 664, Operation Details 665

Part	Description
sock_num	The number of the socket to be released.

Details

STG (Settings) Library

The STG library offers a persistent, convenient storage for your device's settings (operational parameters). It can also play a pivotal role in the device control and monitoring 670. The library is extremely easy to use -- just define a list of all desired settings using a setting configurator 670 and employ simple API calls to work with them.

Setting configurator allows you to specify the names, types, value constraints, etc. of your device's settings and the STG library uses this to automatically calculate memory addresses for storing settings, protect the settings with a checksum, verify the validity of their values, etc. You code is then able to reference setting values

by their names, like this: s=stg_get[696]("IP",0), stg_set[697]("IP",0,"192.168.1.40").

The library keeps your settings in the non-volatile memory or RAM. The non-volatile memory used can be the EEPROM memory (stor.[522]) or the flash disk (fd.[236]). For RAM, you can choose to go with "regular" RAM (the one that stores variables), or "custom" RAM, for which you can create your own access routines[684].

Library Info

Supported platforms:	Any platform.
Files to include:	settings.tbs, settings.tbh (from *current_library_set*\settings\trunk\).
Dependencies:	FILENUM[641] library when the **Use Flash Disk** option is selected.
API procedures: Use API procedures[576] to interact with the library.	stg_start()[691] -- starts the STG library, parses the descriptor file, restores all volatile (RAM) settings to default values. stg_check_all()[691] -- verifies each setting by reading its value. stg_get_def()[692] -- returns the default value for the specified setting's member. stg_restore_multiple()[692] -- initializes (sets the default values for) all or multiple settings. stg_restore_member()[693] -- initializes (sets the default value for) the specified setting's member. stg_get_num_settings()[694] -- returns the number of settings defined in your project. stg_get_num_members()[694] -- returns the number of members for the specified setting. stg_find()[695] -- returns the number of the specified setting or zero if the setting wasn't found. stg_stype_get()[696] -- returns the type of the specified setting. stg_get()[696] -- reads (gets) the value of the specified setting's member; reports errors through callback_stg_error()[700]. stg_set()[697] -- writes (sets) the value of the specified setting's member; reports errors through callback_stg_error()[700]. stg_sg()[698] -- writes (sets) or reads (gets) the specified setting's member; directly returns the execution result. stg_sg_ts()[699] -- available when the **Timestamp** option is enabled (checked), reads or writes the setting modification time /date.

Event procedures:	---
Callback procedures:	callback_stg_error()[700] -- informs of the error during the execution of stg_get()[696] or set_set()[697].
Implement the bodies of callback procedures[576] elsewhere in your project.	callback_stg_pre_get()[700] -- informs that the setting member's value is being read through one of the STG library's procedures; allows to update the setting member's value before it is returned by the STG library. callback_stg_post_set()[701] -- informs that the setting member's value is being written to through one of the STG library's procedures; allows to respond to the value change. callback_stg_vm_read()[702] -- should implement necessary code for reading a byte of data from "custom" volatile memory. Needed only when the **Custom RAM** option is enabled (checked). callback_stg_vm_write()[703] -- should implement necessary code for writing a byte of data to "custom" volatile memory. Needed only when the **Custom RAM** option is enabled (checked).
Required buffer space:	---

Controlling Your Device Through Settings

Like the Windows registry, the STG library may be used to provide a well-organized, persistent storage for your device's operational parameters. In many cases, settings will be the backbone of your Tibbo BASIC project, as numerous other libraries utilize settings for parameter (variable) storage.

One less obvious and very powerful concept of setting usage is that settings can serve as *transit points* for controlling and/or monitoring the device. That is, writing to a setting can cause the related code to be executed *after* the new value is set (post-set[683]), and this code will put the new value "into action". Reading a setting can cause the value of the setting to be updated *first* (pre-get[683]), and only then returned to the caller.

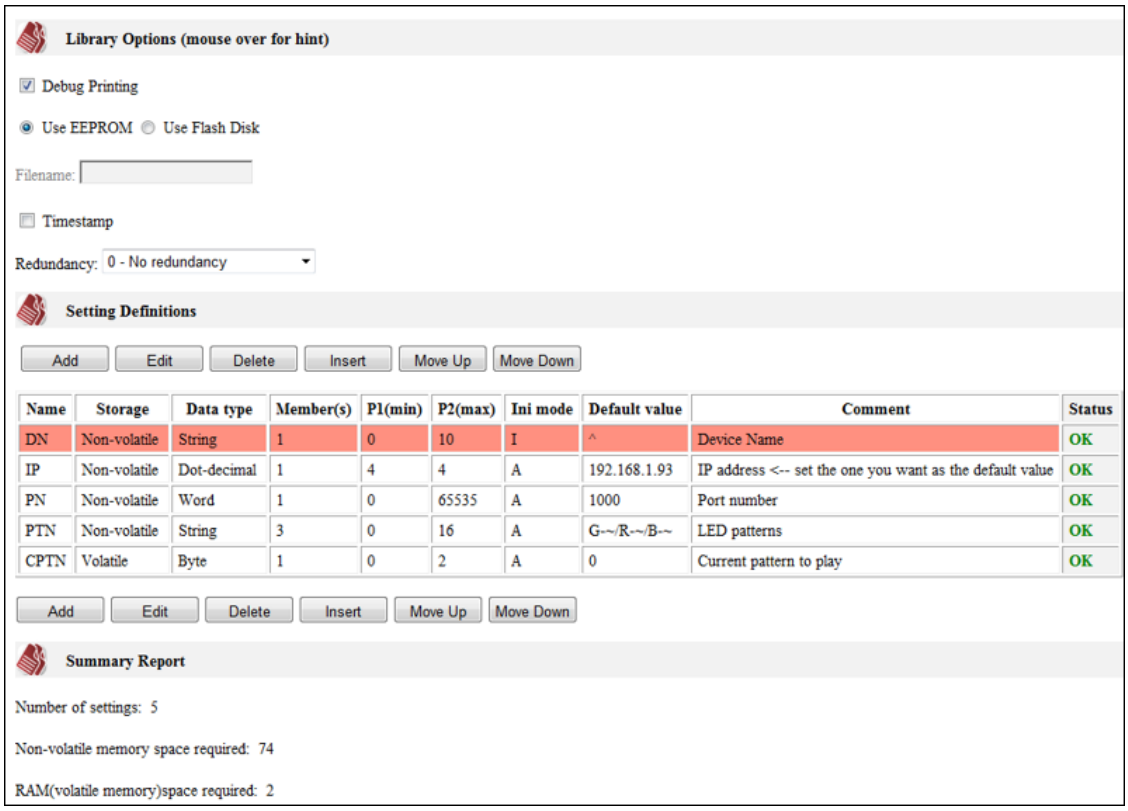
In other words, it is possible to change your device's operation (behavior, mode, etc.) just by writing a new value to a setting. It is also possible to get the current device status just by reading a setting's value. This is a very powerful concept. Master it, and you will be able to create clean, structured applications.

Using Pre-gets and Post-sets[683] continues this discussion.

Setting Configurator

Setting configurator is a JavaScript-based editor[578] for the setting configuration file. Do not confuse the two. Setting descriptor file is a part of your project (it is a resource file). The editor is a "representer" -- it provides a convenient interface for the editing of the configuration file. The configurator also masks the complexity of the underlying configuration file.

In this section:



Library Options (mouse over for hint)

Debug Printing

Use EEPROM Use Flash Disk

Filename:

Timestamp

Redundancy: 0 - No redundancy

Setting Definitions

Add | Edit | Delete | Insert | Move Up | Move Down

Name	Storage	Data type	Member(s)	P1(min)	P2(max)	Ini mode	Default value	Comment	Status
DN	Non-volatile	String	1	0	10	I	^	Device Name	OK
IP	Non-volatile	Dot-decimal	1	4	4	A	192.168.1.93	IP address <-- set the one you want as the default value	OK
PN	Non-volatile	Word	1	0	65535	A	1000	Port number	OK
PTN	Non-volatile	String	3	0	16	A	G~~/R~~/B~~	LED patterns	OK
CPTN	Volatile	Byte	1	0	2	A	0	Current pattern to play	OK

Add | Edit | Delete | Insert | Move Up | Move Down

Summary Report

Number of settings: 5

Non-volatile memory space required: 74

RAM(volatile memory)space required: 2

Library Options

The Library Options section of the configurator screen lists all configurable options.

Debug Printing check box

Check (enable) to print debug information into the output pane. Debug printing only works when the project is in the debug mode 27. Always uncheck this for release, as this will save memory and code space.

Use EEPROM / Use Flash Disk option boxes

Defines whether non-volatile settings are to be stored in the EEPROM or in the file (on the flash disk). Choose *Use EEPROM* whenever possible.

Filename text box

When **Use Flash Disk** is selected, defines the filename in which the setting values will be stored on the flash disk.

Timestamp check box

When checked (enabled), stores the date and time of the most recent setting modification. One timestamp is kept for all members of each setting.

Redundancy drop-down

No Redundancy - only one copy of data is stored for all settings.

Two copies for EEPROM - two copies of data are stored for non-volatile settings. Only one copy is maintained for volatile settings.

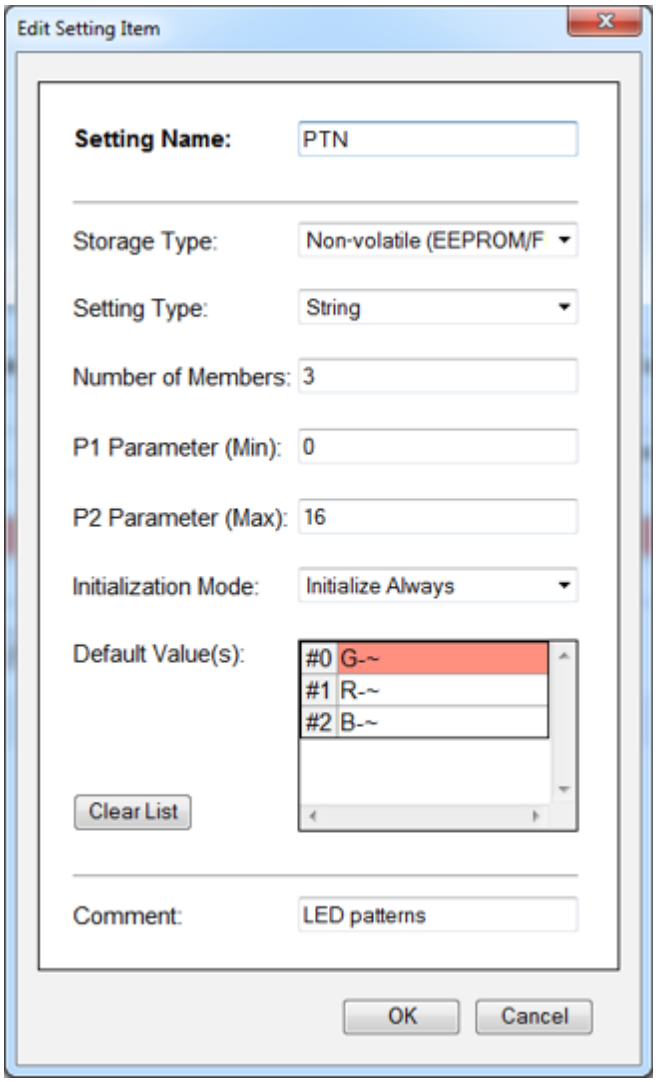
Two copies for ALL - two copies of data are stored for all settings (non-volatile as well as volatile).

Custom RAM

Check (enable) to be able to create your own routines for writing and reading volatile settings through callback_stg_vm_read() 702 and callback_stg_vm_write() 703 .

Editing Settings

Use **Add, Edit**, and **Delete** buttons to create the list of settings.



Setting name

It is this name that you will use to reference settings in stg_set() 697, stg_get() 696, etc. Names are case-sensitive and cannot start with a digit (i.e. 0-9 char). This is because the STG library's procedures would interpret this as a setting number 680, not name. Try to keep the names short, this saves RAM memory space (by reducing the amount of memory needed to keep the list of setting names).

Storage Location

Specifies whether the setting will be stored in the non-volatile memory (EEPROM or flash as defined by the global **Use EEPROM** / **Use Flash Disk** option) or volatile memory (RAM). Obviously, settings that go into the RAM are those that you don't have to keep while the device is powered off. If so, why even use settings? Wouldn't regular variables suffice? Yes and no. Regular variables are fast to access, but using settings for the purpose has its own advantages 670.

Setting Type

You've got byte (0-255 range), word (0-65535 range), string (up to 253 characters), and dot-decimal string 674 (up to 63 bytes).

Number of Members

Each setting must have at least one member, and can be set to include multiple members (like in an array). The maximum number of members 675 for the setting depends on this setting's type.

P1 Parameter (Min)

The minimum value 675 for byte and word settings, minimum length for string and dot-decimal settings.

P2 Parameter (Max)

The maximum value 675 for byte and word settings, maximum length for string and dot-decimal settings.

Initialization Mode

Initialize always -- the setting is unconditionally initialized during stg_restore_multiple() 692 execution.

Initialize when invalid -- the setting is only initialized during stg_restore_multiple() execution if found to contain an invalid or corrupted value.

Default values

A separate default value 675 can be provided for each setting member, but not necessarily.

Comment

Any comment you like. Do add comments -- they really help. These get compiled into the final application binary, so having longer commends increases the size of your application.

Dot-decimal Settings

Dot-decimal settings exist to store parameters such as IP and MAC address. Like strings, they consist of a number of bytes. The difference is that when a dot-decimal setting is read, ddstr() 210 is executed on its members. For example, if the internal value of the setting consists of bytes 192, 168, 1, and 40, then "192.168.1.40" is returned when the setting is read. When the setting is written to, the ddval() 210 is executed, so "192.168.1.40" because a set of 4 bytes (192, 168,

1, and 40) again.

Max Number of Members

Here is what you have when the **Timestamp** option[671] is unchecked:

- **Byte** settings: up to 254 members.
- **Word** settings : up to 127 members.
- **String** and **dot-decimal** settings: up to 254/(P2+1), see above for explanation on the P2 parameter. Example: if P2=10 then a setting can have up to 23 members.

You get less with timestamps enabled:

- **Byte** settings: up to 247 members.
- **Word** settings : up to 123 members.
- **String** and **dot-decimal** settings: up to 247/(P2+1), see above for explanation on the P2 parameter. Example: if P2=10 then a setting can have up to 23 members.

P1 and P2 Parameters

These define minimum (P1) and maximum (P2) values for byte and word settings, and minimum and maximum length for string and dot-decimal settings. Conditions:

- P2 must be greater or equal to P1.
- P2 can't be 0.
- For byte settings, P1 and P2 can't exceed 255.
- For word settings, P1 and P2 can't exceed 65535.
- For string and dot-decimal settings, P1 and P2 can't exceed 254.

Default Setting Values

In the simplest case, you have a setting with one member, and one default value to provide for it... but what to do when the setting is of string or dot-decimal type and the intended default value for this setting is NULL? Don't just leave the field empty, use a special character ^ instead!

Several default values may be supplied, in which case they should be separated by /, like this: "abc/def/123". This string defines three default values. And what if you have, say, five members in this setting? Then, member 0 will be initialized to "abc", member 1 -- to "def", and members 2 through 4 -- to "123". And how about this string -- "abc/^/123"? It specifies that the second initialization value is NULL.

Naturally, ^ and / can't be used in setting strings because they are "special" characters carrying a special meaning. If you need to have them inside the initialization value itself, use C style escape sequences. For example, if the default value is supposed to be "2/3" then write "2**\x2F**3".

The total size of the default value field cannot exceed 255 characters. Hence, you may not be able to provide all the default values you want to. Say, you have a byte setting with 100 members. The default value for each member can take three characters plus the separator. That's up to four characters per one default value. Only 64 such values will fit in 255 characters.

Step-by-step Usage Instructions

In this section:

Getting Started

Minimal steps

1. Add 577 **settings.tbs** and **settings.tbh** files to your project (they are in *current_library_set*\settings\trunk).
2. Add the following to the includes section of the **global.tbh** file:

```
includepp "settings.xtxt"
include "settings\trunk\settings.tbh"
```

3. Create empty callback procedure 576 bodies (presumably in the device.tbs, and according to declarations in **settings.tbh**):
   - callback_stg_error() 700;
   - callback_stg_pre_get() 700;
   - callback_stg_post_set() 701.
4. Add 18 an empty **settings.xtxt** file to your project. In the **Add New File to Project** dialog set the **Type** to *Definition File*, and the **Format** to *STG (Settings) Library*.
5. Define desired settings through the configurator 670.
6. In the On_sys_init() 533 event handler (or some other place you deem logical), add a call to the stg_start() 691. This function may fail, so check the returned status code 690.
7. Take care of setting initialization 677.

Additional Steps

1. By default, non-volatile settings are stored in the EEPROM. You can also store the settings on the flash disk. EEPROM offers limited (but usually sufficient) storage with a faster access, while the flash disk offers a virtually unlimited storage with a slower access. Go with the EEPROM whenever possible. If you want to use the flash disk, select *Use Flash Disk* option in the configurator 670; then set the desired **Filename** (this is the name of the file that will store setting values).
2. Set **Redundancy** to *Non-volatile only* to increase the reliability of settings by keeping two independent copies of their values. The penalty is in the reduced performance of the library (increased setting access time) and doubling of the EEPROM (or flash) memory storage needs (see Operation Details 684 for explanation how memory is allocated for both copies).

Verifying and Initializing Settings

Initialization is the process of making setting values "sane". To be considered sane, the setting value must have correct checksum, and its members must conform to the constraints of P1 and P2 parameters 675.

Non-volatile settings are never initialized automatically -- it is your responsibility as the developer to decide just when and how this happens. If you load a settings-using application onto a device that has never run this application before, then each non-volatile setting will be invalid because all setting checksums will be wrong. Volatile settings are always initialized when you call stg_start() 691.

Any setting can be "repaired" by writing a valid value into it. You can do this using stg_sg() 698 or stg_set() 697 -- see Writing and Reading Settings 678.

A more "automatic" way is to use stg_restore_multiple() 692. This will restore all settings to their default values as defined in the setting descriptor file. In the following example, settings are initialized if the MD button is kept pressed for more than 2 seconds and then released:

```
sub on_button_released()
      if button.time>4 then
            if stg_restore_multiple(EN_STG_INIT_MODE_NORMAL)
<>EN_STG_STATUS_OK then sys.halt
      end if
end sub
```

You can also verify the sanity of each setting in your project by calling stg_check_all() 691. Here is some code you *can* place in on_sys_init() 533 -- it verifies *all* settings and performs initialization if *any* setting is found to be invalid. Notice how we check the code returned by stg_check_all(). EN_STG_STATUS_INVALID 690 is a "good" error, that is, we expect it to happen. There are other errors, however, that are "fatal", for example, EN_STG_STATUS_FAILURE 690. We halt on those.

```
sub on_sys_init()
      dim x as en_stg_status_codes
      dim stg_name as string(STG_MAX_SETTING_NAME_LEN)

      if stg_start()<>EN_STG_STATUS_OK then sys.halt

      x=stg_check_all(stg_name)
      select case x
      case EN_STG_STATUS_OK:
            '--- all good ---

      case EN_STG_STATUS_INVALID, EN_STG_STATUS_FAILURE:
            if stg_restore_multiple(EN_STG_INIT_MODE_NORMAL)
<>EN_STG_STATUS_OK then sys.halt

      case else:
            'some other trouble
            sys.halt
      end select

      comms_init()
end sub
```

There is also a library procedure to restore a single member of a single setting -- stg_restore_member() 693. The default value for any member of any setting can be obtained by calling stg_get_def() 692.

Writing and Reading Settings

In this section:

Using Stg_sg()

Stg_sg() 698 allows you to set (write) or get (read) a setting and directly returns the execution result so you can check if the setting operation was successful (hence, the name "_sg", which means "set/get"). In the following example, we set the setting "IP" to "192.168.1.40":

```
dim result as en_stg_status_codes
...
result=stg_sg("IP",0,"192.168.1.40",EN_STG_SET)
select case result
case EN_STG_STATUS_OK:
      'all good
case EN_STG_STATUS_UNKNOWN, EN_STG_STATUS_INVALID_INDEX:
      'bad setting name or index
case EN_STG_STATUS_FAILURE:
      'failure ro write
case EN_STG_STATUS_INVALID:
      'new setting value is invalidinvalid value
case else:
      'some weird error
end select
...
```

Here is another example where we read the value of the same setting. Notice that stg_sg() returns the setting value indirectly, through one of its arguments:

```
dim result as en_stg_status_codes
```

```
dim s as string
...
result=stg_sg("IP",0,s,EN_STG_GET)
select case result
case EN_STG_STATUS_OK:
      'all good, s now contains current setting value
case EN_STG_STATUS_UNKNOWN, EN_STG_STATUS_INVALID_INDEX:
      'bad setting name or index
case EN_STG_STATUS_FAILURE:
      'failure ro write
case EN_STG_STATUS_INVALID:
      'new setting value is invalid
case else:
      'some weird error
end select
...
```

Using Stg_get() and Stg_set()

Extensive use of the stg_sg() [678] can quickly bloat your project because of all this result checking that you must do everywhere you call stg_sg(). Fortunately, there is another way, but do exercise caution with it -- see details below.

Stg_set() [697] and stg_get() [696] do not return the execution result directly. Instead, they invoke callback_stg_error() [700] whenever any error is detected. This procedure then can be a catch-all point for setting-related errors in your project. Here is an example:

```
...
sub procedure1()
      dim s,s2 as string(STG_MAX_SETTING_VALUE_LEN)
      ...
      s=stg_get("IP",0)
      s2=stg_get("PN",0)
end sub

sub procedure2()
      stg_set("CPTN",0,"2")
end sub

sub callback_stg_error(byref stg_name_or_num as string,index as byte,status
as en_stg_status_codes)
      pat.play("R-",PL_PAT_CANINT) 'setting error -- blink red LED!
aaa:  goto aaa 'This is a "halt" so we hang here forever.
end sub
```

Notice how readable and simple the code above is. You simply read and write settings, and when some error happens, you handle it in callback_stg_error(). Since this is a single place for handling setting errors, you usually include some dramatic code there... something equivalent to the "blue screen of death" in old Windows.

A sobering note -- think when it is OK (and not OK) to use stg_get() and stg_set(). Typically, "it is not OK" when the setting is being written to after some sort of user input. Let's say the user is editing settings via a web-browser interface. He makes a mistake and inputs an invalid new value for the setting. If this part of your code relies on stg_set() then this simple user mistake will lead to the halt of device

operation, which is totally unwarranted.

So, as a rule of thumb, use stg_set() and stg_get() "internally", in places where incorrect user input can't cause errors. In places where user input is processed use stg_sg() 698, which will allow you to check execution result "right there" and respond to the user if there was an erroneous input.

We use the same approach in our own sample project 686.

One additional fact about the stg_get() 696. Not only does it invoke callback_stg_error() whenever there is an error, but also returns the default value of the setting (defined with the setting configurator 670) when possible. This way, if the setting can't be read, you get its default value and your reliable product can continue operating somehow.

Because the TIDE compiler does not allow recursions 66, some STG library's own procedures can't be called from within callback_stg_error() 700. These procedures are:

- Stg_get() 696
- Stg_set() 697

Using Setting Numbers

Up until now we referred to settings by their names. It is also possible to call settings by their number. Settings are numbered in the order of their appearance in the setting configurator 670. Settings are counted from 0.

Here is a neat example where we read (the first member of) every available setting. Notice how the error condition (EN_STG_STATUS_UNKNOWN 690) is used to exit the loop:

```
...
dim s as string
dim f as byte
dim result as en_stg_status_codes
f=0
do
      result=stg_sg(str(f),0,s,EN_STG_GET)
      f=f+1
loop until result<>EN_STG_STATUS_OK

if result<>EN_STG_STATUS_UNKNOWN then
      sys.halt 'some error has occurred!
end if
...
```

Of course, the above example can be implemented in a more conventional way using stg_get_num_settings() 694:

```
...
```

```
dim s as string
dim f as byte
dim result as en_stg_status_codes

for f=0 to stg_get_num_settings()-1
      s=stg_get(str(f),0)
next f
...
```

So, how does the STG library know when to interpret your input as a setting name or number? Simple! If it starts with a digit (0-9), then this is a setting number, otherwise it is a setting name. Names, therefore, can't start with a digit.

```
...
s=stg_get("CPTN",0) 'refer to the setting by its name
s=stg)get("4",0) 'refer to the setting by its number
...
```

Working With Multi-value Settings

A single setting may contain several members, like in an array. Stg_sg() 698, as well as stg_get() 696 and stg_set() 697 require that you specify the setting member you want to read or write. Setting members are counted starting from 0. Get_stg_num_members() will tell you how many members each setting has.

Here is a modified example from the previous topic 680, it goes through every member of every setting:

```
...
dim s as string
dim f,f2,num as byte
dim result as en_stg_status_codes

for f=0 to stg_get_num_settings()-1
      stg_get_num_members(str(f),num)
      for f2=0 to num-1
            s=stg_get(str(f),f2)
      next f2
next f
...
```

Understanding Timestamps

Sometimes it is necessary to remember the date/time of the most recent setting modification. This can be useful, for instance, when synchronizing settings between several devices.

To enable timestamps, click (enable) the **Timestamp** option in the configurator[670]. Timestamps occupy an additional 7 bytes of memory for each setting (in the EEPROM, flash disk file, or in RAM, depending on how the setting is stored). This is why timestamps are disabled by default.

There is a single timestamp for each setting, even if this setting includes multiple members[681]. Timestamps are read and written through the stg_timestamp[689] global variable. Note that it is the responsibility of your application to set and interpret timestamps. The STG library does not access the real-time clock of your system and does not verify the contents of stg_timestamp variable.

To set the new timestamp for the setting, fill out the stg_timestamp variable with desired data prior to calling stg_set()[697] or stg_sg(,,,EN_STG_SET)[698]. Here is an example:

```
...
rtc.getdata(stg_timestamp.ts_daycount,stg_timestamp.ts_mincount,
stg_timestamp.ts_seconds)
stg_timestamp.ts_milsec=0 'our RTC does not provide ms data
stg_set("IP",0,"192.168.1.50")
...
```

You can also set *just the timestamp*, without altering setting value:

```
...
rtc.getdata(stg_timestamp.ts_daycount,stg_timestamp.ts_mincount,
stg_timestamp.ts_seconds)
stg_timestamp.ts_milsec=0 'our RTC does not provide ms data
if stg_set_ts("IP")<>EN_STG_STATUS_OK then sys.halt
...
```

Note that initializing settings[677] (restoring their values to defaults) through stg_restore_multiple()[692] and stg_restore_member()[693] also sets the timestamp of each affected setting.

Stg_timestamp variable is updated every time you read the setting with stg_get()[696] or stg_sg(,,,EN_STG_GET)[698]. In the following example, we read the setting in order to get its timestamp, then set new setting value if the setting hasn't been yet modified today:

```
...
dim daycount,mincount as word
dim seconds as byte

rtc.getdata(daycount,mincount,seconds)
stg_get("IP",0)
if stg_timestamp.ts_daycount<daycount then
     stg_set("IP",0,"ABC")
end if
...
```

Using Pre-gets and Post-sets

Like the Windows registry, STG library may be used to provide a well-organized, persistent storage for your device's operational parameters. One less obvious and very powerful concept of setting usage is that settings can serve as transit points for controlling and/or monitoring the device. That is, writing to a setting can cause some related code to be executed *after* the new value is set (post-write), and this code will put the new value "into action". Reading a setting can cause the value of the setting to be updated *first* (pre-read), and only then returned to the caller.

Imagine, for instance, that there is an "LS" (Lamp State) setting that defined if the "lamp" is on or off. Once we change the setting value through, say, a telnet command, or some kind of setup screen, the state of the lamp should change, too.

The elegant way of achieving this is by using callback_stg_post_set() 701. It is invoked every time stg_set() 697 or stg_sg(,,,EN_STG_SET) 698 is used. This callback procedure offers a catch-all place where you can respond to changing setting values.

Here is the code template for handling the lamp. The beauty of this approach is that the setting may be modified in several different places in your application, but you only need to respond to the setting value change in a single place -- callback_stg_post_set() procedure:

```
sub callback_stg_post_set(byref stg_name_or_num as string, index as byte,
byref stg_value as string)
      if stg_name_or_num="LS" then
            if stg_value=0 then
                  'turn the lamp off
            else
                  'turn the lamp on
            end if
      end if
end sub
```

Note that stg_restore_multiple() 692 and stg_restore_member() 693 also write to settings and callback_stg_post_set() will be called for them too -- once for each setting affected.

Another callback procedure -- callback_stg_pre_get() 700 -- is called every time your application reads a setting through stg_get() 696 or stg_sg(,,,EN_STG_GET) 698. This allows to update the setting value *before* returning it to the caller.

Example: Let's say you have a setting called "CT" (Current Temperature). Here is a code template for automatically updating the setting each time its value is requested:

```
```

```
sub callback_stg_pre_get(byref stg_name_or_num as string,index as byte,byref
stg_value as string)
      if stg_name_or_num="CPTN" then
            stg_value=get_temperature() 'some function in your project that
returns current T
      end if
end sub
```

The use of pre-gets and post-sets is further illustrated in our sample project 686.

Because TIDE compiler does not allow recursions 66, some STG library's own procedures can't be called from within callback_stg_post_set() 701 and callback_stg_pre_get() 700. These procedures are:

- Stg_start() 691
- Stg_check_all() 691
- Stg_restore_multiple() 692
- Stg_restore_member() 693
- Stg_get() 696
- Stg_set() 697
- Stg_sg() 698

Operation Details

STG library automatically allocates the storage for settings. That is, non-volatile and volatile memory addresses where individual settings are stored are calculated automatically.

When non-volatile settings are stored in the EEPROM (default configuration), they are located at the bottom of the EEPROM, right after the special configuration area 197.

When non-volatile settings are stored on a flash disk (*Use Flash Disk* option 670 selected), they are stored in the file specified by the **Filename** option 670.

When the **Custom RAM** option is not selected, the volatile (RAM) settings are stored in an array which is maintained by the library (stg_ram_array variable).

When the **Custom RAM** option is checked, the volatile (RAM) settings are stored using callback_stg_vm_read() 702 and callback-stg_vm_write() 703, which should contain custom code for accessing whatever memory you have decided to use for storing volatile settings (example: internal memory of a real-time clock IC).

When the **Redundancy** option is set to *two copies for non-volatile settings*, the library maintains a second copy of data for non-volatile settings (this doubles the required amount of non-volatile memory) and keeps a single copy for volatile settings. When the **Redundancy** option is set to *two copies for ALL settings*, even volatile settings are stored as two independent copies.

Two copies of setting data are stored like this:

Bottom of EEPROM (flash)

Data for all non-volatile settings (first copy)
Data for all non-volatile settings (second copy)

Bottom of RAM

Data for all volatile settings (first copy)
Data for all volatile settings (second copy)

Each setting is stored in the following format:

Setting storage

Setting data	Timestamp (7 bytes)*	Checksum byte

Only when the **Timestamp option is enabled.*

The checksum is a 255 complement of the modulo 256 (8-bit) sum of all data bytes for the setting except the checksum itself. The amount of storage allocated for setting data depends on this setting's properties. The total amount of memory taken by the setting can't exceed 255 bytes.

Setting data includes one or more setting members depending on the number of members 673 in this setting.

Setting data field for a setting with three members

Member0 data	Member1 data	Member2 data

Here is how member data for settings of different types is stored:

- **Byte** settings require a single byte of storage for each member.
- **Word** settings require two bytes of storage for each member, the data is stored like this:

HIGH byte	LOW byte

- **String** and **dot-decimal** settings require the space equal to the P2 parameter 675 (max length) plus one additional byte to store the current length of the data:

Current length (1 byte)	P2 number of bytes for data storage

Sample Project

This sample project is published on our website under the name **test_stg_lib.zip**:

http://tibbo.com/basic/resources.html.

Let's build a sample application. It will have a listening control socket through which you will be able to send control commands and get replies. These commands will allow you to write and read settings. One of the settings, with multiple members, will define a number of LED patterns. Another setting will select the current pattern to be played. Yet another setting will let you put the selected pattern into play and check if it is still playing.

This application, although largely useless, will demonstrate all of the fine points and advantages of the settings library.

The project uses five settings:

- **DN** (Device Name) -- stores a string that can uniquely identify the device. This setting does nothing for the program's functionality, but it is a nice touch to be able to name the device.
- **IP** (IP address) -- stores the IP address of the device.
- **PN** (Port Number) -- stores the port number on which the device will accept a TCP connection. You will use this TCP connection to send commands and receive replies.
- **PTN** (PaTterNs) -- has three members, each member stores one LED pattern.
- **CPTN** (Current PaTterN) -- specifies which pattern is currently selected.

Design steps:

- Step 1: The Embryo 686
- Step 2: Adding Setting Initialization 687
- Step 3: Adding Comms 688
- Step 4: Completing the Project 689

Step 1: The Embryo

This step corresponds to **test_stg_lib_1**.

The embryonic project defines the settings and compiles correctly. It does nothing else.

The steps

We assume you are not going to type everything in from scratch and just open the **test_agg_lib_1** project. Notice that...

1. **Settings.tbs** and **settings.tbh** are added 577 to the project (from

current_library_set\settings\trunk\). There is also a necessary line in **global.tbh**: **include "settings\trunk\settings.tbh"**.

2. There is a **settings.xtxt** configuration file 670 with type= **configuration file**, and format= **Setting (STG) library**. There is a line in global.tbh that is required for the configuration file to work correctly: **includepp** 96 **"settings.xtxt"**.
3. In the configuration file, **Debug Printing** is *enabled*. This allows you to "see what's going on". Don't forget to *disable* this later, after you've made sure that the library operates as expected.
4. There are empty callback procedures in **device.tbs**:
   - callback_stg_error() 700;
   - callback_stg_pre_get() 700;
   - callback_stg_post_set() 701;
5. On_sys_init() 533 calls stg_start() 691 -- this is required to make the library operational. Notice how we check the code returned by stg_start()!
6. All settings are defined, as on the screenshot shown here 670.

The result

Compile and run the project -- you will see the output like this:

```
STG> ---START---
STG> Initialize RAM (volatile) settings...
STG> STG_RESTORE_MULTIPLE(), init_mode= RAM ONLY
STG> STG_SG()
STG> SET 'CPTN(255)' to '0' : OK
STG> STG_SG()
STG> SET 'PIP(255)' to '0' : OK
STG> RAM (volatile) settings initialized
STG> Number of settings: 6
STG> Non-volatile memory space required: 193
```

You see this printout because the **Debug Printing** option 670 is enabled (checked).

This sample printout from the output pane shows that both RAM settings got initialized -- it happens automatically because, obviously, RAM settings need initialization every time the program runs. As for our non-volatile settings, we are supposed to find a way to initialize them in some convenient way. Read on... 687

Step 2: Adding Setting Initialization

This step corresponds to **test_stg_lib_2**.

Non-volatile settings must be initialized 677 at some point. In our sample project, we check the "health" of settings upon boot using stg_check_all() 691. Should *any* setting turn out to be invalid, we initialize *all* settings with stg_restore_multiple() 692. Granted, this is a very crude way of handling setting initialization, but it works just fine for a simple project like ours.

We also provided a way to initialize the settings at any time -- just press the MD button for more than 2 seconds, then release.

All related changes are in **main.tbs**.

Step 3: Adding Comms

This step corresponds to **test_stg_lib_3**. Procedures of interest are **comms_init()** and **comms_proc_cmd()**. Both are in **device.tbs**.

New stuff

OK, now comms. We use terminal software, for example, our own I/O Ninja software (ninja.tibbo.com) to send commands and receive replies. The "IP" setting defines the IP address of the device, while the "PN" setting specifies the port number on the device side. Setting initialization 687 will cause these settings to have default values of 192.168.1.93 and 1000, as specified through the setting configurator 670.

Writing and Reading Settings 678 already explained the difference between the usage of stg_get() 696/stg_set() 697 and stg_sg() 698. Here is a practical illustration: **comms_init()** uses "simplified" **stg_get()**. With this, there is no need to check the execution result each time we need to read a setting -- should there be any problem, callback_stg_error() 700 will be called and we respond to the error there.

Comms_proc_cmd(), on the contrary, needs to respond to each command with a meaningful status code. This is why it relies on **stg_sg()**, which returns the status code directly.

There are two commands: "S" (set setting) and "G" (get setting). Both commands should end with the CR (shown as <*CR*> below).

Set command format is ***Ssetting_name,index,setting_value***. The command returns the execution result, which is expressed by a single character (see below).

Get command format is ***Gsetting_name,index***. The command returns the execution result. If execution was successful, setting value is also returned.

Execution result codes are: **A** for OK, **C** for when the setting name is unknown or the index is invalid, **F** when there was some sort of failure, **I** when the setting value is invalid, and **U** for "weird internal errors".

The result

Here is a sample printout from I/O Ninja, we write and read one of the "PTN" setting's members. We set this member to "R-G-B-" and then read this value back:

```
Established TCP connection with 192.168.1.40:1000 from 3790
SPTN,2,R-G-B-<CR>
A<CR>
GPTN,2<CR>
AR-G-B-<CR>
```

Step 4: Completing the Project

This step corresponds to **test_stg_lib_4**.

New stuff

OK, settings defined, comms protocol is their to read and write them, now let's put all this to good use. Rules of engagement:

- Writing new value into the "CPTN" setting should play a corresponding pattern stored in the "PTN" setting.
- Briefly pressing the MD button should play the *next* pattern.

Our sample project nicely illustrates what was said in Using Pre-gets and Post-sets 683. When the "CPTN" setting is being written to, callback_stg_post_set() 701 is called and this gives us a chance to "load" another pattern. Notice how this is done! We can't use stg_get() 696 or stg_sg() 698 inside **callback_stg_post_set()** -- this limitation is explained in Using Pre-gets and Post-sets 683. We, however, need to read the "PTN" setting in order to play the new pattern!

Luckily, there is almost always a workaround. In our case, we load a very short blank pattern "-" instead. When this pattern is done playing, the on_pat() 365 event is generated and then we read one of the "PTN" members according to the **pattern_num** variable.

Callback_stg_pre_get() 701 is also useful in our project. When we send a command to read the current value of the "CPTN" setting, the setting is first updated with the current value of the **pattern_num** variable!

Notice also how stg_get_num_members() 694 is used in **play_next_pattern()**. Each time the button is pressed, **pattern_num** is incremented by one, until it reaches the number of members in the "PTN" setting, after which it is reset back to 0.

The result

Here is the I/O Ninja (ninja.tibbo.com) session where we switched between different patterns:

```
Established TCP connection with 192.168.1.93:1000 from 1723
SCPTN,0,1<CR>
A<CR>
ASCPTN,0,2<CR>
A<CR>
```

Try this and observe how different patterns start playing on the LEDs.

Stg_timestamp Global Variable

Stg_timestamp variable is used for working with setting timestamps 681. The variable is only available when the **Timestamp** option 670 is enabled.

Set this variable to the desired value prior to calling stg_sg(,,,EN_STG_SET) 698, stg_set() 697, stg_set_ts() 699, stg_restore_multiple() 692, or stg_restore_members() 694.

The variable will contain the timestamp of the setting in question after the call to stg_sg(,,,EN_STG_GET)[698] or stg_get()[696].

The variable is defined as follows:

```
type struct_stg_timestamp
      ts_daycount as word
      ts_mincount as word
      ts_seconds as byte
      ts_milsec as word
end type
...
declare stg_timestamp as struct_stg_timestamp
```

Note that it is the responsibility of your application to set and interpret timestamps. The STG library does not access the real-time clock of your system and does not verify the contents of stg_timestamp variable.

En_stg_status_codes

Several procedure in the library utilize the en_stg_status_codes enum. This enum has the following members:

0- EN_STG_STATUS_OK: operation completed successfully.

1- EN_STG_STATUS_NOT_STARTED: stg_start()[691] was not used or failed.

2- EN_STG_STATUS_OUT_OF_FILE_NUMBERS: need to open a file and there are no free file numbers left (possible only when STG_STORAGE_MEMORY is "1").

3- EN_STG_STATUS_WRONG_DEFINE: wrong #define value.

4- EN_STG_STATUS_WRONG_DESCRIPTOR: wrong descriptor file data.

5- EN_STG_STATUS_UNKNOWN: unknown setting or invalid setting number.

6- EN_STG_STATUS_INVALID_INDEX: invalid index (out-of-range).

7- EN_STG_STATUS_FAILURE: read failure or write failure (checksum error, hardware malfunction, etc.).

8- EN_STG_STATUS_INVALID: invalid setting value.

Library Procedures

In this section:

Stg_start()

Description: API procedure, starts the STG library, parses the descriptor file, restores all volatile (RAM) settings to default values by calling stg_restore_multiple (EN_STG_INIT_MODE_RAM_ONLY) 692

Syntax: **function stg_start() as en_stg_status_codes**

Returns: One of these en_stg_status_codes 690:
EN_STG_STATUS_OK,
EN_STG_STATUS_OUT_OF_FILE_NUMBERS,
EN_STG_STATUS_WRONG_DEFINE,
EN_STG_STATUS_WRONG_DESCRIPTOR,
EN_STG_STATUS_FAILURE

See Also: Getting Started 676

Details

MUST be called first, before any other procedure in this library is invoked, or EN_STG_STATUS_NOT_STARTED 690 will be returned by every other procedure you call.

When this procedure executes, callback_stg_post_set() 701 is called for each setting whose **Storage Location** is *Non-volatile*. See Editing Settings 673 for details.

Stg_check_all()

Description: API procedure, verifies each setting by reading its value through stg_sg() 698.

Syntax: **function stg_check_all(byref** problem_stg **as string) as en_stg_status_codes**

Returns: One of these en_stg_status_codes 690:
EN_STG_STATUS_OK, EN_STG_STATUS_NOT_STARTED,
EN_STG_STATUS_FAILURE, EN_STG_STATUS_INVALID

See Also: Verifying and Initializing Settings 677

Part	Description
problem_stg	After the procedure execution will contain the name of the problem setting, if any. Will be NULL if no problems were encountered.

Details

Execution stops once any problem for any setting is encountered. Thus, only one problem is reported.

Stg_get_def()

Description: API procedure, returns the default value for the specified setting's member.

Syntax: **function stg_get_def(byref** stg_name_or_num **as string,** index **as byte, byref** def_value **as string) as en_stg_status_codes**

Returns: One of these en_stg_status_codes 690: EN_STG_STATUS_OK, EN_STG_STATUS_NOT_STARTED, EN_STG_STATUS_UNKNOWN, EN_STG_STATUS_INVALID_INDEX.

In case of successful execution returns the default value for the specified setting's member through the def_value argument.

See Also: Verifying and Initializing Settings 677

Part	Description
stg_name_or_num	Setting name or number. If the supplied string does not start with a digit (0-9), then the string is interpreted as the name of the setting. If the string starts with a digit, then this will be interpreted as the setting number. Settings are numbered counting from 0, and in the order of their appearance in the setting configurator 670.
index	Setting member.
def_value	The default setting member's value will be returned through this argument.

Details

This function only accesses and reads the data from the configuration file 670.

Stg_restore_multiple()

Description: API procedure, initializes (sets the default values for) all or multiple settings.

Syntax: **function stg_restore_multiple(**init_mode **as en_stg_init_modes) as en_stg_status_codes**

Returns: One of these en_stg_status_codes 690: EN_STG_STATUS_OK, EN_STG_STATUS_NOT_STARTED, EN_STG_STATUS_FAILURE, EN_STG_STATUS_INVALID.

See Also: Verifying and Initializing Settings 677

Part	Description
init_mode	Initialization mode: Modes 0- EN_STG_INIT_MODE_NORMAL and 1- 1-EN_STG_INIT_MODE_OVERRIDE work on non-volatile and volatile settings. Modes 2- EN_STG_INIT_MODE_RAM_ONLY and 3-EN_STG_INIT_MODE_RAM_ONLY_OVERRIDE only work on volatile settings. Non-volatile settings are not checked at all. Modes 0- EN_STG_INIT_MODE_NORMAL and 2-EN_STG_INIT_MODE_RAM_ONLY cause conditional initialization of settings according to their Initialization Mode 673. If it is *Initialize Always*, then this setting is initialized unconditionally, if it is *Initialize When Invalid*, then this setting will be initialized only if its current value is invalid (EN_STG_STATUS_INVALID 690) or corrupted (EN_STG_STATUS_FAILURE 690). Modes 1- EN_STG_INIT_MODE_OVERRIDE and 3-EN_STG_INIT_MODE_RAM_ONLY_OVERRIDE cause initialization of settings regardless of their validity.
stg_timestamp 689 *variable*	When STG_TIMESTAMP 690 is 1, this variable should be preset with the desired timestamp prior to calling stg_restore_member(). All affected settings will receive the same timestamp.

Details

Default values 675 for setting members can be defined through the setting configurator 670.

When this procedure executes, callback_stg_post_set() 701 is called for each setting being initialized.

Stg_restore_member()

Description: API procedure, initializes (sets the default value for) the specified setting's member.

Syntax: **function stg_restore_member(byref** stg_name_or_num **as string,** index **as byte) as en_stg_status_codes**

Returns: One of these en_stg_status_codes 690:
EN_STG_STATUS_OK, EN_STG_STATUS_NOT_STARTED,
EN_STG_STATUS_UNKNOWN,
EN_STG_STATUS_INVALID_INDEX,
EN_STG_STATUS_FAILURE.

See Also: Verifying and Initializing Settings 677

Part	Description
stg_name_or_num	Setting name or number. If the supplied string does not start with a digit (0-9), then the string is interpreted as the name of the setting. If the string starts with a digit, then this will be interpreted as the setting number. Settings are numbered counting from 0, and in the order of their appearance in the setting configurator 670.
index	Setting member.
stg_timestamp 689 *variable*	When STG_TIMESTAMP 690 is 1, this variable should be preset with the desired timestamp prior to calling stg_restore_member().

Details

Default values 675 for setting members can be defined through the setting configurator 670.

The setting's member is restored regardless of the value of the initialization mode field in the descriptor file.

When this procedure executes, callback stg_post_set() 701 is called for the setting being initialized.

Stg_get_num_settings()

Description: API procedure, returns the number of settings defined in your project.

Syntax: **function stg_get_num_settings() as byte**

Returns: Total number of settings.

See Also: Using Setting Numbers 680

Details

Stg_get_num_members()

Description: API procedure, returns the number of members for the specified setting.

Syntax: **function stg_get_num_members(byref**

stg_name_or_num **as string, byref** num_members **as byte) as en_stg_status_codes**

Returns: One of these en_stg_status_codes 690: EN_STG_STATUS_OK, EN_STG_STATUS_NOT_STARTED, EN_STG_STATUS_UNKNOWN.

In case of successful execution returns the number of members through the num_members argument.

See Also: Working with Multi-value Settings 681

Part	Description
stg_name_or_num	Setting name or number. If the supplied string does not start with a digit (0-9), then the string is interpreted as the name of the setting. If the string starts with a digit, then this will be interpreted as the setting number. Settings are numbered counting from 0, and in the order of their appearance in the setting configurator 670.
num_members	The number of members for this setting will be returned through this argument.

Details

This function only accesses and reads the data from the configuration file 670.

Stg_find()

Description: API procedure, returns the number of the specified setting.

Syntax: **function stg_find(byref** stg_name_or_num **as string) as byte**

Returns: 0 if the specified setting wasn't found, or

setting number +1 if the setting was found. Settings are numbered counting from 0, and in the order of their appearance in the setting configurator 670.

See Also:

Part	Description
stg_name_or_num	Setting name or number. If the supplied string does not start with a digit (0-9), then the string is interpreted as the name of the setting. If the string starts with a digit, then this will be interpreted as the setting number.

Details

If stg_name_or_num argument starts with the digit, this function will simply return the setting number +1, but only if the setting with the specified number does exist (zero will be returned if it doesn't).

Stg_stype_get()

Description: API procedure, returns the type of the specified setting.

Syntax: **function stg_stype_get(byref** stg_name_or_num **as string) as byte**

Returns: 0 if the specified setting wasn't found, or

ASCII code of the character representing the setting type: `B` (byte), `W` (word), `S` (string), `D` (dot-decimal string).

See Also:

Part	Description
stg_name_or_num	Setting name or number. If the supplied string does not start with a digit (0-9), then the string is interpreted as the name of the setting. If the string starts with a digit, then this will be interpreted as the setting number.

Details

Stg_get()

Description: API procedure, reads (gets) the value of the specified setting's member; reports errors through callback_stg_error() 700.

Syntax: **function stg_get(byref** stg_name_or_num **as string,** index **as byte) as string**

Returns: Successful execution: the current value of the specified setting's member.

Some error, except EN_STG_STATUS_UNKNOWN 690 or EN_STG_STATUS_INVALID_INDEX 690: the default value of this setting's member, plus the callback_stg_error() is called, too. One of the following errors could be reported: EN_STG_STATUS_OK, EN_STG_STATUS_NOT_STARTED, EN_STG_STATUS_UNKNOWN, EN_STG_STATUS_INVALID_INDEX, EN_STG_STATUS_FAILURE, EN_STG_STATUS_INVALID.

When STG_TIMESTAMP 690 is 1, also returns the setting change timestamp through the stg_timestamp 689 global variable.

See Also: Using Stg_get() and Stg_set() 679

Part	Description
stg_name_or_num	Setting name or number. If the supplied string does not start with a digit (0-9), then the string is interpreted as the name of the setting. If the string starts with a digit, then this will be interpreted as the setting number. Settings are numbered counting from 0, and in the order of their appearance in the setting configurator 670.
index	Setting member.

Details

Callback_stg_error() 700 is invoked if any error is detected during this procedure's execution. One of the following errors could be reported: EN_STG_STATUS_OK, EN_STG_STATUS_NOT_STARTED, EN_STG_STATUS_UNKNOWN, EN_STG_STATUS_INVALID_INDEX, EN_STG_STATUS_FAILURE, EN_STG_STATUS_INVALID.

Callback_stg_pre_get() 700 is called when this procedure executes. This provides a "central location" for updating setting values before returning them throughstg_get ().

Stg_set()

Description: API procedure, writes (sets) the value of the specified setting's member; reports errors through callback_stg_error() 700.

Syntax: **sub stg_set(byref** stg_name_or_num **as string,** index **as byte, byref** stg_value **as string)**

See Also: Using Stg_get() and Stg_set() 679

Part	Description
stg_name_or_num	Setting name or number. If the supplied string does not start with a digit (0-9), then the string is interpreted as the name of the setting. If the string starts with a digit, then this will be interpreted as the setting number. Settings are numbered counting from 0, and in the order of their appearance in the setting configurator 670.
index	Setting member.
stg_value	New setting member's value to be set.
stg_timestamp 689 *variable*	When STG_TIMESTAMP 690 is 1, this global variable should be preset with the desired timestamp prior to calling stg_set().

Details

Callback_stg_error() 700 is invoked if any error is detected during this procedure's execution. One of the following errors could be reported: EN_STG_STATUS_OK, EN_STG_STATUS_NOT_STARTED, EN_STG_STATUS_UNKNOWN, EN_STG_STATUS_INVALID_INDEX, EN_STG_STATUS_FAILURE, EN_STG_STATUS_INVALID.

Callback_stg_post_set() 701 is called when this procedure executes. This provides a "central location" for responding to changing setting values.

Stg_sg()

Description: API procedure, writes (sets) or reads (gets) the specified setting's member; directly returns the execution result.

Syntax: **function stg_sg(byref** stg_name_or_num **as string,** index **as byte, byref** stg_value **as string,** op **as en_stg_rdwr) as en_stg_status_codes**

Returns: One of these en_stg_status_codes 690: EN_STG_STATUS_OK, EN_STG_STATUS_NOT_STARTED, EN_STG_STATUS_UNKNOWN, EN_STG_STATUS_INVALID_INDEX, EN_STG_STATUS_FAILURE, EN_STG_STATUS_INVALID.

For op= 1- EN_STG_SET, also indirectly returns the current value of the specified setting's member through the stg_value argument.

For op=0- EN_STG_GET, and when STG_TIMESTAMP 690 is 1, also returns the setting change timestamp through the stg_timestamp 689 global variable.

See Also: Using Stg_sg() 678

Part	Description
stg_name_or_num	Setting name or number. If the supplied string does not start with a digit (0-9), then the string is interpreted as the name of the setting. If the string starts with a digit, then this will be interpreted as the setting number. Settings are numbered counting from 0, and in the order of their appearance in the setting configurator 670.
index	Setting member.
stg_value	Setting value. For set operations, the desired new setting value should be supplied through this variable. For get operations, the current setting value will be found in this variable after the stg_sg() executes.
op	Operation: 0- EN_STG_GET: get (read) the setting value. 1- EN_STG_SET: set (write) the setting value.
stg_timestamp 689	For op=1- EN_STG_SET, and when STG_TIMESTAMP 690 is 1,

variable	this global variable should be preset with the desired timestamp prior to calling stg_sg().

Details

Callback_stg_pre_get() 700 is called when this procedure executes with op= 0- EN_STG_GET. This provides a central location for updating setting values before returning them through the stg_get(). Callback_stg_post_set() 701 is called for op= 1- EN_STG_SET. Again, this provides a "central location" for responding to changing setting values.

Callback_stg_error() 700 is never called from within this procedure, so make sure to analyse the status code returned by the stg_sg().

Stg_set_ts()

Description: API procedure, writes (sets) the timestamp for the specified setting. Only available when STG_TIMESTAMP 690 is 1.

Syntax: **function stg_set_ts(byref** stg_name_or_num **as string) as en_stg_status_codes**

See Also: Understanding Timestamps 681

Part	Description
stg_name_or_num	Setting name or number. If the supplied string does not start with a digit (0-9), then the string is interpreted as the name of the setting. If the string starts with a digit, then this will be interpreted as the setting number. Settings are numbered counting from 0, and in the order of their appearance in the setting configurator 670.
stg_timestamp 689 *variable*	This global variable should be preset with the desired timestamp prior to calling stg_set_ts().

Details

This procedure allows to modify the timestamp of the setting without affecting the values of setting members.

New timestamp is *also* set for the setting also when one of the following procedures executes:

- Stg_start() 691 -- for all settings whose Storage Location 673 is *Non-volatile*.
- Stg_restore_multiple() 692 -- for each setting being restored.
- Stg_restore_member() 693 -- for the setting being restored.
- Stg_set() 697 -- for the setting being written to.
- Stg_sg() 698 when op= 1- EN_STG_SET -- for the setting being written to.

Callback_stg_error()

Description: Callback procedure, informs of the error during the execution of stg_get() 696 or set_set() 697. Procedure body has to be created elsewhere in the project (externally with respect to the library).

Syntax: **sub callback_stg_error(byref** stg_name_or_num **as string,** index **as byte,** status **as en_stg_status_codes)**

Returns: ---

See Also: Using Stg_get() and Stg_set() 679

Part	Description
stg_name_or_num	Setting name or number. If the supplied string does not start with a digit (0-9), then the string should be interpreted as the name of the setting. If the string starts with a digit, then this should be interpreted as the setting number. Settings are numbered counting from 0, and in the order of their appearance in the setting configurator 670.
index	Setting member.
status	One of the en_stg_status_codes 690 codes indicating the type of the detected error.

Details

This procedure is invoked when the following procedures execute:

- Stg_get() 696
- Stg_set() 697

Notice that this procedure is never called during the execution of other procedures, even though many of them access settings!

The following procedures can't be called from within callback_stg_error() or recursion 66 error will be generated by the compiler:

- Stg_get() 696
- Stg_set() 697

Callback_stg_pre_get()

Description: Callback procedure, informs that the setting member's value is being read through one of the STG library's procedures; allows to update the setting member's value before it is returned by the STG library. Procedure body has to be created elsewhere in the project (externally with respect to the library).

Syntax: **sub callback_stg_pre_get(byref** stg_name_or_num **as string,** index **as byte, byref** stg_value **as string)**

Returns: ---

See Also: Using Pre-gets and Post-sets 683

Part	Description
stg_name_or_num	Setting name or number. If the supplied string does not start with a digit (0-9), then the string should be interpreted as the name of the setting. If the string starts with a digit, then this should be interpreted as the setting number. Settings are numbered counting from 0, and in the order of their appearance in the setting configurator 670.
index	Setting member.
stg_value	Current value of the specified setting's member. Leave the value unchanged if there is no need to update the setting. Change the value of this argument to have the STG library set the setting's member to this new value.

Details

This procedure is invoked when the following procedures execute:

- Stg_check_all() 691 -- for each setting being checked.
- Stg_get() 696 -- for the setting being read from.
- stg_sg(,,,EN_STG_GET) 698 -- for the setting being read from.

The following procedures can't be called from within callback_stg_pre_get() or recursion 66 error will be generated by the compiler:

- Stg_start() 691
- Stg_check_all() 691
- Stg_restore_multiple() 692
- Stg_restore_member() 693
- Stg_get() 696
- Stg_set() 697
- Stg_sg() 698

Callback_stg_post_set()

Description: Callback procedure, informs that the setting member's value is being written to through one of the STG library's procedures; allows to respond to the value change. Procedure body has to be created elsewhere in the project (externally with respect to the library).

Syntax: **sub callback_stg_post_set(byref** stg_name_or_num **as**

string, index **as byte,byref** stg_value **as string)**

Returns: ---

See Also: Using Pre-gets and Post-sets 683

Part	Description
stg_name_or_num	Setting name or number. If the supplied string does not start with a digit (0-9), then the string should be interpreted as the name of the setting. If the string starts with a digit, then this should be interpreted as the setting number. Settings are numbered counting from 0, and in the order of their appearance in the setting configurator 670.
index	Setting member.
stg_value	New setting member's value that has just been set.

Details

This procedure is invoked when the following procedures execute:

- Stg_start() 691 -- for all settings whose Storage Location 673 is *Non-volatile*.
- Stg_restore_multiple() 692 -- for each setting being restored.
- Stg_restore_member() 693 -- for the setting being restored.
- Stg_set() 697 -- for the setting being written to.
- stg_sg(,,,EN_STG_SET) 698 -- for the setting being written to.

The following procedures can't be called from within callback_stg_post_set() or recursion 66 error will be generated by the compiler:

- Stg_start() 691
- Stg_check_all() 691
- Stg_restore_multiple() 692
- Stg_restore_member() 693
- Stg_get() 696
- Stg_set() 697
- Stg_sg() 698

Callback_stg_vm_read()

Description: Callback procedure, should implement necessary code for reading a byte of data from "custom" volatile memory. Needed only when the **Custom RAM** option 670 is enabled. Procedure body has to be created elsewhere in the project (externally with respect to the library).

Syntax: **function callback_stg_vm_read(**address **as word) as byte**

Returns: ---

See Also: Operation Details 684

Part	Description
address	Memory address to read.

Details

Callback_stg_vm_write()

Description: Callback procedure, should implement necessary code for writing a byte of data to "custom" volatile memory. Invoked only when the **Custom RAM** option 670 is enabled. Procedure body has to be created elsewhere in the project (externally with respect to the library).

Syntax: **sub callback_stg_vm_write(**data_to_write **as byte,** address **as word)**

Returns: ---

See Also: Operation Details 684

Part	Description
data_to_write	Data byte to save to the "custom" volatile memory.
address	Memory address to write.

Details

WLN (Wi-Fi Association) Library

The WLN library works with the GA1000 add-on Wi-Fi module (see Programmable Hardware Manual). The library complements the Wi-Fi (wln. 536) object by providing the following:

- Persistent association with an access point of choice;
- Implementation of WPA1-PSK and WPA2-PSK security protocols (the wln. object doesn't fully support them per se);
- Support for access point switchover (roaming);
- Generation of keepalive messages to prevent disassociation.

The library is event-based and non-blocking -- it quietly runs in the background and

takes a minimal amount of CPU time.

Library Info

Supported platforms:

Any platform with Wi-Fi (wln. 536) interface.

Special case: WPA/WPA2 functionality is not supported on the EM500W 138 platform.

Files to include:

Wln.tbs, wln.tbh (from *current_library_set*\wln\trunk\).

Dependencies:

SOCK 664 library (only if you have #define WLN_WPA 1 721 or #define WLN_KEEP_ALIVE 1 721);

You may also need STG library 668 (for storing WPA/WPA2 parameters as described in Trying WPA 711 code example).

API procedures:

Use API procedures 576 to interact with the library.

wln_start() 724 -- commences attempts to bring up (boot) the Wi-Fi interface, find and associate with the specified wireless network, and then keep associated at all times.

wln_stop() 725 -- shuts down the Wi-Fi interface.

wln_change() 725 -- sets a different target wireless network for the WLN library.

wln_rescan() 726 -- starts the search for the specified access point in order to obtain its signal strength.

wln_wpa_mkey_get() 727 -- calculates the pre-shared master key for WPA1 and WPA2 security modes.

wln_check_association() 728 -- informs whether your device is currently associated with an access point.

Event procedures:

Call event procedures 576 from corresponding event handlers, as described here 619.

wln_proc_timer() 728 -- call it from the on_sys_timer() 533 event handler.

wln_proc_data() 728 -- call it from the on_sock_data_arrival() 489 event handler.

wln_proc_task_complete() 729 -- call it from the on_wln_task_complete() 565 event handler.

wln_proc_event() 729 -- call it from the on_wln_event() 565 event handler.

Callback procedures:

Implement the bodies of callback procedures 576 elsewhere in your project.

callback_wln_ok() 729 -- informs of the successful association with the target wireless network.

callback_wln_failure() 730 -- informs of the failure to find the target wireless network, associate with it, or maintain association.

callback_wln_pre_buffrq() 730 -- informs of the insufficient number of free buffer pages available for use by the library.

callback_wln_mkey_progress_update() 731 -- periodically called from wln_wpa_mkey_get() 727 to inform about the

progress of pre-shared key calculation.

callback_wln_rescan_result()[731] -- informs of the completion of the re-scanning initiated by wln_rescan()[726].

Required buffer space:

5 buffer pages minimum;

+1 additional buffer page if keepalive packets are enabled (see WLN_KEEP_ALIVE[721]);

+2 additional buffer pages if WPA/WPA2 support is enabled (see WLN_WPA[721]).

Step-by-step Usage Instructions

Minimal steps

1. Make sure you have the SOCK[664] library in your project (actually, you only need this if you have #define WLN_WPA 1[721] or #define WLN_KEEP_ALIVE 1[721]).
2. Add[577] **wln.tbs** and **wln.tbh** files to your project (they are in *current_library_set*\wln\trunk).
3. Add #define WLN_DEBUG_PRINT 1[721] to the defines section of the global.tbh file of your project. This way you will "see what's going on". Don't forget to remove this later, after you've made sure that the library operates as expected.

   - Some access points, notably CISCO devices, are so impatient during the WPA1/WPA2 handshake process, that printing debug info makes them timeout. If you notice that the WLN library is stuck in the endless association loop, and you are sure that your password is set correctly, then try disabling debug printing!

4. Define GA1000 interface line mapping. The following code is suggested (unless you built your own hardware, in which case study the Configuring Interface Lines[545] topic of the wln.[536] object's documentation):

```
#if PLATFORM_ID=EM500 or PLATFORM_ID=EM500W
      #define WLN_RESET_MODE 1 'there will be no dedicated reset, and all
other lines are fixed
#elif PLATFORM_ID=EM1206 or PLATFORM_ID=EM1206W
      #define WLN_CLK PL_IO_NUM_14
      #define WLN_CS PL_IO_NUM_15
      #define WLN_DI PL_IO_NUM_12
      #define WLN_DO PL_IO_NUM_13
      #define WLN_RST PL_IO_NUM_11
#else
      'EM1000, NB1010,...
      #define WLN_CLK PL_IO_NUM_53
      #define WLN_CS PL_IO_NUM_49
      #define WLN_DI PL_IO_NUM_52
      #define WLN_DO PL_IO_NUM_50
      #define WLN_RST PL_IO_NUM_51
```

```
#endif
```

5. Add **include "wln\trunk\wln.tbh"** to the includes section of the global.tbh file.
6. Add wln_proc_timer() 728 to the on_sys_timer() 533 event handler code (this library assumes that this event is generated twice per second).
7. Add wln_proc_data() 728 to the on_sock_data_arrival() 489 event handler code.
8. Add wln_proc_task_complete() 729 to the on_wln_task_complete() 565 event handler code. Pass *completed_task* argument of on_wln_task_complete() directly to wln_proc_task_complete().
9. Add wln_proc_event() 729 to the on_wln_event() 565 event handler code. Pass *wln_event* argument of on_wln_event() directly to wln_proc_event().
10. Create empty callback function 576 bodies (presumably in device.tbs): callback_wln_ok() 729, callback_wln_failure() 730, callback_wln_pre_buffrq() 730, callback_wln_rescan_result() 731. Hint: copy from declarations in wln.tbh or from our code example 708.
11. Add **ga1000fw.bin** file to your project. This is the firmware file for the GA1000 Wi-Fi add-on module. You can get it at http://tibbo.com/downloads/basic/firmware.html.
12. Call wln_start() 724 from somewhere. The no-brainer decision is to call from the on_sys_init() 533 event handler. Note that wln_start() may fail, so it is wise to check the returned status code.

All of the above is illustrated in a code example 708.

If you are going to use WPA/WPA2 security

1. Add #define WLN_WPA 1 721 to the defines section of the global.tbh file.
2. Add an empty callback function 576 body (presumably in device.tbs) for callback_wln_mkey_progress_update() 731.
3. Use wln_wpa_mkey_get() 727 to calculate a pre-shared master key, which is required for WPA/WPA2 security. Assign this function's output result to the *key* argument of the wln_start() 724 function.

Wln_wpa_mkey_get() 727 takes up to two minutes to complete. Trying WPA 711 code example not only shows how to deal with WPA/WPA2, but also how to avoid calculating the pre-shared master key repeatedly.

Operation Details

The operation of the WLN library can be observed in the debug mode 27 by adding #define WLN_DEBUG_PRINT 1 721 to the defines section of the global.tbh file of your project. A wealth of status information will then be printed in the console pane as the WLN library operates.

Once the wln_start() 724 is called, the WLN library will attempt to bring up the Wi-Fi interface, then find the specified wireless network and associate with it. The library uses active scanning (wln.activescan 556), which means that it can handle wireless networks that do not broadcast their SSIDs. The operation is persistent -- the library will repeatedly try to find the target wireless network and associate with it. There is no limit on the number of attempts. If the association with the wireless network is lost, the library will try to find the network and associate with it again. If

there are several access points with the same SSID (name) in range, the library will always choose the access point with the strongest signal.

A buffer space of 5 to 8 pages is required for the library to operate. During wln_start() execution, the library will check if the required buffer space is available and call callback_wln_pre_buffrq() 730 if more space is needed. Once successful association is achieved, callback_wln_ok() 729 is called. If scanning or association fails, callback_wln_failure() 730 is invoked to notify your system of the fact. Current association status can also be checked through wln_check_association() 728.

One of the most important features of the WLN library is the support for WPA/WPA2 security. The library will associate your device with any access point configured for WPA-PSK, WPA2-PSK, or mixed WPA-PSK/WPA2-PSK modes.

To connect to an access point configured for the mixed WPA-PSK/WPA2-PSK mode, use WPA2 on the device side. At the moment, WPA security will not work correctly on ad-hoc networks.

Using WPA1 or WPA2 involves calculating the pre-shared master key. Wln_wpa_mkey_get() 727 is provided for key calculation. This function takes up to two minutes to complete. Fortunately, you don't have to recalculate the key all the time -- see Trying WPA 711 code example for details. While wln_wpa_mkey_get() is executing, callback_wln_mkey_progress_update() 731 is called periodically to tell your application of the calculation progress.

The library can simultaneously maintain association with the selected access point and search for another access point with the same or different SSID (name). The search is initiated through wln_rescan() 726 and delivers the result via callback_wln_rescan_result() 731. The primary purpose of rescanning is to find out if there are access points in range that offer a better signal than the current access point. Your application can switch to another access point by calling wln_change() 725. For illustration see Roaming Between Access Points 716 code example.

Some access points disassociate devices after a period of inactivity. To prevent this, you can add #define WLN_KEEP_ALIVE 1 721 to your project. The WLN library will then periodically send out broadcast UDP datagrams (the period is defined by WLN_KEEP_ALIVE_TIMEOUT 721). This is normally enough to prevent access points from kicking your device out.

Calling wln_stop() 725 terminates library operation and shuts down the Wi-Fi interface. Related buffers are released at that time.

Code Examples

Projects in the Code Examples section are published on our website under the name "test_wln_lib".

In this section:

Step 1: The Simplest Example

This and other projects in the Code Examples section are published on our website under the name "test_wln_lib".

Let's start with a simple example of connecting to an access point named "TIBB1", which is configured for WEP64 security. The password is "12345678AB". All your code really has to do is start the "Wi-Fi engine" by calling wln_start() 724.

To illustrate the use of callback procedures, we set green status LED on in callback_wln_ok() 729. We turn this LED off in callback_wln_failure() 730.

Debug output we've got after running the code:

```
WLN> ---START---
WLN> ACTIVE SCAN for TIBB1
WLN> ASSOCIATE with TIBB1 (bssid: 192.63.14.197.236.216, ch: 11)
WLN> ---OK(associated in WEP64, WEP128, or no-security mode)---
```

You can test the "association persistence" by turning your access point off and on. You will see how the library will restore the association:

```
WLN> ERROR: disassociation (or link loss with the access point)
WLN> ACTIVE SCAN for TIBB1
WLN> ERROR: access point not found
WLN> ACTIVE SCAN for TIBB1
WLN> ERROR: access point not found
...
WLN> ACTIVE SCAN for TIBB1
WLN> ASSOCIATE with TIBB1 (bssid: 192.63.14.197.236.216, ch: 11)
WLN> ---OK(associated in WEP64, WEP128, or no-security mode)---
```

And here is the code itself...

global.tbh:

```
'DEFINES-------------------------------------------------------------
#define WLN_DEBUG_PRINT 1

#if PLATFORM_ID=EM500 or PLATFORM_ID=EM500W
     #define WLN_RESET_MODE 1 'there will be no dedicated reset, and all
```

```
other lines are fixed
#elif PLATFORM_ID=EM1206 or PLATFORM_ID=EM1206W
      #define WLN_CLK PL_IO_NUM_14
      #define WLN_CS PL_IO_NUM_15
      #define WLN_DI PL_IO_NUM_12
      #define WLN_DO PL_IO_NUM_13
      #define WLN_RST PL_IO_NUM_11
#else
      'EM1000, NB1010,...
      #define WLN_CLK PL_IO_NUM_53
      #define WLN_CS PL_IO_NUM_49
      #define WLN_DI PL_IO_NUM_52
      #define WLN_DO PL_IO_NUM_50
      #define WLN_RST PL_IO_NUM_51
#endif

'INCLUDES-------------------------------------------------------------
include "sock\trunk\sock.tbh" 'this lib is necessary for the WLN lib's
operation
include "wln\trunk\wln.tbh"
```

main.tbs:

```
include "global.tbh"

'=====================================================================
sub on_sys_init()
      if wln_start("TIBB1",WLN_SECURITY_MODE_WEP64,"12345678AB",
PL_WLN_DOMAIN_FCC)<>WLN_STATUS_OK then
            sys.halt
      end if
end sub

'---------------------------------------------------------------------
sub on_sys_timer()
      wln_proc_timer()
end sub

'---------------------------------------------------------------------
sub on_sock_data_arrival()
      wln_proc_data()
end sub

'---------------------------------------------------------------------
sub on_wln_task_complete(completed_task as pl_wln_tasks)
      wln_proc_task_complete(completed_task)
end sub

'---------------------------------------------------------------------
sub on_wln_event(wln_event as pl_wln_events)
      wln_proc_event(wln_event)
end sub
```

device.tbs:

```
include "global.tbh"
```

```
'====================================================================
sub callback_wln_ok()
     pat.play("G~",PL_PAT_CANINT)
end sub

'--------------------------------------------------------------------
sub callback_wln_failure(wln_state as en_wln_status_codes)
     pat.play("-",PL_PAT_CANINT)
end sub

'--------------------------------------------------------------------
sub callback_wln_pre_buffrq(required_buff_pages as byte)
end sub

'--------------------------------------------------------------------
sub callback_wln_rescan_result(current_rssi as byte, scan_rssi as byte,
different_ap as no_yes)
end sub
```

Step 2: Adding TCP Comms

This and other projects in the Code Examples section are published on our website under the name "test_wln_lib".

Now let's take the previous example and create a TCP socket that will accept connections over the Wi-Fi interface. We will make a simple loopback socket that echoes back the data.

May we suggest our very own I/O NINJA terminal/sniffer software as the tool for testing the loopback socket? Hint: use Connection Socket plugin.

Only main.tbs needs to be changed from the previous example 708. The socket is configured in on_sys_init() 533. Notice how we don't just pick a random socket number to use. We obtain the socket from the SOCK library 664 by calling sock_get() 667. This is very important! Once any part of your project uses the SOCK library (and WLN library does), all other parts of your project must do the same. Mess will ensue if you fail to handle socket numbers correctly.

In the code below, we first call wln_start() 724 and then configure the loopback socket. There is no significance to this order and you can reverse it. Same goes for on_sock_data_arrival() 489. There is no special reason to call wln_proc_data() 728 first, and handle the loopback socket next. You can reverse the order and everything will still work.

main.tbs:

```
include "global.tbh"

dim tcp_sock as byte

'====================================================================
sub on_sys_init()
     wln.ip="192.168.1.50" '<--- set suitable IP here

     if wln_start("TIBB1",WLN_SECURITY_MODE_WEP64,"12345678AB",
```

```
PL_WLN_DOMAIN_FCC)<>WLN_STATUS_OK then
            sys.halt
      end if

      'configure loopback socket
      tcp_sock=sock_get("TCP")
      sock.num=tcp_sock
      sock.rxbuffrq(1)
      sock.txbuffrq(1)
      sys.buffalloc
      sock.protocol=PL_SOCK_PROTOCOL_TCP
      sock.localportlist="1000"
      sock.allowedinterfaces="WLN"
      sock.inconmode=PL_SOCK_INCONMODE_ANY_IP_ANY_PORT
      sock.reconmode=PL_SOCK_RECONMODE_3
end sub

'-------------------------------------------------------------------------
sub on_sys_timer()
      wln_proc_timer()
end sub

'-------------------------------------------------------------------------
sub on_sock_data_arrival()
      wln_proc_data()

      'loopback tcp data
      if sock.num=tcp_sock then
            sock.setdata(sock.getdata(sock.txfree))
            sock.send
      end if
end sub

'-------------------------------------------------------------------------
sub on_wln_task_complete(completed_task as pl_wln_tasks)
      wln_proc_task_complete(completed_task)
end sub

'-------------------------------------------------------------------------
sub on_wln_event(wln_event as pl_wln_events)
      wln_proc_event(wln_event)
end sub
```

Step 3: Trying WPA

This and other projects in the Code Examples section are published on our website under the name "test_wln_lib".

OK then, now let's try to associate with an access point configured for WPA2-PSK security. Your can try WPA-PSK by yourself.

If your access point is configured for WPA-PSK/WPA2-PSK mode, select WPA2 on the device side.

With WPA, things get a bit tricker because of a pre-shared master key (PMK). This is the actual password used between your device and the access point. The key is

calculated from a "human" password (that you define) and the access point's SSID (name). Wln_wpa_mkey_get() does the math. The bad news is that it takes 2 minutes... yep, sorry, 8000 iterations involving sha1 need that long. The good news is that you only need to do this once after changing the password or SSID. You can then just save the resulting key for the future use. STG library comes handy for this.

Our project now has a new function -- connect_to_ap() -- that calls wln_start() . This function takes all the same arguments as wln_start(). Connect_to_ap() stores the SSID, human password, and pre-shared master key into the EEPROM. Wln_wpa_mkey_get() is only called if the SSID or human password change, or if the "PMK" setting is invalid. For the "PMK" setting to be valid, it must contain a 32-character string -- the PMK always has this length. Keeping the PMK in the EEPROM allows us to avoid recalculating it -- a huge time saver.
Callback_wln_mkey_progress_update() is called repeatedly from within wln_wpa_mkey_get() while the PMK calculation is in progress. We put pat.play there to assure the user that the system isn't dead.

Below is the debug output we've got when connecting to the access point named "TIBB1".

```
WLN> ---START---
WLN> ACTIVE SCAN for TIBB1
WLN> ASSOCIATE with TIBB1 (bssid: 192.63.14.197.236.216, ch: 11)
WLN> Pre-associated for WPA1-PSK or WPA2-PSK, starting handshake process
WLN> Pairwise handshake, process step 1 of 4
WLN> Pairwise handshake, process step 2 of 4
WLN> Pairwise handshake, process step 3 of 4
WLN> Pairwise handshake, process step 4 of 4
WLN> ---OK(associated in WPA2-PSK mode)---
```

- Some access points, notably CISCO devices, are so impatient during the WPA1/WPA2 handshaking process, that printing debug info makes them timeout. If you notice that the WLN library is stuck in the endless association loop, and you are sure that your password is set correctly, then try disabling debug printing! It may be the reason why the association fails.

The new iteration of our project is listed below. Notice how we set #define WLN_WPA 1 in global.tbh. This is necessary to enable WPA support.

We show one more trick in this example. Instead of having a listening TCP socket, we have a loopback socket that connects to a certain destination IP whenever there is a successful association. To ensure persistent connection, we put a polling code into on_sys_timer. Not the most beautiful of solutions, but gets the job done. The TCP connection is discarded whenever we get callback_wln_failiure().

Test outgoing connections by setting sock.targetip to your PC's address and using our I/O NINJA terminal/sniffer software. This software has a unique Listener Socket plugin that will allow your PC to receive incoming TCP connections.

global.tbh:

```
'DEFINES-----------------------------------------------------------
#define WLN_DEBUG_PRINT 1
#define WLN_WPA 1

#if PLATFORM_ID=EM500 or PLATFORM_ID=EM500W
      #define WLN_RESET_MODE 1 'there will be no dedicated reset, and all
other lines are fixed
#elif PLATFORM_ID=EM1206 or PLATFORM_ID=EM1206W
      #define WLN_CLK PL_IO_NUM_14
      #define WLN_CS PL_IO_NUM_15
      #define WLN_DI PL_IO_NUM_12
      #define WLN_DO PL_IO_NUM_13
      #define WLN_RST PL_IO_NUM_11
#else
      'EM1000, NB1010,...
      #define WLN_CLK PL_IO_NUM_53
      #define WLN_CS PL_IO_NUM_49
      #define WLN_DI PL_IO_NUM_52
      #define WLN_DO PL_IO_NUM_50
      #define WLN_RST PL_IO_NUM_51
#endif

'INCLUDES----------------------------------------------------------
include "sock\trunk\sock.tbh" 'this lib is necessary for the WLN lib's
operation
include "settings\trunk\settings.tbh" 'this lib is necessary to save pre-
shared master key
includepp "settings.xtxt"
include "wln\trunk\wln.tbh"

'DECLARATIONS------------------------------------------------------
declare function connect_to_ap(byref ap_name as string, security_mode as
pl_wln_security_modes, byref key as string, domain as pl_wln_domains) as
en_wln_status_codes
declare tcp_sock as byte
```

main.tbs:

```
include "global.tbh"

dim tcp_sock as byte

'====================================================================
sub on_sys_init()
      wln.ip="192.168.1.50"                                        '<---
set suitable IP here

      stg_start()

      if connect_to_ap("TIBB1",WLN_SECURITY_MODE_WPA2,"12345678",
PL_WLN_DOMAIN_FCC)<>WLN_STATUS_OK then
            sys.halt
      end if

      'configure loopback socket
      tcp_sock=sock_get("TCP")
      sock.num=tcp_sock
      sock.rxbuffrq(1)
```

```
      sock.txbuffrq(1)
      sys.buffalloc
      sock.protocol=PL_SOCK_PROTOCOL_TCP
      sock.targetinterface=PL_SOCK_INTERFACE_WLN
      sock.targetip="192.168.1.67"                              '<----- set
suitable target IP here
      sock.targetport=1000                                      '<----- set
suitable target port here
end sub

'-------------------------------------------------------------------
sub on_sys_timer()
      wln_proc_timer()

      'this code ensures persistent connection to the target IP
      if wln_check_association()=PL_WLN_ASSOCIATED then
            sock.num=tcp_sock
            if sock.statesimple=PL_SSTS_CLOSED then
                  sock.connect
            end if
      end if
end sub

'-------------------------------------------------------------------
sub on_sock_data_arrival()
      wln_proc_data()

      'loopback tcp data
      if sock.num=tcp_sock then
            sock.setdata(sock.getdata(sock.txfree))
            sock.send
      end if
end sub

'-------------------------------------------------------------------
sub on_wln_task_complete(completed_task as pl_wln_tasks)
      wln_proc_task_complete(completed_task)
end sub

'-------------------------------------------------------------------
sub on_wln_event(wln_event as pl_wln_events)
      wln_proc_event(wln_event)
end sub
```

device.tbs:

```
include "global.tbh"

'===================================================================
function connect_to_ap(byref ap_name as string, security_mode as
pl_wln_security_modes, byref key as string, domain as pl_wln_domains) as
en_wln_status_codes
      dim pmk as string(32)

      #if WLN_WPA
            if security_mode=WLN_SECURITY_MODE_WPA1 or
security_mode=WLN_SECURITY_MODE_WPA2 then
                  if stg_get("APN",0)<>ap_name or stg_get("PW",0)<>key or
stg_sg("PMK",0,pmk,EN_STG_GET)<>EN_STG_STATUS_OK then
```

```
                        'recalculate the key
                        pmk=wln_wpa_mkey_get(key,ap_name)
                        stg_set("PMK",0,pmk)
                        stg_set("APN",0,ap_name)
                        stg_set("PW",0,key)
                  else
                        pmk=stg_get("PMK",0) 'the key stays the same
                  end if
            else
                  pmk=key
            end if
      #else
            pmk=key
      #endif

      connect_to_ap=wln_start(ap_name,security_mode,pmk,domain)
end function

'-------------------------------------------------------------------
sub callback_wln_ok()
      pat.play("G~",PL_PAT_CANINT)
end sub

'-------------------------------------------------------------------
sub callback_wln_failure(wln_state as en_wln_status_codes)
      pat.play("-",PL_PAT_CANINT)
      sock.num=tcp_sock
      sock.discard
end sub

'-------------------------------------------------------------------
sub callback_wln_pre_buffrq(required_buff_pages as byte)
end sub

'-------------------------------------------------------------------
sub callback_wln_rescan_result(current_rssi as byte, scan_rssi as byte,
different_ap as no_yes)
end sub

'-------------------------------------------------------------------
sub callback_wln_mkey_progress_update(progress as byte)
      pat.play("B-**",PL_PAT_CANINT)
end sub

'-------------------------------------------------------------------
sub callback_stg_error(byref stg_name_or_num as string,index as byte,status
as en_stg_status_codes)
end sub

'-------------------------------------------------------------------
sub callback_stg_pre_get(byref stg_name_or_num as string,index as byte,byref
stg_value as string)
end sub

'-------------------------------------------------------------------
sub callback_stg_post_set(byref stg_name_or_num as string, index as byte,
byref stg_value as string)
end sub
```

settings.xtxt (the underlying configuration file):

```
>>APN E      S      1      0      32     A      ^
     Access point name
>>PW  E      S      1      0      32     A      ^
     Password
>>PMK E      S      1      32     32     A
12345678901234567890123456789012 Pre-shared master key

#define STG_DESCRIPTOR_FILE "settings.xtxt"
#define STG_MAX_NUM_SETTINGS 3
#define STG_MAX_SETTING_NAME_LEN 3
#define STG_MAX_SETTING_VALUE_LEN 32
```

Step 4: Roaming Between Access Points

This and other projects in the Code Examples section are published on our website under the name "test_wln_lib".

Finally, here is the coolest example of them all -- the code for a mobile device that roams between several access points that all have the same SSID (name). Typically, you use such a setup to create a larger wireless network. The example from the previous topic would work, but not very well if your device was mobile.

Say, you walk around with your GA1000-based gadget in your pocket. At boot, it would execute wln_start() 724, which would in turn search for the target wireless network, find several access points, and select the one with the strongest signal. The GA1000 would then latch onto this access point and won't let go until the signal gets so bad that the association is lost. The WLN library would then try to find a better access point to work with. Sounds like there is no problem, right?

Wrong! In practice, you will find that at a certain distance from the access point you get into the "grey area of communications" where the link between your gadget and the access point becomes spotty, but not quite bad enough for the association to fail. And, since the association won't quite fail, your device won't try to find a better access point to link to. Meanwhile, you will experience data delivery delays, unstable comms, etc.

Here is the solution. Additional code in on_sys_timer 533 periodically checks current signal strength (wln.rssi 566). Whenever it falls below a predefined constant RSSI_THRESHOLD, the code will do wln_rescan() 726 in an attempt to "find a better deal". Rescan result is checked in callback_wln_rescan_result() 731. If a different and better access point is found, wln_change() 725 will switch your device to using it.

It is important to understand why the program searches for better access points only when the signal drops below RSSI_THRESHOLD. Scanning is a disruptive process that temporarily interferes with data comms. There is no reason to scan if the current signal is perfect! Exactly what value should RSSI_THRESHOLD have depends on your device, antenna, etc. Find it through trial and error.

global.tbh:

```
'DEFINES---------------------------------------------------------------

#define WLN_DEBUG_PRINT 1
#define WLN_WPA 1
```

```
#if PLATFORM_ID=EM500 or PLATFORM_ID=EM500W
      #define WLN_RESET_MODE 1 'there will be no dedicated reset, and all
other lines are fixed
#elif PLATFORM_ID=EM1206 or PLATFORM_ID=EM1206W
      #define WLN_CLK PL_IO_NUM_14
      #define WLN_CS PL_IO_NUM_15
      #define WLN_DI PL_IO_NUM_12
      #define WLN_DO PL_IO_NUM_13
      #define WLN_RST PL_IO_NUM_11
#else
      'EM1000, NB1010,...
      #define WLN_CLK PL_IO_NUM_53
      #define WLN_CS PL_IO_NUM_49
      #define WLN_DI PL_IO_NUM_52
      #define WLN_DO PL_IO_NUM_50
      #define WLN_RST PL_IO_NUM_51
#endif

'INCLUDES-----------------------------------------------------------
include "sock\trunk\sock.tbh" 'this lib is necessary for the WLN lib's
operation
include "settings\trunk\settings.tbh" 'this lib is necessary to save pre-
shared master key
includepp "settings.xtxt"
include "wln\trunk\wln.tbh"

'DECLARATIONS-------------------------------------------------------
declare function connect_to_ap(byref ap_name as string, security_mode as
pl_wln_security_modes, byref key as string, domain as pl_wln_domains) as
en_wln_status_codes
declare tcp_sock as byte

const RESCAN_PERIOD=30                                              '<-----
change as needed
const RSSI_THRESHOLD=200                                      '<----- change
as needed
const AP_NAME="TIBB1"                                               '<-----
change as needed
const AP_PASSWORD="12345678"                                  '<----- change
as needed
const AP_SECURITY=WLN_SECURITY_MODE_WPA2             '<----- change as
needed
'use WLN_SECURITY_MODE_DISABLED, WLN_SECURITY_MODE_WEP64,
WLN_SECURITY_MODE_WEP128, WLN_SECURITY_MODE_WPA1, or WLN_SECURITY_MODE_WPA2
```

main.tbs:

```
include "global.tbh"

'-------------------------------------------------------------------
dim tcp_sock as byte
dim rescan_tmr as byte

'===================================================================
sub on_sys_init()
      wln.ip="192.168.1.50"                                         '<-----
set suitable IP here

      stg_start()
```

```
      if connect_to_ap(AP_NAME,AP_SECURITY,AP_PASSWORD,PL_WLN_DOMAIN_FCC)
<>WLN_STATUS_OK then
            sys.halt
      end if

      'configure loopback socket
      tcp_sock=sock_get("TCP")
      sock.num=tcp_sock
      sock.rxbuffrq(1)
      sock.txbuffrq(1)
      sys.buffalloc
      sock.protocol=PL_SOCK_PROTOCOL_TCP
      sock.targetinterface=PL_SOCK_INTERFACE_WLN
      sock.targetip="192.168.1.67"                              '<----- set
suitable target IP here
      sock.targetport=1000                                      '<----- set
suitable target port here
end sub

'--------------------------------------------------------------------
sub on_sys_timer()
      wln_proc_timer()

      if wln_check_association()=PL_WLN_ASSOCIATED then
            'this code ensures persistent connection to the target IP
            sock.num=tcp_sock
            if sock.statesimple=PL_SSTS_CLOSED then
                  sock.connect
            end if

            'this code handles access point roaming
            if rescan_tmr>0 then
                  rescan_tmr=rescan_tmr-1
                  if rescan_tmr=0 then
                        rescan_tmr=RESCAN_PERIOD
                        if wln.rssi<=RSSI_THRESHOLD then
                              wln_rescan(AP_NAME)
                        end if
                  end if
            end if
      else
            rescan_tmr=RESCAN_PERIOD
      end if
end sub

'--------------------------------------------------------------------
sub on_sock_data_arrival()
      wln_proc_data()

      'loopback tcp data
      if sock.num=tcp_sock then
            sock.setdata(sock.getdata(sock.txfree))
            sock.send
      end if
end sub

'--------------------------------------------------------------------
sub on_wln_task_complete(completed_task as pl_wln_tasks)
      wln_proc_task_complete(completed_task)
end sub
```

```
'-------------------------------------------------------------------
sub on_wln_event(wln_event as pl_wln_events)
      wln_proc_event(wln_event)
end sub
```

device.tbs:

```
include "global.tbh"

'===================================================================
function connect_to_ap(byref ap_name as string, security_mode as
pl_wln_security_modes, byref key as string, domain as pl_wln_domains) as
en_wln_status_codes
      dim pmk as string(32)

      #if WLN_WPA
            if security_mode=WLN_SECURITY_MODE_WPA1 or
security_mode=WLN_SECURITY_MODE_WPA2 then
                  if stg_get("APN",0)<>ap_name or stg_get("PW",0)<>key or
stg_sg("PMK",0,pmk,EN_STG_GET)<>EN_STG_STATUS_OK then
                        'recalculate the key
                        pmk=wln_wpa_mkey_get(key,ap_name)
                        stg_set("PMK",0,pmk)
                        stg_set("APN",0,ap_name)
                        stg_set("PW",0,key)
                  else
                        pmk=stg_get("PMK",0) 'the key stays the same
                  end if
            else
                  pmk=key
            end if
      #else
            pmk=key
      #endif

      connect_to_ap=wln_start(ap_name,security_mode,pmk,domain)
end function

'-------------------------------------------------------------------
sub callback_wln_ok()
      pat.play("G~",PL_PAT_CANINT)
end sub

'-------------------------------------------------------------------
sub callback_wln_failure(wln_state as en_wln_status_codes)
      pat.play("-",PL_PAT_CANINT)
      sock.num=tcp_sock
      sock.discard
end sub

'-------------------------------------------------------------------
sub callback_wln_pre_buffrq(required_buff_pages as byte)
end sub

'-------------------------------------------------------------------
sub callback_wln_rescan_result(current_rssi as byte, scan_rssi as byte,
different_ap as no_yes)
      dim s as string(32)
```

```
      if different_ap=NO or scan_rssi<current_rssi or scan_rssi-
current_rssi<5 then
            exit sub
      end if

      select case AP_SECURITY
      case WLN_SECURITY_MODE_DISABLED:
            s=""
      case WLN_SECURITY_MODE_WEP64,WLN_SECURITY_MODE_WEP128:
            s=AP_PASSWORD
      case else:
            s=stg_get("PMK",0)
      end select

      wln_change(AP_NAME,AP_SECURITY,s)
end sub

'--------------------------------------------------------------------
sub callback_wln_mkey_progress_update(progress as byte)
      pat.play("B-**",PL_PAT_CANINT)
end sub

'--------------------------------------------------------------------
sub callback_stg_error(byref stg_name_or_num as string,index as byte,status
as en_stg_status_codes)
end sub

'--------------------------------------------------------------------
sub callback_stg_pre_get(byref stg_name_or_num as string,index as byte,byref
stg_value as string)
end sub

'--------------------------------------------------------------------
sub callback_stg_post_set(byref stg_name_or_num as string, index as byte,
byref stg_value as string)
end sub
```

settings.xtxt (the underlying configuration file):

```
>>APN E      S      1      0      32     A      ^
      Access point name
>>PW  E      S      1      0      32     A      ^
      Password
>>PMK E      S      1      32     32     A
12345678901234567890123456789012 Pre-shared master key

#define STG_DESCRIPTOR_FILE "settings.xtxt"
#define STG_MAX_NUM_SETTINGS 3
#define STG_MAX_SETTING_NAME_LEN 3
#define STG_MAX_SETTING_VALUE_LEN 32
```

Library Defines (Options)

Any of the options below look cryptic? Read the Operation Details 706 section.

WLN_DEBUG_PRINT (default= 0)

0- no debug information.

1- print debug information into the output pane. Debug printing only works when the project is in the debug mode 27. However, still set this option to 0 for release, as this will save memory and code space.

- Some access points, notably CISCO devices, are so impatient during the WPA1/WPA2 handshake process, that printing debug info makes them timeout. If you notice that the WLN library is stuck in the endless association loop, and you are sure that your password is set correctly, then try disabling debug printing!

WLN_RESET_MODE (default= 0)

0- dedicated RST line. Remember to define this line's mapping with WLN_RST.

1- CS and CLK lines are to generate reset. Use the reset generation circuit as shown on diagram B in Connecting GA1000 201 topic.

WLN_RST (default= PL_IO_NULL)

RST line mapping. Only relevant when WLN_RESET_MODE= 0.

WLN_CS (default= PL_IO_NULL)

CS line mapping.

WLN_DI (default= PL_IO_NULL)

DI line mapping.

WLN_DO (default= PL_IO_NULL)

DO line mapping.

WLN_CLK (default= PL_IO_NULL)

CLK line mapping.

WLN_KEEP_ALIVE (default= 0)

0- do not send keepalive UDP datagrams (saves code and memory space).

1- send keepalive UDP datagrams to prevent disassociation from the access point.

WLN_KEEP_ALIVE_TOUT (default= 120)

Time interval, in 1/2 second increments, between keepalive UDP datagrams. This assumes that on_sys_timer 533 runs at 0.5 second intervals.

Only relevant when WLN_KEEP_ALIVE=1.

WLN_WPA (default= 0)

0- disable WPA1/WPA2 support (saves code and memory space).

1- enable WPA1/WPA2 support.

En_wln_status_codes

Several procedures in the library utilize the en_wln_status_codes enum. This enum has the following members:

0- WLN_STATUS_OK: Success.

1- WLN_STATUS_OUT_OF_SOCKETS: No free sockets available for the library to operate.

2- WLN_STATUS_INSUFFICIENT_BUFFER_SPACE: Insufficient number of buffer pages available and the call to callback_wln_pre_buffrq() 730 failed to cure the problem.

3- WLN_STATUS_MISSING_FIRMWARE_FILE: You forgot to add 132 "ga1000fw.bin" file to your project.

4- WLN_STATUS_BOOT_FAILURE: Wi-Fi hardware could not be booted (improperly connected? turned off? ...).

5- WLN_STATUS_INVALID_SECURITY_MODE: Incorrect security mode specified in the security_mode argument of the wln_start() 724 function.

6- WLN_STATUS_INVALID_WEP_KEY: WEP64 or WEP128 was specified when calling to wln_start() 724, and the length of the key argument is incorrect. The length must be 10 HEX characters (characters `0`~`9`, `A`~`F`, or `a`~`f`) for WEP64, and 26 HEX characters for WEP128.

7- WLN_STATUS_SCANNING_FAILURE: Failed to discover the target wireless network.

8- WLN_STATUS_ASSOCIATION_FAILURE: Failed to associate with the target

wireless network.

9- WLN_STATUS_DISASSOCIATION: Wi-Fi interface got disassociated from the target wireless network.

10- WLN_STATUS_UNEXPECTED_ERROR: Well, this is something... unexpected :).

11- WLN_STATUS_NOT_STARTED: The procedure couldn't be executed because the library hasn't been started (call wln_start() 724 first).

12- WLN_STATUS_BUSY: The procedure couldn't be executed because the library is busy. Try again after a short delay.

Library Procedures

In this section:

Wln_get_info()

Description:	API procedure, returns library-specific information according to the requested information element.
Syntax:	**function wln_get_info(**info_element **as wln_info_elements, byref** extra_data **as string) as string**
Returns:	Requested data in *string* form
See Also:	About _get_info() API Functions 577

Part	Description
info_element	Information element being requested: 0- WLN_INFO_ELEMENT_REQUIRED_BUFFERS: total number

	of buffer pages required for the library to operate.
extra_data	When info_element= WLN_INFO_ELEMENT_REQUIRED_BUFFERS, set this argument to the desired security mode. This is necessary because the security mode affects the amount of required buffer space. Available security modes are defined by the pl_wln_security_modes enum (see security_mode argument of wln_start 724). Since the extra_data argument is of the string type, you have to convert the security mode into a string form, i.e. like this: str(WLN_SECURITY_MODE_WPA1).

Details

Wln_start()

Description: API procedure, commences attempts to bring up (boot) the Wi-Fi interface, find and associate with the specified wireless network, and then keep associated at all times.

Syntax: **wln_start(byref** ap_name **as string,** security_mode **as pl_wln_security_modes, byref** key **as string,** domain **as pl_wln_domains) as en_wln_status_codes**

Returns: One of these en_wln_status_codes 722: WLN_STATUS_OK, WLN_STATUS_OUT_OF_SOCKETS, WLN_STATUS_INSUFFICIENT_BUFFER_SPACE, WLN_STATUS_MISSING_FIRMWARE_FILE, WLN_STATUS_BOOT_FAILURE, WLN_STATUS_INVALID_SECURITY_MODE, WLN_STATUS_INVALID_WEP_KEY

See Also: Step-by-step Usage Instructions 705, Operation Details 706, The Simplest Example 708

Part	Description
ap_name	The SSID (name) of the wireless network to associate with.
security_mode	One of pl_wln_security_modes: 0- WLN_SECURITY_MODE_DISABLED, 1- WLN_SECURITY_MODE_WEP64, 2- WLN_SECURITY_MODE_WEP128, 3- WLN_SECURITY_MODE_WPA1 (will be accepted only if WLN_WPA 721 is defined as 1), 4- WLN_SECURITY_MODE_WPA2 (will be accepted only if WLN_WPA 721 is defined as 1)

key	For WEP64 mode, a string of 10 HEX characters (characters `0`~`9`, `A`~`F`, or `a`~`f`). For WEP128 mode, a string of 26 HEX characters. For WPA1-PSK and WPA2-PSK modes, this is the pre-shared master key, which is not the "human" password. The key can be calculated using wln_wpa_mkey_get() 727.
domain	Defines the set of channels on which the Wi-Fi interface will operate. Possible values are: 0- PL_WLN_DOMAIN_FCC (default): FCC domain (US). Allowed channels: 1-11. 1- PL_WLN_DOMAIN_EU: European Union. Allowed channels: 1-13. 2- PL_WLN_DOMAIN_JAPAN: Japan. Allowed channels: 1-14. 3- PL_WLN_DOMAIN_OTHER: All other countries. Allowed channels: 1-11.

Details

WLN library operation is non-blocking. Wln_start() will quickly return control to your application and the rest of the library's operation will continue in the background.

The WLN library uses active scanning 550 (wln.activescan 556) and, therefore, will be able to find access points that do not broadcast their SSIDs.

If several access points with the same SSID are in range, the WLN library will automatically select the access point with the strongest signal.

Wln_stop()

Description: API procedure, shuts down the Wi-Fi interface.

Syntax: **sub wln_stop()**

Returns: ---

See Also: Step-by-step Usage Instructions 705, Operation Details 706

Details

Wi-Fi interface shutdown includes the hardware reset of the GA1000 module.

Wln_change()

Description: API procedure, sets a different target wireless network for the WLN library.

Syntax: **function wln_change(byref** ap_name **as string,** security_mode **as pl_wln_security_modes, byref** key **as string) as en_wln_status_codes**

Returns: One of these en_wln_status_codes[722]: WLN_STATUS_OK, WLN_STATUS_NOT_STARTED, WLN_STATUS_INVALID_SECURITY_MODE, WLN_STATUS_INVALID_WEP_KEY, WLN_STATUS_BUSY

See Also: Step-by-step Usage Instructions[705], Operation Details[706], Roaming Between Access Points[716]

Part	Description
ap_name	New SSID (name) of the wireless network to associate with.
security_mode	One of pl_wln_security_modes: 0- WLN_SECURITY_MODE_DISABLED, 1- WLN_SECURITY_MODE_WEP64, 2- WLN_SECURITY_MODE_WEP128, 3- WLN_SECURITY_MODE_WPA1 (will be accepted only if WLN_WPA[721] is defined as 1), 4- WLN_SECURITY_MODE_WPA2 (will be accepted only if WLN_WPA[721] is defined as 1)
key	For WEP64 mode, a string of 10 HEX characters (characters `0`~`9`, `A`~`F`, or `a`~`f`). For WEP128 mode, a string of 26 HEX characters. For WPA1-PSK and WPA2-PSK modes, this is the pre-shared master key, which is not the "human" password. The key can be calculated using wln_wpa_mkey_get()[727].

Details

WLN library operation is non-blocking. Wln_change() will quickly return control to your application and the rest of the library's operation will continue in the background.

The WLN library uses active scanning[550] (wln.activescan[556]) and, therefore, will be able to find access points that do not broadcast their SSIDs.

If several access points with the same SSID are in range, the WLN library will automatically select the access point with the strongest signal.

You can't use this function before calling wln_start()[724].

Wln_rescan()

Description: API procedure, starts the search for the specified access point in order to obtain its signal strength.

Syntax: **function wln_rescan(byref** ap_name **as string) as en_wln_status_codes**

Returns: One of these en_wln_status_codes[722]: WLN_STATUS_OK, WLN_STATUS_NOT_STARTED, WLN_STATUS_BUSY

See Also: Step-by-step Usage Instructions 705, Operation Details 706, Roaming Between Access Points 716

Part	Description
ap_name	The SSID (name) of the access point to search for.

Details

The purpose of wln_rescan() is to help you find out if there is an access point in range that has a "better signal". This function starts the process, which then proceeds in the background without stalling your application. Once the scanning is finished, callback_wln_rescan_result() 731 is called.

If several access points with the same SSID are in range, callback_wln_rescan_result() 731 will return the signal strength for the access point with the strongest signal.

The WLN library uses active scanning 550 (wln.activescan 556) and, therefore, will be able to find access points that do not broadcast their SSIDs.

You can't use this function before calling wln_start() 724.

Wln_wpa_mkey_get()

Description: API procedure, calculates the pre-shared master key for WPA1 and WPA2 security modes.

Syntax: **function wln_wpa_mkey_get(byref** password **as string, byref** ssid **as string) as string**

Returns: Pre-shared master key string

See Also: Step-by-step Usage Instructions 705, Operation Details 706, Trying WPA 711

Part	Description
password	The "human" password for the access point.
ssid	SSID (name) of the access point.

Details

In WPA1-PSK and WPA2-PSK security schemes, the password you set and the SSID of the access point are used to calculate a so-called pre-shared master key, which is the actual password used in communications between your device and the access point. The password and the pre-shared master key are not to be confused: the password is the string you manually enter (on the setup page) when configuring the access point. The pre-shared master key is a binary string that you never get to handle directly.

On Tibbo devices, pre-shared master key calculation takes up to 2 minutes. Fortunately, the key needs only be calculated when there is the password or SSID

change.

Callback_wln_mkey_progress_update() 731 is called periodically from within wln_wpa_mkey_get(). This enables your application to indicate the calculation progress to the user (i.e. by drawing a progress bar on the LCD screen).

Wln_check_association()

Description: API procedure, informs whether your device is currently associated with an access point.

Syntax: **function wln_check_association() as pl_wln_association_states**

Returns: 0- PL_WLN_NOT_ASSOCIATED.

1- PL_WLN_ASSOCIATED.

See Also: Step-by-step Usage Instructions 705, Operation Details 706, Trying WPA 711

Details

If you are running with no security or WEP security, then using this API call is the same as checking wln.associationstate 558. For WPA1-PSK and WPA2-PSK security modes, the mere fact that wln.associationstate returns 1- PL_WLN_ASSOCIATED does not indicate association success. This is because WPA1-PSK and WPA2-PSK additionally involve a complex exchange of "handshake" messages. Only after the handshake is completed that the association is finished and wln_check_association () will return 1- PL_WLN_ASSOCIATED.

Wln_proc_timer()

Function: Event procedure, call it from the on_sys_timer() 533 event handler.

Syntax: **sub wln_proc_timer()**

Returns: ---

See Also: Step-by-step Usage Instructions 705, Operation Details 706, The Simplest Example 708

Details

Wln_proc_data()

Function: Event procedure, call it from the on_sock_data_arrival() 489 event handler.

Syntax: **sub wln_proc_data()**

Returns: ---

See Also: Step-by-step Usage Instructions 705, Operation Details 706, The Simplest Example 708

Details

Wln_proc_task_complete()

Function: Event procedure, call it from the on_wln_task_complete() 565 event handler.

Syntax: **sub wln_proc_task_complete()**

Returns: ---

See Also: Step-by-step Usage Instructions 705, Operation Details 706, The Simplest Example 708

Details

Wln_proc_event()

Function: Event procedure, call it from the on_wln_event() 565 event handler.

Syntax: **sub wln_proc_event()**

Returns: ---

See Also: Step-by-step Usage Instructions 705, Operation Details 706, The Simplest Example 708

Details

Callback_wln_ok()

Description: Callback procedure, informs of the successful association with the target wireless network. Procedure body has to be created elsewhere in the project (externally with respect to the library).

Syntax: **sub callback_wln_ok()**

Returns: ---

See Also: Step-by-step Usage Instructions 705, Operation Details 706, The Simplest Example 708

Details

Callback_wln_failure()

Description: Callback procedure, informs of the failure to find the target wireless network, associate with it, or maintain association. Procedure body has to be created elsewhere in the project (externally with respect to the library).

Syntax: **sub callback_wln_failure(**wln_state **as en_wln_status_codes)**

Returns: ---

See Also: Step-by-step Usage Instructions 705, Operation Details 706, The Simplest Example 708

Part	Description
wln_state	One of these en_wln_status_codes 722: WLN_STATUS_SCANNING_FAILURE, WLN_STATUS_ASSOCIATION_FAILURE, WLN_STATUS_DISASSOCIATION, WLN_STATUS_UNEXPECTED_ERROR.

Details

Callback_wln_pre_buffrq()

Description: Callback procedure, informs of the insufficient number of free buffer pages available for use by the library. Procedure body has to be created elsewhere in the project (externally with respect to the library).

Syntax: **sub callback_wln_pre_buffrq(**required_buff_pages **as byte)**

Returns: ---

See Also: Step-by-step Usage Instructions 705, Operation Details 706, The Simplest Example 708

Part	Description
required_buffer_pages	The number of additional buffer pages that the WLN library needs to operate. Your application must free up at least this

number of buffer pages within callback_wln_pre_buffrq() or wln_start() 724 will fail with the WLN_STATUS_INSUFFICIENT_BUFFER_SPACE status code 722 .

Details

This procedure will be called only if there are not enough buffer pages available.

Callback_wln_mkey_progress_update()

Description: Callback procedure, informs of the pre-shared master key calculation progress. Procedure body has to be created elsewhere in the project (externally with respect to the library).

Syntax: **sub callback_wln_mkey_progress_update(**progress **as byte)**

Returns: ---

See Also: Step-by-step Usage Instructions 705, Operation Details 706, Trying WPA 711

Part	Description
progress	Current progress, in %.

Details

Callback_wln_mkey_progress_update() is called periodically from within wln_wpa_mkey_get() 727. This enables your application to indicate the calculation progress to the user (i.e. by drawing a progress bar on the LCD screen).

Callback_wln_rescan_result()

Description: Callback procedure, informs of the completion of the re-scanning initiated by wln_rescan() 726. Procedure body has to be created elsewhere in the project (externally with respect to the library).

Syntax: **sub callback_wln_rescan_result(**current_rssi **as byte,** scan_rssi **as byte,** different_ap **as no_yes)**

Returns: ---

See Also: Step-by-step Usage Instructions 705, Operation Details 706, Roaming Between Access Points 716

Part	Description

current_rssi	Signal strength for the access point your device is currently associated with. This argument will be 0 if your device is not associated with any access point.
scan_rssi	Signal strength for the access point you searched for with wln_rescan() 726. This argument will be 0 if the specified access point wasn't found.
different_ap	Indicates whether or not wln_rescan() has found the same access point as the one your device is currently associated with, or a different access point: 0- NO: the same access point. 1- YES: different access point.

Details

The WLN library determines whether the access point is the same or different by comparing BSSIDs ("MAC addresses") of the access point your device is currently associated with and the access point that was discovered by wln_rescan() 726. This comes handy for your application in case several access points in range have the same SSID (name).

If several access points with the same SSID are in range, callback_wln_rescan_result() will return the signal strength for the access point with the strongest signal.

Update History (for this Manual)

01SEP2012

- Documented DS1100 164, DS1101W 168, and DS1102W 174 platforms.
- Almost completely rewritten the Common Library Info 575 section.
- Documented the AggreGate (AGG) library 580.
- Significantly updated the Setting (SRG) library 668 documentation.
- Documented changes to the gprs_start() 652 of the GPRS library 645 (new **apn** argument).

22DEC2011 release

- Documented ppp. object 366 and GPRS library 645.
- Documented fd.copyfirmwarelzo, and edited Upgrading the Firmware/Application 243.

21NOV2011 release

- Documented ssi. 512 object.

18OCT2011 release

- Documented STG library 668 improvements:

- It is now possible to use escape sequences (i.e. "\xA5") in default value strings -- see *Setting Descriptor File Format*.

- Stg_restore_multiple() 692 now has now have 4 operation modes.
- **STG_REDUNDANCY** now offers three choices.
- new **STG_RAM_TYPE** define added.
- new Callback_stg_vm_read 702 and Callback_stg_vm_write 703 topics.

- Documented changes in the .romfile 370 object, which now supports files over 64KB.
- New topic added: About _get_info() API Functions 577.
- Documented new API procedures: pppoe_get_info() 660, wln_get_info() 723, dhcp_get_info() 636.
- Edited Using HTTP 461 -- SWF files are now supported!

17AUG2011 release

- Documented WLN library 703.
- Updated wln. 536 object documentation (many small edits). Notable changes:
- Removed "Known Limitations" from wln. 536 object documentation -- these limitations do not exist any more.
- Documented wln.activescan 556 and expanded Scanning for Wi-Fi Networks 550.
- Expanded on WPA security (for example, see Setting WPA Mode and Keys 553).
- Documented two new API calls of the STG library 668: stg_find() 695 and stg_stype_get() 696.
- Updated code examples 622 in the DPHP library 618.
- Updated the sample project 686 of the STG library 668.
- Fixed error in Serial Settings 390 topic code sample.
- Added Includepp Statement 96 topic.

20JUL2011 release

- Updated wln. 536 object documentation:
- Dropped "Migrating From the WA1000" and "Rebooting" topics.
- Added "Known Limitations", Setting Wi-Fi Security 552, Setting WPA Mode and Key 553, .Scanresultwpainfo R/O Property 569.
- Reedited the entire text.
- Changed Our Language Philosophy 4 topic to reflect support for floating point variables.
- Documented aes128dec 205, aes128enc 206, rc4 222, strand 225, stror 227, and strxor 228 functions.

23JUN2011 release

- Fixed sock.allowedinterfaces 474 topic to include new values (PPPoE and PPP)
- New topic: sock.availableinterfaces 475

20JUN2011 release

- Documented pppoe. object 369 and PPPOE library 655.
- New topics: Protecting Your Device with a Password 41, Setup (MD) Button (Line) 201.

- Updated: Compiling a Final Binary 15, The Structure of a Project 16, Creating, Opening, and Saving Projects 17, Adding, Removing, and Saving Files 18, Project Settings 38, Device Explorer 39, Special Configuration Section of the EEPROM 197.
- Corrected: Connecting GA1000 201 (diagram C was wrong).
- Corrected: .ip 360 Property (of the net. 358 object).

30MAY2011 release

- EM500 138 platform change: fd. 236 and wln. 536 objects are now supported.
- Fd. 236 object docs change: there is a new 16-PL_FD_STATUS_FLASH_NOT_DETECTED error code 239, and fd.format 276, fd.formatj 277, fd.mount 283, fd.getsector 281, fd.setsector 290 may return it.
- New topics: Connecting External Flash IC 143, Connecting GA1000 201.
- Many small changes in Platform Specifications 138 (especially for the EM500 138 module).

12MAY2011 release

- Removed "EM202" platform documentation, it is no longer supported. We recommend using EM1206 158 or EM1206W 158 instead.
- Cleaned up platform specifications 138 -- many things rearranged:

- Most specs for each platform now fit in a single topic;

- The folder with common information 191 has been expanded and rewritten.

- Updated fd. object documentation:

- Using Checksums 242, Fd.checksum 268 -- several major corrections, original text contained many factual errors!

- Documented transactions 259 (new feature).

- Most topics were edited and updated.

09FEB2011 release

- Released all-new Libraries 572 section, with SOCK 664, FILENUM 641, STG 668 and DHCP 618 library documentation.
- Updated TIDE topic Adding, Removing and Saving Files 18 with new screenshots and "add existing files" option.

10OCT2010 release

- Corrected typographical errors in topics Lbin Function 214, Lhex Function 216.

04OCT2010 release

- Corrected typographical errors in topics Instr Function 214, .Insert Function 213, Date Function 208, Mincount Function 220.

27JUL2010 release

- Documented new feature, incremental project uploads 26.

15JUNE2010 release

- Corrected EM1206 PLL status on boot (PLL ON by default, no PE pin) under Platform-dependent Programming Information.
- Corrected inter-pulse gaps in Wiegand mode 383 for the .ser object (2mS instead of 20mS)

02JUN2010 release

- Documented EM500 138 platform.
- Added Enum pl_io_port_num topic to each platform's description.
- Added Device serial number section to the Platform-dependent Programming Information topic of each platform.
- Changed the Serial Number 530 topic -- it now refers to the Device serial number section (see above).
- Edited Serialnum 534 R/O Property and Setserialnum 535 method topics.

29JUL2009 release

- Uncluttered platform documentation -- made these topics "common":
  - Supported Variable Types (T1000-based Devices) 192;
  - Supported Functions (T1000-based Devices) 192;
  - LED Signals 200;
  - Debug Communications 204;
  - Project Settings Dialog.
- Merged EM1000 and EM1000W platform documentation under a single manual -- EM1000 and EM1000W Platforms 143.
- Added EM1202W platform documentation into the EM1000 platform docs, renamed the section into EM1202 and EM1202W Platforms 151.
- Documented new platforms: EM1206 158 and EM1206W 158, DS1202 181, DS1206 186.
- Reworked Platform Specifications 138 topic.
- Documented new insert 213 function.
- Changes in the wln 536 object manual:
  - Every topic was updated and edited;
  - Added "Migrating From the WA1000" to address the changes in the wln object operation;
  - "Configuring CS Line" renamed into Configuring Interface Lines 545;
  - "Powering Down" renamed to "Rebooting";
  - "Detecting Disassociation or Powerdown" renamed into Detecting Disassociation or Offline State 556;
  - "Enabling Port" renamed into Applying Reset 546;
  - New Creating Own Ad-hoc Network 555 and Terminating Own Ad-hoc Network 556 topics;
- Changes in the button 234 object manual:
  - Expanded the main 234 topic;
  - Documented new button.pressed 235 R/O property;
  - Added information about "debouncing".

- Changes in the pat 363 object manual:

- Documented pat.greenmap 364 and pat.redmap 366 properties;

- Updated all other information in relation to the above.

- Changes in the fd 236 object manual:

- Updated fd.find 274 method;

- Documented new fd.rename 285 method.

- In connection with the above, renamed "Creating and Deleting Files" into Creating, Deleting, and Renaming Files 250, expanded topic content.

- Changes in the sock 421 object manual:

- Updated sock.close 475, sock.reset 495, and sock.discard 478 topics -- these methods are ignored when called from within an HTML page;

- Updated HTTP-related Buffers 462 -- an HTTP socket can now live without the RX buffer. Also, HTTP variables of any size can now be received;

- Reworked Working with HTTP Variables 469 -- this is now a section; explained and documented sock.gethttprqstring 480 and on_sock_postdata 491;

- Added "Redirection and UDP" to the Redirecting Buffers 454 topic;

- Documented URL substitution: new URL Substitution 468 and sock.urlsubstitutes 511 topics;

- Documented the data sinking feature: new Sinking Data 456 and sock.sinkdata 501 topics;

- Documented the timeout counter: expanded Closing Connections 437 topic (see Connection Timeouts), added new sock.toutcounter 507 topic.

- Changes in the ser 378 object manual:

- There is a new data sinking feature, so Sinking Data 399 and sock.sinkdata 501 topics were added;

- Corrected schematic diagram (C) in the Wiegand Mode 383 topic.

31AUG2008 release

- Documented kp. object 304, lcd. object 317.
- Documented md5 218, sha1 223, ddstr 210, and ddval 210 syscalls.
- Documented sys.serialnum 534 and sys.setserialnum 535. Added Serial Number 530 topic.
- Added Using Preprocessor 76 and Scope of Preprocessor Directives 78 topics.
- Updated The Watch 33 topic -- documented new capabilities such as true support for arrays, expressions ("x(y)"), etc.
- Update Project Settings 38 topic -- documented new Customize button.
- Updated EM202 platform -- this platform is now used by "203" devices as well.

04AUG2008 release

- Documented fd object 236.
- Added Legal Information 1 topic.
- Deleted "What's New in R2" and "Migration From Version 1" topics.

10MAR2008 release

- Documented wln object 536.
- Documented new EM1000W platform.
- Documented new romfile.offset 373 R/O property. In connection with this, updated the following topics: Supported Functions (Syscalls) (EM202/200 (-EV), DS202 platform), Romfile Object 370.
- Documented new sock.allowedinterfaces 474, sock.targetinterface 506, and sock.currentinterface 478 properties. In connection with this, also edited the following topics: Accepting Incoming Connections 426, Establishing Outgoing Connections 434, and Checking Connection Status 439. Changed information in the Supported Objects (EM202 platform) topic. Updated sock.localportlist 486, sock.targetinterface 506 property topics. Also edited "Platform-dependent Programming Information" topics of all platforms. EM1000 143 and EM1202 151 platforms got new "Enum pl_sock_interfaces" topics.
- Corrected a mistake in the Main Parameters 358 topic (net 358 object). The topic incorrectly stated that the Tibbo Basic application can't change the MAC address, which is, in fact, possible.
- Correction: default value for the net.ip 360 property is "1.0.0.1", not "127.0.0.1".
- Corrected net.ip 360, net.netmask 360, net.gatewayip 361 (details portion).
- Correction: EM1202 151 platform does not support RTC (rtc 375.) object.
- Edited Enum pl_io_num 147 topic of the EM1000 platform 143 manual to reflect newly supported I/O lines 49-53.
- Added to Understanding TCP Reconnects topic 428 (section about reconnects and HTTP). Note added also to Sock.reconmode Property 492 topic.
- Improved "Supported Functions" and "Supported Objects" topics for all platforms.

04SEP2007 release

- Extended and renamed the Project Browser 22 topic (formerly called "Using the Project Browser"). Also made new screenshot.
- New screenshots in the Code Auto-completion 23 topic. Text edited slightly as well.
- Updated the Tooltips 24 topic, created Supported HTML Tags 26 topic. New data concerns using HTML elements in tooltips.
- Updated the Watch 33 and Scopes in Watch 37 topics -- new screenshots; the text was also edited.
- Extended the Constants 65 topic -- added a new section about escape sequences in string constants.
- Updated Language Element Icons 136 (slight changes only).

09AUG2007 release

- Added the EM1202 151 platform description section.
- Corrected RTC Object 375 topic: should be rtc.getdata and rtc.setdata, not rtc.get and rtc.set.
- Minor corrections in the EM1000 143 platform description section.

12JUN2007 release

- Closing Connections 437 topic contained references to sock.abort method, which does not exist. Correct method name is sock.reset.
- Expanded Establishing Outgoing Connections 434 and Closing Connections 437 topics. Both topics now contain "Do not forget! Connection Handling is fully asynchronous" sections.
- Added "Socket re-use after connection closing" section to the Closing Connections 437 topic.
- New More On the Socket's Asynchronous Nature 441 topic.

12FEB2007 release

- Updated Adding, Removing, and Saving Files 18 topic.
- Added Graphic File Properties Dialog 133 topic.
- Updated Working With HTML 79 topic.
- Significantly expanded Embedding Code Within an HTML File 80 topic -- especially important: all code fragments on the HTML page are parts of one procedure.
- Updated Using HTTP 461, Generating Dynamic HTML pages 466, and Working With HTTP Variables 469 topics.

27DEC2006 release

- Added "What's new in R2" and "Migration From Version 1" topics.
- Updated The Watch 33 -- described new functionality, provided more info on how watch works.
- Scopes in Watch topics 37 -- provided more info on how watch works.
- Updated Using the Project Browser 22 -- selected platform is now visible in the topmost tree node.
- Updated Program Structure 43 -- explained that event handlers can also accept arguments.
- New Exceptions 29 topic
- Updated Variables and Their Types 48 -- added info about dword, long, real, float, and structures.
- Updated Type Conversion 50 -- almost 100% new text.
- New Type Conversion in Expressions 52 -- this section has been "under construction" for a long time.
- New Compile-time Calculations 53 topic.
- Updated Arrays 54 topic -- new ways to declare, etc.
- New Structures 58 topic.
- Updated and renamed "User-defined Types" topic. Now it is called Enumeration Types 59.
- Updated Understanding the Scope of Variables 61 topic.
- New Declaring Variables 64 topic.

- Updated Introduction to Procedures 66 -- explained that event handlers can also accept arguments and can never be functions procedures.
- Updated Dim Statement 86 topic -- new data about ways to define array variables.
- New Type...End Type Statement 100 topic.
- Updated Passing Arguments to Procedures 68 topic (strict byref argument match is now required).
- Updated Goto Statement 93 -- all labels are local!
- New Supported Variable Types topics for each platform (EM202, EM1000).
- Updated Platform-dependent Programming Information topics for each platform (EM202, EM1000).
- EM202 platform no longer supports redirection -- Enum pl_redir topic has been updated.
- Updated Supported Functions (Syscalls) for EM202 and EM1000 platforms -- some stuff in, some stuff out.
- Updated Generating Dynamic HTML Pages 466 topic -- described changed behavior when the same code snippet has to be executed from two instances of the same HTML page being sent to the browser.
- Updated Httpnoclose Property 482 topic -- there is a new "separator" string.
- Updated Pat.play 365 and Beep.play 233 topics -- now "**" means x4 speed.
- New Sys.onsystimerperiod Property 533 topic.
- Updated On_sys_timer Event 533 topic -- to reflect that there is a new sys.onsystimerperiod 533 property.
- New Sock.inconenabledmaster Property 484 topic.
- Updated Accepting Incoming Connections topic -- added material regarding sock.inconenabledmaster 484 property.
- Updated Stor.getdata Method 523, Stor.setdata Method 524, Rtc.getdata Method 376, Rtc.setdata Method 377 topics because all four methods have been renamed.
- New Cfloat Function 207, ftostr Function 211, Lbin Function 214, Lhex Function 216, Lstr Function 216, Lstri Function 217, Lval Function 218, Strtof Function 228 topics.
- Updated Vali Function 230 topic -- this function is no longer available since val 229 function now works both for word (unsigned) and short (signed) conversions.
- Updated Val Function 229 topic to reflect the fact that this function is now used both for word (unsigned) and short (signed) conversions.
- Updated Str Function 225, Stri Function 226, Bin Function 207, Hex Function 212, Val Function 229 topics -- more accurate description and examples.
- Added "declaration" to the description of all events.
- Updated sock.event R/O Property 480 and sock.eventsimple R/O Property 480 topics -- these properties are not longer available.
- Updated On_sock_event Event 480 topic -- this event now carries newstate and newstatesimple arguments that have replaced sock.event 480 and sock.eventsimple 480 R/O properties.
- Updated Checking Connection Status 439 topic to reflect the changes made to the on_sock_event 490.
- New "Split Packet" Mode of TCP Data Processing 453, .Splittcppackets Property 501, and On_sock_tcp_packet_arrival Event 491 topics.
- Updated certain screenshots in several topics.

- Added Image Editor topics: Built-in Image Editor 20, Image Menu 124, Image Editor Toolbar 127, Tool Properties Toolbars 128 (+ all subtopics).
- Updated Adding, Removing, and Saving Files 18 topic (added image editor-related info).

06JULY2006 release

- Added new platform -- EM1000 143.
- Added "Platform revision Programming Information" topics to EM202 and EM1000 143 platform documentation.
- Stor 522 object got new property- stor.base 523. Entire description of the object has been updated because of that.
- Clarification has been added to the romfile 370 object description. This object can only access first 65534 bytes of each file, even if the actual file is larger.
- Entire new beep 232 object has been added.
- New feature in io 294 object -- io.enabled 298 property was added.
- New feature in system 526 object- see PLL Management, sys.currentpll 531, sys.newpll 532, sys.resettype 535.
- New features in serial port 378 object- support for Wiegand and clock/data interfaces. New topics include: Three Modes of the Serial Port 380 with subtopics, ser.mode 409, and ser.autoclose 402. A lot of other topics have been changed- too many to list here.
- Change in sys.buffalloc 530 behavior: now if the serial port (socket) to which the buffer belongs is not closed (idle) the buffer size will remain unchanged. This affects ser.rxbuffrq 415, ser.txbuffrq 419, sock.rxbuffrq 497, sock.txbuffrq 509, sock.tx2buffrq 508, sock.cmdbuffrq 476, sock.rplbuffrq 496, sock.varbuffrq 511.
- Corrected errors in the Enum pl_io_num (pin descriptions were wrong- RTS, CTS, DTR, and DSR lines were shown at incorrect positions).
- Corrected ser.txlen 420, ser.txfree 420, sock.txlen 510, sock.txfree 510 property descriptions. These properties *do not* take into account uncommitted data in the TX buffer (it was stated otherwise previously). Consequently these topics were also edited: Buffer Memory Status 394, TX and RX Buffer Memory Status 447. Ser.notifysent 410, on_ser_data_sent 412, sock.notifysent 487, on_sock_data_sent 489, ser.setdata 418, and sock.setdata 500 have been amended accordingly.
- Corrected mistakes related to date/time conversion functions- date 208 function was erroneously documented as "day" function, weekday 230 function description was missing altogether. Topics of other date/time related functions- year 230, month 221, daycount 209, hours 213, minutes 221, and mincount 220 were slightly corrected.

08MAY2006 release

- Corrected errors in io.Num Property 301 and io.State Property 303

08MAR2006 release

- Updated Preparing Your Hardware 9 with the network upgrade procedure
- Updated Starting a New Project 10
- Updated Making, Uploading and Running an Executable Binary 26

- Updated Project Menu 123 with new entry description for Device Explorer
- Updated Debug Toolbar 126 with new button description for Device Explorer
- Updated and expanded Device Explorer
- Added new functions: Day Function 208, Daycount Function 209, Hours Function 213, Mincount Function 220, Minutes Function 221, Month Function 221, Year Function 230

11JAN2006 release

- Improved indexes -- better context search.
- Added L1008 118, L1009 118

02JAN2006 release

- Initial release of manual.

Index

- D -

- E -

- F -

- G -

- H -

- I -

- J -

- K -

- L -

- M -

- N -

- O -

- P -

- Q -

- R -

- S -

- T -

- U -

- V -

- W -

- X -

- Y -