Introduction

µTasker is an operating system designed especially for embedded applications where a tight control over resources is desired along with a high level of user comfort to produce efficient and highly deterministic code.

The operating system is integrated with TCP/IP stack and important embedded Internet services along side device drivers and system specific project resources.

µTasker and its environment are essentially not hardware specific and can thus be moved between processor platforms with great ease and efficiency.

However the µTasker project setups are very hardware specific since they offer an optimal pre-defined (or a choice of pre-defined) configurations, taking it out of the league of "board support packages (BSP)" to a complete "project support package (PSP)", a feature enabling projects to be greatly accelerated.

This tutorial will give you a flying start to using the µTasker on the Luminary Micro (LM) LM3SXXXX. It can be fully simulated using VisualStudio 6.0 or higher and has target settings for the Lumiary Micro Evaluation Boards for the LM3S1968, LM3S6965 and LM3S8962 – other type can be easily created from these.

Although there are various ways to work with the LM3SXXXX, the following method is described in this tutorial:

- Programming of the chip is performed using either the inbuilt FTDI JTAG USB interface on the evaluation boards or an external JTAG debugger (like Segger J-Link, Rowley CrossConnect or Olimex ARM-USB-OCD). If the LM serial loader is programmed to the chip also the LM FLASH Programmer can be used to load via UART0 – this can generally only be performed once since it then overwrites the serial loader.
- Compiling and debugging is performed using IAR Embedded Workbench V4.42A or using Rowley CrossWorks V1.7 Build 7 and GNU compiler.

The LM3SXXXX is based on ARM's Cortex™ M3 core. The advantages of this core is that it operates using the Thumb-2 instruction set which automatically combines the advantages of the original ARM and Thumb modes, resulting in code with approximately the same density as Thumb but with performance around 20% higher than the original ARM mode. The core integrates a nested vectored interrupt controller (NVIC) supporting up to 240 interrupts with programmable priorities.

Luminary Micro is one of the first companies to offer devices with the Cortex™ M3 code and has very quickly extended it offering to well over 100 derivatives whereby a large percentage include on-board Ethernet with PHY (supporting MDI/MDI-X crossover).

The small footprint and easy configurability of the µTasker project also allows the smallest parts (eg. the LM3S101 in 24 pin package with 8k FLASH and 2k SRAM) to be used.

Some commercial compilers and debuggers are available as evaluation versions. Often they have a 32k code limit which is suitable for the µTasker demo project – although it is not possible to activate absolutely every feature at the same time, useful and interesting projects can be performed to see it in full action.

Other commercial compilers only allow a smaller code size for evaluations. These are not mentioned anymore since 16k is not adequate to produce serious demonstrations of embedded IP work.

Even of you don't have the tools available at the moment, the µTasker simulator will enable you to get started. Therefore there should be nothing standing in your way to getting your first, powerful embedded IP project up and running.

Getting started

You are probably itching to see something in action and so why hang around? Let's start with something that will already impress you and your friends – no simple and basically useless demo which blinks an LED in a forever loop but something seriously professional and for real life projects very handy.

First I will assume that you have VisualStudio installed on your PC since we will first simulate everything – but don't worry, we are not going to see something attempting to interpret the instructions of the processor and requiring 2 minutes to simulate a couple of seconds of the application, instead we will see your PC operating in "real time" as the target processor. Your PC will not realise that the processor is simulated and so when you try contacting it by pinging it or browsing to it with Internet Explorer, we will see that your PC will sends IP frames to the network and will see answers on the network from the simulated device. Other PCs or IP enabled embedded devices on the network will also be able to communicate with the simulated device. You can also capture the frames using a sniffer tool (we will use Ethereal/Wireshark) for further analysis and playback…. Getting excited? In a few minutes you will see it in action!

> If you haven't VisualStudio (6.0 or newer) then there are trial versions [http://msdn.microsoft.com/vstudio/products/trial/] and you can probably pick up a 6.0 version for very little cash on Ebay. VS 6.0 is adequate for our work as are the newer light editions. It contains a world class C-compiler and editor as well as loads of other tools which make it a must, even for embedded work.

The simulator requires also WinPCap installed (from http://www.winpcap.org) which is an industry-standard tool for link-layer network access in Windows environments. It is also required to be installed in order for the network sniffer Ethereal/Wireshark to run (see later on in the tutorial for details of its interaction capabilities with the µTasker simulator). Therefore it is just as well to install Ethereal/Wireshark before getting started since it is a great tool which you will never want to be with out again…(get it from http://www.ethereal.com / or http://www.wireshark.org/ and see the discussion towards the end of this document).

Note that Wireshark was originally known as Ethereal. Ethereal development has effectively moved to the newly named project… for an interesting explanation of the reasons, see the following Linux feature: http://www.linux.com/feature/54968

If you don't want to see the simulator in action – which would be a big mistake as you will miss the opportunity to save many hours of your own project time later – there is target code which can be loaded to the target and will also run. This is also detailed later on in the tutorial.

You will find that the simulator can be used for most of the real development work. First it allows testing things which you may not already have available as hardware (even if you have no evaluation board and cross compiler for it yet, you can start writing and testing your code) – it will allow you to use a matrix keyboard, LCD, I^2C EEPROM or SPI EEPROM (and more) connected to the virtual target without having to get your soldering iron out to connect it. And it is so accurate that you can then cross compile to the target and it will (almost certainly) work there as well. This is the last time I will say it … ***don't make the BIG MISTAKE of taking a short cut when starting and diving into coding on the target***. If you are an embedded SW professional you will be missing the chance of saving enough time in a year for a couple of months extra vacation. If you are a hobby user, you will be missing the change to get out more…!

So now I've said it and you have heard - so let's roll!

Fasten your seat belts since we are about to roll

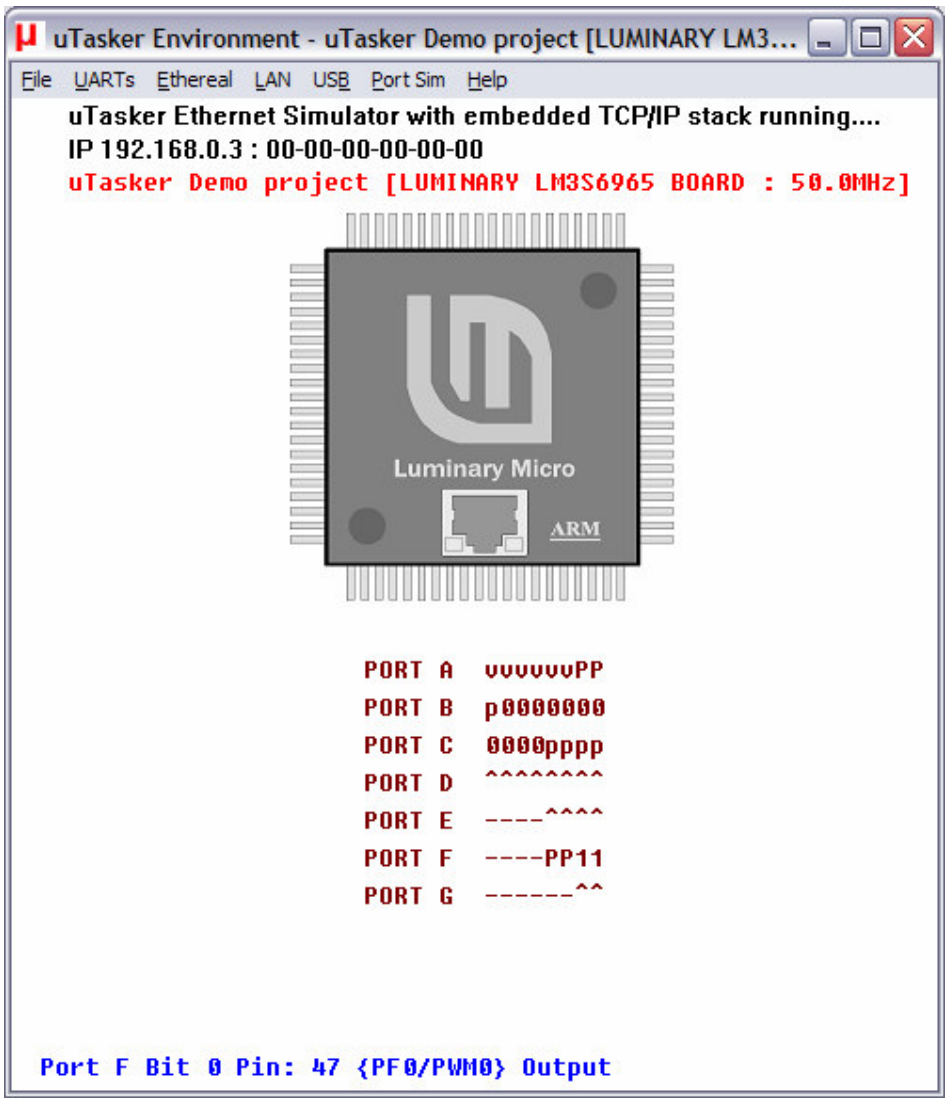
1. Simply copy the complete µTasker folder to your PC and go to "uTaskerV1.3\Applications\uTaskerV1.3". This is a ready to run project directory showing a useful application using network resources.
2. Move to the sub-directory "Simulator" and open the VisualStudio project workspace uTaskerV1-3.dsw. This project is in the VisualStudio 6.0 format to ensure compatibility and, if you have a higher version, simply say that it should be converted to the new format – no problems are involved and it should build with no warnings.
3. *Ensure that the compiler is set up for the LM3SXXXX target in the project's pre-compiler settings: look for the pre-processor define **_LM3SXXXX**. If instead you find that the project is set up for another target, eg _HW_NE64 or _M5223X simply overwrite this with **_LM3SXXXX**.*
   ***Depending on which of the LM boards you actually want to work with you can set this up in config.h. Choose EK_LM3S6965 or EK_LM3S8962 – others such as EK_LM3S1968 will operate but without Ethernet so are not discussed here**.*
   Build the project (use F7 as short cut) and you should find that everything compiles and links without any warnings.
4. I would love to be able to say "Execute" but there are a couple of things which have to be checked before we can do this. First of all, you will need to be connected to a network, meaning that your PC must have a LAN cable inserted and the LAN must be operational – either connected to another PC using a crossover cable or to a router or hub (wireless links tend not to work for some reason).
   Then we have to be sure that the network settings allow your PC to speak with the simulated device. The IP address must be within the local subnet:
   Open the C-file **application.c** in the project and check that the following default settings match your network settings (check what your PC uses in a DOS window with "ipconfig") <u>but DON'T copy the PC's own IP address – the simulated device needs its own!!</u>

```
static const NETWORK_PARAMETERS network_default = {
(AUTO_NEGOTIATE | FULL_DUPLEX | RX_FLOW_CONTROL),    //
usNetworkOptions - see driver.h for other possibilities
{0x00, 0x00, 0x00, 0x00, 0x00, 0x00},  // ucOurMAC - our
                                        default MAC
{ 192, 168, 0, 3 },                    // ucOurIP - our
                                        default IP address
{ 255, 255, 254, 0 },                  // ucNetMask - our
                                        default network mask
{ 192, 168, 0, 1 }                     // ucDefGW - our
                                        default gateway
};
```

   Just make sure that the network mask matches, that the device's IP address is within the local network and that the IP address defined doesn't collide with another one on the local network. After making any modifications, simply compile the changes and we will be in business.

5. So now I can say EXECUTE (use F5 as short cut).

   You will see the simulated device working away on your PC screen. There will also be one port output toggling away. Let's take a quick look at it. This is in fact the watchdog routine which is called every 200ms which is also toggling the output so that we can see that all is well (Port F.0]).



If you hover your mouse over the port which is toggling, or other available ones, you will see which pin number it has on the device, its name and present use, as well as the possible peripheral functions which it can be programmed to perform. In the screen shot above this is seen by the line in blue at the bottom of the simulation window.

Now open the file *uTasker\watchdog.c* (in the VisualStudio project manager). Put your cursor on the call to retrigger the watchdog "**fnRetriggerWatchdog**()" and hit the F9 key (set a break point). Almost immediately the program will halt and you can use F11 to step into the hardware specific routine responsible for triggering the watchdog and in our case also toggling the port output. The routine performs a watchdog feed

sequence and returns. Remove the breakpoint and let the simulated device run again using F5.

6. OK so at least you are convinced that it is really running our project code, but it is not exactly something to write home about.

   Let's get down to more serious stuff:
   There are some menu items in the µTasker environment simulation window. Open "`LAN | Select working NIC`" and select the network card you would like the simulator to use – don't worry, it will be shared with anything else which is already using it.
   Open a DOS window and do the standard PING test - "ping 192.168.0.3" if you didn't need to change anything, or the IP address you defined if you did.
   The simulated device should now be responding. That means that your PC has sent ICMP PING test messages to the network and received answers from some network device (or course our internal simulated device – but no one will know the difference). You can also try sending the ping test from another PC on the network and it will also receive an answer.
   If you have a network sniffer you can also record the data from the network – don't worry if you haven't used a network sniffer before because we will come back to this later on and use Ethereal/Wireshark to do the job.
   Watch the LAN indicators on the processor's virtual LAN connector. The left one will blink green when a frame is received – if it is accepted by the device (matching address, broadcast or all when in promiscuous mode). The right one will blink red whenever the simulated device sends a frame to the network.

7. I suggest that you now close the µTasker environment simulation window using the normal method `File | Exit` or by clicking on the close cross in the top right hand corner.
   This standard termination will cause the selected NIC to be saved to a file in the project simulation directory called **NIC.ini** and so you will not have to configure it the next time you start the µTasker environment simulation.

8. Now I'm sure you are not yet satisfied with the progress, so let's execute the project again (short cut F5) and this time we will do something a bit more interesting.
   Start Internet Explorer 6, FireFTP or other FTP client you may have. Connect to the IP address of the simulated LM3SXXXX ftp://192.168.0.3 (or the address of your device).
   If you have Internet Explorer 7 this is a bit trickier – check the document "uTaskerFTP.doc" for a full discussion and tips and tricks. If all else fails, the DOS FTP program will certainly work.
   When connected to the device you will see an empty directory. It is normal that the directory is empty – I mean this is a fresh device which has never before been programmed…..
   Now open a second Internet Explorer window and enter the address http://192.168.0.3 You will see a web page displaying that the requested file was not found, which is correct since we have not loaded any web pages yet. The 404 error page is in fact compiled as standard into the code of the web server and it is returned when any requested page is not found….

9. Now look in the project directory "uTaskerV1.3\Applications\uTaskerV1.3\WebPages\WebPagesLM3SXXXX". These are web pages which we will now load to the device.
   If you are using Explorer 6 you can transfer all of the files to the FTP server by simply selecting all *.htm; *.jpg;*.gif files and throwing them over to the FTP window (that is,

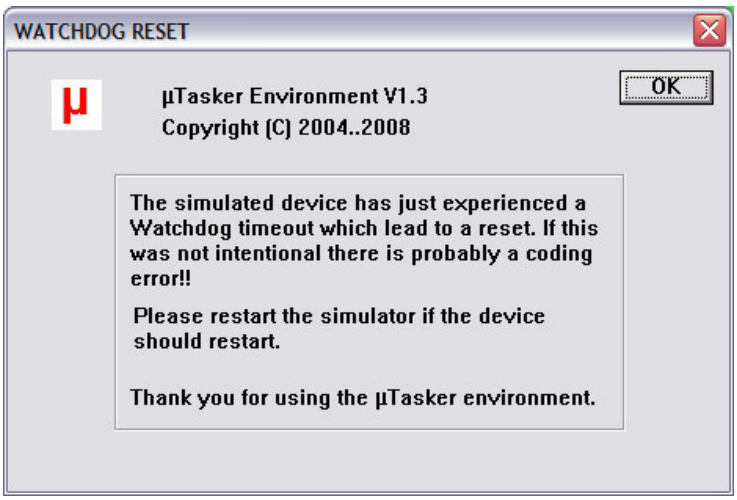
using drag-and-drop). Otherwise copy them with whatever method your FTP client supports.
Then go back to the web browser window and make a refresh. Now you will see the start side and can navigate to a number of other sides.
Before we get into any more details about the web pages and their uses, please modify and save the Device ID (on the start page – default "uTasker Number 1") to any name of your choice (In the following sections the name LM3SXXXX is assumed) and close the µTasker environment simulation window by using the normal exit method

By doing this, the contents of the device's simulated FLASH (Flash is used to save the web pages from the file system and also any parameters, for example the device ID which you have modified) are saved to a file in the project simulation directory called **FLASH_LM3SXXXX.ini**.

Now execute again (F5) and check that the loaded web pages are all there and that after a refresh of the start side also the device ID which you chose is correct.
Now you should understand the operation of the simulator; *on a normal exit it saves all present values and settings and on the next start the device has the saved FLASH contents as at the last program exit*. This is exactly how the real device works since, after a reset, its FLASH contents are as saved – I mean, that is what FLASH is all about.
If however you have modified FLASH contents and do not want them to be saved, you can always avoid this by quitting the debugger (short cut SHIFT F5) which causes the simulator to be stopped without performing the normal save process.
If you wish to start with a fresh device (with blank FLASH contents) then simply delete the **FLASH_LM3SXXXX.ini** file…

10. There is one important setting which we should perform before continuing; that is to set a MAC address to our new device.
    So browse to the LAN configuration side, where you will see that the default MAC address is 00-00-00-00-00-00. This works, but we really should set a new one – as long as we are not directly connected to the Internet any non-used value can be set. You make one up. For example set 00-11-22-33-44-55 (watch that the entry format is correct) and then click on "modify / validate settings". The entry field will be set inactive since it is no longer zero and can not be changed a second time - should your want to reset the value before we commit it to (simulated) FLASH click on "Reset changes", otherwise click on "Save changes".

    The device is commanded to be reset at this point after a short delay of about 2 seconds to allow it to complete the web page serving. The simulator will open a dialog box to indicate that it has been terminated with a reset:



You can restart the simulator after the reset (which was interpreted as a normal program exit and so the changes to the FLASH have also be saved to the hard disk and will be restored on the next program start) by using F5 as normal.

You may have read on the LAN configuration side that the new setting should be validated within 3 minutes – this is a safety mechanism so that falsely set critical values do not leave a remote device unreachable and means that we should establish a connection within three minutes of the new start to verify that all is well, otherwise the device will automatically delete the newly set values, reload the original ones and can then be contacted as before, using the original settings. So we will now validate our new setting (new MAC) so that it will always be used in the future.

Refresh the web pages by clicking first on “Go back to menu page” and then “Configure LAN Interface” again.

You may well have a shock because it probably doesn’t respond….but don’t get worried because your PC is still trying to reach the IP address using the previous MAC address, which is no longer valid (hence an incorrectly configured device can become unreachable). In a DOS windows type in “arp –d” which will delete the PC’s ARP table – which is mapping IP address to MAC addresses and then it will work on the second attempt.

The page will not allow any parameters to be modified and also the check box “Settings validated” shows that the device is waiting for the new values to be validated. To perform this, click on “Modify / validate settings”, after which the parameters will be displayed as validated. (*Don’t forget to terminate the simulator normally later so that it is really the case*).

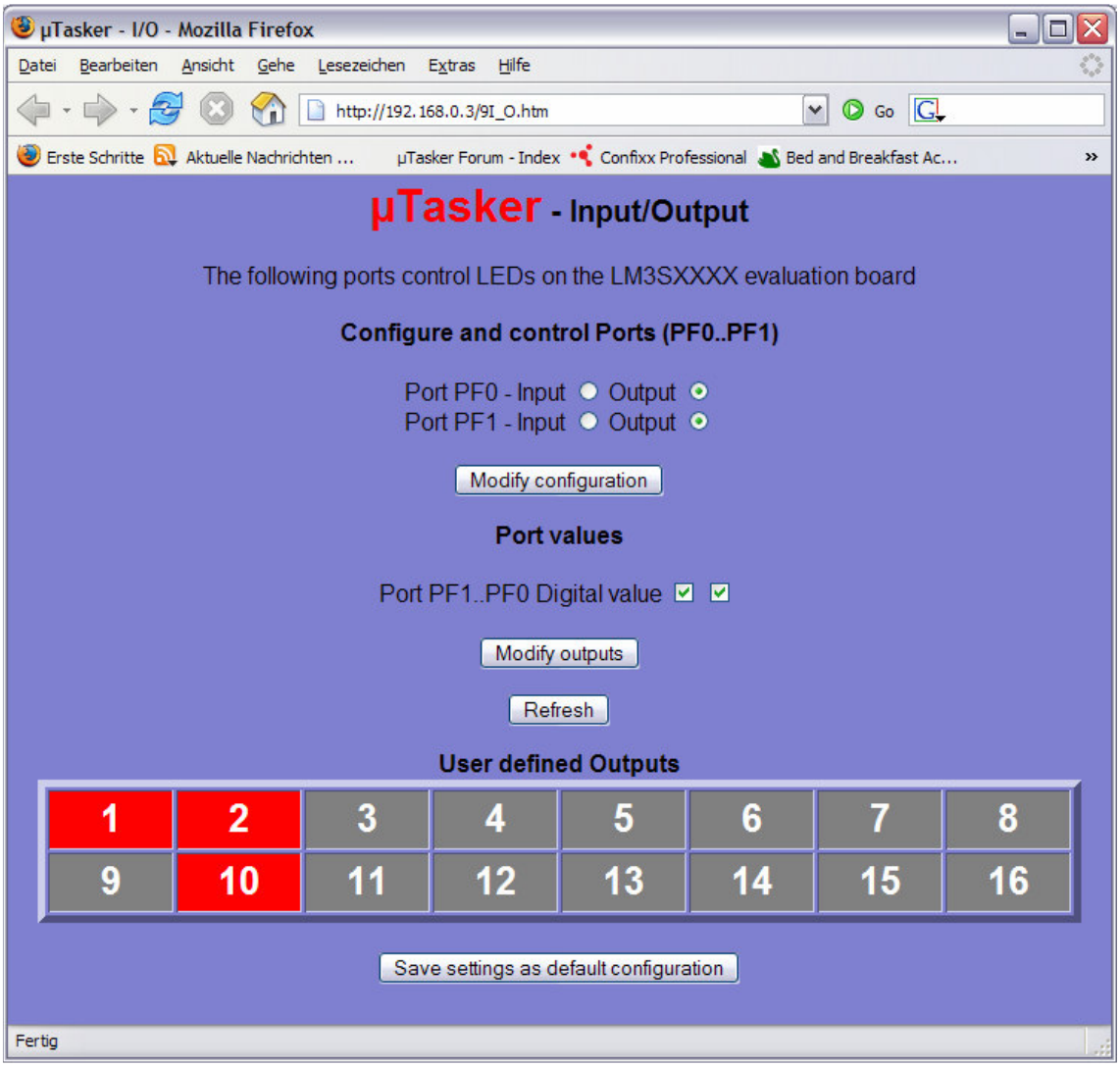
Should the 3 minutes have elapsed before you managed all that, the simulated device will again reset, clearing the temporary settings and reverting back to the initial MAC address. Just start the simulator again and repeat this step.

11. Now you may be thinking that it is all well and good having a web server running on a device, sitting somewhere on the Internet but you probably want to use the device for controlling something - in its simplest form by switching something on or off. Using a relay this could be something quite powerful which can also have really useful benefits; imagine browsing to your device and commanding it to open the blinds or turn on the lawn sprinkler…the list is endless…
    Look at the simulated LM3SXXXX and imagine that you have connected your lawn sprinkler to the Port F1. Presently it is at the state input ‘1’ [signified by ‘^’], the default

state when the software starts assuming a pull-up on the input. Now click on the menu page link to open an I/O window – it is shown in the following screen shot. Notice that the port PF1 is being displayed as an input and its digital value as active ('1') in its check box.

Do a quick test: Hover your mouse over the port PF1 in the simulator – it will tell you which pin it is of your device. Now click – this has toggled the simulated input (each time you click, it will change state again) – leave it at the low state '0', displayed as 'v'. Now refresh the I/O web page by clicking in the Refresh button.
What is the web page now displaying? If it is behaving correctly it will of course be displaying the new port input state.

Now we can configure it as an output and control its state from the I/O page. See next page.



Set PF1 to be an output, rather than an input and click on "Modify configuration". Afterwards set the port state to '1' by un-checking the third port value from the right and click on "Modify outputs".

Watch the state of Port PF1 – you will see that the output is now set and now your lawn would get the water it has been waiting for!
Set the state to '0' and it will be turned off. In this example web page, the 2 Ports PF0 and PF1 outputs can be controlled individually.

PF0 is used as a RUN LED in the demo project and so is configured as an output by default..
If you would like a certain setting to be returned after every reset of the device, simply save the setting by clicking on "Save settings as default configuration" and you will see that they are indeed set automatically when the device is started the next time.

The demo project includes also 16 user controllable outputs. These are automatically configured as outputs in the project code and can be seen as outputs with state '0' when looking at the simulator – see the ports B and C. If you now click on the user defined outputs on the I/O page you will see that they immediately toggle their port output and the state is either displayed as a grey button when '0' or as red buttons when '1'. Also the present state of the 16 outputs can be saved as default states when the target starts the next time.
Note that these outputs can be mapped to any port and port bit in the demo project – it shows a useful example of flexible independent output control. The demo setting actually doesn't control all of these since some of the pins default to JTAG pins on the Luminary devices – this will also be seen in the simulator.

12. The last setting which we will change before having a rest is to activate HTTP user authentication and disable the FTP server so that no one else without our password has entry to the web server and no sneaky person can change the web pages which we have just programmed…

    Browse to the administration page, deactivate FTP and activate HTTP server authentication before selecting the action to "Modify and save server settings". Finally click on "Perform desired action" and close the Internet Explorer.
    Open Internet Explorer again and enter the device's address http://192.168.0.3 (or your address) and this time you will need to enter the user name "ADMIN" and password "uTasker" to get in.
    Try to do something with the FTP server and you will find that it simply won't work anymore. This is because it is no longer running in the simulated device and you can rest assured than no one out there will be able to change anything which you have loaded.

    Note that the FTP server supports its own authentication and also passive mode and options. Please see the document "uTaskerFTP.doc" in the document folder for details and its configuration.

I think that it may be time to take a short break.

I hope that the introduction has shown that we are dealing with something which is very simple to use but is also very powerful in features. There are lots of details which need to be learned to understand and use everything to its fullest capabilities and it is up to you to decide whether you want this or would like to simply use the µTasker and its capabilities as a platform for your own application development. In any case you have just learned the basics of a powerful tool which is not only great fun but can really save you lots of development time.

Testing on the target

Up until now we have been using the simulator and hopefully you will agree too that it is a very useful tool. We have tested some quite useful code designed to run on the LM3SXXXX and done this in an environment enabling us to do embedded IP stuff in real time. We haven't actually done any debugging or added new code but you can probably imagine the advantages of being able to write and test it on the PC before moving to the target.

At the same time you are probably thinking about how that will all work on the real device. Perhaps you are worried that it is all a show after all and the real device will remain as dead as a door nail or at the best crash every time the user breaths heavily. So let's prove that this is not the case by repeating the first part of the tutorial, but this time we will go live…

Compiling the project for the target

The first thing that we need to do is to compile the project for the target. This means that we will be cross compiling the code which we already have been testing with the simulator, using an assembler, a C-compiler and a linker. There are a number of such tools available and unfortunately they are not all compatible in very aspect, even if ANSI C compatible. Basically there is always the chore of getting the thing to reset from the reset vector, meaning that the start-up code must be available and at the correct location. Then there are specialities concerning how we force certain code or variables to certain addresses or regions and how we need to define interrupts routines so that they are handled correctly. Sometimes there is also the need for some special assembler code which does such things as setting the stack pointer or enabling and disabling interrupts.

The µTasker demo project for the LM3SXXXX is delivered with an IAR Embedded Workbench project, a Rowley Associates CrossWorks project and a 'stand-alone' GNU build. The latter is integrated in the VisualStudio project as an optional post-build step. This requires a GNU compiler to be available of the local PC – it will compile the project but doesn't support debugging.

The IAR compiler is a very good compiler, producing highly efficient code so is recommended for professional work. There are evaluation versions available from IAR - which are generally time or code size limited - for anyone wanting to get a feel for the product. http://www.iar.com/

The Rowley Associates CrossWorks is GNU compiler based but supplies a complete development environment including support. This is recommended for professional users who would like to base their work on GNU but don't want the hassle of having to configure the various tools before being able to start productive work. They also offer attractively priced educational and personal licenses. http://www.rowley.co.uk/

Otherwise it should be possible to get the code up and running with any other compiler but it will involve setting up an equivalent project and a certain amount of knowledge of the processor being used and various details about the compiler itself.

By the way, if you can't wait until you have installed the compiler or get your own project up and running, there are files delivered with the µTasker which were compiled with the abovementioned compilers and can be loaded already…

IAR

You can find the project and the target file(s) in the sub-directory called *IAR_LM3SXXXX*:

`uTaskerV1.3.eww` This is the IAR Embedded Workbench project work space

IAR_LM3SXXXX\Release\Exe\uTaskerV1.3.bin This is the file which you can download to the target using LM FLASH Programmer, as explained later

When compiling the project, expect three non-serious compiler warnings but no errors.

Rowley CrossWorks

You can find the project and the target file(s) in the sub-directory called *Rowley_LM3SXXXX*:

`uTaskerV13.hzp` This is the Rowley CrossStudio project work space

IAR_LM3SXXXX\THUMB Flash Release\Exe\uTaskerV1.3.bin This is the file which you can download to the target using LM FLASH Programmer, as explained later

When compiling the project, expect neither warnings nor errors to be emitted by the GNU compiler.

Stand-alone GNU build

The µTasker simulator runs under VisualStudio and has two build configurations. The default is ***uTasker LM3S6XXX***, which we have been using up to now. The second is ***uTasker LM3S6XXX and GNU Build***. This does the same as the first configuration but automatically starts a GNU post-build when the VisualStudio project builds successfully. This is simply the call of a bat file which starts the GNU build, but the advantage is that the output of the GNU build appears in the compile windows.

To activate the GNU build option you simply need to set this as active configuration. In addition you will have to have to set up the bat file to suit the local GNU compiler – just open the bat file and modify the paths so that the GNU tools are found. You will also have to ensure that the version of the GNU compiler that you have does indeed support the Cortex M3.

The bat file (Build_LM3SXXXX.bat) can be found in \uTaskerV1.3\GNU_LM3SXXXX and can also be started by double clicking on it.

GNU_LM3SXXXX *\uTaskerV1.3.bin* This is the file which you can download to the target using LM FLASH Programmer, as explained later

When compiling the project, expect neither warnings nor errors to be emitted by the GNU compiler.

If your goal is a complete open-source tool chain then it is probably best to use the ***Yagarto*** solution. This is based on Eclipse as IDE and enables a complete solution from editing, building, debugging and version management. With low cost development boards and JTAG

debugger from Olimex (www.olimex.com) and the famous tutorial from ***James Lynch*** about how to install, configure and run, professional projects are possible on a shoe-string. The stand-alone GNU build is compatible with this environment and more details about it can be found at http://www.utasker.com/forum/index.php?topic=34.0 (including how to obtain the tutorial mentioned above).

Warning to users of limited compiler versions

The full functionality of the uTasker demo project produces a code size of about 50k..60k, depending on the compiler used. This will be too large for 32k limited versions of commercial compilers and so it will be necessary to remove some functions, depending on which ones are interesting for you and which less. Later there is a comparison of code sizes required by the modules so that you can easily estimate what is possible and what needs to be trimmed out. Interesting projects are however still possible without exceeding the limits.

Downloading the code to the target

Most of the development environments include support for downloading the code to the FLASH of the target so this procedure can be performed quite comfortably using a compatible JTAG debugger. If you own such a compiler and debugger you probably know well enough how this works.

> ***Note for use of external JTAG debugger with Luminary evaluation boards***
>
> *The on-board FTDI interface will be automatically disabled when an external one is detected. It was however found that it is usually necessary to add a non-mounted resistor to the board between +3.3V and Pin 1 of the JTAG connector. Check the diagram to your board and check whether the resistor is mounted or not – if the external JTAG debugger refuses to see the board (usually because it can't see the 3V3), add the resistor (or a short circuit) to solve the problem.*

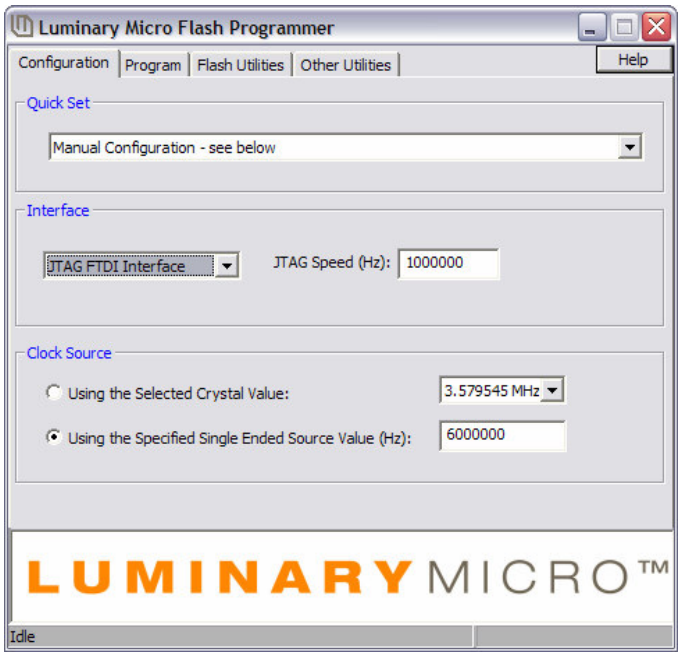
The Luminary Micro chips are supplied with a pre-installed boot loader program. This allows serial programming via SPI or UART0, where SPI can be useful for high-speed production programming. Generally this allows only one-time-programming since the new code will overwrite the loader. It is thus also not very practical for development work. Luminary supplies two different download utilities "LM Flash Programmer" and "sflash.exe", which allow programming via UART and also via FTDI JTAG. [LM Flash programmer has a graphical use interface and allows only programming, while the sflash.exe is purely command line oriented but supports all functions of the Serial FLASH Loader interface]. Since most users will be working with an evaluation board from Luminary with the FTDI JTAG integrated, the LM Flash Programmer is convenient for loading binary files to the evaluation board and is thus described here. The Luminary Micro evaluation boards don't generally include an RS232 connector so UART loading – which does work when an RS232 converter is added – is not very practical.

The Luminary Micro programming software can be downloaded from the Luminar Micro web [http://www.luminarymicro.com] site - this requires a simple registration.

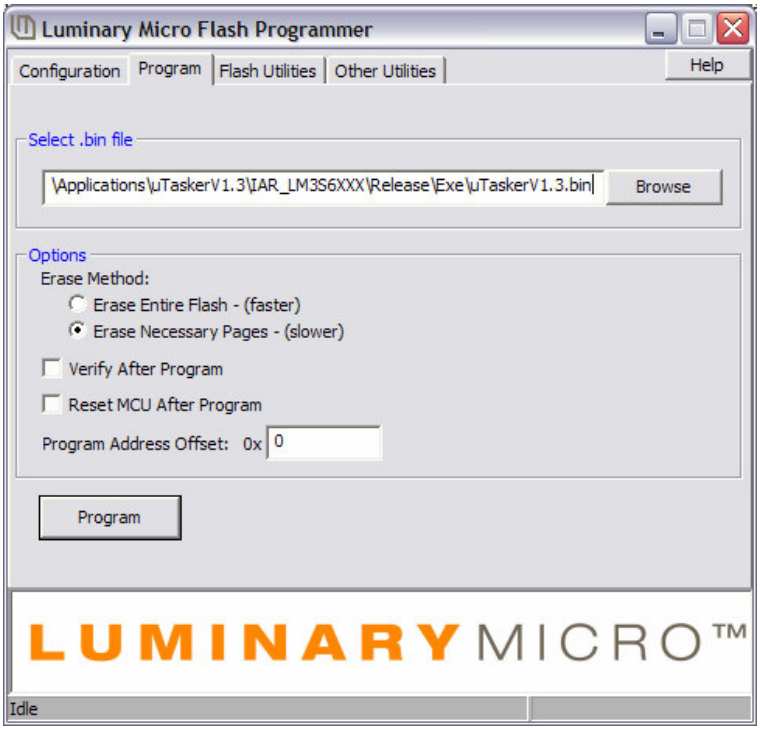
LM Flash Programmer expects a *.bin file as input. The µTasker projects are configured to generate a compatible output file in release mode. The following illustrates how to load the file to the evaluation board via USB – programming a fresh board with the LM boot loader installed is equivalent when a COM port is selected (the LM boot loader is also supplied as binary file together with the program so this can also be programmed using this technique):

Step 1: Connect the evaluation board via USB to your PC (it is assumed that you have already installed the FTDI drivers).

Step 2: Start the LM FLASH PROGRAMMER and ensure that the JTAG FTDI Interface is selected. The other values can be left at their default settings.



Step 3: Change to the programming register and set the path to the binary file to be loaded:



Step 4: Press the program button and wait until the programming has completed. This is seen in the bottom status line.

Step 5. Press the RESET button on the LM evaluation board to start the new software on the board.
Note that the FLASH Programmer can also be used to delete the entire FLASH in the FLASH utilities register. When downloading a program only the used sectors are deleted when left at default configuration – *this is practical when the file system and parameter system should not be deleted when loading new application software*. Also an automatic reset can be performed after the download so that the code starts immediately – *check the appropriate options on the programming page*.

Step 6: Test the project running on the target

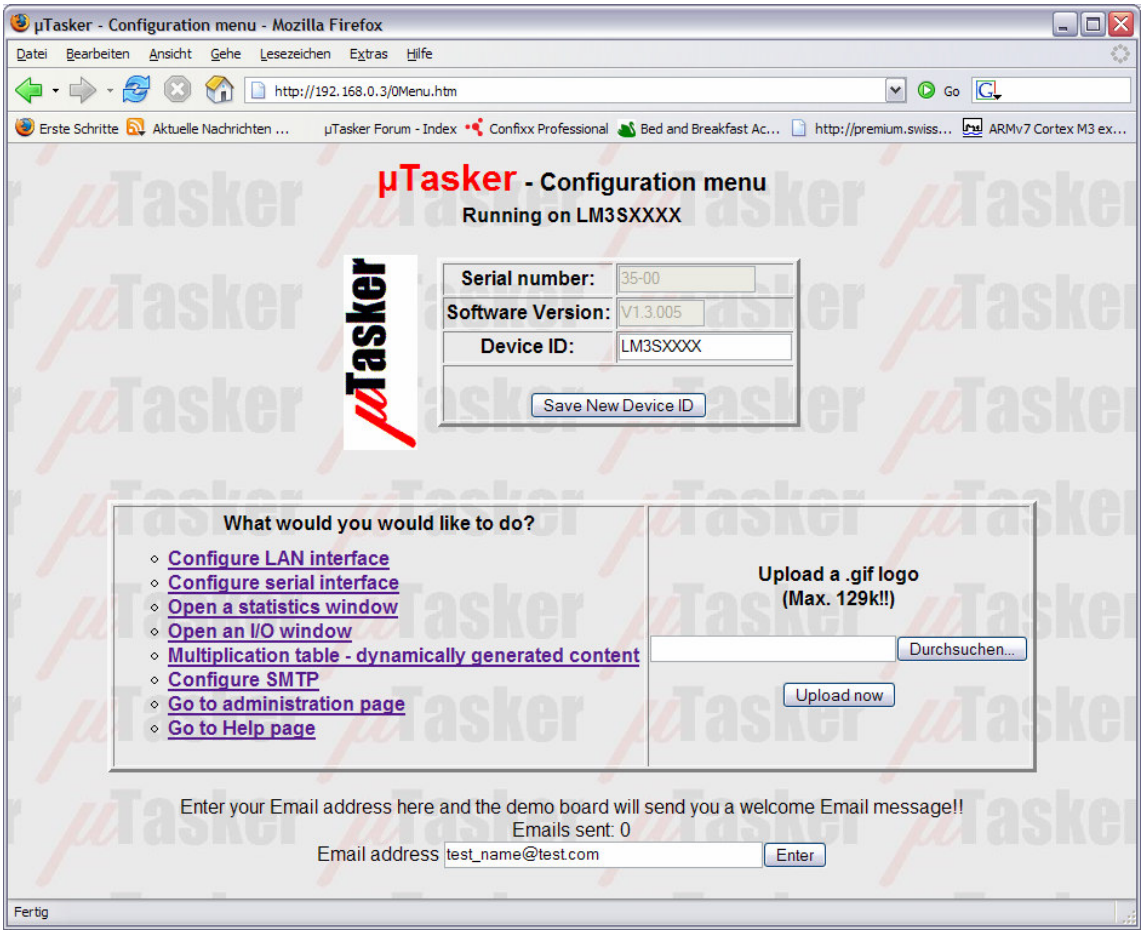
Are you surprised that the target behaves the same as the simulator? You shouldn't really be because that is exactly what the simulator is all about. It allows you to test your real code in real time and once it is working as you want it to, then you can transfer it to the real target. In fact you will find that your real target is not really necessary for most of your development work. Develop on the simulator and ship on the target – that is the way to do things really efficiently.

Note that the I/O page allows the status LED on the Luminary evaluation boards to be controlled and the state of it to be displayed. If PF0 *is reconfigured to be an input the blinking LED will no longer light.*

A tour through the demo web server

Now that you have had the possibility to simulate the demo and also run it on the target hardware it is time to take a more detailed look at what it does and then how it all works. The demo is designed to give you a practical platform to use as a basis for your own developments so it would be good to know how it can be best modified to suite your own needs.

Start side



Serial number – this is a decimal representation of the MAC address programmed in the device. Since the MAC address is normally unique it can serve also as a serial number if required. Notice that the serial number is displayed in grey and can not be modified (disabled).

Software version – this is a string which is defined in **application.h**. It is also disabled so that it can not be modified via the web side.

Device ID – this is a string defined in the `cParameters` structure in **application.c**. The default value can be set in code and also modified via the web page. When 'Save new device ID' is clicked, the new value is also saved to a parameter block in the internal flash so that it can be displayed after the next reset or power up.

*If the option USE_NETBIOS (in config.h) is enabled the device ID will also be used to recognise the device on the network. In this case it can be pinged by using "**ping LM3SXXXX**" rather than "**ping 192.168.0.3**". This also especially useful when DHCP is enabled to obtain an IP address so that communication with the device is possible even if the obtained network settings are not known.*

The name is also valid when browsing to the device. LM3SXXXX as URL entry will connect with the corresponding IP address.

The name of the device should not be longer that 15 characters for NetBIOS to be able to use it!!

Menu – There are links to several other web pages allowing specific configurations to be displayed and modified.

Logo and Upload test – The demo enables the logo shown on the start side to be modified. The logo is a small GIF file which was loaded together with the HTM web pages earlier on. If you look in the web page directory for the LM3SXXXX you will see a further folder called `Alternative_Logo`. This contains another small GIF file (it has to be less that 129k to fit in the space reserved for it in the file system) which you can immediately use to see this working.

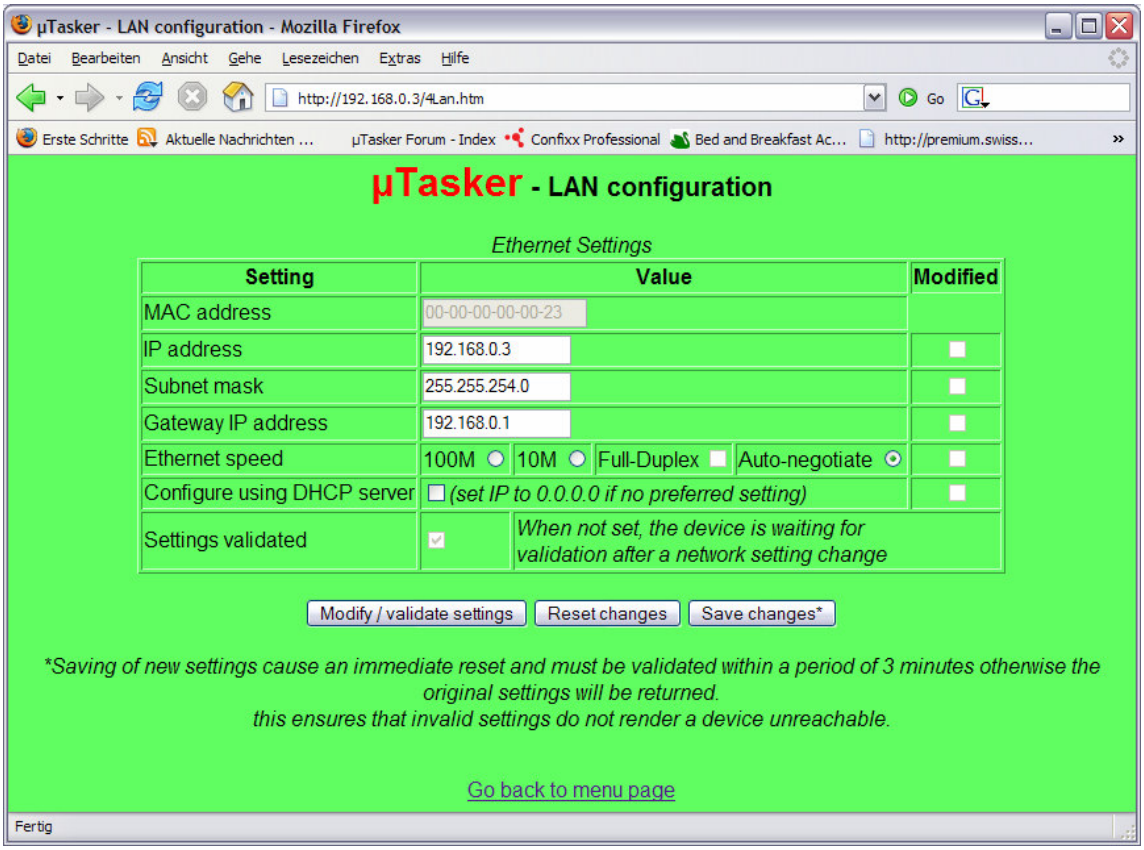
Click on the button to search for the file – Windows will open up an explorer menu and you can go to the "`Alternative_Logo`" directory and select the file called "`wow.gif`".

Finally click on the "Upload now" button and your Browser will use the HTTP post method to send this file to the embedded web server. The uploaded file will be saved to FLASH so that it will always be displayed as logo on this page. If you would like to experiment with other small files then feel free to do so – just ensure that they are not any larger than 129k in size.

Email Address – The Email entry field is active when the SMTP support is enabled (USE_SMTP in config.h). This will enable you to enter an email address and have the demo send a welcome message to it. This is described in detail in the document "uTaskerSMTP.doc". The menu item "Configure SMTP" is also explained in this document. In the case of the LM3SXXXX demo this web page is permanently available.

Note that the start page also includes a background JPG.

LAN configuration



This side is a very critical side since it allows important Ethernet settings to be modified. The first time the device is started, the default MAC address – defined in the `cParameters` structure in **application.c** - is 0-0-0-0-0-0. It can be changed just once, after which it is displayed as disabled. This is because it is normally necessary to program a new device with a unique MAC address which should normally never be changed again.

The IP address, subnet mask and default gateway IP address can be modified as long as the device is not set to DHCP operation. If they are first modified and then the device is set to DHCP mode, they represent preferred settings to be requested during the DHCP procedure or for use as a last resort if no DHCP server is available.

The Ethernet speeds of 10M, 100M or auto-negotiation allow either fixed speed operation or auto-negotiation mode. Generally auto-negotiation is practical. If a fix speed is set it is also possible to choose between full-duplex and half-duplex operation. Full-duplex is generally only used when there are only two directly connected devices on the network.

Values which are not disabled, can be modified and accepted by clicking on “Modify / validate settings” and changed values are indicated by a check box in the configuration table. They have not yet been committed to memory and the previous setting can be returned by clicking on “Reset changes”.

Once you are sure that the modifications are correct, they are saved to flash memory by clicking on “Save changes”. This will cause also a reset of the device after a short delay of

about 2 seconds and it will be necessary to establish a new connection to the device using its new settings and validate the changes.

So why make it so complicated? Well it is simply to be sure that you have not modifying something which will render the device unreachable. For example, you may have changed its Ethernet speed from 100M to 10M although it is connected to a hub which only supports 100M and, to make matters worse, the device is not simply sitting on your office desk at arm's length but is at a customer's site several hours drive away. It would be a nasty situation if you had just lost contact with the equipment and have to somehow get it back on line although everyone at the customer's site has already left for a long weekend…

So this is where the validation part comes in. After the reset, the device sees that there are new settings in memory but that these are only provisional (not validated), the original values are also still available. These new values are nevertheless used and a timer of three minutes duration started. It is now your job to establish a new web server connection to the device, using the new settings and to click on "Modify / validate settings". If you do this the check box named "Settings validated" will be checked, the new parameters are validated in flash and the old ones deleted. This in the knowledge that the new values are also really workable ones – how would you otherwise have been able to validate them?

Imagine however that a change really rendered the device unreachable – for example the LAN speed was really incompatible or you made a mistake with the subnet mask. After three minutes without validation, the provisional values are deleted from flash and, after a further automatic reset, the original settings are used again. After a short down period you can then connect as before the change and perform the modifications again, though correcting the previous mistake. There is therefore no danger of losing contact with the device, even when a mistake is made.

If you are curious about how the parameters work there is a description of them in the document "`uTaskerFileSystem.doc`" in the documents directory. Also check out the µTasker web site to see whether there are new releases of this and any other documents.

Some notes about MAC addresses

If you are using your own device behind a router and it is not visible to the 'outside world' you can in fact program any MAC address that you like because it is in a private area. It just has to be unique in this private area. (This is also valid for a device sitting in a Demiliterized Zone- DMZ)
If however you are selling a product or the device is sitting directly on the Internet then it must have a world wide unique MAC address which has to be purchased from IEEE. It is purchased either as a block (IAB) of 4k MAC addresses at a cost of about $500, or if you are going to produce a large number of pieces of equipment you can purchase a unique company ID (OUI) of 16Million for about $1'600 (plus $2'000 if you don't want the OUI to be registered on the public listing).

It is then your responsibility to manage the assignment of these addresses in your own products.

The registration page is at: http://standards.ieee.org/regauth/index.html

For any one just wanting to make one or two pieces of equipment for hobby use it is a bit much to pay $500 for a bunch of MAC addresses and use just one or two of them. Unfortunately it is no allowed to sell the rest on to people in similar situation because a block must always remain with the individual or organisation purchasing it.

One trick which is often used is to find out what the MAC address is in an old NIC from an old PC which is being scrapped. This MAC address is then used in your own piece of equipment and the old NIC destroyed. You can then be sure that the MAC address is unique and can not disturb when used for any imaginable application.

Note about manual network setting and NetBIOS

The NetBIOS protocol uses local sub-net broadcasts. For this to operate correctly the sub-.net mask used in the configuration must match with the sub-net mask of the local network.

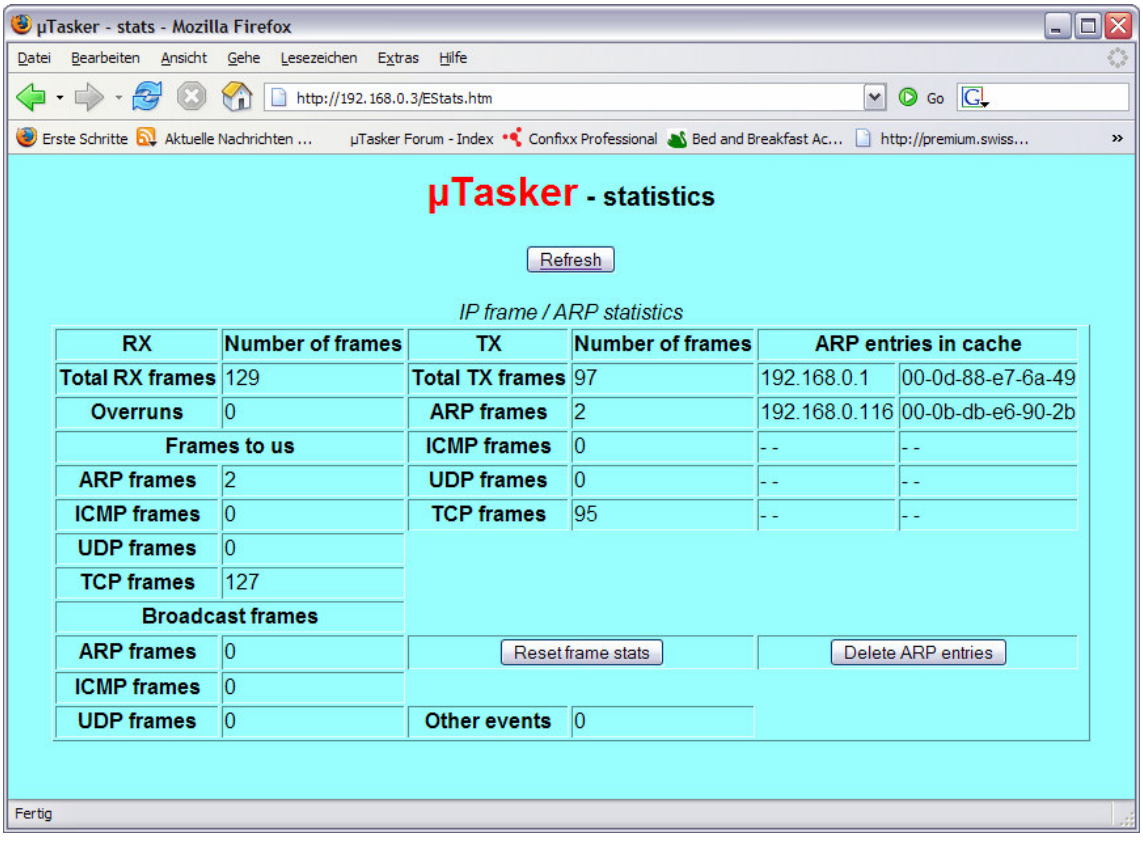
Serial configuration

This web side allows UART0 in the LM3SXXXX to be set. I won't discuss this in any detail here because the serial interface is the topic of a later tutorial, where we will see that the UART can also be fully simulated on the PC. The LM3S6965 contains 3 UARTs which are all supported by the project code.

Statistics

The statistics window opens in a separate window and displays a table of the number of Ethernet frames counted by the Ethernet task. These are classified according to whether they were transmitted, received for the local IP address or received as broadcast frames and divided into IP protocol types. Since the values can continue to change, the user can update the window by clicking on the "Refresh" button. It is also possible to reset the counters if desired by clicking on "Reset frame stats".

The contents of the ARP cache is also displayed and can be cleared by clicking on "Delete ARP entries". These are equivalent to the DOS commands `arp -a` to display the local ARP cache and `arp -d` to delete it. Notice that when the ARP cache is deleted via the web page there will always remain one entry, this being the PC which commanded the deletion of the ARP table and updated the table. Due to the network activity it will always immediately be re-entered…



I/O window

The I/O window was discussed in step 11 of the tutorial and will not be discussed in any more detail here.

Administration side

On this page the FTP server can be activated and deactivated. See also step 12 of the tutorial.

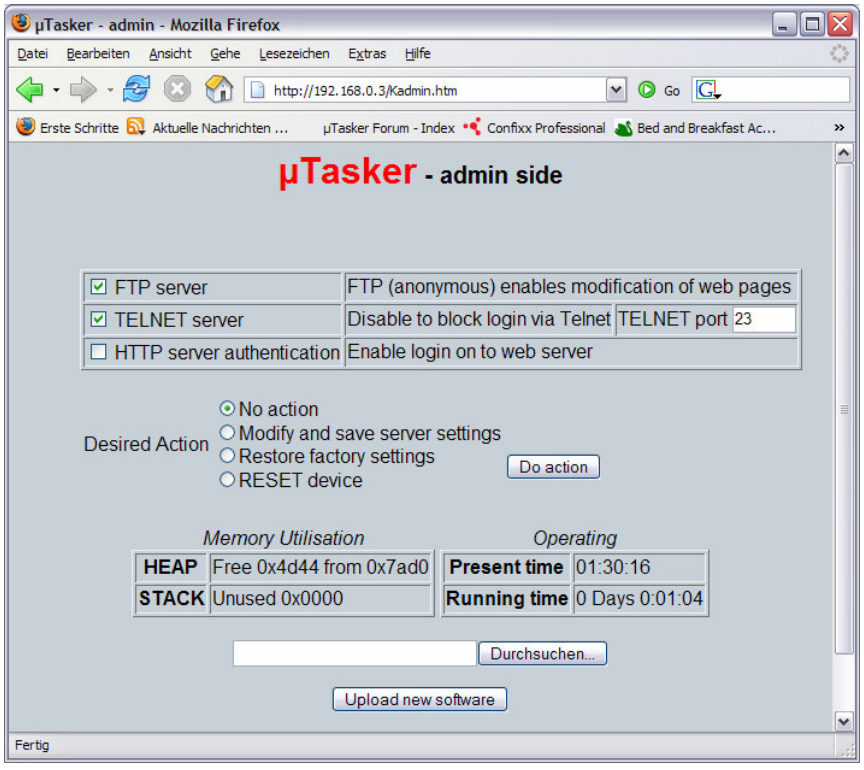
The HTTP server can be configured to require basic authentication, meaning that it requests the user name and password when a new connection is established. These values are contained in the default settings in **application.c.**

A TELNET server can also be enabled or disabled, including setting it to a particular port number. This will not be discussed here further since it is the subject of a later tutorial.

The previous three settings are modified and saved by setting the desired action to “Modify and save server settings” and clicking on “Perform desired action”.

A further action is “Restore factory settings” which returns the default settings as defined in **application.c**. This also provokes a reset of the device and if the default settings cause the network values to be modified it will automatically start a three minute validation period meaning that it will also be necessary to establish a connection using the ‘factory settings’ and validate them otherwise the original ones will be restored. This is again to ensure that there is no danger of loosing contact with a remote device. Note that the MAC address is not reset since the MAC address should always remain unchanged in the device.

The last action is a simple reset of the device.



It is possible to upload new software as long as the project is compiled with the boot loader support. See the boot loader documentation for details of how to activate and use this.

Memory Utilisation

HEAP – the used heap and the maximum defined heap are displayed. Note here that the simulator will often display that more heap is used as needed in the real target due to the fact that some variables will be of different sizes when running on a PC compared to an embedded target. It allows the real use on the target to be monitored and the user can then optimise the amount of heap allocated to just support the worst case.

STACK – the amount of stack which has not been used (safety margin) is measured and displayed. As long as the value remains larger that 0 the system is not experiencing any memory difficulties. A value of zero indicates a critical situation and must be reviewed. The simulator however doesn't perform this stack monitoring and will always display zero; its use is on the target where resources need to be monitored carefully.

Operating

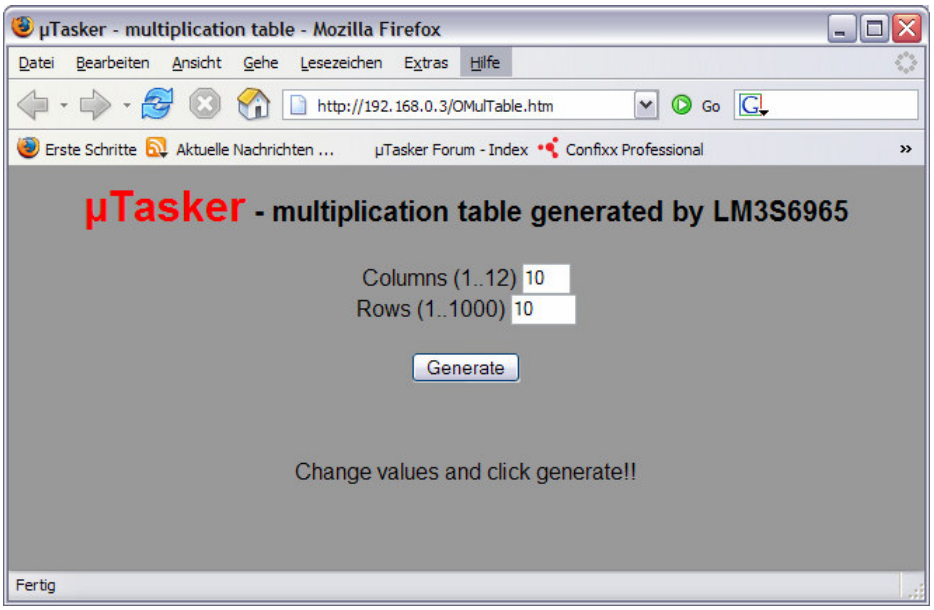
The time since the device started, or since the last reset, is displayed. This is useful when monitoring a remote device to ensure that it is indeed operating stably since a watchdog reset due to a software error would allow the device to recover but possibly not enable its occurrence to be readily detected.

Present time

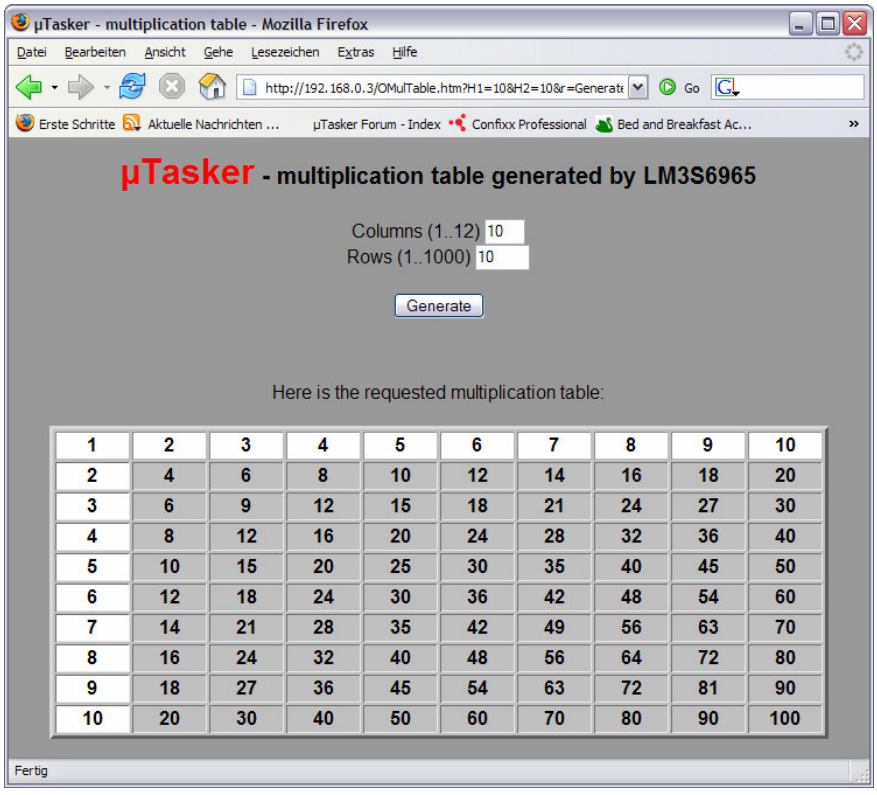
The present time is displayed if it is known. For example if the project is supported by a RTC (Real Time Clock). It is also displayed when the project is configured to use the Time Server protocol and there is a connection with the Internet via the default gateway. This is discussed later in this document.

Multiplication table – dynamically generated content

The menu item opens a special web page which demonstrates how the project can generate dynamic content.



The screen shot above shows how the page looks when it is opened the first time. When the "Generate" button is pressed the page refreshes but looks like this:



1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100

The values for Columns and Rows have been used to generate a multiplication table with these dimensions. The demonstration accepts column values from 1 to 12 and row values from 1 to 1000 and shows how user code can quite easily interface to the HTTP server and generate its own HTML (or Java script etc.) content.

Further demo project support

The demo project includes various other stuff, which is not described here in any great detail - apart from mentioning the main ones that are there. These can be simply removed if they are not of interest or to save code space.

All defines can be found in *config.h.*

#define SUPPORT_LCD

This activates an LCD demonstration.

#define SERIAL_INTERFACE

This activates serial interface support where UART 1 is configured. Connecting at 19'200 and hitting the enter key from a terminal emulator will result in a menu being displayed which allows various configurations to be adjusted.

#define USE_TELNET

A Telnet server allows the same menu as mentioned under the serial interface option to be displayed by connecting via Telnet.

Code sizes

(Code / const / RAM)	IAR compiler V4.42A on LM3S6965 (highest level of optimisation for smallest size)	GNU compiler on LPC2378 as used by Rowley Crossworks Release 1.7 Build 7 GCC 4.1.1 (highest level of optimisation for smallest size) (const + RAM counted together)
Application, menu and other demos	**5'532 / 4'260 / 177**	**7'228 / 4'273**
UART driver and demo	**2'868 / 857 / 27**	**4'512 / 880**
SMTP	**1'204 / 119 / 16**	**1'352 / 121**
TELNET	**1'184 / 5 / 4**	**1'168 / 11**
DNS	**846 / 28 / 16**	**932 / 26**
Application Web demo code	**2'682 / 1'164 / 100**	**3'288 / 1'254**
Support utilities for Web interface	**788 / 0 / 0**	**900 / 0**
HTTP	**2'768 / 672 / 28**	**2'988 / 668**
FTP	**1'716 / 476 / 28**	**2'064 / 474**
TCP	**4'720 / 10 / 13**	**5'112 / 20**
NetBIOS	**264 / 4 / 4**	**288 / 9**
DHCP	**1'962 / 64 / 32**	**2'220 / 60**
UDP	**768 / 2 / 4**	**924 / 6**
ICMP	**354 / 0 / 0**	**376 / 0**
Support utilities for IP	**372 / 0 / 0**	**428 / 0 / 0**
IP	**714 / 2 / 2**	**852 / 4**
ARP	**1'492 / 8 / 4**	**1'504 / 10**
LM3SXXXX HW	**3'024 / 336 / 77**	**3'552 / 380**
Ethernet interface	**818 / 0 / 81**	**924 / 84**
File system and parameters	**708 / 0 / 8**	**796 / 8**
Operating system and driver support	**2'720 / 1 / 45**	**3'480 / 45**
µTasker demo Total	**38'502 / 8'008 / 656 (FLASH = 46'510)**	**43'544 / 7'632 / 1'288 (FLASH = 51'176)**

The RAM values are for static RAM and do not include the HEAP requirements, which is about 13 k bytes for the complete demo project, using the default configuration settings. The FLASH size is purely code and doesn't include the file system and parameter system which can occupy the free space – the file system is used for the web pages which are loaded during the demo.

The settings of certain protocols can be further influenced by their individual configuration – the default configuration of the demo project was used as is and represents probably the most luxurious configuration.

The author used map file information to deduce these values and assumes no responsibility for errors or omissions.

Comparison with LPC2378 processor in Thumb mode
- IAR 48k FLASH / 1k RAM
- GCC 55.5k FLASH / 1.5k RAM

Developing - Testing - Debugging

The demo project is designed to demonstrate a typical use of the operating system and TCP/IP stack. It is useful in itself as a starting point for many applications and this section looks at the support for developing, testing and debugging your own applications.

Until now you haven't actually had to develop anything since the project is delivered fully functional. To develop your own project it will be necessary to make configuration changes to suit your own use, to change code and add code of your own. To learn more about these aspects it is possible to study the documentation about the µTasker operating system and protocol stack. A more hands on approach is also possible by letting the demo project run and walking through the code parts which you are interested in – we did this briefly at the start of the demo but didn't linger to discuss any details. Here we will check out the advantages of the simulator by looking in more depth at a rather more complicated debugging session.

Debugging is performed for a number or reasons. It is a natural consequence of the test phase where unexpected program behaviour is experienced and the causes and reasons need to be understood before correcting the code. It is often also an integral part of the test phase itself when code reviews are performed, exception handling is to be exercised or software validation is required. The simulator allows a high degree of tests to be performed in comfort before going to the target testing phase, where such reviews would be rather more complicated.

So let's test something in the demo project. We'll test the simple PING ECHO utility which we already used once and we'll see how we can get to know the software in a very convenient and efficient manor. We will see how we can manipulate the operation to test and validate special cases and to make corrections in the code (and verify them too). First we will work ON LINE with the simulator, meaning that the simulator will be running effectively in real time and we will capture and analyse events. Afterwards we will see how to do the same OFF LINE, using a recording of the first case (which could also be a recording made when using a real target).

Introducing Ethereal/Wireshark

If you are using the µTasker then you will certainly be wanting to make use of its network capabilities and a tool to monitor network activity is essential. These are often called Network Sniffers since they can monitor, record and analyse network activity by watching what happens on the local Ethernet connection. The µTasker was designed with and around Ethereal/Wireshark, a free and powerful network Sniffer. You can download this from http://www.ethereal.com/ or http://www.wireshark.org/, I use the version 0.10.8.0 of Ethereal but there are newer one's available.

If you don't have this program then don't delay – download it and install it and then we will get down to some work.

Now do the following steps:

1. Start the demo project simulation as described in the first tutorial.

2. Open a DOS window and prepare the PING command “ping 192.168.0.3”

3. Start Ethereal/Wireshark and start monitoring the traffic on your local network (Capture | Start -> OK).

4. Enter return in the DOS windows so that the PING test is started.

5. Wait until the PING test has completed; there are normally 4 test messages sent.

6. Stop the Ethereal/Wireshark recording and save it to the directory Applications\uTaskerV1.3\Simulator\Ethereal with the name ping.eth

7. Make a copy of the file ping.eth in the directory Applications\uTaskerV1.3\Simulator and rename it to Ethernet.eth

At this stage you can also look at the contents of the Ethereal/Wireshark recording and you will see something like the following:

```
No. Time        Source            Destination     Prot  Info
1   0.437770    192.168.0.100     Broadcast       ARP   Who has 192.168.0.3? Tell192.168.0.100
2   0.439926    00:11:22_33:44:55 192.168.0.100   ARP   192.168.0.3 is at 00:11:22:33:44:55
3   0.439933    192.168.0.100     192.168.0.3     ICMP  Echo (ping) request
4   0.449898    192.168.0.3       192.168.0.100   ICMP  Echo (ping) reply
5   1.441345    192.168.0.100     192.168.0.3     ICMP  Echo (ping) request
6   1.441650    192.168.0.3       192.168.0.100   ICMP  Echo (ping) reply
7   2.442782    192.168.0.100     192.168.0.3     ICMP  Echo (ping) request
8   2.442915    192.168.0.3       192.168.0.100   ICMP  Echo (ping) reply
9   3.444225    192.168.0.100     192.168.0.3     ICMP  Echo (ping) request
10  3.444344    192.168.0.3       192.168.0.100   ICMP  Echo (ping) reply
```

It is also very possible that there are other frames on the network from other computers or your own computer talking with others and it is possible to perform many filtering functions to remove these either from the visible display or from the recoding file – just look in the Ethereal/Wireshark Help.

Here my PC has the local address 192.168.0.100 and initially doesn’t know how to find the destination 192.168.03. An ARP resolve is sent and the simulator responds to it, informing at which MAC address its IP can be found at.

The ping test is repeated four times, each time receiving a reply from the simulator.

So let’s see in some more detail how we can check the operation of this procedure and we will begin at the deepest point in the code, the receiving interrupt routine in the Ethernet driver, which is called when a complete Ethernet frame has been received.

In the VisualStudio project, locate the file LM3SXXXX.c (in hardware\LM3SXXXX\) and search for the Ethernet reception interrupt routine called *EMAC_Interrupt()* and put a break point in interrupt routine by positioning the cursor at the start of the routine and pressing F9.

It may be that you will find the code stopping at this break point before starting the ping test, which is probably because the code is receiving broadcast frames from your network. If this

happens before you start the ping test just press F5 to let it run again (which you may have to repeat if there is a lot of activity…).

Repeat the ping test from the DOS window and the execution will stop at the breakpoint which you set in the interrupt routine. Since there will be no answer to the ping test as our code has been stopped, the ping test(s) will fail, but we don't worry about that since we have caught the event which we are going to use to analyse the flow through the driver, memory, the operating system and the ICMP routine. Now we can get to know the code in as much detail as we want…..and since we are working with the LM3SXXXX, this will also give us the opportunity to look at some of its internal registers on the way.

A. Interrupt routine

```
// EMAC interrupt displatcher
//
__interrupt void EMAC_Interrupt(void)
{
    unsigned long ulMask = (unsigned long)MACIM;                          // EMAC
interrupt mask
#ifdef _WINDOWS
    extern unsigned char fnGetFifo_byte(void);
    #define GET_FIFO_LONG() (fnGetFifo_byte()) | (fnGetFifo_byte() << 8) |
(fnGetFifo_byte() << 16) | (fnGetFifo_byte() << 24)
#else
    #define GET_FIFO_LONG() MACDATA
#endif
    while ((MACRIS_IACK & ulMask) != 0) {                                 // while
enabled interrupts are waiting
#ifdef LAN_REPORT_ACTIVITY
        unsigned char EMAC_int_message[ HEADER_LENGTH ] = { 0, 0,
INTERRUPT_TASK_LAN_EXCEPTIONS, INTERRUPT_EVENT, EMAC_TX_INTERRUPT };
#endif
        if (MACRIS_IACK & (RXINT | RXER | FOV)) {                         //
reception frame available
            if (MACRIS_IACK & (RXER | FOV)) {                             // error
                unsigned char EMAC_err_message[ HEADER_LENGTH ];          // = { 0,
0, INTERRUPT_TASK_LAN_EXCEPTIONS, INTERRUPT_EVENT, ETHERNET_RX_ERROR };
                EMAC_err_message[MSG_DESTINATION_NODE] =
EMAC_err_message[MSG_SOURCE_NODE] = 0;
                EMAC_err_message[MSG_DESTINATION_TASK] =
INTERRUPT_TASK_LAN_EXCEPTIONS;
                EMAC_err_message[MSG_SOURCE_TASK] = INTERRUPT_EVENT;
                EMAC_err_message[MSG_INTERRUPT_EVENT] = ETHERNET_RX_ERROR;
                if (MACRIS_IACK & FOV) {
                    EMAC_err_message[MSG_INTERRUPT_EVENT] = ETHERNET_RX_OVERRUN;
                }
                uDisable_Interrupt();                                     // ensure
message can not be interrupted
                fnWrite(INTERNAL_ROUTE, EMAC_err_message, HEADER_LENGTH);// inform
the Ethernet task
                uEnable_Interrupt();                                      // release
            }
            else {
                unsigned long *ptrRxBuffer = (unsigned long
*)ptrEthRxBuffer[iRxEthBufferPut]; // buffer is aligned
                unsigned long ulMacRxFrameLength = GET_FIFO_LONG();
                if (usEthRxValid[iRxEthBufferPut] != 0) {                 // if the
user has not yet retrieved last buffer, we reject latest frame and reset the Rx
FIFO
                    unsigned char EMAC_err_message[ HEADER_LENGTH ];      // = { 0,
0, INTERRUPT_TASK_LAN_EXCEPTIONS, INTERRUPT_EVENT, ETHERNET_RX_ERROR };
                    EMAC_err_message[MSG_DESTINATION_NODE] =
EMAC_err_message[MSG_SOURCE_NODE] = 0;
                    EMAC_err_message[MSG_DESTINATION_TASK] =
INTERRUPT_TASK_LAN_EXCEPTIONS;
                    EMAC_err_message[MSG_SOURCE_TASK] = INTERRUPT_EVENT;
```

```
                    EMAC_err_message[MSG_INTERRUPT_EVENT] = ETHERNET_RX_ERROR;
                    MACRCTL |= MAC_RSTFIFO;                                 // flush
FIFO
                    uDisable_Interrupt();                                   // ensure
message can not be interrupted
                    fnWrite(INTERNAL_ROUTE, EMAC_err_message, HEADER_LENGTH);//
inform the Ethernet task
                    uEnable_Interrupt();                                    // release
                    MACRCTL &= ~MAC_RSTFIFO;
                }
                else {
                    *ptrRxBuffer++ = ulMacRxFrameLength;                    // put the
length and first 2 bytes into the buffer
                    ulMacRxFrameLength &= 0x0000ffff;                       // mask
out the length
                    usEthRxValid[iRxEthBufferPut] = (unsigned
short)ulMacRxFrameLength; // put the length of the received frame in the buffer

                    while (1) {
                        *ptrRxBuffer++ = GET_FIFO_LONG();                   // copy
all data from RX FIFO
                        if (ulMacRxFrameLength <= sizeof(unsigned long)) {
                            break;
                        }
                        ulMacRxFrameLength -= sizeof(unsigned long);
                    }
                    if (++iRxEthBufferPut >=
NUMBER_OF_RX_BUFFERS_IN_ETHERNET_DEVICE) {
                        iRxEthBufferPut = 0;
                    }
                    uDisable_Interrupt();                                   // ensure
message can not be interrupted
                    fnWrite(INTERNAL_ROUTE, (unsigned char*)EMAC_RX_int_message,
HEADER_LENGTH); // inform the Ethernet task
#ifdef LAN_REPORT_ACTIVITY
                    EMAC_int_message[MSG_INTERRUPT_EVENT] = EMAC_RX_INTERRUPT;
                    fnWrite(INTERNAL_ROUTE, (unsigned char*)EMAC_int_message,
HEADER_LENGTH); // inform the task of event
#endif
                    uEnable_Interrupt();                                    // release
                }
            }
            MACRIS_IACK = (RXINT | RXER | FOV);                             // reset
rx flags
#ifdef _WINDOWS
            MACRIS_IACK &= ~(RXINT | RXER | FOV);                           // clear
interrupt flags when simulating
#endif
        }
```

The simulator has just received a frame for our MAC address or a broadcast address (the simulator doesn't disturb us with foreign MAC addresses when the EMAC in the LM3SXXXX has not be set up for promiscuous operation) and here we are in the interrupt routine. Now don't forget that this is the real code which will operate on your target and if this doesn't work properly here it will also not work properly on your target.

We see that such routines are necessarily quite hardware specific. Here we see first accesses to the internal registers, MACIM and MACRIS_IACK. Search for these registers in the LM3SXXXX data sheet if you want to read exactly how it works; here is a very quick overview where we will also look at the registers in the simulated LM3SXXXX.

MACRIS_IACK is the raw interrupt status register in the EMAC. We can search for it in our project by using "search in files", ensuring that the search path starts at the highest level in the µTasker project, and using the search files of type *.c and *.h.

It will be found in the code at several locations but also in the file LM3SXXXX.h, where it is defined as:

```
#define MACRIS_IACK    *(volatile unsigned long*)(ETHERNET_BLOCK + 0x00)
                      // Ethernet MAC Raw Interrupt Status / Interrupt Acknowledge
```

where `ETHERNET_BLOCK` is also define locally twice:

```
#define ETHERNET_BLOCK               ((unsigned char *)(&LM3Sxxxx.ETHERNET))
#define ETHERNET_BLOCK               0x40048000
```

On the target the register is located at long word address `0x400480000` and when simulating it can be found in a structure called `LM3Sxxxx`.

Double click on the word `LM3Sxxxx` so that it becomes highlighted and then drag it into the watch window (Shift F9 opens a quick watch window). Expand the structure by clicking on the cross to the left of the name and you will see the various internal peripherals divided into sub-blocks. *MACRIS_IACK* is in the sub-block ETHERNET so we also expand this sub-structure by clicking on the cross left of its name.

In the sub-block you will see all of the EMAC registers listed in the order in which they occur in memory as well as showing their present values. Step the code (F10) and you will see that the interrupt cause is recognised as a reception frame; the length of the received frame is extracted from the EMAC Rx FIFO before its content is also retrieved from the RX FIFO. Once the frame content has been extracted locally an internal message is sent, defined by

```
static const unsigned char EMAC_RX_int_message[ HEADER_LENGTH ] = { 0, 0 ,
TASK_ETHERNET, INTERRUPT_EVENT, EMAC_RX_INTERRUPT };    // define fixed interrupt
event
```

This internal message is sent to TASK_ETHERNET to inform that a frame has been received and is as such a method used by the operating system to wake up this Ethernet task. If you want to, you can also step into the write function to see the internal workings of how it wakes up the receiver task but I will assume here that all is working well and will just move to TASK_ETHERNET to see it actually receiving this interrupt message.

Before doing the next step, remove the break point in the interrupt routine by setting the cursor to its line and hitting F9.

B. Receive Task

Now open the file ethernet.c in the project directory TCP/IP and set the cursor to the bracket after the tasks routine `void fnTaskEthernet(TTASKTABLE *ptrTaskTable)`

With a right click, the menu item “Run to cursor” can be executed and you will see that this task is indeed started immediately since the program execution stops at this line.

Step slowly through the code using F10 and observe that the task reads the contents and interprets it as an INTERRUPT_EVENT. It calls fnEthernetEvent() to learn about the length of the received frame and to get a pointer to it – use F11 to step into this routine if you would like to see its internal workings, which I will not discuss here.

You will see that the pointer to the frame is called rx_frame and it is defined as a pointer to an ETHERNET_FRAME. Now, as we all know, a pointer is just a pointer – but the fact that it is defined as a pointer to a certain structure will help us greatly to interpret the contents of the

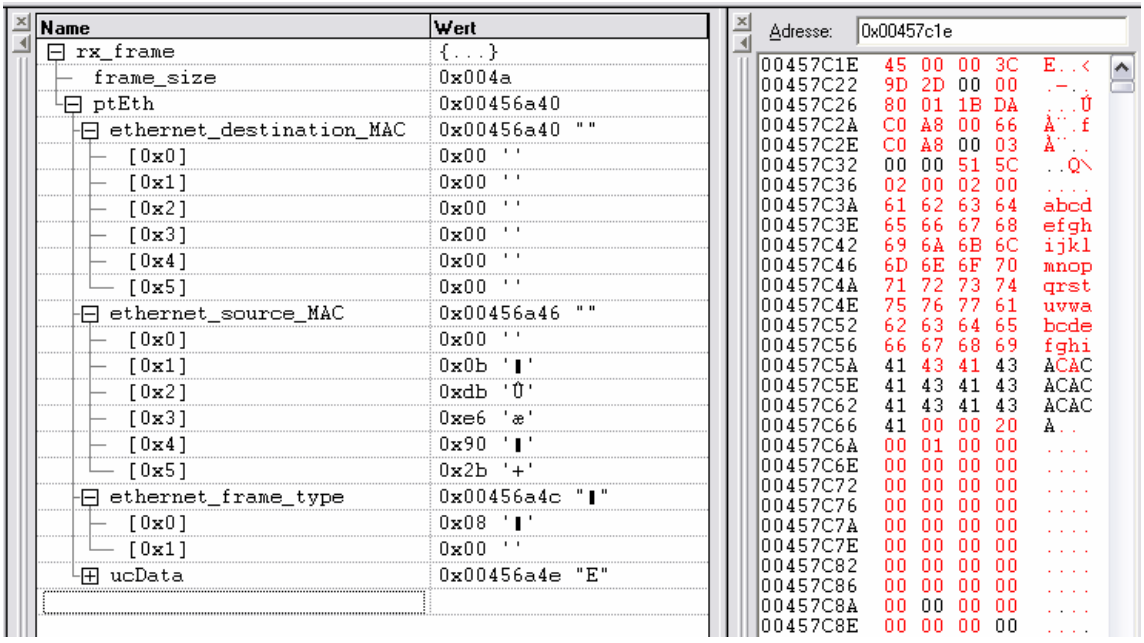
received frame. We will first look at the frame as it is in memory and then how it looks in the debugger as a structure.

C. Receive Frame

Let's start with simply looking at the raw data which we have just received from the Ethernet. Do this by placing rx_frame in a watch windows and expanding it so that the length of the frame (frame_size) and the pointer to the data (ptEth) are visible. ptEth has a value which you can double click on and insert to the clip board before pasting it into the address field of a memory display window (if this windows is not yet visible, activate it in the debug menu). Now you will see the raw Ethernet frame in memory – note that this is in the local computer's memory and the address of its location has nothing to do with the storage place on the real target. However its location is governed by the buffer descriptors in an analogue fashion to the operation in real device.

Now it is not exactly easy to understand the data but you should just be able to make out the MAC address at the beginning and the content of the ping test usually consists of abcdefghijk… which is easily recognised. (Warning: it is possible that the frame which you actually captured is not the ping test but some other network activity, or even an ARP frame if the PC sending the ping had to re-resolve its MAC address. If this happens to be the case, set a break point at the location which the program is at and let it run again and hopefully it will come to this location the next time with the correct contents…).

Go back to the watch windows where the structure of rx_frame is being displayed and expand the sub-structure ptEth. This will start making life rather easier since it is showing us that the Ethernet frame is made up of a destination MAC address, a source MAC address, an Ethernet frame type and some data. Expand each sub-structure to see their exact contents, although it is not yet worth expanding the data field since we don't know what protocol its contents represent so it will not be displayed any better at the moment.



Name	Wert
⊟ rx_frame	{...}
frame_size	0x004a
⊟ ptEth	0x00456a40
⊟ ethernet_destination_MAC	0x00456a40 ""
[0x0]	0x00 ''
[0x1]	0x00 ''
[0x2]	0x00 ''
[0x3]	0x00 ''
[0x4]	0x00 ''
[0x5]	0x00 ''
⊟ ethernet_source_MAC	0x00456a46 ""
[0x0]	0x00 ''
[0x1]	0x0b '▮'
[0x2]	0xdb 'Û'
[0x3]	0xe6 'æ'
[0x4]	0x90 '▮'
[0x5]	0x2b '+'
⊟ ethernet_frame_type	0x00456a4c "▮"
[0x0]	0x08 '▮'
[0x1]	0x00 ''
⊞ ucData	0x00456a4e "E"

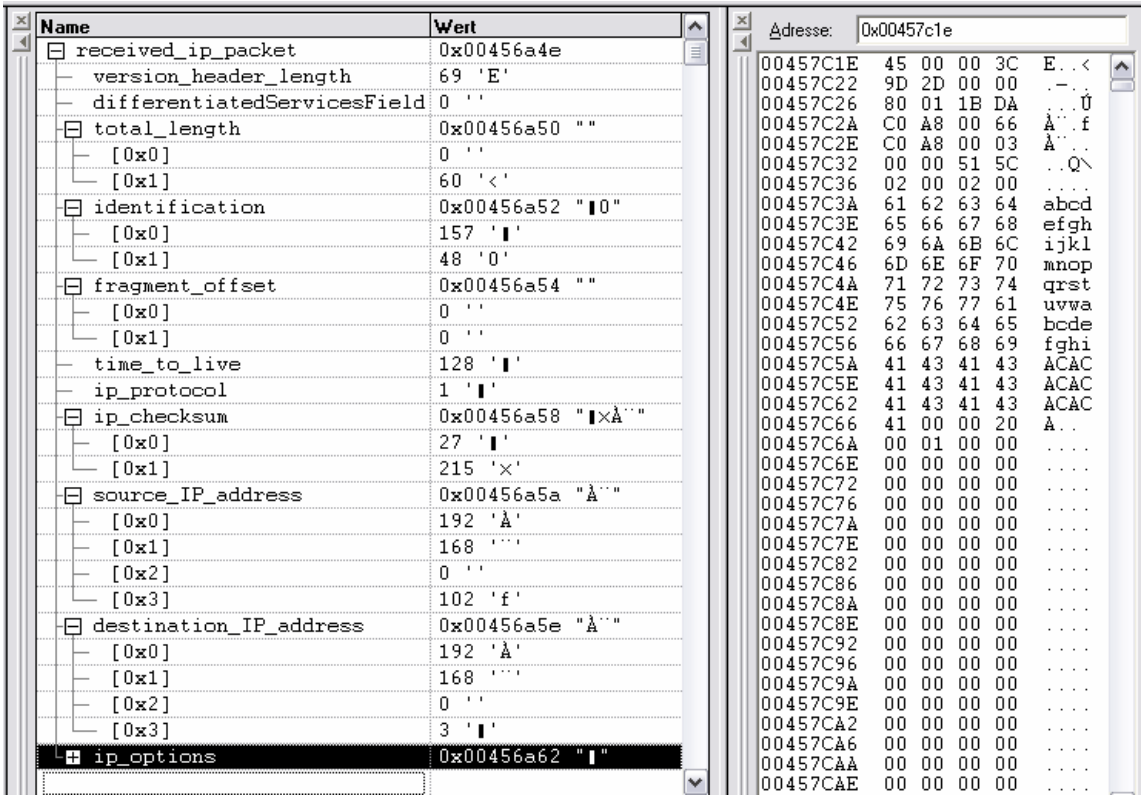
Adresse: 0x00457c1e

```
00457C1E  45 00 00 3C  E..<
00457C22  9D 2D 00 00  .-..
00457C26  80 01 1B DA  ...Û
00457C2A  C0 A8 00 66  À¨.f
00457C2E  C0 A8 00 03  À¨..
00457C32  00 00 51 5C  ..Q\
00457C36  02 00 02 00  ....
00457C3A  61 62 63 64  abcd
00457C3E  65 66 67 68  efgh
00457C42  69 6A 6B 6C  ijkl
00457C46  6D 6E 6F 70  mnop
00457C4A  71 72 73 74  qrst
00457C4E  75 76 77 61  uvwa
00457C52  62 63 64 65  bcde
00457C56  66 67 68 69  fghi
00457C5A  41 43 41 43  ACAC
00457C5E  41 43 41 43  ACAC
00457C62  41 43 41 43  ACAC
00457C66  41 00 00 20  A..
00457C6A  00 01 00 00  ....
00457C6E  00 00 00 00  ....
00457C72  00 00 00 00  ....
00457C76  00 00 00 00  ....
00457C7A  00 00 00 00  ....
00457C7E  00 00 00 00  ....
00457C82  00 00 00 00  ....
00457C86  00 00 00 00  ....
00457C8A  00 00 00 00  ....
00457C8E  00 00 00 00  ....
```

D. Frame protocol

Step slowly using F10 and you will see that the frame is checked for the ARP protocol, which it won't be if it is the ping request which we are analysing. It then calls fnHandleIP(), which we will enter using F11 since all such TCP/IP frames are built on the IP protocol.

After a couple of basic checks of IP frame validity, the pointer received_ip_packet is assigned to the data part of the received frame. Again this structure is very valuable to us since it can be dragged into the watch windows and now we suddenly understand how the IP fields are constructed. We can expand any field we want to see, such as the IP address of the source sending the IP frame. *Note that when looking at IP frames is may be worth requesting the debugger to display the contents in decimal rather than in hexadecimal by using right click and then selecting the display mode. The IP addresses are then better understandably and afterwards you can set the mode back to hexadecimal display since it is better for most other data fields*.



If you continue to step through the routine you will see that the IP frame is interpreted to see whether we are being addressed, it may cause our ARP cache to be updated and the check sum of the IP frame will also be verified. To return without stepping all the way you can use SHIFT F11 to return to the ETHERNET task code, where the protocol type is checked.

E. ICMP Handling

Note that each protocol type can be individually activated or deactivated in config.h. ICMP frames are only handled when the define USE_ICMP is set. If your project doesn't want to support ICMP then it can be simply removed…

Step into the ICMP routine (fnHandleICP()) by using F11. You will see that there is also a checksum in the ICMP field which is verified and the structure ptrICP_frame allows the ICMP fields to be comfortable viewed in the watch window.

Our frame should be of type ECHO_PING, which can also be individually deactivated in your project if you want to support ICMP but do not want the ping test to be replied to.

The received frame is sent back, after modifying a few fields, using the call fnSendIP(). I don't want to go into details about how the IP frame I constructed – you can see this in detail by stepping into the routine and observing what happens – but I should mention how the frame is sent out over the Ethernet since this is again a low level part which uses the LM3SXXXX registers again. Very briefly you should understand that the LM3SXXXX has an TX FIFO to which the frame has to be passed to. The data is first set up in memory to respect the ICMP, IP and Ethernet layers as we observed in the received message and, once completely ready, the transmission is activated by passing it to the FIFO in the routine fnStartEthTx () and setting the transmit bit (NEWTX) in MACTR.

Since the transmission activation takes place in the file LM3SXXXX.c in the function *fnStartEthTx()* so search for it and set a break point there. Once the program reaches this point it will stop and you can see which registers are set up and verify that all is correct.

Here you will also see that the simulator comes into play once the data is ready and the registers have been set up correctly. This code, which will not be discussed here, basically tries to behave as the EMAC transmitter does by interpreting the register set up and transmitting the data buffer contents to your local network.

OFF LINE simulation

Normally the simulator, or the target, will have responded to the ping request very quickly and the ping test would have been successful. We, being human beings, are rather slow and we probably took several minutes to work our way through the code before your ping reply was finally sent back. This was of course much too late as the ping test has already terminated, informing that the test failed.

This is not only a problem with the ping test but with many protocols since they use timers and expect replies within quite a short time, otherwise an error is assumed and links break down. Debugging of such protocols can become quite hard work due to this fact.

This is where the µTasker can save the day again since it supports operation in OFF LINE mode. This means simply that it can interpret Ethereal/Wireshark recordings as if they were read data from the network. Since we previously made a recording we can try this out right now!

A. Prepare a break point

I suggest that you set a break point in the ICMP routine itself since we have already seen how the message arrives there ad this will avoid false triggers due to broadcast frames in the mean time. Let the project run again by hitting F5.

B. Open the Ethereal/Wireshark recording

The recording from earlier was placed in the simulator directory and renamed as ethernet.eth and this is the file which will be loaded when you select Load Ethereal/Wireshark file to playback in the Ethereal/Wireshark menu of the µTasker simulator window.

Perform this action and the Ethereal/Wireshark file will be interpreted. The times that frames arrived will be respected – if there is a gap or 1s between two frames then these two frames will also arrive with a 1s gap. You will see that the recorded ping will be received via the receive Ethernet interrupt routine and be passed up through the software until the breakpoint in the ICMP routine is encountered. The difference is that the break point and stepping in code can not disturb the protocol since also the recording is stopped.

This can be a great advantage when debugging protocols!

Before I finish with this discussion there are a few points which should be noted, so here is a list of all relevant details which could be of use or interest.

1. When an Ethereal/Wireshark recording is played back, the internal NIC is closed so that there is no disturbance from the network.

2. The Ethereal/Wireshark recording can be repeated after it has terminated. There is no limit as to the number of times it can be repeated. Useful for incremental testing of a new piece of code…

3. After an Ethernet playback it is necessary to restart the simulator to use the simulator with the network again.

4. If you find a software error when stepping through the code, there is nothing to stop you making a correction without terminating the simulation session. Often VisualStudio can recompile the new code and continue with the debugging session, thus allowing the correction to be immediately used – this is a major advantage of the VisualStudio environment! Try it and you will learn to love it…

5. Ethereal/Wireshark recordings must not originate from the simulator, they can be recordings from targets or from foreign devices. This means that recordings of a good known sequence can also be used as input to developments (for example the recording of an email being collected by your PC).

It is necessary that the simulator is set up to have the same IP and MAC addresses of the device in the recording and it will then receive all recorded frames to that device. Using this recording it is then possible to develop new code, verifying that it reacts as the original device did at each step.

Using this method, it is even possible to develop quite complicated protocols or services using a known good case as a reference. It can be basically tested before being switched onto the network so that there is a good chance that it will even work first go!

Advanced Topics

Please see the following µTasker documents for further details about the following advanced topics related to the µTasker demo project.

- Global SW and Hardware timer ("***uTaskerTimers.doc***")
- I^2C driver ("***uTaskerIIC.doc***")

Latest documents are available on the µTasker home page http://www.uTasker.com/

Please also watch the µTasker forum for latest information and various specific tutorials: http://www.uTasker.com/forum/

Change history:

0.00/22.03.2008: Provisional first draft.