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 ST STR91XF. It can be fully simulated using VisualStudio 6.0 or higher and has target settings for the IAR STR91XF-EK evaluation board as well as the Raisonance REva board.

Although there are various ways to work with the STR91XF, the following method is described in this tutorial:

- Programming and configuration of the chip is performed using the CAPs program from ST together with the RLink JTAG programmer from Raisonance.
- Debugging is performed using IAR Embedded Workbench and the JLink JTAG debugger

It has to be said that the STR91XF is not the simplest chip in the world to work with. It is quite complex and some of the debug tools don't work very well with it – this is partly due to some bugs in the first silicon releases and partly due to the fact that it is quite new and the tools probably need some refining. It is advisable to familiarise yourself with the latest errata from ST and to carefully adhere to the guidelines in this tutorial: this should help avoid losing unnecessary time.

If you don't have the mentioned tools, other methods certainly are also possible, so use the tutorial as a base line and experiment with what ever tools you do own to produce equivalent results. I would recommend as a starting package the Raisonance REva kit which includes a JTAG programmer/debugger which is also supported by the ST CAPs configuration and programming tool. This is available quite cheaply and includes a GNU based debugger (RIDE). A RIDE project will be delivered with future µTasker STR91XF versions.

The Raisonance RIDE debugger is limited to 16k and so is not detailed here (a 32k limit is required to support debugging of the µTasker demo project) but a full version can be purchased separately if you find that you like it. Note also that the Raisonance forum is very supported so problems can usually be solved very quickly thanks to the help there (http://www.raisonance.com/Forum/punbb/viewforum.php?id=6).

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) 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 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 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 and see the discussion towards the end of the document).

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 is used for most of the following 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

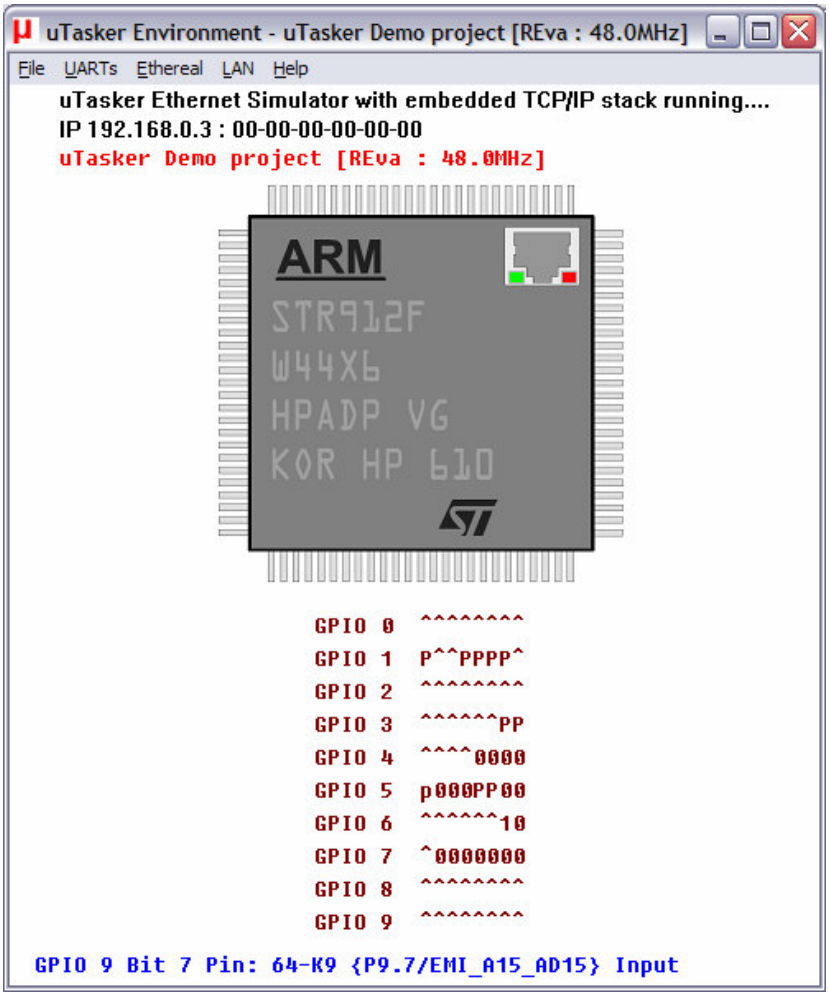
1. Simply copy the complete µTasker folder to your PC and go to "uTaskerV1.3\Applications\uTaskerV1.3". This is a ready to run project directory showing a useful application using network resources.
2. Move to the sub director "Simulator" and open the VisualStudio project workspace uTaskerV1-3.dsw. This 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 STR91XF target in the project's pre-compiler settings: look for the pre-processor define **_STR91XF**. If instead you find that the project is set up for another target, eg _HW_NE64 or _M5223X simply overwrite this with **_STR91XF**.*
   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")

```
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 (PB19]).



   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 shove above this is seen by the line in blue at the bottom of the simulation window.

   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 writes to the corresponding device registers 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 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 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 your Internet Explorer and check its settings in `Tools | Options`. In the extended register, search for the check box to configure passive FTP and ensure that it is deactivated. Once this is the case, exit the option dialogue and type in the following address ftp://192.168.0.3 (or the address of your device). Hit the enter key and you will establish an FTP connection to the simulated device.
   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….

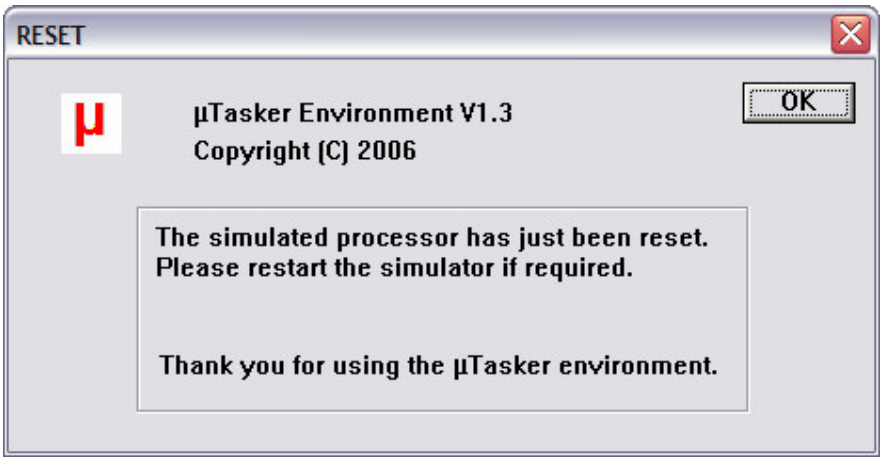
9. Now look in the project directory "uTaskerV1.3\Applications\uTaskerV1.3\WebPages\WebPagesSTR91XF". These are web pages which we will now load to the device.
   You can transfer all of the files to the FTP server by simply selecting all *.htm files and throwing them over to the FTP window (that is, using drag-and-drop).
   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 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_STR91XF.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_STR91XF.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 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.

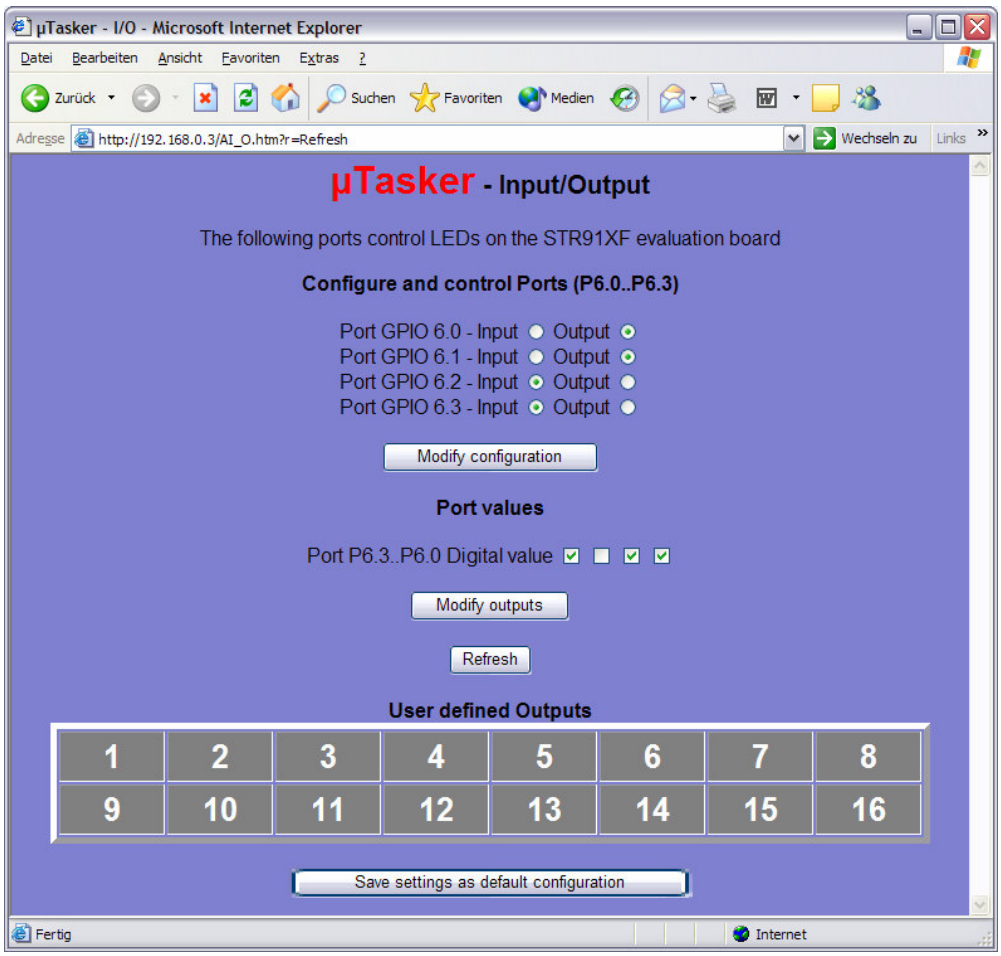
The page will not allow any parameters to be modified and also the check box "Settings validated" shows that the device is waiting for the new values to be validated. To perform this, click on "Modify / validate settings", after which the parameters will be displayed as validated. (*Don't forget to terminate 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 STR91XF and imagine that you have connected your lawn sprinkler to the Port GPIO 6.2. Presently it is at the state input '1' [signified by '^'], the default state when the software starts since there is a pull-up in the device. Now click on the link menu page to open an I/O window – it is shown in the following screen shot. Notice that the port GPIO 6.2is 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 GPIO 6.2 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 GPIO 6.2 to be an output, rather than an input and click on “Modify configuration”. Afterwards set the port state to ‘1’ by setting the third port value from the right and click on “Modify outputs”.
Watch the state of Port GPIO 6.2 – 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, all of the 4 Ports GPIO 6.0.. GPIO 6.3 outputs can be controlled individually.

GPIO 6.0 is used as a RUN LED in the demo project and so is configured as an output by default, as is GPIO 6.1.
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 GPIO 5.0..GPIO 5.1, GPIO 5.4..GPIO 5.6, GPIO 7.0..GPIO 7.6 and GPIO 4.0..GPIO 4.3. 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.

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…

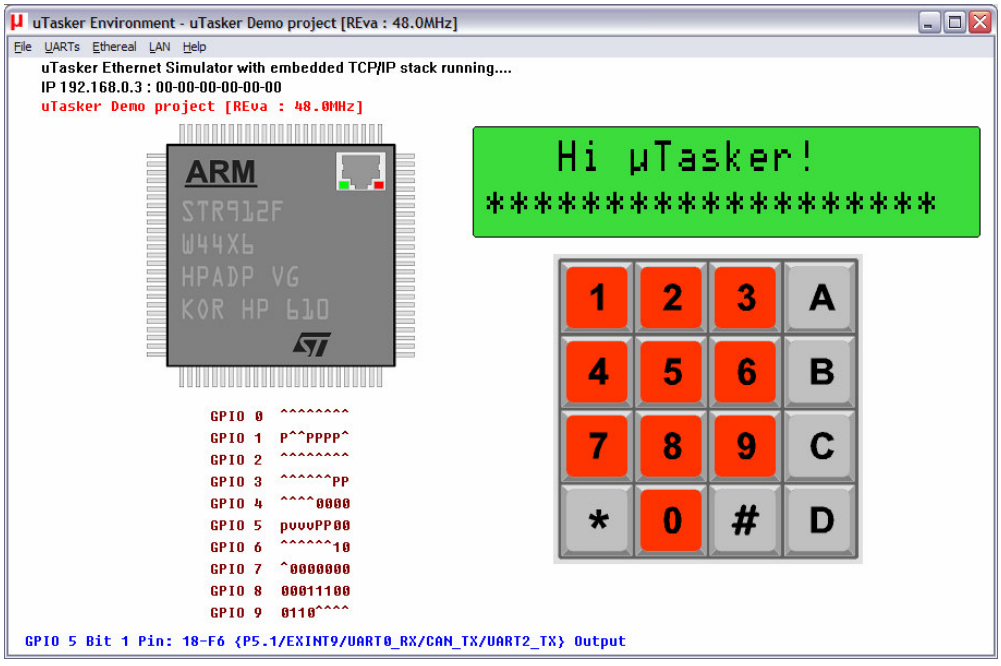
    Browse to the administration page, deactivate FTP and activate HTTP server authentication before selecting the action to “Modify and save server settings”. Finally click on “Perform desired action” and close the Internet Explorer.
    Open Internet Explorer again and enter the device’s address http://192.168.0.3 (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 detail 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.

Just to whet your appetite for what is to follow, here is a screen shot of the same project configured to support an LCD and matrix key pad – just as everything which we have done until now works in the simulate environment, also this works as in reality.



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 STR91XF and done this is 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…

We will use the Raisonance REva although the project also has a configuration for the IAR STR91XF-EK (it can be activated in `config.h` by setting the define STR912_SK rather than the define REva_DAUGHTER).

Notes on Ethernet PHY

The STR91XF includes an integrated EMAC but not a PHY. It is necessary to connect an external PHY in the hardware and the PHY used on the ST STR91XF-EK and the Raisonance REva is the STE100P from ST. The only actual difference between the two boards is that the interrupt line from the PHY and the PHY's address (read in on reset from certain I/O lines). Such differences are handled by the µTasker project configuration

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 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 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 STR91XF is delivered with an IAR Embedded Workbench project. 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, otherwise it should be possible to get the code up and running with any other compiler without too much hassle, but it will involve setting up an equivalent project.

By the way, if you can't wait until you have installed the compiler or got your own project up and running, there is a file delivered with the µTasker which was compiled with the IAR compiler and can be downloaded already…

You can find the project file and the target file(s) in the sub-directory called IAR_STR91XF:

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

IAR_STR91XF\Bank 0 full\Exe\uTaskerV1.3.hex This is the file which you can download to the target using CAPs, as explained later

Using the IAR Embedded Workbench you can open the µTasker project and compiler either for use with the debugger or as a release version (**`Project | Edit Configurations…`** - **Bank 0 full, Bare-minimum CAPs** or **Bare-minimum Upload**).

The "Bank 0 full" project generates code for operation from internal FLASH booting from bank 0 (the default configuration). The "Bare-minimum" versions require booting from a boot program in FLASH bank 1 which further allows complete code updates via Ethernet or Internet. To keep things simple for the moment we will stick with the "Bank 0 full" project build. To learn more about the boot loader capabilities please see the document "uTaskerBoot.doc " and "BM-BootLoader for STR91XF.doc" in the uTaskerBoot project.

When compiling the project, expect one non-serious compiler warning due to the fact that main has a forever loop which can't be exited..

Downloading the code to the target

I will explain how to download the FLASH based target code to the board using the CAPs utility from ST.

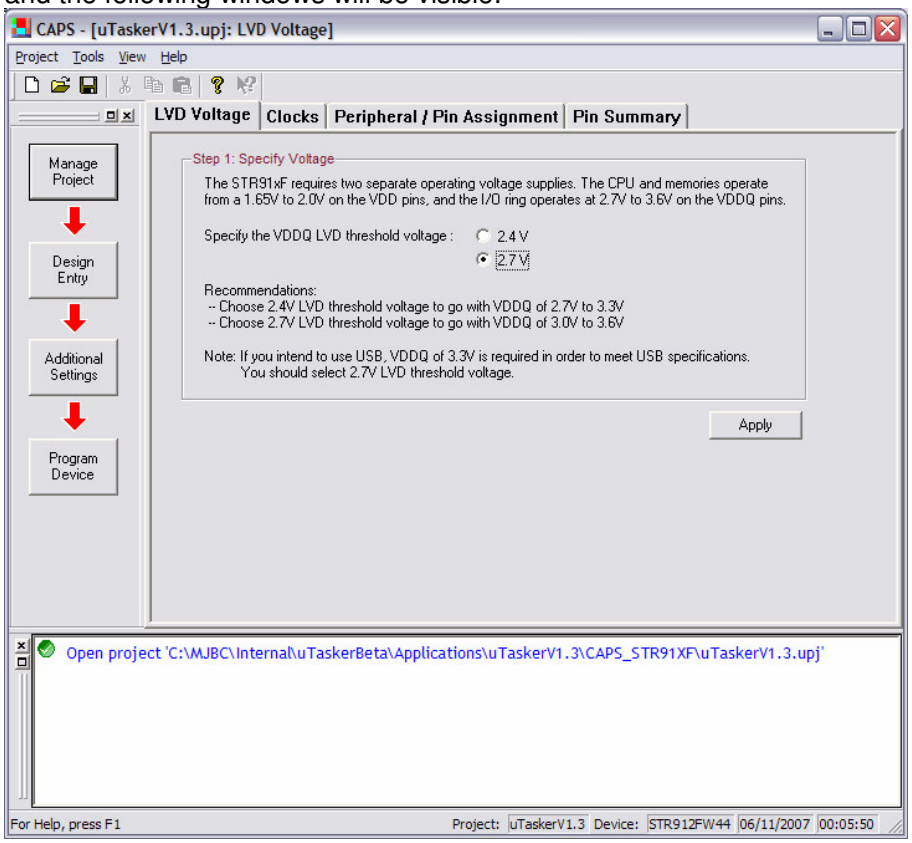
If you don't already have the CAPs program you can get it from the ST web site http://mcu.st.com/mcu/. Click on "downloads" under support. Select the category "STR9 (ARM9) 32-bit Microcontrollers" and click on "submit".
Chose now "Software –PC" and submit again.
Finally download and install the CAPs software according to its own instructions.

Once you have installed CAPS you can start. Here's a step by step guide so that it should work with no hassles the first time for you:

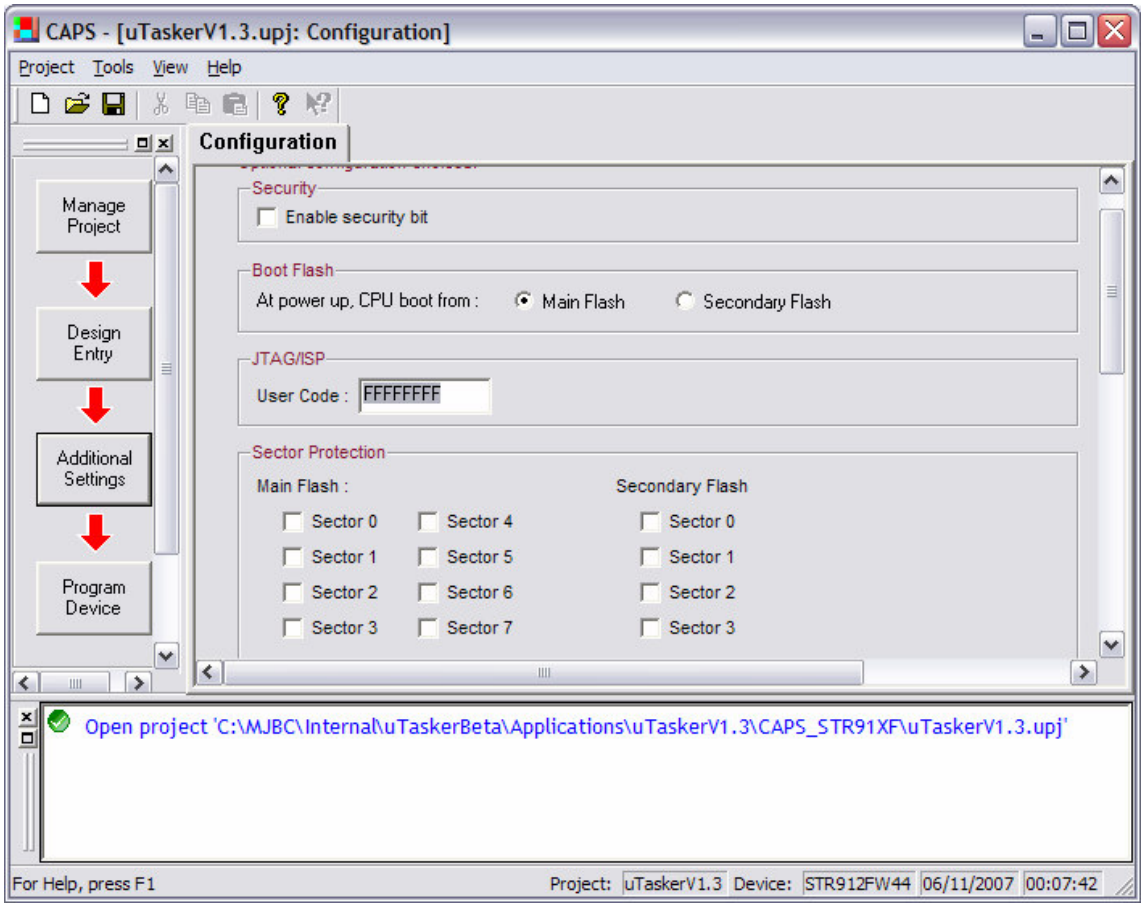
Step 1: Connect your PC via USB with the Raisonance RLink board (either integrated into the REva board, or already broken away from it as described in the REva documentation). If the JLink debugger (RLink) has already been broken away from the evaluation board, connect it via the 20 pin JTAG to the JTAG connector.

Step 2: Power on the evaluation board – its green power LED should light.

Step 3. Start CAPs program and select the command [Project | Open Project…]
Then browse to and open the pre-defined CAPs set up
\Applications\uTaskerV1.3\CAPS_STR91XF\uTaskerV1.3.upj:
You will see the description "uTasker standalone - boot from Bank 0" and a list of processor features. Simply click on the "Open" button at the bottom right of the screen and the following windows will be visible:

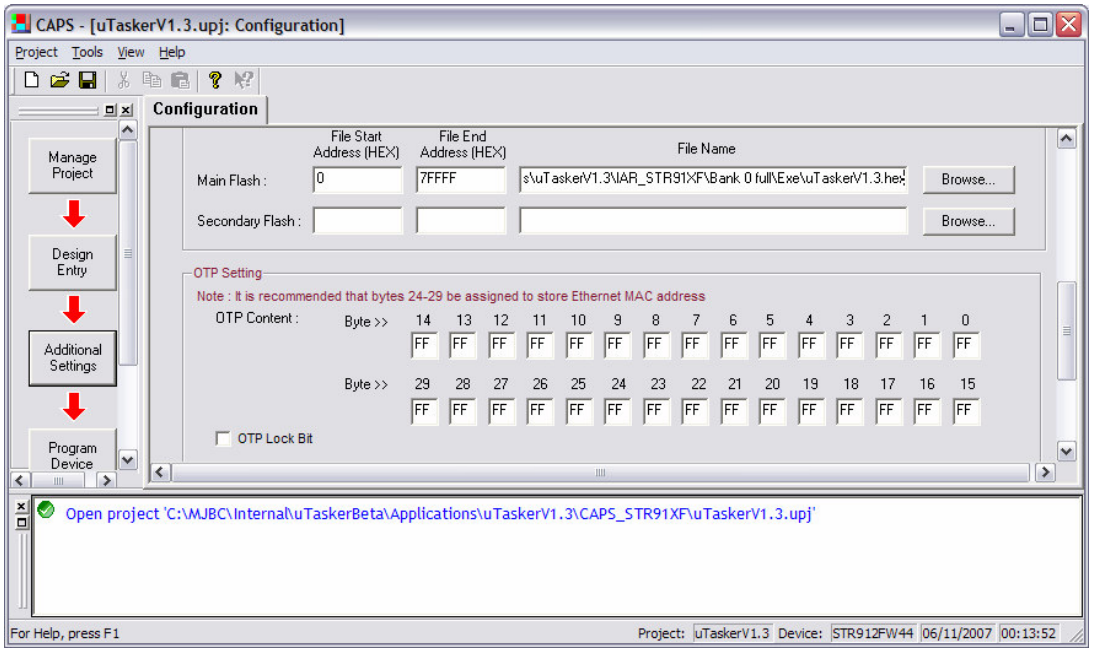


Step 4. Click on the button at the left “Additional Settings” to get to the configuration side:

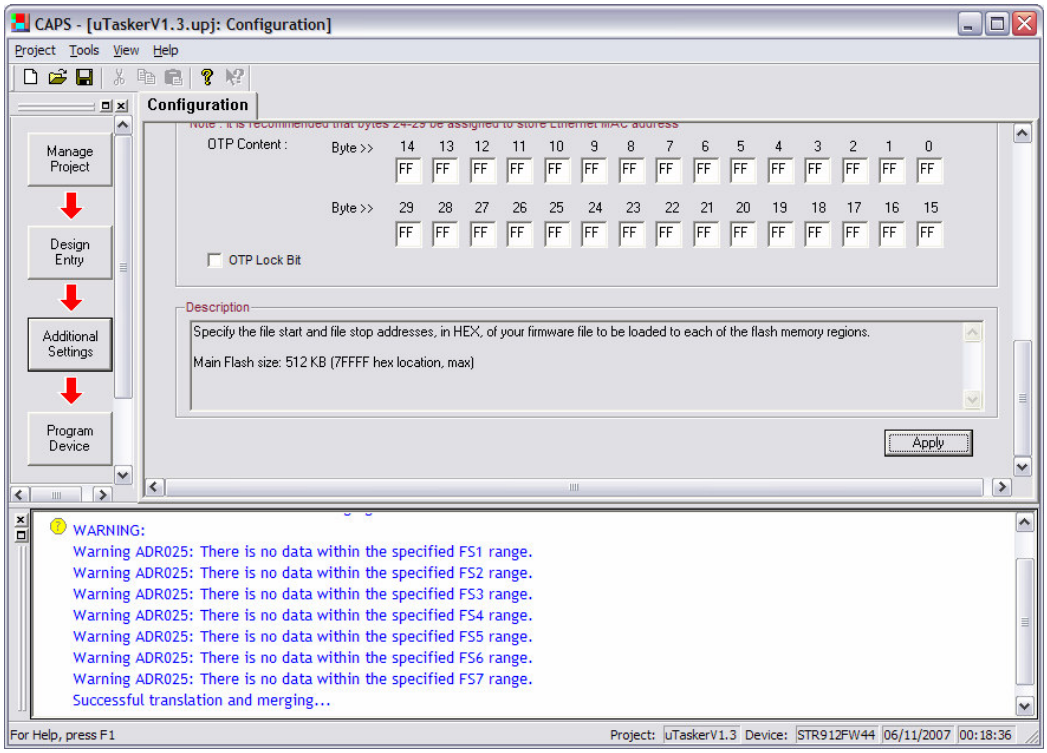


Here it is visible that the chip is to be configured to boot from the main FLASH. This is the default configuration of the chip so is already valid if the chip has not previously been programmed with a different configuration.

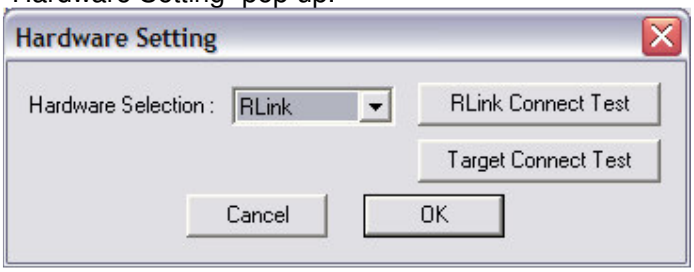
Step 5. The first time that you use the program you will have to set the path to suit your PC. The CAPs program doesn’t seem to be able to work with relative paths and so needs a full path. Scroll down the screen a little until the path setting is visible and using Browse set the hex file which resulted from the µTasker target project build (uTaskerV1.3.hex) as “firmware placement” for the main FLASH (since we are presently working without a boot loader in the secondary FLASH, the secondary FLASH field is left empty.



Step 6. Finally click on the "Apply" button which causes the CAPs too to create a file of its own which will later be downloaded to the target. This step is more interesting when using code in both the main and in the secondary FLASH since this step causes these two parts to be merged into one downloadable file.
You may not see the apply button without scrolling the page to the bottom as shown in the final screen shot. The warnings are normal – they are just informing that our code is not filling all FLASH sectors.

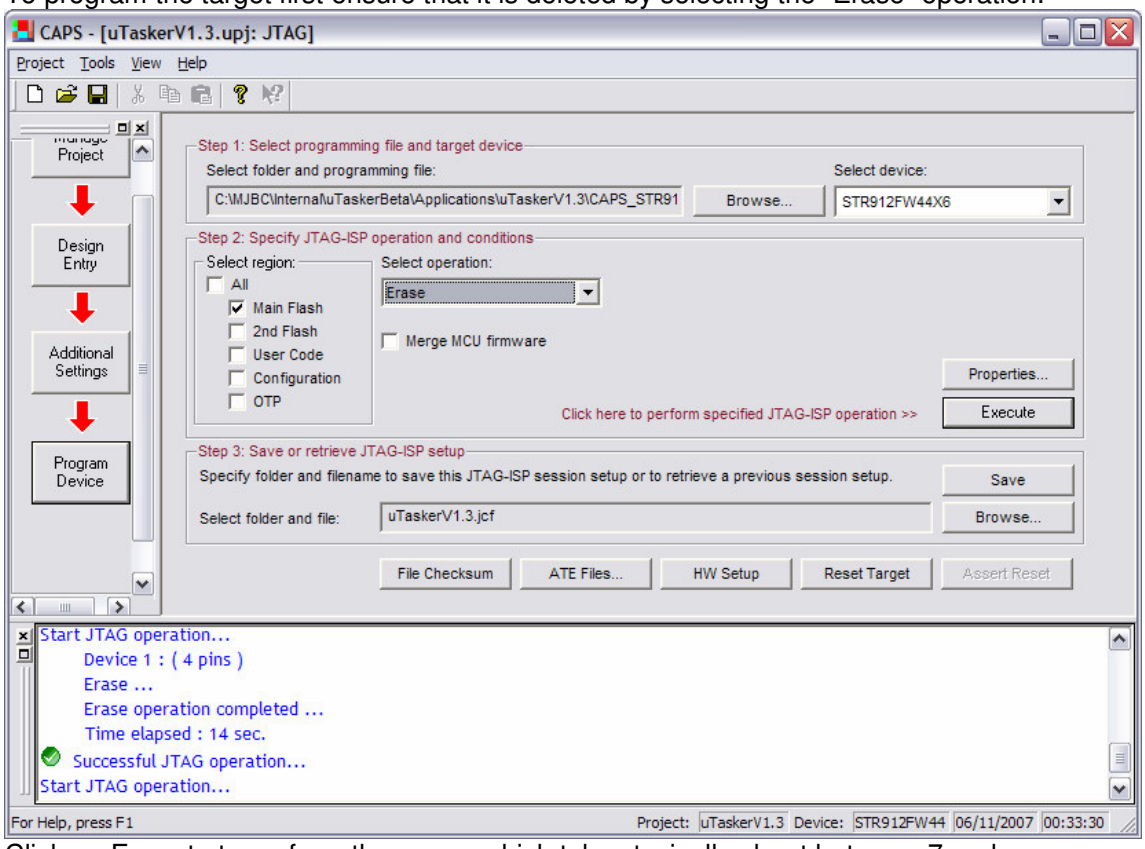


Step 7. now move to “Program device” and click on “Hardware Setup”, which will open the “Hardware Setting” pop-up.



Ensure that the RLink is selected (If you happen to have a FlashLINK this is also supported). You can use the two testes “RLink Connect Test” to ensure that your PC is correctly connected to the RLink and “Target Connection Test” to verify that the CPU is also powered and responding.

To program the target first ensure that it is deleted by selecting the “Erase” operation:



Click on Execute to perform the erase, which takes typically about between 7 and 15s.
Note here that only the main FLASH is erased. If your chip was previously configured to boot from FLASH bank 1 you can also check “Configuration” so that this is also set back to the default configuration.

Check that the programming file is correct. Here it will probably be necessary to set the path on first use to `Applications\uTaskerV1.3\CAPS_STR91XF\uTaskerV1.3.obj`. Finally Select the “Program/Verify” operation and click on “Execute”.

You should be done within a few minutes….

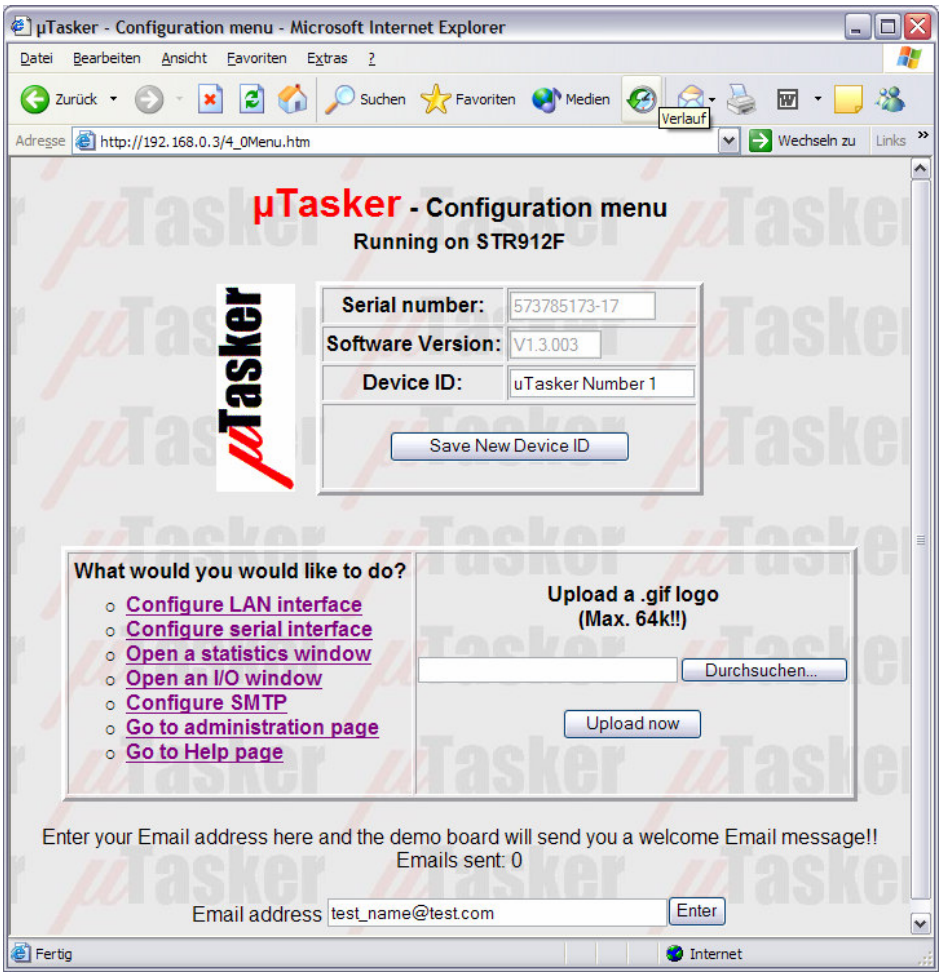
Are you surprised that the target behaves the same as the simulator? You shouldn't really be because that is exactly what the simulator is all about. It allows you to test your real code in real time and once it is working as you want it to, then you can transfer it to the real target. In fact you will 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 LEDs and port lines on the I/O port connector on the REva board to be controlled and the state of these to be displayed. Two LEDs are set as outputs be default. If GPIO 6.0 *is reconfigured to be an input the blinking LED will no longer light.*

A tour through the demo web server

Now that you have 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.

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 GIP file which was loaded together with the HTM web pages earlier on. If you look in the web page directory for the STR91XF you will see a further folder called `Alternative_Logo`. This contains another small GIF file (it has to be less that 64k to fit in the space reserved for it in the file system) which you can immediately use to see this working.

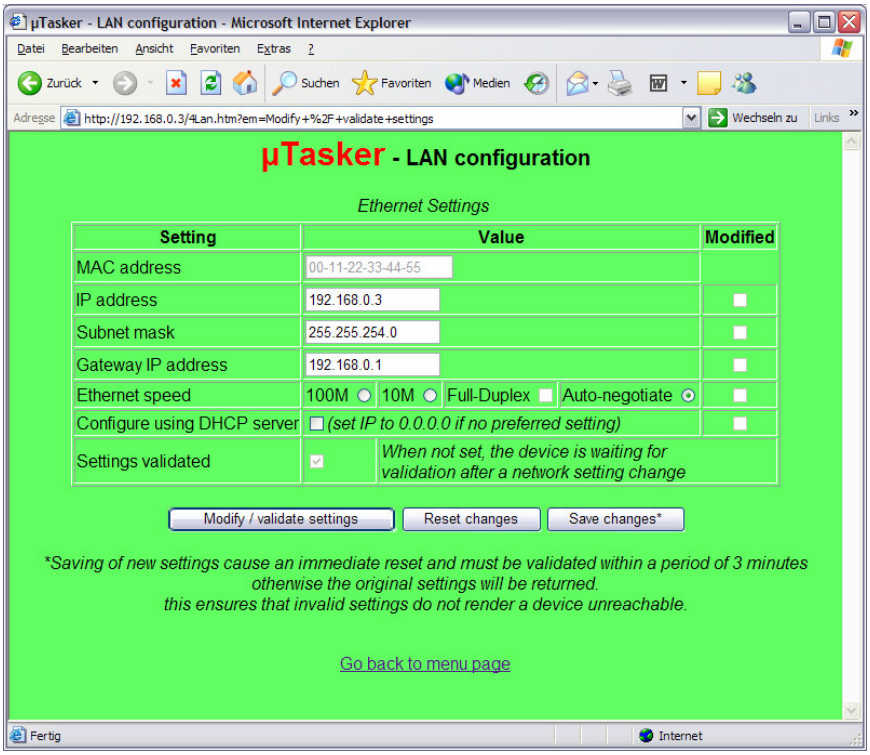
Click on the button to search for the file – Windows will open up an explorer menu and you can go to the "`Alternative_Logo`" directory and select the file called "`wow.gif`".

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 6k in size.

Email Address – The Email entry field is disabled by default in the demo project but it can be simply activated by enabling SMTP support. 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".

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 names "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.

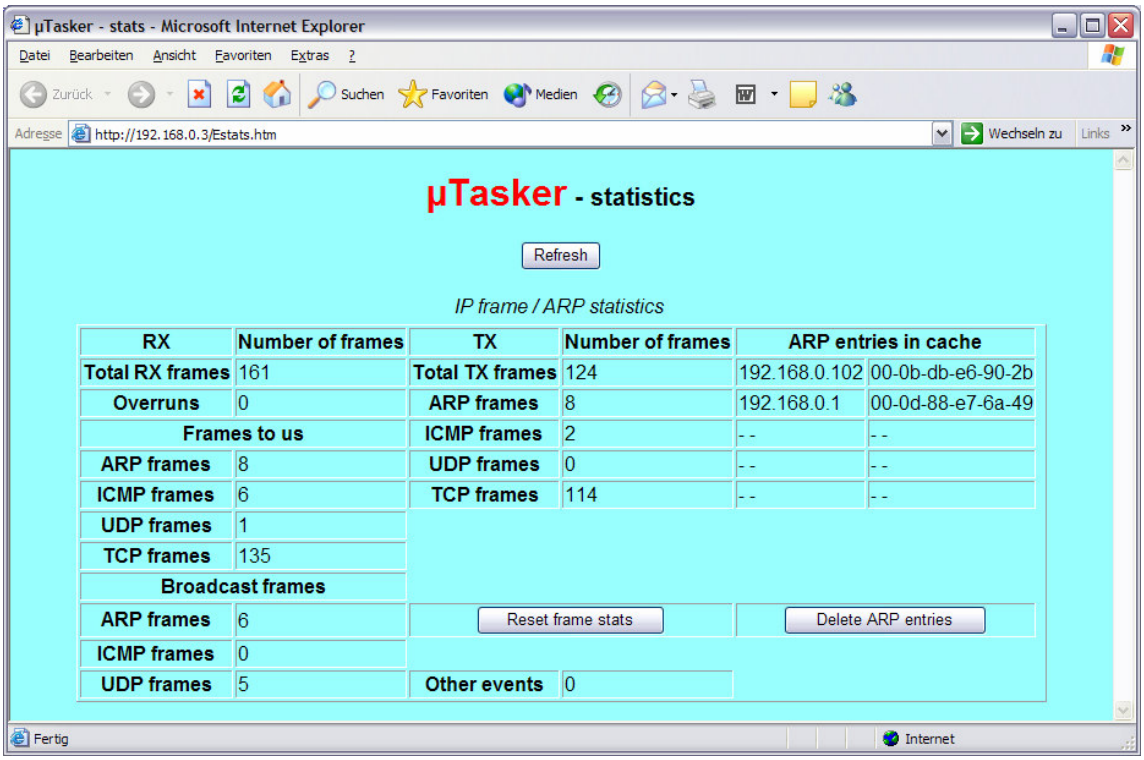
Serial configuration

This web side allows the UART0 in the M5223X 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.

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.

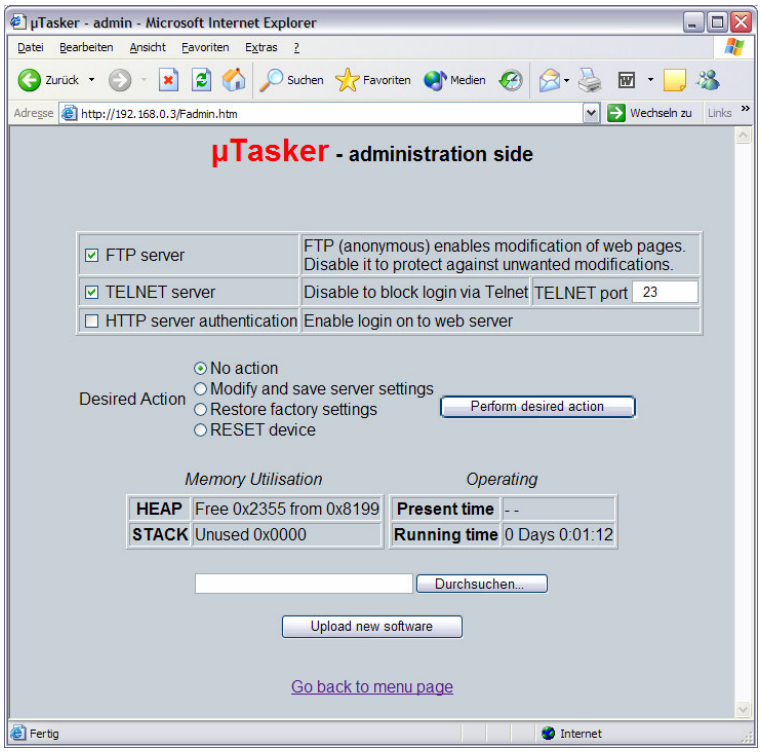
The HTTP server can be configured to require basic authentication, meaning that it requests the user name and password when a new connection is established. These values are contained in the default settings in **application.c.**

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. This is discussed later in this document.

Code sizes

(FLASH / RAM)	IAR compiler on MC9S12NE64 [HCS12] (highest level of compile for smallest size)	IAR compiler on SAM7X in Thumb mode (highest level of compile for smallest size)	IAR compiler on STR91XF in Thumb mode (highest level of compile for smallest size)
SMTP	**-**	**-**	**1'399 / 16**
DNS	**-**	**-**	**929 / 16**
Web server	**5'373 / 60**	**4'509/70**	**4'005 / 94**
FTP	**4'225 / 27**	**4'432/35**	**2'208 / 28**
TCP	**2'474 / 23**	**2'308/7**	**3'454 / 9**
DHCP	**2'660 / 10**	**2'922/32**	**1'995 / 32**
UDP	**480 / 2**	**482/4**	**808 / 4**
ICMP	**829 / 2**	**1'320/2**	**338 / 0**
IP + ARP	**1'991 / 75**	**2'222/77**	**2'598 / 6**
Ethernet	**1'657 / 17**	**1'928/1'561**	**862 / 81**
File system and parameters	**3'394 / 40**	**820/0**	**940 / 12**
General application code	**188 / 0**	**232/0**	**3'854 / 115**
Operating system and driver support	**2'027 / 93**	**3'441 / 92**	**3'421 / 45**
µTasker demo Total	**23'216 / 349 (V1.2 Project)**	**24'134 / 1890 (V1.2 Project)**	**30'884 / 3'092 (V1.3 Project)**

The RAM values are for static RAM and do not include the HEAP requirements, which is about 13 k bytes for this demo project, using the default configuration settings.

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

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

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

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, a free and powerful network Sniffer. You can download this from http://www.ethereal.com/, I use the version 0.10.8.0 but there are newer one's available.

If you haven't 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 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 recording and save it to the directory Applications\uTaskerV1.2\Simulator\Ethereal with the name ping.eth

7. Make a copy of the file ping.eth in the directory Applications\uTaskerV1.2\Simulator and rename it to Ethernet.eth

At this stage you can also look at the contents of the Ethereal 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 00:11:22:33:44:5
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 Help.

Here my PC has the local address 192.168.0.102 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 STR91XF.c (in hardware\STR91XF\) and search for the Ethernet reception interrupt routine called EMAC_Interrupt (). In fact this interrupt routine is common to all Ethernet interrupts so put a break point in the part which detects that the RX_ENTRY_EN flag has been set by positioning the cursor over the appropriate line 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 STR91XF, this will also give us the opportunity to look at some of its internal registers on the way.

A. Interrupt routine

```
__interrupt void EMAC_Interrupt(void)
{
    unsigned long ulInterrupts;

    while ((ulInterrupts = ENET_ISR) & ENET_IER) {  // read the interrupt status
                                        register, which clears all interrupts
        ENET_ISR = CLEAR_ALL_INTS;          // reset flags
        if (ulInterrupts & TX_CURR_DONE_EN) {  // current frame tramsmitted
            ...
        }

        if (ulInterrupts & RX_ENTRY_EN) {  // complete reception frame available
            iInterruptLevel++;              // ensure interrupts remain blocked when
                                               putting message to queue
            fnWrite(INTERNAL_ROUTE, (unsigned char*)EMAC_RX_int_message,
                                     HEADER_LENGTH); // Inform the Ethernet task
            iInterruptLevel--;                                          // release
        }
    }
}
```

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 STR91XF 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 it will also not work properly on your target.

We see that such routines are necessarily quite hardware specific. Here we see accesses to the internal register, ENET_ISR. Search for this registers in the STR91XF 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 STR91XF.

ENET_ISR is the 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 STR91XF.h, where it is defined as:

```
#define ENET_ISR  *(unsigned long *)(ENET_BLOCK + 0x008) // DMA Interrupt Status
                                                          Register
```

where ENET_BLOCK is also define locally twice:

```
  #define ENET_BLOCK ((unsigned char *)(&ucSTR91XF.ucENET)) // Ethernet Controller
  #define ENET_BLOCK  (0x6c000000 + BUFFERED_OFFSET)        // Ethernet Controller
```

On the target the register is located at long word address 0x6c000000 and when simulating it can be found in a structure called ucSTR91XF.

Double click on the word ucSTR91XF 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 peripheral divided into sub-blocks. ENET_ISR is in the sub-block ucENET 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 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.

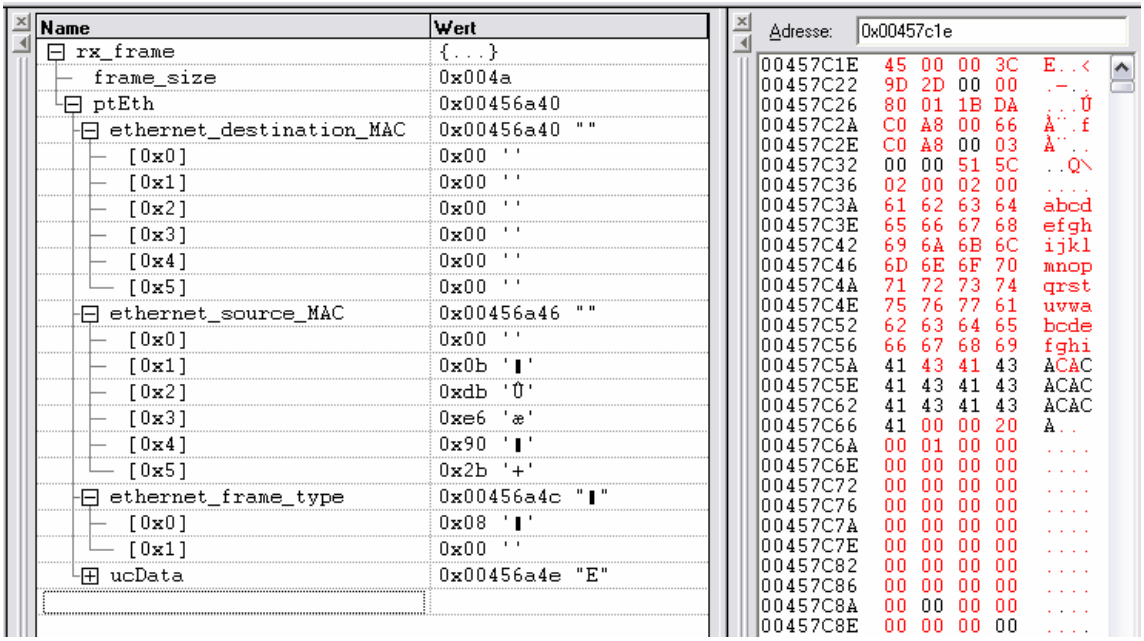
C. Receive Frame

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"

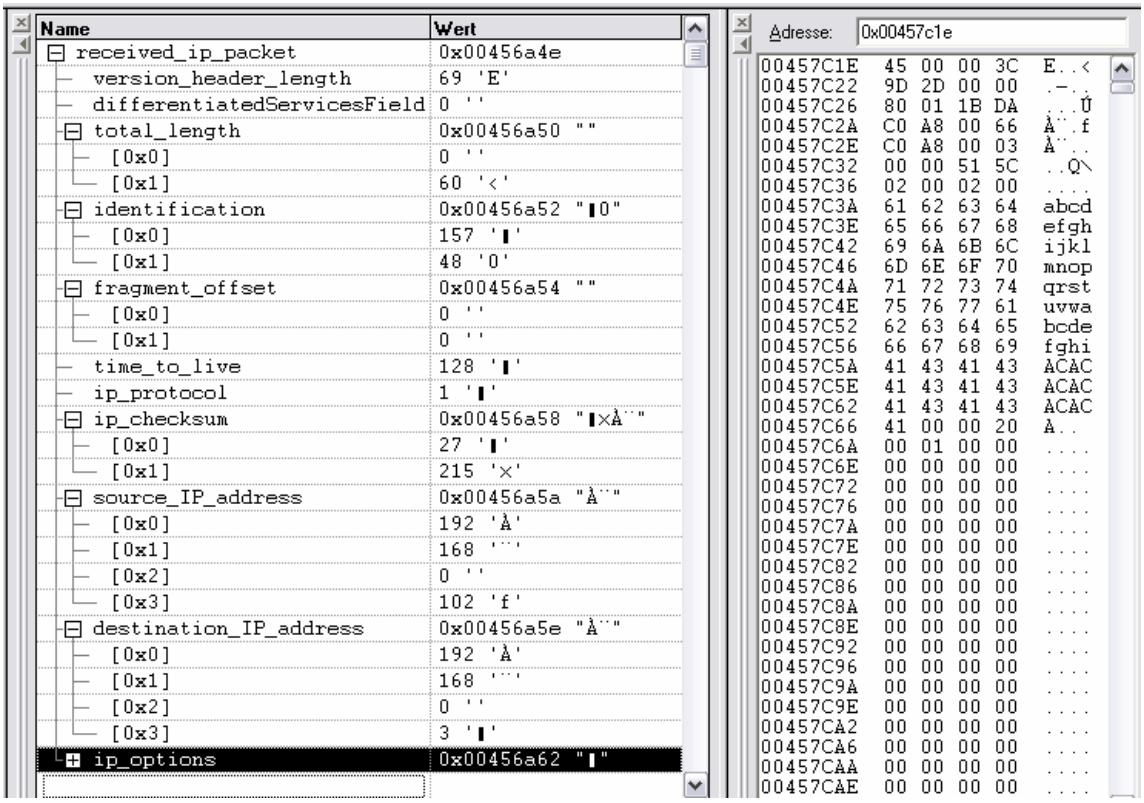
Adresse: 0x00457c1e

```
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 STR91XF registers again. Very briefly you should understand that the STR91XF has been set up with a number of transmission buffers (NUMBER_OF_TX_BUFFERS_IN_ETHERNET_DEVICE in app_hw_str91xf.h) into which the data is copied. The data is set up to respect the ICMP, IP and Ethernet layers as we observed in the received message and, once completely ready, the transmission is activated.

This transmission activation takes place in the file STR91XF.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 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 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 file to playback in the Ethereal menu of the µTasker simulator window.

Perform this action and the Ethereal 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 recording is played back, the internal NIC is closed so that there is no disturbance from the network.

2. The Ethereal 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 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!

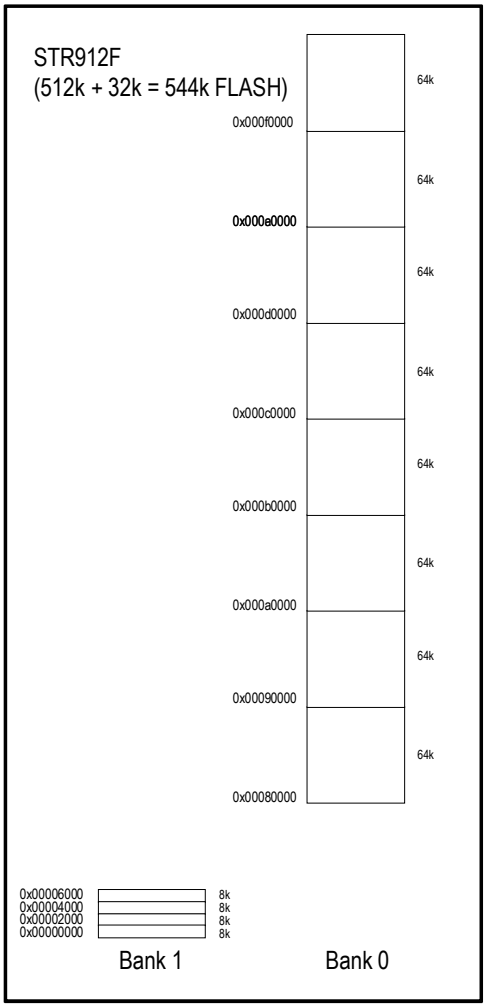
Some specific notes concerning the STR91XF

The STR91XF contains two FLASH banks which can be independently accessed – meaning that sections in one can be deleted or programmed which code is being executed in the other. The locations of the banks in memory can be programmed (or configured) and it is possible to boot from either.

Bank 0 (called the main bank) consists of a number of 64k sectors, depending on the device type. The STR912F has for example 8 x 64k sectors and thus 512k FLASH in this bank. There will be newer parts with more FLASH in this bank and there are smaller parts, for example the STR911X with a total of 256k in this bank.

Bank 1 (called the secondary bank) consists of 4 x 8k sectors, making a total of 32k.

By configuring bank 1 to start at the address 0x00000000 it can be used as a boot block and this is the option chosen in the uTasker project, which results in the following memory map using the STR912F.



A good strategy with this configuration is to have the uTasker Bare-Minimum Boot Loader in the first sector of Bank 1 so that secure software uploads are supported.

Then the last two of the following sectors can be used as a parameter system swap block pair for up to 8k of user settings with safe swap block support.

The remaining 8k block can be used for saving one-time data such as serial numbers of other factory data (or as overflow boot loader place if ever needed).

The main FLASH memory is used for program code, starting at 0x00080000 and occupying a number of sectors, before the rest is defined for use by the uTasker file system.

In the header file app_hw_str91xf.h the following defines can be set to configure the project depending on the boot bank configuration:

```
#define _BOOT_BANK1                     // booting from bank 1 (32k)
#define MAIN_FLASH_SIZE  (512*1024)
```

```
#define SECONDARY_FLASH_SIZE (32*1024)
#ifdef _BOOT_BANK1
    #define OFFSET_BANK0     0x00080000
    #define OFFSET_BANK1     0
#else
    #define OFFSET_BANK0     0x0
    #define OFFSET_BANK1     0x000080000
#endif
```

This suits the STR912F booting from bank 1. When booting from bank 0, the define _BOOT_BANK1 can be simple commented out.

It is recommended to use #define SAVE_COMPLETE_FLASH when working with the STR91XF so that the complete simulated FLASH contents are saved to the hard disc and restored on next start. This ensures that file and parameter information is saved, whether in one or split over both FLASH banks.

In order to learn more about using the µTasker project together with its boot loader, which uses this configuration, please see the documents “uTaskerBoot.doc” and “BM-BootLoader for STR91XF.doc”.

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***”)

Change history:

0.01/10.6.2007: Provisional first draft.