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 Freescale MC9S12NE64. It can be fully simulated using VisualStudio 6.0 or higher and can be run on the DEMO9S12NE64 hardware, in which case the GNU compiler can be used and a serial debugger is supplied in the µTasker NE64 package for downloading and simple debugging. Therefore there should really 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. Lets start with something that will already impress you and your friends – no simple and basically useless demo but something seriously professional and for real life projects very handly.

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 a 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.

Fasten your seat belts since we are about to roll

1. Simply copy the complete µTasker folder to your PC and go to "uTaskerV1.2\Applications\uTaskerV1.2". This is a ready to run project directory showing a typical application using network resources.
2. Move to the sub director "Simulator" and open the VisualStudio project workspace uTaskerV1-2.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.
3. *Ensure that the compiler is set up for the NE64 target in the project's pre-compiler settings: look for the pre-processor define **_HW_NE64**. If instead you find that the project is set up for another target, eg _HW_SAM7X or _M5223X simply overwrite this with **_HW_NE64**.*
   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.
   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.
   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.
   One last point. When the simulated device sends or received frame over the LAN you will also see that a port output changes for a short time – this is a port set as link status output.

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 look in the project directory "uTaskerV1.2\Applications\uTaskerV1.2\WebPages\WebPagesNE64". These are web pages which we will now load to the device.
   Start by loading the file called **T404.htm**. This can be performed simply by using drag-and-drop on to the ftp server window.
   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 only loaded this file and it is returned when any other requested one is not found….

9. Now we can transfer all of the files to the FTP server. Simply select all remaining *.htm files and throw them over to the FTP window.
   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_NE64.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_NE64.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. Let's 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".
    Now the real device would actually be reset at this point but our simulator doesn't allow resetting so we will have to close the simulation by using the normal program exit and then restart, so do exactly that.
    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 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. So we will now validate our new setting (new MAC) so that that it will always be used in the future.
    Open a new Internet Explorer browser and type in the device's address. http://192.168.0.3 (or your address) and 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. On the LAN configuration page click on "modify / validate settings" and the settings will be committed (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 just start the simulator again and repeat the process.

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 NE64 and imagine that you have connected your lawn sprinkler to the output Port G2. Presently it is at the input state '0' [signified by 'v'], the default state when the software starts. Now click on the link to open an I/O window. Set G2 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 G – you will see that the output is set and now your lawn will 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 8 Port G outputs can be controlled individually. If configured as inputs the state of the port is displayed.
    G0 is used as a RUN LED and so is configured as an output by default, as is G1.
    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.
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…
    So 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.

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 NE64 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 DEMO9S12NE64 since it is a readily available board in a cute transparent housing which most people get hold of first before possibly moving on to their own designed target. The first thing to say is that if you also decided to go for the DEMO9S12NE64 then it was a good choice because it allows probably everything that you want to do with an evaluation or prototyping board. If you have another board, including your own design then you will almost certainly find that all of the code will work without any changes there too.

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 is in fact delivered with projects for a number of well known compiler types. Look in the directory uTaskerV1.2\Applications\uTaskerV1.2 and you will see some subdirectories and maybe you find a project for your favourite compiler including its development project. Here we will be using the GNU compiler with a simple batch file since this compiler is available as open source, meaning that it can be used by anyone for any purpose free of charge. Later there will be more details about compilers but I can say the following about the GNU compiler – it works so don't have any worries about using it. It is not the best, so you may do well to invest in a commercial compiler for serious projects where maximum efficiency is important (dense and fast code). For our first project this is not an issue so let's get on and compile the project with the GNU compiler.

Of course if you want to compile the project with the GNU compiler you will need the GNU compiler. If you don't have it yet you can download it from http://www.gnu-m68hc11.org/. I use the version 3.3.4 (2004-08-29) and there is probably a newer one available, which is probably fully compatible. Fortunately the GNU compilers are very user friendly nowadays and are usually delivered with an installation program which will put all directories in the correct place, including user documentation. I will not explain this here in any more detail since it is adequately described in the GNU compiler installation instructions.

By the way, if you can't wait until you have installed the GNU compiler or any other one of your choice, there is a file delivered with the µTasker which was compiled with the GNU compiler and can be downloaded already…

You can find the batch file and the target file(s) in the sub-directory called GNU_NE64:

The following files can be found in this sub-directory:

memory.x	This is a file describing how the target memory is organised and is used by the GNU linker to put the variables and code in the right places for the NE64.
uTaskerV1-2.bat	This is the batch file which will instruct the GNU compiler to compile all source files, link them and create a Motorola-S record for download to the target.
uTaskerV1.2.elf	This is an intermediate result of the process, from which the S-record is derived from.
uTaskerV1.2.psa	This is a pseudo-assembler file of the complete code which we will use later form simple target debugging.
uTaskerV1.2.s19	This is the S-record file which we can download to the target.

You will however have to edit the bat file to suit your environment, which means changing the following line:

```
C:\Programme\usr\bin\m6811-elf-gcc -
IC:\MJBC\Internal\uTaskerBeta\Applications\uTaskerV1.2 -D _HW_NE64 -
g -Os -m68hcs12 -mshort ……..
```

Change the path to where the compiler can be found on your PC and the path to where the project can be found and then you can get on to compiling.

Simply double click on the batch file and the project will be compiled. It is however better to start the batch file from a program which allows the processing to be seen in a window in case of some warning or error messages. Many editors allow a bat file to be started in this manor and I have also added a VisualStudio configuration which includes this step automatically. Simply change the active configuration from "Win32 Debug" to "Win32 Debug and GNU target" in which case the project will be built as normally and then also for the target using the GNU compiler.

Downloading the code to the target

The DEMO9S12NE64 is delivered with a pre-programmed monitor program allowing downloading code via the serial interface and also quite useful debugging support. This monitor code can also be loaded to other boards, but this will require a BDM module. Later on I will explain how you can convert your DEMO9S12NE64 into a BDM but for the present I am assuming that you want to load to your DEMO9S12NE64.

When the target code was created, using the batch file as detailed in the previous section, it copied this downloadable to the directory "uTaskerV1.2\Tools\NE64". Go to this directory and you will find that there is indeed a tool delivered with the µTasker allowing programming via the serial interface. This tool is very reliable but requires this to take place via COM1 so your PC must have COM1 free and this should be connected with the serial interface of the DEMO9S12NE64.

Set the DEMO9S12NE64 switch SW3 to '0' – meaning LOAD and power up the DEMO9S12NE64, or reset it using the reset button.

Start the program NE64_Deb.exe (in the tools directory) by double clicking on it. It should immediately establish a connection and display the resisters of the NE64.

Command `Debug | Reset` and all registers which we displayed in green should be displayed as white (this means that they have been updated and no changes detected).

Command `FLASH | Delete parameter block`, which deletes the contents of FLASH which we will be using as file system, to be sure that there is nothing there which may cause us problems later.

Command `FLASH | Delete Program and program new`. This deletes the program part of FLASH and downloads the S-record file which we have previously created and copied there when we compiled. This file is locally called ETHERNET.s19 (it was renamed when it was copied) and so we do not have to inform the download tool which file it should load.

The download takes about 7 seconds to complete, after which you can start the program using first `Debug | Reset`, followed by `Debug | Run`. Alternatively you can set the DEMO9S12NE64 switch SW3 back to '1' – meaning RUN and reset the DEMO9S12NE64. *Debugging support is however only available when it is started using the debugger*.

Now repeat the first part of the tutorial using the real target instead of the simulator. *If you should have problems when using the target please read the section below entitled "**Some specific notes concerning the MC9S12NE64**" to see whether you can find the explanation for the problem there.*

You should be done within a few minutes….

Are you surprised that it 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 DEMO9S12NE64 to be controlled and the state of these to be displayed. The state of the LOAD-RUN switch SW3 can also be displayed.
The LEDs are set as outputs be default. If PG0 is reconfigured to be an input the blinking LED will no longer light.
Be careful when setting PG4 as an output when SW3 is in the '0' position since it will be short circuited to ground!!

Some specific notes concerning the MC9S12NE64

The MC9S12NE64 has two bugs which require some attention. You can read about these in the errata sheet MC9S12NE64, Mask 1L19S, Rev. April 12, 2005 and both concern the Ethernet interface.

The first is a problem with auto-negotiation when the link partner's LTP is greater than 100ns. I tend to have no problem when using the NE64 in auto-negotiation mode with hubs and switches but do have problems when connecting directly to my laptop with a cross-over cable. Unfortunately auto-negotiation can not be relied upon in every situation. The demo project is set up as default to use auto-negotiation and so if you do have problems with it and the link LED is not lighting on the LAN connector of the DEMO9S12NE64 then it may be necessary to configure the LAN interface to suit the speed of the network which it is connected to. To do this, edit the following line of the `network_default` structure in **`application.c`**

```
    (AUTO_NEGOTIATE | FULL_DUPLEX | RX_FLOW_CONTROL),
```

to `(LAN_10M | FULL_DUPLEX | RX_FLOW_CONTROL),`

or `(LAN_100M | FULL_DUPLEX | RX_FLOW_CONTROL),`

and then recompile.

If the problem with auto-negotiation is really critical then it is also possible to switch between 10M and 100M operation until a link has been found. This is not very efficient and is also not supported by the demo project, but could represent a last ditch cure.

The second problem is to do with the activity LED. If the NE64's EPHY is set to the mode in which it controls the activity LEDs, the activity LED itself (ACTLED) blinks so fast on receive data that it is not visible. Therefore it is generally better to control the LEDs from software and the µTasker is supplied with such support, which is descried here.

First of all, if you would like to try with the EPHY controlling the LEDs itself then simply comment out the following define in the **`config.h`** file

```
// #define LAN_REPORT_ACTIVITY
```

which will remove the software control of the LEDs and force the Ethernet controller to configure the automatic LED control by the EPHY.

When the define is set, a task called `fnNetworkIndicator()` is present [see **`NetworkIndicator.c`**] which receives events from the Ethernet controller when link

changes occur. It then sets the LEDs accordingly and also controls the activity LED with a timer, pulsing it 50ms long for transmitted frames and 100ms long for received frames, making activity well visible.

When the link is down then no LEDs light on the LAN socket. The link LED lights orange for a 10M link and green for a 100M link.

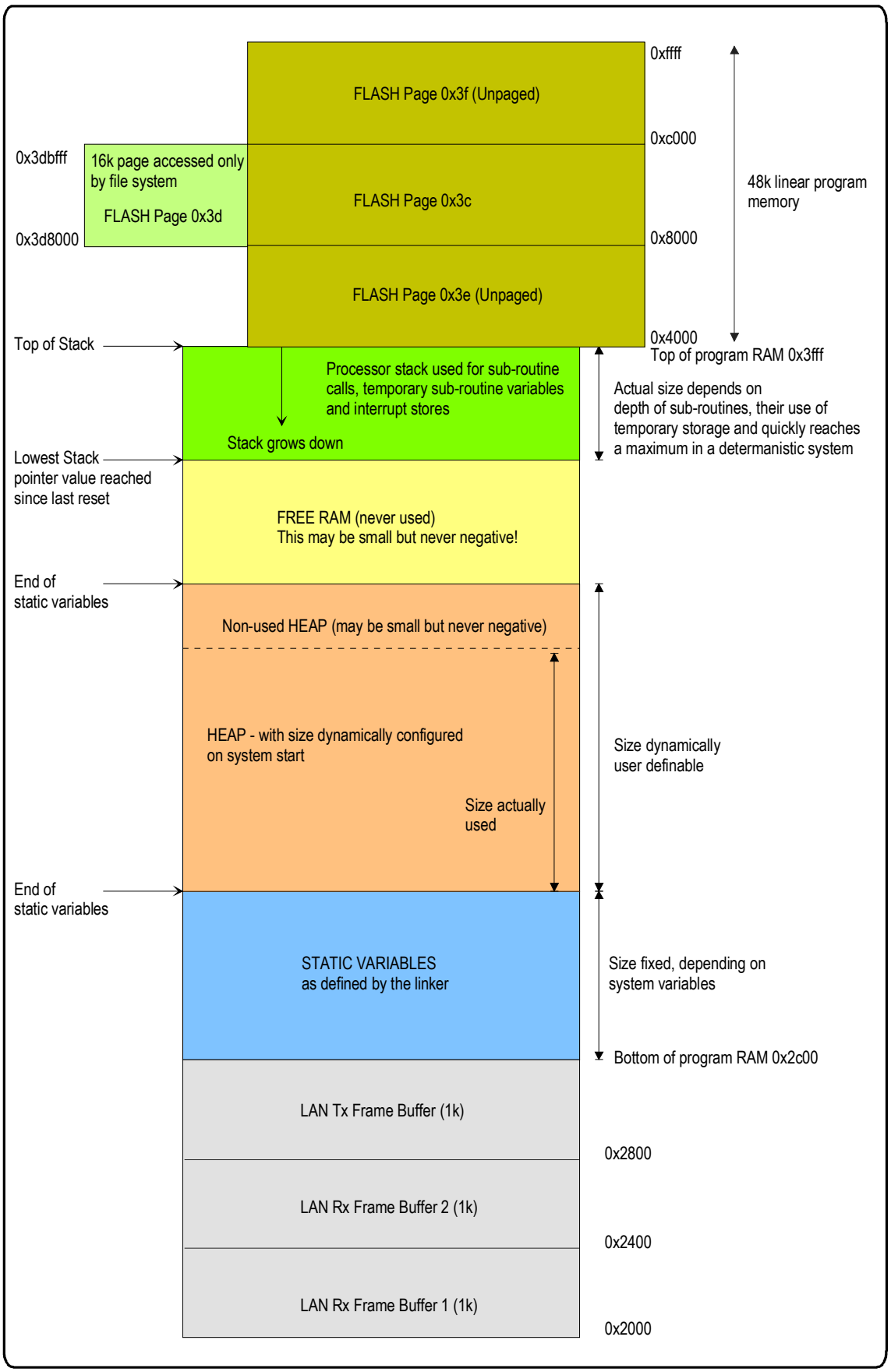
The yellow activity LED blinks when there is network activity either from the NE64 or received by the NE64 for its MAC address or the broadcast address.

Memory utilisation

The demo project uses a linear memory model (without paging). The reason is to achieve highest memory efficiency on the small device by avoiding long pointers and trampoline functions. The file system uses one 16k page, which it switches in and out of memory when accesses are necessary.

The SRAM must also be shared between the application and the Ethernet frame buffers. There are always three frame buffers; one for transmission and two for reception and there sizes can be configured when initialising the Ethernet controller. Since the frame buffers have to start at the beginning of SRAM and their size is configurable, the memory map of the application must respect this and should therefore start using SRA; just after the end of the transmission frame buffer (the ordering is fixed at 2 x Rx and then 1 x Tx).

The complete system memory map can therefore be shown for the demo project using default settings (1k Ethernet frame buffers).


0xffff
FLASH Page 0x3f (Unpaged)
0xc000
0x3dbfff
16k page accessed only by file system
FLASH Page 0x3d
FLASH Page 0x3c
0x3d8000
0x8000
48k linear program memory
FLASH Page 0x3e (Unpaged)
0x4000
Top of program RAM 0x3fff
Top of Stack
Processor stack used for sub-routine calls, temporary sub-routine variables and interrupt stores
Stack grows down
Actual size depends on depth of sub-routines, their use of temporary storage and quickly reaches a maximum in a determanistic system
Lowest Stack pointer value reached since last reset
FREE RAM (never used)
This may be small but never negative!
End of static variables
Non-used HEAP (may be small but never negative)
HEAP - with size dynamically configured on system start
Size actually used
Size dynamically user definable
End of static variables
STATIC VARIABLES
as defined by the linker
Size fixed, depending on system variables
Bottom of program RAM 0x2c00
LAN Tx Frame Buffer (1k)
0x2800
LAN Rx Frame Buffer 2 (1k)
0x2400
LAN Rx Frame Buffer 1 (1k)
0x2000

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.

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 have already been discussed under the specific notes to the NE64.

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 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 customers 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 loosing contact with the device, even when a mistake is made.

Serial configuration

This web side allows the UART1 in the NE64 to be set. I won't discuss this in any detail here because the serial interface is the topic of a later tutorial. It is however worth just mentioning that the UART in the NE64 is a little simple and can not actually be set to all values on the page – we'll ignore this detail for the time being…

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.

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 with 8 or 16 bit architecture. 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.

The methods used for the memory monitoring are detailed in the µTasker main documentation.

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 remover but possibly not enable its occurrence to be readily detected.

IAR Compiler

The µTasker demo is available as IAR Embedded Workbench project in the directory uTaskerV1.2\Applications\uTaskerV1.2\IAR_NE64.

Build the project as normal – there is one warning due to a non-reachable return in `main()` which can be ignored. Then double click on the file toDebug.bat in the sub-directory IAR_NE64\Release\Exe to copy and rename the downloadable S-record to the debugger directory. Alternatively if you have a debugging tool for the IAR Embedded Workbench this can of course be used.

Metroworks CodeWarrior Compiler

The µTasker demo is available as CodeWarrior project in the directory uTaskerV1.2\Applications\uTaskerV1.2\CodeWarrior_NE64.

Build the project as normal – there should be no warnings. The project has already been configured to copy and rename the downloadable S-record to the debugger directory. It is recommended that the downloader tool delivered with the uTasker is used for downloading if the serial monitor is used as the CodeWarrior serial download tool doesn't support downloads without deleting also the uTasker file system – you will otherwise have to reload web pages and change parameters each time! It is recommended that CodeWarrior for the HCS12 versions from V4.5 are used since earlier versions have problems debugging the linearly organised memory.

The uTasker demo project works with the evaluation version of the CodeWarrior since its size is adequately small.

Code size comparisons

(FLASH / RAM)	GNU Compiler 3.3.4 (optimised for smallest size)	IAR compiler (highest level of compile for smallest size)	CodeWarrior compiler (default optimisation)*
Web server	**6'592 / 60**	**5'373 / 60**	**5'734 / 60**
FTP	**4'284 / 20**	**4'225 / 27**	**4'755 / 27**
TCP	**5'862 / 5**	**2'474 / 23**	**2'652 / 3**
DHCP	**3'480 / 31**	**2'660 / 10**	**3'328 / 30**
UDP	**1'354 / 2**	**480 / 2**	**538 / 2**
ICMP	**642 / 0**	**829 / 2**	**908 / 2**
IP + ARP	**4'860 / 4**	**1'991 / 75**	**2'256 / 75**
Ethernet	**4'138 / 90**	**1'657 / 17**	**1'858 / 17**
File system and parameters	**2'882 / 4**	**3'394 / 40**	**1'294 / 40**
General application code	**1'796 / 2**	**188 / 0**	**220 / 0**
Operating system and driver support	**6'968 / 79**	**2'027 / 93**	**2'394 / 30**
µTasker demo Total	**42'888 / 297**	**23'216 / 349**	**25'379 / 286**

**It was found that the default optimisation produced more or less best code density and so it was left like that. Setting to highest density actually generally increased code size slightly.*

The RAM values are for static RAM and do not include the HEAP requirements, which is about 2'200 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 ne64.c (in hardware\ne64\) and search for the Ethernet reception interrupt routine called emac_rx_b_a_c_isr(). In fact the NE64 has two receive interrupt routines on receive buffer full, this on for the A-buffer and emac_rx_b_b_c_isr() for the B-buffer, which are used alternating. So put a break point in one of these by positioning the cursor over the first line of the routine and pressing F9 (you can put breakpoints in both but sooner or later the first one will be used).

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 NE64, this will also give us the opportunity to look at some of its internal registers on the way.

A. Interrupt routine

```
__interrupt void emac_rx_b_a_c_isr(void)
{
  IMASK &= ~RXACIE;                              // mask further use until application
                                                 has read buffer
  eth_rx_control->ETH_queue.chars = RXAEFP;      // put the length of the received
                                                 frame in the buffer
  RXAEFP = 0;                                    // clear reception length
  fnWrite( INTERNAL_ROUTE, (unsigned char*)EMAC_RXA_int_message, HEADER_LENGTH);
                                                 // Inform the Ethernet task
}
```

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 NE64 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 two internal registers, IMASK and RXAEFP. Search for these registers in the NE64 data sheet if you want to read exactly how they work; here is a very quick overview where we will also look at the registers in the simulated NE64.

IMASK is the interrupt mask 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 ne64.h, where it is defined as:

```
#define IMASK  *(unsigned short*)(EMAC_BASE_ADD + 0xc)
```

where EMAC_BASE_ADD is also define locally twice:

```
#define EMAC_BASE_ADD      ((unsigned char *)(&ucNE64.ucSimEMAC)) and
#define EMAC_BASE_ADD       0x140
```

On the target the register is there located at byte address 0x14c and when simulating it can be found in a structure called ucNE64.

Double click on the word ucNE64 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. IMASK is in the sub-block usSimEMAC 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) so that the first instruction is executed and you will see that the content of the IMASK register changes as the bit RXACIE is reset.

The length of the frame is contained in the register RXAEFP, which is also visible in the register list belonging to the EMAC. Notice that the interrupt routine code copies this value and then resets it to zero, so that it can be used again by the EMAC later before sending an internal message defined by

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

This internal message is sent to TASK_ETHERNET to inform that a frame has been received in the A-buffer 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 [do you recognise the length from earlier?]. 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.

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 better at the moment.

D. Frame protocol

Step slowly using F10 and you will see that the frame is check 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 NE64 registers again. Very briefly you should understand that the NE64 has one transmission buffer 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 ne64.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 real 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 don't have to 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!

Change history:

0.08/17.8.2006: Added Memory map description.