µTasker Document

- **iMX RT Tutorial – Ethernet and the Simulator**

Table of Contents

1. 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 modular code.

The operating system is integrated with TCP/IP stack, USB stack, file systems and important embedded Internet services alongside 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 NXP i.MX RT Cortex M7 processors. It can be fully simulated using VisualStudio 2017 or higher and can be run on the NXP™ **iMXRT1020-EVK** board or various others, in which case the MCUXpresso IDE with its on-board CMSIS-DAP debugger is assumed. Building the project using GCC is included as an optional post-build step when working in VisualStudio, but debugging with the GNU tool chain is not discussed. Furthermore, alternative IDEs are available from IAR, Keil,Rowley Crossworks and others which can be used together with the project too.

For more µTasker documentation, including a guide to using the USB capabilities of the i.MX RT project visit: http://www.utasker.com/docs/documentation.html

Note that MODBUS support (serial and TCP) is available as an extension pack for the i.MX RT – see the MODBUS user's guide for full details:
http://www.utasker.com/docs/MODBUS/uTasker_MODBUS.PDF

Specific i.MX RT 1020 documents designed to help better understand and use the device in the µTasker project framework can be found here:

- "**i.MX RT 1021 – the µTasker way**"
https://www.utasker.com/docs/iMX/i.MX_RT_1021_uTasker.pdf

2. 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, but 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 your web browser, 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 Wireshark) for further analysis and playback…. Getting excited? In a few minutes you will see it in action!

> If you haven't VisualStudio (2017 or newer) then there are trial versions and the free Community edition version version [https://visualstudio.microsoft.com/vs/   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 to be installed (from http://www.winpcap.org) which is an industry-standard tool for link-layer network access in Windows environments. It is also used by the network sniffer Wireshark, which every Ethernet developer really should have. Therefore it is simplest to install Wireshark since this includes WinPCap, saving a step, and making for simplest installation. Wireshark can be downloaded from http://www.wireshark.org/ and is discussed 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 and can be obtaned from https://www.utasker.com/iMX/RT1020.html

You will find that the simulator is useful for most real embedded 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, LCDs, SD-cards, I^{2}C EEPROM or SPI FLASH (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 chance to get out more…!

So...

...fasten your seat belts since we are about to roll

1. Simply copy the complete μTasker folder to your PC and go to `\Applications\uTaskerV1.4`. This is a ready to run project directory showing a useful reference application using network resources (and more).

2. Move to the sub director `Simulator` and open the VisualStudio project workspace `uTaskerV1-4.vcxproj`.

3. Ensure that you select a build target for i.MX RT which will supply a pre-processor define `_iMX` and exclude HW sources for other HW targete that may co-exist in the framework (see next point for more details).

4. *The project for i.MX RT actually has two target settings; "uTasker i.MX" and "uTasker i.MX plus GNU build". Both will build the project for the simulator but the second will additionally cross-compile for the hardware target using the GNU compiler if it is installed and paths are set accordingly. For the first steps with the simulator the "uTasker i.MX" target can be used.*

5. Ensure that the project is configured for your target board in `config.h`: for the **iMXRT1020-EVK** select `MIMXRT1020` [*there are various other options that can be tried later, including the NXP boards for i.MX RT 1011, i.MX RT 1015, i.MX RT 1052, i.MX RT 1062 and i.MX RT 1064*].

   Ensure that the defines `BLINKY, HELLO_WORLD` and `REMOVE_PORT_INITIALISATIONS` are disabled (commented out) and that `USE_PARAMETER_BLOCK, SPI_FILE_SYSTEM, INTERNAL_USER_FILES, ETH_INTERFACE, USE_ICMP, USE_TCP, USE_FTP` and `USE_HTTP` are enabled (not commented out)

   Build the project (use F7 as short cut) and you should find that everything compiles and links without any errors.

6. I would love to be able to say "*Execute*" but there are a couple of things which have to be checked (on first use) 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 well but this may improve as the WinPCap capabilities get better with newer version).
   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 uTaskerV1.4 project directory 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.

Only IPv4 settings are shown above, although IPv6 settings can also be configured here.

It is also possible to change the settings when the target is running but we don't want to unnecessarily complicate things at this point and simply slotting in the best values to suit your network is the easiest way to ensure its operation.

7. 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 a 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 (PTA[11]). Since this is connected to the red LED on the board it will be seen that also this LED is blinking as the port output toggles.

Figure 1. µTasker simulator running an i.MX RT 1021 MIMXRT1020 board

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.

8. 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 Wireshark to do the job.
   Watch the LAN indicators on the processor's virtual LAN connector. The left one will blink green when a frame is received – if it is accepted by the device (matching address, broadcast or all when in promiscuous mode). The right one will blink red whenever the simulated device sends a frame to the network.

9. 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.

10. 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 web browser (Internet Explorer, Firefox, Chrome, etc.) and establish a connection with the simulated target: In the URL line type in http://192.168.0.3 (assuming its address has not been changed) and hit the ENTER key. The web browser will request the start page on the web server, which will return the following page the first time used:
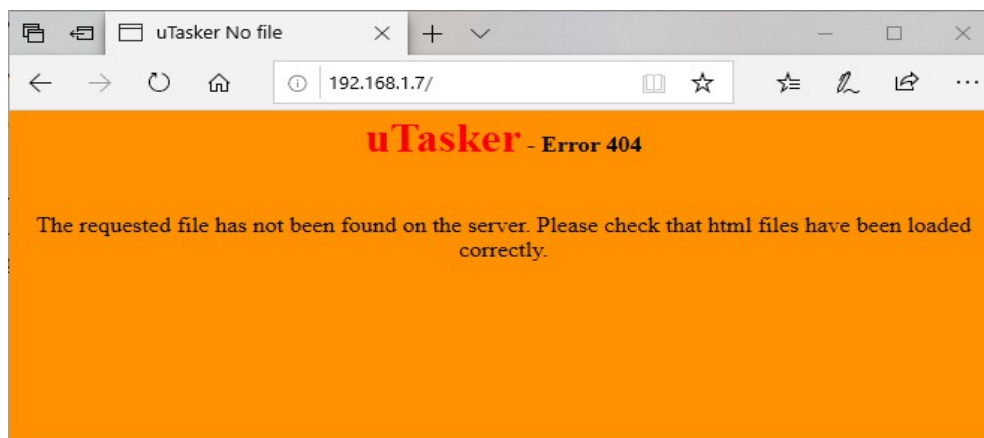

Figure 2. µTasker web server displaying that it has no web pages loaded

This is displaying that the page has not been found on the server, which is simply due to the fact that we haven't loaded any web pages to it yet, so we will do this as next step.

Note however that the web browser may display the NXP favicon. This is because this favicon is embedded in the code and not in the µFileSystem – this is explained in detail in the user file document

http://www.utasker.com/docs/uTasker/uTaskerUserFiles.PDF

If the favicon is not displayed it can be viewed by browing directly to

http://192.168.0.3/favicon.ico

11. Take a look in the project directory
    `\Applications\uTaskerV1.4\WebPages\WebPagesiMX`
    and you will find the web pages prepared for the refernece project. These are HTM, JPG and GIF files. It is possible to load these pages using FTP clients but these tend to work quite differently from another and may require various setups before they operate smoothly. See the following document for more information on this subject:
    http://www.utasker.com/docs/uTasker/uTaskerFTP.PDF

    This is the reason that working with simple DOS FTP tends to be the fastest and simplest method and the bat file `Copy_all.bat` even does all the work for you. Check that its content is set to use the correct IP address and then double-click the file to start the transfer of all of the web pages to the simulated target device. The transfer takes just a few seconds.

    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 pages.
    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 in 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 **IS25XXX.ini**. This is the content of simulated SPI flash used save the parameters and file system in since the i.MX RT don't have internal flash memory. *If a different board is simulated that uses a different flash type you will find a different file name that matches its flash type.*

    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 same FLASH contents as at the last program exit*. This is exactly how the real device works since, after a reset, its SPI FLASH content is 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 **IS25XXX.ini** file from the simulator directory…

12. 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 (*with the exception of some DHCP servers not allocating IP addresses to a device with this MAC address*), 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 first byte doesn't end with a '1', otherwise it would be a multicast MAC address*) and then click on "modify / validate settings". The entry field will be set inactive since it is no longer zero and cannot 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:

Figure 3. µTasker simulator indicating that the processor has just been 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.

13. 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 may still be 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. Some PCs may not always accept this command if not executed with administrator rights so wait a couple of minutes before starting the simulator again, until its ARP table automatically times out.

   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 the process.

14. 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 of possibilities is almost endless…
   Look at the simulated i.MX RT and imagine that you have connected your lawn sprinkler to the output Port GPIO1-18 (known as pad GPIO_AD_B1_02 too].
   Presently it is at the state input '1' [signified by '^'], the default state when the software starts since there is a pull-up on the port. 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 GPIO1-18 is being displayed as an input and its digital value as active ('1') in its check box.

   Do a quick test: Hover your mouse over the port GPIO-18 in the simulator – it will tell you which pin it is on 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 on 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.
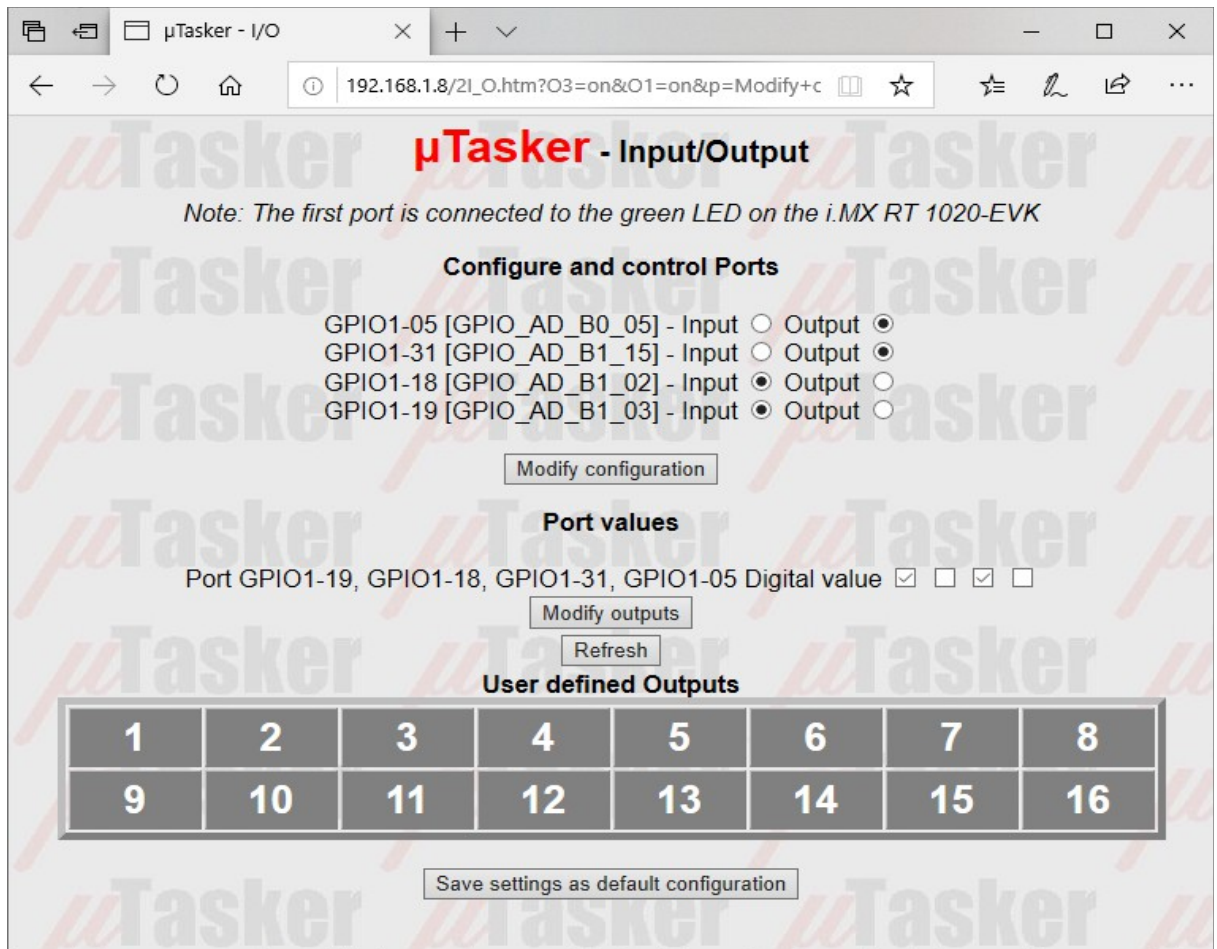
Figure 4. µTasker web server showing the I/O control page

Set GPIO1-18 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 GPIO1-18 on the simulated device – 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 GPIO1-05, GPIO1-31, GPIO1-18 and GPIO1-19 outputs can be controlled individually. Since GPIO1-05 is connected to the green LED on the board you can also stop the LED blinkin by configuring it as an input.

GPIO1-05 is used as a RUN LED in the demo project and so is configured as an output by default, as is GPIO1-32 – often used as a general purpose test output.
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 reference 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 varios outputs on GPIO1 and GPIO2 that are initially in the '0' state. 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 reference

project – it shows a useful example of flexible independent output control.

15. The last setting which we will change before taking a look at the project on the target hardware 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 your web browser..
    Open the web browser 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 (like transfer the file with *Copy_all.bat* or delete the file system contents with `delete_all.bat`) 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.

    You probable noticed a button at the bottom of the administration page which allows a file to be searched for and uploaded to the target as a new software version. This is an example of using the HTTP POST method to transfer data to the embedded device. The μTasker project for the i.MX RT supports the upload of complete new software using this method (as well as by FTP and even USB). For further details about using this capability on the i.MX RT see the document http://www.utasker.com/docs/uTasker/uTasker_BM_Loader.pdf :You will see that you will then be able to develop code for a real target on the internet, including uploading it from anywhere where you have web access to it!

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 terms of 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.

3. 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 i.MX RT and done this in an environment enabling us to do embedded IP stuff in real time. We haven't actually done any debugging or added new code but you can probably already 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…

If you don't want to cross compile the program just now you can also download read-to-run binaries for the i.MX RT 1020 board from https://www.utasker.com/iMX/RT1020.html

<u>Even if cross compiling the application project the boot loader should be downloaded from there and installed on the board</u> so that the the application can subsequently be loaded and started by it. Once more experienced you will be able to build also the boot loader projects, customising them for your own requirements if necessary.

4. Installing the Serial Loader

The serial loader for our board can be found at the link in the previous chapter and is called "`uTaskerSerialLoader_i.MX-RT1020_USB-MSD_Kboot-HID_Kboot-UART.bin`". The loader is an integral part of the μTasker project which allows convenient uploading of applications to the board by local methods, such as USB-MSD, or remote methods – often referred to as OTA (Over-Then-Air) loading. It contains all of the required methods to do this reliably and securely and installs applications that reside in SPI flash when they are to be executed, allowing them to run exclusively in internal memory to achieve maximum performance and minimum interference.

This bootloader needs to be loaded into the SPI flash on the board, which can be performed by a number of methods. Here one is shown that makes use of the processor's ROM loader and a useful program contained in NXP's MCUBootUtility. Start by obtaining this MCUBootUtility project from this link: https://github.com/JayHeng/NXP-MCUBootUtility

The utility doesn't have to be built, although it could be, but is available as executable program (for Windows) in its `/bin` folder - "**NXP-MCUBootUtility.exe**"

When the program is running there are two windows seen:
- the first is a DOS shell where the commands that it sends when operating can be viewed
- the second is a Windows application
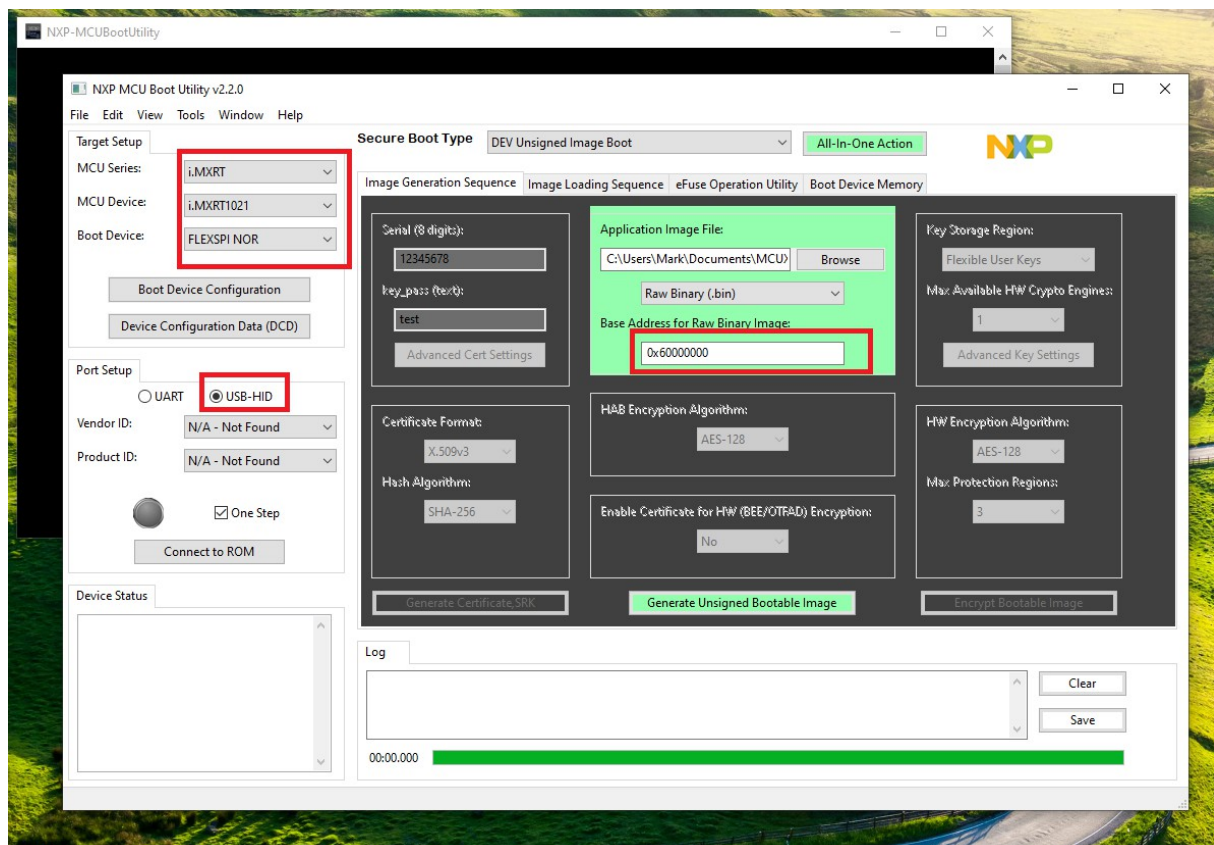
Figure 5. MCUBootUtility program running

On the start screen the MCU Series, MCU Device and Boot Device can be selected as shown. The USB-HID method will be used, whereby the processor's USB cable is connected to the PC. The base-address for the raw binary image is `0x60000000` (6 with 7 zeros), which is the start address of the SPI Flash in the i.MX RT1020 device.

We then need to connect to the processor, which means that the processor must be running in its ROM loader mode. To force the processor into this mode one used the DIP-switch on the bottom of the board.
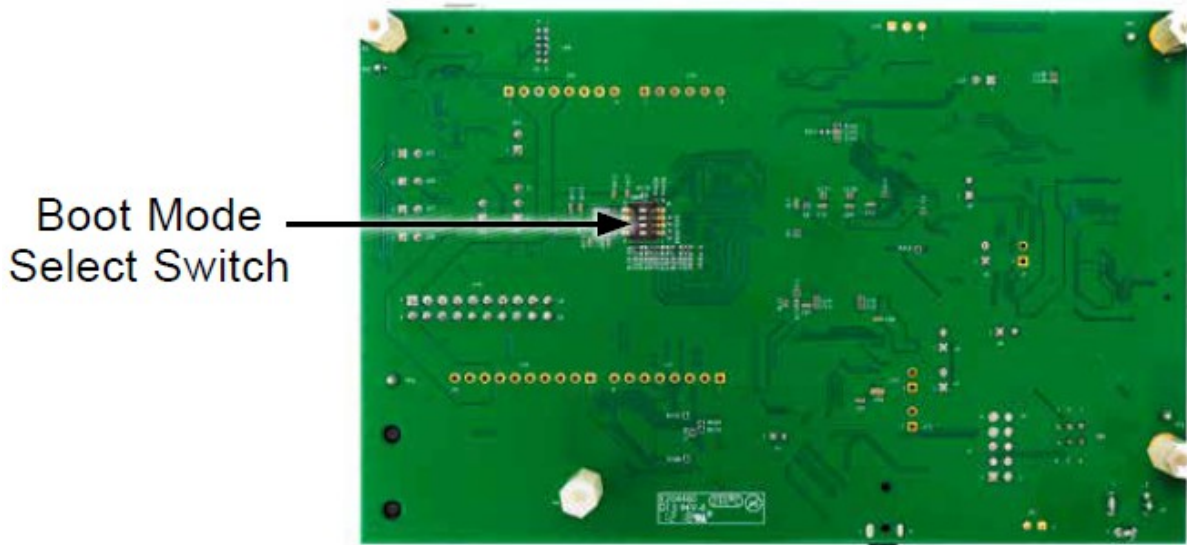
Figure 6. Back side or MIMXRT1020-EVK

Two switch settings will be used:
- for normal operation the switches are set to OFF-OFF-**ON**-OFF, which informs the ROM loader in the device to boot from the SPI-flash; the SPI flash is in fact an 8MByte Quad-SPI flash called the *IS25LP064A*
- for loading the binary file the switches are set to OFF-OFF-OFF-**ON**

Therefore, after changing the switch settings to the second option, the board needs to be reset by pressing either SW5 or SW3, or by power-cycling it. After the processor restarts it will be in the ROM loader mode and its connected USB will enumerate at the PC host. In turn, the MCUBootUtility will detect that the processor is connected and display its vendor and product IDs:
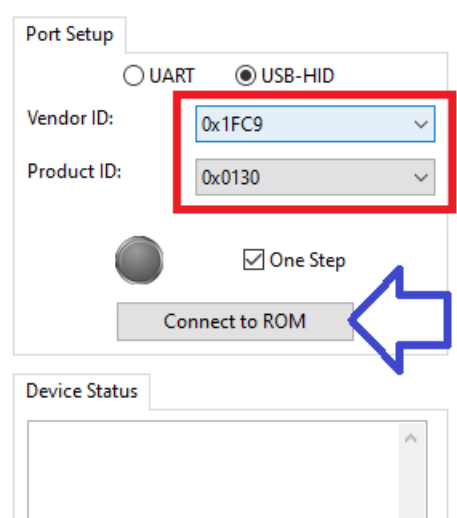
Figure 7. MCUBootUtility recognising connected device via USB-HID

By clicking on the "Connect to ROM" button a connection can be established with the processor (*one will need to get used to the arcade-style sounds that the program makes when connecting*). The following shows the device connected, where information about its SPI flash is displayed:
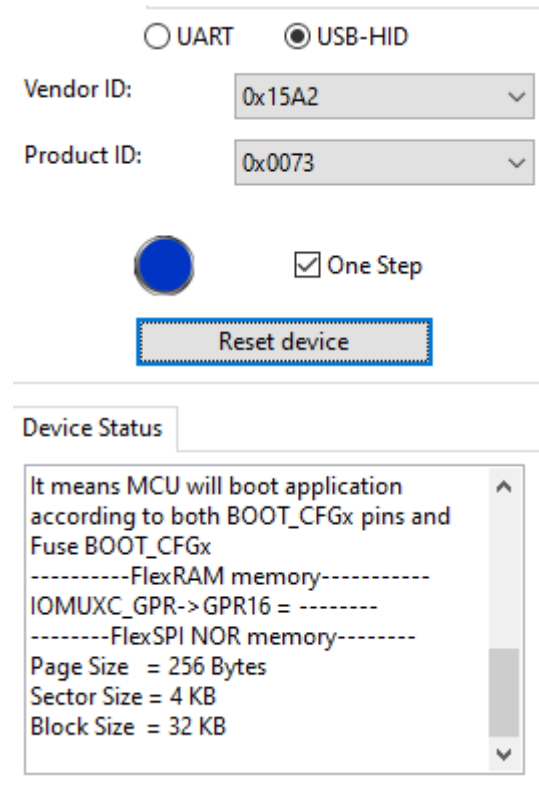
Figure 8. MCUBootUtility connected to device via USB-HID

Now we navigate to the "Boot Device Memory" tab:
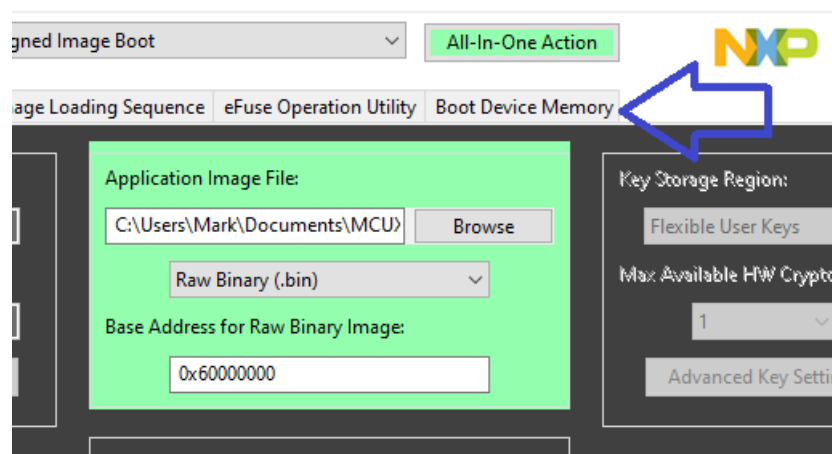
Figure 9. MCUBootUtility Boot Device Memory tab

to get to the screen which we will use for programming our binary file.
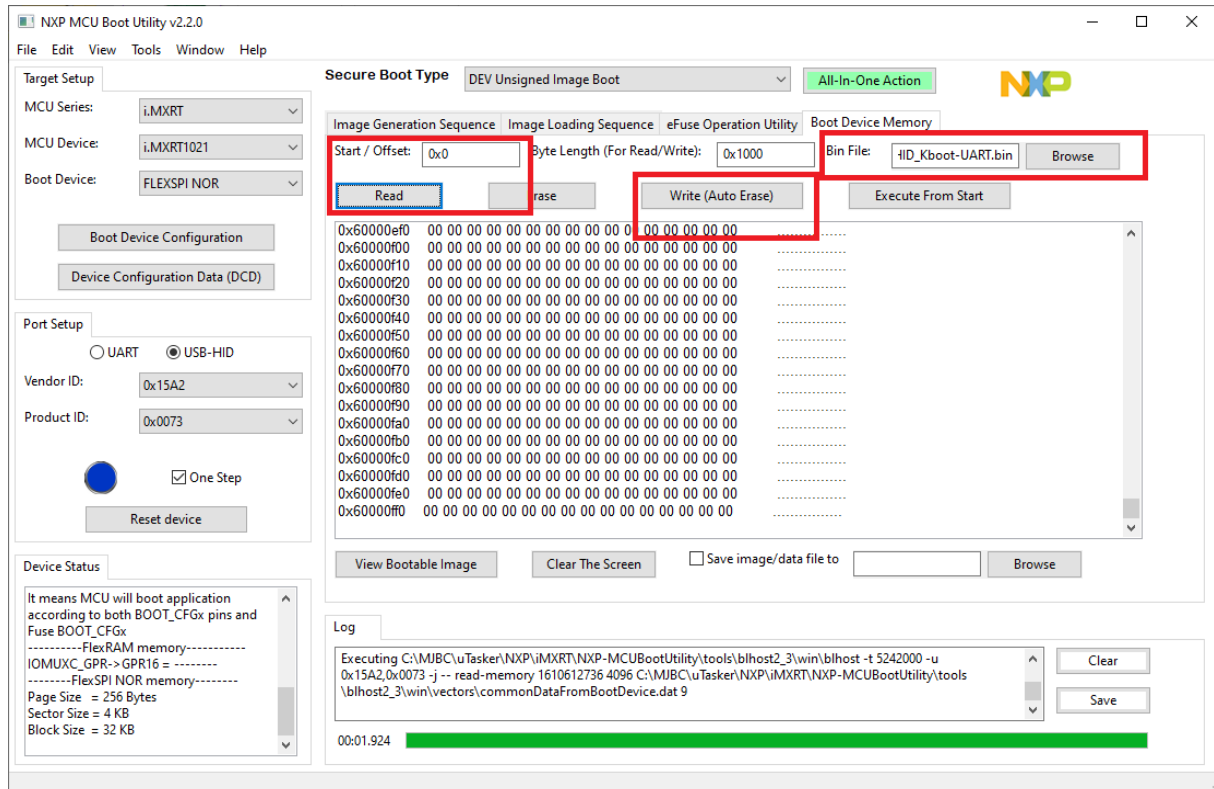
Figure 10. MCUBootUtility ready to download the binary file

This is also useful for viewing the data that is already programmed on the SPI flash chip or erasing selected parts of it. Note that the "*Start / Offset*" setting MUST be 0x0 when loading the boot loader since we need it to be programmed at the very start of the SPI flash. The boot loader binary is selected for the Bin File to be loaded, and all that needs to be done now is to click on the "*Write (Auto-Erase) button*". The binary file is then loaded, which takes about 2s.

Finally one sets the DIP-switch on the underside of the board back to its original OFF-OFF-**ON**-OFF setting and resets or power-cycles in order to start the processor again so that it executes the µTasker serial loader.

Assuming that there is no µTasker application already loaded on the board the USB-MSD loader will be seen as an external hard drive to the PC called "*UPLOAD_DISK*" (it also supports NXP's Kboot protocol but this tends to be less practical for general work and the USB-MSD loader is also much quicker to use).

Assuming the application has been built or the reference application from this link is to be loaded it can be done so by simply dragging the file onto the disk. After the copy has completed the board reboots and starts the application.

In order to enter the µTasker serial loader again the board can be reset or power-cycled with the USER button (SW4) held down; when this is not done the board always boots to a valid loaded application.

The USB-MSD mode of the serial loader is detailed in the µTasker Serial Loader document:
https://www.utasker.com/docs/uTasker/uTaskerSerialLoader.pdf

5. Compiling the Project for the Target

Although the µTasker project can be cross-compiled using a variety of different compilers/IDEs we will stick with Visual Studio at the moment and build it as a post-build GCC step. This in fact can use the same GCC compiler as MCUXpresso uses in order to effectively generate the same object as can be generated by building it in MCUXpresso. *MCUXpresso is NXP's free Eclipse based IDE which tends to be used by a large percentage of i.MX RT developers*. When working with the simulator it tends to be more efficient to edit, debug and generate the object as a single process in Visual Studio.

For a general overview of building the project with various different IDEs see the following link: https://www.utasker.com/kinetis/compilers.html It is intended primarily for use by Kinetis developers but remains basically true also for i.MX RT work.

4.1. Compiling with GCC post-build step in Visual Studio

The project is delivered with a bat file and make file so that the project can be simply compiled with the GNU compiler. It is recommended to install MCUXpresso and use the GCC tools that it installs.

From the µTasker project in Visual Studio open the bat and make files to check that they are configured correctly for the local PC and the device to be built for. The files are included in the VS project so that they can be easily located and opened:
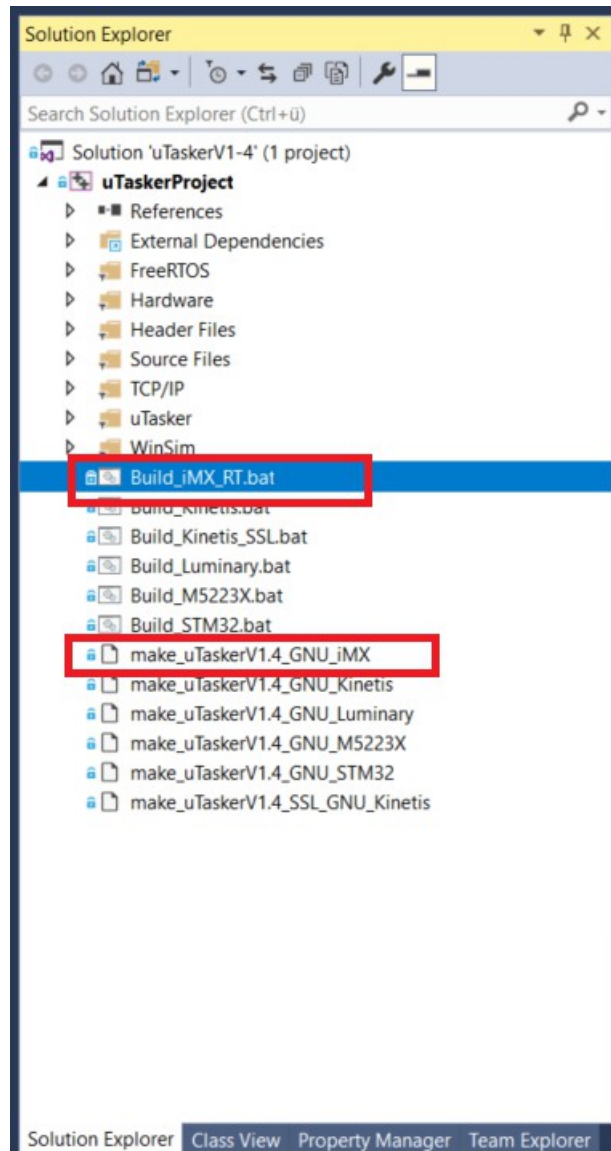
Figure11. Location of bat and make files in VS project

In the "`Build_iMX_RT.bat`" file make sure that the path is set correctly so that the tool chain will be successfully found. Eg, this is the path that MCUXpresso is installed on my PC but change the entry to suit yours if necessary:

```
SET PATH=%PATH%;C:\nxp\MCUXpressoIDE_11.1.0_3209\ide\tools\bin
```

To ensure that the make program (`make.exe`) is compatible, please download the one from here: https://www.utasker.com/forum/index.php?topic=1988.msg7410#msg7410 and copy it to that \tools\bin directory too. *It will not overwrite anything in the directory but adds an alternative make executable.*

In the make file "`make_uTaskerV1.4_GNU_iMX`" verify that the compiler flag setting is correct for a Cortex M7 with double-precision floating point unit:

```
# Compiler flags (thumb mode - little endian) - if a part with single -precision FPU
choose -mfpu=fpv5-sp-d16, else -mfpu=fpv5-d16 (double-precision FPU)
#C_FLAGS = -mcpu=cortex-m7 -mthumb -mfloat-abi=hard -mfpu=fpv5-sp-d16 -mlittle-endian
-mthumb -Wall -Wstrict-prototypes -fmessage-length=0 -fsigned-char -ffunction-sections
-fdata-sections -Wundef
C_FLAGS = -mcpu=cortex-m7 -mthumb -mfloat-abi=hard -mfpu=fpv5-d16 -mlittle-endian
-mthumb -Wall -Wstrict-prototypes -fmessage-length=0 -fsigned-char -ffunction-sections
-fdata-sections -Wundef
```

and that the correct linker script is being used:

```
uTaskerV1.4_BM.elf: $(OBJS) $(OBJS2_BM) iMX_RT_10XX_FlexSPI_NOR_BOOT.ld

        $(CC_LNK) $(C_FLAGS) -Os -g -TiMX_RT_10XX_FlexSPI_NOR_BOOT.ld -nostartfiles
-Xlinker --gc-sections -lm -Wl,-Map=uTaskerV1.4_BM.map -o uTaskerV1.4_BM.elf $(OBJS) $
(OBJS2_BM)
```

In visual studio select the build target "`uTasker i.MX plus GNU build`" rather than

"`uTasker i.MX`" so that the GCC post build step is executed when the project is build. When the build is performed you will see the GCC build steps in the output windows, looking something like this:

```
...
1>arm-none-eabi-gcc -mcpu=cortex-m7 -mthumb -mfloat-abi=hard -mfpu=fpv5-d16 -mlittle-endian
-mthumb -Wall -Wstrict-prototypes -fmessage-length=0 -fsigned-char -ffunction-sections -fdata-
sections -Wundef -I../  -D _GNU -D _iMX -g -c -Os ../../../stack/mqtt.c -o Build/mqtt.o

1>arm-none-eabi-gcc -mcpu=cortex-m7 -mthumb -mfloat-abi=hard -mfpu=fpv5-d16 -mlittle-endian
-mthumb -Wall -Wstrict-prototypes -fmessage-length=0 -fsigned-char -ffunction-sections -fdata-
sections -Wundef -I../  -D _GNU -D _iMX -g -c -Os ../../../stack/secure_layer.c -o
Build/secure_layer.o

1>arm-none-eabi-g++ -mcpu=cortex-m7 -mthumb -mfloat-abi=hard -mfpu=fpv5-d16 -mlittle-endian
-mthumb -Wall -Wstrict-prototypes -fmessage-length=0 -fsigned-char -ffunction-sections -fdata-
sections -Wundef -Os -g -TiMX_RT_1020_FlexSPI_NOR.ld -nostartfiles -Xlinker --gc-sections -lm
-Wl,-Map,"uTaskerV1.4.map" -o "uTaskerV1.4.elf" Build/application.o Build/debug.o
Build/webInterface.o Build/KeyScan.o Build/DMX512.o Build/NetworkIndicator.o
Build/usb_application.o
...
Build/snmp.o Build/tcp.o Build/telnet.o Build/tftp.o Build/udp.o Build/webutils.o
Build/NetBIOS.o Build/zero_config.o Build/ftp_client.o Build/igmp.o Build/ppp.o Build/mqtt.o
Build/secure_layer.o Build/iMX.o

1>arm-none-eabi-objcopy --only-section=.data --only-section=.init --only-section=.text --only-
section=.rodata --only-section=.vectors --output-target=srec uTaskerV1.4_BM.elf
uTaskerV1.4_BM.srec

1>arm-none-eabi-objcopy --only-section=.data --only-section=.init --only-section=.text --only-
section=.rodata --only-section=.vectors --output-target=ihex uTaskerV1.4_BM.elf
uTaskerV1.4_BM.hex

1>arm-none-eabi-objcopy --only-section=.data --only-section=.init --only-section=.text --only-
section=.rodata --only-section=.vectors --output-target=binary uTaskerV1.4_BM.elf
uTaskerV1.4_BM.bin

1>arm-none-eabi-size uTaskerV1.4_BM.elf
```

```
1>   text         data    bss    dec     hex filename
1> 76428          152    5524  82104   140b8 uTaskerV1.4_BM.elf
```

1>arm-none-eabi-gcc -v

1>Using built-in specs.

1>COLLECT_GCC=arm-none-eabi-gcc

1>COLLECT_LTO_WRAPPER=c:/nxp/mcuxpressoide_11.1.0_3209/ide/tools/bin/../lib/gcc/arm-none-eabi/8.3.1/lto-wrapper.exe

1>Target: arm-none-eabi

1>Configured with: /tmp/jenkins/jenkins-GCC-8-build_toolchain_docker-594_20190704_1562200936/src/gcc/configure --build=x86_64-linux-gnu --host=i686-w64-mingw32 --target=arm-none-eabi -prefix=/tmp/jenkins/jenkins-GCC-8-build_toolchain_docker-594_20190704_1562200936/install-mingw
...

1>Thread model: single

1>gcc version 8.3.1 20190703 (release) [gcc-8-branch revision 273027] (GNU Tools for Arm Embedded Processors 8-2019-q3-update)

1>   uTaskerConvert V1.11 - supporting encryption, AES256 and Motorola binary format [2M binary support]

If all is successful a file called "**uTaskerV1.4_application.bin**" will be created in the directory `\Applications\uTaskerV1.4\GNU_iMX` which is in a format suitable for uploading to the board using the µTasker serial loader as discussed in the previous chapter.

The USB-MSD mode of the serial loader is detailed in the µTasker Serial Loader document:
https://www.utasker.com/docs/uTasker/uTaskerSerialLoader.pdf

Often the simulator's target can be left like this so that each time the project is built it can be simulated and loaded to the HW target without requiring any additional steps to be performed.

Once done, make sure that the board is connected to the network and give it a test run and repeat the exercise that you have already done with the simulator.

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, you can simply 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.

6. 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.

6.1. Menu

Figure 12. µTasker web server showing the menu page

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 cannot be modified (disabled).

Software version – this is a string which is defined in `application.h`. It is also disabled so that it cannot 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 remains valid after the next reset or power up.

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

Email address – If you have configured the demo project to support SMTP (`#define USE_SMTP` in `config.h`) then it is possible to enter an email address and send a test email to it. This entry is disabled if the support is not active in the project.

Note that it is necessary to configure some SMTP settings before an email can actually be sent – the default settings will not work since they are fictive and will not correspond with your email provider. Full details to the SMTP and its use in the µTasker demo project are given in the document http://www.utasker.com/docs/uTasker/uTaskerSMTP.PDF

Logo and upload field – The i.MX RT logo was uploaded to the file system as a GIF file when the files were loaded. It is possible to change this logo for a different GIF file by selecting a GIF and pressing the "*Upload now*" button. The GIF may be up to 2.8 Meg in size (*this is the size reserved for it in the internal file system*).

Each time this side is served by the web server it will increment the count value and display the present IP connection. An IPv4 address is shown since the connection was made using IPv4. IF IPv6 operation is enabled it is possible to connect on an IPv6 address and the displayed address will then be the corresponding IPv6 address.

6.2. 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. (*See the short discussion about MAC addresses at the bottom is this section for some details about how they are obtained*.)

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.

If IPv6 support is enabled a global IPv6 address can be assigned to the device as well as an IPv4 tunnelling address for &in4 tunnelling operation – see the IPv6 User's Guide for full details: http://www.utasker.com/docs/uTasker/uTaskerIPV6.PDF

The Ethernet speeds of 10M, 100M or auto-negotiation (full or half-duplex) can be set.

Figure 13. μTasker web server showing the LAN configuration page

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 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 within arm's reach 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 also that these are only provisional (not validated) - the original values are also still available. These new values are nevertheless used for a 'trial-period' 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 'silly' mistake is made.

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.

6.3. Serial Configuration

This web page allows the debug interface UART in the i.MX RT to be set. See the UART User's guide for some more details of this interface:
http://www.utasker.com/docs/uTasker/uTaskerUART.PDF

Figure 14. µTasker web server showing the serial interface configuration page

6.4. 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…

µTasker - statistics

Refresh

IP frame / ARP statistics

RX	Number of frames	TX	Number of frames	ARP entries in cache	
Total RX frames	2845	Total TX frames	1691	192.168.0.99	f0-4d-a2-9d-94-dc
Overruns	0	ARP frames	9	192.168.0.1	00-40-05-d1-df-52
Frames to us		ICMP frames	0	- -	- -
ARP frames	9	UDP frames	0	- -	- -
ICMP frames	0	TCP frames	1682	- -	- -
UDP frames	1061				
TCP frames	1526				
Broadcast frames					
ARP frames	249		Reset frame stats		Delete ARP entries
ICMP frames	0				
UDP frames	0	Other events	0		

IPv6 / Neighbor Cache

RX	Number of frames	TX	Number of frames	NN entries in cache	
				fe80::bd88:231d:59f0:6890	f0-4d-a2-9d-94-dc
				- -	- -
ICMP frames	1	ICMP frames	1	- -	- -
UDP frames	0	UDP frames	0	- -	- -
TCP frames	451	TCP frames	705	- -	- -
		Reset frame stats		Delete NN entries	

Figure 15. µTasker web server showing the statistics page

If IPv6 support is enabled IPv6 counters are also shown. The IPv6 Neighbour cache is similar to the ARC cache as used by IPv4.

6.5.   I/O Page

The I/O window was discussed in step 13 of the tutorial and is not discussed in any more detail here.

6.6.   Administration Side

On this page the FTP server can be activated and deactivated. See also step 14 of the tutorial.

Figure 16. µTasker web server showing the administration page

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.

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 before the original ones will become permanent. This is again to ensure that there is no danger of losing 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. 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.

Present Time – If the demo project is configured to collect the timer from a time server, this displays the present time. The time may also be available if the device which the µTasker is operating on processes a RTC.

Running Time - 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.

Last Reset – The last reset cause is displayed. This can (for example) be due to a power up, a software reset, an external reset or a watchdog reset.

6.7.   LCD Side

When the graphical LCD support is enabled (define `SUPPORT_GLCD` in `config.h`). The LCD page will display the present LCD content. It also allows the user to post am image to be displayed on the graphical LCD.
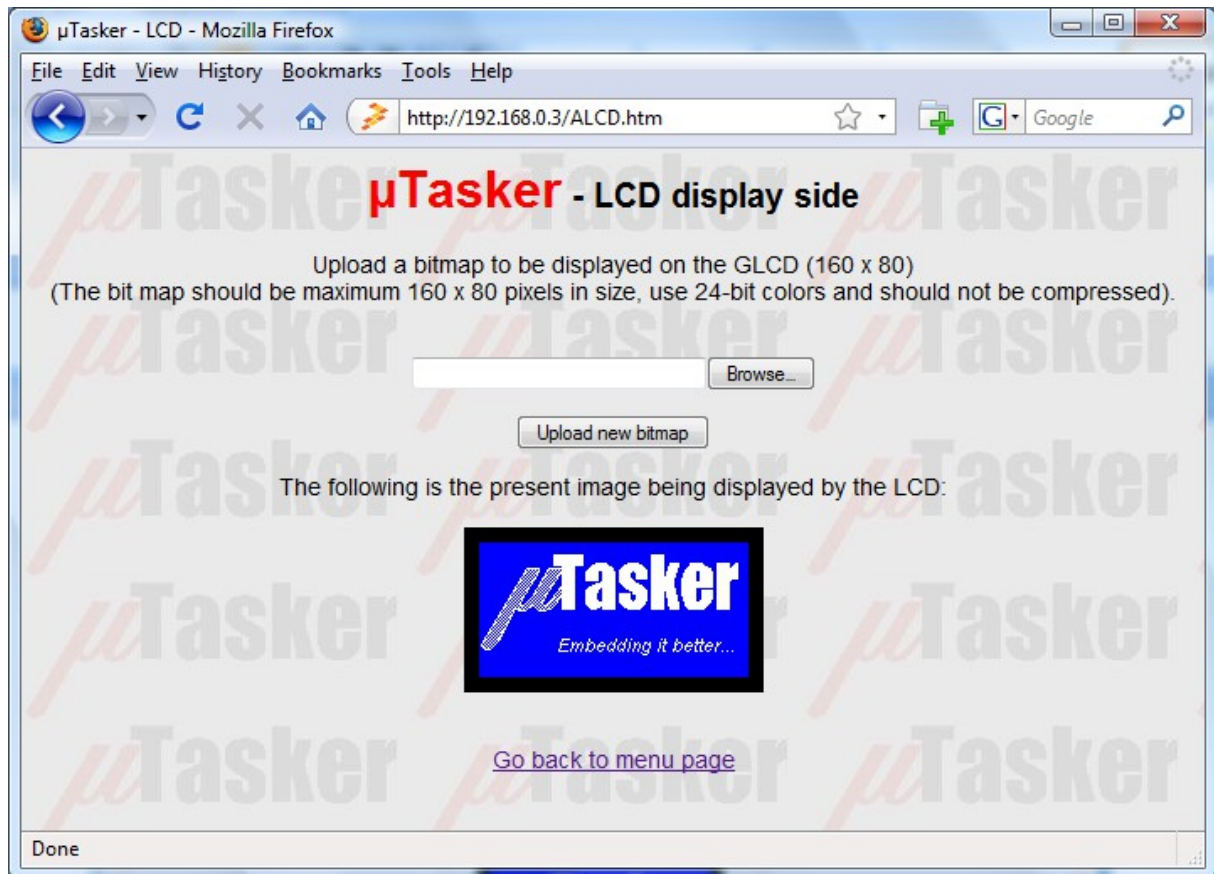

Figure 17. LCD page showing the present LCD image

For full details about the graphical LCD support in the project please consult the LCD User's Guide at http://www.utasker.com/docs/uTasker/uTaskerLCD.PDF Although a 160 x 80 pixel monochrome LCD is shown the i.MX RT project also supports various colour TFT displays.

6.8.   Email Configuration Side

The SMTP configuration can be performed on this web page corresponding to the email provider's requirements.

For more details see the document
http://www.utasker.com/docs/uTasker/uTaskerSMTP.PDF
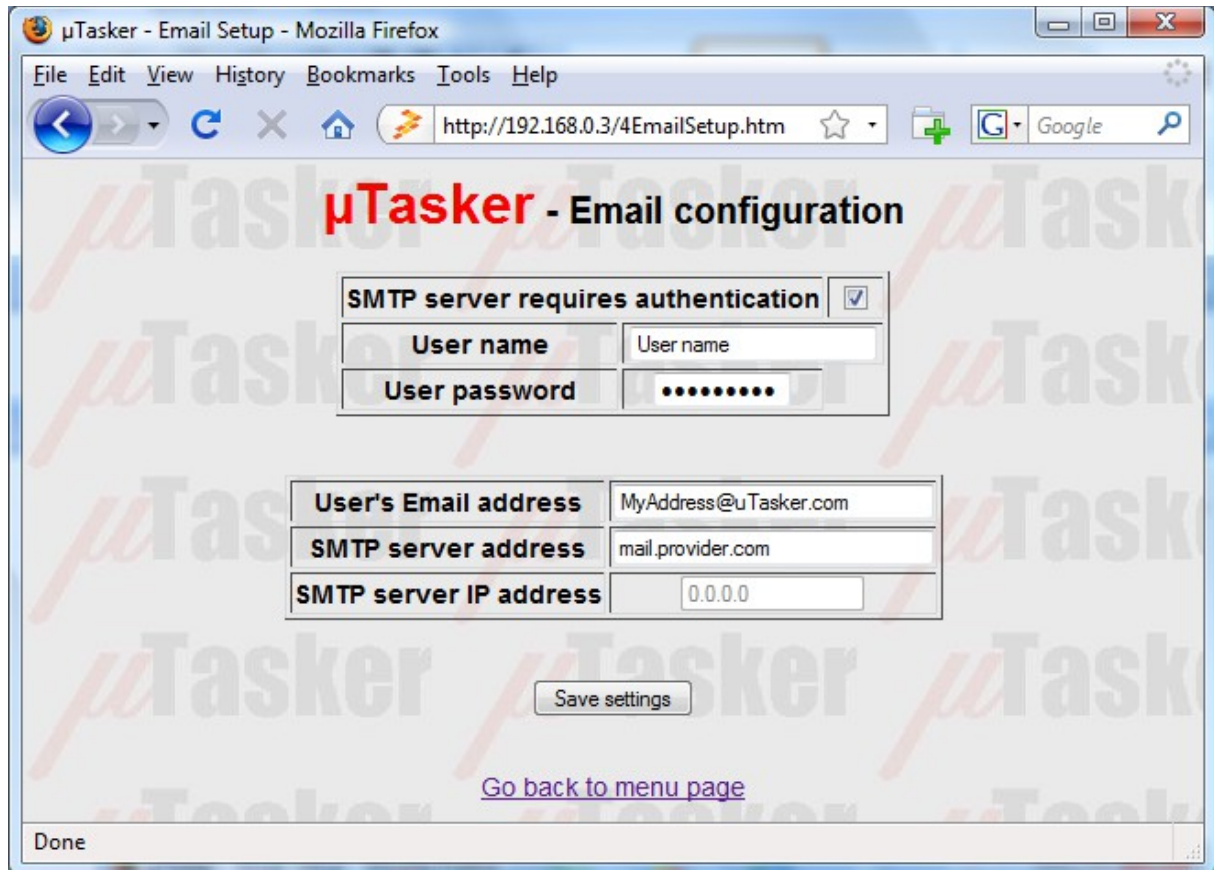
Figure 18. μTasker web server showing the Email configuration page

6.9.   Dynamic Content Generation Side

This web page contains two different demonstrations.

The first allows downloading an 8MByte test file from the board, which is generated on demand.

The second enables the user to configure the dimensions for a multiplication table which is generated on demand by the board.

For full details of this operation see the document
http://www.utasker.com/docs/uTasker/uTaskerV1.3_Content_Generation.PDF

Figure 19. Dynamically generated multiplication table

7. 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 modify existing code and add new 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 of 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 manner. 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 Wireshark

If you are using the µTasker then you will certainly want 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 Wireshark (previously called Ethereal), a free and powerful network Sniffer. You can download this from http://www.wireshark.org/

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 Wireshark and start monitoring the traffic on your local network (`Capture | Interfaces...`) and select the Ethernet interface to be monitored. (`Capture | Start`) to start capturing.

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 Wireshark recording and save it to the directory `Applications\uTaskerV1.4\Simulator\Ethereal\` with the name `ping.pcap` (there is already such a file there as reference too).

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

```
No. Time        Source           Destination      Prot  Info
1   0.437770    192.168.0.100    Broadcast        ARP   Who has 192.168.0.3? Tell192.168.0.100
2   0.439926    00:00:00_11:22:01 192.168.0.100   ARP   192.168.0.3 is at 00:00:00:11:22:01
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 Wireshark Help.

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

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

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

In the VisualStudio project, locate the file `kinetis_ENET.h` (in `hardware\kinetis\`) and search for the Ethernet reception interrupt routine called `_fec_rx_frame_isr ()`. Position the cursor over the first open-bracket after the interrupt subroutine definition and press F9 to set a break point there. [*Note that the file name is not a mistake because the i.MX and Kinetis Ethernet driver is shared between the two projects due to the fact that the Ethernet controller in the parts is more or less the same!*]

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 because 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 i.MX RT, this will also give us the opportunity to look at some of its internal registers on the way.

A. Interrupt routine (*note that this has been simplified for display here*)

```
static __interrupt void _fec_rx_frame_isr(void)
{
    uTaskerStateChange(TASK_ETHERNET, UTASKER_ACTIVATE); // schedule the Ethernet task
    EIR = RXF;                                           // reset interrupt flag
#ifdef _WINDOWS
    EIR &= ~RXF;
#endif
}
```

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

We see that such routines are necessarily quite hardware specific. In fact the i.MX RT interrupt routine is very simple and has only to clear the interrupt flag which is performed on the target by writing a '1' to the corresponding bit. The simulator uses a slightly different technique and clears the bit as is normally performed in a variable. The register EIR (Ethernet Interrupt event Register) can be found in the data sheet to the device if you would like to learn more about its details and it exists in the simulated i.MX RT, where it can be looked at as follows.

Double click on EIR in the code and search for it 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 iMX.h, where it is defined as:

```
#define EIR *(volatile unsigned long *)(EMAC_BASE_ADD + 0x4)
```

where EMAC_BASE_ADD is also defined locally twice:

```
  #define EMAC_BASE_ADD        ((unsigned char *)(&iMX.EMAC))
```
and
```
  #define EMAC_BASE_ADD        (0x402d8000)
```

On the target, the register is located at the address 0x402d8000 and when simulating it can be found in a structure called I.MX.

Double click on the word `iMX` 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. `EMAC_ISR` is in the sub-block `EMAC` so we also expand this sub-structure by clicking on the cross left of its name. If only the EMAC peripheral block is of interest this block can be dragged alone to the watch windows by selecting `iMX.EMAC` and dragging this to there.

In the sub-block you will see all of the EMAC registers listed in the order in which they occur in memory as well as showing their present values. Step the code (F10) and you will see that the content of `EIR` is changed as the code resets the interrupt flag after setting the state of the Ethernet task to be scheduled at the next opportunity

```
uTaskerStateChange(TASK_ETHERNET, UTASKER_ACTIVATE); // schedule the Ethernet task
```

If you want to, you can also step into the `uTaskerStateChange()` 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 continuing with the reception processing

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 open-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 extracts the received frame (when available) using `fnEthernetEvent()`. It learns about the length of the received frame and gets 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 apart from to say that the EMAC in the i.MX RT uses buffer descriptors to coordinate Ethernet frames in RAM. These buffers were configured when the Ethernet interface was initialised and the routine retrieves the relevant information about where the frame is and how long it is…

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.

Note that the ordering of elements in structures may change between version of the uTasker project but the principle of operation remains equivalent.

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 is 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 i.MX RT registers again. Very briefly you should understand that the i.MX RT has been set up with one or more transmission buffers 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 `kinetis_ENET.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.

8. 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 Wireshark 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 Wireshark recording

In the menu `Wireshark` select `Load Wireshark/Ethereal file to play back`. Then select the `ping.eth` file which was previously recorded. If this is later repeated, the command `Replay last File` can be used instead – this is also valid after the simulator is terminated normally.

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

This can be a great advantage when debugging protocols!

Before terminating 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 Wireshark recording is played back, the internal NIC is closed so that there is no disturbance from the network.

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

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

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

5. Wireshark recordings do not 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 as 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!

9.   Conclusion

This tutorial first introduced the µTasker simulator's capabilities to execute the demo project software on a simulated i.MX RT board. It showed the basics of working with the simulator and its Ethernet capabilities before turning attention to running the same project on a target board.

By working with Wireshark and the µTasker simulator's ability to play back Wireshark recordings through its complete software code, interesting methods for increasing development, debugging and test efficiency have been demonstrated.

The i.MX RT project supports various other features including USB. All features can be simulated using similar built-in techniques, thus enabling all aspects of real-world embedded projects to be performed with increased efficiency in comparison with convention development and debug techniques. Combining the µTasker simulator with the features of the µTasker demo project it represents the fastest method of starting and completing embedded projects with the supported devices.

Modifications:
- V1.0 23.02.2020 First Release.