Using the uTasker project with i.MX RT and GCC make file

The make file build method can be used from within VisualStudio – see the i.MX RT tutorial for more details: https://www.utasker.com/docs/iMX/uTaskerV1.4_iMX.pdf

Alternatively the project files can be edited in any editor and cross compiled by executing their respective bat files.

Building the "Bare-Minimum" (BM) Boot Loader [*if there is no archive file for the target being developed for this step should be completed before building the serial loader(s) and application!*]

The "Bare-Minimim" boot loader code is located in the folder
`\Applications\uTaskerBoot\`

Configure the target to be used in the project's `config.h` file – for example
`#define MIMXRT1020`

In the project's make file (\Applications\uTaskerBoot\GNU_iMX\`make_uTaskerBoot_GNU_iMX`) ensure that the floating point setting is correct for the device; in this case it would be
`-mfloat-abi=hard -mfpu=fpv5-d16`
For an i.MX RT processor without double-precision FPU `-mfloat-abi=hard -mfpu=fpv5-sp-d16` is used instead.

The bat file can be edited to produce a dedicated archive file for the target in question by setting the variable
```
set iMX_RT=1020
```

The project can be built by executing
`\Applications\uTaskerBoot\GNU_iMX\Build_iMX_RT.bat` and results in an output binary file
`\Applications\uTaskerBoot\GNU_iMX\`**`uTaskerBoot.bin`**
as well as an archive file
`\Applications\uTaskerBoot\GNU_iMX\`**`uTaskerBoot_1020.bin`** which is designed to be located in SPI flash and boot the processor. This image is however never used alone and is combined with the fall-back loader in the next step

Building the Serial Loader (fall-back loader version) [*if there is no archive file for the target being developed for this step should be completed before building the serial loader and application!*]

The Serial Loader code is located in the folder `\Applications\uTaskerSerialBoot\`

Configure the target to be used in the project's `config.h` file – for example
`#define MIMXRT1020`
and also configure the serial loader options as required. HW details can be configured in the project's `app_hw_iMX.h` file.

Make sure that the `define iMX_FALLBACK_SERIAL_LOADER` is enabled!

In the project's make file (\Applications\uTaskerSerialBoot\GNU_iMX\`make_uTaskerSerialBoot_GNU_iMX`) ensure that the floating point setting is correctfor the device; in this case it would be
`-mfloat-abi=hard -mfpu=fpv5-d16`
For an i.MX RT processor without double-precision FPU `-mfloat-abi=hard -mfpu=fpv5-sp-d16` is used instead.

The bat file can be edited to produce a dedicated archive file for the target in question by setting the variables

```
set iMX_RT=1020
set fallback=1     <---------- must be et to 1
```

The project can be built by executing `\Applications\uTaskerSerialBoot\GNU_iMX\Build_iMX_RT.bat` and results in an output binary file `\Applications\uTaskerBoot\GNU_iMX\`**`uTaskerSerialBoot_BM.bin`**

This output is however not used directly because the code is located to execute in internal RAM (ITC) but its image needs to be saved together with the “BM” boot loader's image in SPI flash. The “BM” boot loader also performs a validity check of the code and prepares the processor's RAM memory so that it can execute in an optimal manner, meaning that the image also needs an authentication header. This header is added during the bat file build where it is also combined with the “BM” boot loader's binary file in order to be loaded to the SPI flash. The resulting image is `\Applications\uTaskerSerialBoot\GNU_iMX\TaskerBootLoaderImage.bin`
A combined “BM” loader + Fall-back loader archive image is also created `\Applications\uTaskerBoot\GNU_iMX\`**`uTaskerFallbackLoaderImage_1020.bin`** image is also cretated.

Loading the combined “BM” boot loader and fall-back serial loader to the board

Instructions to using the NXP MCUBootUtility to load the binary image to the target can be found in the i.MX tutorial, chapter 4: https://www.utasker.com/docs/iMX/uTaskerV1.4_iMX.pdf

For production work complete images including loadable serial loader application can be used instead – see outputs of subsequent steps.

Building and loading the Serial Loader (loadable version) [*if there is no archive file for the target being developed for this step should be completed before building the application!*]

The same method is used to build the programmable serial loader as to build the Fall-back version (fixed and combined with the “BM” loader) with the exception that **the define `iMX_FALLBACK_LOADER` must be disabled!**

The bat file can be edited to produce a dedicated archive file for the target in question by setting the variables

```
set iMX_RT=1020
set fallback=0     <---------- mut be set to 0
```

Furthermore different loader strategies may be chosen when building it and possibly other configuration modifications that suit the working serial loader to be used (rather than the fall-back one)

When built the output `uTaskerSerialLoaderUpload.bin` results (*this is also generated when the fall-back version is built but is not valid for use in that case*).
This file can be loaded to the board using the fall-back loader, which will automatically operate when there is not yet a serial loader installed.
A combined "BM" loader + Fall-back loader + serial loader archive image is also created `\Applications\uTaskerBoot\GNU_iMX\`**`uTaskerBootComplete_1020.bin`** image is also cretated.

Building and loading the application

The Application code is located in the folder `\Applications\uTaskerV1.4\`

When this is built the resulting file is `\Applications\uTaskerV1.4\GNU_iMX\uTaskerV1.4_AES256_application.bin` which can be loaded with the installed serial loader.

The bat file \Applications\uTaskerV1.4\GNU_iMX\Build_iMX_RT.bat can be edited to produce a dedicated archive file for the target in question by setting the variable
```
set iMX_RT=1020
```

It is to be noted that the fall-back loader, the serial loader and the application are encrypted. The loaders automatically recognise the encryption and decrypt directly to internal RAM only when the code is used.
The loader concept also supports unencrypted files but these are depreciated in the uTasker project since they have no benefits (neither in terms of code size, performance or complexity to build).
The application build also outputs a file `\Applications\uTaskerV1.4\GNU_iMX\uTaskerV1.4_XiP.bin` which can also be loaded and executes directly in QSPI flash. It is not encrypted and is a reference to show that code can also run directly from flash, which would usually only be of relevance when the code size exceeds the internal RAM size.

`uTaskerCompleteImage_1020.bin` and `uTaskerCompleteImage_1020.hex` outputs contain "BM" loader + "Fall-back" serial loader + serial loader + application and are useful for production programming to avoid the need to load the various images individually

Note that some bat files contain additional variables:

```
set SECRET_KEY="aes256 secret key"
set VECTOR="initial vector"
set MAGIC=234
set AUTHENTICATION=a748b6531124
```

- **SecretKey** is the AES256 (AES128 key is also derived from it when on-the-fly XiP is used) used to encrypt the code.
- **Vector** is the AES256 initial vector (AES128 nonce is also derived from it when on-the-fly XiP is used)
- **Magic** is the project/product's magic number which is used to ensure that all firmware files that are received are intended for this product
- **Authentication** is an embedded key that is used to authenticate all firmware files that are received

These variables are used to control the tools that generate the versions for uploading purposes and should match with the values in the "BM" boot loader and serial loader code:

```
#define PROJECT_APPLICATION_MAGIC_NUMBER     0x0234   // first nibble should be 0 - the magic number is a simple check in the new code's header to verify that it is intended for our product
#define APPLICATION_AUTHENTICATION_KEY       {0xa7, 0x48, 0xb6, 0x53, 0x11, 0x24} // the new code's CRC is calculated and then this added in order to detect both code errors and code not was not processed with our authentication key
#define APPLICATION_AES256_SECRET_KEY        "aes256 secret key" // the secret key used to encrypt the code content (before adding its header) - this, and the initial vector, should be kept secret in order to ensure security (up to 32 bytes in length)
#define APPLICATION_AES256_INITIAL_VECTOR    "initial vector" // the initial vector used when encrypting the code (up to 16 bytes in length)
```

Application Options

The µTasker loader concept can essentially be used with any application and this application can either execute directly from QSPI flash or be copied to internal RAM for execution. It can also be encrypted (ad decrypted by the loader to internal RAM where it securely runs.

The following options are available, which are communicated to the loader via the application header's magic number. This is added to the application by using the µTasker utility `uTaskerConvert.exe`

Eg.

```
uTaskerConvert.exe uTaskerV1.4_BM.bin uTaskerV1.4_application.bin -0x1234 -a748b6531124
```

The magic number is a 16 bit value whose first nibble indicates its format:

```
#define BOOT_LOADER_TYPE_PLAIN_XiP_RESET_VECTOR  0x0000                  // execute in QSPI flash (execute in place) starting with reset vector
#define BOOT_LOADER_TYPE_PLAIN_RAM_EXECUTION     0x1000                  // copy plain code to ITM and execute there
#define BOOT_LOADER_TYPE_PLAIN_XiP_CONFIG_TABLE  0x2000                  // execute in QSPI flash (execute in place) starting with configuration table
#define BOOT_LOADER_TYPE_PLAIN_SDRAM_EXECUTION   0x3000                  // copy plain code to SDRAM and execute there
#define BOOT_LOADER_TYPE_AES256_SDRAM_EXECUTION  0x4000                  // decrypt AES256 encrypted code to SDRAM and execute there
#define BOOT_LOADER_TYPE_AES128_XiP_RESET_VECTOR 0x5000                  // execute in QSPI flash (execute in place) starting with reset vector using on-the-fly decryption
#define BOOT_LOADER_TYPE_AES128_XiP_CONFIG_TABLE 0x6000                  // execute in QSPI flash (execute in place) starting with configuration table using on-the-fly decryption
#define BOOT_LOADER_TYPE_AES256_RAM_EXECUTION    0x9000                  // decrypt AES256 encrypted code to ITM and execute there
```

Therefore
`0x1234` is a magic number of `0x0234` which should be copied to, and executed in internal RAM (it is linked to the address `0x300`)

`0x9234` is a magic number of `0x0234` which should be copied to, and executed in internal RAM (it is linked to the address `0x300`). The image is additionally AES256 encrypted and the boot loader uses the project's AES26 secret key and initial vector value to decrypt it during the copy

`0x0234` is a magic number of `0x0234` which is executed directly from QSPI flash. The application should be linked to `0x60020108` (the exact value may change with loader type and QSPI flash used) and starts with its reset vector. No flash configuration block is used.

`0x2234` is a magic number of `0x0234` which is executed directly from QSPI flash. The application should be linked to `0x60020108` (the exact value may change with loader type and QSPI flash used) and starts with flash configuration block. The flash configuration block is interpreted in order to find the vector location containing the applications reset vector.

`0x3234` is a magic number of `0x0234` which should be copied to, and executed in external SDRAM (it is linked to the address of the external SDRAM). It can locate its interrupt vectors either to the start of SDRAM or else to interal RAM.

`0x`**`4`**`234` is a magic number of `0x0234` which should be decrypted and copied to, and executed in external SDRAM (it is linked to the address of the external). It can locate its interrupt vectors either to the start of SDRAM or else to interal RAM.

`0x`**`5`**`234` is a magic number of `0x0234` which is AES128 encrypted and executes directly from QSPI flash (using on-the-fly decryption). The application should be linked to `0x60020108` (the exact value may change with loader type and QSPI flash used) and starts with its reset vector. No flash configuration block is used.

`0x`**`6`**`234` is a magic number of `0x0234` which is AES128 encrypted and executes directly from QSPI flash (using on-the-fly decryption). The application should be linked to `0x60020108` (the exact value may change with loader type and QSPI flash used) and starts with flash configuration block. The flash configuration block is interpreted in order to find the vector location containing the applications reset vector.

Valid as from 30.4.2020 when the three-phase loader concept was released

This version 28th August2020