



µTasker Document

Using the µTasker project with i.MX RT in IAR

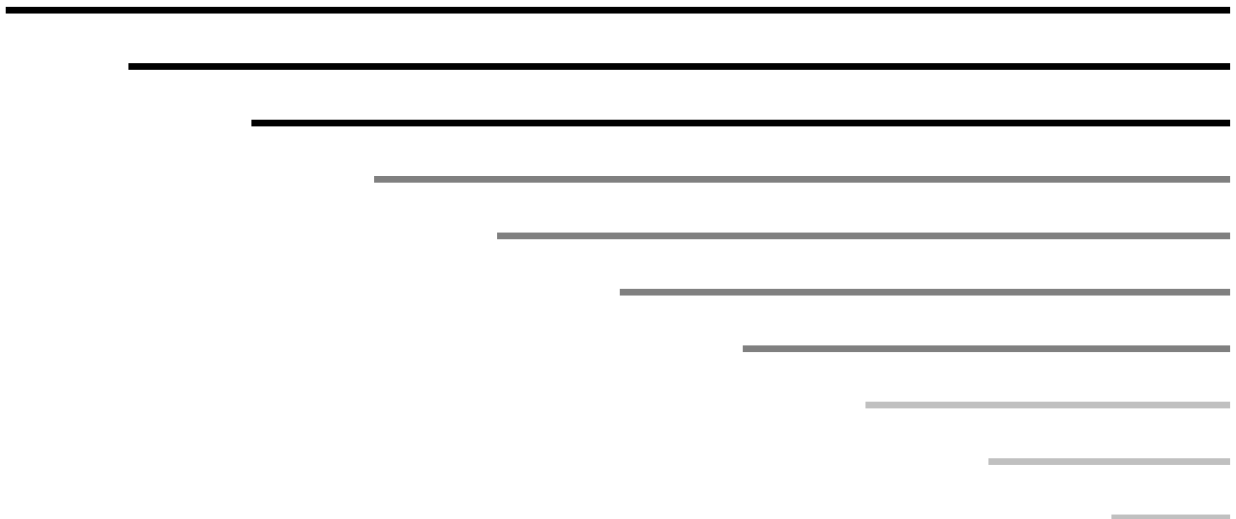


Table of Contents

1. Introduction.....	3
2. Building the “Bare-Minimum (BM)” Boot Loader.....	4
3. Building the “Fall-Back” Serial Loader.....	5
4. Loading the combined “BM” boot loader and fall-back serial loader to the board.....	6
5. Building and Loading the Serial Loader.....	6
6. Debugging the Serial Loader.....	8
7. Building and Loading the Application.....	9
8. Additional Details about the Boot Loader Concept.....	11
9. Using the Boot Loader with Foreign Applications.....	13
10. Pre-Configuring FlexRAM for XiP Application usage.....	21
11. Enabling SDRAM Support if required by the Application or for SDRAM Code Execution	23
12. Conclusion.....	24

1. Introduction

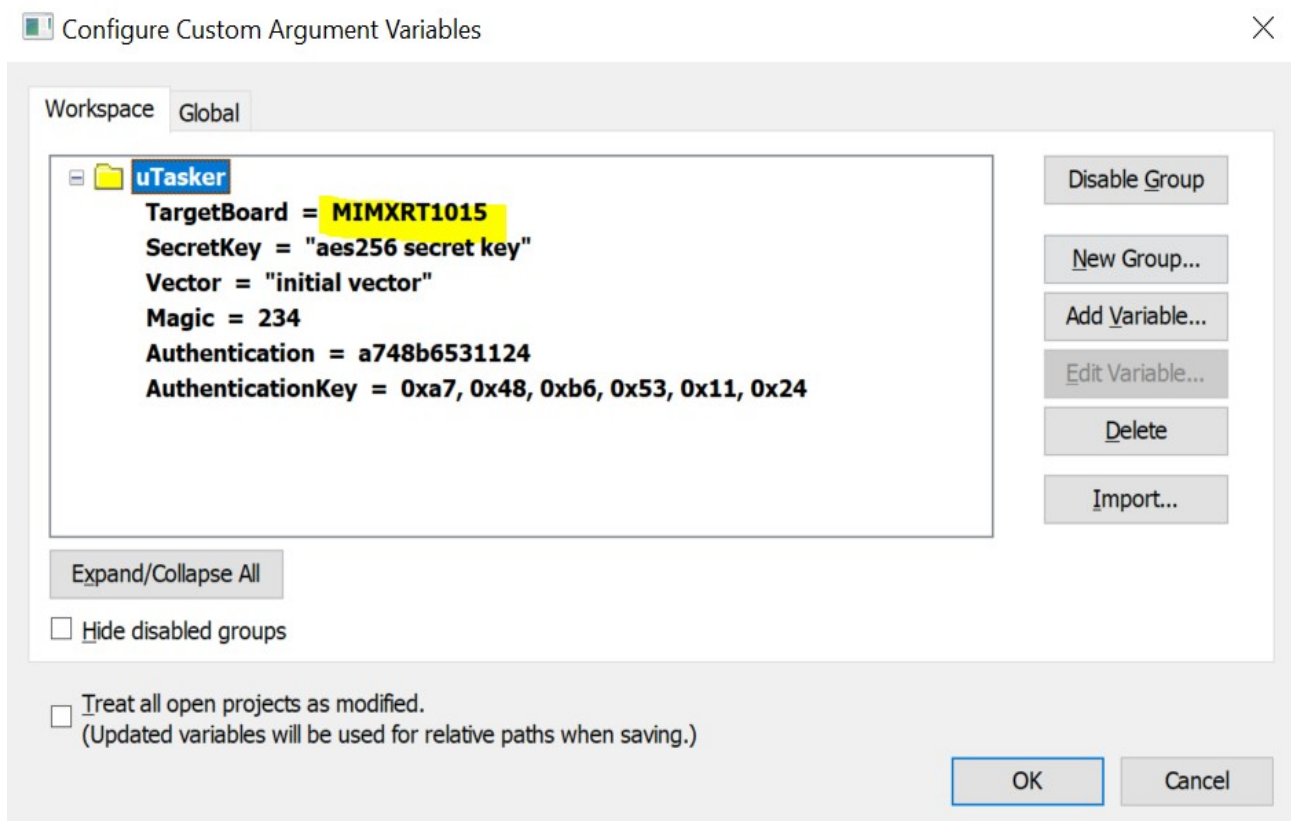
This guide uses the MIMXRT1015 as reference setup but all other boards/processors are effectively equivalent by using their name instead.

It contains a step-by-step guide to configuring the project and building the loader concept, as well as some tips on solving problems if encountered.

The use of non-uTasker application are furthermore described, which can be simply run in QSPI-flash, on-the-fly decoded QSPI Flash, SDRAM or internal RAM.

From 20.8.2020 the three IAR targets are combined in the IAR workspace \ Applications\ uTaskerV1.4\ IAR8_iMX_RT\ uTaskerV1.4.eww and therefore this is the only project file that needs to be opened.

The `TargetBoard` is set globally so that it is valid for all of the projects ("Bare minimum" loader, "Fall-back" serial loader, "Serial" loader and "Application"). It can be configured in the Tools | Configure Custom Argument Variable.. setting:



- **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

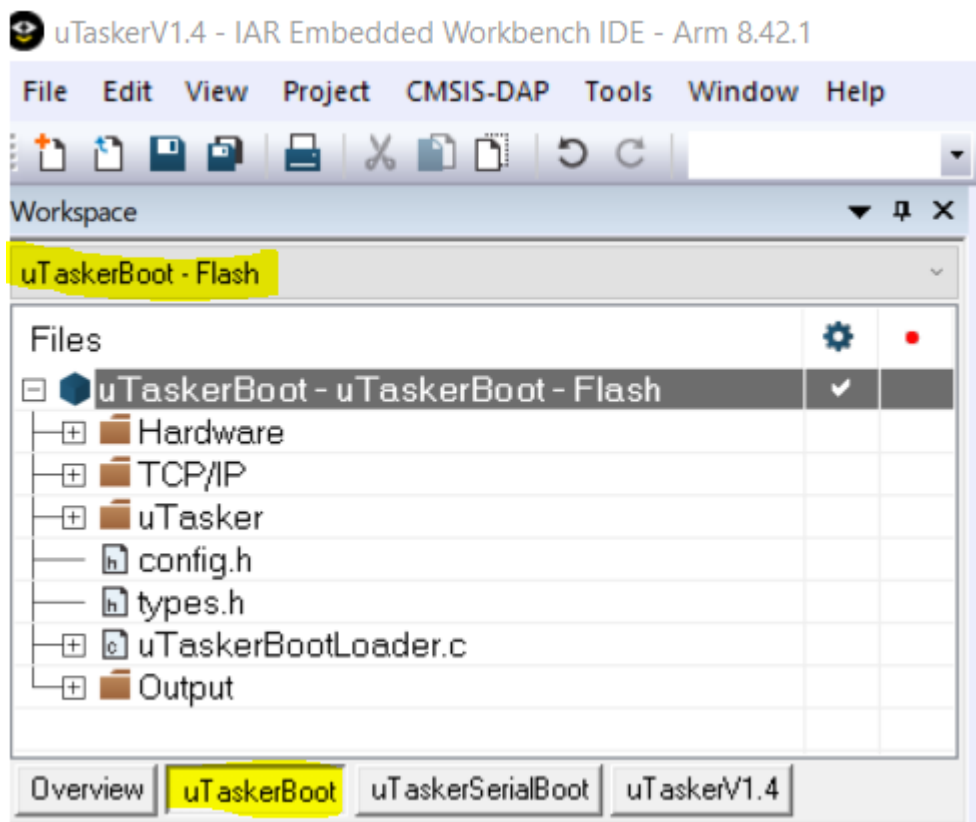
- **Authentication** is an embedded key that is used to authenticate all firmware files that are received (***AuthenticationKey** is the same value entered as a C-code hex byte array*)

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)
```

2. Building the “Bare-Minimum (BM)” Boot Loader

Choose the **uTaskerBoot** project and the **uTaskerBoot – Flash** target:



In the project's options ensure that the general options are set for the correct device; in this case it would be

NXP MIMXRT1015xxx5A

The project can be built and results in an output binary file

\Applications\uTaskerBoot\IAR8_iMX_RT\Objects\

uTaskerBoot_MIMXRT1015.bin

which is designed to be located in SPI flash and boot the processor.

The object can however not work alone and requires the fall-back serial loader to be combined with it, which is detailed in the next section.

3. Building the “Fall-Back” Serial Loader

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 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 (eg. IMX RT 1011) `-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=MIMXRT1020
```

```
set fallback=1 <----- must be set 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) and 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_MIMXRT1020.bin image is also created.

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

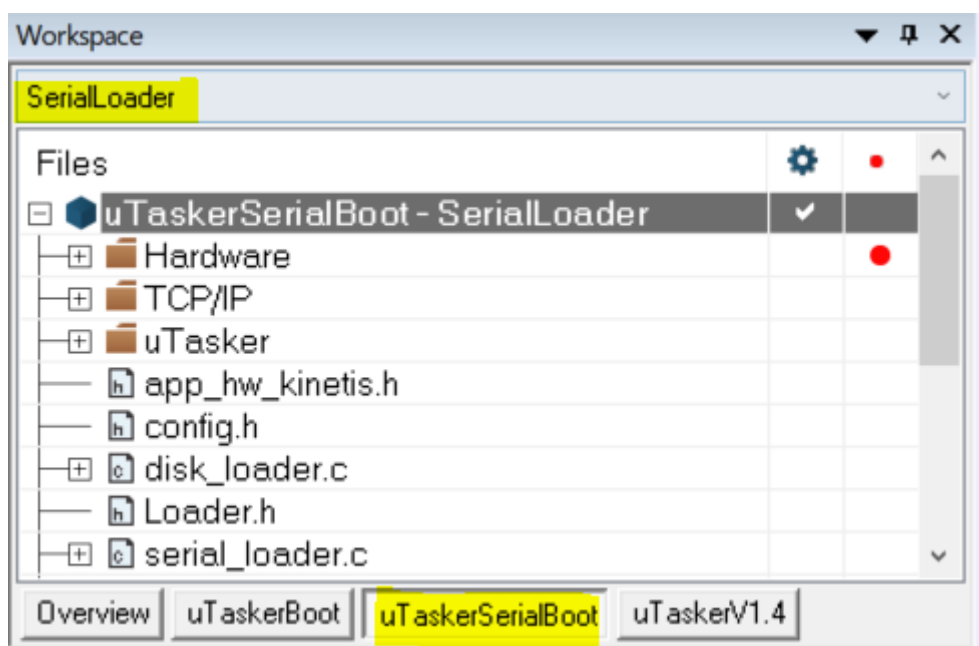
Instructions to using the NXP MCUBootUtility to load the binary image to the target can be found in the i.MX tutorial, chapter 4:

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.

5. Building and Loading the Serial Loader

Choose the **uTaskerSerialBoot** project and the **uTaskerBoot – SerialLoader** target:



In the project's options ensure that the general options are set for the correct device; in this case it would be

NXP MIMXRT1015xxx5A

The project can be built and results in an output binary file

`\Applications\uTaskerSerialBoot\IAR8_iMX\Objects\
uTaskerBootLoaderImage_MIMXRT1015.bin`

This contains the “Bare-minimum” loader, plus the Fall-back loader plus an encrypted version of the serial loader that can be loaded to the board.

`\Applications\uTaskerBoot\GNU_iMX\
uTaskerBootComplete_MIMXRT1020.bin` image is also created and can be loaded using the same technique as discussed in chapter 4.

This target build also result in the output file `\Applications\
uTaskerSerialBoot\IAR8_iMX\Objects\
uTaskerSerialLoaderUpload_MIMXRT1015.bin`

This image can be uploaded to the board using the “Fall-back” serial loader in order to update the serial loader used in the product.

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

6. Debugging the Serial Loader

The serial loader code is executed in RAM and can also be debugged by connecting to the target ("Debug without Downloading"). Ignore any warnings when connecting and click "No" when asked whether you want to continue with the "Run to" command. It is best to set a breakpoint.

The program starts at the routine

```
static void disable_watchdog(void)
```

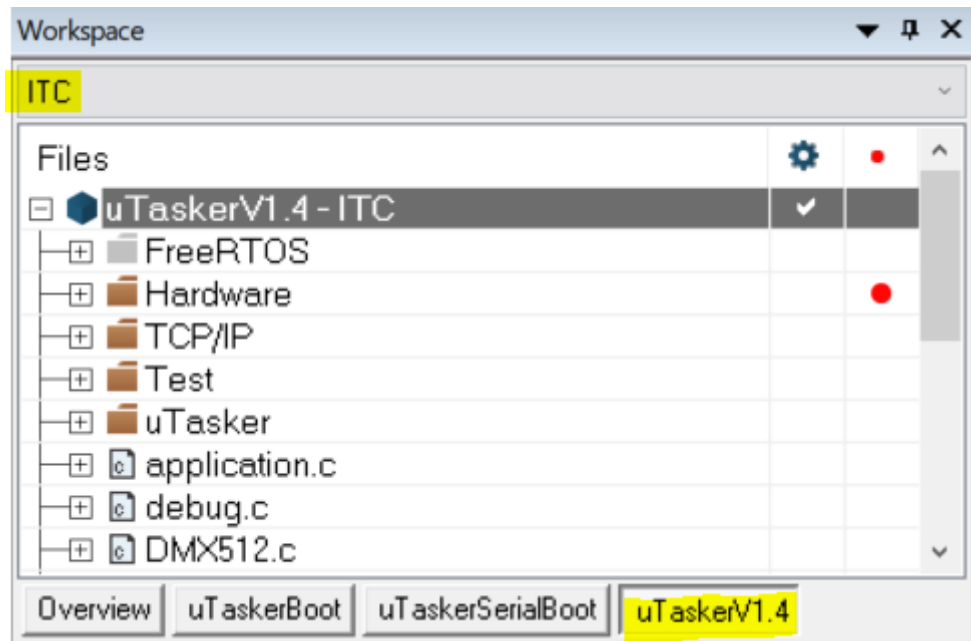
and can be executed and debugged normally from this point on.

It is to be remembered that it is in fact the "BM" boot loader that has started the operation and loaded the Serial Loader's code to RAM. It will be seen that the serial loader's stack pointer has been set by the "BM" boot loader to be close to the top of available data ram (DTC), whereby the FlexRAM in the i.MX RT processor was also configured to be suitable for the optimal allocation of tightly coupled memory between the program code and its data usage.

7. Building and Loading the Application

All previous steps should have been completed in the correct order before building the application!

Choose the **uTaskerV1.4** project and the **uTaskerV1.4 – ITC** target:



In the project's options ensure that the general options are set for the correct device;
in this case it would be

NXP MIMXRT1015xxx5A

The project can be built and results in an output binary file

`\Applications\uTaskerV1.4\IAR8_iMX_RT\Objects\
uTaskerCompleteImage_MIMXRT1015.bin`

This contains the “Bare-minimum” loader, plus the “fall-back” loader, plus the “serial
“loader plus an encrypted version of the application that can be loaded to the board.

The combined image can be loaded using the same technique as discussed in chapter 4.

This target build also result in the output file `\Applications\uTaskerV1.4\
IAR8_iMX_RT\Objects\uTaskerV1.4_AES256_MIMXRT1015.bin`

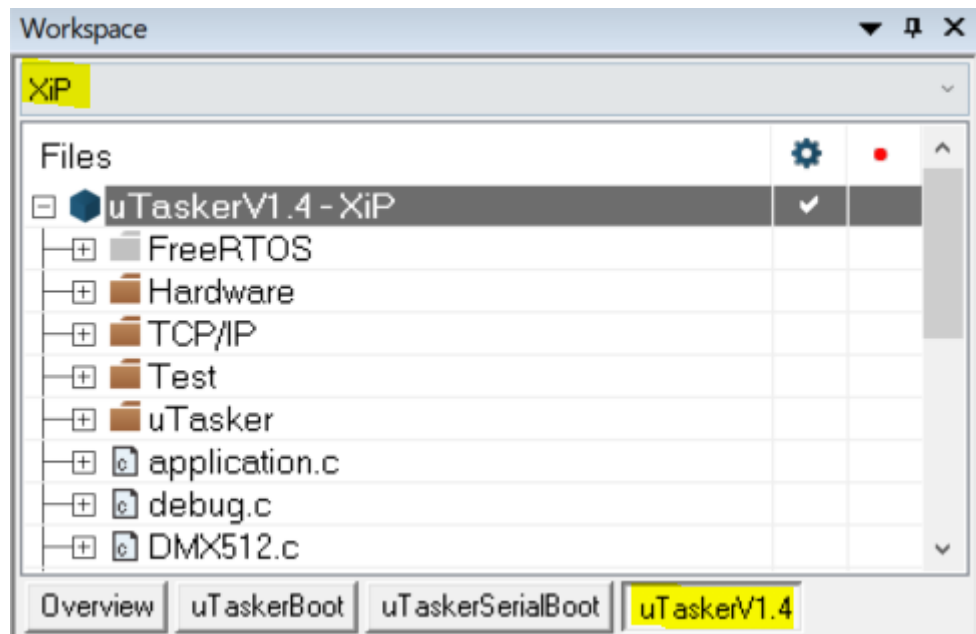
This image can be uploaded to the board using the serial loader in order to update the application used in the product.

Note that the ITC target runs in internal RAM and is generally used in encrypted form whereby the application is stored in AES256 encrypted form in flash. The encrypted

form has no disadvantage over the plain-code form and both run at the same speed once copied to internal instruction RAM.

In the case of large applications that can't be run from internal RAM the XiP target can alternatively be used:

Choose the **uTaskerV1.4** project and the **uTaskerV1.4 – XiP** target:



In the project's options ensure that the general options are set for the correct device;

in this case it would be

NXP MIMXRT1015xxx5A

The project can be built and results in two output binary files

`\Applications\uTaskerV1.4\IAR8_iMX_RT\Objects\
uTaskerV1.4_XiP_MIMXRT1015.bin`

and

`\Applications\uTaskerV1.4\IAR8_iMX_RT\Objects\
uTaskerV1.4_XiP_AES128_MIMXRT1015.bin`

Both of these application can be loaded with the serial loader in order to update the application used in the product.

The plain-code form runs directly in SPI flash (XiP mode) and the AES128 encrypted form will cause the serial loader to configure for “On-The-Fly” description so that the encrypted application in SPI flash can directly run from there.

8. Additional Details about the Boot Loader Concept

The μ Tasker 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 (and 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 μ Tasker 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 ITC
// 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 ITC 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 `0x60020400` (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 `0x60020400` (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 internal RAM.

0x4234 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 internal RAM.

0x5234 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 0x60020400 (the exact value may change with loader type and QSPI flash used) and starts with its reset vector. No flash configuration block is used.

0x6234 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 0x60020400 (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.

9. Using the Boot Loader with Foreign Applications

Although it is hoped that also the uTasker application will be found to be a more advanced solution than the traditional semiconductor manufacturer's framework (SDK) the uTasker loader can be used with applications from any source.

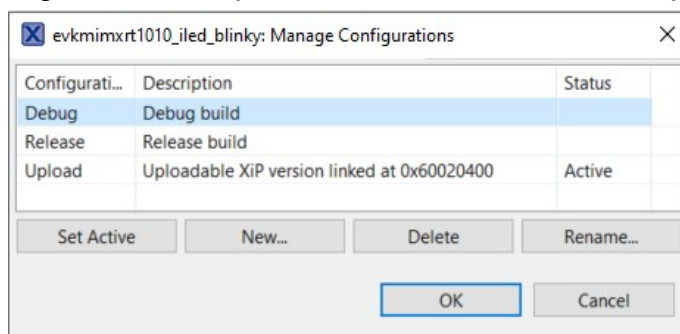
Important – if SDK code requires SDRAM access please also consult the following chapter detailing enabling SDRAM configuration in the boot loader.

This section explains how an existing MCUXpresso project that is running in QSPI flash (XiP) can be very simply used as an upload file to the uTasker loader in either plain code (unencrypted) or on-the-fly encrypted form with almost no development effort.

1. The easiest method of allowing the original code and an uploadable version to be managed in MCUXpresso is to create a new target called "Upload". This is simple to do by making a copy of the original target – eg. "Debug". In the menu "Project | Build Configurations | Manage..." create the new target as a copy of the original one.

Here the new target is seen with a description explaining how it is linked and is set as the active configuration:

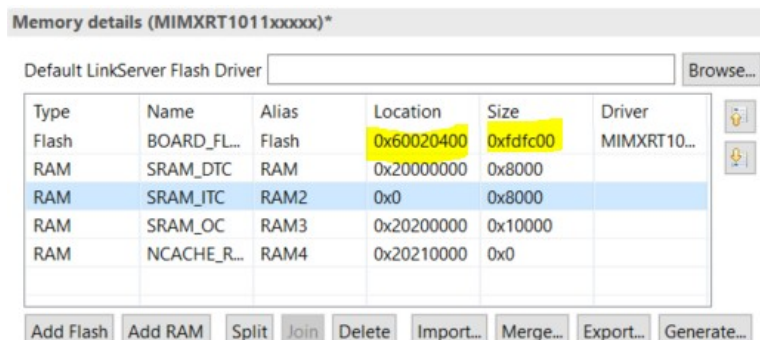
2. This target will initially build identically to the original one and can now be modified to generate an uploadable version for the same project.



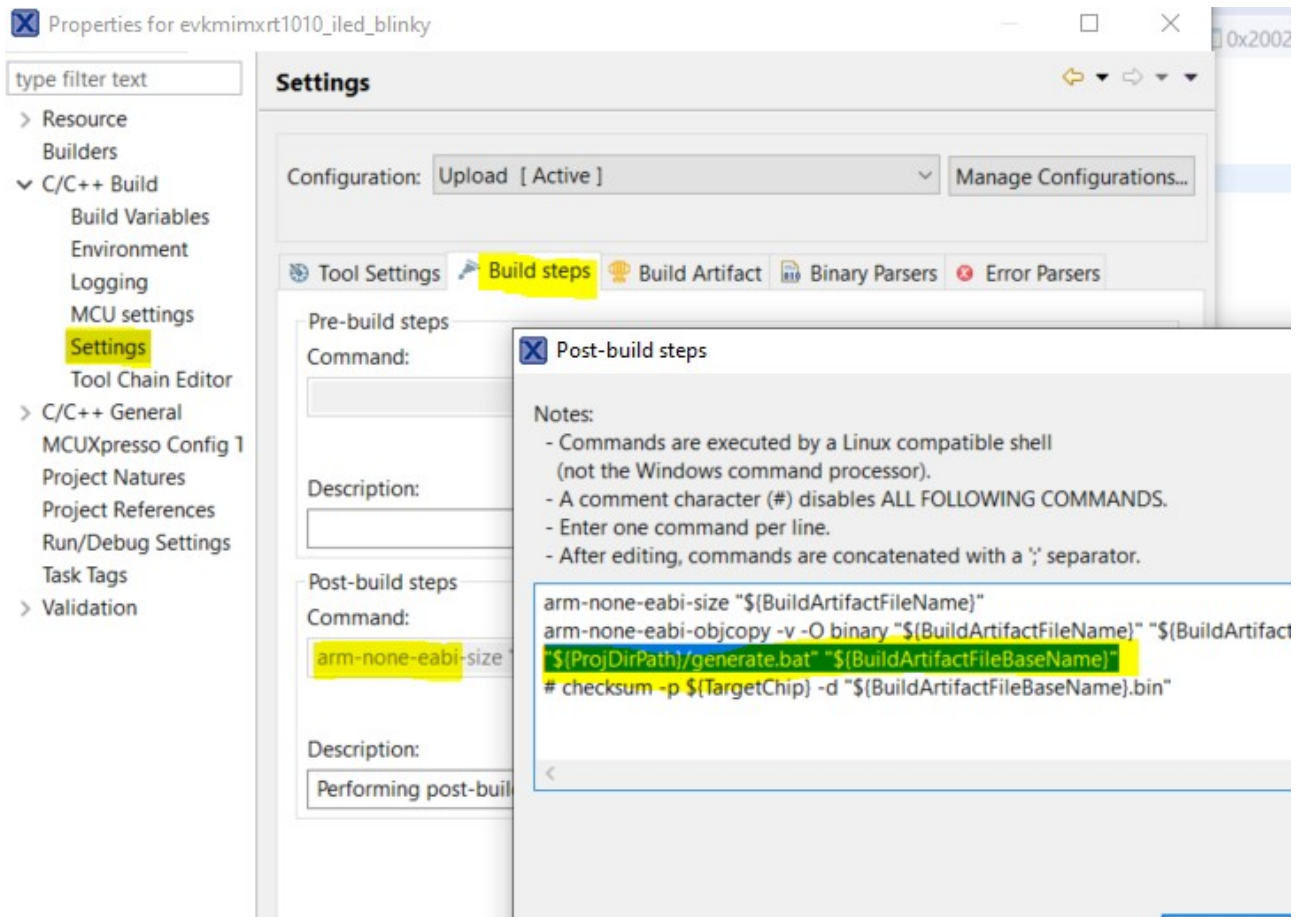
The first modification is to adjust the memory map so that the code is linked to run from the uTasker loader's application address, which is usually 0x60020400 (0x70020400 for i.MX RT 1064 running in internal QSPI flash).

This is performed in the target properties C/C++ Build → MC Settings:

Here it is seen that the normal program start address (0x60000000) had been changed to 0x60020400 *and the size of the flash is reduced accordingly.*



Note that the offset of 0x400 ensures both library and interrupt vector compatibility, when the vectors are in QSPI flash. A value of 0x200 is also possible with the i.MX RT 1011 since it has less vectors, but generally a fixed layout is used to avoid any potential confusion.



3. In the Post-build steps option the binary output is enabled and a bat. file call added:

The line `"${ProjDirPath}/generate.bat" "${BuildArtifactFileName}.bin"` is new and will cause the post build step to be executed each time the target is successfully build. *This bat file is explained later.*

4. A bat file named "Generate.bat" can be created in the root directory of the work space (eg. Where the IDE's .cproject and .project files are located) with the following content:

```
SET PATH=%PATH%;C:\Repositories\uTasker-GIT-Kinetis\Tools

rem - select the target being built for in order to automate
combining production file

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

rem - generate uploadable version (plain code)
uTaskerConvert.exe %1.bin %1_XiP.bin +boot_header.txt -
0x0%MAGIC% -%AUTHENTICATION%

rem - encrypt for OTF XiP operation
rem - used by OTFAD
uTaskerConvert.exe %1.bin %1_OTFAD.bin E=128-60020400 $
%SECRET_KEY% %VECTOR%
uTaskerConvert.exe %1_OTFAD.bin %1_XiP_OTFAD.bin
+boot_header.txt -0x5%MAGIC% -%AUTHENTICATION%
del %1_OTFAD.bin

rem - used by BEE
uTaskerConvert.exe %1.bin %1_BEE.bin E=128B-60020400 $
%SECRET_KEY% %VECTOR%
uTaskerConvert.exe %1_BEE.bin %1_XiP_BEE.bin +boot_header.txt -
0x5%MAGIC% -%AUTHENTICATION%
del %1_BEE.bin
```

- The path to the uTasker tools directory is set as a path variable to match its location on the PC
- The variables (SECRET_KEY, VECTOR etc.) should match the ones used by the uTasker loader configuration
- Note that the magic number's first digit is set to 0 in this case when the plain code output is converted since the content starts with a reset vector and not with a boot configuration, in which case it would be 2 instead. A non-encrypted file called XXXX_XiP.bin is created which is suitable for uploading to the board via the uTasker serial loader, where XXX is the name fo the MCUXpresso project.
- Two encrypted output files are created – one which can be used by processors with OTFAD (like the i.MX RT 1011) and one that can be used by processors with BEE (most others): These output files are called XXXX_XiP_OTFAD.bin and

XXXX_XiP_BEE.bin.

e. Note that the magic number's first digit is set to 5 in the case of the 'on-th-fly' encryption versions which signals that the content starts with the reset vector and not a boot configuration, in which case it would be 6 instead.

5. Note that the bat file uses a boot configuration header file called `boot_header.txt`, which should also be added to the same directory. This header is added before the content of the application XiP code in order to ensure that it is aligned on a boundary that is both suitable for AES128 decryption and also for interrupt vectors to remain being located in QSPI flash (requiring a 1k byte alignment in order to be able to use all possible vectors).

The content of this file can be:

```
// We add 760 bytes of padding between the header and the start
// of code in order to
// align the code on a 1k (0x400) byte boundary (ensures on-
// the-fly decryption compatibility,
// library compatibility and also allows interrupt vectors to
// remain in code)

02f8                // first two bytes specify the length
ffffffffffffffff
ffffffffffffffff    // padding should be 0xff by default
ffffffffffffffff    // and other content is reserved for
future
                    // control of additional configurations
ffffffffffffffff
ffffffffffffffff
ffffffffffffffff
ffffffffffffffff

ffffffffffffffff
ffffffffffffffff
ffffffffffffffff
ffffffffffffffff

ffffffffffffffff
ffffffffffffffff
ffffffffffffffff
ffffffffffffffff

ffffffffffffffff
ffffffffffffffff
ffffffffffffffff
ffffffffffffffff
```


ffffffffffffffffffff
ffffffffffffffffffff
ffffffffffffffffffff
ffffffffffffffffffff

ffffffffffffffffffff
ffffffffffffffffffff
ffffffffffffffffffff
ffffffffffffffffffff

ffffffffffffffffffff
ffffffffffffffffffff
ffffffffffffffffffff
ffffffffffffffffffff

ffffffffffffffffffff
ffffffffffffffffffff
ffffffffffffffffffff
ffffffffffffffffffff

ffffffffffffffffffff
ffffffffffffffffffff
ffffffffffffffffffff
ffffffffffffffffffff

ffffffffffffffffffff
ffffffffffffffffffff
ffffffffffffffffffff
ffffffffffffffffffff

ffffffffffffffffffff
ffffffffffffffffffff
ffffffffffffffffffff
ffffffffffffffffffff

ffffffffffffffffffff
ffffffffffffffffffff
ffffffffffffffffffff
ffffffffffffffffffff

ffffffffffffffffffff
ffffffffffffffffffff
ffffffffffffffffffff
ffffffffffffffffffff

ffffffffffffffffffff
ffffffffffffffffffff
ffffffffffffffffffff
ffffffffffffffffffff

ffffffffffffffffffff

```
ffffffffffffffffffff
ffffffffffffffffffff
ffffffffffffffffffff

ffffffffffffffffffff
ffffffffffffffffffff
ffffffffffffffffffff
ffffffffffffffffffff

ffffffffffffffffffff
ffffffffffffffffffff
ffffffffffffffffffff
ffffffffffffffffffff

ffffffffffffffffffff
ffffffffffffffffffff
ffffffffffffffffffff
ffffffffffffffffffff

ffffffffffffffffffff
ffffffffffffffffffff
ffffffffffffffffffff
ffffffffffffffffffff





ffffffffffffffffffff
ffffffffffffffffffff
ffffffffffffffffffff
ffffffffffffffffffff

ffffffffffffffffffff
ffffffffffffffffffff
ffffffffffffffffffff
ffffffffffffffffffff

ffffffffffffffffffff
ffffffffffffffffffff
ffffffffffffffffffff
ffffffffffffffffffff
```

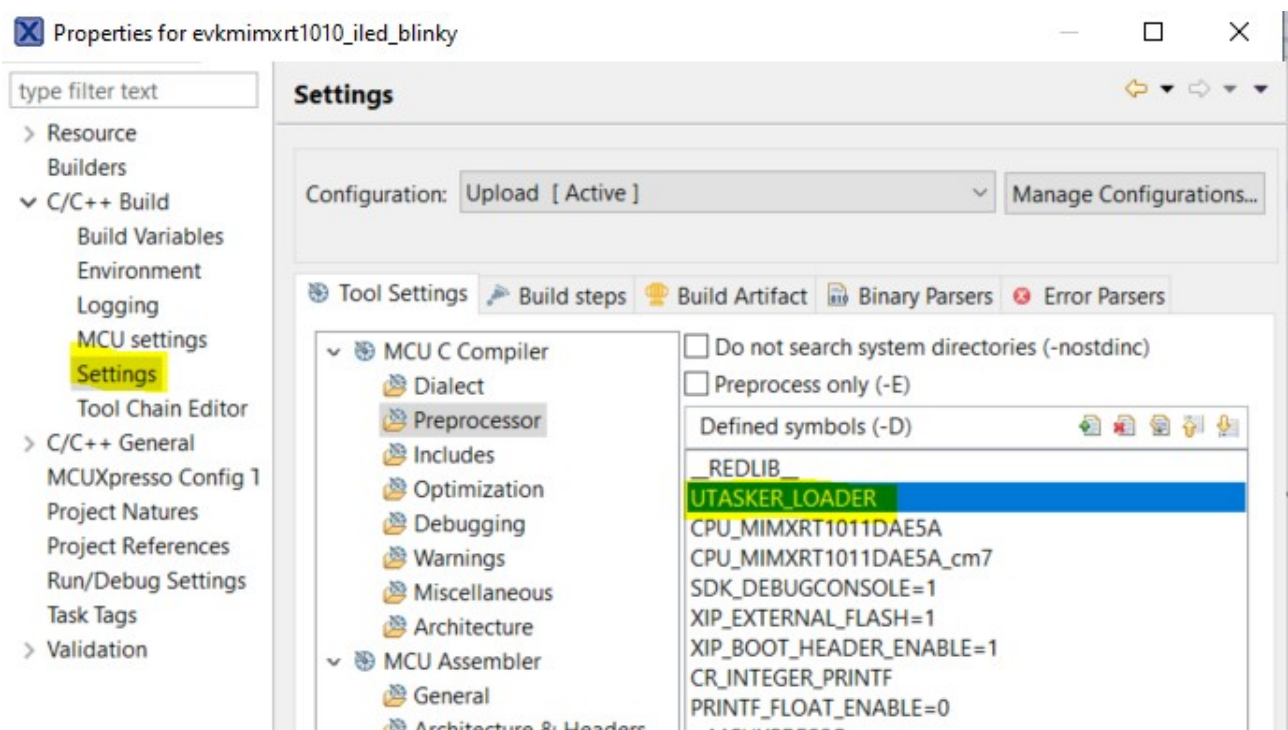
In the MCUXpresso workspace output target directory the following binary files would

be found in the case of building the SDK's blinky project for the i.MX RT 1015:

 evkmimxrt1015_iled_blinky.bin	28/08/2020 20:48	BIN File	13 KB
 evkmimxrt1015_iled_blinky_XiP.bin	28/08/2020 20:48	BIN File	13 KB
 evkmimxrt1015_iled_blinky_XiP_BEE.bin	28/08/2020 20:48	BIN File	13 KB
 evkmimxrt1015_iled_blinky_XiP_OTFAD.bin	28/08/2020 20:48	BIN File	13 KB

The first is the build's binary output, which can not be used in this form, and the following ones are suitable for uploading to the board via the uTasker serial loader as either plain-code or on-the fly encrypted forms. The serial loader recognises the content and automatically configures the on-the-fly decryption modules accordingly (as well as securely managing the AES128 keys) without any further effort on behalf of the developer.

An optional step to allow interrupt vectors to run from ITC, if not the present case, can be performed by adding a pre-processor define called `UTASKER_LOADER` to the C/C++ Build Pre-processor settings:



which will allow some code changes to be made that are only valid when this particular target is built.

In the project's system initialisation – eg. `system_MIMXRT1015.c` - code is added that copies the vectors to RAM when the board starts and sets the vector offset register accordingly:

```
uint32_t SystemCoreClock = DEFAULT_SYSTEM_CLOCK;

/* -----
-- SystemInit()
----- */

void SystemInit (void) {
#if ((__FPU_PRESENT == 1) && (__FPU_USED == 1))
    SCB->CPACR |= ((3UL << 10*2) | (3UL << 11*2));    /* set CP10, CP11 Full Access */
#endif /* ((__FPU_PRESENT == 1) && (__FPU_USED == 1)) */

#if defined UTASKER_LOADER
{
    extern uint32_t g_pfnVectors[]; // Vector table defined in startup code
    int i = 0;
    volatile uint32_t *ptrRam;
    uint32_t *ptrVectors = g_pfnVectors;
    SCB->VTOR = 0; // address of tightly coupled instruction RAM
    ptrRam = (volatile void *)SCB->VTOR;
    while (i++ < 0x300/sizeof(unsigned long)) {
        *ptrRam++ = *ptrVectors++; // copy the vectors from flash to RAM
    }
}
#elif defined(_MCUXPRESSO)
    extern uint32_t g_pfnVectors[]; // Vector table defined in startup code
    SCB->VTOR = (uint32_t)g_pfnVectors;
#endif
}
```

This makes interrupt execution faster than when the vectors are left in QSPI flash and the technique can be used generally too and not be made dependent on the “Upload” configuration. It does also require the SRAM_ITC setting to be adjusted to make space for these as follows:

Memory details (MIMXRT1011xxxxx)*

Default LinkServer Flash Driver

Type	Name	Alias	Location	Size	Driver
Flash	BOARD_FL...	Flash	0x60020400	0xfdfc00	MIMXRT10...
RAM	SRAM_DTC	RAM	0x20000000	0x8000	
RAM	SRAM_ITC	RAM2	0x300	0x7d00	
RAM	SRAM_OC	RAM3	0x20200000	0x10000	
RAM	NCACHE_R...	RAM4	0x20210000	0x0	

10. Pre-Configuring FlexRAM for XiP Application usage

This section shows how to configure alternative FlexRAM configurations for XiP application usage that are performed by the boot loader without requiring application code level configuration or eFuse settings:

Although application start-up code can configure alternative FlexRAM configurations this may prove unnecessarily complicated, involving assembler and needing to carefully understand the technique involved.

When working with the μ Tasker boot loader this becomes child's play since the application developer can simply define the layout that the application would like to be started with and it will be *pre-configured* for it, thus requiring no special code in the application.

The application's reset vector can even directly use FlexRAM areas that would normally not be possible without configuring with eFuses (which is a one-shot process that cannot be reverted and so preferably avoided).

Normally (without any special configuration) the XiP application is started by the μ Tasker boot loader with the FlexRAM configured in its default state. For example, an i.MX RT 106x would have 128k DTC, 128k ITC and 256k OCR (plus a further 512k fixed general purpose OCR2 RAM).

If the application would prefer – *for example* - to have 96k DTC, 256k ITC and 160k OCR (whereby the configurable bank size is always in units of 32k) the following setting change in the `boot_header.txt` file configuration (see the previous chapter for its details) will instruct the μ Tasker loader to prepare it.

First consider the standard header file content:

```
02f8                // first two bytes specify the length
ffffffffffffffff    // padding should be 0xff by default and
other content is reserved for future control of additional
configurations
ffffffffffffffff
ffffffffffffffff
```

followed by further ff padding bytes

....

This has no instructions and serves purely as padding to ensure the application alignment is correct on a suitable address boundary.

In comparison, this one has the desired FlexRAM configuration:

```
02f8                // first two bytes specify the length
030805fffffff      // specify DTC/ITC and OCR bank sizes to be
pre-configured for the application (when not ff)
fffffffffffffffffff
fffffffffffffffffff
```

followed by further ff padding bytes

....

3 banks to be assigned to DTC, 8 to ITC and 5 to OCR

The μ Tasker boot loader will perform the FlexRam configuration according to the specified bank quantities. In addition, assuming DTC hasn't been set to 0, it will ensure that the boot mail box is located in the highest DTC bank so that the application can communicate with the loader via the highest DTC memory locations. *Note that the mail box area also contains some useful information and counters maintained by the loader, such as the last reset cause, how many times the board has been restarted due to watchdog resets and general resets.*

No further application effort is required, making this a very simple, fast and painless way to achieve an XiP based application's preferred FlexRAM layout.

11. Enabling SDRAM Support if required by the Application or for SDRAM Code Execution

If the application is run from SDRAM the boot loader must include support for this and also configure the SDRAM operation.

If the application requires SDRAM access the application can either manually configure the SDRAM in its start-up code before accessing it for the first time or else the boot loader can generally configure it. When the boot loader configures it it does so by adding a DCD (Device Configuration Data) table to its configuration code which defines the registers that the ROM LOADER should write to before the loader is started. The setup is retained when subsequent applications are started so that they can benefit from pre-configured SDRAM operation too.

Note that SDK application users requiring SDRAM access will find it simplest to enable the configuration in the loader and therefore enabling

```
#define BOOT_LOADER_SUPPORTS_SDRAM    // enable when the boot loader is to configure  
SDRAM for subsequent application use (or when application runs in SDRAM)
```

for the target HW in the „uTaskerBoot“ project is recommended for both simplicity and to ensure that unconfigured SDRAM doesn't otherwise cause hard faults when access is attempted by the application.

12. Conclusion

This document has detailed the steps necessary to build the μ Tasker Boot Loader and an application from IAR Embedded Workbench.