Embedding it better...

µTasker Document

Using the µTasker project with i.MX RT in MCUXpresso

Table of Contents

1. Introduction

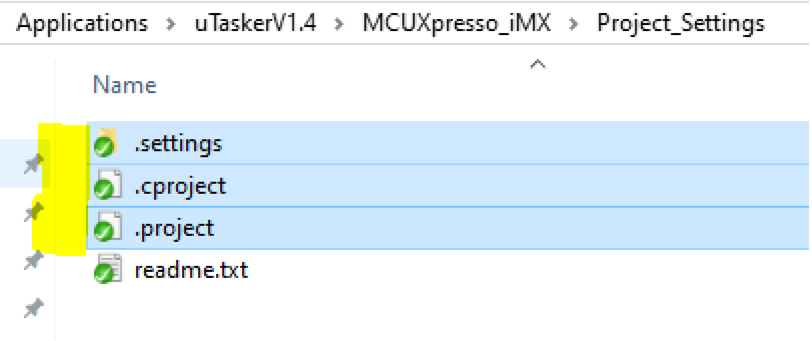
This guide uses the MIMXRT1015 as reference setup but all other boards/processors are effectively equivalent by using their name instead.

It contains a step-by-step guide to importing 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.

2. Importing and Building the Project in MCUXpresso

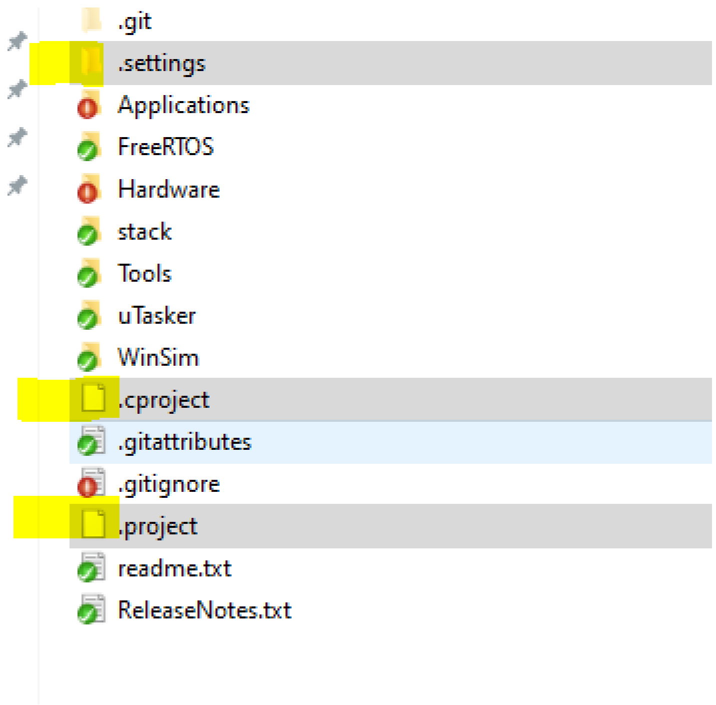
1. Before importing the project into MCUXpresso navigate to the MCUXpresso project settings folder:



Applications > uTaskerV1.4 > MCUXpresso_iMX > Project_Settings

Name
.settings
.cproject
.project
readme.txt

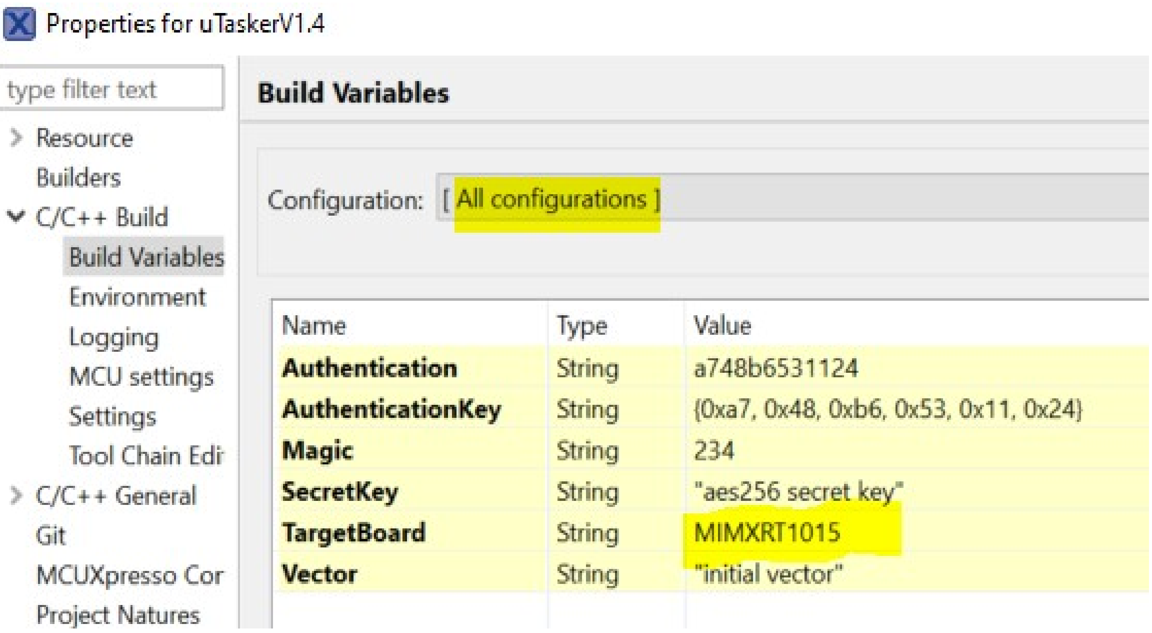
Select the files (without `readme.txt`) and copy them to the root directory [the root directory is the highest level in the repository as shown below]:



.git
.settings
Applications
FreeRTOS
Hardware
stack
Tools
uTasker
WinSim
.cproject
.gitattributes
.gitignore
.project
readme.txt
ReleaseNotes.txt

2. With MCUXpresso running choose “Import”. If there is not a command showing for this, hover the mouse over the “Project Explorer” window and right click to open its context menu. Choose the “Import” command there.
   When the Import dialogue opens expand “General “ and then select “Existing Projects into the Workspace” before clicking on the “Next >” button.
   Enter the path to the project root where the project settings were copied to in step 1 and hit the enter key to activate it. Its project will then show up in the projects windows.
   Without changing any other options click on the “Finish” button.
   This will complete the import, after which the project content will appear in the Project Explorer window.

3. Open the project properties pane and move to the `C/C++ Build | Build Variables` setting:



Properties for uTaskerV1.4

Build Variables

Configuration: [All configurations]

Name	Type	Value
Authentication	String	a748b6531124
AuthenticationKey	String	{0xa7, 0x48, 0xb6, 0x53, 0x11, 0x24}
Magic	String	234
SecretKey	String	"aes256 secret key"
TargetBoard	String	MIMXRT1015
Vector	String	"initial vector"

Be sure to select "All configurations" for Configuration and edit the **TargetBoard** string build variable to match the target to be built. In this case `MIMXRT1015` is used. Other target settings as found in the project's `config.h` file (including custom ones that have been added there) can be selected and then are valid for all subsequent steps.

- **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)
```

4. Set the "`uTaskerBoot (uTasker Boot for XiP)`" target as active configuration, if not already set (either by choosing it in the drop down list that appears right of the build project button [hammer icon] or in the menu `Project | Build Configurations | Set Active | uTaskerBoot (uTasker Boot for XiP)`. *This will generally start the target build but can be cancelled in order to check the next step.*

5. Check that the compiler is set to match the processor by selecting the project and then clicking the right mouse key to select the context menu. Choose "`Properties`" to open the properties dialogue.
   Expand C/C++ Build and select Settings and choose MCU C Compiler – Architecture. Make sure that the Cortex-M7 is selected as Architecture and choose the Floating point configuration to match the processor's FPU unit and/or project requirements. The i.MX RT 1015 has a double-precision FPU and usually uses **FPv5-D16 (Hard ABI)** in order to make use of it. Most other i.MX RT parts also have a double-precision unit but those with single-precision (like I.MX RT 1011) can select **FPv5-SP-D16 (Hard ABI)** instead.

   In case you have installed the NXP SDK you can alternatively choose the board from the SDK MCUs instead, which will correctly set processor defines and ensure no complicatins when setting up the debugger configuration. The following linker script is still needed to be set though.

   Move now to the `MCU Linker – Managed Linker Script` setting and make sure that the Managed linker script is NOT used and instead `iMX_RT_10XX_FlexSPI_NOR.ld` is set as Linker script, with the script path set to "`${ProjDirPath}\Applications\uTaskerBoot\GNU_iMX`"
   Only in the case of the i.MX RT 1064 use `iMX_RT_1064_FlexSPI_NOR.ld` instead.

6. The target can now be built after possibly configuring features required in `Applications\uTaskerBoot\config.h`

   It generates object files in `\Applications\uTaskerBoot\MCUXpresso_iMX\uTaskerBM_loader`

   See the `ReadMe.txt` file in that folder for full details of objects generated, whereby the uTaskerBoot object is not designed to be used alone and is instead used as input for following steps.

7. In order to build the uTasker Fall-back Serial loader choose the target "`uTaskerFallbackLoader`" and ensure that its CPU setting is suitable for the processor. The fallback loader always uses the linker script file \ `Applications\uTaskerV1.4\GNU_iMX\iMX_RT_10XX_FlexSPI_NOR_BOOT.ld`. The serial loader is built using the general project settings controlled by the TargetBoard setting and so no other

changes are needed.
It generates object files in
`\Applications\uTaskerSerialBoot\MCUXpresso_iMX\uTaskerSerialLoader`

See the `ReadMe.txt` file in that folder for full details of objects generated. Since the output of step 6 is used as input to this step it is necessary to follow the build order correctly.

8. In order to build the uTasker Serial loader choose the target "`uTaskerSerialBoot`" and ensure that its CPU settings are suitable for the processor. The serial loader always uses the linker script file `\Applications\uTaskerV1.4\GNU_iMX\iMX_RT_10XX_FlexSPI_NOR_BOOT.ld`. The serial loader is built using the general project settings controlled by the TargetBoard setting and so no other changes are needed.
   It generates object files in
   `\Applications\uTaskerSerialBoot\MCUXpresso_iMX\uTaskerSerialLoader`

   See the `ReadMe.txt` file in that folder for full details of objects generated. Since the output of step 7 is used as input to this step it is necessary to follow the build order correctly.

9. In order to build the uTasker project to be loaded via the serial loader choose the target "`uTaskerV1.4_BM_ITC`" and ensure that its CPU and linker script settings are suitable for the processor. Generally `\Applications\uTaskerV1.4\GNU_iMX\iMX_RT_10XX_FlexSPI_NOR_BOOT.ld` is used as linker script.

   It generates object files in
   `\Applications\uTaskerV1.4\MCUXpresso_iMX\uTaskerV1.4_BM`

   See the `ReadMe.txt` file in that folder for full details of objects generated. Since the output of step 8 is used as input to this step it is necessary to follow the build order correctly.

3. Additional Details about Building and Loading the Boot Loader

In order to build the boot loader the "Bare-Minimum" boot loader, the Fall-back serial loader and also the "Serial" loader targets need to be built. In each cases the i.MX RT target board is automtically selected by the global TargetBoard build variable. *The build order should be repected (see list above) to ensure that the output from each step is available as input to the next step.*

Bare-Minimum Loader details:
The "bare-minimum" loader is built to supply the boot configuration (read by the ROM loader at reset) at the start of SPI flash. It runs directly from SPI flash (XiP – eXecute in Place). The boot configuration supplies details about the SPI flash used and therefore it needs to be built expressly for the HW in hand.
See `__boot_config` in `iMX.c` for the boot configuration, which is generally a setup supplied by the SPI flash manufacturer to ensure optimal configuration and speed of operation.
A post build bat file is executed which generates binary output of the build in the output directory `\Applications\uTaskerBoot\MCUXpresso_iMX\uTaskerBM_loader`

The Fall-back serial loader is linked to run from SRAM (ITC) but its code is combined with the "Bare-minimum" loader code . The "BM" loader is located at the start of the SPI flash (normally 0x60000000 in the XiP memory map) and the "Fall-back" serial loader at 0x60004000.
The combination (after adding a header for identification) is performed by a bat file that is called as a post build step when the serial loader is built.
`\Applications\uTaskerSerialBoot\MCUXpresso_iMX\uTaskerSerialLoader\generate.bat`
This results in a file called `uTaskerFallbackLoaderImage_MIMXRT1015.bin` that includes both the "BM" boot loader and and encrypted version of the fall-back serial bootloader, which can be loaded to the SPI flash and then allows the "Fall-back" serial loader to operate in order to load the Serial loader. The "BM" loader copies the "Fall-back" serial loader to RAM and allows it to start when there is no serial loader present or when it is commanded to do so.

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

Building and loading the Serial Loader (loadable version)

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 `uTaskerSerialBoot` target is used instead.

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_MIMXRT1015` results .
This file is an encrypted version of the serial loader that can be loaded to the board using the fall-back loader, which will automatically operate when there is not yet a serial loader installed.

Additional files `uTaskerBootComplete_MIMXRT1015.bin` and `uTaskerBootComplete_MIMXRT1015.hex` geerated, which are a complete image of the "BM" loader plus the "Fall-back" loader plus the "Serial" loader which can alternatively be programmed in a single step.

Building and loading the application

The application should be built using the target `uTaskerV1.4_BM_ITC`, which is designed to run from RAM. The post built bat file prepares it for uploading and the file to be uploaded is created in the output directory `\Applications\uTaskerV1.4\MCUXpresso_iMX\uTaskerV1.4_BM` and the up-loadable file is called `uTaskerV1.4_AES256_MIMXRT1015.bin (`and `uTaskerV1.4_AES256_MIMXRT1015.srec).` The application image is stored in AES256 encrypted form in flash.

When the application has been uploaded (generally to the address 0x60020100, but determined by the serial loader) and the serial loader is not forced to operate (eg. by the state of an input or after being commanded by the application to do so) the "BM" loader will copy it to RAM and allow it to execute.

As mentioned above, 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 can support 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).

Outputs `uTaskerCompleteImage_MIMXRT1015.bin` and `uTaskerCompleteImage_MIMXRT1015.hex` contain complete images of the "BM" loader plus the "Fall.back" serial loader plus the "Serial" loader plus the Application andn so allows all to be loaded in a single step, which is often practival for production programming.

An additional application target `uTaskerV1.4_BM_XiP` can be build to create a file `\Applications\uTaskerV1.4\MCUXpresso_iMX\uTaskerV1.4_BM_XiP\uTaskerV1.4_BM_XiP_MIMXRT1015.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.
A second output `\Applications\uTaskerV1.4\MCUXpresso_iMX\uTaskerV1.4_BM_XiP\uTaskerV1.4_BM_XiP_AES128_MIMXRT1015.bin` is an encrypted form that is automatically operated with “On-The-Fly” decryption from the QSP flash.

The final application target `uTaskerV1.4_FLASH` generates a stand-alone application in QSPI flash which operates without a boot loader. This target is generally not used for production work since it can't be updated in the field and doesn't benefit from encryption or SRAM operation. Outputs generated are
`\Applications\uTaskerV1.4\MCUXpresso_iMX\uTaskerV1.4_FLASH\uTaskerV1.4_MIMXRT_1015.bin\Applications\uTaskerV1.4\MCUXpresso_iMX\uTaskerV1.4_FLASH\uTaskerV1.4_MIMXRT1015.hex`
`\Applications\uTaskerV1.4\MCUXpresso_iMX\uTaskerV1.4_FLASH\uTaskerV1.4_MIMXRT_1015.srec`

4. Additional Details about Building and Loading the Boot Loader

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 (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 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 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
0x**1**234 is a magic number of 0x0234 which should be copied to, and executed in internal RAM (it is linked to the address 0x300)

0x**9**234 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

0x**0**234 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.

0x**2**234 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.

0x**3**234 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 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.

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 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.

5. Using SDK Applications with the µTasker Loader

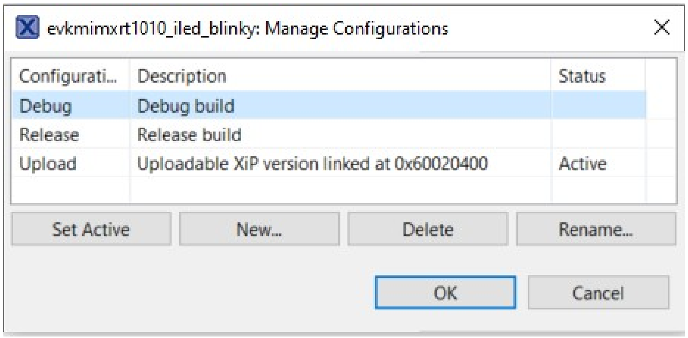
Although it is hoped that also the µTasker application will be found to be a more advanced solution than the traditional semiconductor manufacturer's framework (SDK) the µTasker loader can be used with applications from any source.
Important – *if SDK code requires SDRAM access please also consult the 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 µTasker 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 MCUXpressor is to create a new target clalled "Upload". This is simple to do by making a copy of the orignal 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 built identically to the original one but can now be modified to generate an uploadable version fo the same project.

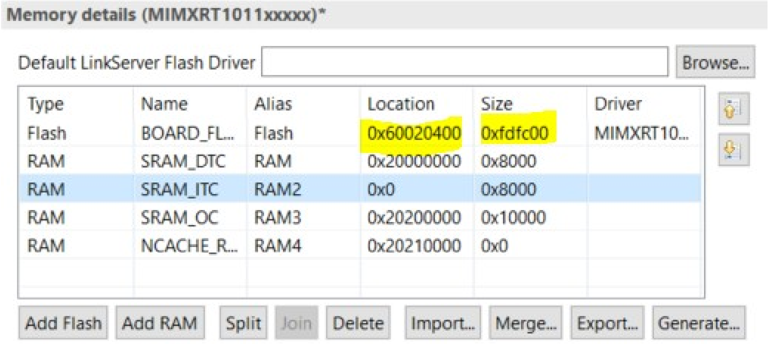


evkmimxrt1010_iled_blinky: Manage Configurations

Configurati...	Description	Status
Debug	Debug build	
Release	Release build	
Upload	Uploadable XiP version linked at 0x60020400	Active

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*.



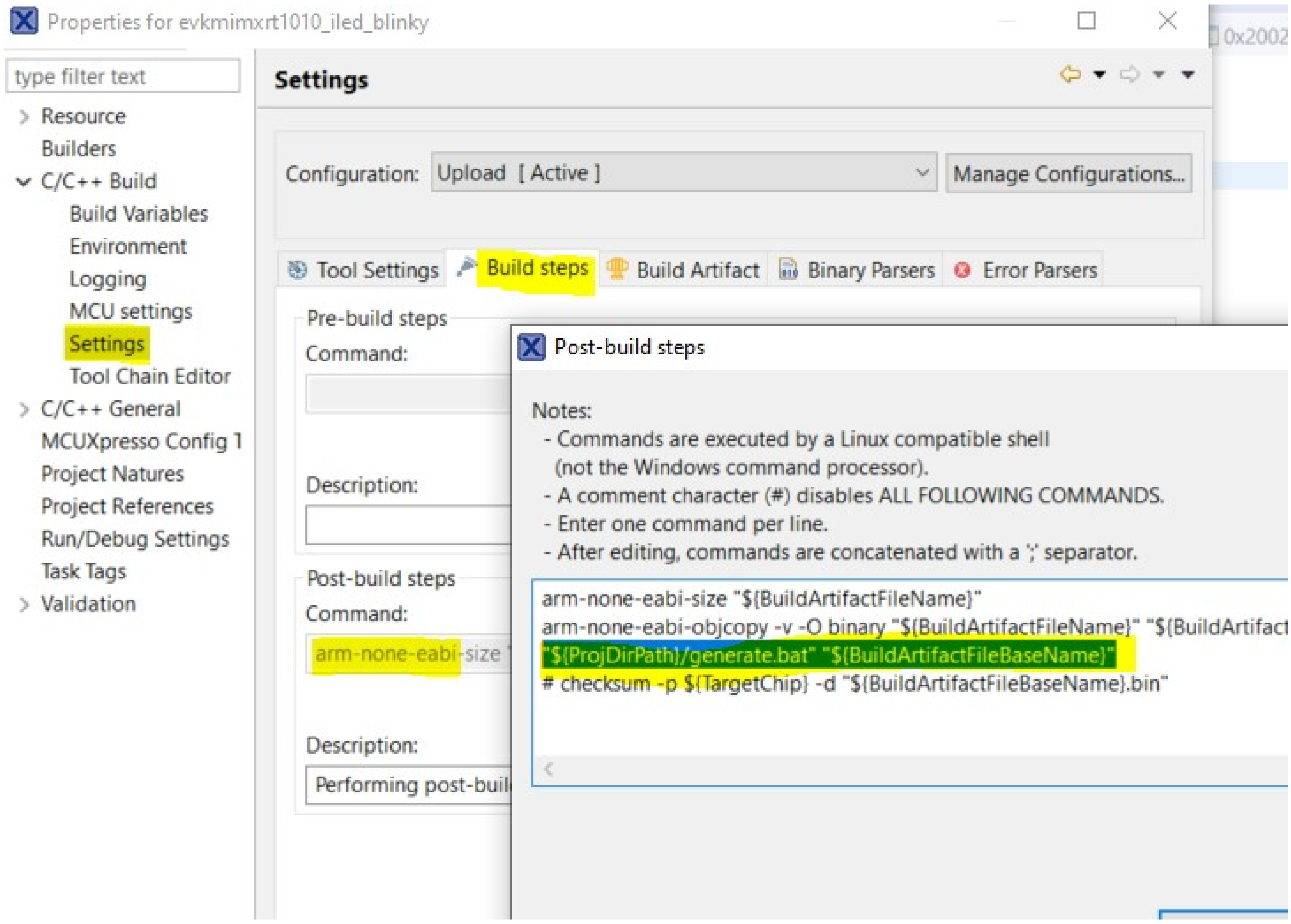
Memory details (MIMXRT1011xxxxx)*

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	0x0	0x8000	
RAM	SRAM_OC	RAM3	0x20200000	0x10000	
RAM	NCACHE_R...	RAM4	0x20210000	0x0	

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 (remove the # before its command) and a bat. file call added:



Configuration: Upload [Active]

Notes:
- Commands are executed by a Linux compatible shell (not the Windows command processor).
- A comment character (#) disables ALL FOLLOWING COMMANDS.
- Enter one command per line.
- After editing, commands are concatenated with a ';' separator.

```
arm-none-eabi-size "${BuildArtifactFileName}"
arm-none-eabi-objcopy -v -O binary "${BuildArtifactFileName}" "${BuildArtifact
"${ProjDirPath}/generate.bat" "${BuildArtifactFileBaseName}"
# checksum -p ${TargetChip} -d "${BuildArtifactFileBaseName}.bin"
```

The line

`"${ProjDirPath}/generate.bat" "${BuildArtifactFileBaseName}"`

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
```

a. The path to the uTasker tools directory is set as a path variable to match its location on the PC
b. The variables (`SECRET_KEY`, `VECTOR` etc.) should match the ones used by the uTasker loader configuration (*the ones above match with the default settings*)
c. 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 of the MCUXpresso project.
d. 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-the-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:

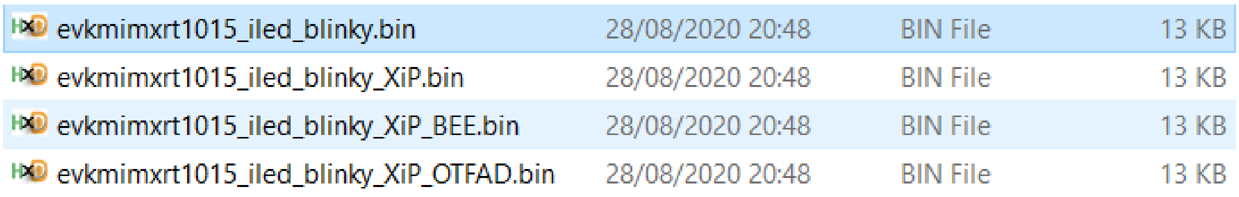
```
// We add 760 bytes of padding between the header and the start
of code in order to
// align the code on a 1k (0x400) byte boundary (ensures on-
the-fly decryption compatibility,
// library compatibility and also allows interrupt vectors to
remain in code)

02f8                   // first two bytes specify the length
ffffffffffff
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
```

```
ffffffffffffffff
ffffffffffffffff
ffffffffffffffff
ffffffffffffffff

ffffffffffffffff
ffffffffffffffff
ffffffffffffffff
ffffffffffffffff

ffffffffffffffff
ffffffffffffffff
ffffffffffffffff
ffffffffffffffff

ffffffffffffffff
ffffffffffffffff
ffffffffffffffff
ffffffffffffffff

ffffffffffffffff
ffffffffffffffff
ffffffffffffffff
ffffffffffffffff

ffffffffffffffff
ffffffffffffffff
ffffffffffffffff
ffffffffffffffff

ffffffffffffffff
ffffffffffffffff
ffffffffffffffff
ffffffffffffffff

ffffffffffffffff
ffffffffffffffff
ffffffffffffffff
ffffffffffffffff

ffffffffffffffff
ffffffffffffffff
ffffffffffffffff
ffffffffffffffff

ffffffffffffffff
ffffffffffffffff
ffffffffffffffff
ffffffffffffffff

ffffffffffffffff
```

```
ffffffffffffffff
ffffffffffffffff
ffffffffffffffff

ffffffffffffffff
ffffffffffffffff
ffffffffffffffff
ffffffffffffffff

ffffffffffffffff
ffffffffffffffff
ffffffffffffffff
ffffffffffffffff

ffffffffffffffff
ffffffffffffffff
ffffffffffffffff
ffffffffffffffff

ffffffffffffffff
ffffffffffffffff
ffffffffffffffff
ffffffffffffffff

ffffffffffffffff
ffffffffffffffff
ffffffffffffffff
ffffffffffffffff

ffffffffffffffff
ffffffffffffffff
ffffffffffffffff
ffffffffffffffff

ffffffffffffffff
ffffffffffffffff
ffffffffffffffff
ffffffffffffffff

ffffffffffffffff
ffffffffffffffff
ffffffffffffffff
ffffffffffffffff
```

This file should also be located in the root directory where `generate.bat` is location.

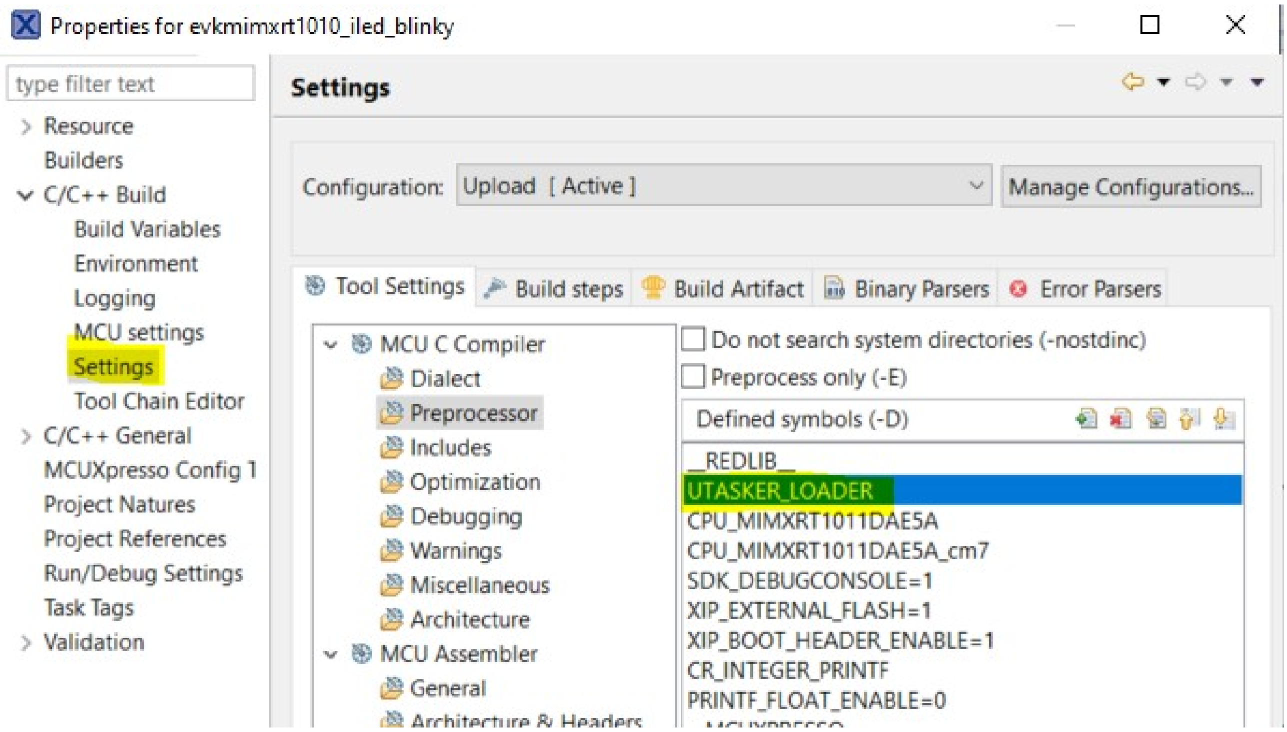
In the MCUXpresso workspace output target directory the following binary files would be found in the case of building the SDK's blinky project for the i.MX RT 1015:



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 is to allow interrupt vectors to run from ITC (this is more efficinet than leaving them in QSPI flash), 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:



Configuration: Upload [Active]

□ Do not search system directories (-nostdinc)

□ Preprocess only (-E)

Defined symbols (-D)

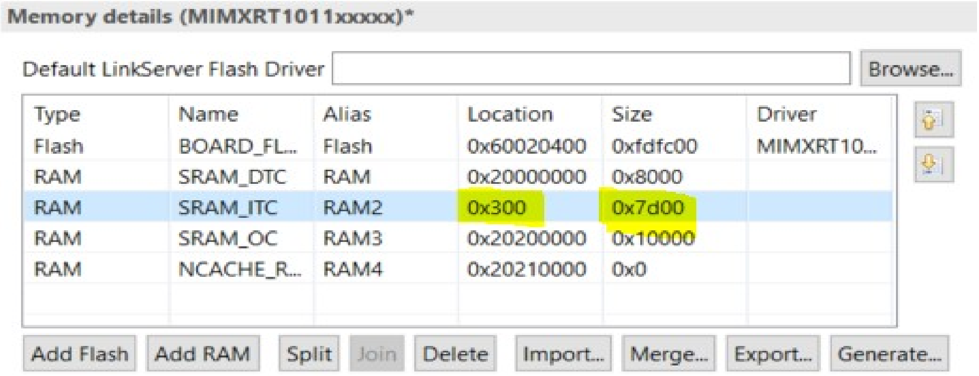
```
__REDLIB__
UTASKER_LOADER
CPU_MIMXRT1011DAE5A
CPU_MIMXRT1011DAE5A_cm7
SDK_DEBUGCONSOLE=1
XIP_EXTERNAL_FLASH=1
XIP_BOOT_HEADER_ENABLE=1
CR_INTEGER_PRINTF
PRINTF_FLOAT_ENABLE=0
```

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	

6. 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 enabled 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.

7. 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 assember and needing to carefully understand the technique involved.

When working with the µTasker 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 µTasker 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 chapter 5 for its details) will instruct the µTasker loader to prepare it.

First consider the standard header file content:

```
02f8                 // first two bytes specify the length

ffffffffffff         // 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

030805ffffff            // specify DTC/ITC and OCR bank sizes to be
pre-configured for the application (when not ff)

ffffffffffffffff

ffffffffffffffff


followed by further ff padding bytes

....
```

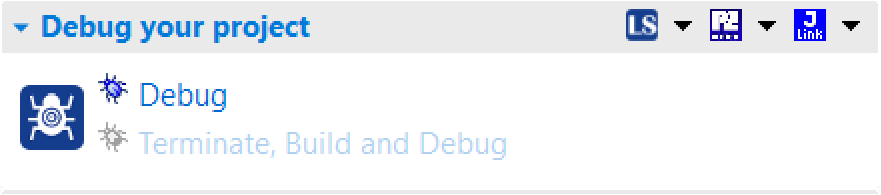
3 banks to be assigned to DTC, 8 to ITC and 5 to OCR

The µTasker 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.*

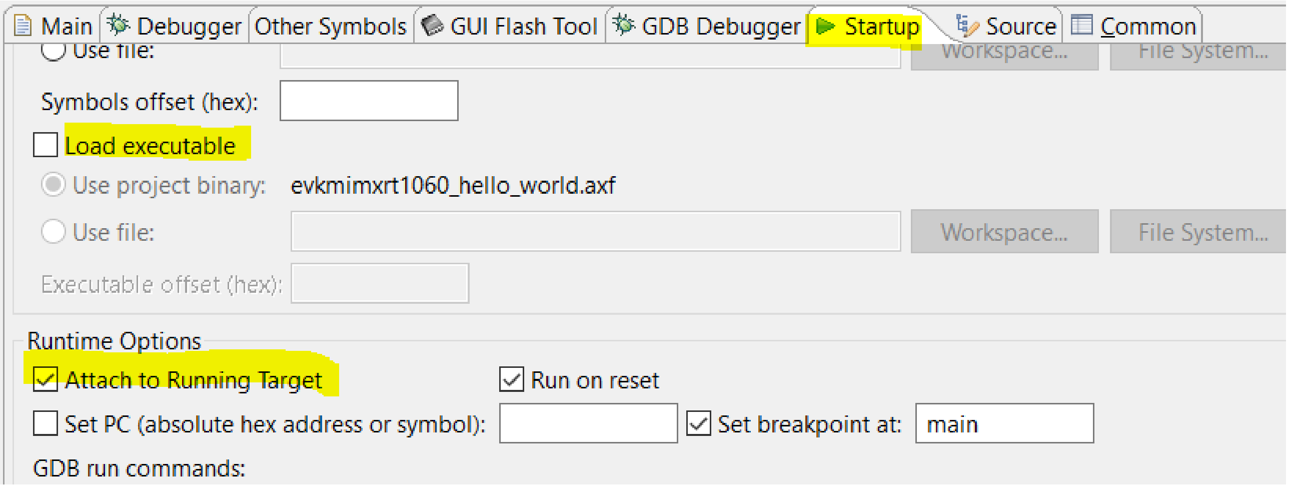
No further application effort is required, making this a very simple, fast and painless way to achieve an XiP based application's preferred FlexRAM layout.

8. Debugging with MCUXpresso

It is advised to have the NXP SDK installed for the board that is being used. This is not necessarily in order to use its code but instead to ensure that the debugger works correctly. If an NXP SDK based application is being built to work together with the µTasker loader this will already mean that the board's debugger configurations are already present.
On very first use a debug target needs to be created, which is simplest done by using MCUXpresso's Debug method in the Quickstart Panel:



When this is executed connected debuggers are searched for and the built image will tend to be loaded, although this will usually fail when the loader is already operating. This is in fact not a problem because we don't want to load the application with the debugger since it is already, or will be, loaded with the loader and the debugger's job is to just do debugging.
Now that the debug configuration has been created it should be edited – to do this the debugger's launch can be opened by double-clicking on it. Otherwise use the menu "`Run | Debug Configurations`" and edit it's "*Startup*" tab setting to NOT "Load executable". Also best reliability tends to be achieved by checking the setting "*Attach to Running Target*"



○ Use file:

Symbols offset (hex):

☐ Load executable

● Use project binary: evkmimxrt1060_hello_world.axf

○ Use file:

Executable offset (hex):

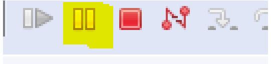
Runtime Options

☑ Attach to Running Target ☑ Run on reset

☐ Set PC (absolute hex address or symbol): ☑ Set breakpoint at: main

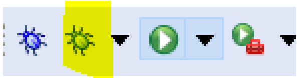
GDB run commands:

Assuming that the application has been loaded to the board via the µTasker loader the debugger can already be connected by clicking on "*Debug*" in this window.
In this mode the target continues running when the debugger has been attached and can be paused by clicking in the suspend icon:



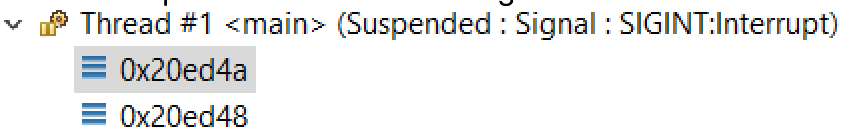
which will pause the execution at some point in the code. The code can now be stepped and breakpoints inserted and run to.

When the debugger is terminated it can be connected again by simply using the debug icon:



9. What to do when things go Horribly Wrong?

Debuggers and i.MX RT and expecially MCUXpresso tend to have some real difficulties working with code that has a fault – eg. Crashes and ends up in the internal ROM loader space. Using the "*Attach to Runing Target*" setting tends to help though and when paused and the following is seen

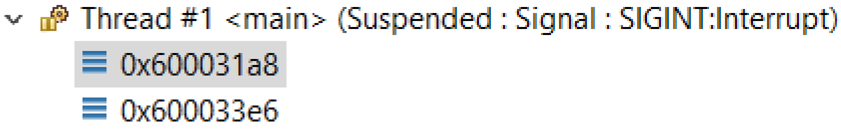


```
Thread #1 <main> (Suspended : Signal : SIGINT:Interrupt)
    0x20ed4a
    0x20ed48
```

it becomes immediately visible that the ROM Loader (these addresses are in its program space) is in operation, which is typically the case after a serious program failure. However any amount of commanding restarts and running will not allow a brekpoint set at the entry point of the failing application (or even a good application) to be hit. In this case the following technique can save the day:
1. Set the breakpoint that you want to stop at (usually very early on in the startup to be sure that it is hit before any failure takes place)
2. Disconnect the debugger from the target
3. Set the µTasker "BM" loader's `WAIT_INPUT` to its active state. Search for its define if not known, for example (the input used can be modified to suit the hardware available)

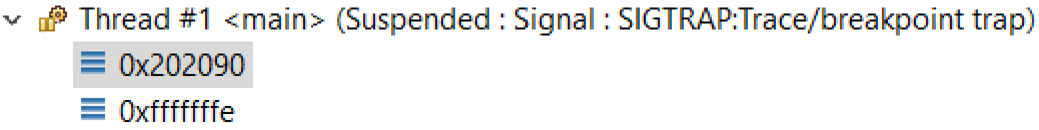
```
#define WAIT_INPUT                          PIN_GPIO_AD_B0_00_GPIO1_IO00
```

4. Reset/Restart the board and the "BM" loader will stay in a start-up loop, waiting for this input to be released again.
5. Connect the debugger again. If paused, the code can be found looping in XiP code space – eg.



```
Thread #1 <main> (Suspended : Signal : SIGINT:Interrupt)
    0x600031a8
    0x600033e6
```

6. With the code executing negate the `WAIT_INPUT`.
The ROM loader will now be hit because the "BM" loader first performs a reset before the application is started. This causes the debugger to stop with these details:



```
Thread #1 <main> (Suspended : Signal : SIGTRAP:Trace/breakpoint trap)
    0x202090
    0xfffffffe
```

7. Finally command "Run" and the breakpoint set in the startup code will now be hit!

From this point the debugger can be used “normally” to step the code and find out what is causing it to fail.

When using MXUXpresso this technique is needed in order to hit an initial breakpoint anywhere in the code if pausing an operating program is not adequate to subsequently start code debugging.

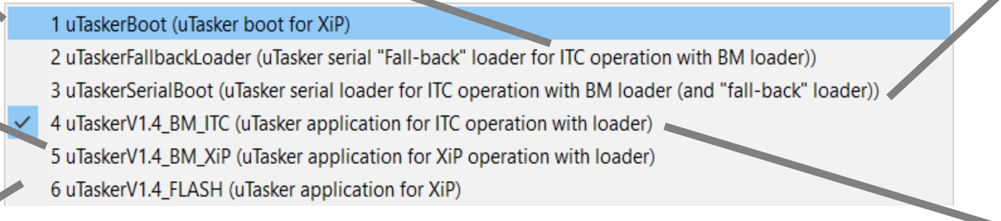
10. Summary

This is the first boot loader that runs in spi flash [XiP – eXecute In Place]. It is linked at 0x60000000 – *or 0x70000000 for i.MX RT 1064*. It is the first target that is built (before building uTaskerSerialBoot (“Fall-back” and serial versions) and combining the three together). In fact the first boot loader is quite useless alone since it works together with the respective serial loader which it copies to instruction RAM so that it can do its work and manipulate spi flash.

This is the “Fall-back” serial boot loader which is built as second step (with define **`iMX_FALLBACK_SERIAL_LOADER)`**. This step also combines it with the first boot loader (that was built in step 1). It is linked to run in instruction RAM (start address 0x300) but is combined with the first boot loader at address 0x60004000 (0x70004000 for i.MX RT 1064). The resulting image from the combining step is programmed to the spi flash so that the first boot loader boots the processor and installs the serial loader in internal RAM and allows it to execute.

*When built without **`iMX_FALLBACK_SERIAL_LOADER`** it is the loadable serial loader*

This is the XiP loadable version (both plain-code and AES128 On-The-Fly decrypted versions)



1 uTaskerBoot (uTasker boot for XiP)
2 uTaskerFallbackLoader (uTasker serial "Fall-back" loader for ITC operation with BM loader))
3 uTaskerSerialBoot (uTasker serial loader for ITC operation with BM loader (and "fall-back" loader))
✓ 4 uTaskerV1.4_BM_ITC (uTasker application for ITC operation with loader)
5 uTaskerV1.4_BM_XiP (uTasker application for XiP operation with loader)
6 uTaskerV1.4_FLASH (uTasker application for XiP)

This is the application that can be loaded using the boot loader concept. It is linked to operate in instruction RAM at address 0x300 and is installed by the first boot loader in the same way that it installs the serial loader. The first boot loader installs and starts executing the application when it is present and when it is not instructed to install and start the serial loader instead. Where this application image is actually stored in spi flash is determined by the loaders and not relevant to building the target itself.

This target builds the application as stand-alone application. It runs without boot loader in flash [XiP] (linked to start at 0x60000000 – or 0x70000000 for i.MX RT 1064). It can't be loaded by a boot loader and needs to be programmed to its spi flash location. *This target is not generally used by the µTasker concept since it is usually more practical and efficient to work with the version that can be loaded by the boot loader*.