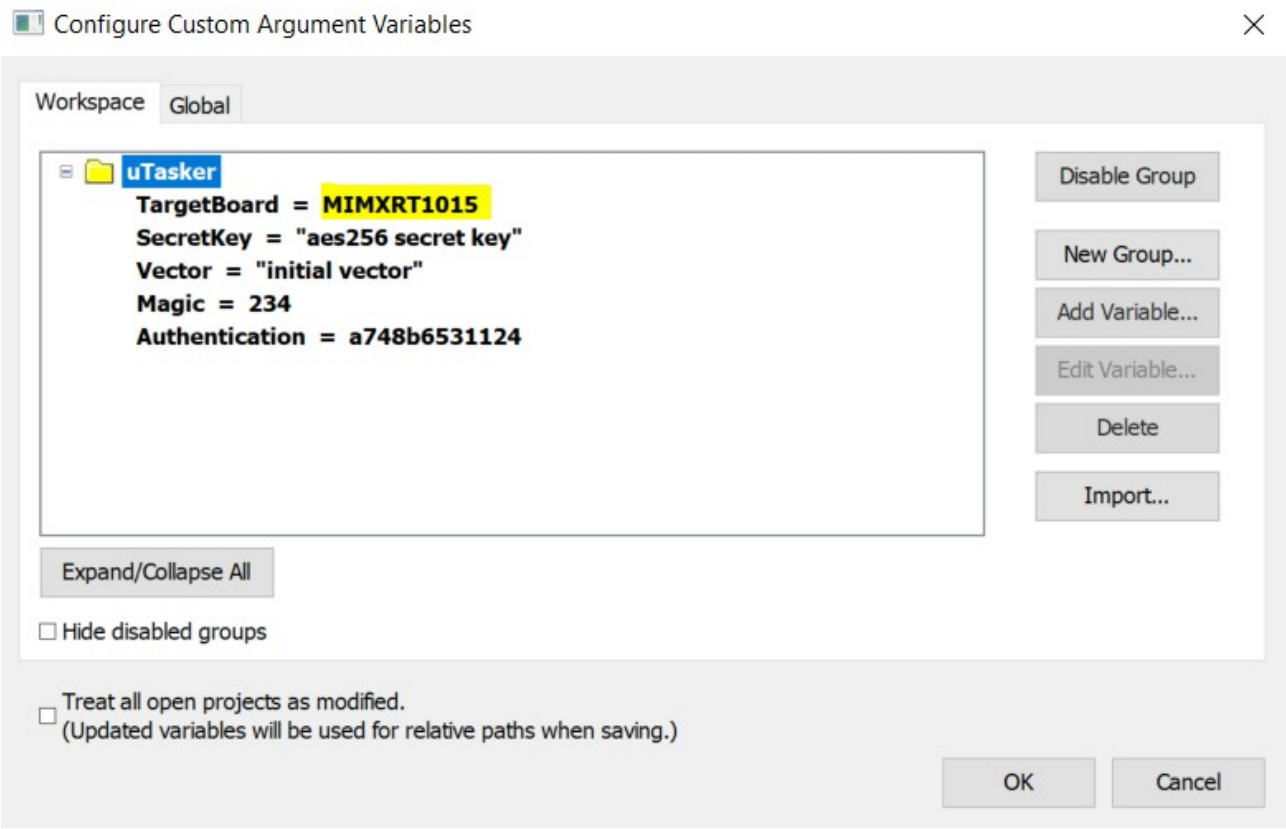
Using the uTasker project with i.MX RT in IAR

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.

This guide uses the `MIMXRT1015` board as reference but other boards are equivalent when the TargetBoard variable is set accordingly.

The TargetBoard is set globally so that it is valid for all of the projects (“Bare minimum” loader, “Fall-back” serual 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

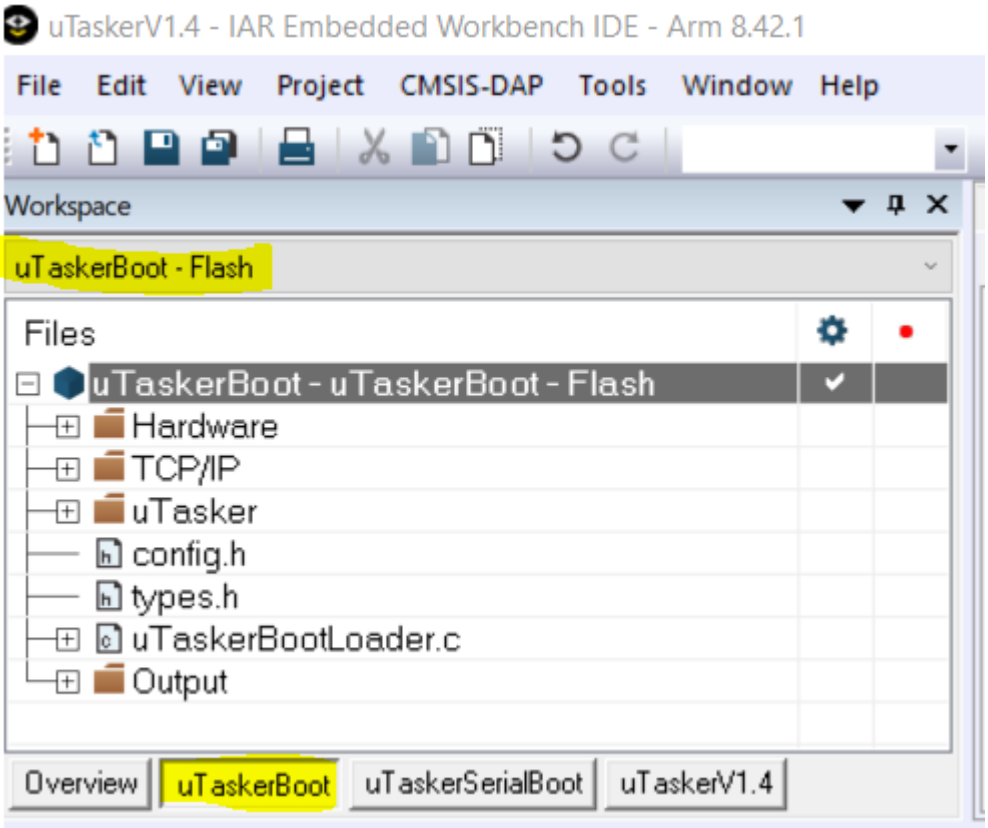
These variables are used to control the tools that generate the versions for uploading purposes and should match with the values in the “BM” boot loader and serial loader code:

```
#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)
```

Building the “Bare-Minimum” Boot Loader

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



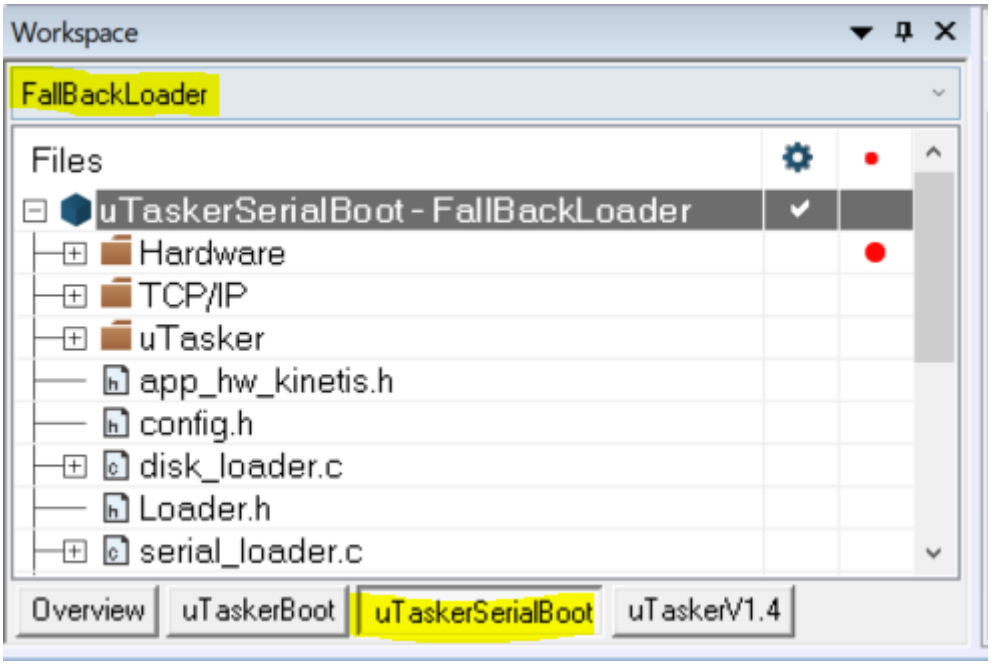
In the project's options ensure that the general options are set for the correct device; in this case it would be
`NXP MIMXRT1015xxx5A`

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.

Building and loading the Fall-back Serial Loader

Choose the **uTaskerSerialBoot** project and the **uTaskerBoot – FallBackLoader** 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\`**`uTaskerFallbackLoaderImage_MIMXRT1015.bin`**

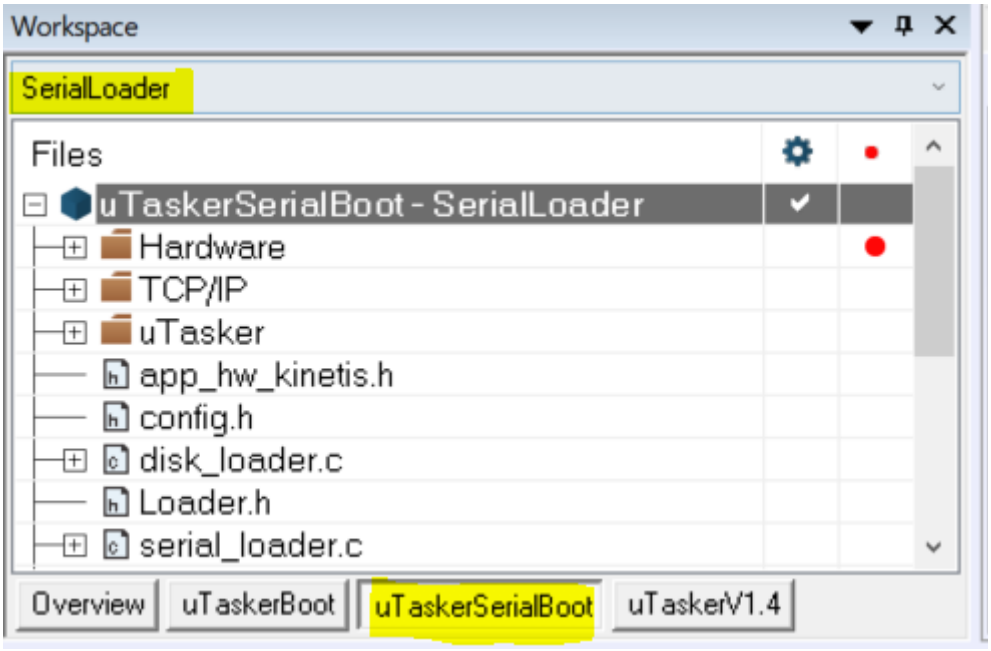
This contains the “Bare-minimum” loader plus an encrypted version of the “Fallback” serial loader that can be loaded to the board.

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 turorial, chapter 4: https://www.utasker.com/docs/iMX/uTaskerV1.4_iMX.pdf

Building and loading the Serial Loader (loadable version)

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.

Loading the combined “BM” boot loader, “fall-back” serial loader and “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 turorial, chapter 4: https://www.utasker.com/docs/iMX/uTaskerV1.4_iMX.pdf

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.

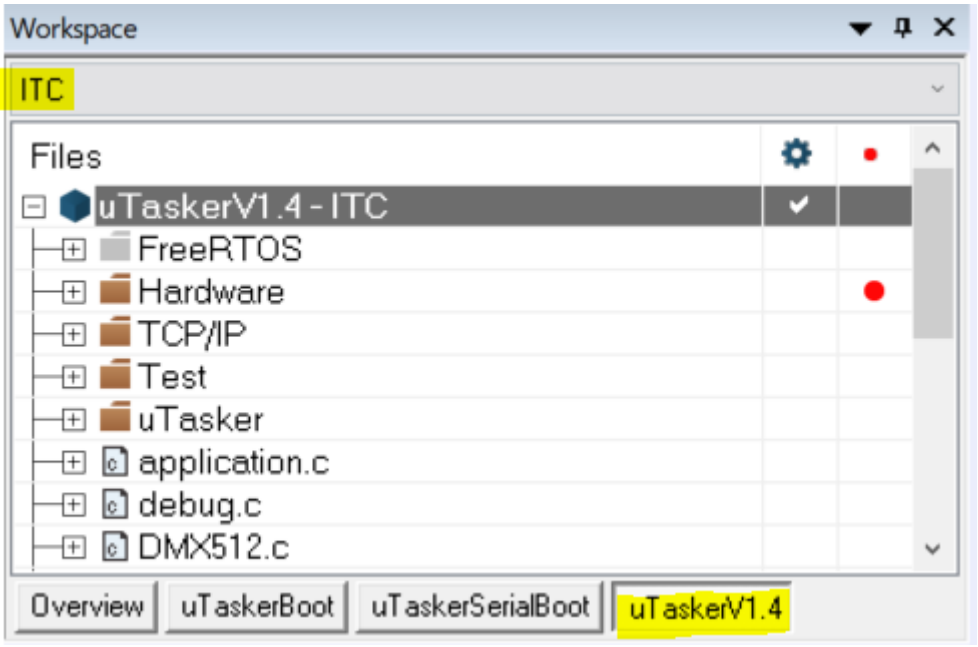
Note that different loader strategies may be chosen when building the “serial” loader to the “fall-back” loader and possibly other configuration modifications that suit the working serial loader to be used (rather than the fall-back one).

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.

Building and loading 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.

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

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

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

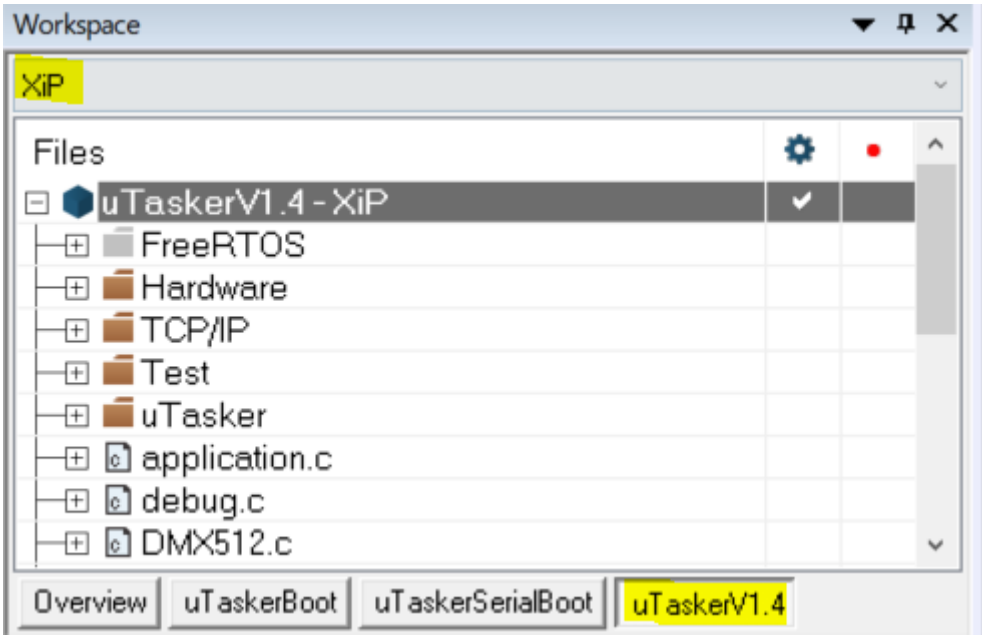
This image can be uploaded to the board using the serial loader in order to update the application used in the product.

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 fo large application 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” decrption so that the encrypted application in SPI flash can directly run from there.

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 August 2020