# µTasker Document

# µTasker "Bare-Minimum" Boot Loader
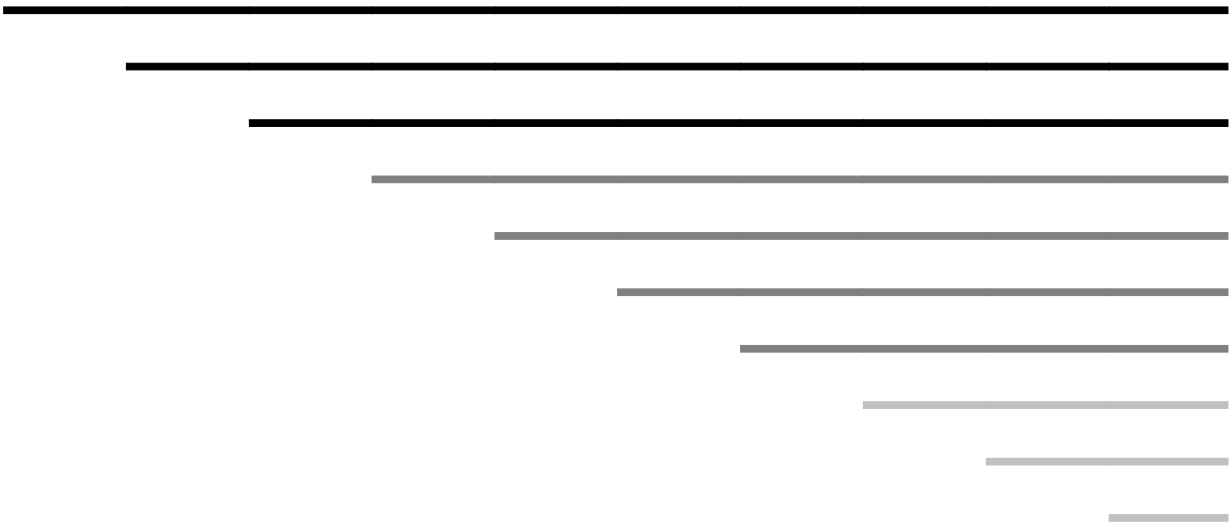
## Table of Contents

# 1. Introduction

µTasker is an operating system designed especially for embedded applications where a tight control over resources is desired along with a high level of user comfort to produce efficient and highly deterministic code.

The operating system is integrated with TCP/IP stack and important embedded Internet services alongside device drivers and system specific project resources.

µTasker and its environment are essentially not hardware specific and can thus be moved between processor platforms with great ease and efficiency.

However the µTasker project setups are very hardware specific since they offer an optimal pre-defined (or a choice of pre-defined) configurations, taking it out of the league of "board support packages (BSP)" to a complete "project support package (PSP)", a feature enabling projects to be greatly accelerated.

Very often there is a requirement for software updates in the field and where possible over the Internet. There are several methods which can be used, each with its advantages and disadvantages. The µTasker boot-loader support is optional in the µTasker environment and chooses a technique to allow Internet enabled uploads of application software based on several IP techniques (FTP, HTTP POST etc.) as well as optional serial methods using an absolute minimum of boot software space. It allows complete uploads of application software including operation system, driver, interrupt and TCP/IP stack code for maximum flexibility but also with reliability as top priority; failed uploads will not result in disaster but can be repeated (or are automatically repeated) until successful.

*As an option, the loaded code can be first encrypted to a form which can be distributed without being interpretable as final machine code. This protects the delivered code from being reverse engineered or used in other unauthorised projects.*

*In addition to the described method of uploading new code to internal FLASH, the uTasker project supports also uploading to external SPI based FLASH, which enables programs almost as large as the internal FLASH to be uploaded since it doesn't need to share this internal FLASH space.*

**µTasker boot-loader strategies**

The µTasker boot-loader strategy can be configured depending on the project requirements. The following options are possible:

- HTTP/FTP upload support integrated in the boot loader (Ethernet, ARP and TCP)

- Serial loader support integrated in the boot loader

- Override options to force serial or HTTP/FTP uploads (eg. Input port state at reset)

- Override of IP configuration option (eg. Input port state at reset) to enable a board with unknown configuration to be contactable via Ethernet – useful mainly together with HTTP/FTP support

- µTasker "***bare-minimum***" loader – the subject of this document.

## 2. µTasker "bare-minimum" loader

The "**bare-minimum**" loader is usually the base for all other configurations. It supports safe updates of new application software which has been prepared by the previous application and as such has the advantage of requiring an absolute minimum amount of space in FLASH. However this solution can also be a very powerful solution since it still can enable the safe update of entire software, including unrestricted operating system, driver and TCP/IP stack patches, over the Internet.

The µTasker "**bare-minimum**" loader is a small standalone program. It doesn't have any of its own network capabilities and doesn't configure the Ethernet interface. It requires no further options to be enabled to achieve its goals.

The µTasker "**bare-minimum**" loader assumes that at start-up one of the following two states exists.
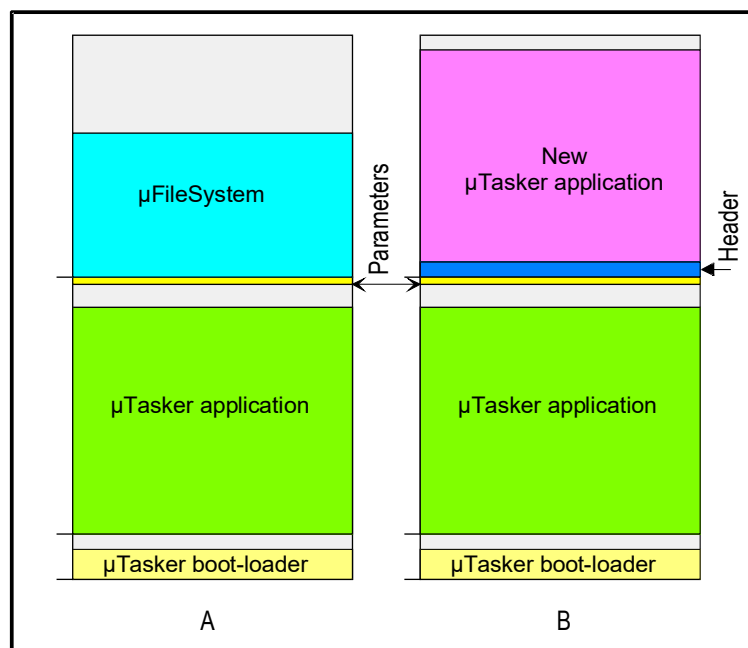


State A is the initial and normal condition. A µTasker application exists in FLASH at the application's start position and there is no new software waiting to be updated. Note that the case where only the µTasker boot-loader exists is not normally a valid case – the initial load (which will normally be performed using a programmer/debugger) is loaded together with the first µTasker application. [If the µTasker boot-loader contains also serial loading support, the application can of course be loaded in a second step – see http://www.utasker.com/docs/uTasker/uTaskerSerialLoader.PDF for details of the serial loader, also known as the secondary boot loader when operating together with the "Bare-minimum loader"].

When the µTasker boot-loader recognises state A it simply allows the µTasker application to start operating. Details about starting the application and the configuration of the application program are given later in this document.

State B exists thanks to the support built in to the µTasker application, which has allowed a new version of a µTasker application to be programmed into the region in FLASH reserved for it. This will usually be performed over the internet (network). It is not the Bare-minimum boot loader itself which is responsible for loading this!

The new µTasker application includes also a small header which allows the µTasker boot-loader to recognise it as a valid copy of code and to also verify its integrity. This is extremely

important since the device will lose its IP uploading support if an unsuitable program were to be loaded in place of the original one.

It is also clear that both the old and new µTasker applications must fit into the available non-volatile space. This is due to the fact that one of the programs must at all times be intact in order to guaranty that the update cannot fail and leave the device in an unusable state. It is not however absolutely necessary that this is in the internal FLASH. If additional serial FLASH is available this can be optionally supported. As long as the internal FLASH has adequate space, it is usually easy to justify using a device with more FLASH than the actually application needs alone due to the valuable additional feature that the IP uploading support represents.

*It is also important to note that the space used to store the µTasker application may overlap with the µTasker file system used during normal operation. For example a M52235 with 256k FLASH can use a 128k file system for web pages and other data alongside the application code of up to 120k. A new µTasker application can be loaded to the file system area ready for programming. No additional FLASH would be required in this example, although the contents of the file system will usually also have to be restored (eg. via FTP) after the upload procedure is complete.*

*If external SPI FLASH is used to save the uploaded code this restrictions doesn't apply. See the SPI FLASH section for more details.*

# 3. Procedure

The µTasker boot-loader is always the first program started after a reset.

It first checks to see whether there is a new µTasker application waiting to be programmed. To be recognised as a valid new µTasker application it must have a valid header and its integrity must be verified according to the header contents. The header is detailed later.

If there is no 'new' application waiting, the existing application will be started.

Should a new and valid µTasker application we waiting to be programmed, the original application will be deleted. It should be noted that a restart of the device during this phase is not dangerous since the delete will be simply repeated.

After the delete phase has terminated the new µTasker application is copied to the application code space.

The new application is now verified according to the integrity check in the header, to be sure that there were no errors when copying.

Finally the intermediate copy of the µTasker application is deleted and the device is reset.

It can be shown that a power fail or other error can be tolerated at any part of the procedure and that the final reset is equivalent to a standard start with the µTasker application programmed.

The program flow diagram of this procedure can be found in Appendix A.

## 3.1. Support of Uploading Secondary Loader and Application

When the define `ADAPTABLE_PARAMETERS` is enabled it is possible to configure the operation to work with two different applications. The first is usually a secondary loader like a serial loader which the "Bare-Minimum" Boot Loader starts, which subsequently allows the application to be programmed or starts the final application.

With this option the intermediate storage area can be used to upload either a new secondary loader or a new application and the "Bare-Minimum" Loader performs the update according to details in the code header as detailed in the following chapter.

Depending on whether the code to be updated is a secondary boot loader or application code the "Bare-Minimum" Boot Loader will also delete and copy to the corresponding destination areas.

The program flow diagram of this procedure when using this option is also in Appendix A.

# 4. Code Header

In order to ensure that the µTasker boot-loader doesn't attempt to program unsuitable code and also to guaranty the integrity of the loaded code a small header is used. The header consists of 8 additional bytes as follows:

```
unsigned long  ulCodeLength;
unsigned short usMagicNumber;
unsigned short usCRC16;
```

The header can be added to the program code by using a simple utility program which accepts a normal binary file, calculates its check sum and adds these 8 bytes to the beginning of the file. This utility program is supplied with the µTasker project (in the tools directory) and is called `uTaskerConvert.exe`

The code length is then used to verify the size of the received code and also for the verification of its CRC value. The magic number is used as a version number and also as simple verification of the compatibility of code otherwise respecting the format – the magic number can be set for a project and kept secret (although it could be eves dropped from the upload traffic).

The CRC is calculated over the complete code (without header) plus a secret block of variable length data common to the boot software and also the generation program but not visible in the transmission. This offers protection against malicious uploads since the check sum must be correct over the visible code and also some secret code.

The µTasker boot-loader checks for a valid waiting software and also when verifying that the newly loaded software has indeed been copied without errors. Only when the new µTasker application has been successfully tested against its CRC value will the backup version finally be deleted.


When `ADAPTABLE_PARAMETERS` is enabled the "Bare-Minimum" Boot Loader recognises two magic numbers; one for the secondary loader and one for the application. Each of the possible upload headers are generated using their own secret key.

## 5.  Setting up the µTasker project, preparing and uploading the file

The µTasker project requires the following small modification to be able to operate together with the "***bare-minimum***" loader.

- Its start address must be set to correspond to the start address in the "***bare-minimum***" loader. This can be set in the linker file.
  This is set up, for example, in the `CodeWarrior.lcf` file in the memory definition like this. In this case 0x800 (2k) has been left at the beginning of FLASH for a µTasker boot-loader is up to 2k. Also the length of the FLASH block has been reduced by the same amount
      flash   (RX)   : ORIGIN = 0x0000800, LENGTH = 0x0003F800

The project is otherwise compatible, including the use of the interrupt vector table. The µTasker project includes a project setup for the boot-loader and an application for "***bare-minimum***" loader.

When the project is compiled a binary image should be created. This can usually be set up in the linker configuration of the project.

Finally the output file is converted to a format with the necessary header. Here is an example of calling the `uTaskerConvert.exe` utility to add a header with a magic number of `0x1234` to the input binary file `uTasker_demo.bin`

      uTaskerConvert uTasker_demo uTasker_update.bin –0x1234
      –a748b6531124

The secret block can be of any length up to 100 bytes and in the example is made up of 6 bytes `0xa7, 0x48, 0xb6, 0x53, 0x11, 0x12`

The output file `uTasker_update.bin` can then be uploaded to the target using whatever method is supported by the presently loaded µTasker application code. The following shows a typical solution using the HTTP POST method which is demonstrated in the standard µTasker demo project assuming that the upload support is availabled and activated.

A standard Web Browser input allows the user to select a file to be loaded from the user's hard disc. In this example the user has already selected a file with the name `sw1.bin` - the format of which has been prepared to include the necessary header.

By clicking on the SW Update button the Browser sends this file using the HTTP POST method. The µTasker application software saves the received data to FLASH at the required location and commands a reset of the board. This starts the µTasker boot-loader update procedure as previously described, after which the new software will be started.

*It is important to note that the uploaded code is either posted to a dedicated FLASH memory region or to within or overlapping the µTasker file system. It is not posted to the parameter region so that the IP configuration of the board remains intact. If this were not respected it would be possible to lose user settings and should thus be avoided. The coordination of the posting itself is resolved in the µTasker application code.*

*If external SPI FLASH is used to save the uploaded code this restrictions doesn't apply. See the SPI FLASH section for more details.*

## 6.  SPI FLASH Support

The µTasker boot-loader can be configured to operate together with external SPI FLASH. SPI FLASH chips enable relatively large amounts of data to be stored in a small device (eg. 512k to 2Meg in SO-8 housing) connected by a high speed SPI (Serial Peripheral Interface) interface. Not only are the devices small but their connection is simple due to the low number of pins.

The "Bare-Minimum" Boot loader operation is equivalent but instead of needing the new code to be copied to a region in internal FLASH, the new code can be copied to the external device. This allows uploads of new code without requiring it to be stored in the internal file system. This means that no data will have to be overwritten (if the code needs to borrow space in the file system). In some circumstances it allows also larger code to be uploaded and updated – *up to the complete size of the internal flash, minus the size of the Bare-Minimum Boot loader itself*.

The boot loader size tends to be a little larger than when only internal FLASH is used due to the fact that it must have both the internal FLASH driver and also an external SPI FLASH driver.

When using the Boot Loader with external SPI FLASH, the define `SPI_SW_UPLOAD` is added to the project set up (`config.h`). See the processor-specific guides for more details at http://www.utasker.com/docs/documentation.html

# 7. µTasker boot-loader example for the Freescale™ Coldfire M5223X

The details of the realisation are rather processor specific and so it is necessary to have a good understanding of the processor it is running on. So we must learn some specifics of the device to understand its restrictions and how and why the solution has been shaped for it.

When the Coldfire starts it automatically reads two long words from the exception vector table. The first is its initial stack pointer value and the second is the initial program counter value. Since the exception vector table is located at reset at the address 0x00000000, in FLASH in the M5223X, it is clear that the µTasker boot-loader must reside at the beginning of FLASH so that it can coordinate the initialisation of the system.

The initial program counter, read from FLASH location 0x00000004, is the start address of the µTasker loader code.

The exception vector table includes in total 256 vectors, including the SP and PC and it is typical to reserve 1k of space at the start of FLASH to contain the interrupt vector addresses used later by the code. Of course the µTasker loader has no idea where these will later be located once the user code has been installed and so the exception vector table is not used in FLASH but rather the code starts directly at the location 0x00000008, thus utilising the FLASH as efficiently as possible.

Note the following complication in the M5223X: *The locations between the addresses 0x400 and 0x417 are read on reset and used to configure some FLASH control registers. Therefore the µTasker boot-loader sets up the compiler and linker to position zeros in these locations. If this is not done, the FLASH configuration can leave FLASH blocks with protection against instruction accesses and so to program failure*.

The stack pointer is set to the top of internal SRAM, positioned by default between 0x20000000 and 0x2000ffff (32k in all M5223X devices). Therefore the content of the first FLASH locations are defined:

```
0x00000000: 0x20008000
0x00000004: 0x00000008
0x00000008: First instruction of µTasker boot-loader
...
...
...
0x00000400: 00000000
0x00000404: 00000000
0x00000408: 00000000
0x0000040c: 00000000
0x00000410: 00000000
0x00000414: 00000000
0x00000418: Further µTasker boot-loader code
...
...
```

The "***bare-minimum***" loader requires less that 2k in the M5223X and so the application starts at 0x00000800 (the second FLASH sector).

Although the start address of the µTasker application has been moved from its usual starting address of 0x00000000 to 0x00000800, no further code changes are necessary. The

µTasker boot-loader takes over the normal reset procedure of loading the stack pointer with the first long word and setting the PC with the second long word and so the reset vector retains its function. The µTasker project sets up the interrupt vectors at the start of SRAM, rather than leaving them in FLASH and so this is also compatible.

Please see also the processor-specific boot loader document for full details of using it with the µTasker boot-loader - http://www.utasker.com/docs/documentation.html

# 8. Encryption option

Often it is undesirable that the software which is up be updated to the remote device is available in a readable form. In order to make it difficult for the program content to be interpreted, the Boot Loader supports also an optional encryption/decryption function. Whether encryption is used or not is defined in the project setup and also in the use of the conversion utility.

When the conversion utility performs encryption it has the following use:

```
uTaskerConvert uTasker_demo uTasker_update.bin -0x1234
-a748b6531124 -ab627735ad192b3561524512 -17cc - f109
```

The additional parameters cause the encryption step to be performed.

`ab627735ad192b3561524512` is an encryption key which is used to transform the data content. It must have a length dividable by 4) and its length determines the strength of the coding.

`17cc` is used to prime a pseudo-random number generator used during the process (should not be zero) which must also match.

`F109` is a shift value in the code which makes it much more difficult to break using brute-force techniques. Without this shift it would be much easier to match known code patterns at the start of the file. Since the start code can be anywhere in the data this avoids this possible weakness.

The header added to the upload file is increased slightly in length due to the need for a second CRC.

```
unsigned long  ulCodeLength;
unsigned short usMagicNumber;
unsigned short usCRC16;
unsigned short usRAWCRC;
```

In this case usCRC16 is the check sum of the encrypted file (as it is stored during the upload) and usRAWCRC is the check sum of the real code (before encryption) so that successful decryption can also be verified.

The decryption process is an additional step in the Boot Loader which is performed when the code is copied to its executable position in FLASH.

It is advisable to always use a different magic numbers for projects with and without encryption. This ensures that encrypted data will never be copied to its executable location by a project without decryption support.

The encryption method can be used with both internal FLASH and also external SPI FLASH.

When `ADAPTABLE_PARAMETERS` is enabled the "Bare-Minimum" Boot Loader uses the same encryption settings for both secondary loader and application codes.

# 9. Verifying Operation with the µTasker Simulator

A common problem when starting with the "Bare-minimum" boot loader is that the upload works but the new code is not programmed. This happens when the configuration of the boot loader doesn't match the configuration of the upload file, but finding exactly what is not matching can sometimes be a bit of a challenge - the boot loader configuration may not automatically match the chip being used or the file system being used in a project and some adjustments may be required.

Note that a good indication of mismatch between boot loader and upload file generally results in the upload working but the boot software starting the old application immediately. This is because it sees that there is new code available but ignores it since it is not valid for its configuration. When the application runs it is possible to see this uploaded code in the file system (eg. by performing an FTP DIR). After a successful upload the boot loader will otherwise delete the upload image.

Using the uTasker simulator allows for comfortable testing of the upload sequence, allowing any problems to be quickly identified, corrected and verified. The following guide is a step-by-step on how this can be performed:

1) Run the uTaskerV1.4 application in the uTasker simulator. Make sure that the web pages are loaded and that the upload form is ready on the "Admin" web side.

2) Compile a new application (cross compile for real target) and convert it ready for an upload (using `uTaskerConvert.exe`). This will be a file called `z_upload.bin` or similar.

3) On the Admin web page perform an upload of this file. It will take about 1s to upload and a web page should appear confirming that all went well, after which the simulator will display a reset window. This means that the simulator has detected a commanded reset and will terminate - click OK.

4) In the simulator directory the content of memory (including the newly uploaded software file) will have been saved in the file `FLASH_M5223X.ini` (name depends on the processor being simulated). This is the internal FLASH content. If working with the file system in external memory (or partly in external memory) there will also be another file called something like `AT45DBXXX.ini` (the name depends on the device being used). These can also be opened in a binary editor to check the content and the position of data in the memory.

5) Copy this/these file(s) from the `uTaskerV1.4` simulation directory to the `uTaskerBoot` simulation directory. Make sure that there is a break point set at the start of the routine `uTaskerBoot()` in `uTaskerBootLoader.c` and start the simulator.

6)* The simulator will immediately stop at the break point and the code can be stepped through. It will enter the routine `fnCheckNewCode()` where the length of the SW file will be collected using the call "`uGetFileLength(UPLOAD_FILE_LOCATION)`". Step into the function and check that the value of `UPLOAD_FILE_LOCATION` corresponds to the address where the uploaded file is expected to be. Note that the simulator has loaded the memory as saved by the application and so is now working with the uploaded data from step 3.

7)* Also the upload header will be fetched with "`fnGetPars(UPLOAD_FILE_LOCATION + FILE_HEADER, (unsigned char*)file_header, SIZE_OF_UPLOAD_HEADER)`". By displaying the struct file_header the upload header can be check against that which is expected (check for example that the magic number matches the one expected from the

conversion utility parameters used to generate the upload file).

8 ) Step further to see whether any parameters don't match with local settings: the magic number accepted; the maximum code length accepted; finally the checksum calculated over the code can be verified "`return (usCRC == file_header->usCRC)`". If something doesn't match it should be quite easy to see what it is; the boot project and conversion parameters (application start address, encryption mode, magic number, secret key, etc.) must match between boot project and application upload file.

Using this technique it is usually quite easy to identify mismatches and then confirm the corrections so that it will work correctly on the target at the next attempt.

The following video shows the "Bare-Minimum" Boot Loader in operation and also the simulator being used to verify its operation: http://youtu.be/e7JvWodljlc

*) Exact code may change with settings. See also the following chapter.

# 10.      Options

Over time the μTasker "Bare-Minimum" Boot Loader has been further developed to ensure usability in various environments on a large number of devices from different families as well as to ensure compatibility in specific circumstances, This section discusses a few options that can be used control specific behavior.

```
#define ADAPTABLE_PARAMETERS    // support secondary bootloader uploads as well as
                                   application uploads
```

The option allows two different codes to be loaded to the intermediate area, each with its own magic numbers and individual authentication. Usually this is used to allow the operating application to prepare new serial loader or application images which, when correctly identified are updated to their respective locations.

```
#define NO_UFILE_HEADER         // the uploaded file has no uFileSystem header
```

Usually the application stores new code as a uFileSyste file with its corresponding file header, which is then respected by the "BM" loader. When this define is set it means that the appliation stores the new code without a uFileSytem header; the "BM" loader then handles it without expecting the uFileSystem header to be present.

```
#define AUTOMATE_UFILE_HEADER   // automatically ensure operation with or without
                                   uFileSystem header
```

When this define is enabled (it overrides NO_UFILE_HEADER) the image at the intermediate storage location an be saved by the application either with or without a uFileSystem header; the "BM" loader tries both formats and detects and of the two so that that the application is free to save with or without, or mix.

## 11.    Conclusion

This document has described in detail the operation and use of the "bare-minimum" method as implemented by the µTasker boot-loader.

The advantage of this method is that the boot code typically occupies only about one FLASH sector (eg. 2k) and still allows safe software uploads over the Internet. The use of SPI Flash as intermediate storage option and encryption has also been discussed.

The `ADAPTABLE_PARAMETERS` also enables the "Bare-Minimum" Boot Loader to be used together with a secondary loader to allow updates of the secondary loader and also application code. Details about the µTasker serial loader (secondary loader) can be found at http://www.utasker.com/docs/uTasker/uTaskerSerialLoader.PDF

Modifications

18.11.2006-0.02 Secret key added

25.08.2007-0.03 Encryption added. External SPI FLASH support capability added.

20.07.2012-0.04 New layout with cover sheet; added ADAPTABLE_PARAMETERS mode and flow diagrams as well as a chapter about verifying operation using the µTasker simulator

20.06.2020-0.05 Added build options chapter

## Appendix A – Flow Diagram of "Bare-Minimum" Boot Loader Operation

"Bare-Minimum" Boot Loader

START (reset)

Read header from intermediate storage area

Magic number matches application ? — **Yes**

Code length matches file length and not too long for destination area ? — **No**

**Yes**

Calculate CRC-16 of code content (optionally including a secret key)

CRC correct? — **No**

**Yes**

Code length matches file length and not too long for destination area ? — **No**

**Yes**

Delete destination area

Copy code from intermediate area to destination area (optionally encrypting content at same time)

Calculate CRC-16 of copied code (optionally including a secret key)

CRC correct? — **No**

**Yes**

Delete source file in intermediate storage area

Reset board

Jump to Application

"Bare-Minimum" Boot Loader for Secondary Loader and Application
(Secondary Loader and Application have different magic numbers and
secret keys but share optional encryption keys)

START (reset)

Read header from intermediate storage area

Magic number matches application? — Yes → Remember type, application size and application location

No

Magic number matches secondary loader? — Yes → Remember type, secondary loader size and location

No

Code length matches file length and not too long for destination area? — No

Yes

Calculate CRC-16 of code content (optionally including a secret key)

CRC correct? — No

Yes

Code length matches file length and not too long for destination area? — No

Yes

Delete destination area

Copy code from intermediate area to destination area (optionally encrypting content at same time)

Calculate CRC-16 of copied code (optionally including a secret key)

CRC correct? — No

Yes

Delete source file in intermediate storage area

Reset board

Jump to secondary loader