Embedding it better...

µTasker Document

µTasker – Serial Loader User's Guide (USB-MSD Host/Device, SD card, KBOOT, AN2295 Developer's Serial Bootloader, Modbus Slave, I^{2}C Slave and Ethernet Web Server/FTP Server)

Table of Contents

1. Introduction

Often there is the requirement to be able to install a small loader software to a board, which then allows application software to be loaded via a serial port, USB or other such methods. Furthermore, the loader should enable the application software to be deleted on request and the programming of further versions as required.

Some processors include a pre-installed loader of this nature which usually works together with a special purpose software tool, thus avoiding the requirement for the initial programming of this loader via an interface such as JTAG. However the majority of processors do not offer this inbuilt option or may not support the peripheral or method as required in a particular application, which is where the ***µTasker Serial Loader*** can find practical and important use.

The ***µTasker Serial Loader*** can be configured for a number of firmware loading methods, whereby one or more are usually possible to allow additional flexibility where needed.

1.1 SREC Loader

This technique allows new firmware to be loaded to the board in Motorola S-record format via a UART – *this doesn't require a special software to be installed on the PC but instead uses a general purpose terminal emulator.*

1.2 AN2295 Developer's Serial Bootloader

This mode allows compatible loading via UART based on Freescale/NXP's well known developer's serial bootloader [https://www.nxp.com/docs/en/application-note/AN2295.pdf], which works together with a PC program that accepts an SREC file and communicates only the minimum amount of information to allow the processor to be able to program the raw content to its program Flash.

1.3 USB-MSD Device Operation

The ***µTasker Serial Loader*** includes USB MSD (mass storage device) device mode of operation when the processor has a built-in USB device interface. The USB-MSD support can be used parallel to, or instead of, other loading techniques.

USB-MSD mode is practical due to the fact that all PC operating systems include a standard MSD host driver which allows such devices to be seen by the PC as an external disk drive. This means that no special installation is required and drag-and-drop control makes its operation very comfortable. Furthermore, the ***µTasker*** implementation retains the name and date of the original file which can aid in version management.

USB-MSD device and host modes can be used together to achieve an OTG (On-The-Go) type device. See USB-MSD Host mode below.

1.4 USB-MSD Host Operation

The ***µTasker Serial Loader*** includes USB MSD (mass storage device) host mode of operation when the processor has a built-in USB host interface. The USB-MSD support can be used parallel to, or instead of, other loading techniques.

This allows the processor to mount a connected memory stick and update its own firmware if the memory stick contains a valid file.

The operation of the USB-MSD Host loader is equivalent to SD-card loader (see the SD card details below) operation once the memory stick has been mounted.

USB-MSD device and host modes can be used together to achieve an OTG (On-The-Go) type device. See USB-MSD Device mode above.

1.5 USB-HID

This allows connection to a PC via USB and uploading firmware using a dedicated PC program; options are "`HIDloader.exe`" from Freescale/NXP AN4764 or Freescale KBOOT compatible loading. The USB-HID mode can be combined with USB-MSD as a composite device for maximum flexibility.

Details for Freescale/NXP's KBOOT can be found at
https://www.nxp.com/support/developer-resources/reference-designs/kinetis-bootloader:KBOOT

1.6 KBOOT UART

Freescale/NXP KBOOT compatible loading via UART.

Details for Freescale/NXP's KBOOT can be found at
https://www.nxp.com/support/developer-resources/reference-designs/kinetis-bootloader:KBOOT

1.7 SD Card Operation

The ***µTasker Serial Loader*** has been extended to include SD card mode of operation which can be used by all processors with SPI or SDIO and an SD card or µSD card slot for the card. FAT16 and FAT32 file systems are supported, allowing a pre-defined file on the card to be used as source for new embedded code.

The original firmware file can be optionally automatically deleted after a successful operation.

1.8 Ethernet Web Server

This option is available for boards with Ethernet and allows firmware uploads from a standard Web browser.

1.9 FTP Server

This option is available for boards with Ethernet and allows firmware uploads from and FTP client.

1.10 Modbus Slave

This option is available over UART (ASCII or RTU) and allows a Modbus master to load new firmware using a combination of commands and writes to a defined area of the Modbus register map.

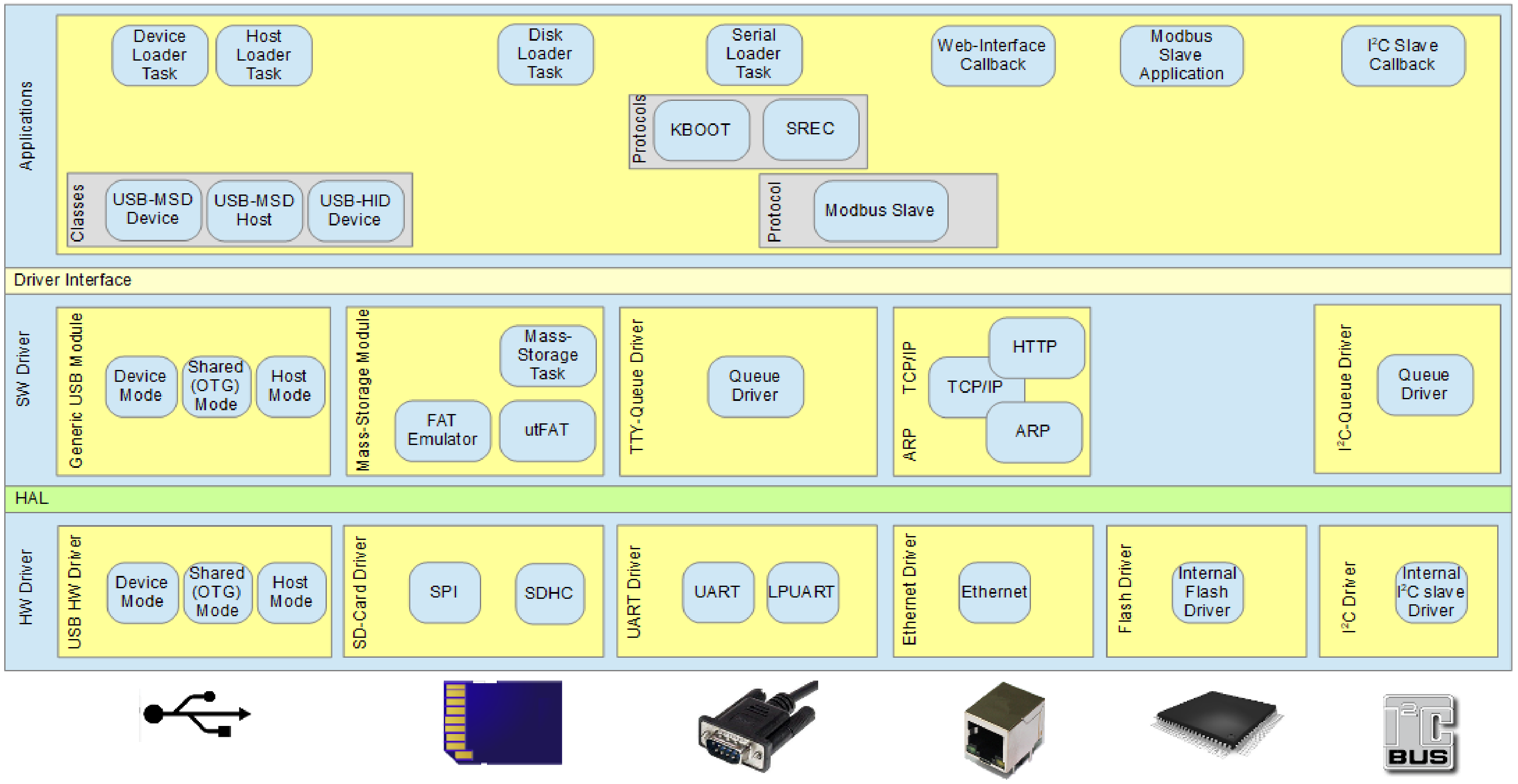
1.11 I^{2}C Loader

This option is available for boards with I^{2}C slave capability and allows firmware uploads from an I^{2}C master.

The only change required in the application software to be able to work with the ***µTasker Serial Loader*** is the linking of its start address to match the application start address as configured in the loader.

See appendix A for some target and compiler specific details.

The ***µTasker Serial Loader*** can also work together with applications from other projects. For details about ensuring that the application can operate correctly see appendix B.



µTasker Serial Loader – Major Components

2. Programming the µTasker Serial Loader

The ***µTasker Serial Loader*** is available as pre-compiled object for loading to many typical demo and evaluation boards. See the µTasker demo software web page to check whether your board is supported: http://www.utasker.com/SW_Demos.html

The object file can be programmed using standard programming tools for the target.

Since the objects are defined for standard configurations there may be some restrictions (like the UART that it uses and the UART configuration). By building the ***µTasker Serial Loader*** (*see following section*) the user is free to define all settings to suit a particular hardware and project.

This is valid for serial, USB-MSD, USB-HID, SD card and Ethernet Web Server modes.

3. Building the µTasker Serial Loader

Users of the µTasker project receive a serial loader project in the package which can be re-configured as required and thus built to suit a particular target or project.

To build the project, simply open the serial loader project in the target development environment, chose the desired target and build.

The following is a typical project path, in this case specifically for a Codewarrior project for an M5223X target:

```
\
Applications\uTaskerSerialBoot\CodeWarrior_M5223X\uTaskerSerialBoot\
uTaskerSerialBoot_CW7.mcp
```

The build process should neither generate errors nor warnings and results in an output file such as `uTaskerSerialLoader.elf`. (*some targets will generate .bin, .hex etc.*)

This file can then be programmed to the target.

The following section describes the use of the ***µTasker Serial Loader*** in its standard configuration. In subsequent sections typical settings are discussed, which enable the behaviour of the project to be changed to suit particular uses. Such details include the selection of the UART to be used, its baud rate and the input(s) which may be used to force the serial loader mode.

4. Using the µTasker Serial Loader - SREC

In order to be able to work with the ***µTasker Serial Loader*** it is necessary to connect a terminal emulator to the UART on the target which is configured to perform the serial loader function. It is advised to use `TeraTermPro`, a free terminal emulator which can be downloaded from http://www.uTasker.com/software/teratermpro.zip (newer versions are available at the Tera Trm Home Page - http://ttssh2.sourceforge.jp/index.html.en) This is an excellent program which has proved to be very reliable on various Windows platforms, including Vista, Windows 7 and Windows 8.1.

The terminal emulator must be programmed to match the setting of the ***µTasker Serial Loader*** (eg. 115200 Baud, 8 Bit, 1 Stop bit, no parity, **XON/XOFF** flow control).

When the ***µTasker Serial Loader*** has been programmed to the target and no application software is loaded it will always start and, when powered (or reset), display a screen similar to the following:

```
uTasker Serial Loader
=====================
[0x00008080/0x000287ff]
me = mass erase
bc = blank check
dc = delete code
ld = start load
go = start application
>
```

This screen can be requested again by typing in a question mark (?).

There are several commands available – these commands are case-insensitive.

A command is terminated by the ENTER key and invalid input can be deleted by using the back-space key. The terminal emulator may be configured to send a carriage return or a carriage return + line feed.

`me = mass erase`

This is an optional command but is useful when working with a secured chip. A secured chip will not allow debugging (*to protect the content of FLASH from being read*) and can usually only be recovered by performing a mass erase of the FLASH. This erase also deletes the ***µTasker Serial Loader***, but un-secures the device. *It should only be used in special cases and requires the user to answer positively to two questions when executed, to prevent its unintended execution.*

`bc = blank check`

The blank check verifies that the complete application FLASH area is deleted. Only when this is the case should a new load be performed; attempting to load to non-deleted FLASH will cause a failure to be declared and the sequence must then be restarted. If the FLASH is indeed blank the test will confirm this with the message

```
Checking Flash...  EMPTY!
```

A non-blank FLASH will cause the following message to be displayed (the address of the offending data may also be displayed):

```
Checking Flash...  NOT BLANK!
```

`dc = delete code`

Delete the application area in the FLASH. This should be performed *before* a load is started but is only necessary if there is already application code programmed, as declared by the result of the blank check.

```
> dc
Delete code [y/n] ? >

Deleting code...successful
```

Note that the user must confirm the delete before it is executed.

`ld = start load`

Starts the S-REC loading process. When this command is executed the ***µTasker Serial Loader*** will wait for S-REC tokens to be received and save these to the appropriate area of application FLASH. The following is a typical sequence:

```
uTasker Serial Loader
=====================
[0x00008080/0x000287ff]
me = mass erase
bc = blank check
dc = delete code
ld = start load
go = start application
> bc
Checking Flash...  EMPTY!
> ld
Please start S-REC download:
```

Now the user should start the transfer of the appropriate file. Using `TeraTermPro` this is very easy since it can be simply dragged to the terminal emulator window (*drag-and-drop*) and then continues automatically.

```
Please start S-REC
download: ....................................................
..............................................................
..............................................................
..............................................................
..............................................................
..............................................................
..............................................................
..............................................................
..............................................................
..............................................................
..............................................................
..............................................................
..............................................................
..............................................................
..............................................................
.................................................

Terminated - restarting...
```

During the download each received S-REC token is displayed as a dot. Once the complete file has been received the message "`Terminated - restarting...`" is displayed and the target board will reset. After the reset the loaded application will run immediately.

Note that if a download is interrupted the mode can be quit by using `CTRL + c`.

In case of loading errors (invalid S-REC, or program code which is declared for outside of the application space) the loading will be terminated with an error:

```
SREC-error!! (Ctrl+r to reset)
```

`CTRL + r` will reset the board so that the loading can be attempted again.

Note also that the application will not be recognised in this case due to the fact that the first application bytes are always programmed as very last operation, once all other bytes of the code have successfully been programmed. The ***µTasker Serial Loader*** will recognise that there is no application loaded and start the serial loader again if the first 8 bytes of application code do not exist.

`go = start application`

This command can be used to jump to the application code. This will be performed whether there is application code loaded or not. It is useful when the serial loader mode has been forced and the application code should however be started, as well as for other test purposes.

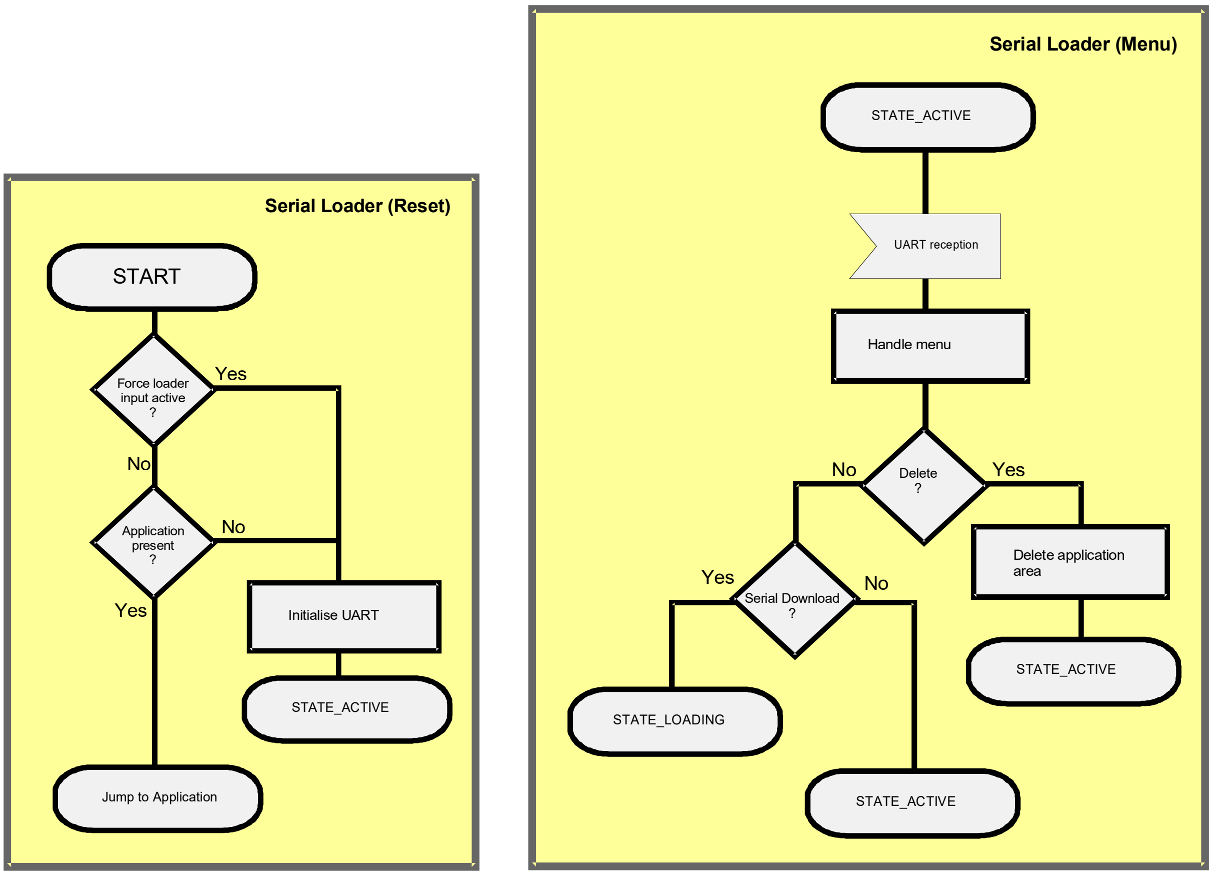
Should the following, or similar, be seen during the loading process it means that the address in the S-REC file is conflicting with the serial loader's own code area (signalled by each !) and each S-REC line has been ignored:

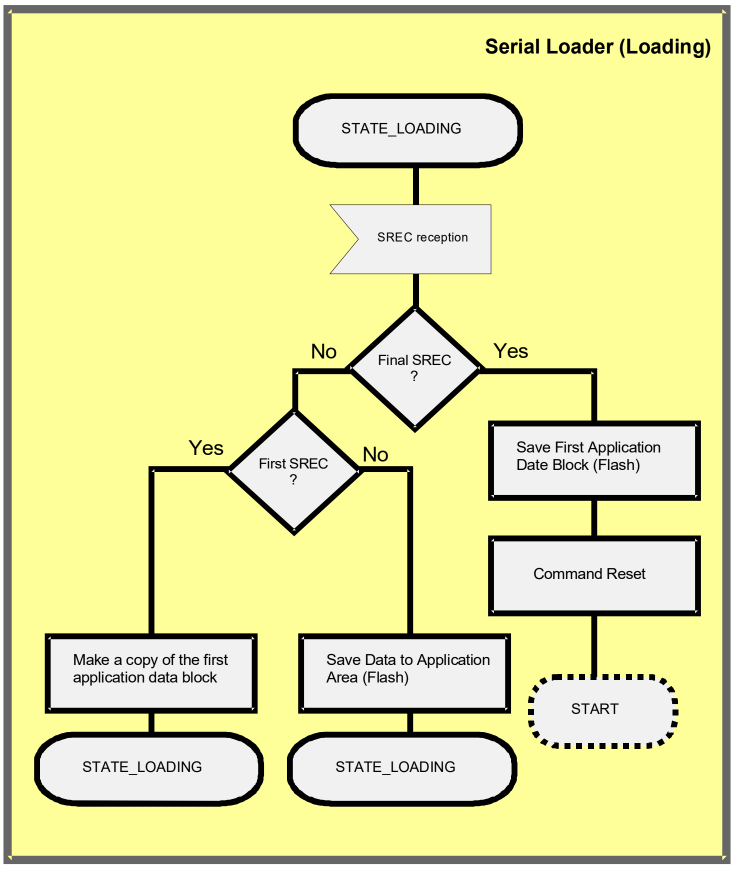
```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!........................
.............................................................
```

Possibly the normal ….. are seen after a number of !!!!!! as the S-REC addresses advance to the valid application area.

In such a case it means that the application has not been configured to start at the correct address location and needs to be corrected before continuing. Loaded code is also invalid and so will not run – *the serial loader should be forced to start again and the invalid code deleted*.

A simplified state-event diagram of the serial loader operation is shown below for the start-up loader decisions, the menu and download operations.





Configuring the µTasker Serial Loader Project

The following settings control the project configuration.

`Config.h`

The main target is defined here as in all µTasker projects.
For SREC mode of operation the define `SERIAL_INTERFACE` must be **enabled** and the defines `KBOOT_LOADER and DEVELOPERS_LOADER` must be ***disabled***.

`app_hw_xxxx.h` (*hardware specific configuration file*)

The CPU speed (eg. PLL) is determined in this file as well as details of the hardware target, such as package used.

Most hardware options (like timer, ADC, DMA etc.) have been removed since they are not used by the serial loader. Whether the µTasker Serial Loader operates with active watchdog can be configured with the defines `INIT_WATCHDOG_DISABLE()` and `WATCHDOG_DISABLE()` [`INIT_WATCHDOG_DISABLE()` can be used to configure an input to check whether the watchdog should be enabled or disabled; `WATCHDOG_DISABLE()` configures the decision check itself – set 1 to always disable the watchdog]. *These settings are equivalent to the settings in the main µTasker project*.

`INIT_WATCHDOG_LED()` defines a port as output to be used to flash an LED at the watchdog rate (5Hz) during the serial loader operation. The toggling of the port (often connected to an LED for visualisation) is defined by `TOGGLE_WATCHDOG_LED()`. [*Note that the µTasker application flashes the LED at 2.5Hz and so the serial boot loader phase can be distinguished from the application phase by the blink speed*].

`FORCE_BOOT()` is a check to decide whether the serial loader mode should be forced, whereby its configuration can be usually combined using the `INIT_WATCHDOG_DISABLE()` define. This will allow the serial loader mode to be entered even if there is an application program loaded, which may be important in case of there being an application in memory which has a serious bug and otherwise doesn't allow the mode to be resumed.

UART buffer space:

```
#define TX_BUFFER_SIZE   (512) // the size of RS232 input and
                                                output buffers
#define RX_BUFFER_SIZE   (512)
```

Note that the output buffer size should be adequate to contain the complete menu listing. The input buffer should be set to a value adequately buffering the UART while FLASH is being programmed. If the buffer does become more that 80% full the XON/XOFF protocol will ensure that the terminal emulator program stops sending further data for a short time.

Loader.h

Contains the following configuration defines regarding UART characteristics and application FLASH space:

```
#define MASS_ERASE    // support a mass-erase command. This is used together
                         with a protected FLASH configuration. When the
                         FLASH is protected, downloads are still possible
                         but the debug interface is blocked. This allows a
                         commanded delete of the complete FLASH content
                         (including serial loader) to unblock the debug
                         interface


#define LOADER_UART  0     // the serial interface used by the serial loader

#define SERIAL_INTERFACE_MODE (CHAR_8 + NO_PARITY + ONE_STOP + USE_XON_OFF +
CHAR_MODE)
                          // the UART configuration

#define SERIAL_SPEED    SERIAL_BAUD_115200     // the Baud rate of the UART


#define UTASKER_APP_START  (8 * 1024) // application starts at this address

#define UTASKER_APP_END   (unsigned char *)(UTASKER_APP_START +
                           ((256 - 8) * 1024)) // end of application space
```

The example above shows UART 0 being used at 115'200 Baud, 8 bits, no parity, one stop but and XON/XOFF flow control. The µTasker Serial Loader ass reserved 8k space (the actual space required depends on processor type and compiler) and the rest of a 256k FLASH memory is allowed to be used by the application program. To match this example the application should be linked to match the start address (8k) and can be up to 248k in size. In some cases there may be additional parameters maintained in FLASH and so the application space can be reduced to avoid these being deleted when the application code is erased.

When the SREC loader is used together with USB-MSD a file called "SOFTWARE.S19" is visible on the hard drive after firmware has been loaded. This file has the size of the code but a fixed time/date.

5. Intermediate Buffer Required for Certain Processor Types

Not all processors enable programming of each individual line of SREC code as it is received. The reasons may be restrictions concerning the smallest block size that can be programmed to FLASH at each time or the fact that the programming time for the content of the smallest block is too long for further SREC data to be received without UART reception overrun taking place. In these instances an intermediate buffer is required.

When the processor requires the use of an intermediate buffer this will be automatically activated in the project for this processor:

```
    #define INTERMEDIATE_PROG_BUFFER  (16 * 1024)
```

The size of the buffer is also defined and is chosen to be suitable for the FLASH type and memory available. Its job is to store the binary data (after extraction from each SREC line) and assumes that the SREC being received is constructed in a linear fashion (addresses increment continuously*). Once this buffer becomes full it will be programmed to FLASH.

Although the programming of a large block of data is an efficient method, the programming time can be quite long. Usually interrupts are also disabled during programming and so no further SREC lines can be received during this interval. For this reason the reception flow is first halted by sending an XOFF character to the terminal, <u>which is expected to be configured for XON/XOFF flow control operation</u>. Since terminal emulators may take a short time to react to the XOFF character, a timer of 0.2s is used before the programming actually commences; during this time any SREC reception data is still accepted by the UART receiver but is held in the input buffer and not yet decoded into binary data.

The block programming time is not critical since data flow has been stopped. Once the block has been successfully committed to memory the download continues after an XON character is sent, informing the terminal emulator that it may continue with the date transfer.

During large SREC downloads the buffer programming may repeat several times. Once the end of the SREC is detected the last buffer is committed, followed by the first data block, so that the new application becomes ready for use after the following automatic reset.

The following shows a typical download with an intermediate buffer of about 16k in size. When the XOFF is send before buffer programming starts a * is seen. When the XON is sent to continue with the data transfer a second * is sent.

```
Please start S-REC download:.......................................................
...................................................................................
...................................................................................
...................................................................................
...................................................................................
...................................................................................
...................................................................................
...................................................................................
...................................................................................
...................................................................................
..........................................................................**......
...................................................................................
...............................................................

Terminated - restarting...
```

*Small 'holes' in the SREC can be tolerated; these are filled with 0xff in the intermediate buffer. If however larger holes are detected (larger than 100 bytes) which occur towards the end of the intermediate buffer it may not be possible to fill them. In this case the download will be aborted with the error message "`SREC hole!! (Ctrl+r` `to reset)`". Either the value INTERMEDIATE_BUFFER_RESERVE needs to be increased to reserve more space at the end of the buffer to cope for larger holes, or else the SREC file should be filled out with lines of 0xff content rather than the spaces.

6. Preparing an Application and working with the Serial Loader

Any application can be loaded as long as its object file is in S-Record format. Its start address (link address) must be set to correspond to the setting in the µTasker Serial Loader (`UTASKER_APP_START`); the code at this address will be started as if it were at the normal reset vector location.

If the serial loader is not forced (*force input not activated*) the application software will always be started if it is programmed in FLASH; the first 8 bytes at its start location are checked to decide whether to start it or not (the first 4 bytes are however programmed as last task during loading and partly loaded programs will thus never be recognised as valid).

If new code is to be loaded there are two possibilities to enter the µTasker Serial Loader: Either the defined input is used to force the mode after a reset or else the application can delete itself (the first sector in the application area is adequate) and restart the board (usually with help of the watchdog). Since there is subsequently no valid application detected by the serial loader it will then enter the serial loader mode, allowing further downloads.

See appendix B for a list of points to consider to ensure application compatibility when working with the boot loader. The application programmer may need to consider the following points to ensure full compatibility with SREC loader:

1. If the application is started via the "go" command the UART and its pins will be configured. Although this case is an exception, the application should not assume the state of the peripheral at reset and may choose to reset these if appropriate.

See appendix A for target and compiler specific details.

7. USB-MSD Device Boot Loader

The USB-MSD boot loader operation is available when the processor has an in-built USB device interface. This mode is enabled by setting the define `USB_INTERFACE` in `config.h` as well as `USB_MSD_DEVICE_LOADER` and can operate in parallel with other boot loader modes – see the appropriate chapters for details concerning configuring other modes.

The default format of the firmware to be updated is binary. However the USB-MSD loader also supports SREC and Intel HEX formats as options. To enable SREC operation the define `USB_MSD_ACCEPTS_SREC_FILES` can be enable and to enable HEX operation `USB_MSD_ACCEPTS_HEX_FILES`. The USB-MSD loader will then recognise the format that it is receiving and perform the loading appropriately. If no binary mode support is required it can also be disabled using `USB_MSD_REJECTS_BINARY_FILES`.

When the USB interface is enabled, the application file `usb_device_loader.c` is used to control a USB Mass-Storage-Device class interface, which allows the PC host to see the FLASH space as an external disk drive. The application code works together with `USB_drv.c` and the HW USB driver to emulate the disk, allowing software to be loaded, viewed and deleted. Optionally, loaded software can also be copied back to the PC host, which can furthermore be password protected if this is not to be made available generally.

The USB-MSD interface emulates a FAT12 file system. The reason for choosing FAT12 is the fact that it is the preferred FAT file system for disk space up to 2MByte; Windows XP will, for example, not allow FAT32 operation on smaller disks than 32MByte and will also automatically switch from FAT16 to FAT12 for disk sizes less than 2MByte. Although it would be possible to work with FAT32 with most modern PC operating systems, the requirement for compatibility with Windows XP meant that only FAT12 was realistic.

The operation of the USB-MSD boot loader is shown in two videos:

http://www.youtube.com/watch?v=H4TYM9jY2-g

> The first video shows the basic operation, involving connecting to the PC host, viewing loaded software, deleting existing software and uploading new software.

http://www.youtube.com/watch?v=e4oFBn_M5wo

> The second video shows the optional password protection of the copy of loaded software back to the PC host.

Note that the USB-MSD Device mode of operation can also be combined with USB-MSD Host mode – see the USB-MSD Host Bootloader chapter for more details.

Since the actual use of the USB-MSD boot loader is fairly simple and fully illustrated in the videos the following discussion will concentrate on SW details.

In `Loader.h` there are three defines that configure the USB-MSD boot loader operation:

```
#define ROOT_FILE_ENTRIES         4     // when USB MSD loader, this many directory
                                           entries are set to the start of FLASH -
                                           the application start is shifted by this
                                           amount x 32 bytes

#define ENABLE_READBACK                // allow USB to transfer present application to PC

#define READ_PASSWORD         "enable file read from the device by dragging this file
to the disk"                           // password with maximum length of 512 bytes
```

`ROOT_FILE_ENTRIES` defines the amount of space reserved at the start of the application FLASH area for saving file entries belonging to the file used to load the software. A value of 4 reserves 128 bytes of space (0x80) which is adequate to save the file details up to a long file name of 39 bytes (based on the fact that LFN saves 13 uni-code characters in each file entry space, with the final entry holding a DOS compatible 8:3 name). If the file name were to be restricted to a short file name (8:3 format) a single file entry would be adequate, but this may be too restrictive for general use. In case of the requirement for very long file names the number of entries can be set higher and extra Flash space will be reserved (20 would hold the longest LFN possible but would require 640 bytes to do so). By saving the file entry it is possible to display the file details when the file is viewed – this makes for simple software version management since the original file automatically has its creation date and time as well as its size and software name.

In case the name of the uploaded software is too large to fit in the reserved file object area its long file name is not used and instead only its short file name saved. This means that an over-long file name will appear as something like `UTASKE~1.BIN`, although its size and date/time information are still correct.

The root file entry also makes it simple to display the software file and its details since the FAT emulation simply needs to recognise when the USB host is requesting the content of the root directory (identified by the sector address that is being requested) and return the same details that were saved as part of the software upload.

The size of the area reserved for root file entries also has a consequence on the linking address of the application data. For example, assuming that the boot software requires about 16k of space and the application data can then start at the address 16k, the application link address is in fact 16k plus the size of the root file entries:

```
#define UTASKER_APP_START    (16 * 1024) // application starts at this address

#define ROOT_FILE_ENTRIES    4  // when USB MSD loader, this many directory entries
                                   are set to the start of FLASH - the application
                                   start is shifted by this amount x 32 bytes
```

The application jump address is in fact 0x4080 (0x4000 + 4x0x20) – this is the address at which the application is linked to start at.

Note that the application jump address for the SREC loader alone would be 0x4000. The jump address for the USB-MSD loader is therefore not the same at 0x4080. If, however, both USB-MSD loader and SREC USB-MSD loader are used in parallel 0x4080 is also required so that both are compatible.

`ENABLE_READBACK` activates support for copying back software content from the embedded processor to the USB host. If this is not enabled the file will be visible but attempts to copy it to the PC will fail with an error message suggesting the file is no longer available.

`READ_PASSWORD` is a password string of up to 512 bytes in length. When it is enabled and `ENABLE_READBACK` is also active, it will be possible to copy software back to the PC host but its content will be filled out with zeros by default. The reason for this behaviour is to prohibit the software content to be retrieved by non-authorised users (often the processor will also be set to secured mode so that the content can also not be retrieved via debug interfaces).

When an authorised user copies a password file (a simple text file containing the same string as the password) to the disk drive (see the second video) uploads are then enabled until the next reset of the device takes place. The authorised user can then copy the software stored in the device's FLASH back to the USB host with its same name, details and content.

Finally note that the USB-MSD boot loader works with *binary files* and not SRECs. This means that the linker may need to be configured to generate binary output. In the case of linkers generating MOTOROLA binary output this can be converted to RAW binary by using the `uTaskerConvert` utility. The following shows it being used to generate raw binary from a MOTOROLA binary input called `uTasker_BM.bin`:

```
uTaskerConvert uTasker_BM.bin raw.bin -b
```

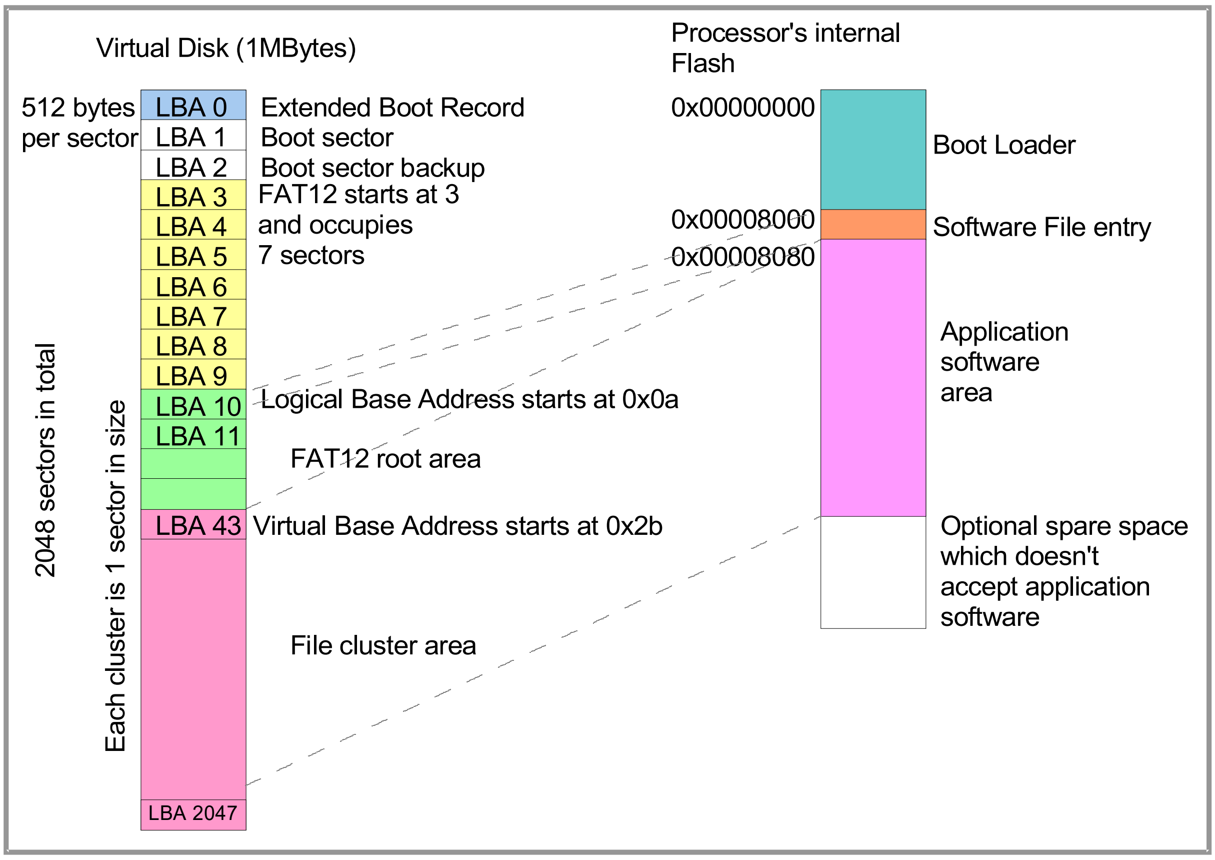
7.1 USB MSD Device Implementation Details

In order to work as a disk drive to a PC host the serial loader needs to behave as if it were such a disk drive, although its main goal is to allow software to be written into its internal flash memory. When the PC host connects to the serial loader via USB the serial loader informs that it is a USB-MSD class and gives information so that the PC host believes that it is a disk drive. The following main class commands are handled:

- *SCSI command Inquiry* (0x12). Each time this is requested by the PC host the serial loader returns fixed details about it being a removable medium, with vendor "uTasker" and product type "USB MSD Loader"
- *SCSI command Read Format Capabilities* (0x23). Each time this is requested by the PC host the serial loader returns fixed details about the virtual disk status, size and block length. The size is defined by `DISK_SIZE`, which can be up to 2Mbytes in size using FAT12 format. The size is usually set to reflect the application flash size, although larger size declaration is no problem.
- *SCSI command Read Capacity* (0x25). Each time this is requested by the PC host the serial loader returns fixed details as to the total capacity of the disk; the number of blocks and the block size.
- *SCSI command Mode Sense (6)* (0x1a). Each time this is requested by the PC host the serial loader returns an error, unless it is a request for all pages in which case it informs that it is a floppy disk without any write protection.
- *SCSI command Request Sense* (0x03). Each time this is requested by the PC host the serial loader returns fixed details with the `Sense Key` set to "`No sense`".
- *SCSI command Test Unit Ready* (0x00). Each time this is requested by the PC host the serial loader returns an acknowledgement to inform that the disk is still operational.
- *SCSI command Read (10)* (0x28) or *Read (12)* (0xa8). When the PC host reads data from the disk the serial loader will mimic the contents of a formatted disk and return one of the following types of data content:
  - if the requested logical block address (LBA) is 0 it returns the content of a fixed extended boot record containing information about where the boot sector is located.
  - if the LBA corresponding to the location of the boot sector is requested (or its backup location) [`BOOT_SECTOR_LOCATION` or (`BOOT_SECTOR_LOCATION + BACKUP_ROOT_SECTOR)`] a fixed boot sector content will be returned. This contains various details about the FAT and a volume label "`UPLOAD_DISK`" which the host PC displays along with the disk.
  - if the logical block address corresponds to the logical base address or is greater but less that the virtual base address it means that the PC host is reading the root directory of the disk. If there is no software loaded this causes an empty root directory to be returned (containing just a volume label entry).
  If software is loaded a file object corresponding to the file is returned – *it will later be seen that this file object was written by the host PC when the software file was saved and the same entry is saved to the internal flash so that it can be returned*.
  - if the logical block address matches the start of the FAT area an empty FAT will be returned (with initial default cluster entries) as if no data were present on the disk. In case there is software loaded the FAT area sector read return FAT cluster information as if the program were located in a contiguous series of sectors starting from the virtual base address. This is important since the PC host reads the FAT area when connecting and will declare the file as corrupted (not allowing deletes or copies to take place correctly) if the FAT doesn't not contain corresponding cluster details

corresponding to the file's content.
- if the logical block address is equal to or larger than the virtual base address it causes zero filled sectors to be returned when there is no application software present or it causes data from the internal flash to be returned as it if were located in this area.
- all other logical block addresses that don't match the previous ones will cause zero filled sectors to be returned as would be the case from a deleted disk. This means that the size of the disk that is simulated by the interface can be larger than the physical size of the internal flash in the processor. Generally the disk size will be declared somewhat larger to account for the additional space occupied by the FAT, boot sectors and such.

- *SCSI command Write (10)* (0x2a) or *Write (12)* (0xaa). When the PC host writes data to the disk the serial loader will mimic the contents of a formatted disk and write content as follows:
  - if the logical block address is equal to the logical base address the PC host is writing to the root directory which takes place when deleting files or writing file information. The content of such writes is saved in memory for future use (it will be saved to internal flash). If there is application software present and the first file entry is deleted or its file length is set to zero it triggers a delete of the application area in internal flash.
  - if the logical block address is equal to or higher that the virtual base address it indicates that the PC host is writing file content and this content is written to internal flash.
  - if the writes are to other areas or to a logical block address above the highest address accepted by the application area it is ignored but still acknowledged so that the PC host doesn't see an error.



This diagram illustrates the relationship between the physical flash memory in the processor and the virtual disk drive that is emulated by the USB-MSD loader interface (*flash addresses may vary*). As can be seen, the first file object (representing the software file written by the host) is stored in the processor's memory just before the start of the application code so that it can be returned when the host reads the root directory of the emulated disk – this is then displayed as its original file name, with time, date and size. The size information is also used by the disk emulator to correctly construct the FAT area entries.

Typically the PC host will write the file information before actually writing the file content and so it is not possible to use the write of a file size to the file object to detect when the file itself has been fully written. Instead a re-triggerable mono-stable software timer is used to monitor writes to the file content area and when there has been no further write for a period of 3s it is assumed that the file transfer has terminated and the file object entry is also committed to flash (along with the first block of program) so that the new application is ready to be started. The board will reset at this point and start executing the new code.

Note that the existence of code at the first address in the application area is used to identify whether there is valid software available. The first application block is not written immediately when received but is buffered and only committed as final operation when the complete file has been received. This avoids an interrupted transfer from being understood as valid software in case the file transfer would not complete due to a power failure or other cause – this is the same strategy as used by the UART loader.

7.2 Software Password Protection

When a host write is performed to file cluster area although there is already software present it indicates that a file is being copied with a different name to the software file presently stored. This is interpreted as a "password file" write and the content is not stored. The content is however compared with a local password which can temporarily unlock code protection.

If password protection is enabled [`READ_PASSWORD`] reading a file back from the embedded system without the password being first entered results in the file content being returned as zero filled sectors. Once the password has been entered (by copying the file with the password as content) it will allow the file's data to be read back. *The password entry remains valid until the next reset*.

The capability of reading back the software content to the host is enabled by `ENABLE_READBACK`.

7.3 Software Overwriting

When a new software version is to be loaded which has a different file name it is necessary to first delete the existing software. If however the software to be loaded has the same name (with possibly different time/date, content and size) it can be copied to the disk without first deleting the original application software. This results in the PC asking first whether the file is to be replaced and the replace actual is equivalent to a delete followed by a write which may however be a more convenient in some cases.

7.4 Compatibility and Disclaimer

As PC operating systems develop and new upgrades and service packs become available it has been found that the interaction with disk drives has changed. For example, from Windows 8.1 the operating system writes hidden system volumes to newly connected drives which needs to be suitably handled by the FAT emulation so that it doesn't mistake these writes as parts of firmware upload data and also needs to be able to handle the fact that the file system layout becomes un-predicatable. *Such changes have been made to ensure such compatibility with the most recent Windows OS without requiring Windows settings to be changed*.

Similar behaviour is found with MAC and Linux OS versions and it can't be excluded that this behaviour will not change at any time after a new PC software update is performed.

For this reason it has been decided to maintain the ***µTasker Serial Loader*** for correct operation with Windows. This means that the USB-MSD mode of operation will be further developed if and when such need is detected, but it can't be guaranteed that a particular version which fulfils this aim today will remain fully compatible during its lifetime after installation.

Due to the diversity of other PC operating systems the correct operation with other systems apart from Windows is not guaranteed although attempts will be made to ensure that major systems are basically operational.

Therefore, the following maintenance goals and disclaimers are valid for use of the USB-MSD loader:

1. The correct operation with Windows XP, Vista, Windows 7, Windows 8 as well as enhancements required due to service pack updates is a part of service agreements for users of the ***µTasker Serial Loader***
2. Use of the ***µTasker Serial Loader*** in non-Windows environments is not guaranteed or maintained, although efforts will be made to achieve a good level of compatibility
3. Although efforts will be made to solve new compatibility issues if and when they arise, no responsibility is accepted for operational difficulties due to potential future changes in PC behaviour

8. USB-MSD Host Bootloader

Processors and boards supporting USB host can make use of the USB-MSD Host Bootloader mode. This allows loading firmware from an inserted memory stick. The operation is equivalent to the SD Card Bootloader which is described later in this document once the inserted memory stick has been enumerated and subsequently mounted.

In order to activate this mode the following defines are enabled in config.h:

```
#define USB_INTERFACE

#define USB_MSD_HOST_LOADER
```

The result is that the USB host loader task is enabled (`usb_host_loader.c`) which configures the USB interface for host mode of operation and implements the USB-MSD host control. Once the memory stick has been enumerated and its detaisl are known the rest of the operation is performed by the disk loader task (`disk_loader.c`) which is also used by the SD card loader mode. See the SD card description for the subsequent operating details.

It is possible to enable both USB-MSD Host and USB-MSD Device modes at the same time. This means that both `USB_MSD_DEVICE_LOADER` and `USB_MSD_HOST_LOADER` are enabled together. In this configuration the USB-MSD Device mode is initially configured and the USB-MSD device task waits for 0.5s to see whether there is a USB host connected that performs enumeration. Should this be the case it remains in the USB-MSD Device loader mode so that new firmware can be loaded using the method explained in the USB-MSD Device Loader chapter.

If, after 0.5s, no USB host has been detected the USB-MSD Host task is activated, which re-configures the USB interface to host mode and attempts to enumerate a connected memory stick (after applying USB power to the bus).

When the UART is enabled the USB-MSD Host activity can be monitored at its debug output. The following shows a typical session where a memory stick is mounted and a file found on it is used to update the firmware:

```
 uTasker Serial Loader
=======================
[0x00008080/0x00025fff]
bc = blank check
dc = delete code
ld = start load
go = start application
> Switching to host mode
USB FS device detected
USB device information ready:
USB2.0 device with 64 byte pipe
Vendor/Product = 0x0781/0x5406
Manufacturer = "SanDisk"
Product = "U3 Cruzer Micro"
Serial Number = "43172009D7514E7"

Bus-powered device (max. 100mA) with 1 interface(s)
Mass Storage Class : Sub-class = 0x06 interface protocol = 0x50
Endpoints:
1 = BULK IN with size 64
2 = BULK OUT with size 64
Enumerated (1)
LUN = 2
UFI INQUIRY -> Status transport - Passed
UFI REQUEST SENSE -> Status transport - Passed
UFI FORMAT CAP. -> Stall on EP-1
EP-1 cleared
Status transport - Failed
UFI FORMAT CAP. -> Stall on EP-1
EP-1 cleared
Status transport - Failed
UFI FORMAT CAP. -> Status transport - Passed
UFI READ CAP. -> Status transport - Passed
Mem-Stick mounting...
***Disk E mounted
Mem-Stick present
**********************************************File valid
**********************************************Software Updated
```

It is to be noted that some memory sticks may take a short time to become ready and return USB stalls when requests are made before it can respond. Some stalls and Status transport failures are thus normal for some memory sticks, while others may just not respond until they are ready to do so (can be one or two seconds but varies greatly between device model and manufacturer).

9. USB-HID

Freescale/NXP device users can make use of Freescale/NXP's KBOOT programming utility to load code via USB using USB-HID (human interface device)

In order to activate this mode the following defines are enabled in `config.h`:

```
#define USB_INTERFACE
#define HID_LOADER
#define KBOOT_HID_LOADER
```

This selects the USB interface, the USB-HID mode and specifies that its protocol is that used by KBOOT.

The utilisation is compatible with Freescale/NXP's KBOOT implementation and so all further information about its utilisation can be found at https://www.nxp.com/support/developer-resources/reference-designs/kinetis-bootloader:KBOOT

If the define `KBOOT_HID_LOADER` is removed the USB-HID protocol used will be compatible with an older Freescale/NXP loader called "`HIDloader.exe`" and described in the Freescale/NXP application note AN4764.
http://cache.freescale.com/files/32bit/doc/app_note/AN4764.pdf

The USB-HID mode can be used in parallel with USB-MSD, in which case a composite device is used.

When the USB-HID KBOOT loader is used together with USB-MSD a file called "`KBOOTUSB.BIN`" is visible on the hard drive after firmware has been loaded. This file has the size of the code but a fixed time/date.

10. KBOOT UART

Freescale/NXP device users can make use of Freescale/NXP's KBOOT programming utility to load code via UART.

In order to activate this mode the following defines are enabled in `config.h`

```
#define SERIAL_INTERFACE

#define KBOOT_LOADER
```

This selects the UART interface and specifies that the protocol is that used by KBOOT. *This mode overrides the SREC mode of operation.*

The utilisation is compatible with Freescale/NXP's KBOOT implementation and so all further information about its utilisation can be found at https://www.nxp.com/support/developer-resources/reference-designs/kinetis-bootloader:KBOOT

When the serial KBOOT loader is used together with USB-MSD a file called "`KBOOTSER.BIN`" is visible on the hard drive after firmware has been loaded. This file has the size of the code but a fixed time/date.

11. AN2295 Developer's Serial Bootloader

Freescale device users can make use of Freescale/NXP's AN2295 Developer's Serial Bootloader programming utility to load code via UART.

In order to activate this mode the following defines are enabled in `config.h`

```
#define SERIAL_INTERFACE
#define DEVELOPERS_LOADER
```

Optionally, the programmed code can be read back when the define

```
    #define DEVELOPERS_LOADER_READ
```

is enabled.

The communication is optionally secured by CRCs when the define

```
    #define DEVELOPERS_LOADER_CRC
```

is enabled.

This selects the UART interface and specifies that the protocol is that used by the application note's PC software. *This mode overrides the SREC and KBOOT modes of operation.*

The utilisation is compatible with Freescale/NXP's AN2295 Kinetis (version 8) implementation and so all further information about its utilisation can be found at https://www.nxp.com/docs/en/application-note/AN2295.pdf

When the AN2295 Developer's Serial Bootloader is used together with USB-MSD a file called "`DEVELOPE.S19`" is visible on the hard drive after firmware has been loaded. This file has the size of the code but a fixed time/date.

12. SD-Card Loader

The SD-Card boot loader operation is available when the processor has either SPI or SDIO interfaces and the board has an SD card or µSD card socket. FAT16 and/or FAT32 formatted cards are supported, based on the µtFAT (FAT compatible module in the µTasker project).

This mode is enabled by setting the define `SDCARD_SUPPORT` in `config.h` and can operate in parallel with other loader modes. *Since the SD card loader always checks the SD card for new software it may be necessary to adjust the logic when used together with methods that check an input to force boot loader mode and otherwise immediately jump to a loaded application. Such logic can be controlled by specific code added to* `fnUserHWInit()` *in* `Loader.c`*, chosen according to the projects requirements and loader priorities.*

Default operation is in SPI mode, which is possible with most processors. The SPI connection details are set in the file `app_hw_xxxx.h` (depending on processor family used). If the processor incorporates an SDIO/SDHC controller, this can be enabled with `#define SD_CONTROLLER_AVAILABLE` in the hardware specific file. Since the boot loader usually only needs to be able to read from the SD card and not write, or format it, a minimum configuration of the µtFAT is possible by using the following example configuration in `config.h`:

```
#ifdef SDCARD_SUPPORT
    #define SD_CARD_RETRY_INTERVAL    2 // attempt SD card init. at 2s intervals
    #define UT_DIRECTORIES_AVAILABLE  1 // this many directories objects are
                                           available for allocation
  //#define UTMANAGED_FILE_COUNT      10 // no managed files required
  //#define UTFAT_LFN_READ               // no long file name read support
    #define STR_EQUIV_ON                 // ensure that this routine is available
    #ifdef UTFAT_LFN_READ
        #define MAX_UTFAT_FILE_NAME  (100) // the max. file name length supported
    #endif
  //#define UTFAT_WRITE                  // disable write functions
  //#define UTFAT16                      // support only FAT32 (not FAT16)
    #define UTFAT_RETURN_FILE_CREATION_TIME // when a file is opened, its creation
                                      time and date is returned in the file object
    #define UTFAT_DISABLE_DEBUG_OUT      // disable all debug output from utFAT
#endif
```

The SD card loader always first checks to see whether there is an SD card available. If this is not the case it will jump to the existing application (if present). The following steps are the main ones that take place when there is a card present:

- As long as the SD card could be mounted, a software file at a specific location and with a specific name is checked for. The name and location is defined in `loader.h` with default:

```
#define NEW_SOFTWARE_FILE         "software.bin"
```

- If the file is found, the content is compared with the loaded application
- If no file is found or if the content is the same as the present application the application is started
- If the file is different it causes the loader to overwrite the present application with it and then starts it

If preferred, an option to use wild-card name matching can be enabled with the define `WILDCARD_FILES`.

In this case `#define NEW_SOFTWARE_FILE    "software*.bin"`
allows the software to be used in conjunction with a version number, whereby the first (usually only) matching file is loaded.

The file is saved with a small header containing CRC and a secret key to ensure that foreign files or corrupted files don't get loaded (by mistake or by malicious intentions). The header is added to the binary output of the application build by running the utility uTaskerConvert; this format is the same as used by the "Bare-Minimum" loader as discussed in http://www.utasker.com/docs/uTasker/uTasker_BM_Loader.pdf. An example of converting the application software is:

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

The matching configuration in `Loader.h` would then be:

```
    #define NEW_SOFTWARE_FILE     "software.bin"

    #define VALID_VERSION_MAGIC_NUMBER   0x1234
    #define _SECRET_KEY                  {0xa7, 0x48, 0xb6, 0x53, 0x11, 0x24}
```

Using the option `ENCRYPTED_CARD_CONTENT` requires the firmware to be additionally encrypted, whereby the serial loader decrypts it when copying the firmware to the application Flash. An example of encrypting the software, also described in detail in the "Bare-Minimum" loader document, is :

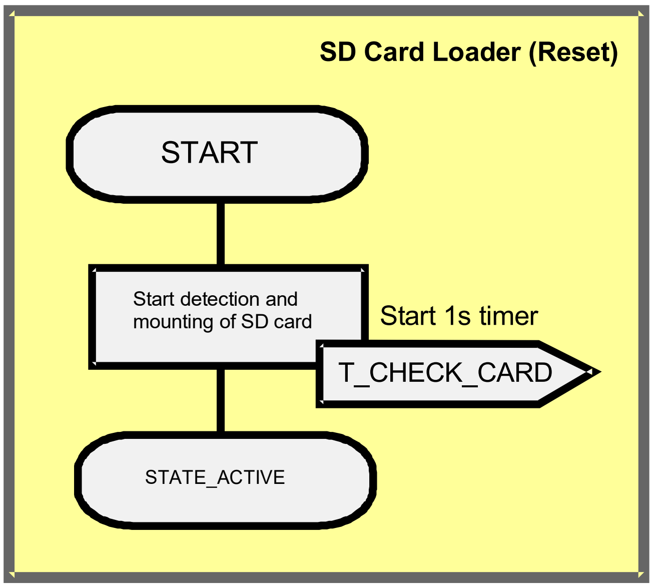
```
uTaskerConvert.exe uTaskerV1.4_BM.bin software.bin -0x1235
-b748b6531124 -ff25a788f2e681338777 -afe1 -c298
```

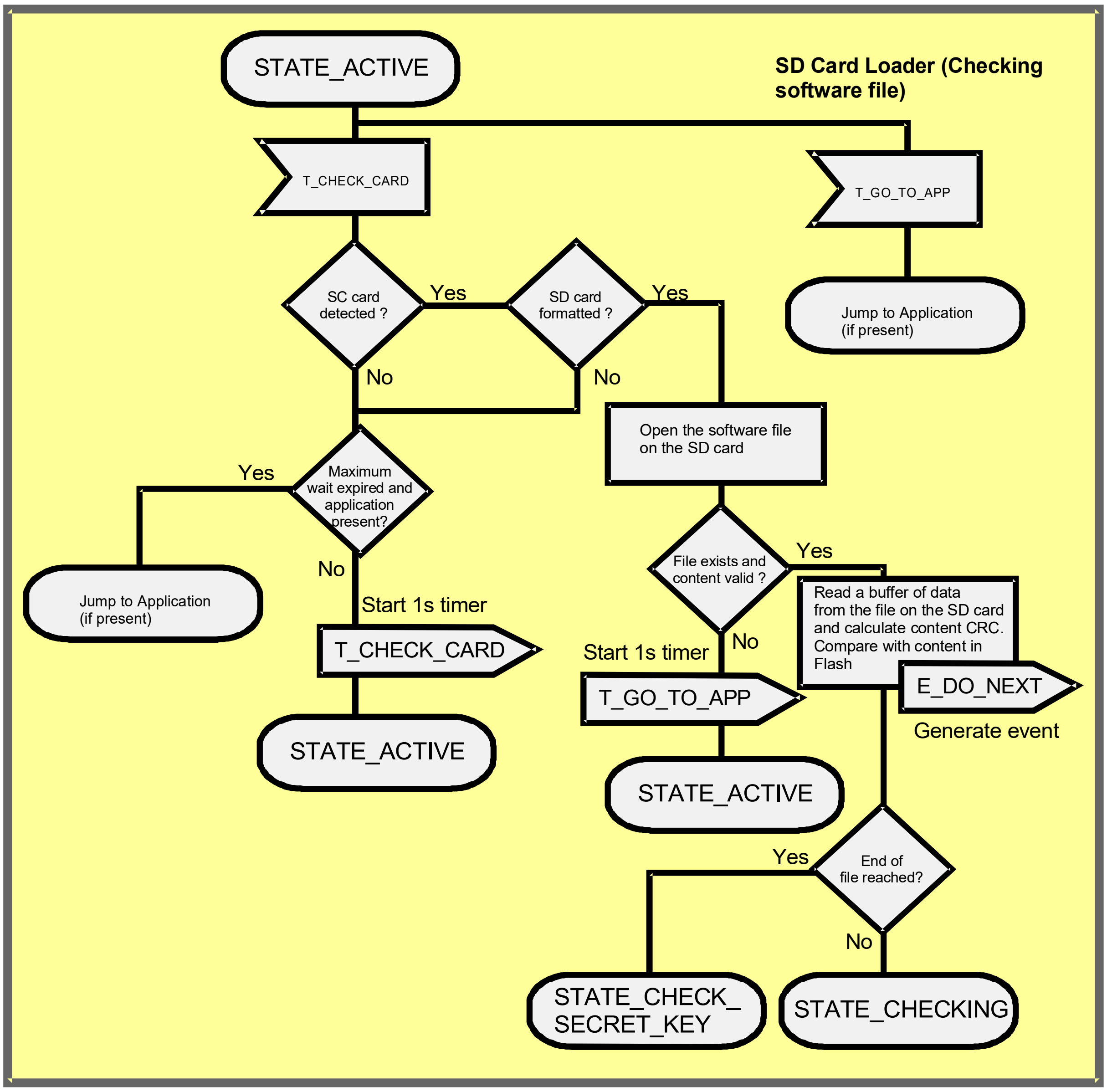
The matching configuration in `Loader.h` would then be:

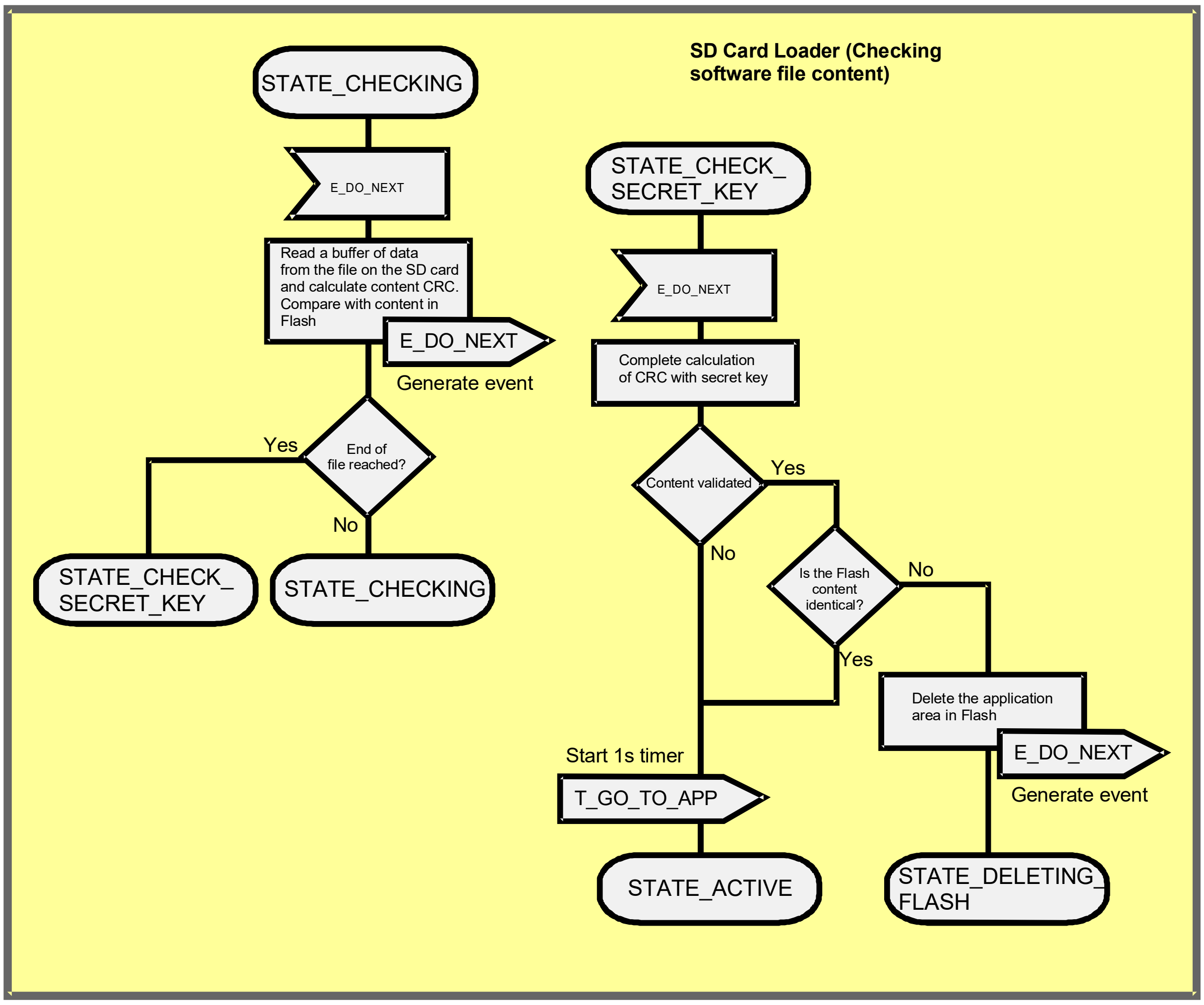
```
    #define NEW_SOFTWARE_FILE     "sd_card_enc.bin"
    #define VALID_VERSION_MAGIC_NUMBER   0x1235
    #define _SECRET_KEY           {0xb7, 0x48, 0xb6, 0x53, 0x11, 0x24}
    static const unsigned char ucDecrypt[] =
         {0xff, 0x25, 0xa7, 0x88, 0xf2, 0xe6, 0x81, 0x33, 0x87, 0x77};
                            // must be even in length (dividable by unsigned short)
    #define KEY_PRIME             0xafe1         // never set to 0
    #define CODE_OFFSET           0xc298         // ensure that this value
                                          is a multiple of the smallest flash
                                          programming entity size
                                         (divisible by 8 is suitable for all
                                          Kinetis parts)
```

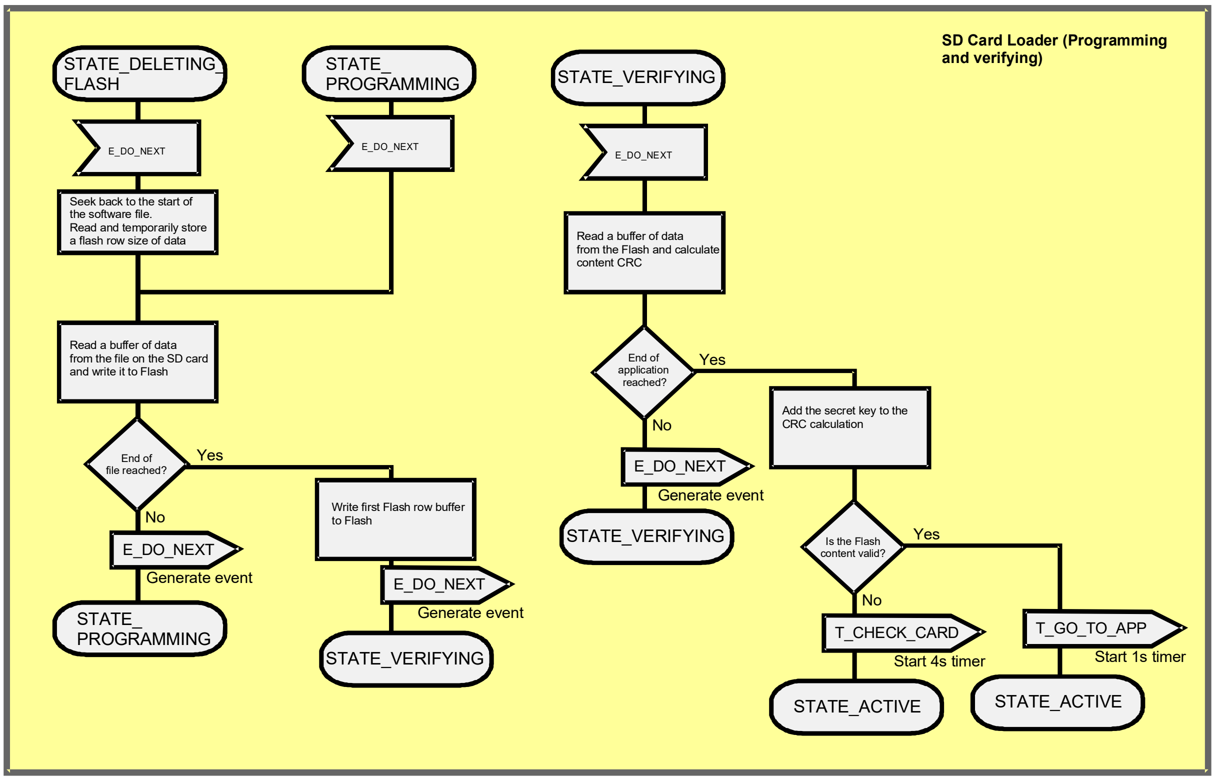
If the software file on the disk should be automatically deleted after successful loading, this can be enabled with the define `DELETE_SDCARD_FILE_AFTER_UPDATE`. Since the SD card needs to be written to in order to delete the file, the utFAT option `UTFAT_WRITE` needs also to be enabled in this case, which also increases the size of the serial loader due to the additional functions involved.

The detailed SD card operation is depicted in the following state-event diagrams (not showing options of decryption of encrypted content nor deleting SD card content on completion):


SD Card Loader (Reset)
START
Start detection and mounting of SD card
Start 1s timer
T_CHECK_CARD
STATE_ACTIVE


STATE_ACTIVE
SD Card Loader (Checking software file)
T_CHECK_CARD
T_GO_TO_APP
SC card detected ?
Yes
SD card formatted ?
Yes
No
No
Jump to Application (if present)
Open the software file on the SD card
Maximum wait expired and application present?
Yes
No
Jump to Application (if present)
Start 1s timer
T_CHECK_CARD
STATE_ACTIVE
File exists and content valid ?
Yes
No
Read a buffer of data from the file on the SD card and calculate content CRC. Compare with content in Flash
E_DO_NEXT
Generate event
Start 1s timer
T_GO_TO_APP
STATE_ACTIVE
End of file reached?
Yes
No
STATE_CHECK_SECRET_KEY
STATE_CHECKING

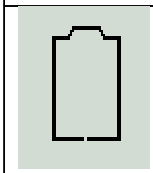
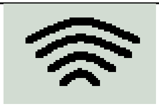
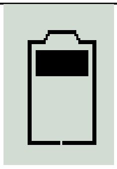
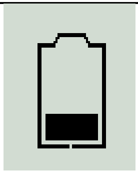
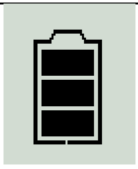

SD Card Loader (Checking software file content)
STATE_CHECKING
E_DO_NEXT
Read a buffer of data from the file on the SD card and calculate content CRC. Compare with content in Flash
E_DO_NEXT
Generate event
End of file reached?
Yes
No
STATE_CHECK_ SECRET_KEY
STATE_CHECKING
STATE_CHECK_ SECRET_KEY
E_DO_NEXT
Complete calculation of CRC with secret key
Content validated
Yes
No
Is the Flash content identical?
No
Yes
Delete the application area in Flash
E_DO_NEXT
Generate event
Start 1s timer
T_GO_TO_APP
STATE_ACTIVE
STATE_DELETING_ FLASH


SD Card Loader (Programming and verifying)
STATE_DELETING_ FLASH
E_DO_NEXT
Seek back to the start of the software file. Read and temporarily store a flash row size of data
STATE_ PROGRAMMING
E_DO_NEXT
Read a buffer of data from the file on the SD card and write it to Flash
End of file reached?
Yes
No
E_DO_NEXT
Generate event
STATE_ PROGRAMMING
Write first Flash row buffer to Flash
E_DO_NEXT
Generate event
STATE_VERIFYING
STATE_VERIFYING
E_DO_NEXT
Read a buffer of data from the Flash and calculate content CRC
End of application reached?
Yes
No
E_DO_NEXT
Generate event
STATE_VERIFYING
Add the secret key to the CRC calculation
Is the Flash content valid?
Yes
No
T_CHECK_CARD
Start 4s timer
STATE_ACTIVE
T_GO_TO_APP
Start 1s timer
STATE_ACTIVE

At various points in the SD card loading process defines are included that can be used to display its present state. This allows an indication of the progress to be displayed, for example on LED outputs. In the case of the Kinetis K40 KwikStik project the progress is displayed on its LCD display by assigning these as follows:

```
#define _DISPLAY_SD_CARD_NOT_PRESENT()    SET_SLCD(39TO36, QS_39TO36_POUNCE_LOGO)
#define _DISPLAY_SD_CARD_NOT_FORMATTED() SET_SLCD(3TO0,   QS_3TO0_JLINK_SYMBOL)
#define _DISPLAY_NO_FILE()          SET_SLCD(3TO0,   QS_3TO0_CONNECTION_SYMBOL)
#define _DISPLAY_SD_CARD_PRESENT() SET_SLCD(3TO0,   QS_3TO0_BATTERY_SYMBOL)
#define _DISPLAY_VALID_CONTENT()   SET_SLCD(39TO36, QS_39TO36_BATTERY_CHARGE_1)
#define _DISPLAY_INVALID_CONTENT() SET_SLCD(3TO0,   QS_3TO0_BATTERY_CHARGE_3)
#define _DISPLAY_SW_OK()            SET_SLCD(39TO36, QS_39TO36_BATTERY_CHARGE_2); \
                                    SET_SLCD(3TO0,   QS_3TO0_BATTERY_CHARGE_3)
#define _DISPLAY_ERROR()            SET_SLCD(3TO0,   QS_3TO0_CLOCK_SYMBOL)
```

This causes the following symbols to be displayed, depending on the SD card loading state and progress. *The exact output methods are however no the subject of this document and the SLCD can be found at* http://www.utasker.com/docs/uTasker/uTasker_SLCD.pdf

pounce CONSULTUING	SD card not present `_DISPLAY_SD_CARD_NOT_PRESENT()`
	SD card present but not formatted `_DISPLAY_SD_CARD_NOT_FORMATTED()`
	SD card is present `_DISPLAY_SD_CARD_PRESENT()`
	SD card is present but has no software file on it `_DISPLAY_NO_FILE()`
	SD card is present but has an invalid software file `_DISPLAY_INVALID_CONTENT()`
	SD card is present and it has a valid software file `_DISPLAY_VALID_CONTENT()`
	SD card is present and it has a valid software file which matches the application in flash (either initial state or after updating the flash content) `_DISPLAY_SW_OK()`

⏰	Software update was performed but the new flash content has a mismatch (this should normally never occur but it would lead to a new programming attempt after a short delay) `_DISPLAY_ERROR()`

Note that the define `FORCE_BOOT()` in `fnUserHWInit()` in `Loader.c` controls whether an existing application software is immediately started without passing through the SD card loader. If the SD card socket has a card detect switch integrated into it, it can be useful to use this to force the SD card loader mode when a card is inserted and then allow the application to be immediately started when no SD card is in the socket.

The Kinetis KwikStik, for example, can do this by defining the check as

```
#define FORCE_BOOT()   (_READ_PORT_MASK(E, PORTE_BIT27) == 0)
```

whereby the input is configured previously together with other such configurations by

```
#define INIT_WATCHDOG_DISABLE()   _CONFIG_PORT_INPUT_FAST_HIGH(E,
(PORTE_BIT27), 0)
```

When the SD card loader is used together with USB-MSD a file called “`SOFTWARE.BIN`” is visible on the hard drive after firmware has been loaded. This file has the size of the code and also the date/time from the original file on the SD card.

12.1 Loading Multiple Files and Controlling the location of the Firmware File(s)

In some situations it may be required to allow the serial loader to not only update the processor's firmware from an SD card but also to allow the same method to be used to update additional data to other locations; an example is when the processor is responsible for maintaining the firmware of sub-modules, such as FPGAs, that may need to be programmed from an image in the processor's memory space each time the system starts.

To accomodate such requirements the following options can be used:

```
#define SDCARD_FILE_COUNT   3    // 3 different files are to be loaded from the SD card
```

The default setting (valid also when not defined) is a value of 1, which means that the operation follows the procedure as already discussed in this chapter. However when multiple files are specified the serial loader implements this number of loaders corresponding to the SD card flow chart and each operating in parallel to update these multiple files.

The location of the firmware file(s) can be controlled using the option:

```
#define VARIABLE_FW_DIRECTORY    // allow the firmware directory to be configurable
(rather than being fixed in the root directory)
```

whereby the directory locations and the file names are specified by (for example):

```
#define FIRMWARE_DISK_LOCATION    "NewCode/Processor"
#define FIRMWARE2_DISK_LOCATION   "NewCode/FPGA"
#define FIRMWARE3_DISK_LOCATION   "NewCode/DSP"

#define NEW_SOFTWARE_FILE         "software*.bin"
#define NEW_SOFTWARE2_FILE        "fpga*.bin"
#define NEW_SOFTWARE3_FILE        "dsp*.bin"
```

Since the destination and maximum size of each of the files to be loaded will be different additional defines control these details (for example):

```
#define FIRMWARE2_DESTINATION_ADDRESS      (SIZE_OF_FLASH)
#define FIRMWARE3_DESTINATION_ADDRESS      (SIZE_OF_FLASH + SPI_DATA_FLASH_0_SIZE)
```

whereby the destination address of the first file (the first file is always the processor firmware) is already specified by `_UTASKER_APP_START_`.

Similarly the maximum file sizes are specified as

```
#define FIRMWARE2_MAX_SIZE     (64 * 1024)
#define FIRMWARE3_MAX_SIZE     (64 * 1024)
```

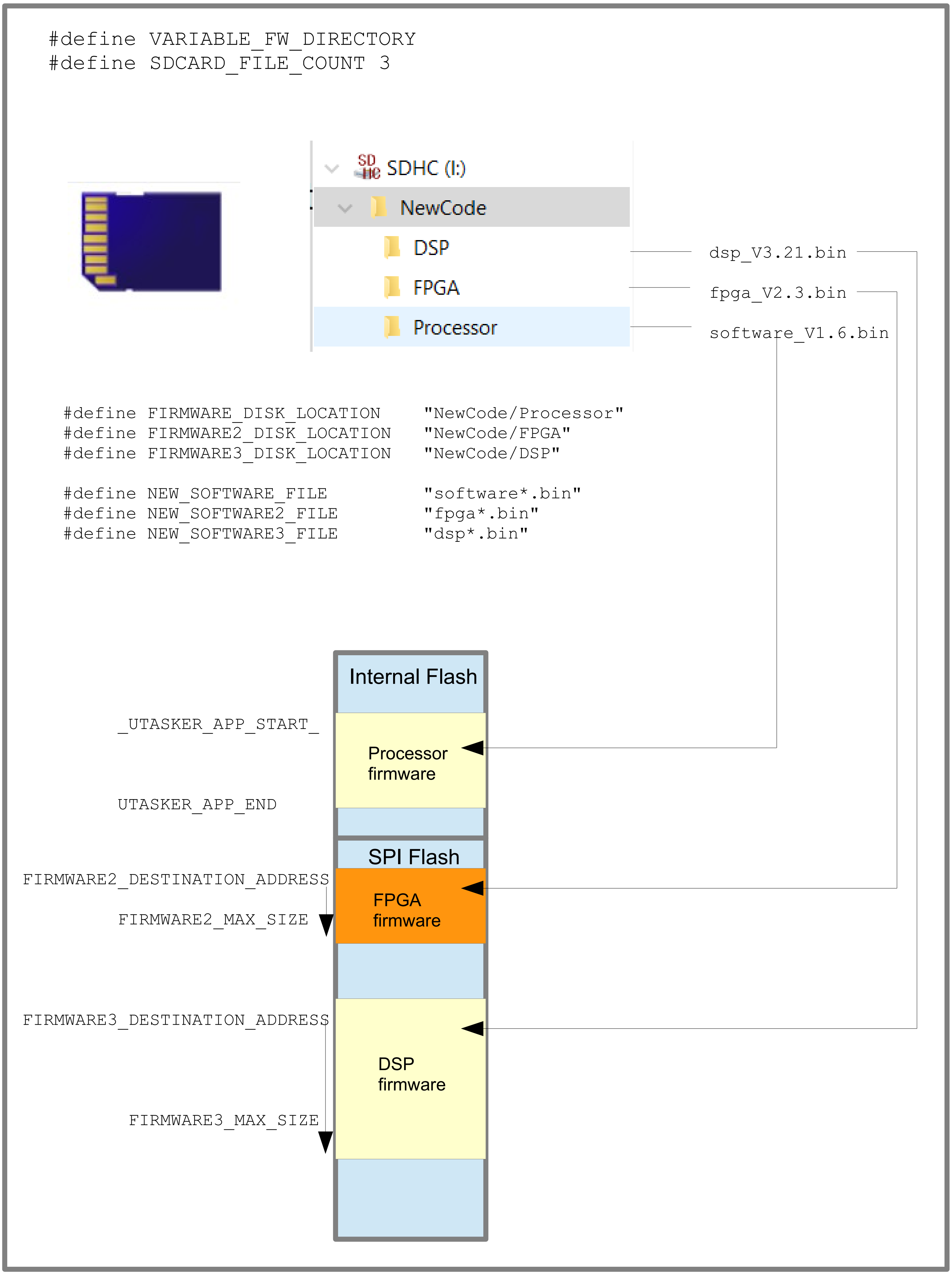
wherey the maximum size of the first file (the first file is always the processor firmware) is already specified by `UTASKER_APP_END`.

Each of the files that are located on the SD card need to have been converted to a file with upload header and all use the same authentication and security settings.

For clarity this configuration is show in graphical form on the next page showing a practical scenario where the processor firmware is updated from the SD card to the processor's internal Flash. In addition two further firmware versions are updated to two different regions of an external SPI flash (viewed as following the internal processor Flash in the virtual memory map). It is assumed that the processor is responsible for loading or updating code from this SPI Flash storage to accompanying devices (an FPGA and a DSP) or that these further devices have direct access to the new code and can then load or update themselves form the new source.

Such a configuration makes upgrading such a set of firmware versions simple to manage and to perform.

Although not detailed here the µTasker Flash drivers can be simply configured to include support for various SPI connected Flash devices, after which the control of the memory regions used is fully automated based on the firmware destination addresses defined – it is even possible to upload new firmware to an area straddling Flash memory types (eg. The data starts in internal flash and ends in external SPI flash), although such layouts would probably be rare.



```
#define VARIABLE_FW_DIRECTORY
#define SDCARD_FILE_COUNT 3
```

```
#define FIRMWARE_DISK_LOCATION     "NewCode/Processor"
#define FIRMWARE2_DISK_LOCATION    "NewCode/FPGA"
#define FIRMWARE3_DISK_LOCATION    "NewCode/DSP"

#define NEW_SOFTWARE_FILE          "software*.bin"
#define NEW_SOFTWARE2_FILE         "fpga*.bin"
#define NEW_SOFTWARE3_FILE         "dsp*.bin"
```

13. Ethernet Web Server

Devices with Ethernet can activate a web server dedicated for firmware uploading by enabling

```
    #define ETH_INTERFACE
```

and

```
  #define USE_HTTP
```

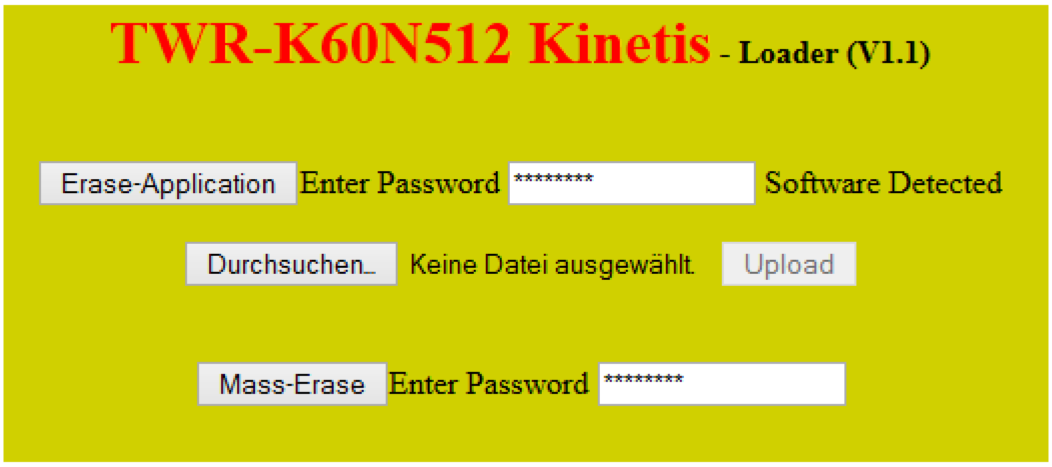
in `config.h`

(The web server method can be used in parallel with the FTP method explained in the following chapter)

This sets the board on a fixed IP address of 192.168.0.125 and a fixed MAC address of 00-00-00-00-00-05. *[Teensy 4.1 users can enable `#define MAC_FROM_USER_REG` to use the MAC address that is pre-programmed by the manufacturer in the eFUSEs of their boards].*

These values can be modified in the code as required or takes from alternative storage locations as appropriate for the board and project in question.

By browsing to this IP address using any browser of choice the following page is returned – note that the page is controlled by embedded HTML and can thus be modified in appearance to suit a project/product.


TWR-K60N512 Kinetis - Loader (V1.1)

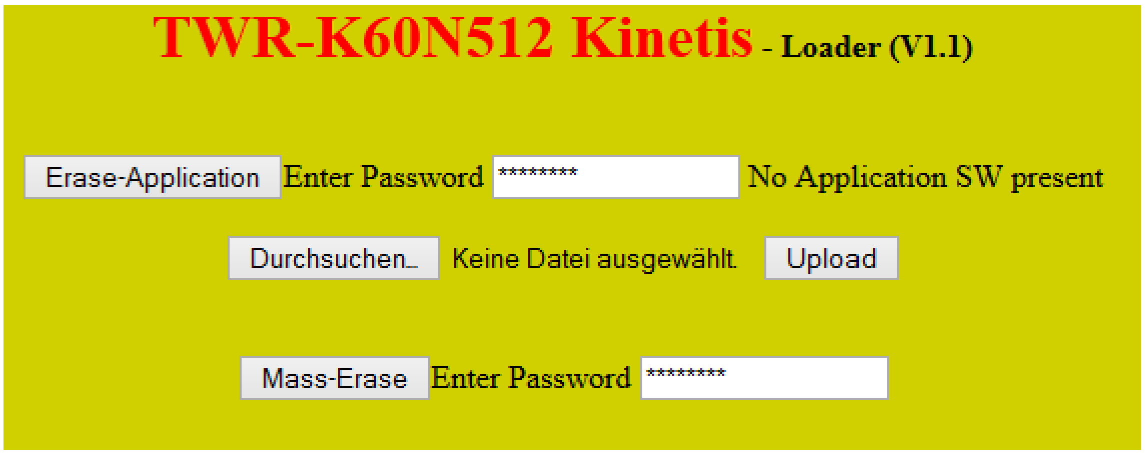
This screen shot shows that there is application software already loaded ad so the first step will be to delete this software so that new firmware can be uploaded. This is performed by entering the erase password and then clicking on the Erase-Application button.

The default application erase password is

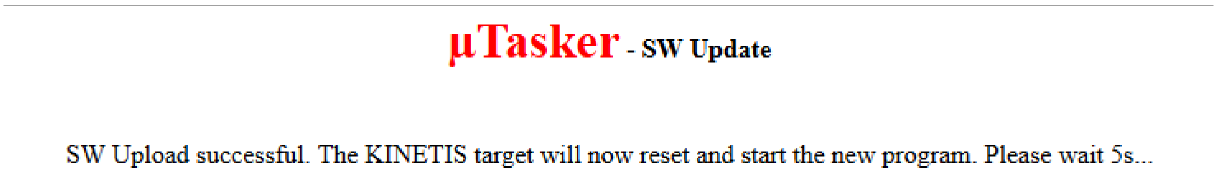
p12X-k3ve2B1O2Ba

and can be modified in the code to suit the project in question.

The next screen shot shows that the application has been deleted and the Upload button is no longer disabled (compare with the previous screen-shot).


TWR-K60N512 Kinetis - Loader (V1.1)

The file to be uploaded is selected and then the Upload button is pressed in order to perform the transfer, after which the screen will show


µTasker - SW Update

SW Upload successful. The KINETIS target will now reset and start the new program. Please wait 5s...

which indicates that the file was successfully accepted and programmed to Flash. The new application software will now start.

The Mass-Erase button allows the complete flash (including the boot loader) to be deleted. This can be useful in case the device has been set to a protected mode which doesn't accept debugger contact because a mass erase will generally also reset protection to that the device can be accessed again. *The button is neither needed nor used for normal firmware upload activity.*

The default mass-erase password is

mMm122-aHHHQq1x8

and can be modified in the code to suit the project in question.

The board running the Ethernet loader can also be pinged on its IP address.

If the option `SUPPORT_HTTP_POST_FILE_NAME` is enabled a file with the name of the uploaded data is visible on the hard drive after firmware has been loaded (when used together with USB-MSD). This file has the name and size of the code but a fixed time/data due to the fact that HTTP post doesn't preserve the time/date of the uploaded file.

If the options `SUPPORT_HTTP_POST_FILE_NAME` is not enabled a file called "`WEB_LOAD.BIN`" is visible on the hard drive after firmware has been loaded. This file has the size of the code but a fixed time/date.

14. FTP Server

Devices with Ethernet and with a two step loader technique (see the “Bare-Minimum” Loader https://www.utasker.com/docs/uTasker/uTasker_BM_Loader.pdf) can activate an FTP server by enabling

```
#define ETH_INTERFACE
```

and

```
#define USE_FTP
```

in `config.h`

(The FTP server method can be used in parallel with the web server method explained in the previous chapter)

This sets the board on a fixed IP address of 192.168.0.125 and a fixed MAC address of 00-00-00-00-00-05. *[Teensy 4.1 users can enable `#define MAC_FROM_USER_REG` to use the MAC address that is pre-programmed by the manufacturer in the eFUSEs of their boards].*

These values can be modified in the code as required or takes from alternative storage locations as appropriate for the board and project in question.

The FTP server uses the µFileSystem as internal file system to save received data to, whereby a certain file location is defined to be that used by new applications (or new serial loaders if the concept supports serial loader updating too). The new application is required to be prepared with an upload header (see details in the SD card loader chapter) and can be optionally encrypted.

The following shows connecting and loading a file using the DOS FTP client in a command shell to a Teensy 4.1 (as reference) whereby the FTP server is in anonymous mode requiring no user/password login (simple hit the enter key in each case):

```
ftp 192.168.1.125
Connected to 192.168.1.125.
220 Welcome i.MX RT FTP.
500 What?.
User (192.168.1.125:(none)):
331 Enter pass.
Password:
230 Log OK.
ftp>
ftp> dir
200 OK.
150 Data.
-r Empty
226 OK.
ftp: 13 bytes received in 0.04Seconds 0.33Kbytes/sec.
ftp> put uTaskerV1.4.15_AES256_TEENSY_4_1.bin z.bin
200 OK.
150 Data.
226 OK.
ftp: 93320 bytes sent in 0.20Seconds 461.98Kbytes/sec.
ftp> dir
200 OK.
150 Data.
-rwxrwxrwx 1 502 502 93320 Feb 9 2020 z.BIN
226 OK.
ftp: 48 bytes received in 0.05Seconds 0.98Kbytes/sec.
ftp>
```

Notice that in this case the new firmware is renamed/saved to "`z.bin`" which is the required location in this board's case.

Once the upload has terminated successfully the board restarts and the "Bare-Minimum" loader completes the process by copying the intermediate file to its final location.

The board running the FTP loader can also be pinged on its IP address.

When used together with USB-MSD a file called "`OTA_FW_.BIN`" is visible on the hard drive after firmware has been loaded. This file has the size of the code but a fixed time/date.

15. Modbus Slave Loader

Devices with UART or Ethernet can activate a Modbus slave dedicated to receiving new firmware:

```
#define USE_MODBUS
```

in `config.h`

The UART used if configured by `LOADER_UART`

and the Modbus slave configuration in `modbus_app.c` by (example)

```
static const MODBUS_PARS cMODBUS_default = {
    MODBUS_PAR_BLOCK_VERSION,
#if defined MODBUS_SERIAL_INTERFACES && defined SERIAL_INTERFACE
    {
        (1),                            // slave address
    },
    {
        (CHAR_8 | NO_PARITY | ONE_STOP | CHAR_MODE /*| UART_SINGLE_WIRE_MODE*/),
                                        // serial interface settings
    },
    {
        SERIAL_BAUD_115200,             // baud rate of serial interface
    },
    {
        (MODBUS_MODE_RTU | MODBUS_SERIAL_SLAVE/* | MODBUS_RS485_POSITIVE*/),
                                       // RTU mode as slave
    },
    #if defined MODBUS_ASCII
    {
        (DELAY_LIMIT)(2 * SEC),        // inter-character delays greater than 2s are
                                          considered errors in ASCII mode
    },
    {
        0x0a,                          // ASCII mode line feed character
    },
    #endif
    #if defined MODBUS_SUPPORT_SERIAL_LINE_FUNCTIONS &&
       !defined NO_SLAVE_MODBUS_REPORT_SLAVE_ID
    { 'u', 'T', 'a', 's', 'k', 'e', 'r', '-', 'M', 'O', 'D', 'B', 'U', 'S', '-',
      's', 'l','a','v','e' },
    #endif
#endif
};
```

Although the Modbus protocol is old it is still often found being used in many modern processes due to the fact that it is simple, flexible and robust. The Modbus slave configuration [`modbus_app.c`] allows uploading new firmware via a Modbus master (TCP or serial in ASCII or RTU mode) using the following strategy (the technique has been proven to be simple to implement and works reliably in numerous products but changing or extending the method is also simple if needed):

- The slave has a block of **66** holding registers which are located by default at the start of the holding register address range; the 65th holds the present loader state.

```
#define HOLDING_REGS_START    0      // start address
#define HOLDING_REGS_END      65     // end address
```

Each multiple register (65 register) write to the holding register start address (0 in this configuration) consists of the following content, remembering that holding registers are 16 bit each:

```
    HEADER     [index byte | data length byte]
    DATA0      [data 0 byte | data 1 byte]
    DATA1      [data 2 byte | data 3 byte]
    ..
    DATA63     [data 126 byte | data 127 byte]
```

where index byte is a simple counter (*initial value may be any value value between 0 and 0xff*) that should increment or decrement (*or simply change between frames*). If the same value is received in two consecutive frame receptions the content will be ignored since it is recognised as a repetition (which can occur if the Modbus slave response is corrupted).

data length is the valid data content in bytes (128 for a full 64 holding register content).

data n is the firmware byte content, packed in the holding registers in big-endian order (*they are converted to the correct byte ordering for the target at reception, if needed*).

- The master writes each block of new firmware as a single write to the 65 holding registers, after which the content is programmed (on first write the application area of flash is automatically erased too). *The data length value is initially always 128*.

- The master repeats writing 65 register blocks (and 128 data bytes) to transfer and program the rest of the new code in a linear fashion.

- When the master has transferred all the new code it sends the last block with a data length of less that 128 bytes; only the valid bytes are written and understood to signify the end of the firmware upload file. *If the firmware length happens to divisible by 128 it sends a further dummy holding register write to the start of the register area with zero data length to close*.

- When the transfer has completed the new firmware is started automatically after a delay of 1s. During this time a read of the final holding register will return the value `0x5555`, which indicates a successful complete upload has taken place.

Should there be an address or frame length error during the process the final holding register will be read as `0xaa01`.

If the final holding register is read during the loading process it will return the last index byte value that has been received.

For general details about configuring the Modbus protocol please consult the Modbus guide: http://www.utasker.com/docs/MODBUS/uTasker_MODBUS.PDF

16. I^{2}C Slave Loader

Devices with an I^{2}C slave controller can be configured to receive firmware from an I^{2}C master.

```
#define I2C_INTERFACE
```

in `config.h`

For general details about I^{2}C slave driver operation see the document
http://www.utasker.com/docs/uTasker/uTasker_I2C.pdf

whereby the I^{2}C slave loader uses the interrupt call-back method as described in the document.

When the I^{2}C loader starts it checks the application area for erased flash so that it can inform of its state if requested by the I^{2}C master. It then waits to be addressed on its unique I^{2}C slave address that is configured by the define

```
#define OUR_SLAVE_ADDRESS                0x50
```

in `Loader.h`.

This value means that it is written to at address **0x50** and read from address **0x51**.

The master must coordinate the loading by first checking the state of the slave's application flash, which can be performed by reading three bytes from its read address.

`[`**`0x51`**`] [Vmajor] [Vminor] [Flash state]`

- The first byte of data returned is the I^{2}C loader's major version number.
- The second byte read is the minor version number.
- The third byte is the flash state: 0x00 means that the flash is not erased and 0x01 means that it is erased.

If more data is read, the three information bytes cycle, for example:

`[`**`0x51`**`] [Vmajor] [Vminor] [Flash state] [Vmajor] [Vminor] [Flash state]`, etc.

If the application flash is not signalling being in an erased state (0x01) the master can command its deletion by writing the command 0x00, with confirmation key 0x52 0x84, which gives the I^{2}C write frame

`[`**`0x50`**`] [0x00] [0x52] [0x84]`

and waiting until the slave changes its flash state (0x01), for example by polling the state until it is read being set accordingly.

A delete command will not be executed if the flash is already erased so it is also safe to always sent a delete command and then poll the flash state until it indicates that it is ready for programming.

Programming is performed by writing data with the command 0x01. The data can be written in blocks of any length:

```
[0x50] [0x01] [D0] [D1] [Dn] …....
```

where Dn is the binary data to be programmed, starting at the first address in the application flash area and increasing linearly.

Once the program loading has started, reads from the slave will no longer return the version number and flash status but instead will return the *programming state*. The master can thus check the programming state between writing blocks of data by reading 5 bytes as follows:

```
[0x51] [state] [length3] [length2] [length1] [length0]
```

where the state will be 0x01 when the loading is taking place and no errors have been encountered. Length[4] is a counter (length0 is the least significant byte and length3 the most significant byte in a long word representation) which indicates the amount of data that has been received. The master can use this to verify that no data was lost during the process.

The data read is also cycled through these 5 bytes of information in case more data were to be read.

Should the master ever wish to read the version and flash status after programming has started it can write a dummy command

```
[0x50] [0x03]
```

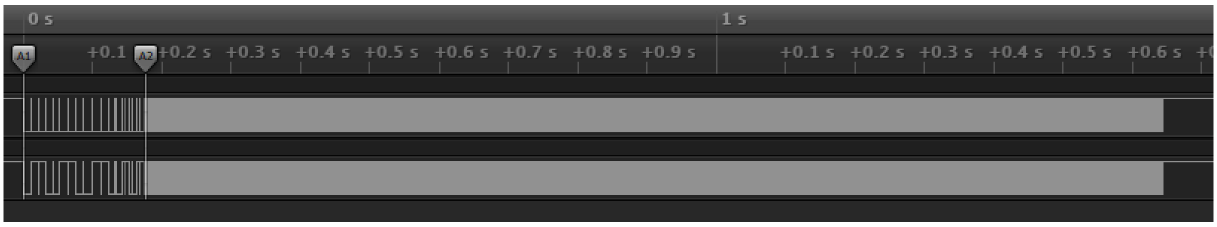
which will reset the read mode. Once any further programming data has been sent the read mode will be set back to the programming state again.

Once the master has programmed all data to the slave it can command a reset so that the slave can restart and execute the new code. This is performed by sending the reset command 0x02, together with confirmation key 0x55 0xaa:

```
[0x50] [0x02] [0x55] [0xaa]
```

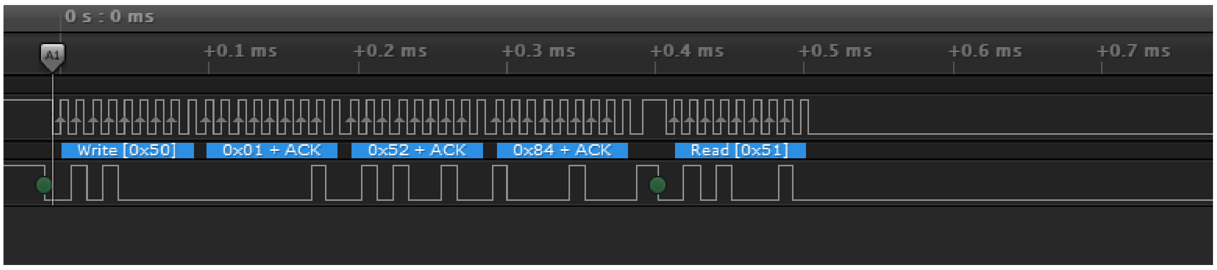
The slave also uses this command to commit the start of the program code, which is had saved to an intermediate buffer rather than writing directly to the start of flash. This is so that an interruption of the programming process will leave the start of the application flash empty so that the serial loader will automatically restart and not attempt to execute a non-completed program.

The following shows a logic analyser trace of the programming of a small program to a device via its slave I^{2}C loader.

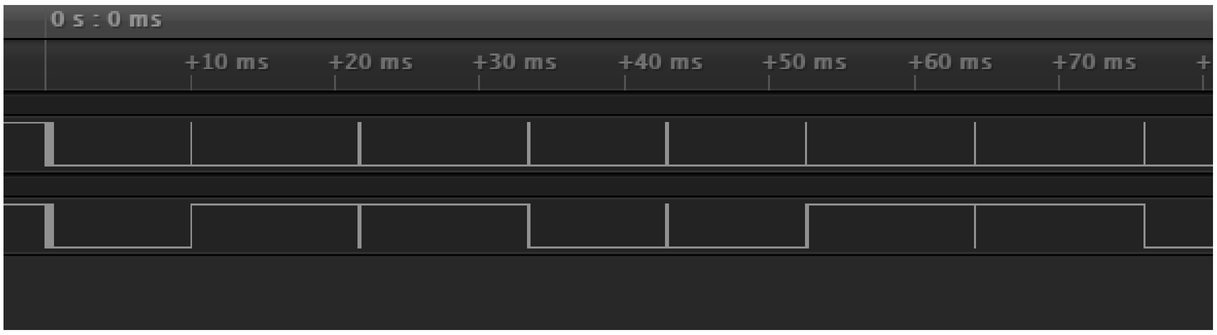


It is interesting to note that the initial flash deletion phase is clearly visible due to the fact that the slave I^{2}C device holds the I^{2}C bus when it is not ready to complete operation (known as clock-stretching).

A closer view of the delete command at the start of the recording shows that the following read command (to check the erasure state after the delete command) doesn't complete yet due to the fact that the processor is in the process of deleting a sector if its flash which can require several ms to complete.



During the delete phase, where a number of flash sectors are being erased, it is seen that the slave device can only respond to interrupts in between each sector erase and so a pattern emerges:



where the erase duration of each sector is seen to be around 10ms (*example of 1k sectors on KL02 processor*).

Due to the I^{2}C slave's ability to hold the bus when the reception cannot yet be handled no additional software handshaking is required during the programming process.

17. Conclusion

The µTasker Serial Loader may be programmed to a target board and used as an in-circuit tool for loading application software via UART or a number of other interfaces. By rebuilding the serial loader project the user can customise the characteristics of the loader to suit individual target configurations.

In addition, the µTasker Serial Loader can be configured to operate as a USB-MSD device (devices with USB device support); this can be in parallel to the SREC serial mode or as an alternative to it. The USB-MSD uses binary files rather than SREC files and it was noted that the link address of the application file is shifted when the USB-MSD mode is used (which is also valid for the SREC operation when used in parallel) due to the fact that root file entry content is also saved together with the software.

A further mode of operation that is possible is to load application software from an SD card interface. In this mode of operation a software file name is defined which is checked for on the SD card; if this file exists its content is compared with the present application software loaded and loaded in case the file content is different to it.

When USB Host is available memory sticks can be used in a similar fashion to SD card loading, whereby the USB-MSD Host loader can also be combined with the USB-MSD Device loader to automatically detect whether a host PC is connected or mount an inserted memory stick instead.

In Modbus environments the Modbus slave loader can operate on UARTs (ASCII or RTU) or via TCP/IP.

Devices connected to an I^2C bus can be programmed by a bus master in the I^2C slave loader mode.

Optional Freescale/NXP KBOOT compatible modes are supported on the UART and as USB-HID class interface. The USB-HID mode can operate together with USB-MSD as composite USB device.

Optional Freescale/NXP AN2295 compatible mode is supported on the UART, in parallel with other loaders if required.

Boards with Ethernet can also make use of Ethernet Web Server boot loading, which enables firmware updates to be performed from any browser.

This document has illustrated the operation of the serial boot loader and its various modes, and detailed how it can operate together with any application program.

Modifications:

V0.01 5.5.2009: - Initial draft – work in progress. Not officially released.

V0.02 8.5.2009: - Added appendix with target/compiler specific details. Not officially released.

V0.03 19.11.2009: - Added AVR32 setup and discussion of optional intermediate buffer use during download. Not officially released.

V0.04 21.10.2010: - Correct LED blink speed during serial boot loader phase.

V1.00 04.06.2011: First release including USB-MSD operation

V1.01 25.11.2013: Serial Loader state diagrams and SD card boot loader mode added

V1.02 18.01.2014: Added MSD-USB implementation details and Kinetis CodeWarrior configuration

V1.03 22.01.2015: Added Ethernet Web Server loading and KBOOT serial / USB-HID support, a well as highlighting composite parallel modes of operation.

V1.04 29.01.2015: Added details of encrypted SD card content support and the behaviour when UB-MSD LFN is longer than that supported by the upload file object.

V1.05 01.06.2015: Added AN2295 Developer's Serial Bootloader option.

V1.06 30.10.2015: Added USB-MSD Host Loader mode, including combining with USB-MSD Device operation. Note that V1.2 of the serial loader (with USB Host support) changes some project file names as follows:

- `SDLoader.c` renamed to `disk_loader.c`
- `usb__loader.c` renamed to `usb_device_loader.c`
- `usb_host_loader.c` added

The USB-MSD Device loader configuration define was changed from `USB_MSD_LOADER` to `USB_MSD_DEVICE_LOADER`.

V1.07 5.9.2017: Added USB-MSD device SREC and Intel HEX format support.

V1.08 17.1.2018: Added Modbus slave (description not yet completed) and I^{2}C slave support.

V1.09 15.10.2018: Add appendix C with discussions of Serial Loader Mailbox and configuring application area and security settings.

V1.10 21.9.2020: Add option to update multiple files from different locations on SD card/memory stick to various processor memory locations.

V1.11 21.12.2020: Added details about the Modbus slave updating protocol.

V1.12 31.12.2020: Extend and correct details about the Modbus slave updating protocol.

V1.13 25.4.2021: Add FTP server method description.

Appendix A – Target and Compiler Specific Details

This appendix contains note which are relevant to specific targets and compilers, such as how the application is correcty linked to work together with the ***µTasker Serial Loader***.

a) Coldfire CodeWarrior

The location of the application code in FLASH is determined by the linker script file (*.lcf) as used by the project target. In a 'stand-alone' project the memory will be defined so that the reset vectors are positioned at the start of FLASH (0x00000000) as shown in the typical excerpt below:

```
MEMORY
{
    flash1  (RX)      : ORIGIN = 0x00000000, LENGTH = 0x000400
    flashconfig (RX)  : ORIGIN = 0x00000400, LENGTH = 0x00000018
    flash2  (RX)      : ORIGIN = 0x00000420, LENGTH = 0x003FBE0
    vectorram(RWX)    : ORIGIN = 0x20000000, LENGTH = 0x00000400
    sram    (RWX)     : ORIGIN = 0x20000400, LENGTH = 0x00007C00
    ipsbar  (RWX)     : ORIGIN = 0x40000000, LENGTH = 0x0
}

SECTIONS
{
      .ipsbar       : {} > ipsbar

      .flash1 :
      {
             Startup.s (.text)
             .                      = ALIGN(0x10);

    } > flash1

      .flashconfig :
      {
             flash_config.s (.text)
    } > flashconfig

      .flash2 :
      {
             .                      = ALIGN(0x10);
             *(.text)
             .                      = ALIGN(0x10);

             *(.rodata)
             ___DATA_ROM     = .;
    } > flash2

...
```

Code a-1 Typical memory configuration for a stand-alone application

The sector flash1 is reserved for the reset vector (`startup.s`) and the flash configuration sector is used to locate FLASH configuration values which are read by the device at reset

In order to locate the reset vectors at the start address defined for operation with the serial loader, the linker script content can be changed as follows:

```
MEMORY
{
    flash     (RX)   : ORIGIN = 0x00002800, LENGTH = 0x0003D800
    vectorram (RWX) : ORIGIN = 0x20000000, LENGTH = 0x00000400
    sram      (RWX) : ORIGIN = 0x20000400, LENGTH = 0x00007C00
    ipsbar    (RWX) : ORIGIN = 0x40000000, LENGTH = 0x0
}

SECTIONS
{
       .ipsbar       : {} > ipsbar

       .flash :
       {
              Startup.s (.text)
              .                      = ALIGN(0x10);
              *(.text)
              .                      = ALIGN(0x10);
              *(.rodata)
              ___DATA_ROM     = .;
    } > flash
...
```

Code a-2 Modified linker script for use with the serial loader

Note that the serial loader code is responsible for the FLASH configuration and so this is no longer needed in the application. The application's start up code is still set to be at the start of the FLASH, but its address is now located at 0x2800 (10k), assuming that this is suitable for the Coldfire serial loader.

The serial loader doesn't define interrupt vectors within the FLASH vector area. It is therefore important that the application works with interrupt vectors configured within RAM; this is however the typical operation used and is also used by the µTasker application projects.

b) Kinetis CodeWarrior

The location of the application code in FLASH is determined by the linker script file (*.lcf) as used by the project target. In a 'stand-alone' project the memory will be defined so that the reset vectors are positioned at the start of FLASH (0x00000000) as shown in the typical excerpt below:

```
MEMORY {
    m_reset (RX): ORIGIN = 0x0, LENGTH = 0x8
    m_config (RX): ORIGIN = 0x400, LENGTH = 0x10
    prog_flash (RX) : ORIGIN = 0x410, LENGTH = 0x80000-0x410
    sram (RW) : ORIGIN = 0x1FFF01b0, LENGTH = 0x00020000-0x1b0
}

KEEP_SECTION { .RESET }
KEEP_SECTION { .FCONFIG }

SECTIONS {
.start: {
      ALIGNALL(4);
      __vector_table = .;
      * (.RESET)
            .= ALIGN(0x8) ;
} > m_reset
.config: {
      ALIGNALL(4);
      __flash_config = .;
      * (.FCONFIG)
            .= ALIGN(0x8) ;
} > m_config
.app_text: {
            .                    = ALIGN(0x10);
            *(.text)
            .                    = ALIGN(0x10);

            *(.rodata)
            ___DATA_ROM     = .;
} > prog_flash

...
}
```

Code b-1 Typical memory configuration for a stand-alone application

The sector `m_reset` is reserved for the reset vector (`__vector_table`) and the flash configuration sector (`m_config`) is used to locate FLASH configuration values which are read by the device at reset

In order to locate the reset vectors at the start address defined for operation with the serial loader, the linker script content can be changed as follows:

```
MEMORY {
    m_reset (RX)     : ORIGIN = 0x8080, LENGTH = 0x20
    prog_flash (RX) : ORIGIN = 0x80a0, LENGTH = 0x80000-0x80a0
    sram (RW)        : ORIGIN = 0x1FFF01b0, LENGTH = 0x00020000-0x1b0
}

KEEP_SECTION { .RESET }
KEEP_SECTION { .FCONFIG }

SECTIONS {
.start: {
      ALIGNALL(4);
      __vector_table = .;
      * (.RESET)
             .= ALIGN(0x8) ;
} > m_reset
.config: {
      ALIGNALL(4);
      __flash_config = .;
      * (.FCONFIG)
             .= ALIGN(0x8) ;
} > m_config
.app_text: {
             .                      = ALIGN(0x10);
             *(.text)
             .                      = ALIGN(0x10);

             *(.rodata)
             ___DATA_ROM     = .;
} > prog_flash

...
}
```

Code b-2 Modified linker script for use with the serial loader

Note that the serial loader code is responsible for the FLASH configuration and so this is no longer needed in the application. The application's start up code is still set to be at the start of the FLASH, but its address is now located at 0x8080, assuming that this is suitable for the Kinetis serial loader.

The serial loader doesn't define interrupt vectors within the FLASH vector area. It is therefore important that the application works with interrupt vectors configured within RAM; this is however the typical operation used and is also used by the µTasker application projects.

c) Luminary-Micro Evaluation Boards and GCC Compiler

The Luminary-Micro evaluation boards do not include a serial interface in form of the normal DBUB connection and RS232 driver. Instead UART0 is connected to a channel of an FT2232 device. This device allows a USB connection to the development PC with two USB device channels; the first one used by the Luminary USB Debugger and the second as a serial connection to UART0 (Virtual COM).

In order to work with the serial interface it is only necessary to connect the normal debug USB cable (*which is also used for downloading code using the* ***Luminary Micro Flash Programmer****, for example*) and then open a terminal emulator on the Virtual COM port which was installed. How the Virtual COM port can be checked and reconfigured for use by different COM ports is explained in detail in chapter 4 of the *µTasker USB Demo document*: http://www.utasker.com/docs/uTasker/uTaskerV1.3_USB_Demo.PDF

It is recommended to use `TeraTerminalPro` for serial loading (further details about this program are also contained in the document above).

The position of the application code is controlled by the linker script file (*.ld) as used by the target configuration. A typical configuration for the application linked to the start address 0x00000000 is shown below:

```
MEMORY
{
    FLASH (rx) : ORIGIN = 0x00000000, LENGTH = 0x0003f000
    SRAM (wx)  : ORIGIN = 0x20000000, LENGTH = 0x00010000
}

SECTIONS
{
  __SRAM_segment_start__  = 0x20000000;
  __SRAM_segment_end__    = 0x20010000;
  __FLASH_segment_start__ = 0x00000000;
...
```

Code c-1 Typical memory configuration for a stand-alone application

In order to locate the application code at a different start address it is necessary to modify the segment start as in the following example:

```
MEMORY
{
    FLASH (rx) : ORIGIN = 0x00000000, LENGTH = 0x0003f000
    SRAM (wx)  : ORIGIN = 0x20000000, LENGTH = 0x00010000
}

SECTIONS
{
  __SRAM_segment_start__  = 0x20000000;
  __SRAM_segment_end__    = 0x20010000;
  __FLASH_segment_start__ = 0x00002000;
...
```

Code c-2 Modified linker script for use with the serial loader

The application's start up code is now located from 0x2000 (8k), assuming that this is suitable for the Luminary Micro serial loader.

d) AVR32 - AT32UC3B0256 on EVK1101 using IAR

The ATMEL AVR32 EVK1101 has an RS232 connector on USART1. This can be configured as interface to use to download the new SREC file.

It is recommended to use `TeraTerminalPro` for serial loading (further details about this program are also contained in the document above).

The position of the application code is controlled by the linker script file (*.xcl) as used by the target configuration. A typical configuration for the application linked to the start address 0x80000000 is shown below:

```
-Z(CODE)RESET=80000000-800003FF
-Z@(CODE)EVTAB=80000100-8003FFFF
-Z@(CODE)EV100=80000100-8003FFFF
-P(CODE)EVSEG=80001000-8003FFFF
-P(CODE)CODE32=80000000-8003FFFF
-P(CONST)DATA32_C=80000000-8003FFFF
-Z(CONST)INITTAB,DIFUNCT=80000000-8007FFFF
-Z(CONST)CHECKSUM,SWITCH=80000000-8007FFFF
-Z(CONST)DATA21_ID,DATA32_ID=80000000-8007FFFF
-Z(CONST)RAMCODE21_ID,RAMCODE32_ID=80000000-8007FFFF
-Z(CONST)ACTAB,HTAB=80000000-8007FFFF
```

Code d-1 Typical memory configuration for a stand-alone application

In order to locate the application code at a different start address it is necessary to modify the segment start as in the following example:

```
-Z(CODE)RESET=80002800-80002aff
-Z@(CODE)EVTAB=80002900-8003FFFF
-Z@(CODE)EV100=80002900-8003FFFF
-P(CODE)EVSEG=80003800-8003FFFF
-P(CODE)CODE32=80002800-8003FFFF
-P(CONST)DATA32_C=80002800-8003FFFF
-Z(CONST)INITTAB,DIFUNCT=80002800-8007FFFF
-Z(CONST)CHECKSUM,SWITCH=80002800-8007FFFF
-Z(CONST)DATA21_ID,DATA32_ID=80002800-8007FFFF
-Z(CONST)RAMCODE21_ID,RAMCODE32_ID=80002800-8007FFFF
-Z(CONST)ACTAB,HTAB=80028000-8007FFFF
```

Code d-2 Modified linker script for use with the serial loader

The application's start up code is now located from 0x2800 (10k), assuming that this is suitable for the AVR32 serial loader.

Further, note that the stack pointer start location is also defined in the linker script. It is taken from the RAM settings, which should correspond to the size of the available SRAM in the device being used.

```
-Z(CODE)RAMCODE21=00000105-00007FFF
-Z(DATA)DATA21_I,DATA21_Z,DATA21_N=00000105-00007FFF
-Z(CODE)RAMCODE32=00000105-00007FFF
-Z(DATA)DATA32_I,DATA32_Z,DATA32_N=00000105-00007FFF
-Z(DATA)TRACEBUFFER=00000105-00007FFF
-Z(DATA)SSTACK+_SSTACK_SIZE#00000105-00007FFF
-Z(DATA)CSTACK+_CSTACK_SIZE#00000105-00007FFF
-Z(DATA)HEAP+_HEAP_SIZE=00000105-00007FFF
```

Code d-3 Linker script configured for 32k SRAM

The AVR32 requires the use of an intermediate buffer (see chapter 6). Its size can be typically set to 16k for a processor type with 32k of internal SRAM, whereby 16k is reserved for the intermediate binary buffer and another 4k by the UART's input buffer.

When working with projects with IAR, make sure that the correct linker script is selected for the target (for example `lnkuc3b0256_bm.xcl` for the AT32UCB application, or `lnkuc3a0512_bm.xcl` for the AT32UCA application) with the correct linker start address to match the loader used. When debugging with IAR make sure that the linker option is set to generate debug information for C-Spy, including all Module-local symbols. In the compiler setting ensure that the output generates debug information. In the general options select the exact target device being used – this will ensure that the target can be correctly connected to and source level debugging works correctly.

Appendix B – Application Requirements for use together with the µTasker Serial Loader

In order for an application to be able to operate together with the µTasker Serial Loader there are several points to be considered. Generally there should be no difficulty to achieve compatibility as long as the guide is followed:

1. The application must be linked to start at the application defined by the serial loader. The start of this code must include the standard reset vector table so that the jump from the serial loader to the application can take place as if it were a normal reset entry. For ARM Cortex M processors this means that the first long word address holds the value of the initial stack pointer and the second the initial program counter.
2. The µTasker Serial Loader will configure the watchdog operation; either enable it or disable it as configured in its code. If the watchdog is enabled the application must service it to avoid a reset when it fires or disable it if this is allowed on the HW being used.
3. The µTasker Serial Loader configures processor clocks and jumps to the application with clocks configured at their defined operating speed. Either the application an continue using this setting or an adjust to suit its final requirements; the code used to do this should consider that the clocks/PLLs may already be configured and not cause subsequent errors by assuming the reset state.
4. The µTasker Serial Loader jumps to the application with interrupts disabled. All peripherals are essentially set to their default states, although may have been temporarily used.
   Exceptions: The GPIO used by the µTasker Serial Loader used for forced loading, watchdog operation and run LED may be programmed when the application starts up. The application should therefore not assume the state of these ports at reset.
5. The µTasker Serial Loader will have configured any flash configurations since it occupies the physical reset area. Configurations in the application area are generally superfluous and can be removed if desired (to save code space).
6. Since the µTasker Serial Loader always runs to check the state of the application code its variables will also be configured. This means that RAM will not have random values as is normally the case after a power on reset.
7. The µTasker Serial Loader will generally occupy the default interrupt vector table. This means that the application needs to ensure that its interrupt vectors are configured accordingly for their physical location when in Flash. When these are relocated to RAM as standard operation there is no consequence. See also the link below or more details on this topic.

The following is a guide for Teensy 3.1 Arduino/Teensyduino users to allow their application to be used together with the uTasker Serial Loader:

http://www.utasker.com/forum/index.php?topic=1869.0

Appendix C – Serial Loader Mailbox, Ensuring its Application Settings and Enabling Security

One question that is fundamental to the use of the serial loader is when it should operate (when there is an application already present)?

This is controlled by the macro `FORCE_BOOT()` which can be `(1)` so that it always starts but is often the check of a digital input to see whether the user wants it to be used or not. Various other possibilities could however be more suitable for a particular application, meaning that a mixture of various factors could also exist.

Although it is always recommended to have an input to force the serial loader to start that can be used in an emergency, the use of the Serial Loader Mailbox is an option that can be used to allow the application to make the decision as to whether the loader be started or not. It works by the application placing a command in an SRAM location that is otherwise never used by the system, commanding a reset, and the serial loader detecting this valid command.

The location reserved for this is the *very last short word in SRAM* and to preserve this, both the serial loader and the µTasker application always set their stack pointers to at least one long word below the top of SRAM in order to guarantee that the last one is never used (and potentially corrupted in the process). *A non-µTasker application must therefore also ensure this in order to be able to reliably work with the Serial Loader Mailbox technique*.

This is an example of how the application commands the serial loader to start:

```
*BOOT_MAIL_BOX = RESET_TO_SERIAL_LOADER;
fnResetBoard();
```

whereby the first line sets the command (defined as `0x89a2`) to the mailbox entry (last unsigned short location in SRAM – see the defines in the hardware headers for full details on the target in question).
The second line is the standard reset command, which commands a core reset.

Assuming the serial loader uses an input for basic control (eg.):

```
#define FORCE_BOOT()   (_READ_PORT_MASK(E, SWITCH_3) == 0)
```

its decision can be adapted to include the mail box as follows:

```
#define FORCE_BOOT()    (((SOFTWARE_RESET_DETECTED()) && (*BOOT_MAIL_BOX ==
RESET_TO_SERIAL_LOADER)) || (_READ_PORT_MASK(E, SWITCH_3) == 0))
```

The `SOFTWARE_RESET_DETECTED()` macro checks to see whether the last reset was a commanded reset (rather than a power up reset, watchdog reset, etc., in which case it would not consider the mail box value). The result is thus a reliable detection of the command, plus the original option of using an input too.

Once the serial loader has checked the mailbox value it always resets it to ensure that it can not be recognised multiple times!

The µTasker project doesn't use the linker script to configure the stack if at all possible (which would be difficult to maintain and keep portable) but instead does this in code. The Cortex-M4 reset vector is show as reference:

```
    (void *)(RAM_START_ADDRESS + (SIZE_OF_RAM - NON_INITIALISED_RAM_SIZE)),
                                                 // stack pointer to top of RAM
    (void (*)(void))START_CODE,                  // start address
```

whereby the first entry is the initial stack pointer value that the processor loads and the second is the initial program counter (typically the address of `main()`).

Rather than setting the stack pointer value to the top of the SRAM area it is set to a lower value, governed by `NON_INITIALISED_RAM_SIZE`. This is *at least* 4 in an application and at least 16 for the serial loader and includes an optional user value called `PERSISTENT_RAM_SIZE`, which defaults to 0.
The user can add a non-zero project definition for `PERSISTENT_RAM_SIZE` if a larger area of untouched SRAM could be of benefit for the application in question.

When the serial loader is set up the start address of the application and the size of the application are two important parameters, which are both defined, for a particular target, in `Loader.h`. Examples are:

```
#define UTASKER_APP_START     (32 * 1024)       // application starts at this address

#define UTASKER_APP_END       (unsigned char *)(UTASKER_APP_START + (128 * 1024))
                           // end of application space - after maximum application size
```

Obviously the application is expected to be at the address 0x8000 (32k) and its maximum size is 128k.

> *Beware however that when USB-MSD is in use the application start address is in fact a file object and the application itself starts after this file object (often 0x80 bytes further and thus 0x8080 in this example). See*
> ```
> #define _UTASKER_APP_START_    (UTASKER_APP_START + (ROOT_FILE_ENTRIES * 32))
> ```
> *for details.*

The choice of application start (or link) address is generally dictated by the **size of the serial loader** itself because the application must start at an address beyond the end of the serial loader and also in a Flash sector that is not shared with the serial loader's code. If the serial loader is only 7k in size and the flash granularity were to be 4k this would give 8k as the typically optimal application start address (not leaving unused sectors between the two).

If the remaining Flash is to be used exclusively by the application program the application size can be set to

```
#define UTASKER_APP_END    (unsigned char *)(SIZE_OF_FLASH)    // end of application
space - after maximum application size
```

however, in many instances the application may be using the Flash also for saving parameters (eg. µParameterSystem) to or for a file system (eg. µfileSystem) and so space for these must be reserved after the maximum application area.

This means that the decision for maximum application size is project specific and a compromise between the amount of Flash preserved (not deleted when new code is loaded) and the maximum application size that is ever expected in the future. Its choice should thus be considered carefully!

It is to be noted that Kinetis processors have their Flash configuration (which can include security settings to protect Flash areas or block reading Flash content using a debugger) in the first Flash sector between addresses 0x400 and 0x40f. Therefore it is the serial loader that delivers this setting and not the application.

When security for program code is required the serial loader must be configured correctly (and not the application, which can't influence it). An example of the default, non-protected flash security setting is

```
#define KINETIS_FLASH_CONFIGURATION_SECURITY   (FTFL_FSEC_SEC_UNSECURE |
FTFL_FSEC_FSLACC_GRANTED | FTFL_FSEC_MEEN_ENABLED | FTFL_FSEC_KEYEN_ENABLED)
```

which can be changed to

```
#define KINETIS_FLASH_CONFIGURATION_SECURITY    (FTFL_FSEC_SEC_SECURE |
FTFL_FSEC_FSLACC_GRANTED | FTFL_FSEC_MEEN_ENABLED | FTFL_FSEC_KEYEN_ENABLED)
```

to achieve this.

Beware that some programming tools don't allow setting security by default (they will modify this byte to avoid securing devices by mistake) and if this turns out to be the case, consult the tool manufacturer's manual for details of overriding it.

Furthermore, Teensy users with HalfKay programming devices on board should carefully evaluate security settings since it can't be guaranteed that certain configurations may cause this loading device to no longer operate.