



μTasker Document

**μTasker – Serial Loader User's Guide
(including USB-MSD)**

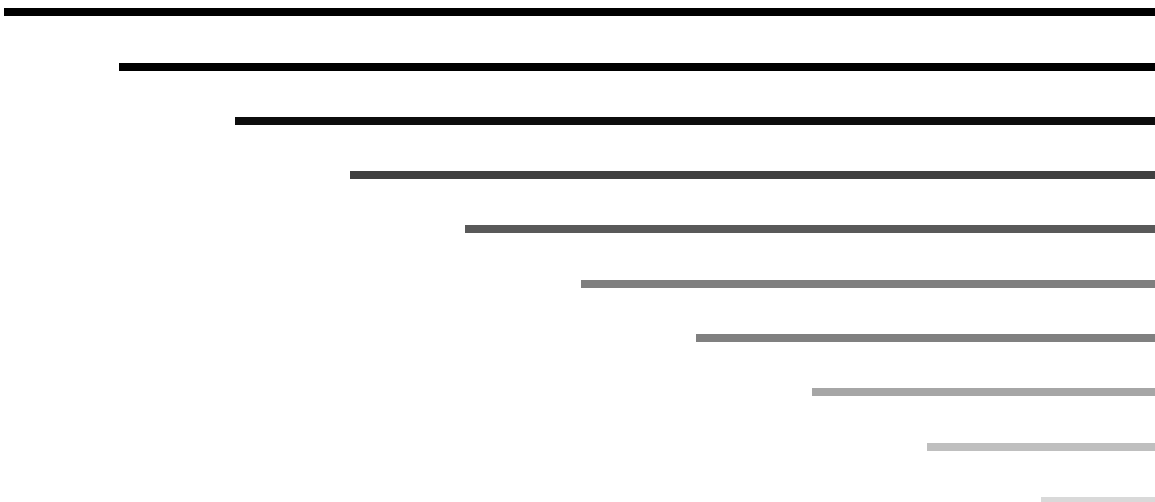


Table of Contents

1. Introduction.....	3
1.1 USB_MSD Operation	3
2. Programming the µTasker Serial Loader.....	3
3. Building the µTasker Serial Loader	4
4. Using the µTasker Serial Loader.....	4
5. Configuring the µTasker Serial Loader Project.....	8
6. Intermediate Buffer Required for Certain Processor Types	10
7. Preparing an Application and working with the Serial Loader.....	11
8. USB-MSD Boot Loader	12
9. Conclusion	14
Appendix A – Target and Compiler Specific Details.....	15
a) Coldfire CodeWarrior.....	15
b) Luminary-Micro Evaluation Boards and GCC Compiler	17
c) AVR32 - AT32UC3B0256 on EVK1101 using IAR.....	18

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. 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, which is where the **µTasker Serial Loader** can find practical and important use.

The **µTasker Serial Loader** doesn't require a special software tool to be installed on the PC used to load the software but instead uses a general purpose terminal emulator. The loading format is the standard Motorola S-REC format as can be output by all compiler packages.

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 target and compiler specific details.

1.1 USB_MSD Operation

The **µTasker Serial Loader** has been extended to include USB MSD (mass storage device) mode of operation when the processor has a built-in USB device interface. The USB-MSD support can be used parallel to the SREC serial support or can be used instead of the SREC serial mode.

USB-MSD mode was chosen due to the fact that all PC operating systems include a standard MSD 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.

Details of this mode are given after the SREC serial loader description.

2. Programming the µTasker Serial Loader

The **µTasker Serial Loader** is available as pre-compiled object for loading to various 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 both SREC serial and USB-MSD 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`.

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

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>. This is an excellent program which has proved to be very reliable on various Windows platforms, including Vista and Windows 7.

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 sikilar to the following:

```
uTasker Serial Loader  
=====  
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.

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.

5. 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. The serial interface and µFileSystem are activated and these settings should generally not be changed.

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 flashed 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    (8*1024)
```

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 than 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 bit and XON/XOFF flow control. The µTasker Serial Loader has 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.

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.

7. 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.

The application programmer may need to consider the following points to ensure full compatibility:

1. The µTasker Serial Loader will set the CPU clock as determined by its configuration. This may mean that the PLL is already configured when the application starts, either with the desired frequency or a different one. The application configuration should therefore be prepared that the PLL is not necessarily in its reset configuration state. If the PLL frequency matches the speed used by the application, the application doesn't need to perform any initialisation of its own!
2. If the µTasker Serial Loader is configured to run with active watchdog the watchdog will already be active when the application is called. If the application doesn't service the watchdog the serial loader should not be configured for watchdog use since it will subsequently fire once the application starts.
3. 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.
4. 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.
5. 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.

8. USB-MSD 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` and can operate either in parallel with the SREC serial boot loader mode or else as alternate to it – this depends on the define `SERIAL_INTERFACE` in `config.h`.

When the USB interface is enabled, the application file `usb_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.

Since the actual operation 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 unicode 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.

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-authorized 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
```

9. Conclusion

The µTasker Serial Loader may be programmed to a target board and used as an in-circuit tool for loading application software via a UART. 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.

This document has illustrated the operation of the serial loader 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.

V1.0 04.06.2011: First release including USB-MSD operation

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 correctly 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);
        .                = 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) 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 b-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 b-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.

c) 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 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:

```
-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 c-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 c-23 Linker script configured for 32k SRAM

The AVR32 require 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.