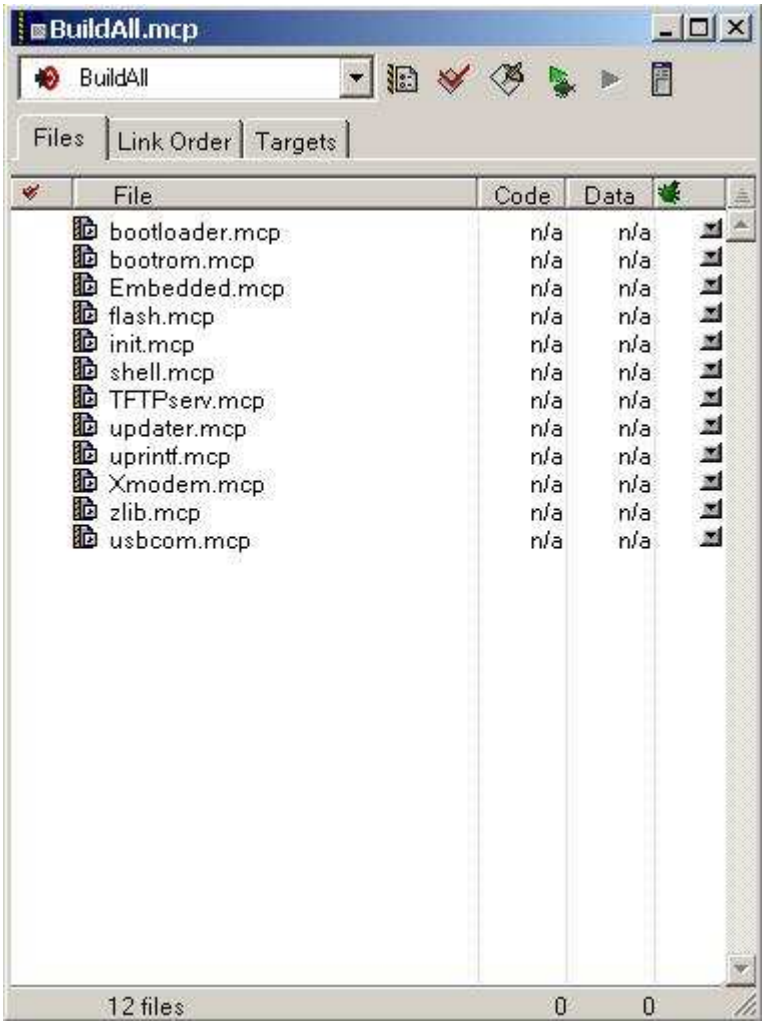
NUC700 Bootloader Source Code Modification Guide

This document only describes the bootloader source code structure. Please refer to bootloader user's manual for the usage of bootloader.

1. Introduction to Bootloader Source Code

NUC700 bootloader is developed in ADS 1.2 environment. The whole bootloader project is included in <Bootloader source code path>\BuidlAll\BuildAll.mcp, programmer should open this project to modify or build the sub-projects of bootloader. After open the BuildAll.mcp project file, following project window should be shown in CodeWarrior IDE.



As you can see, BuildAll.map contains several sub-projects and each of them contains two or more targets, following table describe the function of each target.

Project	Target	Description

init.mcp	little	Little endian code to initialize target board
	big	Big endian code to initialize target board
Flash.mcp	flash_li	Little endian mode flash erase/write library Programmer should edit this code for support other new flash.
	flash_bi	Big endian mode flash erase/write library
	flash_demo	Little endian demo program using flash_li, programmer could use this code verify the library
Shell.map	shell_li	Little endian shell library support TFTP
	shell_bi	Big endian shell library support TFTP
	shell_usb_li	Little endian shell library support USB (*)
	shell_usb_bi	Big endian shell library support USB (*)
Tftpserv.map	tftpserv_li	Little endian Network driver and TFTP library
	tftpsev_bi	Big endian Network driver and TFTP library
	little	Little endian TFTP demo program
	big	Big endian TFTP demo program
Uprintf.map	uprintf_li	Little endian UART library
	uprintf_bi	Big endian UART library
	little	Little endian UART demo code
Xmodem.mcp	xmodem.li	Little endian Xmodem library
	xmodem.bi	Big endian Xmodem library
	Little	Little endian Xmodem demo program
	Big	Big endian Xmodem demo program
Zlib.mcp	zlib.li	Little endian decompress library
	zlib.bi	Big endian decompress library
Usbcom.mcp	usbcom_li	Little endian USB library (*)
	usbcom_bi	Big endian USB library (*)
Embedded.mcp	embedded_li	Little endian system initialize library for stand alone application
	embedded_bi	Big endian system initialize library for stand alone application
	demo	Demo code using embedded_li
bootloader.map	little	Little endian mode bootloader

	big	Big endian mode bootloader
Updater.mcp	little	Little endian mode updater
	big	Big endian mode updater
	semihosted_little	Semihosted little endian mode updater (used with ICE)
	semihosted_big	Semihosted big endian mode updater (used with ICE)

(*): Not included in NUC740 bootloader source code

The all object files was removed form Nuvoton's bootloader source code package before delivering to customers, hence programmer should re-build the project according to the dependency. First build all the libraries, which including flash, shell, tftp, USB, uprintf, Xmodem, and zlib libraries. Then build the bootloader, and build the updater last. Some libraries also come along with demo application. Programmer could use it to test the function of those libraries.

The entry point of bootloader is in init.s. This assembly code will do following task sequentially. Disable interrupt, disable cache, set clock skew, remap memory, and configure SDRAM, set stack pointer, and then jump into __main function, which will call main function later.

Right after boot up, all codes of bootloader resides in ROM, but before entering main function, some of the code and data will be copied to SDRAM by __main(), the location of each code and data section is decided by the scatter load file in the bootloader project. The vector table is also copied to address 0 during this process.

In main function's job is very simple. It decides the total size of SDRAM, set up UART0 as console, check the on board boot ROM type and read the bootloader configuration from image 0, setup the stack and heap boundary for applications. If ESC key is not pressed in three seconds after boot up, main function will call BootProcess() to process the images in flash. And in last step, main function will call sh() to enter debug shell.

Bootloader provides some SWIs. User applications can interact with bootloader via SWI. For example, user applications use SWI to get stack and heap boundary from bootloader. If bootloader call printf() function, the output data is passed to bootloader via SWI, and bootloader will send the data to UART0. If an ICE is connected with the target board, debug

software may intercept the SWIs and service them. In this case, the SWI request will not be passed to bootloader. These SWI handlers are store in switrap.s, and exception numbers are store in except_h.s

Bootloader implements its own heap manager. It will use the memory space assigned in image 0 as the heap pool. This heap manager will provide dynamic allocate memory for network driver, decompress library and semihosted program. The heap manager is implements in heap.c.

Please be noted, the code size of bootloader must be less than 64kB, and the code/ data reside in RAM could not exceed 0x8000 minus total stack size (currently it is 1024 bytes defined in init.s). Due to the 64kB limitation, bootloader can not support USB and MAC at the same time, programmers could select the interface to support by choosing the different target in shell.map. The TFTP library and USB library will not be linked into bootloader at the same time.

2. Modify Bootloader Source Code

Bootloader source code may need some modification to make it work on different hardware. Below list how to make these modifications, which include modifying SDRAM size, supporting new NOR flash, adding new shell command.

To support a new shell command, two files needed to be update, command.c in bootloader.mcp and sh.c in shell.mcp. Add the command function in shell.c, which shall follow the format: int foo(int argc, char *argv[]), and add an entry for the new command in NU_commands[] array declared in command.c.

To support new NOR flash, programmers should add the write/ erase/ lock/ block size/ read PID/ functions for the new flash and add these function as well as the vendor ID and product ID to an new entry in flash[] in flash.c. The prototype of these functions could be found in structure flash_t defined in flash.h.

Write function is used to write one block of data to flash. Erase function is used to erase a block in flash. Lock function is used to lock or unlock a bock in flash. For certain flash types, all blocks are locked after power on by default, so write/ erase function needs to call this function before they can really do their jobs. Block size function is used to return the block size starting from an address assigned by function’s parameter. Read PID function is used to

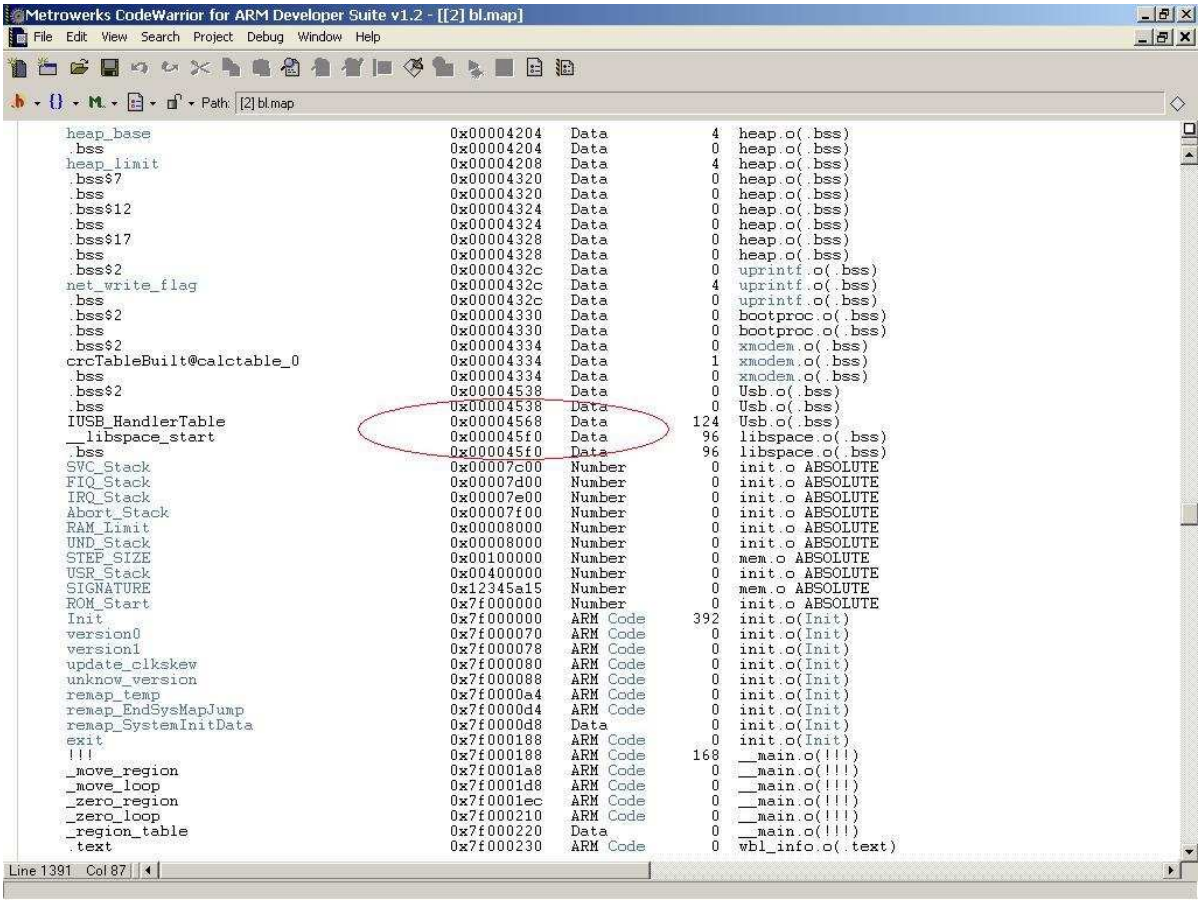
read the vendor ID and product ID from NOR flash to decide the flash size and the access functions for this flash. Many flashes are already supported by bootloader, and the access functions of new flash are likely to be the same with some supported flash, in this case, programmers can use the existing functions in the new entry of flash[] array.

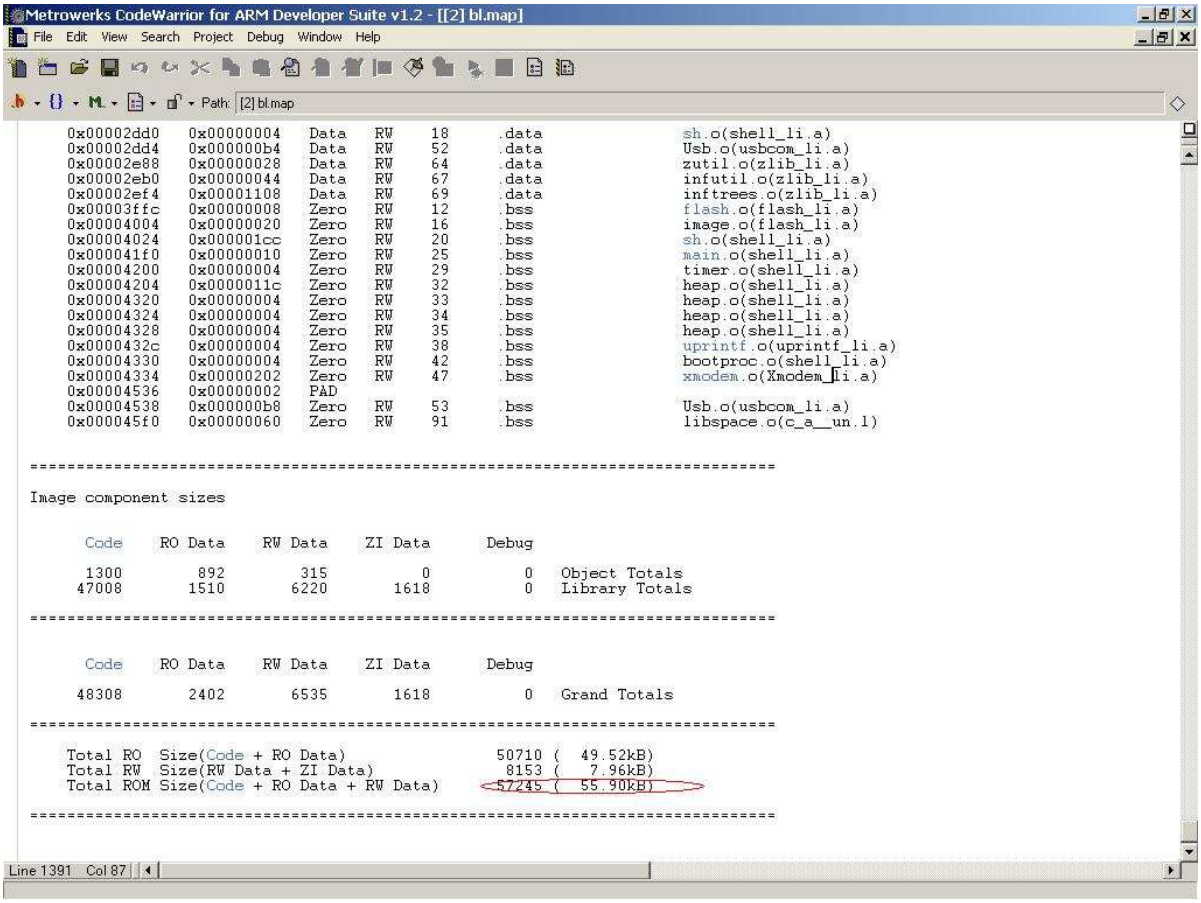
Bootloader decides if the write/ erase are complete by polling flash's data pin. But not every flash has the same polling mechanism. A new polling function might need to be provided for write/ erase function to decide the flash's status.

Bootloader will try to decide the SDRAM size and type on board in init.s, but it cannot detect all configurations correctly. In case of using a configuration not support by bootloader, programmers need to modify init.s to make bootloader boot up correctly. To do so, the code after label "remap_EndSysMapJump" and before command "B %FT0" should be removed and the correct SDRAM setting should be filled into data section right after label "remap_SystemInitData". These fields are the value to be written to EBI control registers from EBICON to SDTIME1. Programmers should check to datasheet to decide the correct value of these registers.

As mentioned in previous section, total code size of bootloader cannot exceed 64kB and code/ data reside in RAM could not exceed 0x8000 minus total stack size Programmers should check the memory map every time they re-build the bootloader and make sure the binary code meets such constrain. Total code size is listed at the bottom of the map file. And the RAM usage of global symbols, local symbols, are scattered in different parts of map file.

Follow images are partial of a map file, the top most address of local symbol resides in RAM and total code size could be found in it.


Metrowerks CodeWarrior for ARM Developer Suite v1.2 - [[2] bl.map]
File Edit View Search Project Debug Window Help
Path: [2] bl.map
heap_base 0x00004204 Data 4 heap.o(.bss)
.bss 0x00004204 Data 0 heap.o(.bss)
heap_limit 0x00004208 Data 4 heap.o(.bss)
.bss$7 0x00004320 Data 0 heap.o(.bss)
.bss 0x00004320 Data 0 heap.o(.bss)
.bss$12 0x00004324 Data 0 heap.o(.bss)
.bss 0x00004324 Data 0 heap.o(.bss)
.bss$17 0x00004328 Data 0 heap.o(.bss)
.bss 0x00004328 Data 0 heap.o(.bss)
.bss$2 0x0000432c Data 0 uprintf.o(.bss)
net_write_flag 0x0000432c Data 4 uprintf.o(.bss)
.bss 0x0000432c Data 0 uprintf.o(.bss)
.bss$2 0x00004330 Data 0 bootproc.o(.bss)
.bss 0x00004330 Data 0 bootproc.o(.bss)
.bss$2 0x00004334 Data 0 xmodem.o(.bss)
crcTableBuilt@calctable_0 0x00004334 Data 1 xmodem.o(.bss)
.bss 0x00004334 Data 0 xmodem.o(.bss)
.bss$2 0x00004538 Data 0 Usb.o(.bss)
.bss 0x00004538 Data 0 Usb.o(.bss)
IUSB_HandlerTable 0x00004568 Data 124 Usb.o(.bss)
__libspace_start 0x000045f0 Data 96 libspace.o(.bss)
.bss 0x000045f0 Data 96 libspace.o(.bss)
SVC_Stack 0x00007c00 Number 0 init.o ABSOLUTE
FIQ_Stack 0x00007d00 Number 0 init.o ABSOLUTE
IRQ_Stack 0x00007e00 Number 0 init.o ABSOLUTE
Abort_Stack 0x00007f00 Number 0 init.o ABSOLUTE
RAM_Limit 0x00008000 Number 0 init.o ABSOLUTE
UND_Stack 0x00008000 Number 0 init.o ABSOLUTE
STEP_SIZE 0x00100000 Number 0 mem.o ABSOLUTE
USR_Stack 0x00400000 Number 0 init.o ABSOLUTE
SIGNATURE 0x12345a15 Number 0 mem.o ABSOLUTE
ROM_Start 0x7f000000 Number 0 init.o ABSOLUTE
Init 0x7f000000 ARM Code 392 init.o(Init)
version0 0x7f000070 ARM Code 0 init.o(Init)
version1 0x7f000078 ARM Code 0 init.o(Init)
update_clkskew 0x7f000080 ARM Code 0 init.o(Init)
unknow_version 0x7f000088 ARM Code 0 init.o(Init)
remap_temp 0x7f0000a4 ARM Code 0 init.o(Init)
remap_EndSysMapJump 0x7f0000d4 ARM Code 0 init.o(Init)
remap_SystemInitData 0x7f0000d8 Data 0 init.o(Init)
exit 0x7f000188 ARM Code 0 init.o(Init)
!!! 0x7f000188 ARM Code 168 __main.o(!!!)
_move_region 0x7f0001a8 ARM Code 0 __main.o(!!!)
_move_loop 0x7f0001d8 ARM Code 0 __main.o(!!!)
_zero_region 0x7f0001ec ARM Code 0 __main.o(!!!)
_zero_loop 0x7f000210 ARM Code 0 __main.o(!!!)
_region_table 0x7f000220 Data 0 __main.o(!!!)
.text 0x7f000230 ARM Code 0 wbl_info.o(.text)
Line 1391 Col 87



3. Using Embedded Library

The sub-project Embedded is not used by bootloader. It contains the code necessary to initialize the hardware, provide SWI service, and setup vector table. If for some project that bootloader is not required, programmers could link this library to their application, and write the image to the starting address of boot flash. There's a demo application in this project shows how to link with this library.

Revision History

Version	Date	Description
1.0	Mar. 12 2009	Init. Version