M16C/62P

Example Serial Flash Bootloader With Xmodem Data Transfer Using IAR C-Compiler

Introduction

All Renesas Flash microcontrollers have the ability to self-program their Flash memory. In order to perform this task code must run on the microcontroller to receive the data from an external source, load it into the Flash and manipulate the Flash control registers. The program that performs this task is typically called a bootloader.

Although Renesas M16C microcontrollers have a built-in bootloader called 'standard serial I/O mode' it can sometimes impose constraints on the system being designed. For example, the data transfer must be by a specific serial channel on the device and the mode must be entered via setting a combination of pins to predetermined logic levels and resetting the micro. The M16C Flash devices do offer the ability for the user to change this serial I/O code but this must be done via an external programmer unit in parallel I/O mode. When a different communications medium must be used or when the boot code must be entered via software a user provided bootloader is required. It is the purpose of this apps note to describe the concepts behind such a bootloader and its implementation using a Renesas M16C/62P microcontroller and the 3-D Kit platform.

This implementation uses a UART channel as the method to communicate with the device for simplicity. The bootloader is intended to be menu driven from a PC based terminal program with the data to be programmed being transferred via the Xmodem comms protocol. This provides an easy to understand and implement demonstration, although it is expected that the user may need to change this for inclusion in a final application.

The project, including all source code, is available for download with this apps note. The application has been developed using the M16C IAR C compiler, M16C KD30 ROM monitor debugger and tested using a Renesas M16C/62P 3-D Kit.

It is recommended that this apps note is read in conjunction with application note REG05B0021-0100.

Contents

The Application

This application note by means of a demo application covers many of the concepts and considerations required when developing a bootloader. The hardware used is an M16C/62P 3-D Kit with an additional RS232 driver and connector added in order to connect UART0 to the host machine running terminal emulation software. This allows KD30 ROM monitor debugging to be possible. Figure 1 shows a block diagram representation of the hardware.

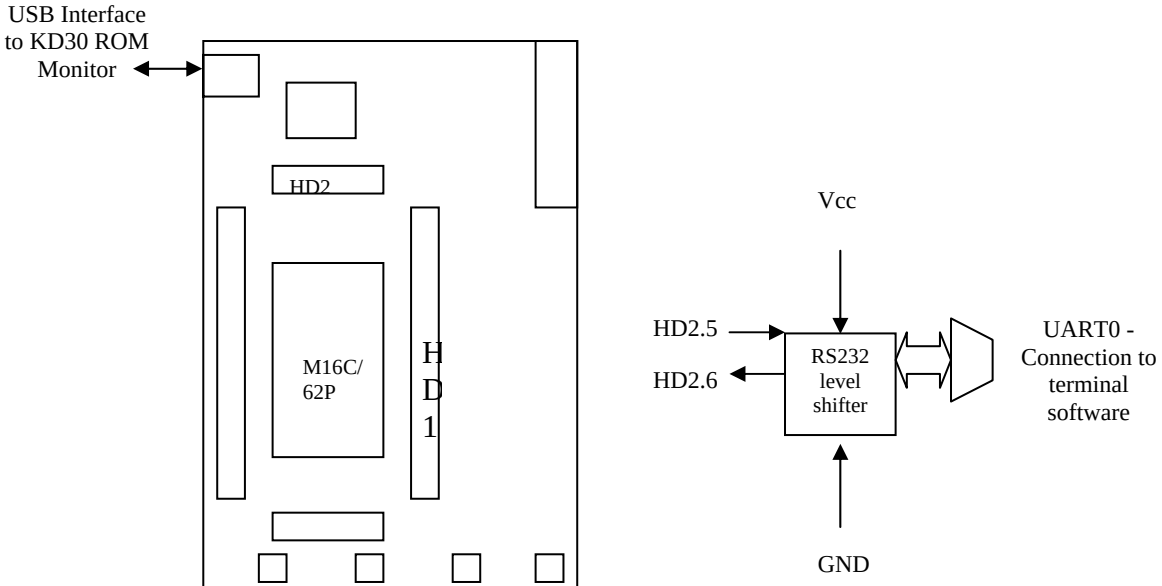

Figure 1: M16C/62P 3-D Kit Based System Block Diagram

When the application starts it configures UART0 to 115200 baud 8-N-1 and listens for any serial activity for 3 seconds. If no activity is detected then the reset address in the user code area (0xA0020) is read and if valid, i.e. not the erased state (0xFFFFFFFF), it is used as the reset vector and program execution continues at the address pointed to by this vector. Otherwise, if serial activity is detected it is assumed that a terminal emulator has generated the traffic, by someone hitting the space bar for example, and a menu is displayed as shown in figure 2.

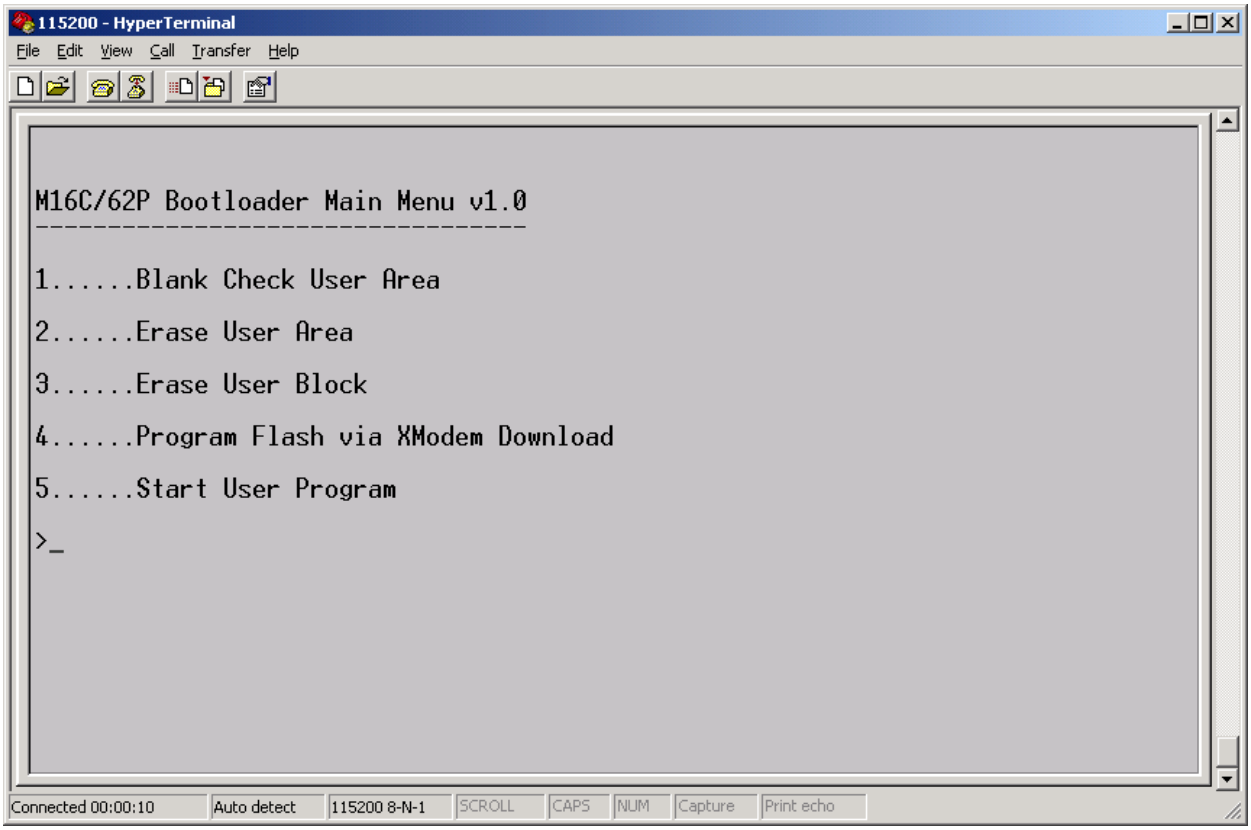

Figure 2: Bootloader Terminal Window

The menu options should certainly need no explanation to what they do rather how they do it. This will be covered in the rest of this application note.

Figure 3 shows the memory map of the bootloader and target program memory areas.

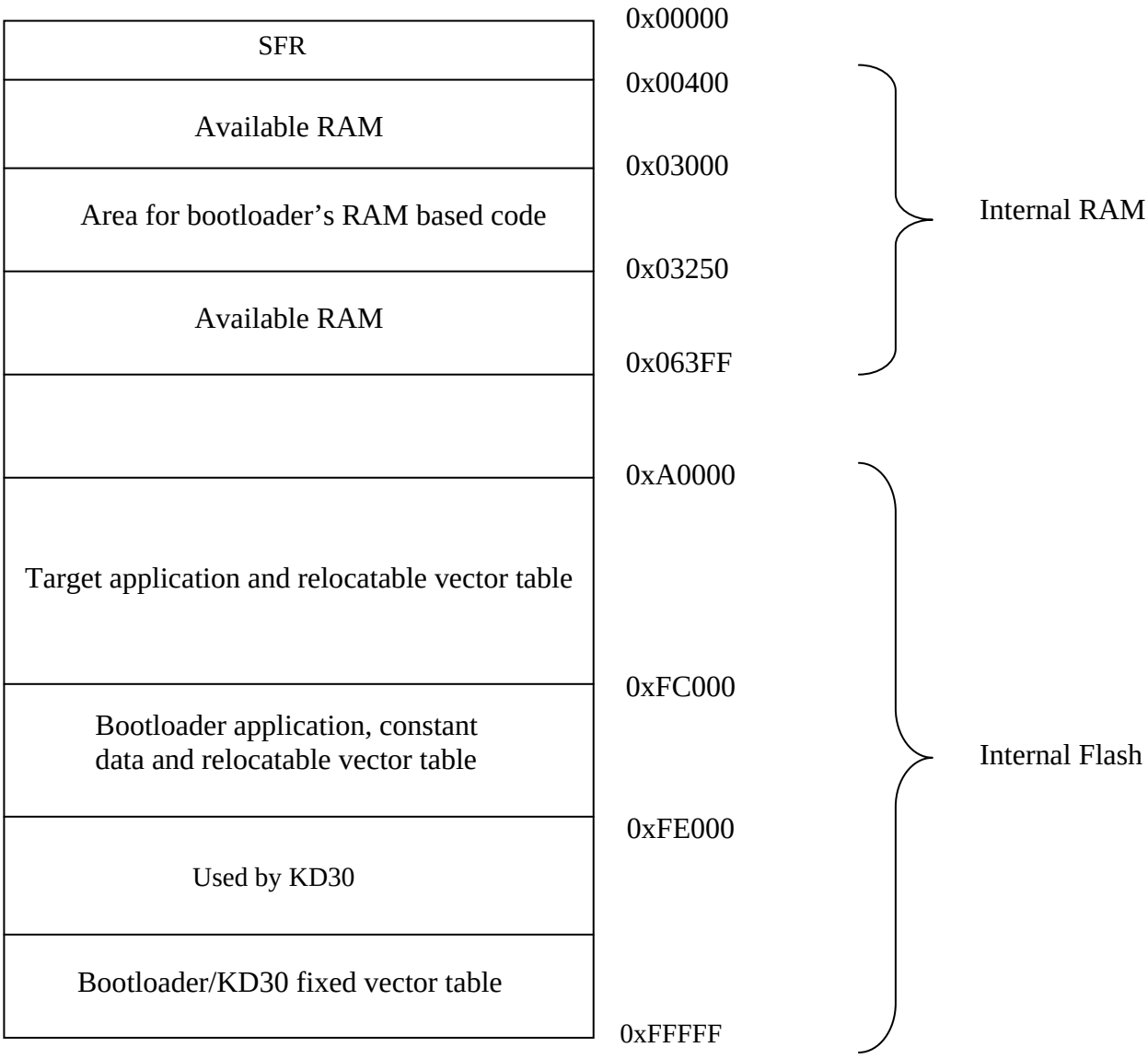

Figure 3: Bootloader and Target Application Memory Map

As can be seen from figure 3, the main bootloader application code sits in the Flash memory between addresses H'FC000 and H'FDFFF. The Flash located between H'FE000 and H'FFFDB is used by the KD30 ROM monitor. If the bootloader is not used with KD30 then this Flash can be used by the bootloader making more memory available for the target application. This leaves 368kB of internal Flash memory available between addresses H'A0000 and H'FBFFF for the user's target application.

The bootloader uses CPU rewrite mode 0 when erasing and programming the internal Flash memory. A detailed explanation of CPU rewrite modes 0 and 1 can be found the M16C/62P series hardware manual. CPU rewrite mode 0 was chosen as it allows the bootloader to be easily replaced if required. CPU rewrite mode 0 requires that the code performing the erasing and programming of the Flash memory executes somewhere other than the Flash memory. In this application this code executes from internal RAM. Approximately 600 bytes of RAM are required for this erasing and

programming functionality starting at address H‘3000. This address has been chosen in order to guarantee no user variables, KD30 data or stack are over written. If KD30 is not being used then this address can be changed if required.

Building Application Code for Execution from Internal RAM

When programming or erasing the internal Flash memory of Renesas’ M16C/62P series in CPU rewrite mode 0 code must execute from outside of the Flash memory. Typically this means from internal RAM. At first the solution to the problem of how to do this seems straight forward enough. At runtime copy the program or erase routine from Flash to RAM and call it via a function pointer. Better still, if using the M16C IAR compiler, use the ‘-Q’ scatter loading linker directive. In many cases these methods will work but unfortunately they cannot be guaranteed. The reason being that any jumps within the code or to subroutines may refer to absolute addresses.

The solution to this problem is to link the code that must run from RAM to the actual RAM addresses at build time. This can introduce further problems. The first is that of library routines. If a RAM based function is part of a larger project, such as a bootloader, then it may happily run from RAM but include calls to library routines that are linked to Flash addresses causing accesses to Flash memory at undesirable moments during execution. Even something as innocuous as the C statement below can result in a library call.

```
i = 1 << some_variable;
```

Therefore, simply looking through the C source and avoiding calls to functions such as ‘printf’ is not enough to guarantee that there are no library calls to Flash based routines.

The second issue concerning copying functions from Flash to RAM is that of constant data. If the RAM routine makes reference to constant data, including items such as string literals, this can cause the Flash memory to be accessed.

A third consideration is how to get code that is linked to RAM into Flash for storage at build time and then back into RAM at runtime for execution.

A solution to these problems is to place the RAM based routines into a completely separate project with all code, variable and constant data linked to the RAM addresses. This eliminates the problems of jumps back into Flash for code, libraries and constant data. Getting this code from the RAM addresses into the Flash for storage at build time can be achieved by using the ‘motice_cl’ utility and method described in application note REG05B0021-0100. Please see this apps note for further information on this utility.

This utility converts a s-record file into a constant C structure. For example, the erase and programming functions are built as a separate project and linked to RAM. The linker is configured so that it outputs a s-record file for this project. This file is processed by ‘motice_cl’ which turns it into a constant C structure which can then be included in the bootloader project. As the structure is constant data it resides in the Flash. When the erase or program function is to be called by the bootloader the constant structure data is copied to the correct place in RAM and called by a function pointer. While the called function is executing only RAM is accessed for program code and data as this is all the routine knows about as it has been linked to RAM addresses as a separate project.

The above method relies on 3 things being known at runtime. These are:

1. The start address that the RAM code should be copied to from Flash. This is achieved by storing the constant data as part of a structure which contains the start address (put there by 'motice_cl' from the s-record) and the length of the data.
2. The size of the data to be copied to RAM so the copying routine knows how much data to move. See the explanation above for how this is known.
3. If the RAM based code contains multiple functions, e.g. erase and program routines, the start addresses for these functions must be known so they can be correctly called by function pointers. This is achieved by loading these addresses into a function pointer table starting at the beginning of the RAM code area. Althought the addresses of the functions may change, the location of where the value and order of these are installed does not and is known by the bootloader. So, all the bootloader must do is read the correct address and call the function via a pointer.

Figure 4 shows the hierarchy of the bootloader project with its dependency on the flash erase/program project.

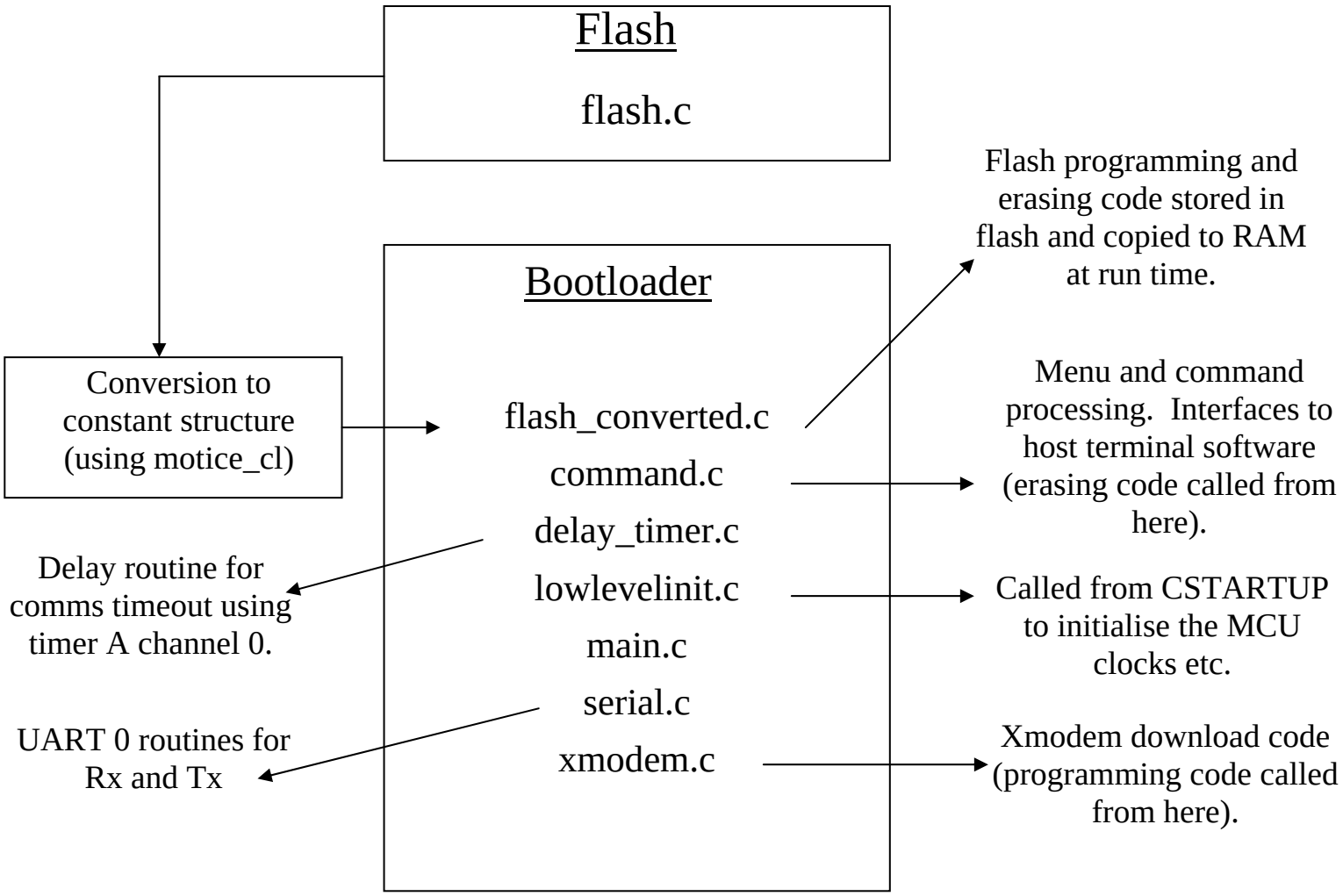

Figure 4: Bootloader Project Hierarchy

It is important that the RAM space in the bootloader project is kept free for use by the Flash programming and erasing functions. In this application this RAM starts at H‘3000. It is left to the user to ensure that this area of memory if kept free. Alternatively the addresses used for the RAM based code can be moved if H‘3000 is not suitable. However, this must be done for both the bootloader and flash projects.

There are 2 build configurations for the ‘bootloader’ project, ‘debug’ and ‘release’. Both configurations can easily be examined from the IAR Workbench IDE. The ‘debug’ configuration builds code suitable for debugging on the 3-D Kit under the KD30 ROM monitor. The ‘release’ configuration builds the project so that it executes as a standalone application running on the 3D Kit.

When debugging the ‘bootloader’ application using the KD30 ROM monitor it is important that that the ROM monitor is started in ‘FreeRunMode’. This reduces the amount of polling activity occuring between KD30 on the host PC and the target. Experience has shown that xmodem transfer problems occur if KD30 is not configured for ‘FreeRunMode’ operation. Figure 5 below shows a screenshot from the KD30 startup dialog showing the setting for ‘FreeRunMode’ operation.

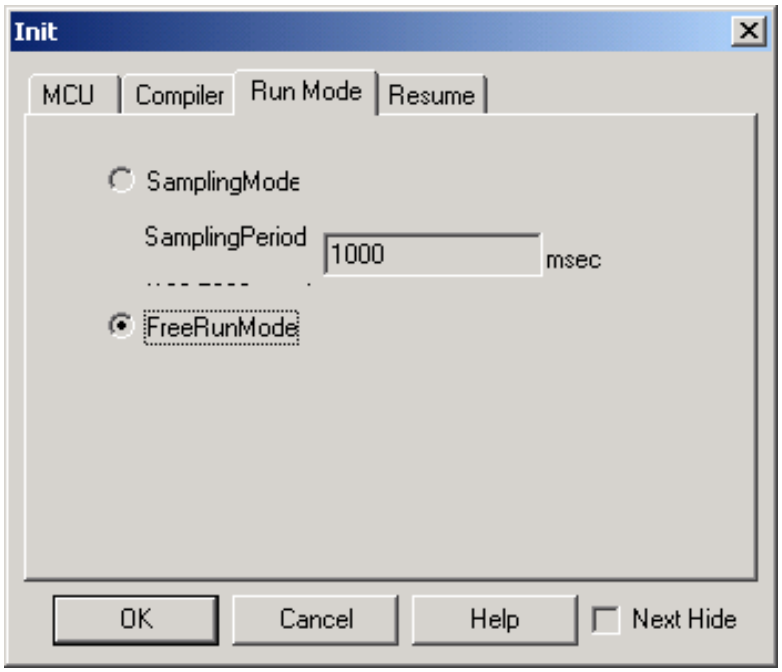

Figure 5: FreeRunMode Configuration of KD30

For the ‘bootloader‘,‘flash’ and ‘ledflash’ projects the linker command files ‘lnkcm16c.xcl’ and ‘lnkm16c_mot.xcl’ found in the project directories should be used rather than the standard linker command files found in the IAR toolchain directory. The screenshot in figure 6 shows how the linker options should be changed in order to override the default linker command file.

The file ‘lnkcm16c.xcl’ should be used with the ‘debug’ build of the projects and the file ‘lnkm16c_mot.xcl’ with the ‘release’ build. Please note that each project has its own version of these linker command (xcl) files.

The ‘bootloader’ project’s ‘debug’ and ‘release’ builds both have optimisation disabled.

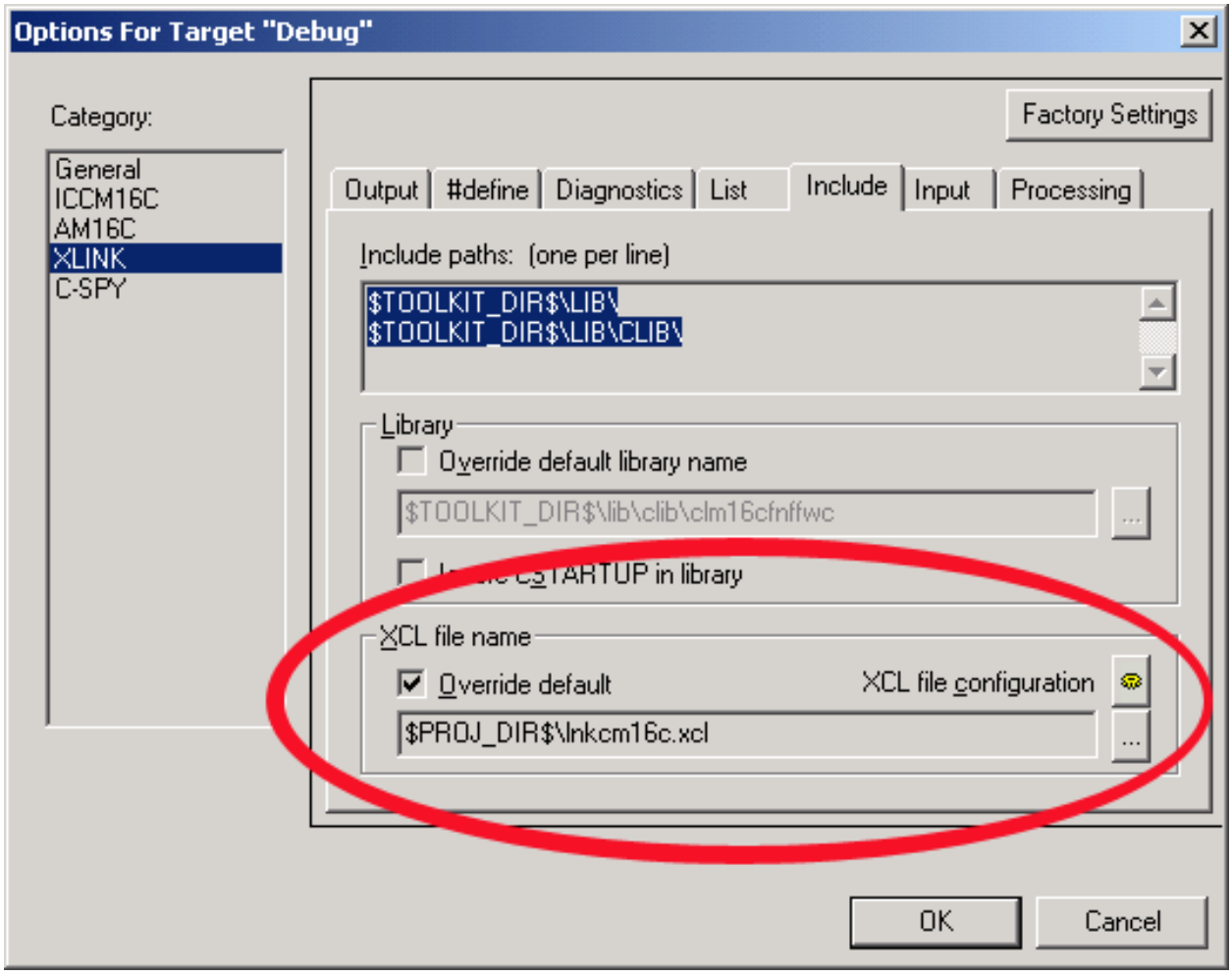

Figure 6: Changing the Default Linker Command File

Only the ‘release’ build is valid for the ‘flash’ project as when using the ‘motice_cl’ utility it is not possible to generate valid debug output from the compiler and linker.

As previously described, in point 3, a function pointer table is located at the start of the RAM based code for the erasing and programming routines found in the ‘flash’ project. These function pointers enable the bootloader to call the RAM based functions. The code segment below from ‘flash.c’ shows the instantiation of this function pointer table.

```
typedef unsigned char (*pt2Function)( unsigned long, unsigned short
* );
#pragma location="CPROGERA"
__root const pt2Function fp[] = {
     BlockErase,
     Program_128_Bytes
};
```

The '__root' IAR compiler extended keyword is used to force the compiler to generate code for this function pointer table even though it looks like this data is never used. Of course this table is used by the bootloader project.

The IAR linker command file, 'lnkcm16c_mot.xcl', for the 'flash' project has an entry placing the user defined segment 'CPROGERA' starting at address H'3000. This results in the entry for 'BlockErase' being placed at address H'3000 and the entry for 'Program_128_Bytes' at H'3004. As the function pointers are located in Flash they must be placed in a 'FARCONST' type segment. The snippet below from the linker command file 'lnkcm16c_mot.xcl' shows the entry for the 'CPROGERA' segment.

```
-Z(FARCONST)CPROGERA=0x3000
```

Both the 'BlockErase' and 'Program_128_Bytes' functions in the 'flash' project feature pointers to addresses contained in the Flash memory space. These pointers are automatic variables and so are located on the stack contained in the near addressable RAM memory area. However, as they point to the Flash memory they actually point to far data. Therefore, it is important that the IAR Workbench configuration for the 'flash' project reflects this in the project options dialog's general setting. Figure 7 shows this configuration setting.

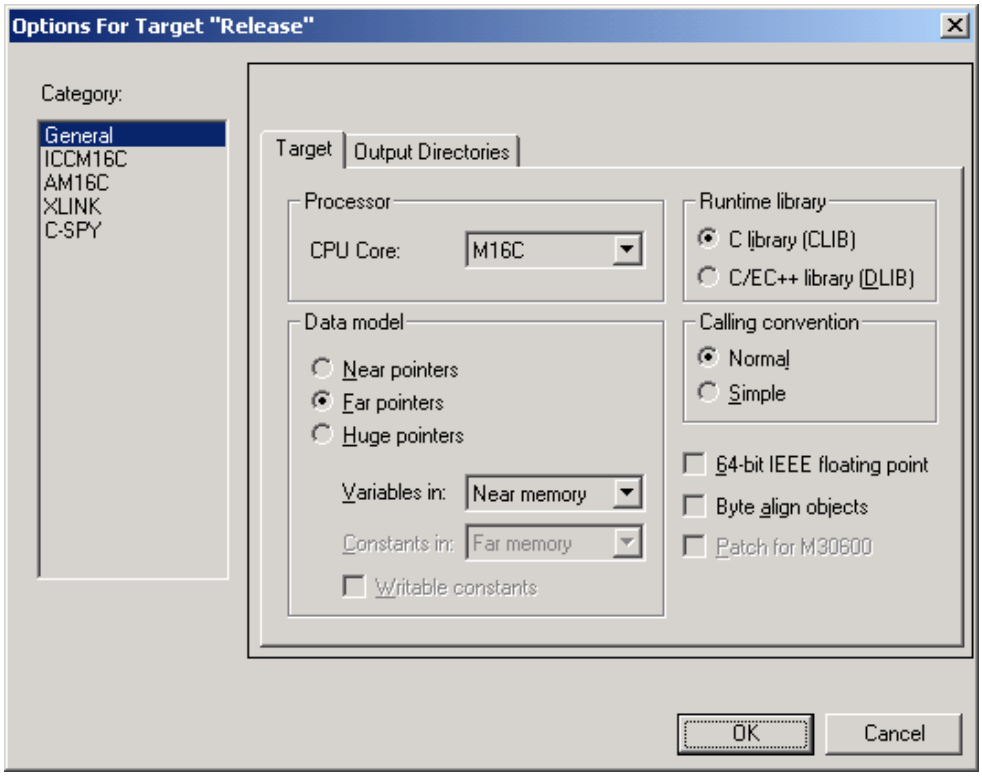

Figure 7: Far Pointers Should be Used by the Flash Project

In order to help eliminate the chance that the wrong pointer data model is used the declaration of the pointer uses the IAR compiler extended keyword '__data20'.

Calling the RAM Based Programming and Erasing Routines

Running the RAM based code is a case of copying the constant data from the Flash based structure to the correct RAM address and calling the relevant erasing of programming function via a function pointer. The following code example shows this in action and is taken from the 'bootloader' project file 'command.c '.

```
void InitCommandHandler (void)
{
…
     // copy all RAM based code to RAM
     memcpy( (void *) flashcode.start_address, &flashcode.data[0],
flashcode.data_length );
}
```

```
void Command_2 (void)
{
…
// load pointer to erase block function
ptr = (unsigned long *) ( INT_RAM_BASE_ADDRESS +
( ERASEBLOCK_OFFSET * 4 ) );
fp = (pt2FunctionErase) *ptr;

// call the erase function
Status = fp ( uc, 0 );
…
}
```

The 'Program_128_Bytes' function is passed two parameters. The first is the address to start programming at and the second is a pointer to the data to be programmed. The 'EraseBlock' function again takes two parameters. The first is the block number to be erased and the second is just a dummy value but is required so the same function pointer type can be used to call both the programming and erasing functions.

Protection of the Bootloader

As the bootloader is responsible for erasing and programming the Flash memory it is important that the bootloader code itself is not corrupted. In this bootloader example this is achieved by software. When a request is made to erase a block or program the Flash the block number or address is checked to ensure that it is not in the bootloader's code space.

An alternative to the software approach of protecting the bootloader is to make use of the M16C's Flash lock bit feature. This facility makes it possible to protect Flash blocks from being erased or programmed until the protection is removed. This feature has not been implemented in the bootloader example. Further details can be obtained from the hardware manual. In some situations it may be advantageous to be able to reload the bootloader. There are a number of different methods of achieving this including loading the new bootloader into another area of Flash memory first then copying it over the original after it is fully downloaded. This eliminates the possibility of the Flash memory being corrupted due to a comms failure. The ability to reprogram the bootloader is not covered any further by this apps note nor by the example application.

The Target Application

The target application that is downloaded into Flash by the bootloader must be modified slightly from a standard application to take into account that it must live with the bootloader. The issues requiring consideration include those below.

1. Base address. In a normal M16C application the Flash memory has the reset address at H'FFFFC. When using the described bootloader the start address of the user Flash area is H'A0000 and so the reset address is H'A0020. This is reflected in the IAR linker command file for the segment 'INTVEC1 '.
2. Interrupts. There is no need to handle relocatable interrupts any differently within the target application as the M16C INTB register makes it possible to locate the vector table anywhere in the memory map. The intialisation of the INTB register is performed by the 'CSTARTUP' code contained in the runtime library. The M16C also features fixed vectors which cannot be relocated using the INTB register. The fixed vector table contains the vectors for interrupts including NMI, undefined instruction, watchdog timer oscillation stop detection etc. As this vector table cannot be relocated these interrupts are handled by ISRs contained in the bootloader project. If necessary, a custom 'CSTARTUP.S34' could be used to call ISRs located in the target application.
3. Output format. The format of the target application's output file must be in such a form that the bootloader can accept and extract the contents and load it into the correct Flash memory addresses. In the case of this bootloader the output file should be in binary form so it can be downloaded via xmodem by a terminal program into the M16C. In this application the binary file was produced by converting the s-record file from the IAR linker into binary. There are a number of utilities available which can perform this conversion. The GNU GCC utility 'objcopy' can be used with the below command line.

```
h8300-elf-objcopy.exe -v -I srec LedFlash.mot -O binary
LedFlash.bin
```

The 'h8300-elf-objcopy.exe' version of objcopy can be obtained as part of the GNU GCC toolchain for Renesas H8 microcontrollers available for download from KPIT's website.

The target application contained in the 'LedFlash' project is a simple one designed to run on the M16C/62P 3-D Kit. The program flashes the LEDs on the board in a chaser sequence. The delay between the LEDs turning on and off is produced using timer A to generate a compare match interrupt. The target application's code is contained in the 'LedFlash' project.

Putting It All Together

The debug build of the bootloader application is designed to be used with the KD30 ROM monitor on the M16C/62P 3-D Kit. The release build can be loaded directly into the Flash of the micro using 'FlashStart' for standalone operation.

When the bootloader is started something similar to the screenshot in figure 8 should be seen if the keyboard is pressed within 3 seconds of it starting.

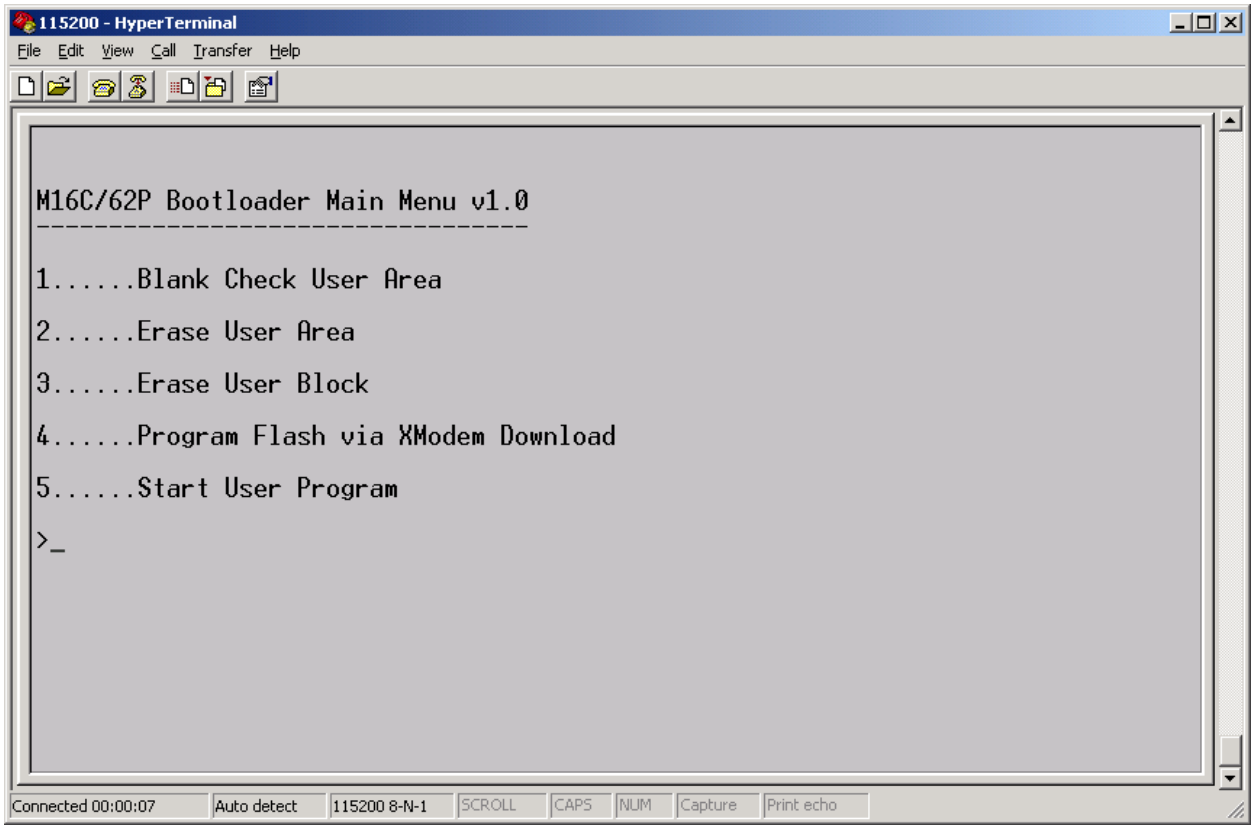

Figure 8: Main Bootloader Menu

Option 1 performs a blank check on the user Flash area as shown in figure 9.

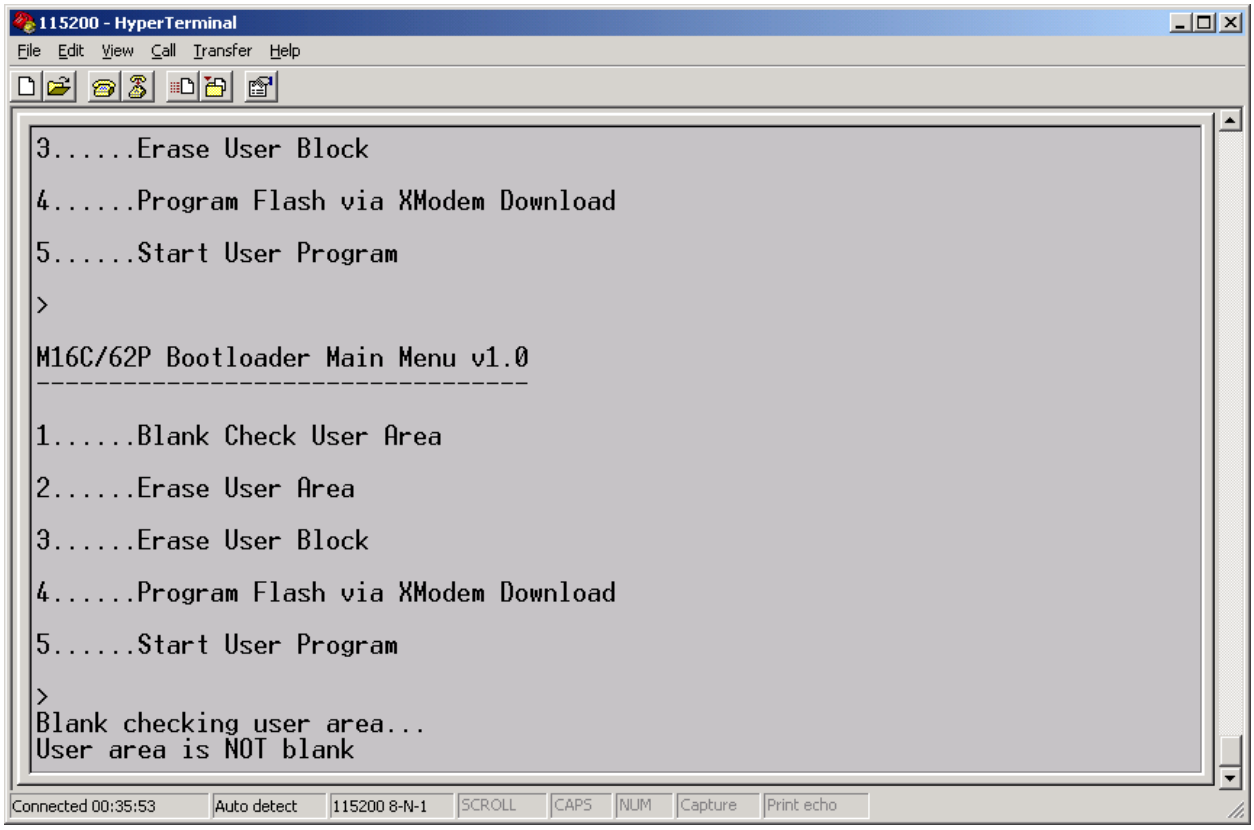

Figure 9: Blank Checking the User Area

All of the user (target) area of Flash can be erased via option 2 as shown in figure 10. Remember that the contents of the Flash must be erased before attempting programming.

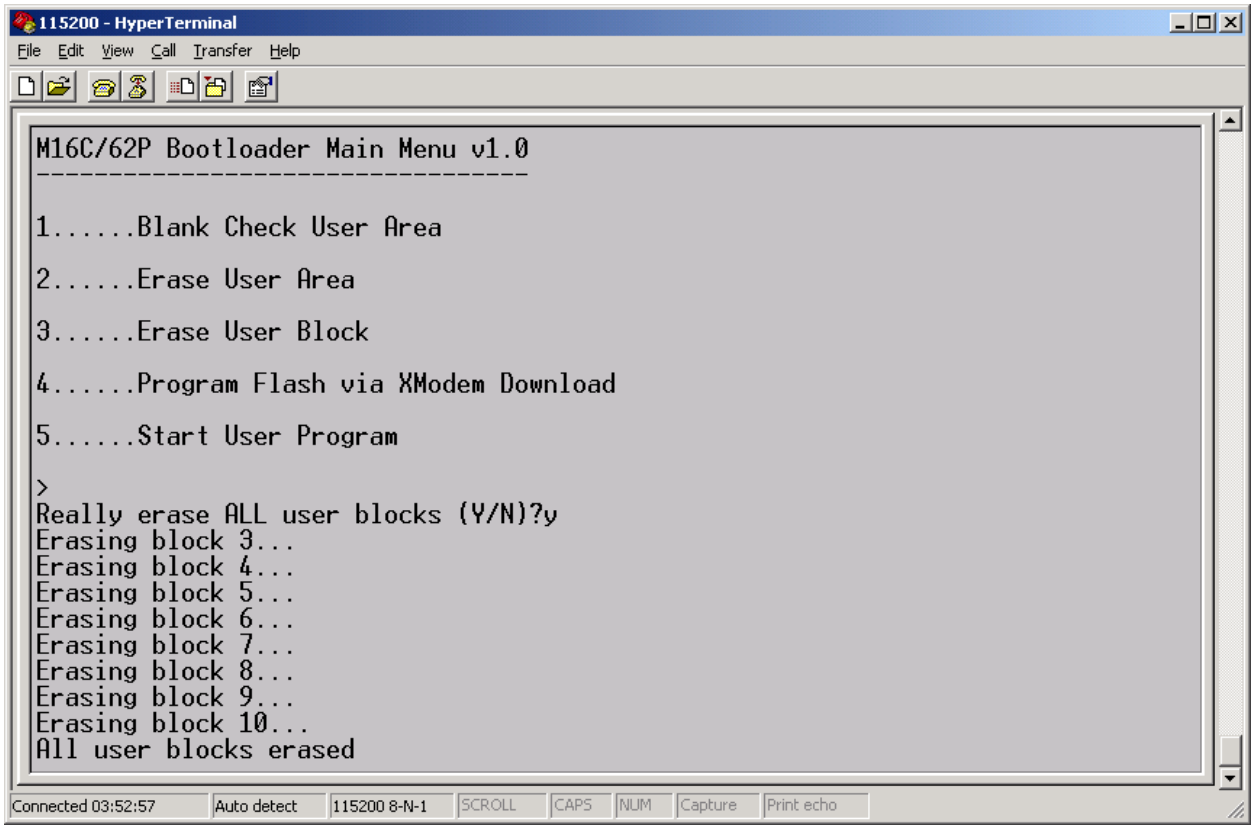

Figure 10: Erasing All User Flash Blocks

With all the user blocks erased the target application can be downloaded into the device. The screenshot in figure 11 shows that when option 4 is selected a 32-bit address must be entered which is the address at which Flash programming will begin. In the example this address is H‘000A0000. Here a transfer is set-up in the terminal program using xmodem which starts in response to the bootloader sending a start signal to the host. A 1kB binary file containing the target program is downloaded.

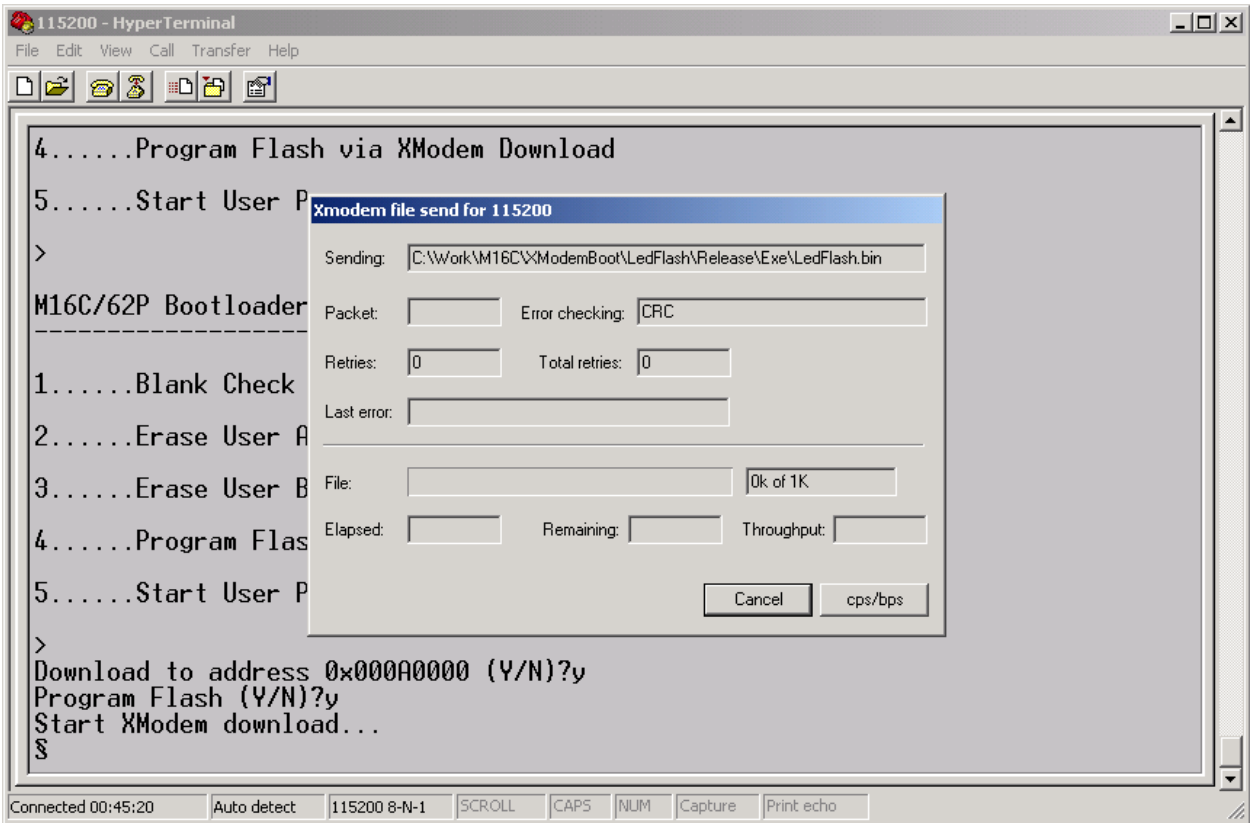

Figure 11: Downloading the Target Application Using XModem

With the target program successfully in the Flash memory it can be executed either by selecting option 5 or by resetting the bootloader and waiting 3 seconds without any activity on the serial channel.

Summary

Although Renesas Flash microcontrollers provide a built-in bootloader there are situations where a custom bootloader is advantageous. It is hoped that this application note has gone some way in discussing the areas that need consideration when developing such a bootloader. The sample application should hopefully provide a basis for further development.

Website and Support

Renesas Technology Website
http://www.renesas.com/

Inquiries
http://www.renesas.com/inquiry
csc@renesas.com

Notes regarding these materials

1. This document is provided for reference purposes only so that Renesas customers may select the appropriate Renesas products for their use. Renesas neither makes warranties or representations with respect to the accuracy or completeness of the information contained in this document nor grants any license to any intellectual property rights or any other rights of Renesas or any third party with respect to the information in this document.
2. Renesas shall have no liability for damages or infringement of any intellectual property or other rights arising out of the use of any information in this document, including, but not limited to, product data, diagrams, charts, programs, algorithms, and application circuit examples.
3. You should not use the products or the technology described in this document for the purpose of military applications such as the development of weapons of mass destruction or for the purpose of any other military use. When exporting the products or technology described herein, you should follow the applicable export control laws and regulations, and procedures required by such laws and regulations.
4. All information included in this document such as product data, diagrams, charts, programs, algorithms, and application circuit examples, is current as of the date this document is issued. Such information, however, is subject to change without any prior notice. Before purchasing or using any Renesas products listed in this document, please confirm the latest product information with a Renesas sales office. Also, please pay regular and careful attention to additional and different information to be disclosed by Renesas such as that disclosed through our website. (http://www.renesas.com)
5. Renesas has used reasonable care in compiling the information included in this document, but Renesas assumes no liability whatsoever for any damages incurred as a result of errors or omissions in the information included in this document.
6. When using or otherwise relying on the information in this document, you should evaluate the information in light of the total system before deciding about the applicability of such information to the intended application. Renesas makes no representations, warranties or guaranties regarding the suitability of its products for any particular application and specifically disclaims any liability arising out of the application and use of the information in this document or Renesas products.
7. With the exception of products specified by Renesas as suitable for automobile applications, Renesas products are not designed, manufactured or tested for applications or otherwise in systems the failure or malfunction of which may cause a direct threat to human life or create a risk of human injury or which require especially high quality and reliability such as safety systems, or equipment or systems for transportation and traffic, healthcare, combustion control, aerospace and aeronautics, nuclear power, or undersea communication transmission. If you are considering the use of our products for such purposes, please contact a Renesas sales office beforehand. Renesas shall have no liability for damages arising out of the uses set forth above.
8. Notwithstanding the preceding paragraph, you should not use Renesas products for the purposes listed below:
   (1) artificial life support devices or systems
   (2) surgical implantations
   (3) healthcare intervention (e.g., excision, administration of medication, etc.)
   (4) any other purposes that pose a direct threat to human life

   Renesas shall have no liability for damages arising out of the uses set forth in the above and purchasers who elect to use Renesas products in any of the foregoing applications shall indemnify and hold harmless Renesas Technology Corp., its affiliated companies and their officers, directors, and employees against any and all damages arising out of such applications.
9. You should use the products described herein within the range specified by Renesas, especially with respect to the maximum rating, operating supply voltage range, movement power voltage range, heat radiation characteristics, installation and other product characteristics. Renesas shall have no liability for malfunctions or damages arising out of the use of Renesas products beyond such specified ranges.
10. Although Renesas endeavors to improve the quality and reliability of its products, IC products have specific characteristics such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Please be sure to implement safety measures to guard against the possibility of physical injury, and injury or damage caused by fire in the event of the failure of a Renesas product, such as safety design for hardware and software including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other applicable measures. Among others, since the evaluation of microcomputer software alone is very difficult, please evaluate the safety of the final products or system manufactured by you.
11. In case Renesas products listed in this document are detached from the products to which the Renesas products are attached or affixed, the risk of accident such as swallowing by infants and small children is very high. You should implement safety measures so that Renesas products may not be easily detached from your products. Renesas shall have no liability for damages arising out of such detachment.
12. This document may not be reproduced or duplicated, in any form, in whole or in part, without prior written approval from Renesas.
13. Please contact a Renesas sales office if you have any questions regarding the information contained in this document, Renesas semiconductor products, or if you have any other inquiries.