How to build a simple serial bootloader for PIC32

Diego Mendes (diego.mendes@ua.pt), Cristóvão Cruz (cac@ua.pt)
Department of Electronics, Telecommunications and Informatics / Telecommunications Institute
University of Aveiro
Portugal

Version 1.1

1 Introduction

This guide is aimed for anyone looking to build a simple bootloader for a generic development board based on the PIC32MX Family. Still, the guide itself is oriented for the DETPIC32 from the University of Aveiro.

This guide is essentially a simplification of the application note AN1388 from Ganapathi Ramachandra available on the Microchip website[1].

2 Prerequisites

For this guide it is required that the user has a version of the MPLAB X IDE and the XC32 1.20 compiler installed.

If this is not the case, the user can get the required installers as well as the required installation instructions in:
`http://ww1.microchip.com/downloads/mplab/X/index.html`

3 Preparing the files and MPLAB project

1. Download the source code from:

   `ww1.microchip.com/downloads/en/AppNotes/AN1388_Source_Code_011112.zip`

2. Decompress the zip file and execute the Universal Bootloader. This step requires Microsoft Windows or a windows emulator (e.g. wine for Linux environments) as the Universal Bootloader is distributed in the form of an executable ("Universal Bootloader_011112.exe");

3. The installation folder, defined by the user during step 2, contains two different folders. One, called "Firmware", contains the actual bootloader code, executed by the PIC32 during the boot phase. The other one ("PC

[1]http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1824&appnote=en554836

Application") contains a Windows-based Graphical User Interface (GUI) that shall be executed in the host PC whenever the user wants to upload applications to the PIC32.
The supplied bootloader is adapted to the Microchip's Explorer 16 board, which has several communication interfaces (e.g. Ethernet, USB, SD card). From these, the DETPIC32 board is only equipped with an UART interface. For this reason, it necessary to execute the MPLABX IDE and open the project "[INSTPATH]/Firmware/ Bootloader/MPLAB_X_Workspace/UART_Btl_Ex
where [INSTPATH] is the installation directory, defined at step 2;

4. In the project properties, change the compiler to the one you have installed, XC 1.20. The "Project properties" menu can be accessed e.g. by right-clicking on the name of the project;

5. On Linux and MAC, change reference to "Bootloader.h" to "BootLoader.h" (notice the capital L), in line 41 of "BootLoader.c"

6. On Linux and MAC, change "FrameWork" to "Framework" (notice the capital W) in line 42 of "BootLoader.c"

7. On Linux and MAC, all the "\" in the include directives must be changed to "/". The affected lines are 39 to 42 of BootLoader.c, 39 to 42 of NVMem.c, 40 to 44 of Framework.c, 42 to 44 of Uart.c and 56 of BootLoader.h.

8. The provided linker script must be changed to work on XC32 1.20, so, in line 57 of uart_btl_ex16.ld change

```
exception_mem              : ORIGIN = 0x9FC01E00, LENGTH = 0x200
```

   to

```
exception_mem              : ORIGIN = 0x9FC01E00, LENGTH = 0xA00
```

9. After line 8 of the same file, add the following:

```
OPTIONAL("libmchp_peripheral.a")
OPTIONAL("libmchp_peripheral_32MX795F512H.a")
```

4 Adapting the code to the DETPIC32 constraints

1. In the project properties menu, change the Device to PIC32MX795F512H

2. We want the application to be placed into the "User Application Code and IVT" area[2], which is placed between 0x9D000000 and 0x9D07FFFF. So open the file "BootLoader.h" and at line 43 edit the values for the following MACROS:

```
#define APP_FLASH_BASE_ADDRESS 0x9D000000
#define APP_FLASH_END_ADDRESS 0x9D07FFFF
#define USER_APP_RESET_ADDRESS (0x9D000000 + 0x1000 + 0x970)
```

[2]AN1388 FIGURE 1, pag. 2

3. As mentioned above, the bootloader is adapted to the Explorer 16 board. This board is equipped with a button that allows entering on the "Firmware Upgrade Mode". However, this switch is not present in the DETPIC32 board. Since the DETPIC32, by itself, does not contain any switch or button, besides the reset one, it was decided to use only the time to control the process. When the bootloader starts (after power up or reset) it tries to connect to the PC application during 1 second. After that time, if no connection is established, the control is passed to the application previously loaded (if any).
   Additionally, the UART interface, by default, is configured to UART2. However, in the DETPIC32 board, only UART1 has an external interface, so, this aspect must also be changed.

4. Let's start by the UART. For the DETPIC32 board we must configure the bootloader to access the UART1, so open the header "Uart.h" and at line 29 change

   ```
   #define Ux(y) U2##y
   ```

   to

   ```
   #define Ux(y) U1##y
   ```

5. Now for the main code I suggest the following.

   ```
   INT main(void) {

     UINT pbClk;

     /* Setup configuration */
     pbClk = SYSTEMConfig(SYS_FREQ, SYS_CFG_WAIT_STATES | SYS_CFG_PCACHE);

     /* Initialize the transport layer - UART/USB/Ethernet */
     TRANS_LAYER_Init(pbClk);

     /* Stays in the firmware upgrade mode while no timeout
     occurs or if no application is present or if  the GUI
     is connected to the MCU */
     while ((ReadCoreTimer() < 0x1312D00) || WORKING() || !ValidAppPresent()) {
       /* Enter firmware upgrade mode. */
       /* Be in loop, looking for commands from PC */
       TRANS_LAYER_Task(); /* Run Transport layer tasks */
       FRAMEWORK_FrameWorkTask(); /* Run frame work related
       tasks (Handling Rx frame, process frame and so on) */
     }

     /* Close trasnport layer. */
     TRANS_LAYER_Close();

     /* No trigger + valid application = run application. */
     JumpToApp();

     return 0;
   }
   ```

   The "ReadCoreTimer() < 0x1312D00" condition forces the bootloader to wait for 1 second before moving on to the application, if there is a valid one. If you wish to have a different waiting time you only need to change the value in this condition. Remember that the Core Timer counter is incremented every two periods of the CPU clock[3].

   If only the above condition was used, after 1 second the firmware loading process would be interrupted (the "while()" cycle would terminate), even if a transfer was ongoing. To prevent this, a WORKING() function is used to detect ongoing transfers. This function tests a flag that is set

[3]http://blog.flyingpic24.com/2009/04/02/using-the-32-bit-core-timer/

when a connection is established and reset when the connection terminates. This way the bootloader leaves the loop only when it receives a "JMP_TO_APP" command from the GUI or the PIC32 is reset.

To make this possible we must open the "Framework.c" file and add the following global variable and function:

```
static BOOL Working = FALSE;

BOOL WORKING(void){
    return Working;
}
```

The next step consists in editing the "HandleCommand" function on the same file. Start by editing the case "READ_BOOT_INFO", in line 188, (this request is sent every time the GUI tries to establish a connection to the bootloader so we use this to our favor).

```
case READ_BOOT_INFO: //Read boot loader version info.
  memcpy(&TxBuff.Data[1], BootInfo,2);
  //Set the transmit frame length.
  TxBuff.Len = 2 + 1;//Boot Info Fields + command
  Working=TRUE;
  break;
```

And finally in the "JMP_TO_APP" case, line 228,

```
case JMP_TO_APP:
  // Exit firmware upgrade mode.
  RunApplication = TRUE;
  Working=FALSE;
  break;
```

6. The bootloader is now ready.

5 Preparing the application's linker

In order to create programs that can be booted by this bootloader, you must use a custom linker script. This section explains the process of creating that linker script.

1. Right click the project » New » Empty File and create a *.ld file with the name you prefer(ex: linker.ld)

2. Copy the contents of "<XC32>\pic32mx\lib\ldscripts\elf32pic32mx.x" to the newly created file. <XC32> is the compiler installation folder.

3. Copy the following lines of code and paste them after the PROVIDE directives and before "SECTIONS".

```
/**************************************************************************
* Processor-specific object file. Contains SFR definitions.
**************************************************************************/
INPUT("processor.o")
/**************************************************************************
* For interrupt vector handling
**************************************************************************/
```

```
PROVIDE(_vector_spacing = 0x00000001);
/* _ebase_address value must be same as the ORIGIN value of exception_mem
(see below) */
_ebase_address = 0x9D000000;
/*************************************************************************
* Memory Address Equates
*************************************************************************/
/* Equate _RESET_ADDR to the ORIGIN value of kseg1_boot_mem (see below) */
_RESET_ADDR = (0x9D000000 + 0x1000 + 0x970);
/*Map _BEV_EXCPT_ADDR and _DBG_EXCPT_ADDR in to kseg1_boot_mem (see below) */
/* Place _BEV_EXCPT_ADDR at an offset of 0x380 to _RESET_ADDR */
/* Place _DBG_EXCPT_ADDR at an offset of 0x480 to _RESET_ADDR */
_BEV_EXCPT_ADDR = (0x9D000000 + 0x1000 + 0x970 + 0x380);
_DBG_EXCPT_ADDR = (0x9D000000 + 0x1000 + 0x970 + 0x480);
_DBG_CODE_ADDR = 0xBFC02000;
_DBG_CODE_SIZE = 0xFF0 ;
_GEN_EXCPT_ADDR = _ebase_address + 0x180;
/*************************************************************************
* Memory Regions
*
* Memory regions without attributes cannot be used for orphaned sections.
* Only sections specifically assigned to these regions can be allocated
* into these regions.
*************************************************************************/
MEMORY{
/* IVT is mapped into the exception_mem. ORIGIN value of exception_mem must
align with 4K address boundary. Keep the default value for the length */
exception_mem : ORIGIN = 0x9D000000, LENGTH = 0x1000
/* Place kseg0_boot_mem adjacent to exception_mem. Keep the default value
for the length */
kseg0_boot_mem : ORIGIN = (0x9D000000 + 0x1000), LENGTH = 0x970
/* C Start-up code is mapped into kseg1_boot_mem. Place kseg1_boot_mem
adjacent to
kseg0_boot_mem. Keep the default value for the length */
kseg1_boot_mem : ORIGIN = (0x9D000000 + 0x1000 + 0x970), LENGTH = 0x490
/*All C files (Text and Data) are mapped into kseg0_program_mem. Place
kseg0_program_mem adjacent to kseg1_boot_mem. Change the length of
kseg0_program_mem as required. In this example, 512 KB Flash size
is shrunk as follows: */
kseg0_program_mem (rx):ORIGIN = (0x9D000000 + 0x1000 + 0x970 + 0x490),
LENGTH = (0x80000 - (0x1000 + 0x970 + 0x490))
debug_exec_mem : ORIGIN = 0xBFC02000, LENGTH = 0xFF0
config3 : ORIGIN = 0xBFC02FF0, LENGTH = 0x4
config2 : ORIGIN = 0xBFC02FF4, LENGTH = 0x4
config1 : ORIGIN = 0xBFC02FF8, LENGTH = 0x4
config0 : ORIGIN = 0xBFC02FFC, LENGTH = 0x4
kseg1_data_mem (w!x) : ORIGIN = 0xA0000000, LENGTH = 0x20000
sfrs : ORIGIN = 0xBF800000, LENGTH = 0x100000
configsfrs : ORIGIN = 0xBFC02FF0, LENGTH = 0x10
}
```

4. Remove the "INCLUDE procdefs.ld" directive.
5. Copy the content of the SECTIONS directive from `<XC32>\lib\proc\32MX795F512H\procdefs.ld` and paste at the beginning of the existing SECTIONS directory of the linker script.
6. *** VERY IMPORTANT *** *For every application you want to load onto the DETPIC32, you must add this linker to its project.*

6 Closing remarks

By now you should have a functioning bootloader and the required linker for your projects. If you wish to use these files in different development boards the changes required are minimal so it should easy, still reading the AN1388 is advised.

7 Changelog

2014-10-01
Pedro Fonseca and Paulo Pedreiras. Small text and code fixes.