AN58170

Extending FX1/ FX2LP Code Memory Using Code Banking

Author: Anand Srinivasan

Associated Project: Yes

Associated Part Family: EZ-USB® (FX/FX1/FX2/FX2LP)

Software Version: Keil uVision2

More code examples? We heard you.

For a consolidated list of USB Hi-Speed Code Examples, please visit our code examples web page.

The EZ-USB® family of chips has an 8051 core, which has 16 address lines. The chip is able to access 64 KB of external memory. However, the firmware size sometimes may exceed 64 KB. This application note describes methods of overcoming this 64 KB limitation and also demonstrates the implementation of one such method.

1 Introduction

The amount of memory that a microcontroller can address is restricted by its address and data bus widths. For example, the 8051 in EZ-USB® parts can address only 64 KB of memory because its address bus is 16 bits wide and data bus is 8 bits wide. This restriction in turn restricts the size of the firmware. This restriction of firmware size can be overcome by code/memory banking.

2 Overview

Code banked firmware consists of banks and common area (The number of common areas depends on the application. In most cases, there is only one.) The operations involving bank switching in code banked firmware is handled by common area. Depending on the algorithm used, there can be common area in each bank or one bank that is the common area. Even if a common area is present in each bank, the common area code is the same regardless of the bank.

2.1 Possible Methods

There are many possible methods to achieve code banking. This application note describes two possible methods, differentiated by the resource used to achieve bank switching. These are:

- Code banking using xdataport
- Code banking using GPIOs

xdataport is the address and data port used by the microcontroller to access external memory.

2.1.1 Code Banking Using xdataport

This method uses external logic to implement hardware registers. In this method, a memory location is defined as a register for storing the bank selected. When this register is written to, the corresponding address and data is exposed on the address and data bus of the microcontroller. This data is stored and used by the external logic to manipulate the address read from the memory element.

2.1.2 Code Banking Using GPIOs

This method uses GPIOs as bank selectors. Depending on the number of banks, the number of GPIOs used varies. This solution may not be feasible in conditions where GPIOs are a critical resource.

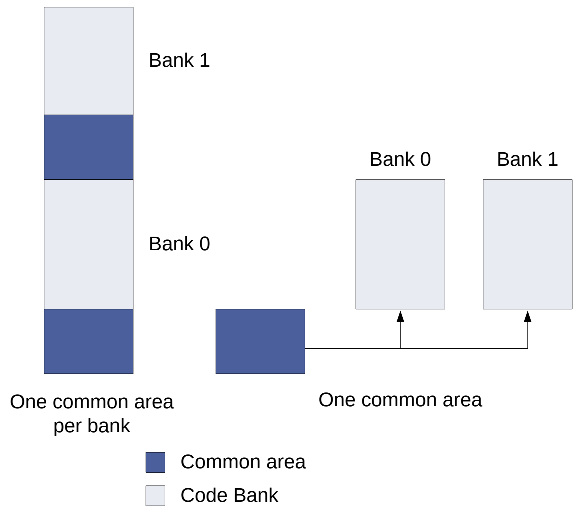
2.2 Code Banking Memory Architecture

There are two possible memory allocation architectures of code banked firmware, based on the common area configuration used. They are:

- Code banking using one common area
- Code banking using common in each bank

Figure 1 represents two architectures. The logic behind having a common area is that it is accessible no matter which bank is selected. This makes it the ideal block to handle operations that involve bank switching.

Figure 1. Code Banking Based on Common Area



2.2.1 Code Banking Using One Common Area

This method is very effective in terms of memory use because it avoids the memory waste that can occur when there is more than one common area. This algorithm involves manipulation of the MSBs of the address bus to achieve bank switching. Bank0 is commonly used as common area. In this case, the external logic must make sure that the common area is accessible regardless of the bank selected. This method involves complex hardware. In this method, the common area contains bank switching logic, code, and common variables.

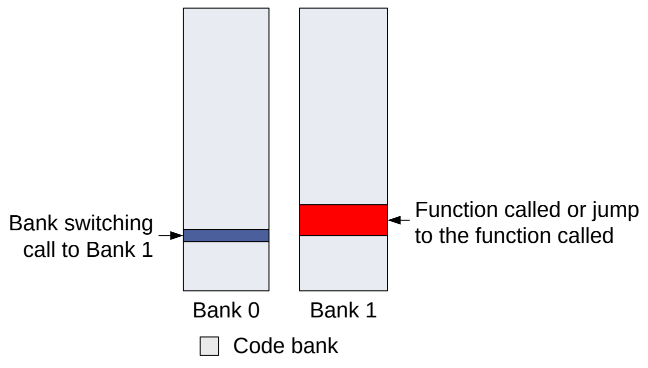
2.2.2 Code Banking Using Common Area per Bank

This method uses data in the hardware register or GPIO directly as address lines to the memory element. This allows the presence of common area for each bank. Although this method can cause memory waste, the hardware is less complex. In this method, the common area contains only the code and bank switching logic.

2.3 Importance of Common Area

The importance of common area lies in interbank function calls. An interbank function call can be seen as two steps: bank switching and start of execution from the function called. Consider that there is no common area present in this scenario; you will lose access to the current bank once you switch banks. Because of this, the memory location next to that of the bank switching code must contain the function called or a jump to that function, as shown in Figure 2. This is a very stringent requirement considering the fact that even a small modification can adversely affect this delicate memory alignment.

Figure 2. Bank Switching Without Common Area


Bank switching call to Bank 1
Function called or jump to the function called
Bank 0
Bank 1
Code bank

Absence of common area makes the sharing of common variables very difficult. This is also a problem for a one common area per bank configuration. This is a situation where the Scratch data RAM of the EZ-USB® FX1™, FX2™, and FX2LP™ parts can be used to our advantage, as it is accessible regardless of the code bank.

3 Demonstration

This is a demonstration of code banking with two code banks, using a GPIO as the bank selector. This description is based on the EZ-USB® FX2LP™ DVK. Though it is demonstrated in the FX2LP part, this concept can be extended to other chips in the EZ-USB family.

3.1 Hardware

The two code banks are implemented in the external SRAM (U3) of the DVK. In this instance, GPIO pin PE6 (Port E pin 6) is the bank selector. In the DVK, the external memory is a 128K SRAM to facilitate code banking. The A16 (Address pin 16) pin of the external RAM is connected to 3.3V through a 10K resistor R29. Connecting PE6 to A16 of the external RAM converts R29 into the pull up resistor for the line. PE6 can source up to 4 mA; thus, R29 provides enough resistance in the line for it to be pulled low safely.

3.2 Firmware

This firmware development demonstration leverages the Keil µVision® IDE's support for code banking. This section provides an overview of the firmware and a step by step method of how this code banking firmware was developed.

3.2.1 Overview

The firmware consists of two banks residing in memory locations 0x4000 to 0x4FFF. Each bank has three functions. Each of these functions are called using an IN vendor request. Based on the vendor request, a single byte is returned to the host. One function of each bank demonstrates an interbank function call. Table 1 describes what each vendor command does.

Table 1. Vendor Commands

Vendor Request Code (Hex)	wValue (Hex)	Function Called	Value Returned
A6	X	c_bank_0_function_1()	01
A7	X	c_bank_0_function_2()	02
A8	Any value other than 01	c_bank_0_function_3()	11
A8	01	c_bank_0_function_3()	12
A9	X	c_bank_1_function_1()	11
AA	X	c_bank_1_function_2()	12
AB	Any value other than 01	c_bank_1_function_3()	01
AB	01	c_bank_1_function_3()	02

Note X denotes that wValue of the vendor request can take any value.

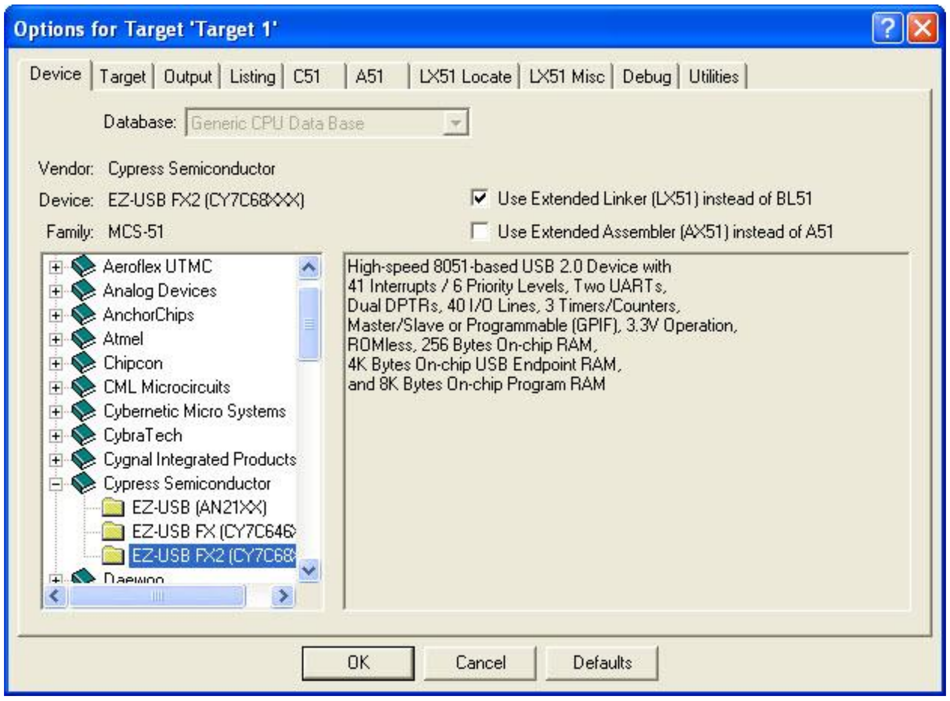
3.2.2 Settings

You must configure some settings for the firmware to support code banking. This section describes which settings to change.

By default, when the project is created the Project hierarchy contains the **Target 1** folder and the **Source Group 1** folder.

1. In the project window on the left side of the µVision IDE, select **Target 1**.
2. On the **Project** menu, select **Options for Target 'Target 1'**. The **Options for Target 'Target 1'** window opens.
3. Click the **Device** tab. Select the **Use Extended Linker (LX51) instead of BL51** check box, as shown in Table 1.

Figure 3. Device Tab Setting in Options for Target 'Target 1' Window



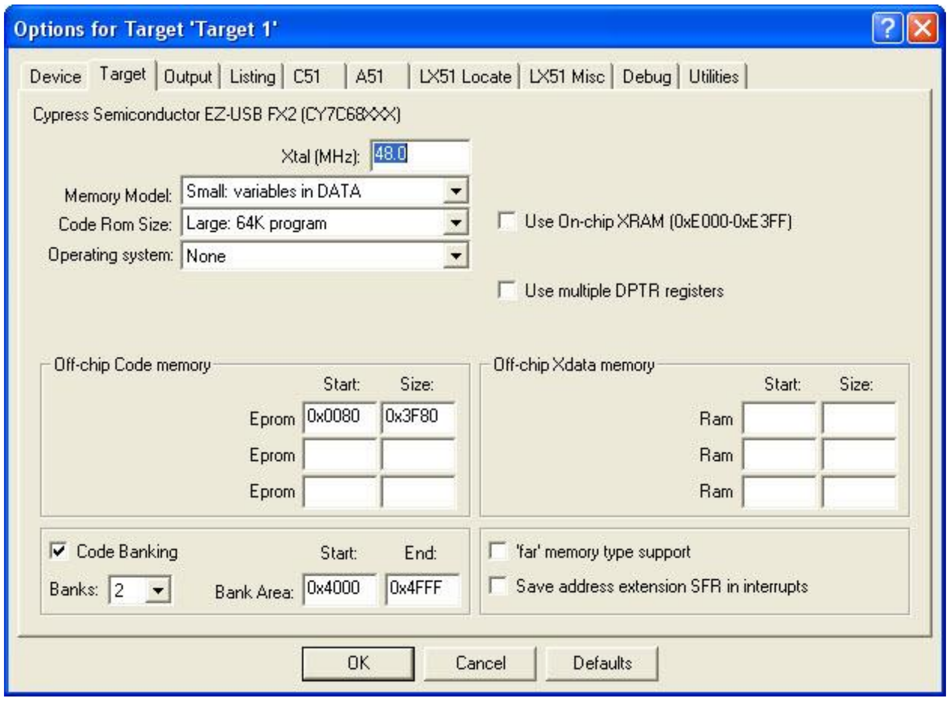
Vendor: Cypress Semiconductor
Device: EZ-USB FX2 (CY7C68XXX)
Family: MCS-51

[✓] Use Extended Linker (LX51) instead of BL51
[ ] Use Extended Assembler (AX51) instead of A51

High-speed 8051-based USB 2.0 Device with
41 Interrupts / 6 Priority Levels, Two UARTs,
Dual DPTRs, 40 I/O Lines, 3 Timers/Counters,
Master/Slave or Programmable (GPIF), 3.3V Operation,
ROMless, 256 Bytes On-chip RAM,
4K Bytes On-chip USB Endpoint RAM,
and 8K Bytes On-chip Program RAM

4. Click the **Target** tab and select the **Code Banking** check box.
   a. Type the number of banks in the **Banks:** field.
   b. Fill the code banks **Start:** and **End:** addresses in the **Bank Area:** field, as shown in Figure 4.
   c. You use the information in this tab to specify the memory locations occupied by code and data memory. You do this using the **Off-chip Code memory** and **Off-chip Xdata memory** fields, if the **Use memory layout from Target dialog** check box is selected in the **LX51 Locate** tab. For this demonstration, **Off-chip Code memory** starts at 0x80 and the length is 0x3F80.

Figure 4. Target Tab Setting in Options for Target 'Target 1' Window



Options for Target 'Target 1'

Tabs: Device | Target | Output | Listing | C51 | A51 | LX51 Locate | LX51 Misc | Debug | Utilities

Cypress Semiconductor EZ-USB FX2 (CY7C68XXX)

Xtal (MHz): 48.0

Memory Model: Small: variables in DATA

Code Rom Size: Large: 64K program

Operating system: None

☐ Use On-chip XRAM (0xE000-0xE3FF)

☐ Use multiple DPTR registers

Off-chip Code memory

	Start:	Size:
Eprom	0x0080	0x3F80
Eprom		
Eprom		

Off-chip Xdata memory

	Start:	Size:
Ram		
Ram		
Ram		

☑ Code Banking — Banks: 2 — Bank Area: Start: 0x4000 End: 0x4FFF

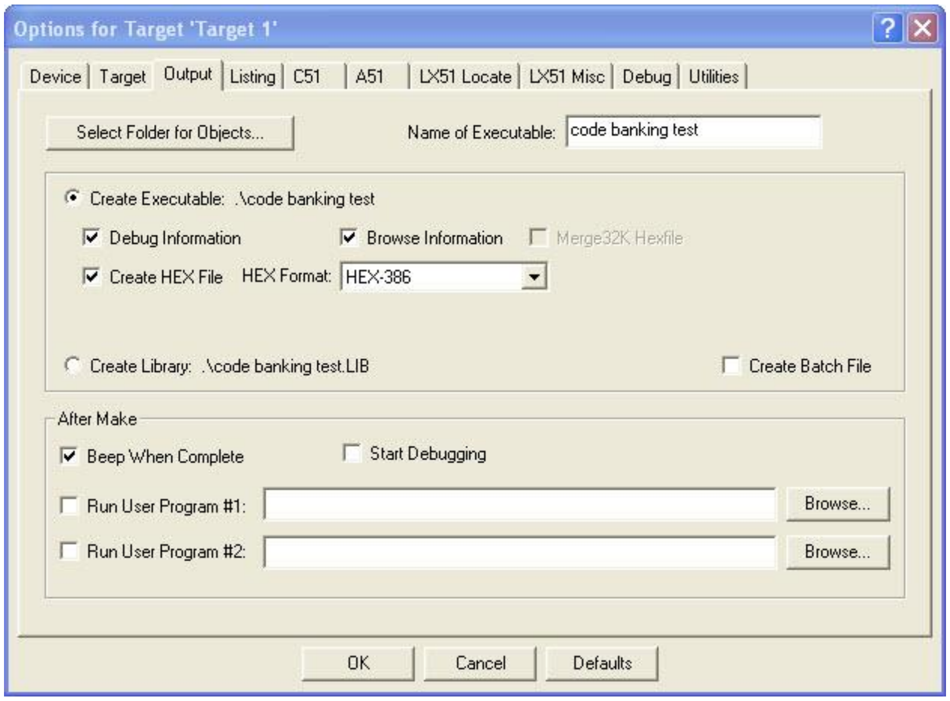
☐ 'far' memory type support

☐ Save address extension SFR in interrupts

OK | Cancel | Defaults

5. Click the **Output** tab and select **HEX-386** from the **HEX Format:** menu, as shown in Figure 5. This allows the hex file output to contain extended linear address records. (These records specify the code bank of that part of the hex file.)

Figure 5. Output Tab Setting in Options for Target 'Target 1' Window



Options for Target 'Target 1'

Tabs: Device | Target | Output | Listing | C51 | A51 | LX51 Locate | LX51 Misc | Debug | Utilities

Select Folder for Objects...

Name of Executable: code banking test

● Create Executable: .\code banking test

☑ Debug Information ☑ Browse Information ☐ Merge32K Hexfile

☑ Create HEX File HEX Format: HEX-386

○ Create Library: .\code banking test.LIB ☐ Create Batch File

After Make

☑ Beep When Complete ☐ Start Debugging

☐ Run User Program #1: Browse...

☐ Run User Program #2: Browse...

OK | Cancel | Defaults

3.2.3 Firmware Files

Installing the Keil µVision IDE provides the *L51_BANK.A51* file to aid in the development of code banked firmware. After installation, you can find this file in C:\Keil\LIB\ (the path varies based upon the location chosen during installation). This file manages the bank switching logic. The framework files provided by Cypress (*EZUSB.LIB*, *USBJmpTb.OBJ*, and so on) manage the basic USB device operations. The *L51_BANK.A51* file and the framework files must reside in common area. The framework files are here: C:\Cypress\USB\Target (properly subdivided in folders) after installing the EZ-USB FX2LP DVK. The *L51_BANK.A51* file contains comments to provide information about what a particular line does and how to modify the file based on the code banking logic used. The following list describes the lines in the *L51_BANK.A51* file that are relevant to this demonstration.

- `?B_NBANKS        EQU  4    ; Define maximum Number of Banks`

This line provides the number of banks to be implemented. In this example the number of banks implemented is 2, so this line is modified to

```
?B_NBANKS        EQU  2    ; Define maximum Number of Banks
```

- `?B_MODE          EQU  0    ; 0 for Bank-Switching via 8051 Port`

This line provides the resource used for bank switching. Because this example uses GPIO, ?B_MODE must be 0, so this line is left unchanged.

- `?B_VAR_BANKING   EQU  0    ; Variable Banking via L51_BANK (far memory support)`

This line denotes whether the *L51_BANK.A51* module is used. In this example it is used, so this line is modified to

```
?B_VAR_BANKING   EQU  1    ; Variable Banking via L51_BANK (far memory support)
```

- `?B_RST_BANK      EQU  0xFF ; specifies the active code bank number after CPU`

This line provides the code bank selected after the CPU comes out of reset. In this example it is code bank 0, so this line is modified to

```
?B_RST_BANK      EQU  0x00 ; specifies the active code bank number after CPU
```

- Based on the ?B_MODE value selected, modify the appropriate lines of code under **IF ?B_MODE =** statement. In this case ?B_MODE = 0, so make the following modification under **IF ?B_MODE = 0**.

```
P1               DATA    90H      ; I/O Port Address
?B_PORT          EQU     P1       ; default is P1
?B_FIRSTBIT      EQU     2        ; default is Bit 2
```

This code denotes the 8051 port from which the bank selector GPIO pins are used. In this example, it is port E pin 6, so these lines are modified to

```
PE               DATA    0B1H     ; I/O Port Address
?B_PORT          EQU     PE       ; default is P1
?B_FIRSTBIT      EQU     6        ; default is Bit 2
```

-
  ```
  ?B?XSTART   EQU   0x8000                    ;   Start   of   xdata   bank   area
  ?B?XEND   EQU 0xFFFF      ; Stop of xdata bank area
  ```

These lines provide the start and end address of the banks. In this example 0x4000 to 0x4FFF is used, so these lines are modified to

```
?B?XSTART EQU 0x4000      ; Start of xdata bank area
?B?XEND   EQU 0x4FFF      ; Stop of xdata bank area
```

In this case, these values are not based on any limitations and so are used for this demonstration. The stop of xdata bank area value can be as high as 0xFFFF. So theoretically, this demonstration can be modified to handle a firmware size of 112 KB.

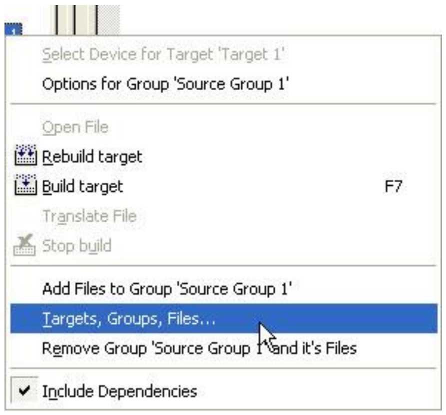
3.3 Assigning Code Banks

In the µVision you assign code banks to files that form the firmware. Coding such that the file contains only functions that belong to a particular code bank makes it easier to assign code banks. In this case, commonness of variables defined at specific memory locations to all code banks and common areas is not handled by the compiler; that is, if that variable is accessed it just leads to access of that memory location without involving the bank switching that might be required.

Code banks in which a function resides should be decided such that there are fewer interbank function calls. This increases the execution speed. This section describes the steps to assign code bank

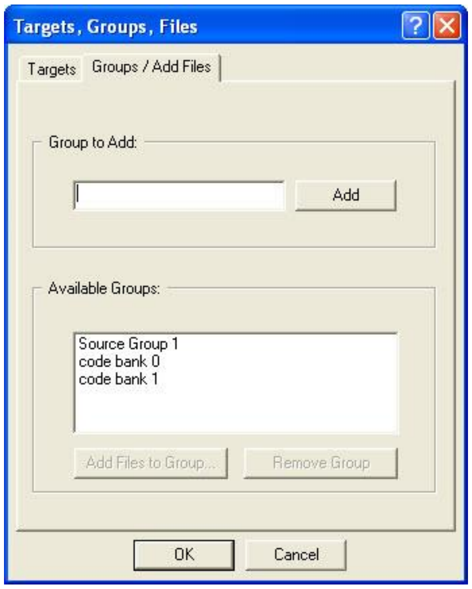
1. Right-click **Target 1** or **Source Group 1** in the project window and select **Targets, Groups, Files…** from the menu that appears, as shown in Figure 6.

Figure 6. Opening Targets, Group, Files Window


Select Device for Target 'Target 1'
Options for Group 'Source Group 1'
Open File
Rebuild target
Build target
F7
Translate File
Stop build
Add Files to Group 'Source Group 1'
Targets, Groups, Files…
Remove Group 'Source Group 1' and it's Files
Include Dependencies

2. In the **Groups/Add Files** tab of this window, type the name of the group under Target1 that you want to create. In the **Group to Add** field and click **Add**, as shown in Figure 7. Notice that this group is immediately added to the **Available Groups** list. When you click **OK**, this hierarchy is reflected in the Target 1 subtree in the project window. It is a good practice to name the groups with the code bank number. Figure 7 shows that the groups **code bank 0** and **code bank 1** are created.

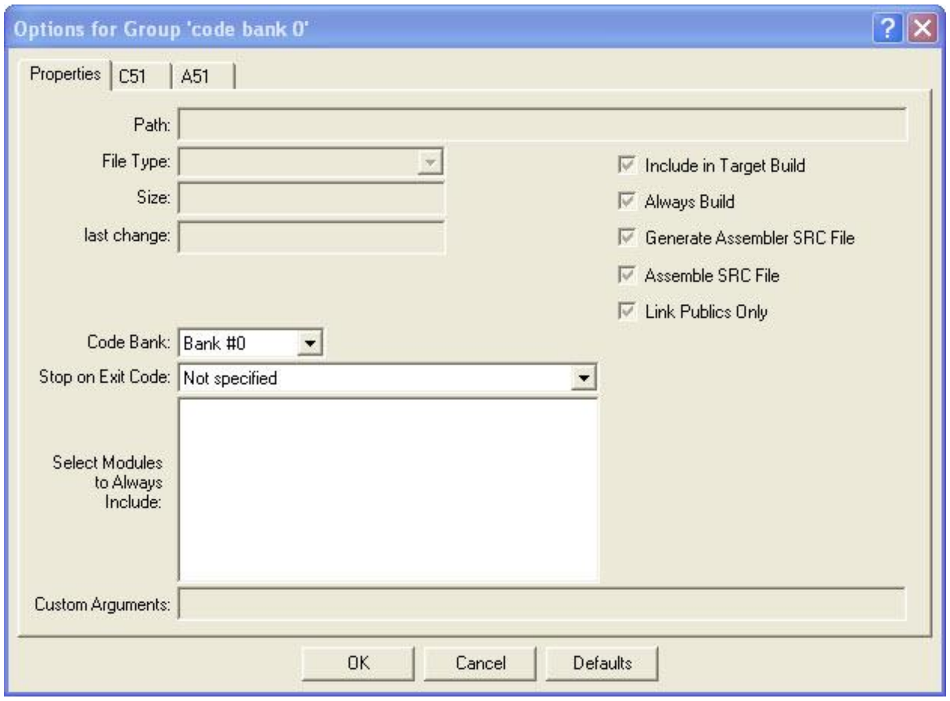
Figure 7. Target, Groups, Files Window



Source Group 1
code bank 0
code bank 1

3. Right-click the group name and select **Add files to Group '*Group name*'**. Add the files belonging to that code bank/common area. ***Group name*** reflects the name of the group that was clicked on. Each code bank contains one .c file with three functions, and Source Group 1 contains the framework files and the *L51_BANK.A51*. These are added to the appropriate group.
4. Right-click the group name and select **Options for Group '*Group name*'**. The **Options for Group '*Group name*'** window appears. ***Group name*** reflects the name of the group that was clicked on. In this window, select the code bank to which this group belongs, using the **Code Bank:** field. By default, **Source Group 1** is in Common, and. **code bank 0** and **code bank 1** are placed in Bank #0 and Bank #1, respectively.

Figure 8. Options for Group 'Group Name' Window


Options for Group 'code bank 0'

Properties | C51 | A51

Path:
File Type:
Size:
last change:
Code Bank: Bank #0
Stop on Exit Code: Not specified
Select Modules to Always Include:
Custom Arguments:

☑ Include in Target Build
☑ Always Build
☑ Generate Assembler SRC File
☑ Assemble SRC File
☑ Link Publics Only

OK | Cancel | Defaults

5. Compile the firmware.

3.4 Host Application

The host applications that are available on cypress.com, such as CyConsole and Control Centre, do not support code banking. The host application provided with this application note is a *CyAPI.lib* (available as part of **SuiteUSB** 3.4 - USB Development tools for Visual Studio) based command line utility that takes the code banked firmware as input and downloads it onto EZ-USB. This section contains a description of how this application works

Note the device has to be bound to *CyUSB.sys* for the firmware download. This host application is capable of downloading non-code banked firmware as well.

3.4.1 Overview

The hex output file produced using µVision follows Intel hex format. In this firmware, since the format HEX-386 was selected for the output file format, the hex file will contain an extended linear address record to denote the code bank. This record has a record type value 04. Because the host applications provided by Cypress do not parse this record properly, they do not support code banking. In the host application provided with this application note, while parsing the hex files the record type value is used to switch bank and download the code corresponding to that bank. This host application is generic because the code bank parameters (start and end address, number of code banks and so on) can vary based on the application. This algorithm can be optimized using the correct code bank parameters.

3.4.2 Code Banker

This firmware is based on a vend_ax example that comes as part of the DVK. In the EZ-USB parts, the A0 vendor command is used to write to internal memory and is handled by the device without any firmware support. The A0 vendor command cannot write to external memory directly. To facilitate the download of code corresponding to external memory, you must implement an algorithm. This is where code banker firmware plays a key role. This is how the algorithm works:

1. Develop code banker firmware residing only in internal memory, based on the needs of the application.
2. Using the A0 vendor command, the host application downloads code banker to internal memory. Code banker supports the A3 vendor command, which can write to external memory.
3. The host application downloads the code that resides in external memory using the A3 vendor command.
4. The host application uses the A0 vendor command to download that part of the firmware which resides in internal memory.

So for any firmware that has part or all of its code residing in external memory, code banker is used in the firmware download process. Code banker uses AD vendor command to support the downloading of code banked firmware. This vendor command switches banks so that code can be downloaded to the other banks. In this case, this vendor command uses wValue of the vendor command to send the bank information. Based on the design, this vendor command must be modified to incorporate the proper bank switching algorithm.

3.4.3 Firmware Download

This is the algorithm for downloading a code banked firmware performed by this host application.

1. Parse the Intel hex file and get the firmware for bank 0.
2. The host application downloads firmware for bank 0, using A3 and A0 vendor commands.
3. The hex file provides the code bank number. Now the AD vendor command switches bank appropriately.
4. The firmware download for this particular bank is accomplished using the A3 and A0 vendor commands
5. Steps 3 and 4 are repeated until the code for all the banks is downloaded.

4 Summary

This document describes how to develop an application when the firmware size exceeds the maximum size that the microcontroller can handle. It also demonstrates one such algorithm and its implementation using the EZ-USB FX2LP DVK.

About the Author

Name: Anand Srinivasan

Title: Applications Engineer

Document History

Document Title: AN58170 – Extending FX1/ FX2LP Code Memory Using Code Banking

Document Number: 001-58170

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	2826553	AASI/DSG	12/11/2009	New application note.
*A	3150316	AASI	01/21/2011	Added more comments in the host application Added lines to explain that the host application is CyAPI.lib based and that it is capable of downloading non-code banked firmware.
*B	3863093	GAYA	01/09/2013	Updated in new template.
*C	4778544	GAYA	05/26/2015	Updated template
*D	5139570	GAYA	03/30/2016	Added reference to HS code example page in the Abstract.
*E	5754794	GAYA	09/20/2017	Changed the AN title from "Code/Memory Banking Using EZ-USB" to " Extending FX1/ FX2LP Code Memory Using Code Banking". Removed "AN21XX" from the "Associated Part Family: EZ-USB®" Updated template

Worldwide Sales and Design Support

Cypress maintains a worldwide network of offices, solution centers, manufacturer's representatives, and distributors. To find the office closest to you, visit us at Cypress Locations.

Products

ARM® Cortex® Microcontrollers	cypress.com/arm
Automotive	cypress.com/automotive
Clocks & Buffers	cypress.com/clocks
Interface	cypress.com/interface
Internet of Things	cypress.com/iot
Memory	cypress.com/memory
Microcontrollers	cypress.com/mcu
PSoC	cypress.com/psoc
Power Management ICs	cypress.com/pmic
Touch Sensing	cypress.com/touch
USB Controllers	cypress.com/usb
Wireless Connectivity	cypress.com/wireless

PSoC® Solutions

PSoC 1 | PSoC 3 | PSoC 4 | PSoC 5LP | PSoC 6

Cypress Developer Community

Forums | WICED IOT Forums | Projects | Videos | Blogs | Training | Components

Technical Support

cypress.com/support

All other trademarks or registered trademarks referenced herein are the property of their respective owners.

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709

© Cypress Semiconductor Corporation, 2009-2017. This document is the property of Cypress Semiconductor Corporation and its subsidiaries, including Spansion LLC ("Cypress"). This document, including any software or firmware included or referenced in this document ("Software"), is owned by Cypress under the intellectual property laws and treaties of the United States and other countries worldwide. Cypress reserves all rights under such laws and treaties and does not, except as specifically stated in this paragraph, grant any license under its patents, copyrights, trademarks, or other intellectual property rights. If the Software is not accompanied by a license agreement and you do not otherwise have a written agreement with Cypress governing the use of the Software, then Cypress hereby grants you a personal, non-exclusive, nontransferable license (without the right to sublicense) (1) under its copyright rights in the Software (a) for Software provided in source code form, to modify and reproduce the Software solely for use with Cypress hardware products, only internally within your organization, and (b) to distribute the Software in binary code form externally to end users (either directly or indirectly through resellers and distributors), solely for use on Cypress hardware product units, and (2) under those claims of Cypress's patents that are infringed by the Software (as provided by Cypress, unmodified) to make, use, distribute, and import the Software solely for use with Cypress hardware products. Any other use, reproduction, modification, translation, or compilation of the Software is prohibited.
TO THE EXTENT PERMITTED BY APPLICABLE LAW, CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS DOCUMENT OR ANY SOFTWARE OR ACCOMPANYING HARDWARE, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. To the extent permitted by applicable law, Cypress reserves the right to make changes to this document without further notice. Cypress does not assume any liability arising out of the application or use of any product or circuit described in this document. Any information provided in this document, including any sample design information or programming code, is provided only for reference purposes. It is the responsibility of the user of this document to properly design, program, and test the functionality and safety of any application made of this information and any resulting product. Cypress products are not designed, intended, or authorized for use as critical components in systems designed or intended for the operation of weapons, weapons systems, nuclear installations, life-support devices or systems, other medical devices or systems (including resuscitation equipment and surgical implants), pollution control or hazardous substances management, or other uses where the failure of the device or system could cause personal injury, death, or property damage ("Unintended Uses"). A critical component is any component of a device or system whose failure to perform can be reasonably expected to cause the failure of the device or system, or to affect its safety or effectiveness. Cypress is not liable, in whole or in part, and you shall and hereby do release Cypress from any claim, damage, or other liability arising from or related to all Unintended Uses of Cypress products. You shall indemnify and hold Cypress harmless from and against all claims, costs, damages, and other liabilities, including claims for personal injury or death, arising from or related to any Unintended Uses of Cypress products.
Cypress, the Cypress logo, Spansion, the Spansion logo, and combinations thereof, WICED, PSoC, CapSense, EZ-USB, F-RAM, and Traveo are trademarks or registered trademarks of Cypress in the United States and other countries. For a more complete list of Cypress trademarks, visit cypress.com. Other names and brands may be claimed as property of their respective owners.