AN61244

Firmware Optimization in EZ-USB®

Author: Gayathri Vasudevan
Associated Project: Yes
Associated Part Family: AN21xx/FX/FX1/FX2/FX2LP
Software Version: Keil uVision2
Related Application Notes: None

If you have a question, or need help with this application note, contact the author at gaya@cypress.com.

More code examples? We heard you.

For a consolidated list of USB Hi-Speed Code Examples, visit this page.

The EZ-USB® family of chips has an 8051 core and uses the standard 8051 instruction set. However, it has a few enhancements compared to the standard 8051. This application note describes firmware optimization methods in EZ-USB. Some of these methods are common for any processor and some specific to the 8051 core of EZ-USB.

Contents

Introduction

Every firmware designer wants optimal memory footprint and execution speed. The trade-off between these competing goals can be worked out based on the end application. This application note describes methods of achieving firmware optimization with these goals in mind. It also discusses the advantages and disadvantages of each method.

Overview

The type of firmware optimization required is based on the end application. In a cost conscious environment, the focus can be on the memory footprint. In a run-time sensitive environment, the focus is on the execution speed. Not all optimizations are suitable for every design. The methods of optimization are discussed based on a demonstration firmware that comes with this document. This firmware is developed to emphasize the advantage and disadvantage of each method. The firmware only demonstrates achievable optimizations and does not have an end purpose. Use this to weigh the advantages and disadvantages of each method based on the end application and design accordingly.

Note that some of the methods described are complex and require significant effort and advanced knowledge in embedded design.

8051 Enhancements in EZ-USB

The 8051 core of EZ-USB has few enhancements compared to the standard 8051. Particularly relevant to the application note are the following

- Wasted bus cycles are eliminated; an instruction cycle uses only four clocks, rather than the 12 clocks in the standard 8051
- The EZ-USB's CPU clock runs at 12 MHz, 24 MHz, or 48 MHz - up to four times the clock speed of the standard 8051
- Dual data pointers. Used to accelerate the data memory block moves
- A high speed external memory interfaces with a non-multiplexed 16 bit address bus
- Two autopointers (auto incrementing data pointers)
- EZ-USB specific SFRs (such as MPAGE SFR)
- A hardware pointer for SETUP data, and logic to process entire CONTROL transfers automatically

How to Look for Optimization

In firmware optimization, certain modifications reduce firmware size up to a byte. It is difficult to go through the hex file or assembly code to understand how the particular modification provides the intended optimization. How do you understand this at the assembly level?

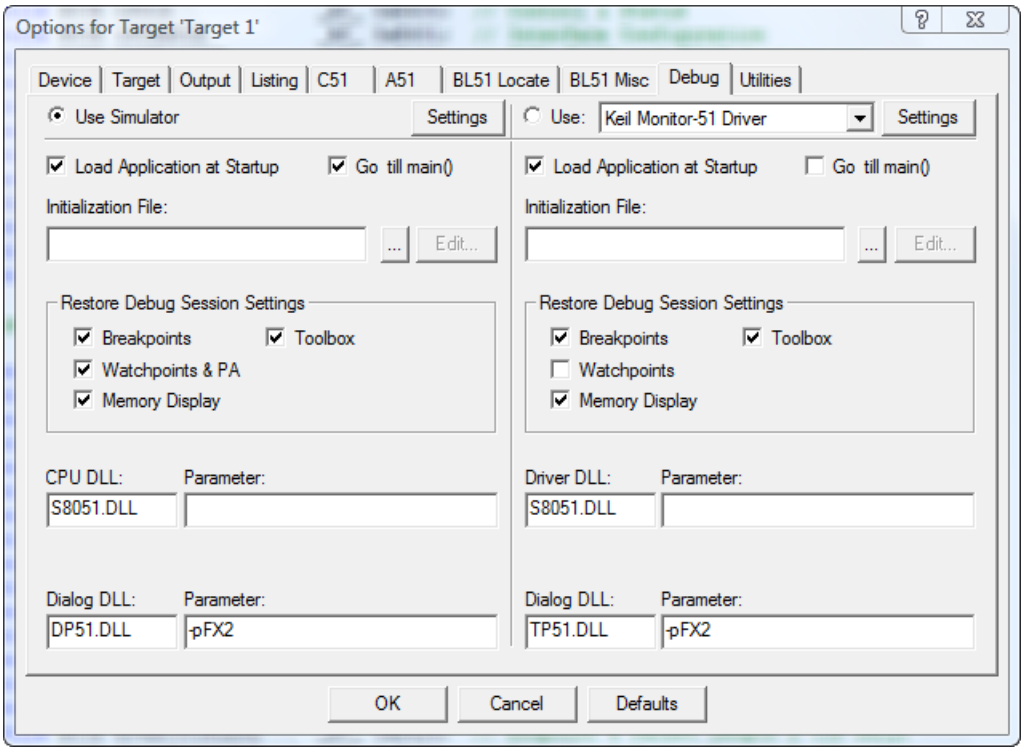
This can be viewed using the simulator feature available as part of the debug interface of Keil uVision. Following is a step by step description to achieve this:

1. In the project window of the IDE select **Target 1**. This is the name given to the target in the project and is used throughout this document to refer to the target selected. You may vary the name when creating your project.
2. In the Project menu, select **Options for Target 'Target 1'**. The select **Options for Target 'Target 1'** Window appears.
3. In the Debug tab of this window, select the **Use Simulator** radio button as shown in Figure 1. Press **OK**.
4. Press **Start/Stop Debug Session**; the debug session starts, allowing code to be stepped through. Because the environment is in simulation, it does not download code to the device.
5. Go to the code segment where the effect of the modification is to be studied. Right-click the line and click **Show Disassembly at 0xXXXXXXXX**. Here **XXXXXXXX** changes based on the line of code that is clicked.
6. In the Disassembly window that appears, the code is seen in dark red color (in uVision2, this varies based on settings or the version used). Below each line of code, the corresponding assembly language translation and hex value is displayed.
7. This helps to understand the optimization at the microcontroller code execution level without having to reverse engineer the hex file and do the tedious task of cross-referencing with the 8051 instruction set to understand the optimization.

The memory footprint optimization can be observed through the compilation statement in the output window of the IDE. This is displayed in the form of Program Size: data=xxxx xdata=yyyy code=zzzz. Here xxxx, yyyy, and zzzz are the corresponding sizes. "data" represents amount of internal memory of the 8051 core of EZ-USB used to store data. "xdata" represents the memory external to the 8051 used to store data. This memory use is high because the registers are defined as xdata in the framework. Therefore, even if you do not enter any value for them in the firmware, these locations are reserved for data. "code" represents the amount of memory occupied by code.

Note The on-chip RAM of 8051 is also external to the 8051 and comes under xdata memory.

Figure 1. Debug Tab Settings for Options for Target ‘Target 1’ Window



Demonstration Firmware

The demonstration firmware is developed based on FX2LP, using the framework files provided by Cypress. The framework files can be found in `C:\Cypress\USB\Target` (properly subdivided in folders) after installing FX2LP DVK. The firmware uses two endpoints (EP2 and EP6) apart from the control endpoint. The firmware constantly polls for packets in EP2 and loops it to EP6. This functionality is used to demonstrate firmware optimization that can be achieved by leveraging the autopointer feature of EZ-USB. The firmware also implements five (B1 to B5) vendor commands, which fill the EP6 buffer with hex values in ascending order. These vendor commands are used to demonstrate certain methods of optimization. One vendor command (B6) which copies the data of EP2 (if it has a packet) to EP0, is used to demonstrate the optimization achieved using the dual data pointer feature of EZ-USB.

Possible Methods

There are many possible methods of achieving firmware optimization. This application note describes six possible methods. Some of these methods are already well known and are described here for completeness. These are

- Reuse of variables
- Reuse of code
- Use of MPAGE SFR
- Use of data pointers
- Use of autopointers
- Use of setup data pointer

Demonstration

The firmware uses compiler directive #ifdef in c files and assembler directive $if in assembly files (*.a51) to enable or disable part of the firmware and make the exercise user friendly.

Define preprocessor symbols for #ifdef statements:

1. In the project window to the left of the IDE, select **Target 1**.
2. In the Project menu, select **Options for Target ‘Target 1’**”. The **Options for Target ‘Target 1’** window appears.
3. In the C51 tab, define the preprocessor symbols as shown in Figure 2, separated by comma (,). The preprocessor symbol used to demonstrate a particular method is described in the corresponding section.

Set value to conditional assembly control symbols for the $if statements:

1. In the project window to the left of the IDE, select **Target 1**.
2. In the Project menu, select **Options for Target 'Target 1'**" in it. The **Options for Target 'Target 1'** window appears.
3. In the A51 Tab, set the values to the conditional assembly control symbols as shown in **Error! eference source not found.** on page 4, separated by comma (,). The value used for the conditional assembly control symbol in the demonstration of a particular method is described in the corresponding section.

Figure 2. Defining Preprocessor Symbol

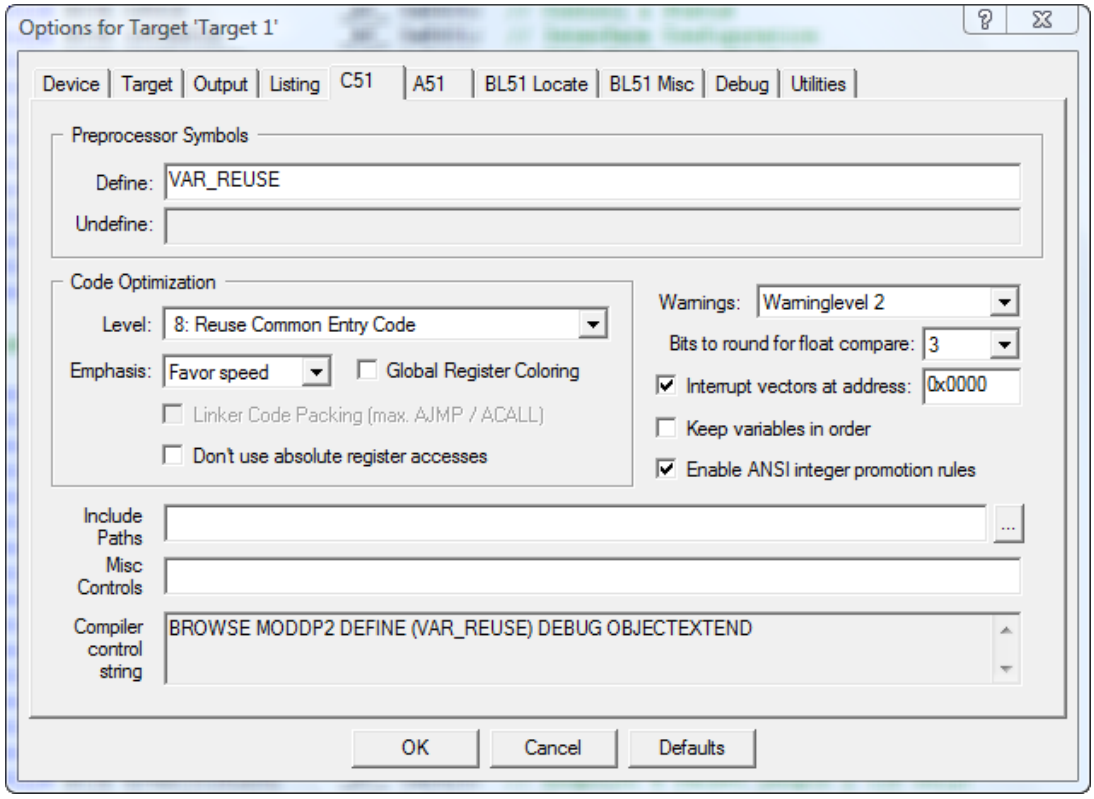
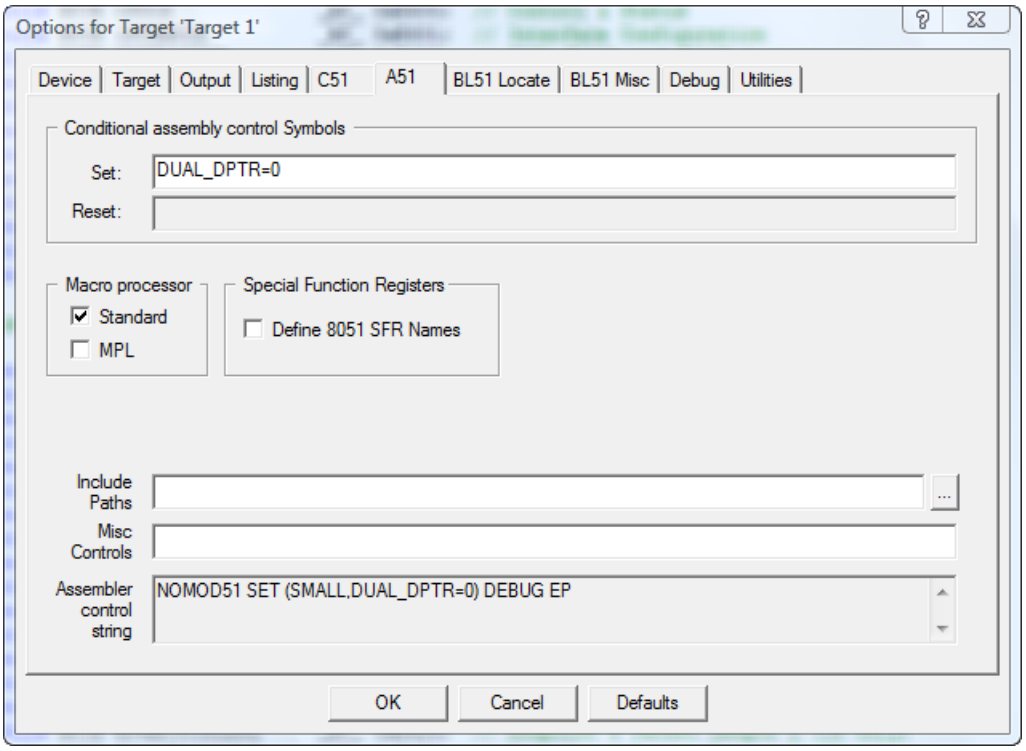


Figure 3. Setting Conditional Assembly Control Symbols



Reuse of Variables

In the demonstration, in DR_VendorCmnd() function of firmware_optimization.c, each of the five vendor commands (0xB1, 0xB2, 0xB3, 0xB4 and 0xB5) use a different variable as counter (i_var_reuse, j_var_reuse, k_var_reuse, l_var_reuse and m_var_reuse respectively) in the 'for' loop. Define the preprocessor symbol **VAR_REUSE** to use a single counter variable **i_var_reuse**. Compile the firmware and note the amount of memory used.

Now undefine the preprocessor symbol **VAR_REUSE** to use a separate variable for each of these vendor commands. Compile the firmware; the difference in the amount of data memory used is seen. This is because the variable is defined as WORD. If it is defined as xdata WORD, the variables will be placed in xdata space and the amount of xdata memory used is reduced.

Advantage

The external memory of EZ-USB stores code and data separately, but the memory space is shared. This optimization frees memory for use as data/code memory.

Disadvantage

This method increases the complexity of the code - care must be taken to properly initialize the reused variable. This is necessary because the variable should not carry modifications done to its value during previous use.

Reuse of Code

In the demonstration, in DR_VendorCmnd() function of firmware_optimization.c, though five of the vendor commands (0xB1, 0xB2, 0xB3, 0xB4 and 0xB5) fill the EP6 buffer with different values, the algorithm used to fill the buffer is similar. Define the preprocessor symbol **CODE_REUSE** to handle these vendor commands using the function **void Fill_EP6(BYTE factor)** in firmware_optimization.c. Compile the firmware and note the amount of memory used.

Now undefine the preprocessor symbol **CODE_REUSE** to handle each of these vendor commands separately rather than a common function. Compile the firmware; the difference in the amount of code memory used is seen.

Advantage

This method reduces the code memory used.

Disadvantage

This method introduces jumps between memory locations during code execution; this reduces the execution speed.

Use of MPAGE SFR

MPAGE SFR in EZ-USB resides at 0x92 and is used to specify the upper address byte of MOVX operation using @R0/@R1. Here, @ is used to denote the value in the register is the address of a memory location. Assigning MPAGE with the MSB of memory location accessed most often reduces the number of bytes of code used to access that location, thus reducing code size.

In this method, the firmware must have a statement to assign a value to MPAGE SFR. This statement takes three bytes of code. When a data space is addressed using normal method, the assembly code equivalent is to

```
MOV DPTR,#Address
MOV A,#Value
MOVX @DPTR, A (or) MOVX A,@DPTR
```

Here, `#Address` is the 2 byte address and `#Value` is the data to write/read at `#Address`. This method uses six (3+2+1) bytes of code to access this memory location.

When using the MPAGE SFR method, the assembly code equivalent is to

```
MOV R0,#AddressL
MOV A,#Value
MOVX @R0,A (or) MOVX A,@R0
```

Here `#AddressL` is the LSB and `#AddressH` the MSB of `#Address`. `#AddressH` is given through MPAGE SFR. This assembly equivalent also holds true if R1 is used instead of R0. This method uses five (2+2+1) bytes of code to access this memory location. For each memory access that uses this method, 1 byte of code is saved. Therefore, using this method for more than three memory accesses result in code size reduction.

In the demonstration, in TD_Init() function of firmware_optimization.c, seven accesses to registers in location 0xE6XX are made. The last four accesses are to the same location; only one of them use MPAGE method (the other three use one MOVX for every access because there is no change in `#AddressL`/`#Address` and `#value`). Therefore, there are four accesses using MPAGE method providing a code size reduction of 1 byte. To help easier differentiation, the registers using MPAGE method of access are defined with a P_ prefix. They are initialized using the LSB of their location and are of type **BYTE volatile pdata**. This method is shown in mpage_initilization.h of the demonstration firmware. Though the initialization and access method are different they address the same memory location. This means CPUCS and P_CPUCS are the same register, only their access method is different. Define the preprocessor symbol **MPAGE_SFR** to use this method of memory access. Compile the firmware and note the amount of memory used.

Now undefine the preprocessor symbol **MPAGE_SFR** to use the normal method of access. Compile the firmware; the difference in the amount of code memory used is seen.

Advantage

This method provides reduction in code memory used.

Disadvantage

This method requires initialization of registers accessed. Increase in code complexity - ensure that MPAGE is initialized to the proper value before the access.

Note In the newer parts of the EZ-USB series (FX1/FX2/FX2LP), the registers reside in the 0xE600 to 0xE6FF range and GPIF waveform in 0xE400 to 0xE47F range. Designs which access the registers often or modify parts of GPIF waveform can use this method.

Use of Data Pointers

EZ-USB has dual data pointers; the data pointer being used can be selected using DPS SFR that resides at 0x86. Consider a situation during memory transfer using the data pointer, if the firmware has to access another location for data (for example, an ISR which transfers memory is triggered) and then return to continue from where it left off. Then the current address pointed to by the data pointer is pushed to the stack, the new value is loaded, and data is accessed. When it returns, the previous value pops from the stack and execution continues. Even while moving blocks of data between memory locations, the data pointer must be loaded twice for each transaction. This is expensive in terms of code and time. In EZ-USB, the dual data pointers allow the firmware to switch to the other data pointer; when it returns, it just has to switch back to the previous data pointer. The bits other than the SEL bit (bit 0) in DPS are unused. The easiest way to switch is using INC DPS. This not only leads to reduction in code size but also faster execution. Follow these steps to inform the compiler that both data pointer are being used:

1. In the project window to the left of the IDE, select **Target 1**.
2. In the Project menu, select **Options for Target 'Target 1'**" in it. The **Options for Target 'Target 1'** window appears.
3. In the Target tab of this window, check the **Use multiple DPTR registers** as shown in Figure 4.

This is done because the compiler must push and pop the registers corresponding to both data pointers in cases where there is a possibility of the data pointer values being corrupt. The compiler may use this for optimization through compiler based function (such as memcpy and memmove). These are compiler based optimization and are beyond the scope of this document hence not discussed here.

In the demonstration, in DR_VendorCmnd() function of firmware_optimization.c, vendor command 0xB6 sends a 1 byte packet if there is no packet in EP2; otherwise, the first 64 byte of EP2 is copied to EP0 and sent to the host. The 64 byte copying is done through dual_dptr_func() in dual_data_pointer.a51 which demonstrates how this firmware optimization can be used. Set the conditional assembly control symbol DUAL_DPTR to 1 (**DUAL_DPTR=1**) to use two data pointers to handle this function. Compile the firmware and note the amount of memory used.

Now set the conditional assembly control symbol DUAL_DPTR to 0 (**DUAL_DPTR=0**) to use single data pointer to handle this function. Compile the firmware; the difference in the amount of code memory used is seen.

The key thing to be noted is the number of instructions inside the loop. The execution speed increases based on the number of loops.

Advantage

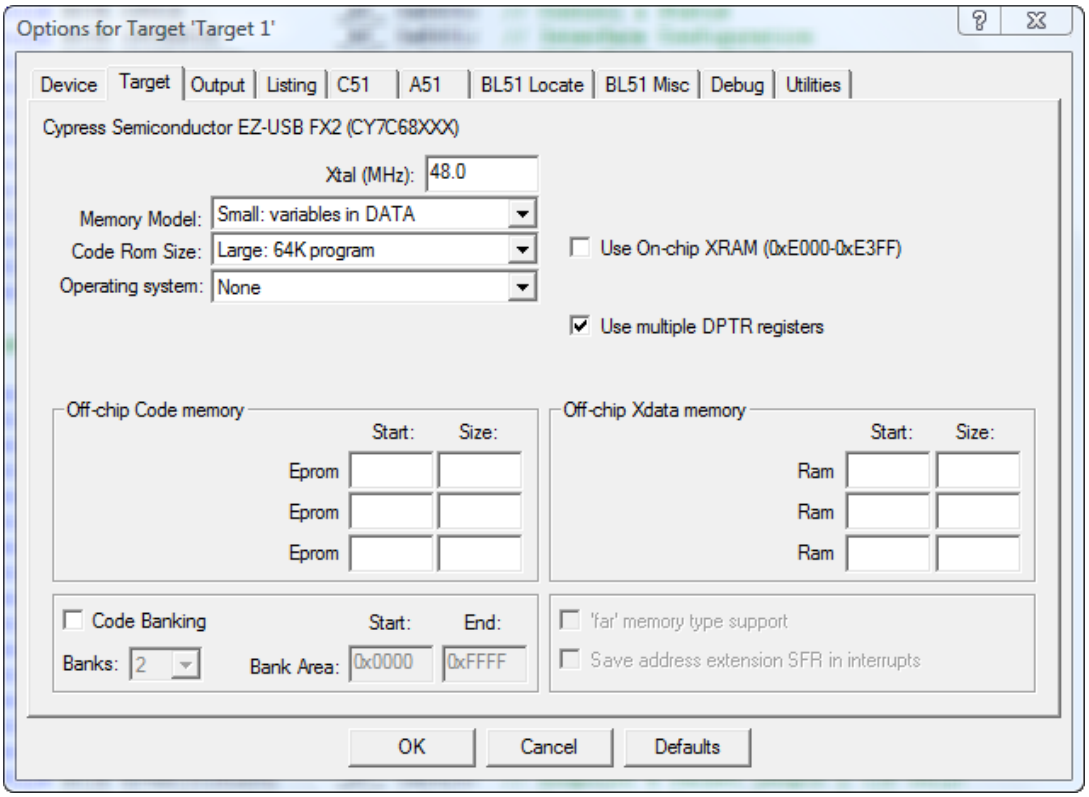
This method provides increase in execution speed.

Disadvantage

Increase of code complexity; care must be taken to ensure that the data pointers are initialized to the proper value before access.

Note EP2 and EP6 are quad-buffered and the EP2 data is constantly being moved to EP6. There must be five packets EZ-USB endpoint (EP2 plus EP6) buffers for EP2 to not be empty.

Figure 4. Target Tab Settings for Options for Target ‘Target 1’ Window



Use of Autopointers

EZ-USB provides two special data pointers called autopointers. These pointers automatically increment after each byte transfer. Using autopointers, the firmware can access contiguous blocks of on or off-chip data memory as a FIFO. A read from or write to an autopointer data register accesses the address pointed to by the corresponding autopointer address register, which increments on every data-register access. To read or write a contiguous block of memory (for example, an endpoint buffer) using an autopointer, load the autopointer's address register with the starting address of the block. Then, repeatedly read or write the autopointer's data register. This helps you achieve code size reduction and also speed of execution. This is because you are reading/writing a single memory location and the data pointer (DPTR) does not have to update for each memory access. This reduces the number of instruction cycles and code space used to increment the memory location to be accessed for each memory access. Define the preprocessor symbol **AUTOPTR** to use this method of memory access for looping EP2 data to EP6 in TD_Poll() function of firmware_optimization.c. Compile the firmware and note the amount of memory used.

Now undefine the preprocessor symbol **AUTOPTR** to use array arithmetic to loop the data. Compile the firmware; the difference in the amount of code memory used is seen.

Advantage

This method provides increase in execution speed.

Disadvantage

Increase of code complexity; care must be taken to ensure that the autopointers are initialized to the proper value before access.

Note The older part of EZ-USB series (AN21xx/FX) had only one autopointer. Therefore, one location (destination or source) address increment and so on must be handled by the firmware.

Use of Setup Data Pointer

The USB host sends device requests using control transfers over endpoint 0. Some requests require EZ-USB to return data over EP0 (for example, descriptor requests during enumeration). In these cases setup data pointer is useful. Just load the setup data pointer with the starting location of the data. It takes the packet length from the length field of the descriptor and fills and arms EP0 with the required data. This helps you achieve code size reduction and also speed of execution. This is because the setup data pointer is taking care of filling and arming the endpoint, 8051 execution cycles are not required other than to just write the start address to SUDPTRH:L (Setup data pointer high and low address register). This results in reduction of code space used. Define the preprocessor symbol **SETUPPTR** to use this method of memory access to send Device descriptor in SetupCommand() function of fw.c. Compile the firmware and note the amount of memory used.

Now undefine the preprocessor symbol **SETUPPTR** to manually load the EP0 buffer using array arithmetic and commit the packet by writing to EP0BCH:L registers. Compile the firmware; the difference in the amount of code memory used is seen.

Advantage

It provides increase in execution speed.

Note There are optimizations other than the ones described that are provided by the compiler based on settings. These optimizations are beyond the scope of this document and hence have not been discussed here.

Refer to the Technical Reference Manual for more details about the registers/SFRs.

Summary

This document explains the method of code-optimization of EZ-USB firmware. It provides information on how to leverage the special features of EZ-USB to our advantage. The firmware accompanying this application note demonstrates some of these methods.

About the Author

Name:	Gayathri Vasudevan
Title:	Applications Engineer
Contact:	gaya@cypress.com

Document History

Document Title: AN61244 - Firmware Optimization in EZ-USB®

Document Number: 001-61244

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	2924857	AASI	04/29/2010	New application note
*A	3998258	GAYA	05/13/2013	Updated in new template. Completing Sunset Review.
*B	5312716	GAYA	06/17/2016	Added link to related code examples on page 1. Updated the template.
*C	5702247	AESATP12	04/26/2017	Updated logo and copyright.

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

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