This application note contains software examples to use when getting started using MPC5500 and MPC5600 family devices. Complete code is available for downloading to target such as an evaluation board.

Table 1. MPC5500 & MPC5600 Family Definitions for this Application Note

<table>
<thead>
<tr>
<th>Family Name</th>
<th>Devices Included</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPC551x</td>
<td>MPC5514, MPC5515, MPC5516, MPC5517</td>
</tr>
<tr>
<td>MPC555x</td>
<td>MPC5533, MPC5534, MPC5553, MPC5554, MPC5561, MPC5565, MPC5566, MPC5567, MPC55632M, MPC55633M, MPC55634M</td>
</tr>
<tr>
<td>MPC56xxB/P/S</td>
<td>MPC5602B or C, MPC5603B, MPC5604 B or C, MPC5604P, MPC5602S, MPC5604S, MPC5606S</td>
</tr>
</tbody>
</table>

The family definitions in Table 1 are used to categorize the different devices discussed in this document. Examples included here will illustrate any differences between families, and between the different members in a family.
Source code, necessary build files, and debugger scripts are available for all examples. Example code provides two executable outputs: one that locates the program in internal flash, and another that locates the program in internal RAM.

For RAM-based programs, debugger scripts are provided to initialize the MMU, initialize SRAM, and set the instruction pointer to the beginning of main code.

For flash-based programs:

- The MMU is initialized by Boot Assist Module (BAM) code that generally executes after reset.
- SRAM is initialized by added code to a modified startup (“crt0”) file.
- The instruction pointer (as well as some reset configuration settings) is defined by the Reset Configuration Half Word (RCHW) in the boot section of the startup file for flash.

**NOTE**

For EVBs with 40 MHz Crystal:
MPC5561 and MPC5567 EVBs use 40 MHz crystals instead of the usual 8 MHz crystal. In this case, the default frequency will be 30 MHz instead of 12 MHz. Examples that alter the PLL frequency will need software modification for proper operation using a 40 MHz crystal. (For more information, see Section 10.2.2, “MPC555x,” part of Section 8.2, “Design,” in Section 8, “PLL: Initializing System Clock,” for code changes when using 40 MHz crystal.)
### Table 2. Devices and Applicable Examples

<table>
<thead>
<tr>
<th>Example</th>
<th>MPC551x Devices</th>
<th>MPC555x Devices</th>
<th>MPC56xxBPS Devices</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Time Base: Time Measurement</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>2 Interrupts: Decrementer</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>3 Interrupts: Fixed-Interval Timer</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>4 INTC: Software Vector Mode</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>5 INTC: Hardware Vector Mode</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>6 INTC: Software Vector Mode, VLE</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>7 INTC: Hardware Vector Mode, VLE</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>8 MMU: Create TLB Entry</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>9 Cache: Cache as RAM</td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>10 PLL: Initializing System Clock</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>11 PLL: Initializing System Clock</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12 FMPLL: Frequency Modulation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13 Modes: Low Power</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14 eDMA: Block Move</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>15 eSCI: Simple Transmit &amp; Receive</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>16 eSCI: LIN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17 LINFlex: LIN Transmit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18 eMIOS: Modulus Counter, OPWM</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>19 eMIOS: PEC, OPWFM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20 eTPU: Set 1 PWM Function</td>
<td>x</td>
<td></td>
<td>x (except MPC5561)</td>
</tr>
<tr>
<td>21 eQADC: Single Software Scan</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>22 ADC: SW Trigger, Cont Scan</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23 ADC-CTU: eMIOS Trigger</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24 DSPI: SPI to SPI</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>25 FlexCAN: Transmit and Receive</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>26 Flash: Configuration</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>
## Revision History

<table>
<thead>
<tr>
<th>Revision</th>
<th>Changes</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Minor text changes in various examples, plus code modifications for example 2, “Interrupts: Decrementer,” and example 19, “Flash: Configuration.”</td>
<td>11/2008</td>
</tr>
</tbody>
</table>
| 3        | • Added 3 new examples: PLL: System Clock (MPC56xxB/P/S), INTC: SW Vector Mode, VLE Instructions, INTC: HW Vector Mode, VLE Instructions  
          • Expanded and updated Flash: Configuration  
          • Updated 5 other examples to support MPC560xB, MPC560xP, MPC560xS (“MPC56xxB/P/S”)  
          • Revised FlexCAN example  
          • Minor text changes in other examples | 07/2009 |
| 4        | • Added 4 new examples: Modes: Low Power  
          LINFlex: LIN Transmit  
          ADC: Software trigger, Continuous Scan  
          CTU: eMIOS Trigger  
          • Updated other MPC56xxB/P/S examples  
          • Minor text changes in other examples | 04/2010 |
|          | The revision number was not incremented because no substantive changes were made to the content.  
          • Front page: Add SafeAssure branding.  
          • Title and first page: Add Qorivva branding.  
          • Back page: Apply new back page format. | 04/2012 |
1 Time Base: Time Measurement

1.1 Description

Task: Using the Time Base (TB), measure the number of system clocks to execute a section of code in any type of memory. Put the result into a register.

This program demonstrates how to manage special purpose registers (spr’s). Special purpose registers are not memory-mapped, and are read or written using a general purpose register (gpr). Only two instructions are used with spr’s: move to spr (mtspr) and move from spr (mfspr).

The 64-bit Time Base consists of two 32-bit special purpose registers: TBL for the lower 32 bits and TBU for the upper 32 bits. It is enabled by setting the Time Base Enable (TBE) bit in the Hardware Implementation Register 0 (spr HID0). When enabled it counts at system clock frequency, after some initial delay due to pipelining. If sysclk = 12 MHz (the default for MPC555x devices with an 8 MHz crystal), it would take about six minutes for TBL to overflow to TBU. MPC551x devices have a default frequency of 16 MHz.

NOTE

Debuggers may or may not stop the time base when code is not running, such as at a breakpoint.

Exercise: Modify the program to record the TBL and TBU at the start of the desired code sequence, and move their values to spr SPRG0 and spr SPRG1. At the end of the measured code, record both again and move their values to spr SPRG2 and spr SPRG3. Examine these registers and calculate the time difference.
## 1.2 Design

### Table 3. Time Base: Time Measurement

<table>
<thead>
<tr>
<th>Step</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Initialize Time Base = 0</td>
<td>li r4, 0</td>
<td>Load immediate data of 0 to r4</td>
</tr>
<tr>
<td></td>
<td>mttbu r4</td>
<td>Move r4 to TBU</td>
</tr>
<tr>
<td></td>
<td>mttbl r4</td>
<td>Move r4 to TBL</td>
</tr>
<tr>
<td>2. Enable Time Base: set HID0[TBE]=1</td>
<td>mfhid0 r5</td>
<td>Move from spr HID0 to r5</td>
</tr>
<tr>
<td></td>
<td>li r4, 0x4000</td>
<td>Load immediate data of 0x4000 to r4</td>
</tr>
<tr>
<td></td>
<td>or r5, r4, r5</td>
<td>OR r4 (0x4000 0000) with r5 (HID0 value)</td>
</tr>
<tr>
<td></td>
<td>mthid0 r5</td>
<td>Move result to HID0</td>
</tr>
<tr>
<td>3. Execute some code</td>
<td>nop</td>
<td></td>
</tr>
<tr>
<td></td>
<td>nop</td>
<td></td>
</tr>
<tr>
<td></td>
<td>nop</td>
<td></td>
</tr>
<tr>
<td></td>
<td>nop</td>
<td></td>
</tr>
<tr>
<td></td>
<td>nop</td>
<td></td>
</tr>
<tr>
<td>4. Record TBL</td>
<td>mftbl r5</td>
<td>Move TBL to r5 to store TBL value</td>
</tr>
</tbody>
</table>

## 1.3 Code

```assembly
li r4, 0        # INITIALIZE TIME BASE=0
mttbu r4        # Load immediate data of 0 to r4
mttbl r4        # Move r4 to TBU
mfhid0 r5       # MOVE TIME BASE
li r4, 0x4000   # Move from spr HID0 to r5 (copies HID0)
or r5, r4, r5   # Load immed. data of 0x4000 to r4
mthid0 r5       # OR r4 (0x4000 0000) with r5 (HID0 value)
mftbl r5        # Move result to HID0
nop              # EXECUTE SOME CODE
nop              # RECORD TBL
mftbl r5        # Move TBL to r5 to store TBL value
```
2 Interrupts: Decrementer

2.1 Description

Task: Use the Decrementer (DEC) exception to generate a periodic interrupt roughly every second. Design the interrupt handler, including interrupt service routine (ISR), to be written entirely in assembler without enabling nested interrupts. The ISR simply counts the number of Decrementer interrupts, toggles a GPIO output, and clears the Decrementer interrupt flag. Assume the system clock runs at the default frequency of 16 MHz for MPC551x (16 MHz Internal Reference Clock) or 12 MHz for MPC555x (1.5 × 8 MHz crystal).

Exercise: Connect the output to an LED on an MPC55xx evaluation board or oscilloscope. After verifying proper operation, change the Decrementer timeout to 1 kHz.

Figure 2. Decrementer Example

Table 4. Signals for Decrementer Example

<table>
<thead>
<tr>
<th>Signal</th>
<th>Pin Name</th>
<th>SIU PCR No.</th>
<th>Package Pin No.</th>
<th>Function Name</th>
<th>SIU PCR No.</th>
<th>Package Pin No.</th>
<th>EVB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>144 QFP</td>
<td>176 QFP</td>
<td>GPIO[114]</td>
<td></td>
<td>496 BGA</td>
<td>xPC 563M</td>
</tr>
<tr>
<td>GPIO</td>
<td>PH2</td>
<td>114</td>
<td>22 30 L1</td>
<td>GPIO[114]</td>
<td>M5 N3 L3 N3</td>
<td>416 BGA 324 BGA 208 BGA 144 QFP</td>
<td>PJ9–1</td>
</tr>
</tbody>
</table>

Qorivva Simple Cookbook, Rev. 4

Freescale Semiconductor 7
2.2 Design

For MPC551x, the DEC interrupt causes the CPU to vector to the sum of the common Interrupt Vector Prefix (IVPR0:19) plus a fixed offset for each interrupt, which is 16 bytes times the IVOR number. The MPC551x program flow is shown below.

For MPC555x, the DEC interrupt causes the CPU to vector to the sum of the common Interrupt Vector Prefix (IVPR0:15) plus the interrupt’s individual Interrupt Vector Offset (IVOR10:27). The MPC555x program flow is shown below, followed by design steps and a stack frame implementation.
Because the program’s interrupt handler is entirely in assembler, only the registers that are used will need to be saved in the stack. Two registers are used in the interrupt service routine, so only these two registers are saved and restored in the prologue and epilogue, besides the stack pointer (r1). External input interrupts, enabled by MSR[EE], are not re-enabled in the interrupt handler, so SRR0:1 need not be saved in the stack frame.

### Table 5. Initialization Steps

<table>
<thead>
<tr>
<th>Step</th>
<th>Relevant Bit Fields</th>
<th>Pseudo Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global variable</td>
<td></td>
<td>int DECctr = 0</td>
</tr>
<tr>
<td>initIrqVectors</td>
<td></td>
<td>spr IVPR = __IVPR_VALUE</td>
</tr>
<tr>
<td>initDEC</td>
<td></td>
<td>spr DEC = 0x00B7 1B00</td>
</tr>
<tr>
<td>initGPIO</td>
<td></td>
<td>spr DECAR = 0x00B7 1B00</td>
</tr>
<tr>
<td>enableExtIrq</td>
<td></td>
<td>wrteei 1</td>
</tr>
<tr>
<td>waitloop</td>
<td></td>
<td>while 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step</th>
<th>Relevant Bit Fields</th>
<th>Pseudo Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSC551x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSC555x</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 6. Stack Frame for Decrementer Interrupt (16 bytes)

<table>
<thead>
<tr>
<th>Stack Frame Area</th>
<th>Register</th>
<th>Location in Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit GPRs</td>
<td>r4</td>
<td>sp + 0x0C</td>
</tr>
<tr>
<td></td>
<td>r3</td>
<td>sp + 0x08</td>
</tr>
<tr>
<td>LR area</td>
<td>LR placeholder</td>
<td>sp + 0x04</td>
</tr>
<tr>
<td>back chain</td>
<td>r1 (which is sp)</td>
<td>sp</td>
</tr>
</tbody>
</table>
Table 7. IVOR10 (Decrementer) Interrupt Handler Steps  
(MPC555x: Must be 16 byte aligned and within 64KB of address in IVPR. 
MPC551x does not require alignment because it is the destination of a branch instruction.)

<table>
<thead>
<tr>
<th>Step</th>
<th>Relevant Bit Fields</th>
<th>Pseudo Code (MPC551x &amp; MPC555x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>prolog</td>
<td>Create stack frame</td>
<td>stwu sp, −0x10 (sp)</td>
</tr>
<tr>
<td></td>
<td>Save registers in stack frame</td>
<td>store required registers</td>
</tr>
<tr>
<td>DecISR</td>
<td>Increment DECctr</td>
<td>DECctr++</td>
</tr>
<tr>
<td></td>
<td>Toggle GPIO</td>
<td>SIU_GPDO[114] = SIU_GPDI[114]++</td>
</tr>
<tr>
<td></td>
<td>Clear decrementer interrupt flag</td>
<td>DIS = 1 spr TSR = 0x0800 0000</td>
</tr>
<tr>
<td>epilog</td>
<td>Restore registers from stack frame</td>
<td>restore registers</td>
</tr>
<tr>
<td></td>
<td>Restore stack frame space</td>
<td>addi sp, sp, 0x50</td>
</tr>
<tr>
<td></td>
<td>Return</td>
<td>rfi</td>
</tr>
</tbody>
</table>
2.3 Code

2.3.1 main.c file

/* main.c - Decrementer interrupt example */
/* Rev 1.0 April 19, 2004 S.Mihalik, Copyright Freescale, 2007 All Rights Reserved */
/* Rev 1.1 May 15 2006 SM- Changed GPIO205 to GPIO195 for use in 324, 208 packages */
/* Rev 1.2 Aug 13 2006 SM- Passed spr IVPR value from link file, changed GPIO pin */
/* and for MPC551x removed code to load spr IVOR10 */
/* Notes: */
/* 1. MMU not initialized; must be done by debug scripts or BAM */
/* 2. SRAM not initialized; must be done by debug scripts or in a crt0 type file */
#include "mpc563m.h" /* Use proper include file such as mpc5510.h or mpc5554.h */
extern IVOR10Handler();
extern uint32_t __IVPR_VALUE; /* Interrupt Vector Prefix value from link file */

uint32_t DECctr = 0;   /* Counter for Decrementer interrupts */
uint32_t GPDI_114_ADDR = (uint32_t)&SIU.GPDI[114].R;   /* GPDI[114] reg. addr. */
uint32_t GPDO_114_ADDR = (uint32_t)&SIU.GPDO[114].R;   /* GPDO[114] reg. addr. */

asm void initIrqVectors(void) {
    lis       r0, __IVPR_VALUE@h    /* IVPR value is passed from link file */
    ori       r0, r0,__IVPR_VALUE@l /* Note: IVPR lower bits are unused in MPC555x*/
    mtivpr    r0
    /* The following two lines are required for MPC555x, and are not used for MPC551x*/
    li       r0, IVOR10Handler@l    /* IVOR10(Dec) = lower half of handler address */
    mtivor10 r0
}

asm void initDEC(void) {
    lis      r0, 0x00B7      /* Load initial DEC value of 12M (0x00B7 1B00) */
    ori      r0, r0, 0x1B00
    mtdec    r0
    mtdecar  r0              /* Load same initial value to DECAR */
    lis      r0, 0x0440      /* Enable DEC interrupt and auto-reload*/
    mttcr    r0
    li       r0, 0x4000      /* Enable Time Base and Decrementer (set TBEN) */
    mthid0   r0
}

void main (void) {
    volatile uint32_t i=0;    /* Dummy idle counter */
    initIrqVectors();         /* Initialize interrupt vectors registers*/
    initDEC();                /* Initialize Decrementer routine */
    SIU.PCR[114].R = 0x0303;  /* Init. pin for GPIO output that can be read as input */
    asm (" wrteei 1");       /* Enable external interrupts (INTC, DEC, FIT) */
    while (1) { i++; }        /* Loop forever */
}
2.3.2 handler.s file

# handlers.s - Decrementer (IVOR10) interrupt example
# Rev 1.0: Sept 2, 2004, S Mihalik,
# Rev 1.1: May 15, 2006 S.M.- Use GPIO[195] instead of GPIO[205]
# Rev 1.2: Jun 13, 2007 S.M. Added .section .ivor handlers for linking
# and changed GPIO address to global variable
# Rev 1.3 Nov 1 2008 SM - Used r4 instead of r0 to clear DIS flag in spr TSR#
Copyright Freescale Semiconductor, Inc. 2007. All rights reserved

# STACK FRAME DESIGN: Depth: 4 words (0x10, or 16 bytes)
# ************* ______________
#   0x0C     *    r4     *    |GPR Save Area
#   0x08     *    r3     * ___|__________
#   0x04     * resvd- LR * Reserved for calling function
#   0x00     *  SP(GPR1) * Backchain (same as gpr1 in GPRs)
# *************

.globl IVOR10Handler
.extern DECctr            # Counter of Decrementer interrupts taken
.extern GPDI_114_ADDR     # GPDI[114] register address
.extern GPDO_114_ADDR     # GPDO[114] register address

.section .ivor_handlers

# Align IVOR handlers on a 16 byte boundary for MPC555x
# .align 4  # GHS, Cygnus, Diab(default) use .align 4
# .align 16 # CodeWarrior requires .align 16

IVOR10Handler:
prolog:                          # PROLOGUE
   stwu   r1, -0x10 (r1)          # Create stack frame and store back chain
   stw    r3, 0x08 (r1)
   stw    r4, 0x0C (r1)

DECisr:
   lis    r3, DECctr@ha           # Read DECctr
   addi   r4, r4, 0x1             # Increment DECctr
   stw    r4, DECctr@l (r3)       # Write back new DECctr

   lis    r3, GPDI_114_ADDR@ha    # Get pointer to memory containing GPDI[114] address
   lwz    r3, GPDI_114_ADDR@l(r3)
   addi   r4, r4, 1               # Add one to state for toggle effect
   lis    r3, GPDO_114_ADDR@ha    # Get pointer to memory containing GPDO[114] address
   lwz    r3, GPDO_114_ADDR@l(r3)
   stb    r4, 0 (r3)               # Output toggled GPIO 114 pin state to reg. GPDO[114]

   lis    r4, 0x0800              # Write "1" clear Dec Interrupt Status (DIS) flag
   mtspr  336, r4                 # DIS flag is in spr TSR (spr 336)

epilog:                          # EPILOGUE
   lwz    r4, 0x0C (r1)           # Restore gprs
   lwz    r3, 0x08 (r1)
   addi   r1, r1, 0x10            # Restore space on stack
   rfi                             # End of Interrupt
2.3.3   ivor_branch_table.s file (MPC551x only)

# ivor_branch_table.s - for use with MPC551x only
# Description: Branch table for 16 MPC551x core interrupts
# Copyright Freescale 2007. All Rights Reserved
# Rev 1.0 Jul 6 2007 S Mihalik - Initial version
# Rev 1.1 Aug 30 2007 SM - Made IVOR10Handler extern

.extern IVOR10Handler

.section .ivor_branch_table

.equs SIXTEENgetBytes, 16   # 16 byte alignment required for table entries
  # Diab compiler uses value of 4 (2**4=16)
  # CodeWarrior, GHS, Cygnus use 16

.align SIXTEENgetBytes
IVOR0trap:  b IVOR0trap  # IVOR 0 interrupt handler
.align SIXTEENgetBytes
IVOR1trap:  b IVOR1trap  # IVOR 1 interrupt handler
.align SIXTEENgetBytes
IVOR2trap:  b IVOR2trap  # IVOR 2 interrupt handler
.align SIXTEENgetBytes
IVOR3trap:  b IVOR3trap  # IVOR 3 interrupt handler
.align SIXTEENgetBytes
IVOR4trap:  b IVOR4trap  # IVOR 4 interrupt handler
.align SIXTEENgetBytes
IVOR5trap:  b IVOR5trap  # IVOR 5 interrupt handler
.align SIXTEENgetBytes
IVOR6trap:  b IVOR6trap  # IVOR 6 interrupt handler
.align SIXTEENgetBytes
IVOR7trap:  b IVOR7trap  # IVOR 7 interrupt handler
.align SIXTEENgetBytes
IVOR8trap:  b IVOR8trap  # IVOR 8 interrupt handler
.align SIXTEENgetBytes
IVOR9trap:  b IVOR9trap  # IVOR 9 interrupt handler
.align SIXTEENgetBytes
  b IVOR10Handler  # IVOR 10 interrupt handler (Decrementer)
.align SIXTEENgetBytes
IVOR11trap: b IVOR11trap  # IVOR 11 interrupt handler
.align SIXTEENgetBytes
IVOR12trap: b IVOR12trap  # IVOR 12 interrupt handler
.align SIXTEENgetBytes
IVOR13trap: b IVOR13trap  # IVOR 13 interrupt handler
.align SIXTEENgetBytes
IVOR14trap: b IVOR14trap  # IVOR 14 interrupt handler
.align SIXTEENgetBytes
IVOR15trap: b IVOR15trap  # IVOR15 interrupt handler
3 Interrupts: Fixed-Interval Timer

3.1 Description

Task: Use the Fixed-Interval Timer (FIT) exception to generate a periodic interrupt roughly every second. Design the FIT’s interrupt handler to allow the interrupt service routine (ISR) to be written in C. The FIT’s ISR simply counts interrupts and toggles a GPIO output. The interrupt handler will not enable nested interrupts. Assume the system clock runs at the default frequency, which is 16 MHz for an MPC551x and 12 MHz for an MPC555x with an 8 MHz crystal.

The link files, make file, and start up file are the same as for the Decrementer example except for the output file names.

Exercise: Connect the GPIO to an LED on an MPC55xx evaluation board or oscilloscope. After verifying proper operation, change the FIT timeout rate to about 4 Hz.

Figure 5. Fixed-Interval Interrupt Example

Table 8. Signals for Fixed-Interrupt Example

<table>
<thead>
<tr>
<th>Signal</th>
<th>Pin Name</th>
<th>SIU PCR No.</th>
<th>Package Pin No.</th>
<th>Function Name</th>
<th>SIU PCR No.</th>
<th>Package Pin No.</th>
<th>EVB</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPIO</td>
<td>PH2</td>
<td>114</td>
<td>144 QFP</td>
<td>GPIO[114]</td>
<td>114</td>
<td>496 BGA</td>
<td>xPC 563M</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>176 QFP</td>
<td></td>
<td></td>
<td>416 BGA</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>208 BGA</td>
<td></td>
<td></td>
<td>324 BGA</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>8 MHz (MPC563x)</td>
<td></td>
<td></td>
<td>208 BGA</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>16 MHz (MPC551x)</td>
<td></td>
<td></td>
<td>xPC</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>12 MHz (MPC555x)</td>
<td></td>
<td></td>
<td>563M</td>
<td></td>
</tr>
</tbody>
</table>
3.2 Design

For a FIT timeout to occur, the selected Time Base bit must transition from 0 to 1. As the Time Base continues to increment, the bit will transition back to 0 then to 1 again before the next timeout. Therefore for Time Base bit \( n \) (\( n \) is counted from the least significant bit of TBL), the timeout period is:

\[
2 \times 2^n \times \text{sysclock periods.}
\]

For one second timeout using a 12 MHz sysclk, 1 second = \( 2 \times 2^n \times 83 \frac{1}{3} \) ns. Solving for integer \( n \), we find when \( n = 22 \), there is about a 0.7 second timeout. This \( n = 22 \) corresponds to TBL bit \( 31 - n = TBL \) bit 9.

For MPC551x, the FIT interrupt causes the CPU to vector to the sum of the common Interrupt Vector Prefix (IVPR\(_{0:19}\)) plus a fixed offset for each interrupt, which is 16 bytes times the IVOR number. The MPC551x program flow is shown below.

![Figure 6. MPC551x Program Flow](image)

For MPC555x, the DEC interrupt causes the CPU to vector to the sum of the common Interrupt Vector Prefix (IVPR\(_{0:15}\)) plus the interrupt’s individual Interrupt Vector Offset (IVOR\(_{11:27}\)). The MPC555x program flow is shown below, followed by design steps and a stack frame implementation.

![Figure 7. MPC555x Program Flow](image)
The ISR will be written in C, so the handler’s prologue will save all the registers the compiler might use, namely the volatile registers as defined in the e500 ABI.\(^1\) External input interrupts, enabled by MSR[EE], are not re-enabled in the interrupt handler, so SRR0:1 need not be saved in the stack frame.

---

\(^1\) The SPE's accumulator is not saved in the MPC555x prologue. It is assumed SPE instructions are not used in this example.
Table 10. Stack Frame for FIT Interrupt Handler (80 bytes)

<table>
<thead>
<tr>
<th>Stack Frame Area</th>
<th>Register</th>
<th>Location in Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit GPRs</td>
<td>r12</td>
<td>sp + 0x4C</td>
</tr>
<tr>
<td></td>
<td>r11</td>
<td>sp + 0x48</td>
</tr>
<tr>
<td></td>
<td>r10</td>
<td>sp + 0x44</td>
</tr>
<tr>
<td></td>
<td>r9</td>
<td>sp + 0x40</td>
</tr>
<tr>
<td></td>
<td>r8</td>
<td>sp + 0x3C</td>
</tr>
<tr>
<td></td>
<td>r7</td>
<td>sp + 0x38</td>
</tr>
<tr>
<td></td>
<td>r6</td>
<td>sp + 0x34</td>
</tr>
<tr>
<td></td>
<td>r5</td>
<td>sp + 0x30</td>
</tr>
<tr>
<td></td>
<td>r4</td>
<td>sp + 0x2C</td>
</tr>
<tr>
<td></td>
<td>r3</td>
<td>sp + 0x28</td>
</tr>
<tr>
<td></td>
<td>r0</td>
<td>sp + 0x24</td>
</tr>
<tr>
<td>CR</td>
<td>CR</td>
<td>sp + 0x20</td>
</tr>
<tr>
<td>locals and padding</td>
<td>XER</td>
<td>sp + 0x1C</td>
</tr>
<tr>
<td></td>
<td>CTR</td>
<td>sp + 0x18</td>
</tr>
<tr>
<td></td>
<td>LR</td>
<td>sp + 0x14</td>
</tr>
<tr>
<td></td>
<td>padding</td>
<td>sp + 0x10</td>
</tr>
<tr>
<td></td>
<td>padding</td>
<td>sp + 0x0C</td>
</tr>
<tr>
<td></td>
<td>padding</td>
<td>sp + 0x08</td>
</tr>
<tr>
<td>LR area</td>
<td>LR placeholder</td>
<td>sp + 0x04</td>
</tr>
<tr>
<td>back chain</td>
<td>r1 (which is sp)</td>
<td>sp</td>
</tr>
</tbody>
</table>

Table 11. Interrupt Handler (IVOR11Handler): Fixed-Interval Timer

(MPC555x: Must be 16-byte aligned and within 64 KB of address in IVPR. MPC551x does not require alignment because it is the destination of a branch instruction.)

<table>
<thead>
<tr>
<th>Step</th>
<th>Relevant Bit Fields</th>
<th>Pseudo Code (MPC551x &amp; MPC555x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>prolog</td>
<td>Create stack frame</td>
<td>stwu sp, –0x50 (sp)</td>
</tr>
<tr>
<td></td>
<td>Save registers in stack frame</td>
<td>store required registers</td>
</tr>
<tr>
<td>FitISR</td>
<td>Increment FITctr</td>
<td>FITctr++</td>
</tr>
<tr>
<td></td>
<td>Toggle GPIO114</td>
<td>SIU_GPDO[114] = ~SIU_GPD[114]</td>
</tr>
<tr>
<td></td>
<td>Clear FIT interrupt status flag</td>
<td>FIS = 1</td>
</tr>
<tr>
<td>epilog</td>
<td>Restore registers from stack frame</td>
<td>restore registers</td>
</tr>
<tr>
<td></td>
<td>Restore stack frame space</td>
<td>addi sp, sp, 0x50</td>
</tr>
<tr>
<td></td>
<td>Return</td>
<td>rfi</td>
</tr>
</tbody>
</table>

Qorivva Simple Cookbook, Rev. 4

Freescale Semiconductor 17
3.3 Code

3.3.1 main.c file

/* main.c - Fixed Interval Timer interrupt example */
/* Rev 1.0 April 19, 2004 S.Mihalik, Copyright Freescale, 2004 All Rights Reserved */
/* Rev 1.1 Sept 1, 2004 SM - simplified, syntax changes */
/* Rev 1.2 Aug 15 2006 SM- Changed GPIO205 to GPIO195 for use in 324, 208 packages */
/* Rev 1.3 Aug 12 2006 SM - Changed variable i to volatile */
/* Rev 1.4 Jun 18 2007 SM - Passed spr IVPR value from link file, changed GPIO pin */
/* Notes: */
/* 1. MMU not initialized; must be done by debug scripts or BAM */
/* 2. SRAM not initialized; must be done by debug scripts or in cr0 type file */
#include "mpc563m.h" /* Use proper include file such as mpc5510.h or mpc5554.h */
extern IVOR11Handler(); /* Needed for MPC555x only */
extern uint32_t __IVPR_VALUE; /* Interrupt Vector Prefix value from link file */
uint32_t FITctr = 0; /* Counter for Fixed Interval Timer interrupts */
asm void initIrqVectors(void) {
    lis r0, __IVPR_VALUE@h /* IVPR value is passed from link file */
    ori r0, r0, __IVPR_VALUE@l /* Note: IVPR lower bits are unused in MPC555x */
    mtivpr r0
    /* The following two lines are required for MPC555x, and are not used for MPC551x */
    li r0, IVOR11Handler@l /* IVOR11 = lower half of handler address */
    mtivor11 r0
}
asm void initFIT(void) {
    li r0, 0 /* Initialize time base to 0 */
    mttbu r0
    mttbl r0
    lis r0, 0x0181 /* Enable FIT interrupt and set*/
    ori r0, r0, 0x4000 /* FP=0, FPEXT=A for 0.7 sec timeout */
    mtcr r0
    li r0, 0x4000 /* Enable Time Base and Decrementer (set TBEN) */
    mthid0 r0
}
void main (void) {
    volatile uint32_t i = 0; /* Dummy idle counter */
    initIrqVectors(); /* Initialize interrupt vectors registers*/
    initFIT(); /* Initialize FIT routine */
    SIU.PCR[114].R= 0x0303; /* Initialize GPIO as output. */
    asm (" wrteei 1"); /* Enable external interrupts (INTC, DEC, FIT) */
    while (1) { i++; } /* Loop forever */
}
asm void ClrFitFlag(void) {
    lis r0, 0x0400
    mttsr r0 /* Write "1" clear FIT Interrupt Status flag */
}
void FitISR(void) {
    FITctr++; /* Increment interrupt counter */
    SIU.GPDO[114].R = ~SIU.GPDI[114].R; /* Toggle GPIO output */
    ClrFitFlag(); /* Clear FIT's flag */
}
3.3.2 handlers.file

# handlers.s - FIT (IVOR11) interrupt example
# Rev 1.0: April 9, 2004, S Mihalik,
# Rev 1.1: June 18, 2007 SM Added .section .ivor_handlers for linking
# Rev 1.2: Aug 23 2007 DF: Made FitISR .extern
# Copyright Freescale Semiconductor, Inc. 2007. All rights reserved

# STACK FRAME DESIGN: Depth: 20 words (0xA0, or 80 bytes)
# ************* ______________
#   0x4C     *  GPR12    *    ^
#   0x48     *  GPR11    *    |
#   0x44     *  GPR10    *    |
#   0x40     *  GPR9     *    |
#   0x3C     *  GPR8     *    |
#   0x38     *  GPR7     *    |
#   0x34     *  GPR6     *    |
#   0x30     *  GPR5     *    |
#   0x2C     *  GPR4     *    |
#   0x28     *  GPR3     *    |
#   0x24     *  GPR0     * v  
#   0x20     *  CR       * __CR__________
#   0x1C     *  XER      *    ^
#   0x18     *  CTR      *    |
#   0x14     *  LR       * locals & padding for 16 B alignment
#   0x10     * padding    *    |
#   0x0C     * padding    *    |
#   0x08     * padding    * v  
#   0x04     * resvd- LR * Reserved for calling function
#   0x00     * SP       * Backchain (same as gpr1 in GPRs)
# *************

.extern FitISR
.globl IVOR11Handler
.section .ivor_handlers

.align 16       # Align IVOR handlers on a 16 byte (2**4) boundary
# GHS, Cygnus, Diab(default) use .align 4; CodeWarrior .align 16

prolog:       # PROLOGUE
    stwu r1, -0x50 (r1) # Create stack frame and store back chain
    stw r12, 0x4C (r1) # Store gprs
    stw r11, 0x48 (r1)
    stw r10, 0x44 (r1)
    stw r9, 0x40 (r1)
    stw r8, 0x3C (r1)
    stw r7, 0x38 (r1)
    stw r6, 0x34 (r1)
    stw r5, 0x30 (r1)
    stw r4, 0x2C (r1)
    stw r3, 0x28 (r1)
    stw r0, 0x24 (r1)
    mfCR r0     # Store CR
    stw r0, 0x20 (r1) *
    mfXER r0    # Store XER
    stw r0, 0x1C (r1)
    mfCTR r0   # Store CTR
    stw r0, 0x18 (r1)
    mfLR r0    # Store LR
    stw r0, 0x14 (r1)
    bl FitISR  # Execute FIT ISR, but return here
epilog:                # EPILOGUE
   lwz    r0,  0x14 (r1) # Restore LR
   mtLR   r0
   lwz    r0,  0x18 (r1) # Restore CTR
   mtCTR  r0
   lwz    r0,  0x1C (r1) # Restore XER
   mtXER  r0
   lwz    r0,  0x20 (r1) # Restore CR
   mtcrf  0xff, r0
   lwz    r0,  0x24 (r1) # Restore gprs
   lwz    r3,  0x28 (r1)
   lwz    r4,  0x2C (r1)
   lwz    r5,  0x30 (r1)
   lwz    r6,  0x34 (r1)
   lwz    r7,  0x38 (r1)
   lwz    r8,  0x3C (r1)
   lwz    r9,  0x40 (r1)
   lwz    r10, 0x44 (r1)
   lwz    r11, 0x48 (r1)
   lwz    r12, 0x4C (r1)
   addi   r1, r1, 0x50   # Restore space on stack
   rfi    # End of Interrupt
### 3.3.3 ivor_branch_table.s file (MPC551x only)

```assembly
.extern IVOR11Handler
.section .ivor_branch_table

.equ SIXTEEN_BYTES, 16   # 16 byte alignment required for table entries
# Diab compiler uses value of 4 (2**4=16)
# CodeWarrior, GHS, Cygnus use 16

.align SIXTEEN_BYTES
IVOR0trap:  b IVOR0trap  # IVOR 0 interrupt handler
.align SIXTEEN_BYTES
IVOR1trap:  b IVOR1trap  # IVOR 1 interrupt handler
.align SIXTEEN_BYTES
IVOR2trap:  b IVOR2trap  # IVOR 2 interrupt handler
.align SIXTEEN_BYTES
IVOR3trap:  b IVOR3trap  # IVOR 3 interrupt handler
.align SIXTEEN_BYTES
IVOR4trap:  b IVOR4trap  # IVOR 4 interrupt handler
.align SIXTEEN_BYTES
IVOR5trap:  b IVOR5trap  # IVOR 5 interrupt handler
.align SIXTEEN_BYTES
IVOR6trap:  b IVOR6trap  # IVOR 6 interrupt handler
.align SIXTEEN_BYTES
IVOR7trap:  b IVOR7trap  # IVOR 7 interrupt handler
.align SIXTEEN_BYTES
IVOR8trap:  b IVOR8trap  # IVOR 8 interrupt handler
.align SIXTEEN_BYTES
IVOR9trap:  b IVOR9trap  # IVOR 9 interrupt handler
.align SIXTEEN_BYTES
IVOR10trap: b IVOR10trap # IVOR 10 interrupt handler
.align SIXTEEN_BYTES
    b IVOR11Handler  # IVOR 11 interrupt handler (Fixed Interval Timer)
    .align SIXTEEN_BYTES
IVOR12trap: b IVOR12trap # IVOR 12 interrupt handler
    .align SIXTEEN_BYTES
IVOR13trap: b IVOR13trap # IVOR 13 interrupt handler
    .align SIXTEEN_BYTES
IVOR14trap: b IVOR14trap # IVOR 14 interrupt handler
    .align SIXTEEN_BYTES
IVOR15trap: b IVOR15trap # IVOR15 interrupt handler
```
4 INTC: Software Vector Mode

4.1 Description

**Task:** Using the Interrupt Controller (INTC) in software vector mode, this program provides two interrupts that show nesting. The first interrupt is generated from an eMIOS channel at a 1 kHz rate using the MPC555x default system clock. For the MPC551x, the faster default system clock produces an eMIOS channel timeout rate of approximately $1 \text{kHz} \times \frac{16}{12} = 1.33 \text{kHz}$. The interrupt handler will re-enable interrupts and later, in its interrupt service routine (ISR), invoke a second interrupt every other time. This provides an approximate 1 millisecond task (ISR) from the eMIOS channel and an approximate 2 millisecond task (ISR) from the software interrupt. The software interrupt will have a higher priority, so the 1 eMIOS ISR is preempted by the software interrupt. Both ISRs will have a counter.

This example is identical to INTC: Hardware Vector Mode except for the differences between software and hardware vector modes.

**Note:** eMIOS channel 0 interrupt vector numbers are different on MPC551x and MPC555x devices. Also, to generate the timed interrupt, the eMIOS channel mode will be the modulus counter for MPC555x devices that have that mode, or the newer modulus counter buffered mode.

The ISRs will be written in C, so the appropriate registers will be saved and restored in the handler. The SPE will not be used, so its accumulator will not be saved in the stack frame of the interrupt handler.

**Exercise:** Write a third interrupt handler that uses a software interrupt of a higher priority than the others. Invoke this new software interrupt from one of the other ISRs.

---

**Figure 8. Software Vector Mode Example**
4.2 Design

The overall program flow for MPC551x is shown below.

When an interrupt occurs, it is routed to either or both cores, as defined in the INTC_PSR for that vector. In this example only one processor is selected in the MPC551x for interrupts, processor 0 (e200z1). The selection is defined in INTC_PSR for each enabled interrupt, which here is eMIOS channel 0 and software interrupt 4.

For MPC551x, there can be a different ISR Vector table for each core because each core has its own special purpose register, IVPR. The Vector Table Base Address is defined in each core’s INTC_ACKR, that is, either INTC_ACK_PRC0 for e200z1 and INTC_IACK_PRC1 for e200z0. This example uses only the e200z1 core.

Figure 9. MPC551x Program Flow
The overall program flow for MPC555x is shown below.

Figure 10. MPC555x Program Flow

External input interrupt taken

<table>
<thead>
<tr>
<th>IntcIsrVectors.c file</th>
<th>handlers.s file</th>
<th>main.c file</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISR Vector Table:</td>
<td>vector to IVPR0_{15} + IVOR4_{16:27}</td>
<td>main {</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>init IVPR, IVOR4</td>
</tr>
<tr>
<td>SW IRQ 4 ISR addr</td>
<td></td>
<td>init INTC</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>init eMIOS Chan 0</td>
</tr>
<tr>
<td>eMIOS Ch 0 ISR addr</td>
<td></td>
<td>init SW 4 interrupt</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>enable interrupts</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>wait forever</td>
</tr>
</tbody>
</table>

IVOR4Handler:
prologue:
save SRR0:1,
fetch ISR vector,
re-enable interrupts
save registers
branch to fetched ISR vector
epilogue (return from ISR):
restore most registers
ensure prior stores completed
disable interrupts
write to EOIR to restore priority
restore SRR0:1
return

emiosCh0ISR {  
increment counter  
if odd count, invoke SW 4 IRQ  
clear flag  
}

SwIrq4ISR {  
increment counter  
clear flag  
}
## 4.2.1 Initialization

### Table 12. Initialization: INTC in Software Vector Mode

<table>
<thead>
<tr>
<th>Step</th>
<th>Relevant Bit Fields</th>
<th>Pseudo Code</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Variable Initializations</strong></td>
<td>Counter for eMIOS channel 0 interrupts</td>
<td>int emiosCh0Ctr = 0</td>
</tr>
<tr>
<td></td>
<td>Counter for software 4 interrupts</td>
<td>int SWirq4Ctr = 0</td>
</tr>
<tr>
<td><strong>Table Initializations</strong></td>
<td>Load INTC’s ISR vector table with addresses for:</td>
<td>&amp;Swirq4ISR &amp;emiosCh0ISR</td>
</tr>
<tr>
<td></td>
<td>INTC Vector 4 ISR (SW interrupt request 4)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>INTC Vector 51 ISR (eMIOS channel 0)</td>
<td></td>
</tr>
<tr>
<td>initIqVectors</td>
<td>Load the common interrupt vector prefix to spr IVPR</td>
<td>spr IVPR = __IVPR_VALUE</td>
</tr>
<tr>
<td></td>
<td>MPC555x: Initialize external input interrupt offset to the lower half of IVOR4 handler’s address</td>
<td></td>
</tr>
<tr>
<td>initINTC</td>
<td>Initialize INTC:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Configure for software vector mode (HVEN=0)</td>
<td>INTC_MCR = 0</td>
</tr>
<tr>
<td></td>
<td>• Keep vector offset (VTES) at four bytes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(MPC551x note: only proc’r 0, e200z1 is used here)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• MPC551x: Initialize external input interrupt offset to the lower half of IVOR4 handler’s address</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• MPC555x: Initialize external input interrupt offset to the lower half of IVOR4 handler’s address</td>
<td></td>
</tr>
<tr>
<td>initEMIOS</td>
<td>Initialize eMIOS global settings for all channels:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Prescale sysclk by 12 for eMIOS clk</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Enable eMIOS clock</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Enable global time base</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Enable stopping channels in debug mode</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MPC551x: Ensure channel is not disabled</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Channel 0 Interrupt: raise priority to 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• MPC551x: Select processor(s) to interrupt</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• MPC551x: Select processor(s) to interrupt</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Channel 0: set max count = 1,000 clks (~1 µs each)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Channel 0: Enable channel as up counter &amp; enable IRQ</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Mode (MPC551x, 563x) = mod. up counter buffered</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Mode (MPC555x) = modulus up counter</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Counter bus select = internal counter</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Channel prescaler = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Enable channel prescaler</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Enable freezing count in debug mode</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Assign flag to assert IRQ (instead of DMA)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Enable flag to request IRQ</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MPC555x:</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>initSwirq4</strong> Software Interrupt 4: raise priority to 2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• MPC551x: Specify which processor(s) to interrupt</td>
<td></td>
</tr>
<tr>
<td></td>
<td>enableExtIq</td>
<td>Enable recognition of requests to INTC by lowering the INTC’s current priority to 0 from default of 15</td>
</tr>
<tr>
<td></td>
<td>waitloop</td>
<td>wait forever</td>
</tr>
</tbody>
</table>

Qorivva Simple Cookbook, Rev. 4

Freescale Semiconductor

25
### 4.2.2 Interrupt Handler

#### Table 13. Stack Frame for INTC Interrupt Handler

<table>
<thead>
<tr>
<th>Stack Frame Area</th>
<th>Register</th>
<th>Location in Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit GPRs</td>
<td>r12</td>
<td>sp + 0x4C</td>
</tr>
<tr>
<td></td>
<td>r11</td>
<td>sp + 0x48</td>
</tr>
<tr>
<td></td>
<td>r10</td>
<td>sp + 0x44</td>
</tr>
<tr>
<td></td>
<td>r9</td>
<td>sp + 0x40</td>
</tr>
<tr>
<td></td>
<td>r8</td>
<td>sp + 0x3C</td>
</tr>
<tr>
<td></td>
<td>r7</td>
<td>sp + 0x38</td>
</tr>
<tr>
<td></td>
<td>r6</td>
<td>sp + 0x34</td>
</tr>
<tr>
<td></td>
<td>r5</td>
<td>sp + 0x30</td>
</tr>
<tr>
<td></td>
<td>r4</td>
<td>sp + 0x2C</td>
</tr>
<tr>
<td></td>
<td>r3</td>
<td>sp + 0x28</td>
</tr>
<tr>
<td></td>
<td>r0</td>
<td>sp + 0x24</td>
</tr>
<tr>
<td>CR</td>
<td>CR</td>
<td>sp + 0x20</td>
</tr>
<tr>
<td>locals and padding</td>
<td>XER</td>
<td>sp + 0x1C</td>
</tr>
<tr>
<td></td>
<td>CTR</td>
<td>sp + 0x18</td>
</tr>
<tr>
<td></td>
<td>LR</td>
<td>sp + 0x14</td>
</tr>
<tr>
<td></td>
<td>SRR1</td>
<td>sp + 0x10</td>
</tr>
<tr>
<td></td>
<td>SRR0</td>
<td>sp + 0x0C</td>
</tr>
<tr>
<td></td>
<td>padding</td>
<td>sp + 0x08</td>
</tr>
<tr>
<td>LR area</td>
<td>LR placeholder</td>
<td>sp + 0x04</td>
</tr>
<tr>
<td>back chain</td>
<td>r1</td>
<td>sp</td>
</tr>
</tbody>
</table>
### Table 14. IVOR4 Interrupt Handler (INTC in Software Vector Mode)

(Contents of the table are too long to be included here, but it describes steps for creating a stack frame, saving registers, setting up ISRs, and restoring registers, among other things.)

<table>
<thead>
<tr>
<th>Step</th>
<th>Relevant Bit Fields</th>
<th>Pseudo Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>prolog</td>
<td>Create stack frame</td>
<td>stwu sp, -0x50 (sp)</td>
</tr>
<tr>
<td></td>
<td>Save SRR0:1 because nested interrupts will be allowed</td>
<td>store SRR0:1, r3 registers to stack frame</td>
</tr>
<tr>
<td></td>
<td>Read pointer into ISR Vector Table</td>
<td>r3 = INTC_IACKR_PRC0</td>
</tr>
<tr>
<td></td>
<td>Re-enable external interrupts by setting MSR[EE]</td>
<td>EE = 1 wrteei 1</td>
</tr>
<tr>
<td></td>
<td>Read ISR address from ISR Vector Table and store into LR</td>
<td>lwz r3, 0x0 (r3) mlti r3</td>
</tr>
<tr>
<td></td>
<td>Save other appropriate registers for C ISR</td>
<td>store other registers to stack frame</td>
</tr>
<tr>
<td></td>
<td>Branch to ISR, saving return address</td>
<td>blrl</td>
</tr>
<tr>
<td>epilog</td>
<td>Restore registers from stack frame except SRR0:1 and two working registers</td>
<td>load most registers from stack frame</td>
</tr>
<tr>
<td></td>
<td>Ensure interrupt flag in ISR has completed clearing before writing to INTC_EOIR</td>
<td>mbar</td>
</tr>
<tr>
<td></td>
<td>Disable external interrupts by clearing MSR[EE]</td>
<td>EE = 0 wrteei 0</td>
</tr>
<tr>
<td></td>
<td>Restore former INTC’s Current Priority</td>
<td>write 0 to INTC_EOIR_PRC0</td>
</tr>
<tr>
<td></td>
<td>Restore SRR0:1 and working registers</td>
<td>restore SRR0:1, working registers from stack frame</td>
</tr>
<tr>
<td></td>
<td>Restore stack frame space</td>
<td>addi sp, sp, 0x50</td>
</tr>
<tr>
<td></td>
<td>Return</td>
<td>rfi</td>
</tr>
</tbody>
</table>

### 4.2.3 Interrupt Service Routines (ISRs)

#### Table 15. ISR for eMIOS Channel 0

<table>
<thead>
<tr>
<th>Step</th>
<th>Relevant Bit Fields</th>
<th>Pseudo Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>emiosCh0ISR</td>
<td>Increment emiosCh0Ctr</td>
<td>emiosCh0Ctr ++</td>
</tr>
<tr>
<td></td>
<td>If emiosCh0Ctr is even, invoke Software Interrupt 4</td>
<td>if ((emiosCh0Ctr&amp;1)), INTC_SSCIR[4] = 2</td>
</tr>
<tr>
<td></td>
<td>Clear eMIOS Ch 0 interrupt flag by writing 1 to it</td>
<td>FLAG = 1 EMIOS_CSR0[FLAG] = 1</td>
</tr>
</tbody>
</table>

#### Table 16. ISR for Software Interrupt 4

<table>
<thead>
<tr>
<th>Step</th>
<th>Relevant Bit Fields</th>
<th>Pseudo Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>swIRQ4ISR</td>
<td>Increment SWirq4ctr</td>
<td>SWirq4ctr++</td>
</tr>
<tr>
<td></td>
<td>Clear Software IRQ 4</td>
<td>CLR = 1 INTC_SSCIR[4] = 1</td>
</tr>
</tbody>
</table>
### 4.3 Code

#### 4.3.1 main.c file

```c
/* main.c - Software vector mode program using C isr */
/* Jan 15, 2004 S.Mihalik -- Copyright Freescale, 2004. All Rights Reserved */
/* Feb 9, 2004 S.M. - removed unused cache init & enabled eMIOS FREEZE */
/* Mar 4, 2004 S.M. - added software interrupt 7 code */
/* May 19,2006 S.M.- renamed SWirq7Ctr to SWirq4Ctr for consistency */
/* Aug 12 2006 S.M. - made i volatile (to get fewer Nexus messages in loop) */
/* Jul 17 2007 SM - Passed IVPR value from link file, used relevant names */
/* on even count eMIOS Ch 0 ISRs, changed timeout to 1 msec*/
/* and changed EMIOS_MCR[PRE] & EMIOS Chan 0 A Register values */
/* Copyright Freescale Semiconductor, In.c 2007 All rights reserved. */
/* Notes: */
/* 1. MMU not initialized; must be done by debug scripts or BAM */
/* 2. SRAM not initialized; must be done by debug scripts or in a crt0 type file */
/* 3. Cache is not used */
#include "mpc563m.h" /* Use proper include file like mpc5510.h or mpc5544.h */

extern IVOR4Handler();
extern uint32_t __IVPR_VALUE; /* Interrupt Vector Prefix value from link file*/
extern const vuint32_t IntcIsrVectorTable[];
int emiosCh0Ctr = 0;  /* Counter for eMIOS channel 0 interrupts */
int SWirq4Ctr = 0;    /* Counter for software interrupt 4 */
asm void initIrqVectors(void) {
  lis      r3, __IVPR_VALUE@h      /* IVPR value is passed from link file */
  ori      r3, r3, __IVPR_VALUE@l
  mtivpr   r3
  /* The following two lines are required for MPC555x, and are not used for MPC551x*/
  li  r3, IVOR4Handler@l      /* IVOR4 = lower half of handler address */
  mtivor4r3
}

void initINTC(void) {
  /* Use the first 3 or next 3 lines: */
  /* Use the first 3 or next 3 lines for MPC551x or MPC555x: */
  /* INTC.MCR.B.HVEN PRC0 = 0; */ /* MPC551x Proc'r 0: initialize for SW vector mode */
  /* INTC.MCR.B.VTES PRC0 = 0; */ /* MPC551x Proc'r 0: default vector table 4B offsets*/
  /* INTC.MCR.B.HVEN = 0; */ /* MPC555x: initialize for SW vector mode */
  /* INTC.MCR.B.VTES = 0; */ /* MPC555x: Use default vector table 4B offsets */
  /* INTC.IACKR.R = (uint32_t) &IntcIsrVectorTable[0]; */ /* MPC555x INTC ISR table base */
}

void initEMIOS(void) {
  EMIOS.MCR.B.GPRE= 11;         /* Divide sysclk by (11+1) for eMIOS clock */
  EMIOS.MCR.B.GPREN = 1;         /* Enable eMIOS clock */
  EMIOS.MCR.B.GTBE = 1;          /* Enable global time base */
  EMIOS.MCR.B.FRZ = 1;           /* Enable stopping channels when in debug mode */
  /* Following for MPC551x only: */
  /* EMIOS.UCDIS.R = 0; */ /* Ensure all channels are enabled */
  /* Use one of the following two lines: */
  /* INTC.PSR[58].R = 1; */ /* MPC551x: Raise eMIOS chan 0 IRQ priority = 1 */
  /* INTC.PSR[51].R = 1; */ /* MPC555x: Raise eMIOS chan 0 IRQ priority = 1 */
  EMIOS.CH[0].CADR.R = 999;      /* Period will be 999+1 = 1000 channel clocks */
  /* Use one of the next two lines: MCB mode is not in all MPC555x devices */
  EMIOS.CH[0].CCR.B.MODE = 0x50; /* MPC551x or MPC563x: Mod. Ctr. Buffered (MCB) */
  EMIOS.CH[0].CCR.B.MODE = 0x10; /* MPC555x: Modulus Counter (MC), internal clk */
  EMIOS.CH[0].CCR.B.BSL = 0x3;   /* Use internal counter */
  EMIOS.CH[0].CCR.B.UCPREN = 1; /* Enable prescaler; uses default divide by 1 */
  EMIOS.CH[0].CCR.B.FREN = 1;   /* Freeze channel registers when in debug mode */
  EMIOS.CH[0].CCR.B.FEN=1; /* Flag enables interrupt */
}
```

Qorivva Simple Cookbook, Rev. 4
void initSwIrq4(void) {
    INTC.PSR[4].R = 2;              /* Software interrupt 4 IRQ priority = 2 */
}

void enableIrq(void) {
    /* Use one of the following two lines to lower the INTC current priority */
    /*INTC.CPR_PRC0.B.PRI = 0; */  /* MPC551x Proc'r 0: Lower INTC's current priority */
    INTC.CPR.B.PRI = 0;          /* MPC555x: Lower INTC's current priority */
    asm(" wrteei 1");           /* Enable external interrupts */
}

void main (void) {
    vuint32_t i = 0;    /* Dummy idle counter */
    intIrqVectors();   /* Initialize exceptions: only need to load IVPR */
    initINTC();         /* Initialize INTC for software vector mode */
    initEMIOS();        /* Initialize eMIOS channel 0 for 1kHz IRQ, priority 2 */
    initSwIrq4();       /* Initialize software interrupt 4 */
    enableIrq();        /* Ensure INTC current priority=0 & enable IRQ */

    while (1) {
        i++;
    }
}

void emiosCh0ISR(void) {
    emiosCh0Ctr++;               /* Increment interrupt counter */
    if ((emiosCh0Ctr & 1)==0) {  /* If emiosCh0Ctr is even*/
        INTC.SSCIR[4].R = 2;     /* then invoke software interrupt 4 */
    }
    EMIOS.CH[0].CSR.B.FLAG=1;    /* Clear channel's flag */
}

void SwIrq4ISR(void) {
    SWirq4Ctr++;                 /* Increment interrupt counter */
    INTC.SSCIR[4].R = 1;         /* Clear channel's flag */
}
4.3.2 handlers.s file

# handlers.s - INTC software vector mode example
# Description: Creates prolog, epilog for C ISR and enables nested interrupts
# Rev 1.0: April 23, 2004, S Mihalik,
# Rev 1.1 Aug 2, 2004 SM - delayed writing to EOIR until after disabling EE in epilog
# Rev 1.2 Sept 8 2004 SM - optimized & corrected r3,r4 restore sequence from rev 1.1
# Rev 1.2 Sept 21 2004 SM - optimized by minimizing time interrupts are disabled
# Rev 1.3 Jul 2 2007 SM - Changes for MPC551x and mapped to .ivor_handlers section
# Copyright Freescale Semiconductor, Inc. 2007. All rights reserved

# STACK FRAME DESIGN: Depth: 20 words (0xA0, or 80 bytes)
# ************* ______________
#   0x4C     *  GPR12    *    ^
#   0x48     *  GPR11    *    |
#   0x44     *  GPR10    *    |
#   0x40     *  GPR9     *    |
#   0x3C     *  GPR8     *    |
#   0x38     *  GPR7     *    |
#   0x34     *  GPR6     *    |
#   0x30     *  GPR5     *    |
#   0x2C     *  GPR4     *    |
#   0x28     *  GPR3     *    |
#   0x24     *  GPR2     *    |
#   0x20     *  GPR1     *    |
#   0x1C     *  XER      *    |
#   0x18     *  CTR      *    |
#   0x14     *  SR1      *    |
#   0x10     *  SRR0     *    |
#   0x0C     *  SRR1     *    |
#   0x08     *  padding  *    |
#   0x04     * resvd- LR * Reserved for Calling function
#   0x00     *  SP       * Backchain (same as gpr1 in GPRs)
# *************

.section .ivor_handlers
.globl IVOR4Handler
.align 16    # Align IVOR handlers on a 16 byte boundary for MPC555x
   # GHS, Cygnus, Diab(default) use .align 4; CodeWarrior .align 16

.equ INTC_IACKR_PRC0, 0xfff48010   # MPC551x: Proc'r 0 Interrupt Acknowledge Reg addr
.equ INTC_EOIR_PRC0, 0xfff48018    # MPC551x: Proc'r 0 End Of Interrupt Reg. addr
.equ INTC_IACKR, 0xfff48010        # MPC555x: Interrupt Acknowledge Reg. addr.
.equ INTC_EOIR, 0xfff48018         # MPC555x: End Of Interrupt Reg. addr.

IVOR4Handler:
prolog:                       # PROLOGUE
   stwu    r1, -0x50 (r1)      # Create stack frame and store back chain
   stw     r3,  0x28 (r1)      # Store a working register
   mfsrr0  r3                  # Store SRR0:1 (must be done before enabling EE)
   stw     r3,  0x0C (r1)
   mfsrr1  r3
   stw     r3,  0x10 (r1)

# The following two lines are for MPC551x processors: */
# lis    r3, INTC_IACKR_PRC0@ha  # Read proc'0 ptr into ISR Vector Table, store in r3
# lwz    r3, INTC_IACKR_PRC0@l(r3)
# The following two lines are for MPC555x processors: */
# lis    r3, INTC_IACKR@ha       # Read pointer into ISR Vector Table & store in r3
# lwz    r3, INTC_IACKR@l(r3)   # Read ISR address from ISR Vector Table using pointer

wrteei 1            # Set MSR[EE]=1(must wait a couple clocks after reading IACKR)
   stw     r4,  0x2C (r1)      # Store a second working register
   mf1r    r4                   # Store LR (LR will be used for ISR Vector)
   stw     r4,  0x14 (r1)
   mtlr    r3                    # Store ISR address to LR to use for branching later
stw r12, 0x4C (r1)  # Store rest of gprs
stw r11, 0x48 (r1)
stw r10, 0x44 (r1)
stw r9, 0x40 (r1)
stw r8, 0x3C (r1)
stw r7, 0x38 (r1)
stw r6, 0x34 (r1)
stw r5, 0x30 (r1)
stw r0, 0x24 (r1)
mfcrr r3                   # Store CR
stw r3, 0x20 (r1)          # Store XER
stw r3, 0x1C (r1)          # Store CTR
stw r3, 0x18 (r1)
blrl                          # Branch to ISR, but return here
epilog:                         # EPILOGUE
lwz r3, 0x14 (r1)          # Restore LR
mtlr r3
lwz r3, 0x18 (r1)          # Restore CTR
mtcctr r3                   # Restore XER
mtxer r3
lwz r3, 0x20 (r1)          # Restore CR
mtcclf 0xff, r3
lwz r0, 0x24 (r1)          # Restore other gprs except working registers
lwz r5, 0x30 (r1)
lwz r6, 0x34 (r1)
lwz r7, 0x38 (r1)
lwz r8, 0x3C (r1)
lwz r9, 0x40 (r1)
lwz r10, 0x44 (r1)
lwz r11, 0x48 (r1)
lwz r12, 0x4C (r1)
mbar 0                     # Ensure store to clear interrupt's flag bit completed
# The following line is for the MPC551x:
# lis r3, INTC_EOIR_PRC0@ha MPC551x: Load upper half of proc'r 0 EIOR addr to r3
# The following line is for the MPC555x:
# lis r3, INTC_EOIR@ha MPC555x: Load upper half of EIOR address to r3
li r4, 0        # Disable interrupts for rest of handler
# The following line is for MPC551x:
# stw r4, INTC_EOIR_PRC0@l(r3) MPC551x: Write 0 to proc'r 0 INTC_EOIR
# The following line is for MPC555x:
# stw r4, INTC_EOIR@l(r3) MPC555x: Write 0 to INTC_EOIR
lwz r3, 0x0C (r1)         # Restore SRR0
mtsrr0 r3
lwz r3, 0x10 (r1)         # Restore SRR1
mtsrr1 r3
lwz r4, 0x2C (r1)         # Restore working registers
lwz r3, 0x28 (r1)
addi r1, r1, 0x50         # Delete stack frame
rfi                          # End of Interrupt
### 4.3.3 IntcIsrVectors.c file

/* IntcIsrVectors.c - table of ISRs for INTC in SW vector Mode */
/* Description: Contains addresses for 310 ISRs vectors */
/* Table address gets loaded to INTC_IACKR */
/* Alignment: MPC551x: 2 KB after a 4KB boundary; MPC555x: 64 KB */
/* Copyright Freescale Semiconductor Inc 2007. All rights reserved. */
/* April 22, 2004 S. Mihalik */
/* March 16, 2006 S. Mihalik - Modified for compile with Diab 5.3 */
/* Jun 29 2006 SM - Used pragma align instead of hard coding address */
/* Jul 5 2007 SM - alignment now done in link file; changes for MPC551x */
/* Aug 30 2007 SM - Added pragma for CodeWarrior */

#include "mpc563m.h" /* Use proper include file like mpc5510.h or mpc5554.h */

void dummy (void);

extern void SwIrq4ISR(void);
extern void emiosCh0ISR(void);

/* Use pragma next two lines with CodeWarrior compile */
#pragma section data_type ".intc_sw_isr_vector_table" ".intc_sw_isr_vector_table" data_mode=far_abs
uint32_t IntcIsrVectorTable[] = {

/* Use next two lines with Diab compile */
/* #pragma section CONST ".intc_sw_isr_vector_table" */ /* Diab compiler pragma */
/* const uint32_t IntcIsrVectorTable[] = { */

(uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, /* ISRs 00 - 04 */
(uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, /* ISRs 05 - 09 */
(uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, /* ISRs 10 - 14 */
(uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, /* ISRs 15 - 19 */
(uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, /* ISRs 20 - 24 */
(uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, /* ISRs 25 - 29 */
(uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, /* ISRs 30 - 34 */
(uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, /* ISRs 35 - 39 */
(uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, /* ISRs 40 - 44 */
(uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, /* ISRs 45 - 49 */
/* Use next 2 lines for MPC551x: */
/* (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, */ /* ISRs 50 - 54 */
/* (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, */ /* ISRs 55 - 59 */
/* Use next 2 lines with Diab compile */
/* (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, */ /* ISRs 55 - 59 */

0x10000000, /* Used for alignment; MPC551x: 2 KB after a 4KB boundary; MPC555x: 64 KB */
0x20000000, /* Used for alignment; MPC551x: 2 KB after a 4KB boundary; MPC555x: 64 KB */
0x30000000, /* Used for alignment; MPC551x: 2 KB after a 4KB boundary; MPC555x: 64 KB */
0x40000000, /* Used for alignment; MPC551x: 2 KB after a 4KB boundary; MPC555x: 64 KB */
0x50000000, /* Used for alignment; MPC551x: 2 KB after a 4KB boundary; MPC555x: 64 KB */
0x60000000, /* Used for alignment; MPC551x: 2 KB after a 4KB boundary; MPC555x: 64 KB */
0x70000000, /* Used for alignment; MPC551x: 2 KB after a 4KB boundary; MPC555x: 64 KB */
0x80000000, /* Used for alignment; MPC551x: 2 KB after a 4KB boundary; MPC555x: 64 KB */
0x90000000, /* Used for alignment; MPC551x: 2 KB after a 4KB boundary; MPC555x: 64 KB */
0xA0000000, /* Used for alignment; MPC551x: 2 KB after a 4KB boundary; MPC555x: 64 KB */
0xB0000000, /* Used for alignment; MPC551x: 2 KB after a 4KB boundary; MPC555x: 64 KB */
0xC0000000, /* Used for alignment; MPC551x: 2 KB after a 4KB boundary; MPC555x: 64 KB */
0xD0000000, /* Used for alignment; MPC551x: 2 KB after a 4KB boundary; MPC555x: 64 KB */
0xE0000000, /* Used for alignment; MPC551x: 2 KB after a 4KB boundary; MPC555x: 64 KB */
0xF0000000, /* Used for alignment; MPC551x: 2 KB after a 4KB boundary; MPC555x: 64 KB */
/* Use next 2 lines for MPC551x: */
/* (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, */ /* ISRs 55 - 59 */
/* (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, */ /* ISRs 55 - 59 */

0x60000000, /* Used for alignment; MPC551x: 2 KB after a 4KB boundary; MPC555x: 64 KB */
0x50000000, /* Used for alignment; MPC551x: 2 KB after a 4KB boundary; MPC555x: 64 KB */
0x40000000, /* Used for alignment; MPC551x: 2 KB after a 4KB boundary; MPC555x: 64 KB */
0x30000000, /* Used for alignment; MPC551x: 2 KB after a 4KB boundary; MPC555x: 64 KB */
0x20000000, /* Used for alignment; MPC551x: 2 KB after a 4KB boundary; MPC555x: 64 KB */
0x10000000, /* Used for alignment; MPC551x: 2 KB after a 4KB boundary; MPC555x: 64 KB */
0x00000000, /* Used for alignment; MPC551x: 2 KB after a 4KB boundary; MPC555x: 64 KB */
(uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, /* ISRs 240 - 244 */
(uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, /* ISRs 245 - 249 */
(uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, /* ISRs 250 - 254 */
(uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, /* ISRs 255 - 259 */
(uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, /* ISRs 260 - 264 */
(uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, /* ISRs 265 - 269 */
(uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, /* ISRs 270 - 274 */
(uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, /* ISRs 275 - 279 */
(uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, /* ISRs 280 - 284 */
(uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, /* ISRs 285 - 289 */
(uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, /* ISRs 290 - 294 */
(uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, /* ISRs 295 - 299 */
(uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, /* ISRs 300 - 304 */
(uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, (uint32_t)&dummy, /* ISRs 305 - 309 */
);

void dummy (void) {
    while (1) {}; /* Wait forever or for watchdog timeout */
}
4.3.4  **ivor_branch_table.s file (MPC551x only)**

```
# iver_branch_table.s - for use with MPC551x only
# Description: Branch table for 16 MPC551x core interrupts
# Copyright Freescale 2007. All Rights Reserved
# Rev 1.0 Jul 6 2007 S Mihalik - Initial version
# Rev 1.1 Aug 30 2007 SM - Made IVOR4Handler extern

.extern IVOR4Handler

.align SIXTEEN_BYTES

.section .ivor_branch_table

.equ SIXTEEN_BYTES, 16    # 16 byte alignment required for table entries
    # Diab compiler uses value of 4 (2**4=16)
    # CodeWarrior, GHS, Cygnus use 16

    .align SIXTEEN_BYTES
IVOR0trap:  b IVOR0trap  # IVOR 0 interrupt handler
    .align SIXTEEN_BYTES
IVOR1trap:  b IVOR1trap  # IVOR 1 interrupt handler
    .align SIXTEEN_BYTES
IVOR2trap:  b IVOR2trap  # IVOR 2 interrupt handler
    .align SIXTEEN_BYTES
IVOR3trap:  b IVOR3trap  # IVOR 3 interrupt handler
    .align SIXTEEN_BYTES
    b IVOR4Handler  # IVOR 4 interrupt handler (External Interrupt)
    .align SIXTEEN_BYTES
IVOR5trap:  b IVOR5trap  # IVOR 5 interrupt handler
    .align SIXTEEN_BYTES
IVOR6trap:  b IVOR6trap  # IVOR 6 interrupt handler
    .align SIXTEEN_BYTES
IVOR7trap:  b IVOR7trap  # IVOR 7 interrupt handler
    .align SIXTEEN_BYTES
IVOR8trap:  b IVOR8trap  # IVOR 8 interrupt handler
    .align SIXTEEN_BYTES
IVOR9trap:  b IVOR9trap  # IVOR 9 interrupt handler
    .align SIXTEEN_BYTES
IVOR10trap: b IVOR10trap # IVOR 10 interrupt handler
    .align SIXTEEN_BYTES
IVOR11trap: b IVOR11trap # IVOR 11 interrupt handler
    .align SIXTEEN_BYTES
IVOR12trap: b IVOR12trap # IVOR 12 interrupt handler
    .align SIXTEEN_BYTES
IVOR13trap: b IVOR13trap # IVOR 13 interrupt handler
    .align SIXTEEN_BYTES
IVOR14trap: b IVOR14trap # IVOR 14 interrupt handler
    .align SIXTEEN_BYTES
IVOR15trap: b IVOR15trap # IVOR15 interrupt handler
```
5 INTC: Hardware Vector Mode

5.1 Description

Task: Using the Interrupt Controller (INTC) in hardware vector mode, this program provides two interrupts that show nesting. The first interrupt is generated from an eMIOS channel at a 1 kHz rate using the MPC555x default system clock. For the MPC551x, the faster default system clock produces an eMIOS channel timeout rate of approximately $1 \text{ kHz} \times \frac{16}{12} = 1.33 \text{ kHz}$. The interrupt handler will re-enable interrupts and later, in its interrupt service routine (ISR), invoke a second interrupt every other time. This provides an approximate 1 millisecond task (ISR) from the eMIOS channel and an approximate 2 millisecond task (ISR) from the software interrupt. The software interrupt will have a higher priority, so the one eMIOS ISR is preempted by the software interrupt. Both ISRs will have a counter.

This example is identical to INTC: Software Vector Mode, except for the differences between software and hardware vector modes.

Note: eMIOS channel 0 interrupt vector numbers are different on MPC551x and MPC555x devices. Also, to generate the timed interrupt, the eMIOS channel mode will be the modulus counter for MPC555x devices that have that mode, or the newer modulus counter buffered mode.

The ISRs will be written in C, so the appropriate registers will be saved and restored in the handler. The SPE will not be used, so its accumulator will not be saved in the stack frame of the interrupt handler. $^1$

Exercise: Write a third interrupt handler that uses a software interrupt of a higher priority than the others. Invoke this new software interrupt from one of the other ISRs.

---

1. The SPE's accumulator is not saved in the prologue. It is assumed SPE instructions are not used in this example.
5.2 Design

The overall program flow is shown below, followed by design steps and a stack frame implementation. The INTC when used in hardware vector mode uses a branch table for getting to each INTC vector’s handler. These are shown as handler_0, handler_1, etc. (Note: the code in this example does not have handlers for each INTC vector, so a common dummy trap address is used.)

For MPC551x, the second core has its own special purpose register, IVPR, so the second core would have its own IntcHandlerBranchTable (not included in this example).

Also for MPC551x, either or both processors can receive the interrupt request from the Interrupt Controller. In this example only one processor is selected in the MPC551x, processor 0 (e200z1). The selection is defined in INTC_PSR for each enabled interrupt, which here is eMIOS channel 0 and software interrupt 4.

![Figure 12. Overall Program Flow](image)

**Figure 12. Overall Program Flow** (Shown for MPC555x which uses INTC vector 51 for eMIOS channel 0. MPC551x uses INTC vector 58 for eMIOS channel 0.)
### 5.2.1 Initialization

Table 17. Initialization: INTC in Hardware Vector Mode

<table>
<thead>
<tr>
<th>Step</th>
<th>Relevant Bit Fields</th>
<th>Pseudo Code</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Variable Initializations</strong></td>
<td>Counter for eMIOS channel 0 interrupts</td>
<td>int emiosCh0Ctr = 0</td>
</tr>
<tr>
<td></td>
<td>Counter for software 4 interrupts</td>
<td>int SWirq4Ctr = 0</td>
</tr>
<tr>
<td><strong>Table Initializations</strong></td>
<td>Load INTC HW Branch Table with ISR names for:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>INTC Vector 4 ISR (SW interrupt request 4)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>INTC Vector 51 ISR (eMIOS channel 0)</td>
<td></td>
</tr>
<tr>
<td><strong>initirqVectors</strong></td>
<td>Load the common interrupt vector prefix to spr IVPR. Value is defined in the link file.</td>
<td>spr IVPR = __IVPR_VALUE</td>
</tr>
<tr>
<td><strong>initINTC</strong></td>
<td>Initialize INTC:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Configure for hardware vector mode (MPC551x note: only proc'0, e200z1, is used here)</td>
<td>HVEN_PRC0 = 0 (551x)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HVEN = 0 (MPC555x)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INTC_MCR[HVEN_PRC0] = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INTC_MCR[HVEN] = 1</td>
</tr>
<tr>
<td><strong>initEMIOS</strong></td>
<td>Initialize EMIOS global settings for all channels:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Prescale sysclk by 12 for eMIOS clock</td>
<td>GPRE = 11 (0xB)</td>
</tr>
<tr>
<td></td>
<td>• Enable eMIOS clock</td>
<td>GPREN = 1</td>
</tr>
<tr>
<td></td>
<td>• Enable global time base</td>
<td>GTBE = 1</td>
</tr>
<tr>
<td></td>
<td>• Enable stopping channels in debug mode</td>
<td>FRZ = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>EMIOS_MCR = 0x3400 B000</td>
</tr>
<tr>
<td></td>
<td>MPC551x: Ensure channel is not disabled</td>
<td>EMIOS_UDPIS = 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>Channel 0 Interrupt:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Raise priority to 1</td>
<td>INTC_PSR[PRI] = 0x01</td>
</tr>
<tr>
<td></td>
<td>• MPC551x: Select processor(s) to interrupt</td>
<td>INTC_PSR[PRC_SEL] = 0 (e200z1)</td>
</tr>
<tr>
<td></td>
<td>Channel 0: set max count = 1,000 clks (~ 1 μs each)</td>
<td>A = 999</td>
</tr>
<tr>
<td></td>
<td></td>
<td>EMIOS_A[0] = 999</td>
</tr>
<tr>
<td></td>
<td>Channel 0: Enable channel as up counter &amp; enable IRQ</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Mode (MPC551x, 563x) = modulus up counter buffered</td>
<td>MODE = 0x50 or</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MODE = 0x10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BSL = 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>UCPRE = 0 (default)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>UCPREN = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FREN = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DMA = 0 (default)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FEN = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>EMIOS_CCR[0] = 0x8202 0650</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MPC55x:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>EMIOS_CCR[0] = 0x8202 0610</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MPC563x:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>EMIOS_CCR[0] = 0x8202 0650</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>initSwIrq4</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Raise priority to 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• MPC551x: Specify which processor(s) to interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INTC_PSR[4] = 0x02</td>
</tr>
<tr>
<td></td>
<td><strong>enableExtIrq</strong></td>
<td>Enable recognition of requests to INTC by lowering the INTC’s current priority to 0 from default of 15 (MPC551x note: only proc’0, e200z1, is used here)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>waitloop</strong></td>
<td>wait forever</td>
</tr>
</tbody>
</table>

Qorivva Simple Cookbook, Rev. 4
## 5.2.2 Interrupt Handlers

Table 18. Stack Frame for INTC Interrupt Handler  
(Same as for Software Vector Mode Example)

<table>
<thead>
<tr>
<th>Stack Frame Area</th>
<th>Register</th>
<th>Location in Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit GPRs</td>
<td>r12</td>
<td>sp + 0x4C</td>
</tr>
<tr>
<td></td>
<td>r11</td>
<td>sp + 0x48</td>
</tr>
<tr>
<td></td>
<td>r10</td>
<td>sp + 0x44</td>
</tr>
<tr>
<td></td>
<td>r9</td>
<td>sp + 0x40</td>
</tr>
<tr>
<td></td>
<td>r8</td>
<td>sp + 0x3C</td>
</tr>
<tr>
<td></td>
<td>r7</td>
<td>sp + 0x38</td>
</tr>
<tr>
<td></td>
<td>r6</td>
<td>sp + 0x34</td>
</tr>
<tr>
<td></td>
<td>r5</td>
<td>sp + 0x30</td>
</tr>
<tr>
<td></td>
<td>r4</td>
<td>sp + 0x2C</td>
</tr>
<tr>
<td></td>
<td>r3</td>
<td>sp + 0x28</td>
</tr>
<tr>
<td></td>
<td>r0</td>
<td>sp + 0x24</td>
</tr>
<tr>
<td>CR</td>
<td>CR</td>
<td>sp + 0x20</td>
</tr>
<tr>
<td>locals and padding</td>
<td>XER</td>
<td>sp + 0x1C</td>
</tr>
<tr>
<td></td>
<td>CTR</td>
<td>sp + 0x18</td>
</tr>
<tr>
<td></td>
<td>LR</td>
<td>sp + 0x14</td>
</tr>
<tr>
<td></td>
<td>SRR1</td>
<td>sp + 0x10</td>
</tr>
<tr>
<td></td>
<td>SRR0</td>
<td>sp + 0x0C</td>
</tr>
<tr>
<td></td>
<td>padding</td>
<td>sp + 0x08</td>
</tr>
<tr>
<td>LR area</td>
<td>LR placeholder</td>
<td>sp + 0x04</td>
</tr>
<tr>
<td>back chain</td>
<td>r1</td>
<td>sp</td>
</tr>
</tbody>
</table>
### Table 19. eMIOS Channel 0 Interrupt Handler (INTC in Hardware Vector Mode)

<table>
<thead>
<tr>
<th>Step</th>
<th>Relevant Bit Fields</th>
<th>Pseudo Code</th>
<th>MPC51x</th>
<th>MPC555x</th>
</tr>
</thead>
<tbody>
<tr>
<td>prolog</td>
<td>Create stack frame</td>
<td>stwu sp, − 0x50 (sp)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Save SRR0:1 because nested interrupts will be allowed</td>
<td>store SRR0:1, r3 registers to stack frame</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Re-enable external interrupts by setting MSR[EE]</td>
<td>EE = 1 wrteei 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Save other appropriate registers for C ISR</td>
<td>store other registers to stack frame</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Branch to ISR, saving return address</td>
<td>bl emiosCh0ISR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>emiosCh0ISR</td>
<td>Increment emiosCh0Ctr</td>
<td>emiosCh0Ctr++</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>If emiosCh0Ctr is even, invoke software Interrupt 4</td>
<td>if ((emiosCh0Ctr&amp;1), INTC_SSCIR[4] = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Clear eMIOS Ch 0 interrupt flag by writing 1 to it</td>
<td>FLAG = 1 EMIOS_CSR0[FLAG] = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>epilog</td>
<td>Restore registers from stack frame except SRR0:1 and two working registers</td>
<td>load most registers from stack frame</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Ensure interrupt flag in ISR has completed clearing before writing to INTC_EOIR</td>
<td>mbar</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Disable external interrupts by clearing MSR[EE]</td>
<td>EE = 0 wrteei 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Restore former INTC’s current priority</td>
<td>write 0 to INTC_EOIR_PRC0 write 0 to INTC_EOIR</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Restore SRR0:1 and working registers</td>
<td>restore SRR0:1, working registers from stack frame</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Restore stack frame space</td>
<td>addi sp, sp, 0x50</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Return</td>
<td>rfi</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 20. Software 4 Interrupt Handler (INTC in Hardware Vector Mode)

<table>
<thead>
<tr>
<th>Step</th>
<th>Relevant Bit Fields</th>
<th>Pseudo Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>prolog</td>
<td>Create stack frame</td>
<td>stw sp, – 0x50 (sp)</td>
</tr>
<tr>
<td></td>
<td>Save SRR0:1</td>
<td>store SRR0:1, r3 registers to stack frame</td>
</tr>
<tr>
<td></td>
<td>Re-enable external interrupts by setting MSR[EE]</td>
<td>EE = 1 wrteei 1</td>
</tr>
<tr>
<td></td>
<td>Save other appropriate registers for C ISR</td>
<td>store other registers to stack frame</td>
</tr>
<tr>
<td></td>
<td>Branch to ISR, saving return address</td>
<td>bl SWIrq4ISR</td>
</tr>
<tr>
<td>SWIrq4ISR</td>
<td>Increment SWIrq4Ctr</td>
<td>SWIrq4Ctr++</td>
</tr>
<tr>
<td></td>
<td>Clear software IRQ 4 interrupt flag by writing 1 to it</td>
<td>CLR = 1 INTC_SSCIR4 = 1</td>
</tr>
<tr>
<td>epilog</td>
<td>Restore registers from stack frame except SRR0:1 and two working registers</td>
<td>load most registers from stack frame</td>
</tr>
<tr>
<td></td>
<td>Ensure interrupt flag in ISR has completed clearing before writing to INTC_EOIR</td>
<td>mbar</td>
</tr>
<tr>
<td></td>
<td>Disable external interrupts by clearing MSR[EE]</td>
<td>EE = 0 wrteei 0</td>
</tr>
<tr>
<td></td>
<td>Restore former INTC’s current priority</td>
<td>write 0 to INTC_EOIR_PRC0 write 0 to INTC_EOIR</td>
</tr>
<tr>
<td></td>
<td>Restore SRR0:1 and working registers</td>
<td>restore SRR0:1, working registers from stack frame</td>
</tr>
<tr>
<td></td>
<td>Restore stack frame space</td>
<td>addi sp, sp, 0x50</td>
</tr>
<tr>
<td></td>
<td>Return</td>
<td>rfi</td>
</tr>
</tbody>
</table>
5.3  Code

5.3.1  main.c file

/* main.c - INTC in Hardware vector mode using C ISRs */
/* Copyright Freescale Semiconductor, Inc. 2007. All Rights Reserved */
/* Rev 0.1 Oct 1 2004 Steve Mihalik */
/* Rev 1.0 May 19, 2006 S.M.- renamed SWIrq7Ctr to SWirq4Ctr for consistency */
/* Rev 1.1 Aug 12 1006 SM - Made i volatile (to get fewer Nexus msgs in loop) */
/* Jul 17 2007 SM - Passed IVPR value from link file, used relevant names */
/* for MPC551x bit fields & registers, invoked SW interrupt */
/* on even count eMIOS Ch 0 ISRs, changed timeout to 1 msec */
/* and changed EMIOS_MCR[PRE] & EMIOS Chan 0 A Register values */
/* Notes: */
/* 1. MMU not initialized; must be done by debug scripts or BAM */
/* 2. SRAM not initialized; must be done by debug scripts or in a crt0 type file */
/* 3. Cache is not used */
#include "mpc563m.h" /* Use proper include file such as mpc5510.h or mpc5554.h */
extern uint32_t __IVPR_VALUE; /* Interrupt Vector Prefix value from link file */
int emiosCh0Ctr = 0; /* Counter for eMIOS channel 0 interrupts */
int SWirq4Ctr = 0;  /* Counter for software interrupt 4 */
asm void initIrqVectors(void) {
    lis r3, __IVPR_VALUE@h  /* IVPR value is passed from link file */
    ori r3, r3, __IVPR_VALUE@l  /* Note: IVPR lower bits are unused in MPC555x */
    mtivpr r3
}

void initINTC(void) {
    /* Use one of the next two lines: */
    /*INTC.MCR.B.HVEN_PRC0 = 1; */ /* MPC551x: Proc'r 0: initialize for HW vector mode */
    INTC.MCR.B.HVEN = 1;          /* MPC555x: initialize for HW vector mode */
}

void initEMIOS(void) {
    EMIOS.MCR.B.GPRE = 11;       /* Divide sysclk by (11+1) for eMIOS clock */
    EMIOS.MCR.B.GPRE = 1;        /* Enable eMIOS clock */
    EMIOS.MCR.B.GTBE = 1;        /* Enable global time base */
    EMIOS.MCR.B.FRZ = 1;         /* Enable stopping channels when in debug mode */
    /* Following for MPC551x only: */
    /*EMIOS.UCDIS.R = 0; */       /* MPC551x: Ensure all channels are enabled */
    /* Use one of the following two lines: */
    /*INTC.PSR[58].R = 1; */      /* MPC551x: Raise eMIOS chan 0 IRQ priority = 1 */
    INTC.PSR[51].R = 1;          /* MPC555x: Raise eMIOS chan 0 IRQ priority = 1 */
    EMIOS.CH[0].CADR.R = 999;    /* Period will be 999+1=1000 channel clocks */
    /* Use one of the following two lines for mode: */
    /*EMIOS.CH[0].CCR.B.MODE = 0x50; */ /* MPC551x or MPC563x: Mod. Counter Buffered (MCB) */
    /*EMIOS.CH[0].CCR.B.MODE = 0x10;*/ /* MPC555x: Modulus Counter (MC) */
    EMIOS.CH[0].CCR.B.BSL = 0x3; /* Use internal counter */
    EMIOS.CH[0].CCR.B.UCPREN = 1; /* Enable prescaler; uses default divide by 1 */
    EMIOS.CH[0].CCR.B.FREN = 1;  /* Freeze channel registers when in debug mode */
    EMIOS.CH[0].CCR.B.FEN=1;    /* Flag enables interrupt */
}

void initSwIrq4(void) {
    INTC.PSR[4].R = 2;  /* Software interrupt 4 IRQ priority = 2 */
}

void enableIrq(void) {
    /* Use one of the following two lines: */
    /*INTC.CPR_PRC0.B.PRI = 0; */ /* MPC551x Proc'r 0: Lower INTC's current priority */
    INTC.CPR.B.PRI = 0;        /* MPC555x: Lower INTC's current priority */
    asm("wrteei 1");      /* Enable external interrupts */
}
void main (void) {
    vuint32_t i = 0;    /* Dummy idle counter */
    initIrqVectors();   /* Initialize exceptions: only need to load IVPR */
    initINTC();         /* Initialize INTC for hardware vector mode */
    initEMIOS();        /* Initialize eMIOS channel 0 for 1kHz IRQ, priority 2 */
    initSwIrq4();       /* Initialize software interrupt 4 */
    enableIrq();        /* Ensure INTC current priority=0 & enable IRQ */

    while (1) {
        i++;
    }
}

void emiosCh0ISR(void) {
    emiosCh0Ctr++;  /* Increment interrupt counter */
    if (((emiosCh0Ctr & 1)==0) { /* If emiosCh0Ctr is even*/
        INTC.SSCIR[4].R = 2;  /* then invoke software interrupt 4 */
    }
    EMIOS.CH[0].CSR.B.FLAG=1; /* Clear channel's flag */
}

void SwIrq4ISR(void) {
    SWirq4Ctr++;  /* Increment interrupt counter */
    INTC.SSCIR[4].R = 1; /* Clear channel's flag */
}
5.3.2 handlers.s file

# handlers.s - INTC hardware vector mode example
# Description: Creates prolog, epilog for C ISR and enables nested interrupts
# Rev 1.0 Jan 5, 2004 S Mihalik
# Rev 1.1 Aug 2, 2004 SM - delayed writing to EOIR until after EE is disabled
# Rev 1.2 Jul 18, 2005 SM - .org, .section & .extern changes for CodeWarrior 1.5
# Rev 1.3 Jul 6, 2007 SM - Moved isr_hw_brachn table to new file, added new 551x EOIR
# Rev 1.4 Aug 30 2007 SM - Added .text directive
# Copyright Freescale Semiconductor, Inc. 2007. All rights reserved

# STACK FRAME DESIGN: Depth: 20 words (0xA0, or 80 bytes)
# ****************************
# 0x4C     * GPR12    *    ^
# 0x48     * GPR11    *    |
# 0x44     * GPR10    *    |
# 0x40     * GPR9     *    |
# 0x3C     * GPR8     *    |
# 0x38     * GPR7     * GPRs (32 bit)
# 0x34     * GPR6     *    |
# 0x30     * GPR5     *    |
# 0x2C     * GPR4     *    |
# 0x28     * GPR3     *    |
# 0x24     * GPR0     *    |
# 0x20     * CR       * CR
# 0x1C     * XER      *    ^
# 0x18     * CTR      *    |
# 0x14     * LR       * locals & padding for 16 B alignment
# 0x10     * SRR1     *    |
# 0x0C     * SRR0     *    |
# 0x08     * padding   *    |
# 0x04     * resvd- LR * Reserved for Calling function
# 0x00     * SP       * Backchain (same as gpr1 in GPRs)
# ****************************

.equ INTC_EOIR,0xfff48018  # MPC555x: End Of Interrupt Reg. addr.
.equ INTC_EOIR_PRC0, 0xfff48018 # MPC551x: Proc'r 0 End Of Interrupt Reg. addr.

.extern emiosCh0ISR
.extern SwIrq4ISR
.globl emiosCh0Handler
.globl SwIrq4Handler

.text
emiosCh0Handler:

```asm
# PROLOGUE
stwu r1, -0x50 (r1)   # Create stack frame and store back chain
stw r3, 0x28 (r1)    # Store a working register
mferr0 r3            # Store SRR0:1 (must be done before enabling EE)
stw r3, 0x0C (r1)
mferr1 r3
stw r3, 0x10 (r1)
wrteei 1             # Set MSR[EE]=1
stw r12, 0x4C (r1)   # Store rest of gprs
stw r11, 0x48 (r1)
stw r10, 0x44 (r1)
stw r9, 0x40 (r1)
stw r8, 0x3C (r1)
stw r7, 0x38 (r1)
stw r6, 0x34 (r1)
stw r5, 0x30 (r1)
stw r4, 0x2C (r1)
stw r0, 0x24 (r1)
mfcr r3               # Store CR
stw r3, 0x20 (r1)    # Store XER
stw r3, 0x1C (r1)
mfctr r3             # Store CTR
stw r3, 0x18 (r1)
mflr r3              # Store LR
stw r3, 0x14 (r1)
bl emiosCh0ISR

# EPILOGUE
lwz r3, 0x14 (r1)    # Restore LR
mtlr r3
lwz r3, 0x18 (r1)    # Restore CTR
mtctr r3
lwz r3, 0x1C (r1)    # Restore XER
mtxer r3
lwz r3, 0x20 (r1)    # Restore CR
mtcrf 0xff, r3
lwz r0, 0x24 (r1)    # Restore other gprs except working registers
lwz r5, 0x30 (r1)
lwz r6, 0x34 (r1)
lwz r7, 0x38 (r1)
lwz r8, 0x3C (r1)
lwz r9, 0x40 (r1)
lwz r10, 0x44 (r1)
lwz r11, 0x48 (r1)
lwz r12, 0x4C (r1)
mbar 0               # Ensure store to clear flag bit has completed
```

# The following line is for the MPC551x:
# lis r3, INTC_EOIR_PRC0@ha# MPC551x: Load upper half of proc'r 0 EIOR addr to r3
# The following line is for the MPC555x:
# lis r3, INTC_EOIR@ha# MPC555x: Load upper half of EIOR address to r3
# li r4, 0
# wrteei 0        # Disable interrupts for rest of handler
# The following line is for MPC551x:
# stw r4, INTC_EOIR_PRC0@l(r3)# MPC551x: Write 0 to proc'r 0 INTC_EOIR
# The following line is for MPC555x:
stw r4, INTC_EOIR@l(r3)  # MPC555x: Write 0 to INTC_EOIR
lwz r3, 0x0C (r1)     # Restore SRR0
mtsrr0 r3
lwz r3, 0x10 (r1)    # Restore SRR1
mtsrr1 r3
lwz r4, 0x2C (r1)    # Restore working registers
lwz r3, 0x28 (r1)
addi r1, r1, 0x50    # Delete stack frame
rfi               # End of Interrupt
```
SwIrq4Handler:

STWU R1, -0x50 (R1)   # Create stack frame and store back chain
STW R3, 0x28 (R1)   # Store a working register
MFSRR0 R3  # Store SRR0:1 (must be done before enabling EE)
STW R3, 0x0C (R1)
MFSRR1 R3
STW R3, 0x10 (R1)

WRTEEI 1  # Set MSR[EE]=1

STW R12, 0x4C (R1)    # Store rest of gprs
STW R11, 0x48 (R1)
STW R10, 0x44 (R1)
STW R9, 0x40 (R1)
STW R8, 0x3C (R1)
STW R7, 0x38 (R1)
STW R6, 0x34 (R1)
STW R5, 0x30 (R1)
STW R4, 0x2C (R1)
STW R0, 0x24 (R1)
MFCR R3  # Store CR
STW R3, 0x20 (R1)  # Store XER
STW R3, 0x1C (R1)  # Store CTR
STW R3, 0x18 (R1)
MFLR R3  # Store LR
STW R3, 0x14 (R1)
BL SwIrq4ISR  # EPILOGUE

LWZ R3, 0x14 (R1)    # Restore LR
MTLIR R3  # Restore CR
LWZ R3, 0x18 (R1)  # Restore CTR
MTCTR R3  # Restore XER
MTXER R3
LWZ R3, 0x20 (R1)  # Restore CR
MTCRF 0xff, R3
LWZ R0, 0x24 (R1)   # Restore other gprs except working registers
LWZ R5, 0x30 (R1)
LWZ R6, 0x34 (R1)
LWZ R7, 0x38 (R1)
LWZ R8, 0x3C (R1)
LWZ R9, 0x40 (R1)
LWZ R10, 0x44 (R1)
LWZ R11, 0x48 (R1)
LWZ R12, 0x4C (R1)
MBAR 0  # Ensure store to clear flag bit has completed

# The following line is for the MPC551x:
# LIS R3, INTC_EOIR_PRC0@ha  MPC551x: Load upper half of proc' r 0 EIOR addr to r3
# The following line is for the MPC555x:
# LIS R3, INTC_EOIR@ha  MPC555x: Load upper half of EIOR address to r3
# LI R4, 0
# WRTEEI 0  # Disable interrupts for rest of handler
# The following line is for MPC551x:
# BL SwIrq4ISR  MPC551x: Write 0 to proc' r 0 INTC_EOIR
# The following line is for MPC555x:
STW R4, INTC_EOIR_PRC0@l(r3)  # MPC555x: Write 0 to INTC_EOIR
LWZ R3, 0x0C (R1)  # Restore SRR0
MTSRR0 R3
LWZ R3, 0x10 (R1)  # Restore SRR1
MTSRR1 R3
LWZ R4, 0x2C (R1)  # Restore working registers
LWZ R3, 0x28 (R1)
ADDI R1, R1, 0x50  # Delete stack frame
RFI  # End of Interrupt
5.3.3 intc_hw_branch_table.s (partial) file (MPC555x shown)

# intc_hw_branch_table.s - INTC hardware vector mode branch table example
# Description: INTC vector branch table when using INTC in HW vector mode
# **** NOTE **** ONLY 100 EXAMPLE VECTORS ARE IMPLEMENTED HERE
# Rev 1.0 Jul 2, 2007 S Mihalik
# Rev 1.1 Aug 30 1007 SM - Made SwIrq4Handler, emiosCh0Handler .extern
# Copyright Freescale Semiconductor, Inc. 2007. All rights reserved

.section .intc_hw_branch_table
.extern SwIrq4Handler
.extern emiosCh0Handler

# .equ ALIGN_OFFSET, 4   # MPC551x: 4 byte branch alignments (Diab, GHS use 2, CodeWarrior 4)
# .equ ALIGN_OFFSET, 16  # MPC555x: 16 byte branch alignments (Diab, GHS use 4, CodeWarrior 16)

IntcHandlerBranchTable: # Only 100 example vectors are implemented here
    # MPC555x: This table must have 64 KB alignment
    # MPC551x: Requires 2 KB alignment after 4KB boundary

.align ALIGN_OFFSET
hw_vect0:   b   hw_vect0        #INTC HW vector 0
.align ALIGN_OFFSET
hw_vect1:   b   hw_vect1        #INTC HW vector 1
.align ALIGN_OFFSET
hw_vect2:   b   hw_vect2        #INTC HW vector 2
.align ALIGN_OFFSET
hw_vect3:   b   hw_vect3        #INTC HW vector 3
.align ALIGN_OFFSET
hw_vect4:   b   SwIrq4Handler   # SW IRQ 4
.align ALIGN_OFFSET
hw_vect5:   b   hw_vect5        #INTC HW vector 5

... etc. for other vectors
hw_vect50:  b   hw_vect50       #INTC HW vector 50
.align ALIGN_OFFSET

# Use 1 of the next 2 lines
hw_vect51: b   emiosCh0Handler  #MPC555x: eMIOS Ch 0
#hw_vect51: b   hw_vect51       #INTC HW vector 51
.align ALIGN_OFFSET
hw_vect52: b   hw_vect52       #INTC HW vector 52
.align ALIGN_OFFSET
hw_vect53: b   hw_vect53       #INTC HW vector 53
.align ALIGN_OFFSET
hw_vect54: b   hw_vect54       #INTC HW vector 54
.align ALIGN_OFFSET
hw_vect55: b   hw_vect55       #INTC HW vector 55
.align ALIGN_OFFSET
hw_vect56: b   hw_vect56       #INTC HW vector 56
.align ALIGN_OFFSET
hw_vect57: b   hw_vect57       #INTC HW vector 57
.align ALIGN_OFFSET

# Use 1 of the next 2 lines
#hw_vect58: b   emiosCh0Handler #MPC551x: eMIOS Ch 0
hw_vect58: b   hw_vect58       #INTC HW vector 58
.align ALIGN_OFFSET
hw_vect59: b   hw_vect59       #INTC HW vector 59
.align ALIGN_OFFSET

... etc. for other vectors
5.3.4  ivor_branch_table.s file (MPC551x only)

# ivor_branch_table.s - for use with MPC551x only
# Description:  Branch table for 16 MPC551x core interrupts
# Copyright Freescale 2007.  All Rights Reserved
# Rev 1.0 Jul 6 2007 S Mihalik - Initial version

.section .ivor_branch_table

.equ SIXTEEN_BYTES, 16    # 16 byte alignment required for table entries
# Diab compiler uses value of 4 (2**4=16)
# CodeWarrior, GHS, Cygnus use 16

.align SIXTEEN_BYTES
IVOR0trap:  b IVOR0trap # IVOR 0 interrupt handler
    .align SIXTEEN_BYTES
IVOR1trap:  b IVOR1trap # IVOR 1 interrupt handler
    .align SIXTEEN_BYTES
IVOR2trap:  b IVOR2trap # IVOR 2 interrupt handler
    .align SIXTEEN_BYTES
IVOR3trap:  b IVOR3trap # IVOR 3 interrupt handler
    .align SIXTEEN_BYTES
IVOR4trap:  b IVOR4trap # IVOR 4 interrupt handler
    .align SIXTEEN_BYTES
IVOR5trap:  b IVOR5trap # IVOR 5 interrupt handler
    .align SIXTEEN_BYTES
IVOR6trap:  b IVOR6trap # IVOR 6 interrupt handler
    .align SIXTEEN_BYTES
IVOR7trap:  b IVOR7trap # IVOR 7 interrupt handler
    .align SIXTEEN_BYTES
IVOR8trap:  b IVOR8trap # IVOR 8 interrupt handler
    .align SIXTEEN_BYTES
IVOR9trap:  b IVOR9trap # IVOR 9 interrupt handler
    .align SIXTEEN_BYTES
IVOR10trap: b IVOR10trap # IVOR10 interrupt handler
    .align SIXTEEN_BYTES
IVOR11trap: b IVOR11trap # IVOR11 interrupt handler
    .align SIXTEEN_BYTES
IVOR12trap: b IVOR12trap # IVOR12 interrupt handler
    .align SIXTEEN_BYTES
IVOR13trap: b IVOR13trap # IVOR13 interrupt handler
    .align SIXTEEN_BYTES
IVOR14trap: b IVOR14trap # IVOR14 interrupt handler
    .align SIXTEEN_BYTES
IVOR15trap: b IVOR15trap # IVOR15 interrupt handler

Qorivva Simple Cookbook, Rev. 4

Freescale Semiconductor 47
6 INTC: Software Vector Mode, VLE Instructions

6.1 Description

Task: Using the Interrupt Controller (INTC) in software vector mode, this program provides two interrupts that show nesting. The main differences from the other INTC SW Mode, Classic Instructions example, are:

- VLE instructions are used (not available in MPC5553, MPC5554)
- Prologue/epilogue uses VLE assembly instructions
- Programmable Interrupt Timer (PIT) is used as the interrupt timer (note: PIT is not available in some MPC55xx devices, but an eMIOS channel could be used)
- The timer will count at the system clock rate (causing an interrupt at a count value of 0)
- System clock is set to 64 MHz

A relative interrupt response can be measured by reading the PIT count value in the first line of the interrupt service routine. An alternate method is to put a breakpoint in the beginning of the service routine and to read the count register. NOTE: to get a true interrupt performance measurement, additional software is needed to initialize branch target buffers, configure flash, and enable cache (if implemented), as shown in other examples in this application note.

The interrupt handler will re-enable interrupts and later, in its interrupt service routine (ISR), invoke a second interrupt every other time. This will provide an approximate 1 ms task (ISR) from the PIT and an approximate 2 ms task (ISR) from the software interrupt. The software interrupt will have a higher priority, so the 1 PIT ISR is preempted by the software interrupt. Both ISRs will have a counter.

The ISRs will be written in C, so the appropriate registers will be saved and restored in the handler. The SPE will not be used, so its accumulator will not be saved in the stack frame of the interrupt handler.

Exercise: Write a third interrupt handler which uses a software interrupt of a higher priority than the others. Invoke this new software interrupt from one of the other ISRs.

![Figure 13. Software Vector Mode, VLE Instructions Example](image-url)
6.2 Design

The overall program flow for MPC551x and MPC56xxB/P/S is shown below. When an interrupt occurs, it is routed to either or both cores, as defined in the INTC_PS for that vector.

For MPC551x, in this example only one processor is selected for interrupts: processor 0 (e200z1). The selection is defined in INTC_PS for each enabled interrupt — in this case PIT 1 and software interrupt 4.

For MPC551x, there can be a different ISR vector table for each core because each core has its own special purpose register, IVPR. The Vector Table Base Address is defined in each core’s INTC_ACKR, that is, either INTC_ACK_PRC0 (for e200z1) or INTC_IACK_PRC1 (for e200z0). This example only uses the e200z1 core.

![Diagram of program flow](image)

**Figure 14. MPC551x, MPC56xxPBS Program Flow**
The overall program flow for MPC555x is shown below.

**Figure 15. MPC555x Program Flow**

External Input interrupt taken

handlers_vle.s or handlers_new_vle.s file

vector to IVPR[0:15] + IVOR[16:27]

main.c file

IntcIsrVectors.c file

ISR Vector Table:

... SW IRQ 4 ISR addr
... PIT 1 ISR addr
...
...

IVOR4Handler:
prologue:
  save SRR0:1,
  fetch ISR vector,
  re-enable interrupts
  save registers
  branch to fetched ISR vector
epilogue (return from ISR):
  restore most registers
  ensure prior stores completed
  disable interrupts
  write to EOIR to restore priority
  restore SRR0:1
  return

main {
  init IVPR, IVOR4
  init INTC
  init PIT1
  init SW 4 interrupt
  enable interrupts
  wait forever
}

Pit1ISR {
  increment counter
  if odd count, invoke SW 4 IRQ
  clear flag
}

SwIRQ4ISR {
  increment counter
  clear flag
}
6.2.1 Modes Use Summary (MPC56xxB/P/S only)

Mode Transition is required for changing mode entry registers. Hence even enabling the crystal oscillator to be active in the default mode (DRUN) requires enabling the crystal oscillator in appropriate mode configuration register (ME_xxxx_MC) then initiating a mode transition. After reset, the mode is switched in this example from the default mode (DRUN) to RUN0 mode. The following table summarizes the mode settings used.

Table 21. Mode Configurations for MPC56xxB/P/S INTC SW Mode, VLE Instructions Example

<table>
<thead>
<tr>
<th>Mode</th>
<th>Mode Config. Register</th>
<th>Mode Config. Register Value</th>
<th>Settings</th>
<th>Clock Sources</th>
<th>Memory Power Mode</th>
<th>Main Voltage Reg.</th>
<th>I/O Power Down Ctrl</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>16MHz IRC</td>
<td>On</td>
<td>Normal</td>
<td>On</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Off</td>
<td>Off</td>
<td>Normal</td>
<td>Off</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Off</td>
<td>Off</td>
<td>Normal</td>
<td>Off</td>
</tr>
<tr>
<td>DRUN</td>
<td>ME_DRUN_MC</td>
<td>0x001F0010 (default)</td>
<td></td>
<td>On</td>
<td>Normal</td>
<td>Normal</td>
<td>On</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Off</td>
<td>Normal</td>
<td>Normal</td>
<td>Off</td>
</tr>
<tr>
<td>RUN0</td>
<td>ME_RUN0_MC</td>
<td>0x001F007D</td>
<td></td>
<td>On</td>
<td>Normal</td>
<td>Normal</td>
<td>On</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>On</td>
<td>Normal</td>
<td>Normal</td>
<td>Off</td>
</tr>
</tbody>
</table>
| Other modes are not used in example

Peripherals also have configurations to gate clocks on and off, enabling low power. The following table summarizes the peripheral configurations used in this example.

Table 22. Peripheral Configurations for MPC56xxB/P/S INTC SW Vector Mode, VLE Example

<table>
<thead>
<tr>
<th>Peripheral Config.</th>
<th>Peripheral Config. Register</th>
<th>Enabled Modes</th>
<th>Peripherals Selecting Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>RUN3 RUN2 RUN1 RUN0 DRUN SAFE TEST RESET Peripheral PCTL Reg. #</td>
<td></td>
</tr>
<tr>
<td>PC1</td>
<td>ME_RUNPC_1</td>
<td>0 0 1 0 0 0 0 0 0 PIT, RTI</td>
<td>92</td>
</tr>
</tbody>
</table>
| Other peripheral configurations are not used in example
### Table 23. Initialization: INTC in Software Vector Mode, VLE Instructions

<table>
<thead>
<tr>
<th>Step</th>
<th>Relevant Bit Fields</th>
<th>Pseudo Code</th>
</tr>
</thead>
</table>
| **Variable Init** | Counter for PIT 1 interrupts  
Counter for software 4 interrupts | int Pit1Ctr = 0  
int SWIRQ4Ctr = 0 |
| **Table Init** | Load INTC's ISR vector table with addresses for:  
- SWIRQ4ISR: INTC vector #4  
- Pit1ISR: INTC vector: MPC551x: #149,  
MPC563x: #302,  
MPC56xxB/P/S: #60 | &SwIrq4ISR  
&Pit1ISR |
| init Modes And Clock (MPC56xxBPS only) | Enable desired modes | DRUN=1, RUN0 = 1  
| Initialize PLL0 dividers to provide 64 MHz for input crystal before mode configuration:  
- 8 MHz xtal: FMPLL[0]_CR=0x02400100  
- 40 MHz xtal: FMPLL[0]_CR=0x12400100  
(MPC56xxP & 8 MHz xtal requires changing default CMU_CSR value. See PLL example.) | -  
8 MHz Crystal:  
CGM_  
FMPLL[0]_CR = 0x02400100 |
| Configure RUN0 Mode:  
- I/O Output power-down: no safe gating (default)  
- Main Voltage regulator is on (default)  
- Data, code flash in normal mode (default)  
- PLL0 is switched on  
- Crystal oscillator is switched on  
- 16 MHz IRC is switched on (default)  
- Select PLL0 (system pll) as sysclk | PDO=0  
MVRON=1  
DFLAON, CFLAON=3  
PLL0ON=1  
XOSCOON=1  
16MHz_IRCON=1  
SYSCLK=0x4 | -  
ME_RUN_MC = 0x001F 0070 |
| MPC56xxB/S:  
- Peri. Config.1: run in DRUN mode only | RUN0 = 1 | -  
ME_RUN_PC1 = 0000 0010 |
| Assign peripheral configuration to peripherals:  
- PIT, RTI: select ME_RUN_PC1 | RUN_CFG = 0 | -  
ME_PCTL92 = 0x01 |
| Initiate software mode transition to RUN0 mode  
- Mode & key, then mode & inverted key  
- Wait for transition to complete  
- Verify current mode is RUN0 | TARGET_MODE = RUN0  
S_TRANS  
CURRENTMODE | -  
(See example PLL: Initializing System Clock) |
| init Sysclk | Initialize sysclk to 64 MHz, running from PLL | (See example PLL: Initializing System Clock) |
| disable Watchdog | Disable watchdog by writing keys to Status Register, then clearing WEN (MPC56xxBPS only) | -  
See PLL Initialization example |
Table 23. Initialization: INTC in Software Vector Mode, VLE Instructions (continued)

<table>
<thead>
<tr>
<th>Step</th>
<th>Relevant Bit Fields</th>
<th>Pseudo Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>init Irq Vectors</td>
<td>Load the common interrupt vector prefix to spr IVPR.</td>
<td>spr IVPR = __IVPR_VALUE</td>
</tr>
<tr>
<td>MPC555x: Initialize external input interrupt offset to the lower half of IVROR4 handler's address</td>
<td>-</td>
<td>spr IVOR4 = IVOR4Handler @l</td>
</tr>
<tr>
<td>init INTC</td>
<td>Initialize INTC:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Configure for software vector mode (HVEN=0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Keep vector offset (VTES) at 4 bytes (MPC551x note: only proc'0, e200z1 is used)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HVEN_PRC0=0(551x)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VTES_PRC0=0(551x)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HVEN = 0 (MPC555x)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VTES = 0 (MPC555x)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>INTC_MCR = 0x00000001</td>
<td></td>
</tr>
<tr>
<td>Initialize ISR vector table base address per link file (MPC551x note: only proc'0, e200z1, is used here)</td>
<td>VTBA = passed from linker</td>
<td>INTC_IACKR_PRC0 = __IACKR_VALUE</td>
</tr>
<tr>
<td>init PIT</td>
<td>MPC551x: Init system clock divider for PIT, RTI to divide by 1 (also applies to other Group 1 peripherals of eSCI A, IIC)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LPCLKDIV1 = 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(default)</td>
<td></td>
</tr>
<tr>
<td>Global controls:</td>
<td>MDIS = 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FRZ = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PIT_CTRL = 0x00000000</td>
<td>PIT_PITMCRR = 0x00000001</td>
</tr>
<tr>
<td>Load a start count values for 64 MHz sysclk (PITs count down from value at sysclk rate)</td>
<td>PIT_TVVAL1 = 64,000</td>
<td>PIT_TIMER1_LDVAL1 = 64,000</td>
</tr>
<tr>
<td></td>
<td>• PIT 1 Timeout = 64M/(64MHz sysclk/sec)=1ms</td>
<td></td>
</tr>
<tr>
<td>Enable PIT 1 timer to count (counts down)</td>
<td>PEN1(551x) = 1 or TEN(555x,56xxBPS) = 1</td>
<td></td>
</tr>
<tr>
<td>Enable PIT 1 to request IRQ</td>
<td>TIE1 = 1</td>
<td>PIT_PITEN = 0x00000002</td>
</tr>
<tr>
<td>MPC551x: Assign PIT 1 flag for IRQ instead of DMA</td>
<td>ISEL = 1</td>
<td>PIT_INTEN = 0x00000002</td>
</tr>
<tr>
<td>Configure PIT1 interrupts:</td>
<td>PRC_SEL = 0 (z1)</td>
<td>INTC_PSRR = 0x0000001</td>
</tr>
<tr>
<td></td>
<td>• MPC551x: Select processor 0 (e200z1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Select priority 1</td>
<td></td>
</tr>
<tr>
<td>init SwIrq4</td>
<td>Software Interrupt 4:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Raise priority to 2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• MPC551x: Select processor(s) to interrupt</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PRI = 2</td>
<td>INTC_PSRR[4] = 0x0002</td>
</tr>
<tr>
<td>enable ExIrq</td>
<td>Enable recognition of requests to INTC by lowering the INTC's current priority to 0 from default of 15</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PRI = 0</td>
<td>INTC_CPR[PRI] = 0</td>
</tr>
<tr>
<td>waitloop</td>
<td>wait forever</td>
<td></td>
</tr>
</tbody>
</table>
## 6.2.2 Interrupt Handler

### Table 24. Stack Frame for INTC Interrupt Handler

<table>
<thead>
<tr>
<th>Stack Frame Area</th>
<th>Register</th>
<th>Location in Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 bit GPRs</td>
<td>r12</td>
<td>sp + 0x4C</td>
</tr>
<tr>
<td></td>
<td>r11</td>
<td>sp + 0x48</td>
</tr>
<tr>
<td></td>
<td>r10</td>
<td>sp + 0x44</td>
</tr>
<tr>
<td></td>
<td>r9</td>
<td>sp + 0x40</td>
</tr>
<tr>
<td></td>
<td>r8</td>
<td>sp + 0x3C</td>
</tr>
<tr>
<td></td>
<td>r7</td>
<td>sp + 0x38</td>
</tr>
<tr>
<td></td>
<td>r6</td>
<td>sp + 0x34</td>
</tr>
<tr>
<td></td>
<td>r5</td>
<td>sp + 0x30</td>
</tr>
<tr>
<td></td>
<td>r4</td>
<td>sp + 0x2C</td>
</tr>
<tr>
<td></td>
<td>r3</td>
<td>sp + 0x28</td>
</tr>
<tr>
<td></td>
<td>r0</td>
<td>sp + 0x24</td>
</tr>
<tr>
<td>CR</td>
<td>CR</td>
<td>sp + 0x20</td>
</tr>
<tr>
<td>locals and padding</td>
<td>XER</td>
<td>sp + 0x1C</td>
</tr>
<tr>
<td></td>
<td>CTR</td>
<td>sp + 0x18</td>
</tr>
<tr>
<td></td>
<td>LR</td>
<td>sp + 0x14</td>
</tr>
<tr>
<td></td>
<td>SRR1</td>
<td>sp + 0x10</td>
</tr>
<tr>
<td></td>
<td>SRR0</td>
<td>sp + 0x0C</td>
</tr>
<tr>
<td></td>
<td>padding</td>
<td>sp + 0x08</td>
</tr>
<tr>
<td>LR area</td>
<td>LR placeholder</td>
<td>sp + 0x04</td>
</tr>
<tr>
<td>back chain</td>
<td>r1</td>
<td>sp</td>
</tr>
</tbody>
</table>
Table 25. IVOR4 Interrupt Handler (INTC in Software Vector Mode, VLE Instructions)
(MPC555x: Must be 16-byte aligned and within 64 KB of address in IVPR.
MPC551x does not require alignment because it is the destination of a branch instruction.)

<table>
<thead>
<tr>
<th>Step</th>
<th>Relevant Bit Fields</th>
<th>Pseudo Code</th>
<th>MPC551x</th>
<th>MPC555x</th>
<th>MPC56xxBPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>prolog</td>
<td>Create stack frame</td>
<td>e_stwu sp, -0x50 (sp)</td>
<td>store SRR0:1, r3 registers to stack frame</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Save SRR0:1 because nested interrupts will be allowed</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Read pointer into ISR Vector Table</td>
<td>r3 = INTC_IACKR_PRC0</td>
<td>r3 = INTC_IACKR</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Re-enable external interrupts by setting MSR[EE]</td>
<td>EE = 1</td>
<td>wrteei 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Read ISR address from ISR Vector Table and store into LR</td>
<td>se_lwz r3, 0x0 (r3)</td>
<td>mtlr r3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Save other appropriate registers for C ISR</td>
<td></td>
<td>store other registers to stack frame</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Branch to ISR, saving return address</td>
<td>se_blrl</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>epilog</td>
<td>Restore registers from stack frame except SRR0:1 and two working registers</td>
<td>load most registers from stack frame</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Ensure interrupt flag in ISR has completed clearing before writing to INTC_EOIR</td>
<td>mbar</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Disable external interrupts by clearing MSR[EE]</td>
<td>EE = 0</td>
<td>wrteei 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Restore former INTC’s Current Priority</td>
<td>write 0 to INTC_EOIR_PRC0</td>
<td>write 0 to INTC_EOIR</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Restore SRR0:1 and working registers</td>
<td></td>
<td>restore SRR0:1, working registers from stack frame</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Restore stack frame space</td>
<td>e_add16i sp, sp, 0x50</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Return</td>
<td>se_rfi</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### 6.2.3 Interrupt Service Routines (ISRs)

#### Table 26. ISR for PIT1

<table>
<thead>
<tr>
<th>Step</th>
<th>Relevant Bit Fields</th>
<th>Pseudo Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pit1ISR</td>
<td>Increment Pit1Ctr</td>
<td>Pit1Ctr ++</td>
</tr>
<tr>
<td>If Pit1Ctr is even, invoke Software Interrupt 4</td>
<td>SET=1</td>
<td>if ((Pit1Ctr&amp;1) ==0), INTC_SSCIR[4] = 2</td>
</tr>
<tr>
<td>Clear Pit1Ctr interrupt flag by writing 1 to it</td>
<td>TIF1 = 1 or TIF = 1</td>
<td>PIT_PITFLG[TIF1] = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PIT_TFLG1[TIF] = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PIT_CH1_TFLG[TIF] = 1</td>
</tr>
</tbody>
</table>

#### Table 27. ISR for Software Interrupt 4

<table>
<thead>
<tr>
<th>Step</th>
<th>Relevant Bit Fields</th>
<th>Pseudo Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>swiIRQ4ISR</td>
<td>Increment SWirq4ctr</td>
<td>SWirq4ctr++</td>
</tr>
<tr>
<td>Clear Software IRQ 4</td>
<td>CLR=1</td>
<td>INTC_SSCIR[4] = 1</td>
</tr>
</tbody>
</table>
6.3 Code

6.3.1 main.c files

6.3.1.1 main.c (MPC551x shown with 8 MHz crystal)

/* main.c - Software vector mode program using C isr */
/* Aug 12 2009 SM - Based on AN2865 INTC Software Vector Mode example*/
/* Copyright Freescale Semiconductor, Inc. 2009 All rights reserved. */

#include "mpc5510.h" /* Use proper include file such as mpc5510.h or mpc5554.h */
extern IVOR4Handler();
extern uint32_t __IVPR_VALUE; /* Interrupt Vector Prefix value from link file*/
extern const vuint32_t IntcIsrVectorTable[];

uint32_t Pit1Ctr = 0;   /* Counter for PIT 1 interrupts */
uint32_t SWirq4Ctr = 0; /* Counter for software interrupt 4 */

void initSysclk(void) {
    /* Use 2 of the next 4 lines: */
    FMPLL.ESYNCR2.R = 0x00000007;     /* 8MHz xtal: ERFD to initial value of 7 */
    FMPLL.ESYNCR1.R = 0xF0000020;     /* 8MHz xtal: CLKCFG=PLL, EPREDIV=0, EMFD=0x20 */
    /*FMPLL.ESYNCR2.R = 0x00000005; */  /* 12MHz xtal: ERFD to initial value of 5 */
    /*FMPLL.ESYNCR1.R = 0xF0020030; */  /* 12MHz xtal: CLKCFG=PLL, EPREDIV=2, EMFD=0x30*/
    CRP.CLKSPC.B.XOSCEN = 1;          /* Enable external oscillator */
    while (FMPLL.SYNSR.B.LOCK != 1) {}; /* Wait for PLL to LOCK */
    /* Use 1 of the next 2 lines: */
    FMPLL.ESYNCR2.R = 0x00000005;     /* 8MHz xtal: ERFD change for 64 MHz sysclk */
    /*FMPLL.ESYNCR2.R = 0x00000003; */  /* 12MHz xtal: ERFD change for 64 MHz sysclk */
    SIU.SYSCLK.B.SYSLKSEL = 2;       /* Select PLL for sysclk */
}

void disableWatchdog(void) {
    SWT.SR.R = 0x0000c520;     /* Write keys to clear soft lock bit */
    SWT.SR.R = 0x0000d928;
    SWT.CR.R = 0x8000010a;     /* Clear watchdog enable (WEN) */
}

asm void initIrgVectors(void) {
    lis r3, __IVPR_VALUE@h          /* IVPR value is passed from link file */
    ori      r3, r3, __IVPR_VALUE@l
    mtivpr   r3
}

void initINTC(void) {
    INTC.MCR.B.HVEN_PRC0 = 0;     /* MPC551x Proc'r 0: initialize for SW vector mode*/
    INTC.MCR.B.VTES_PRC0 = 0;     /* MPC551x Proc'r 0: default vector table 4B offsets */
    INTC.IACKR_PRC0.R = (uint32_t) &IntcIsrVectorTable[0]; /* MPC551x: ISR table base*/
}

void initPIT(void) {
    SIU.SYSLK.B.LPCLKDIV1 = 0;    /* Divide sysclk by 1 for Group 1 modules */
    PIT.PITCTRL.R = 0;           /* Ensure PIT is not disabled */
    PIT.TLVAL[1].R = 64000;      /* Timeout= 64K sysclks x 1sec/64M sysclks= 1 ms */
    PIT.PITINTEN.R = 0x00000002; /* Enable PIT 1 flag to request INTC or DMA */
    PIT.PITINTSEL.R = 0x00000006; /* Assign PIT 1 flag to select IRQ, not DMA req. */
    PIT.PITEN.B.PEN1 = 1;       /* Start PIT counting */
    INTC.PSR[149].R = 0x01;     /* PIT 1 interrupt selects proc'r 0 & priority 1 */
}

void initSwIq4(void) {
    INTC.PSR[4].R = 2;          /* Software interrupt 4 IRQ priority = 2 */
}
void enableIrq(void) {
    INTC.CPR_PRC0.B.PRI = 0;  /* MPC551x Proc'0: Lower INTC's current priority */
    asm(" wrteel 1");  /* Enable external interrupts */
}

void main (void) {
    vuint32_t i = 0;  /* Dummy idle counter */
    initSysclk();  /* Set sysclk to 64 MHz */
    initIrqVectors();  /* Initialize exceptions: only need to load IVPR */
    initINTC();  /* Initialize INTC for software vector mode */
    initPIT();  /* Initialize PIT1 for 1kHz IRQ, priority 2 */
    initSwIrq4();  /* Initialize software interrupt 4 */
    enableIrq();  /* Ensure INTC current prority=0 & enable IRQ */
    while (1) {
        i++;
    }
}

void Pit1ISR(void) {
    Pit1Ctr++;  /* Increment interrupt counter */
    if (((Pit1Ctr & 1)==0) {  /* If Pit1Ctr is even */
        INTC.SSCIR[4].R = 2;  /* then invoke software interrupt 4 */
        PIT.PITFLG.B.TIF1 = 1;  /* Clear PIT 1 flag by writing 1 */
    }
}

void SwIrq4ISR(void) {
    SwIrq4Ctr++;  /* Increment interrupt counter */
    INTC.SSCIR[4].R = 1;  /* Clear channel's flag */
}
6.3.1.2  main.c (MPC555x: MPC563xM shown with 8 MHz crystal)

/* main.c - Software vector mode program using C isr */
/* Feb 03 2009 SM - Based on AN2865 INTC Software Vector Mode example*/
/* Copyright Freescale Semiconductor, Inc. 2009 All rights reserved. */

#include "mpc563m.h"  /* Use proper include file such as mpc5510.h or mpc5544.h */
extern IVOR4Handler();
extern uint32_t __IVPR_VALUE; /* Interrupt Vector Prefix value from link file*/
extern const vuint32_t IntcIsrVectorTable[];
    uint32_t Pit1Ctr = 0;   /* Counter for PIT 1 interrupts */
    uint32_t SWirq4Ctr = 0; /* Counter for software interrupt 4 */

void initSysclk(void) {   /* Initialize sysclk to 64MHz for 8 MHz crystal*/
    FMPLL.ESYNCR1.B.CLKCFG = 0X7;    /* Change clk to PLL normal mode from crystal */
    FMPLL.SYNCR.R = 0x16080000;     /* Initial values: PREDIV=1, MPD=12, RFD=1 */
    while (FMPLL.SYNSR.B.LOCK != 1) {}; /* Wait for FMPLL to LOCK */
    FMPLL.SYNCR.R = 0x16000000;     /* Final value for 64 MHz: RFD=0 */
}

asm void initIrqVectors(void) {
    lis r3, __IVPR_VALUE@h        /* IVPR value is passed from link file */
    ori  r3, r3, __IVPR_VALUE@l
    mtivpr r3
    li  r3, IVOR4Handler@l         /* IVOR4 = lower half of handler address */
    mtivor4r3
}

void initINTC(void) {
    INTC.MCR.B.HVEN = 0;      /* Initialize for SW vector mode */
    INTC.MCR.B.VTES = 0;      /* Use default vector table 4B offsets */
    INTC.IACKR.R=(uint32_t) &IntcIsrVectorTable[0]; /* INTC ISR table base */
}

void initPIT(void) {
    PIT.MCR.R = 0x00000001;   /* Enable PIT module & freeze count during debug */
    PIT.TIMER[1].LDVAL.R = 64000; /* Timeout= 64K sysclks x 1sec/64M sysclks= 1 ms */
    PIT.TIMER[1].TCTRL.R = 0x00000003; /* Start timer counting & freeze during debug */
    INTC.PSR[302].R = 0x1;       /* PIT 1 interrupt has priority 1 */
}

void initSwIrq4(void) {
    INTC.PSR[4].R = 2;          /* Software interrupt 4 IRQ priority = 2 */
}

void enableIrq(void) {
    INTC.CPR.B.PRI = 0;        /* Lower INTC's current priority */
    asm("wrteei 1");           /* Enable external interrupts */
}

void main (void) {        /* Dummy idle counter */
    uint32_t i = 0;
    initSysclk();       /* Set sysclk to 64 MHz */
    initIrqVectors();   /* Initialize exceptions: only need to load IVPR */
    initINTC();         /* Initialize INTC for software vector mode */
    initPIT();          /* Initialize PIT for 1kHz IRQ, priority 2 */
    initSwIrq4();       /* Initialize software interrupt 4 */
    enableIrq();        /* Ensure INTC current priority=0 & enable IRQ */
    while (1) {         /* Increment interrupt counter */
        i++;
    }
}

void Pit1ISR(void) {
    Pit1Ctr++;           /* Increment interrupt counter */
    if ((Pit1Ctr & 1)==0) { /* If PIT1Ctr is even*/
        INTC.SSCIR[4].R = 2; /* then invoke software interrupt 4 */
    }
    PIT.TIMER[1].TFLG.B.TIF = 1; /* Clear PIT 1 flag by writing 1 */
}

void SwIrq4ISR(void) { /* Increment interrupt counter */
    SWirq4Ctr++;        /* Clear channel's flag */
    INTC.SSCIR[4].R = 1;
}
**6.3.1.3 main.c file (MPC56xxB/P/S - MPC56xxS shown with 8 MHz crystal)**

/* main.c - Software vector mode program using C isr */
/* Jan 15, 2009 S.Mihalik- Initial version based on previous AN2865 example */
/* May 22 2009 S. Mihalik- Simplified by removing unneeded sysclk, PCTL code */
/* Jul 03 2009 S Mihalik - Simplified code */
/* Mar 15 2010 SM - modified initModesAndClocks, updated header */
/* Copyright Freescale Semiconductor, Inc. 2009, 2010. All rights reserved. */
#include "56xxS_0204.h" /* Use proper include file */
extern IVOR4Handler();
extern uint32_t _IVPR_VALUE; /* Interrupt Vector Prefix vaue from link file*/
extern const vuint32_t IntcIsrVectorTable[];

**void initModesAndClock(void) {**
ME.MER.R = 0x0000001D; /* Enable DRUN, RUN0, SAFE, RESET modes */
CGM.FPLL[0].CR.R = 0x02400100; /* Initialize PLL before turning it on: */
ME.RUN[0].R = 0x001F0074; /* 8 MHz xtal: Set PLL0 to 64 MHz */
ME.RUNPC[1].R = 0x00000010; /* Peri.Cfg. 1 settings: only run in RUN0 mode */
ME.PCTL[92].R = 0x01; /* PIT, RTI: select ME_RUN_PC[1] */
ME.MCTL.R = 0x40005AF0; /* Mode Transition to enter RUN0 mode: */
ME.MCTL.R = 0x4000A50F; /* Enter RUN0 Mode & Key */
while (ME.GS.B.S_MTRANS) {} /* Wait for mode transition to complete */
while(ME.GS.B.S_CURRENTMODE != 4) {} /* Verify RUN0 is the current mode */
**}**

**void disableWatchdog(void) {**
SWT.SR.R = 0x0000c520; /* Write keys to clear soft lock bit */
SWT.CR.R = 0x8000010A; /* Clear watchdog enable (WEN) */
**}**

**asm void initIrqVectors(void) {**
    _IVPR_VALUE@h
    __IVPR_VALUE@l
    mtivpr   r3
**}**

**void initINTC(void) {**
INTC.MCR.B.HVEN = 0; /* Initialize for SW vector mode */
INTC.MCR.B.VTES = 0; /* Use default vector table 4B offsets */
INTC.IACKR.R = (uint32_t) &IntcIsrVectorTable[0]; /* INTC ISR table base */
**}**

**void initPIT(void) {**
PIT.MCR.R = 0x00000001; /* Enable PIT and configure stop in debug mode */
PIT.CH[1].LDVAL.R = 64000; /* Timeout= 64K sysclks x 1sec/64M sysclks= 1 ms */
PIT.CH[1].TCTRL.R = 0x00000003; /* Enable PIT1 interrupt & start PIT counting */
INTC.PSR[60].R = 0x01; /* PIT 1 interrupt vector with priority 1 */
**}**

**void initSwIrq4(void) {**
INTC.PSR[4].R = 2; /* Software interrupt 4 IRQ priority = 2 */
**}**

**void enableIrg(void) {**
INTC.CPR.B.PRI = 0; /* Lower INTC's current priority */
asm(" wrteei 1"); /* Enable external interrupts */
**}**

Qorivva Simple Cookbook, Rev. 4
void main (void) {
    uint32_t i = 0;                 /* Dummy idle counter */
    initModesAndClock(); /* MPC56xxP/B/S: Initialize mode entries, set sysclk=64 MHz*/
    disableWatchdog();   /* Disable watchdog */
    initIrqVectors();    /* Initialize exceptions: only need to load IVPR */
    initINTC();          /* Initialize INTC for software vector mode */
    initPIT();           /* Initialize PIT1 for 1kHz IRQ, priority 2 */
    initSwIrq4();        /* Initialize software interrupt 4 */
    enableIrq();         /* Ensure INTC current prority=0 & enable IRQ */

    while (1) {
        i++;
    }
}

void Pit1ISR(void) {
    Pit1Ctr++;                  /* Increment interrupt counter */
    if ((Pit1Ctr & 1)==0) {     /* If Pit1Ctr is even*/
        INTC.SSCIR[4].R = 2;    /* then nvoke software interrupt 4 */
    }
    PIT.CH[1].TFLG.B.TIF = 1;   /* CLear PIT 1 flag by writing 1 */
}

void SwIrq4ISR(void) {
    SwIrq4Ctr++;                /* Increment interrupt counter */
    INTC.SSCIR[4].R = 1;        /* Clear channel's flag */
}
6.3.2 handlers_vle.s file

# handlers_vle.s - INTC software vector mode example using VLE instructions
# Description: Creates prolog, epilog for C ISR and enables nested interrupts
# Rev 1.0: April 23, 2004, S Mihalik,
# Rev 1.1 Aug 2, 2004 SM - delayed writing to EOIR until after disabling EE in epilog
# Rev 1.2 Sept 8 2004 SM - optimized & corrected r3,r4 restore sequence from rev 1.1
# Rev 1.2 Sept 21 2004 SM - optimized by minimizing time interrupts are disabled
# Rev 1.2 Sept 21 2004 SM - optimized by minimizing time interrupts are disabled
# Rev 1.2 Oct 8 2004 SM - optimized & corrected r3,r4 restore sequence from rev 1.1
# Rev 1.2 Oct 8 2004 SM - optimized by minimizing time interrupts are disabled
# Rev 1.3 Jul 2 2007 SM - Changed for MPC551x and mapped to .ivor handlers section
# Rev 1.4 Jan 22 2009 SM - Modified for VLE instructions, CodeWarrior 2.4 alpha
# Copyright Freescale Semiconductor, Inc. 2007. All rights reserved

# STACK FRAME DESIGN: Depth: 20 words (0xA0, or \ 80 bytes)
# **************
#  0x4C     *  GPR12    *    ^
#  0x48     *  GPR11    *    |
#  0x44     *  GPR10    *    |
#  0x40     *  GPR9     *    |
#  0x3C     *  GPR8     *    |
#  0x38     *  GPR7     *  GPRs (32 bit)
#  0x34     *  GPR6     *    |
#  0x30     *  GPR5     *    |
#  0x2C     *  GPR4     *    |
#  0x28     *  GPR3     *    |
#  0x24     *  GPR0     *    |
#  0x20     *  CR       * __CR__________
#  0x1C     *  XER      *    ^
#  0x18     *  CTR      *    |
#  0x14     *  LR       * locals & padding for 16 B alignment
#  0x10     *  SRR1     *    |
#  0x0C     *  SRR0     *    |
#  0x08     *  padding   * ___v__________
#  0x04     * resvd- LR  * Reserved for calling function
#  0x00     *  SP       * Backchain (same as gpr1 in GPRs)
# **************

.section .ivor_handlers, text_vle
.globl IVOR4Handler
.align 16
# Align IVOR handlers on a 16 byte boundary for MPC555x
# GHS, Cygnus, Diab(default) use .align 4; Metrowerks .align 16
.equiv INTC_IACKR_PRC0, 0xfff48010 # MPC551x: Proc 0 Interrupt Acknowledge Reg. addr.
equiv INTC_IACKR_PRC0, 0xfff48018 # MPC551x: Proc 0 End Of Interrupt Reg. addr.
equiv INTC_IACKR, 0xfff48010      # Single Core: Interrupt AcknowledgeReg. addr.
equiv INTC_EOIR, 0xfff48018       # Single Core: End Of Interrupt Reg. addr.

IVOR4Handler:

prolog:
    # PROLOGUE
    e_stwu r1, -0x50 (r1)       # Create stack frame and store back chain
    se_stw r3, 0x28 (r1)        # Store a working register
    mfsrr0 r3                   # Note: use se_ form for r0-7, r24-31 with positive offset
    se_stw r3, 0x0C (r1)        # Store SRR0:1 (must be done before enabling EE)
    mfsrri r3
    se_stw r3, 0x10 (r1)        # Use 2 of the next 4 lines for appropriate processor:
    e_lis r3, INTC_IACKR@ha     # Single core: Read pointer into ISR Vector Table
    e_lwz r3, INTC_IACKR@l(r3)  # Single core
    # e_lis r3, INTC_IACKR_PRC0@ha # MPC551x: Read proc'0 pointer into ISR Vector Table
    # e_lwz r3, INTC_IACKR_PRC0@l(r3) # MPC551x
    se_lwz r3, 0x0(r3)          # Read ISR address from ISR Vector Table using pointer
    wrteei 1                    # Set MSR[EE]=1(wait a couple clocks after reading IACKR)
    se_stw r4, 0x2C (r1)        # Store a second working register
    se_mflr r4                  # Store LR (LR will be used for ISR Vector)
    se_stw r4, 0x14 (r1)
se_mtlr r3                   # Store ISR address to LR to use for branching later

se_mtlr r3                   # Store CR
mfxer r3                     # Store XER
se_mfctr r3                  # Store CTR
se_stw r3, 0x20 (r1)         # Store rest of gprs
se_lwz r3, 0x14 (r1)         # Restore LR
se_mtlr r3                   # Restore CTR
se_lwz r3, 0x18 (r1)         # Restore XER
se_lwz r3, 0x20 (r1)         # Restore CR
se_lwz r3, 0x24 (r1)         # EPILOGUE
se_lwz r5, 0x30 (r1)         # Restore other gprs except working registers
se_lwz r6, 0x34 (r1)         # Ensure store to clear interrupt's flag bit completed
mbar 0
se_lwz r7, 0x38 (r1)
se_lwz r8, 0x3C (r1)
e_lwz r9, 0x40 (r1)
e_lwz r10, 0x44 (r1)
e_lwz r11, 0x48 (r1)
e_lwz r12, 0x4C (r1)
e_lwz r3, INTC_EOIR@ha # Single Core: Load upper half of EIOR address to r3
se_li r4, 0
wrteei 0                     # Disable interrupts for rest of handler
se_lwz r3, 0x0C (r1)         # Use 1 of the following 2 lines:
se_lwz r4, INTC_EOIR@l(r3)   # Restore SRR0
se_lwz r3, 0x10 (r1)         # Use 1 of the following 2 lines:
se_lwz r4, 0x2C (r1)         # Single Core: Write 0 to proc'r 0 INTC_EIOR
se_lwz r3, 0x28 (r1)         # Restore working registers
se_add16i r1, r1, 0x50       # Delete stack frame
se_rfi                        # End of Interrupt
6.3.3 IntcIsrVectors.c file

/* IntcIsrVectors.c - table of ISRs for INTC in SW vector Mode */
/* Description: Contains addresses for first 250 ISR vectors */
/* Table address gets loaded to INTC_IACKR */
/* Alignment: MPC551x MPC56xxP/B/S: 2 KB after a 4KB boundary; MPC555x: 64 KB */
/* Apr 22, 2004 S. Mihalik */
/* Mar 16, 2006 S. Mihalik - Modified for compile with Diab 5.3 */
/* Jun 29 2006 SM - Used pragma align instead of hard coding address */
/* Jul 15 2007 SM - alignment now done in link file; changes for MPC551x */
/* Oct 22 2008 SM - Changed to use PIT1 ISR instead of eMIOS Ch 0 ISR */
#include "typedefs.h"

void dummy (void);

extern void SwIRQISR(void);
extern void Pit1ISR(void);

/* Use next two lines with Diab compile */
#pragma section CONST "intc_sw_isr_vector_table" /* Diab compiler pragma */

/* Use pragma next two lines with CodeWarrior compile */
#pragma section data_type ".intc_sw_isr_vector_table" ".intc_sw_isr_vector_table" data_mode=far_abs

/* Use the next line for MPC551x or MPC56xxP/B/S */
void dummy (void); /* Wait forever or for watchdog timeout */

while (1) {}
6.3.4  ivor_branch_table.s file (MPC551x, MPC56xxPBS)

# ivor_branch_table.s - for use with MPC551x, MPC56xxP, MPC56xxB, MPC56xxS only
# Description:  Branch table for 16 MPC551x core interrupts
# Copyright Freescale 2007.  All Rights Reserved
# Rev 1.0 Jul 6 2007 S Mihalik - Initial version
# Rev 1.1 Aug 30 2007 SM - Made IVOR4Handler extern
# Rev 1.2 Sep 9 2008 SM - Converted assembly to VLE syntax

.extern IVOR4Handler
.section .ivor_branch_table,text_vle

.equ SIXTEEN_BYTES, 16   # 16 byte alignment required for table entries
# Diab compiler uses value of 4 (2**4=16)
# CodeWarrior, GHS, Cygnus use 16
.align SIXTEEN_BYTES
IVOR0trap: e_b IVOR0trap # IVOR 0 interrupt handler
.align SIXTEEN_BYTES
IVOR1trap: e_b IVOR1trap # IVOR 1 interrupt handler
.align SIXTEEN_BYTES
IVOR2trap: e_b IVOR2trap # IVOR 2 interrupt handler
.align SIXTEEN_BYTES
IVOR3trap: e_b IVOR3trap # IVOR 3 interrupt handler
.align SIXTEEN_BYTES
e_b IVOR4Handler # IVOR 4 interrupt handler (External Interrupt)
.align SIXTEEN_BYTES
IVOR5trap: e_b IVOR5trap # IVOR 5 interrupt handler
.align SIXTEEN_BYTES
IVOR6trap: e_b IVOR6trap # IVOR 6 interrupt handler
.align SIXTEEN_BYTES
IVOR7trap: e_b IVOR7trap # IVOR 7 interrupt handler
.align SIXTEEN_BYTES
IVOR8trap: e_b IVOR8trap # IVOR 8 interrupt handler
.align SIXTEEN_BYTES
IVOR9trap: e_b IVOR9trap # IVOR 9 interrupt handler
.align SIXTEEN_BYTES
IVOR10trap: e_b IVOR10trap # IVOR 10 interrupt handler
.align SIXTEEN_BYTES
IVOR11trap: e_b IVOR11trap # IVOR 11 interrupt handler
.align SIXTEEN_BYTES
IVOR12trap: e_b IVOR12trap # IVOR 12 interrupt handler
.align SIXTEEN_BYTES
IVOR13trap: e_b IVOR13trap # IVOR 13 interrupt handler
.align SIXTEEN_BYTES
IVOR14trap: e_b IVOR14trap # IVOR 14 interrupt handler
.align SIXTEEN_BYTES
IVOR15trap: e_b IVOR15trap # IVOR15 interrupt handler
7 INTC: Hardware Vector Mode, VLE Instructions

7.1 Description

Task: Using the Interrupt Controller (INTC) in hardware vector mode, this program provides two interrupts that show nesting. The main differences from the other INTC SW Mode, Classic Instructions example, are:

- VLE instructions are used (not available on MPC5553, MPC5554)
- Prologue/epilogue uses VLE assembly instructions
- Programmable Interrupt Timer (PIT) is used as the interrupt timer (note: PIT is not available in some MPC55xx devices, but an eMIOS channel could be used)
- Timer will count at the system clock rate (causing an interrupt at a count value of 0)
- System clock is set to 64 MHz

A relative interrupt response can be measured by reading the PIT count value in the first line of the interrupt service routine. An alternate method is to put a breakpoint in the beginning of the service routine and read the count register. NOTE: to get a true interrupt performance measurement, additional software is needed to initialize branch target buffers, configure flash, and enable cache (if implemented), as shown in other examples in this application note.

The interrupt handler will re-enable interrupts and later, in its interrupt service routine (ISR), invoke a second interrupt every other time. This provides an approximate 1 ms task (ISR) from the PIT and an approximate 2 ms task (ISR) from the software interrupt. The software interrupt will have a higher priority, so the 1 PIT ISR is preempted by the software interrupt. Both ISRs will have a counter.

The ISRs will be written in C, so the appropriate registers will be saved and restored in the handler. The SPE will not be used, so its accumulator will not be saved in the stack frame of the interrupt handler.

Exercise: Write a third interrupt handler which uses a software interrupt of a higher priority than the others. Invoke this new software interrupt from one of the other ISRs.

Figure 16. Hardware Vector Mode, VLE Instructions Example
7.2 Design

The overall program flow is shown below, followed by design steps and a stack frame implementation. The INTC when used in hardware vector mode uses a branch table for getting to each INTC vector’s handler. These are shown as handler_0, handler_1, etc. (Note: the code in this example does not have handlers for each INTC vector, so a common dummy trap address is used.)

For MPC551x, the second core has its own special purpose register IVPR, so the second core would have its own IntcHandlerBranchTable (not included in this example).

Also for MPC551x, either or both processors can receive the interrupt request from the Interrupt Controller. In this example, only one processor is selected in the MPC551x: processor 0 (e200z1). The selection is defined in INTC_PSR for each enabled interrupt, which here is PIT1 and software interrupt 4.

```
intc_hw_branch_table.s file

Pic1Handler:
prologue:
save registers
re-enable interrupts
branch to Pit1ISR
epilogue (return from ISR):
restore most registers
ensure prior stores completed
disable interrupts
write to EOIR to restore priority
restore SRR0:1
return

```

```
main.c file

main {
  init IVPR
  init INTC
  init Swlrq4
  init PIT1
  enable interrupts
  wait forever
}

Pit1ISR {
  increment counter
  if odd count, invoke SW 4 IRQ
  clear flag
}

Swlrq4ISR {
  increment counter
  clear flag
}
```

```
vector to:
MPC551x: IVPR0:19 + 0x800 + 149 × (0x4)
MPC555x: IVPR0:15 + 302 × (0x10)
MPC56xxBPS: IVPR0:19 + 0x800 + 60 × (0x4)
```

Figure 17. INTC HW Vector Mode, VLE Instructions. Overall Program Flow showing PIT 1 Interrupt
7.2.1 Modes Use Summary (MPC56xxB/P/S only)

Mode Transition is required for changing mode entry registers. Hence even enabling the crystal oscillator to be active in the default mode (DRUN) requires enabling the crystal oscillator in appropriate mode configuration register (ME_xxxx_MC) then initiating a mode transition. After reset, the mode is switched in this example from the default mode (DRUN) to RUN0 mode. The following table summarizes the mode settings used.

Table 28. Mode Configurations for MPC56xxB/P/S INTC HW Vector Mode VLE Example

<table>
<thead>
<tr>
<th>Mode</th>
<th>Mode Config. Register</th>
<th>Settings</th>
<th>Modes are enabled in ME_ME Register.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mode Config. Register</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sysclk Selection</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Clock Sources</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Memory Power Mode</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Main Voltage Reg.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>I/O Power Down Ctrl.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DRUN</td>
<td>ME_DRUN_MC</td>
<td>0x001F 0010 (default)</td>
<td>16 MHz IRC</td>
</tr>
<tr>
<td>RUN0</td>
<td>ME_RUN0_MC</td>
<td>0x001F 007D</td>
<td>PLL0</td>
</tr>
</tbody>
</table>

Other modes are not used in example

Peripherals also have configurations to gate clocks on and off, enabling low power. The following table summarizes the peripheral configurations used in this example.

Table 29. Peripheral Configurations for MPC56xxB/P/S INTC HW Vector Mode VLE Example

<table>
<thead>
<tr>
<th>Peripheral Config.</th>
<th>Peri. Config. Register</th>
<th>Enabled Modes</th>
<th>Peripherals Selecting Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>RUN3</td>
<td>RUN2</td>
</tr>
<tr>
<td>PC1</td>
<td>ME_RUNPC_1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Other peripheral configurations are not used in example
## Initialization

### Table 30. Initialization: INTC in Hardware Vector Mode, VLE Instructions

<table>
<thead>
<tr>
<th>Step</th>
<th>Relevant Bit Fields</th>
<th>Pseudo Code</th>
</tr>
</thead>
</table>
| **Variable Init** | Counter for PIT 1 interrupts  
Counter for software 4 interrupts | int Pit1Ctr = 0
int SWIrq4Ctr = 0 |
| **Table Init** | Load INTC HW Branch Table with ISR names for:  
INTC Vector for SW interrupt request 4  
INTC Vector for PIT 1 | e_b SwIrq4ISR  
e_b Pit1ISR |
| init Modes And Clock | Enable desired modes | ME_ME = 0x0000 001D |
| (MPC56xxBPS only) | **Initialize PLL0 dividers to provide 64 MHz for input crystal before mode configuration:**  
• 8 MHz xtal: FMPLL[0]_CR=0x02400100  
• 40 MHz xtal: FMPLL[0]_CR=0x12400100 (MPC56xxP & 8 MHz xtal requires changing default CMU_CSR value. See PLL example.) | **8 MHz Crystal:**  
CGM_  
FMPLL[0]_CR =0x02400100 |
| | Configure RUN0 Mode:  
• I/O Output power-down: no safe gating (default)  
• Main Voltage regulator is on (default)  
• Data, code flash in normal mode (default)  
• PLL0 is switched on  
• Crystal oscillator is switched on  
• 16 MHz IRC is switched on (default)  
• Select PLL0 (system pll) as sysclk | PDO=0  
MVRON=1  
DFLAON,  
CFLAON= 3  
PLL0ON=1  
XOSC0ON=1  
16MHz_IRCON=1  
SYSCLK=0x4 |
| | Assign peripheral configuration to peripherals:  
PIT, RTI: select ME_RUN_PC1 | ME_RUN_MC = 0x001F 0070 |
| | **Initialise software mode transition to RUN0 mode:**  
• Mode & key, then mode & inverted key  
• Wait for transition to complete  
• Verify current mode is RUN0 | TARGET_MODE = RUN0  
S_TRANS  
CURRENTMODE |
| | **Init Sysclk** | Initialize sysclk to 64 MHz, running from PLL |
| | **disable Watchdog** | Disable watchdog by writing keys to Status Register, then clearing WEN (MPC56xxBPS only) |

### Initialize PLL0 dividers to provide 64 MHz for input crystal before mode configuration:
- **8 MHz xtal:** FMPLL[0]_CR=0x02400100
- **40 MHz xtal:** FMPLL[0]_CR=0x12400100

(MPC56xxP & 8 MHz xtal requires changing default CMU_CSR value. See PLL example.)

### Configure RUN0 Mode:
- I/O Output power-down: no safe gating (default)
- Main Voltage regulator is on (default)
- Data, code flash in normal mode (default)
- PLL0 is switched on
- Crystal oscillator is switched on
- 16 MHz IRC is switched on (default)
- Select PLL0 (system pll) as sysclk

### Assign peripheral configuration to peripherals:
PIT, RTI: select ME_RUN_PC1

### Init Sysclk
Initialize sysclk to 64 MHz, running from PLL

### disable Watchdog
Disable watchdog by writing keys to Status Register, then clearing WEN (MPC56xxBPS only)

(See Section 10, “PLL: Initializing System Clock (MPC551x, MPC55xx)”)

See PLL Initialization example
### Table 30. Initialization: INTC in Hardware Vector Mode, VLE Instructions (continued)

<table>
<thead>
<tr>
<th>Step</th>
<th>Relevant Bit Fields</th>
<th>Pseudo Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>init</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vectors</td>
<td>Load the common interrupt vector prefix to spr IVPR. Value is defined in the link file.</td>
<td>spr IVPR = __IVPR_VALUE</td>
</tr>
<tr>
<td>init</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INTC</td>
<td>Initialize INTC:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Configure for Hardware vector mode (MPC551x note: only proc 0, e200z1, is used here)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HVEN_PRC0(551x)</td>
<td>INT_C_MCR[HV_EN_PRC0] = 1</td>
</tr>
<tr>
<td></td>
<td>HVEN (555x) = 0</td>
<td>INT_C_MCR[HVEN] = 1</td>
</tr>
<tr>
<td>init</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PIT</td>
<td>MPC551x: Init system clock divider for PIT, RTI to divide by 1 (also applies to other Group 1 peripherals of eSCI A, IIC)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LPCLKDIV1 = 0</td>
<td>SIU_SYSCLK[LPCLKDIV1] = 0</td>
</tr>
<tr>
<td></td>
<td>(default)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MDIS = 0</td>
<td>PIT_CTRL = 0x0000 0000</td>
</tr>
<tr>
<td></td>
<td>FRZ = 1</td>
<td>PIT_PITMC8 = 0x0000 0001</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Global controls:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Enable module</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• MPC555x, 56xxBPS: Freeze in debug mode</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Load a start count value for 64 MHz sysclk (PITs count down from value at sysclk rate)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• PIT 1 Timeout = 64 M / (64 MHz sysclk/sec) = 1 ms</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>PIT_TV = 64,000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PIT_TIMER1 = 64,000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PIT_CH1 = 64,000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Enable PIT 1 timer to count (counts down)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Enable PIT 1 to request IRQ</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MPC551x: Assign PIT 1 flag for IRQ instead of DMA</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PEN1(551x) = 1,</td>
<td>PIT_PITEN = 0x0000 0002</td>
</tr>
<tr>
<td></td>
<td>TEN (555x,</td>
<td>PIT_INTEN = 0x0000 0002</td>
</tr>
<tr>
<td></td>
<td>56xxBPS)=1</td>
<td>PIT_INTSEL = 0x0000 0002</td>
</tr>
<tr>
<td></td>
<td>TIE1 or TIE = 1</td>
<td>PIT_TIMER1 = 0x0000 0003</td>
</tr>
<tr>
<td></td>
<td>ISEL = 1</td>
<td>PIT_CH1 = 0x0000 0003</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Configure PIT1 interrupts:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• MPC551x: Select processor 0 (e200z1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Select priority 1</td>
<td>INT_C_PSR[149] = 0x01</td>
</tr>
<tr>
<td></td>
<td>PRC_SEL= 0 (z1)</td>
<td>INT_C_PSR[302] = 0x01</td>
</tr>
<tr>
<td></td>
<td>PRI = 1</td>
<td>INT_C_PSR[60] = 0x01</td>
</tr>
<tr>
<td>init</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SwIrrq4</td>
<td>Software Interrupt 4:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Raise priority to 2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• MPC551x: Select processor(s) to interrupt</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PRI = 2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PRC_SEL=0 (z1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>INT_C_PSR[4] = 0x02</td>
</tr>
<tr>
<td>enable</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ExtIrrq</td>
<td>Enable recognition of requests to INTC by lowering the INTC’s current priority to 0 from default of 15</td>
<td>INT_C_PPRP_PR(0) = 0</td>
</tr>
<tr>
<td></td>
<td>PRI = 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>INT_C_PPRP[4] = 0x01</td>
</tr>
<tr>
<td></td>
<td>Enable external interrupts: set MSR[EE] = 1</td>
<td>while 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Qorivva Simple Cookbook, Rev. 4**
### 7.2.2 Interrupt Handlers

**Table 31. Stack Frame for INTC Interrupt Handler (same as for Software Vector Mode example)**

<table>
<thead>
<tr>
<th>Stack Frame Area</th>
<th>Register</th>
<th>Location in Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 bit GPRs</td>
<td>r12</td>
<td>sp + 0x4C</td>
</tr>
<tr>
<td></td>
<td>r11</td>
<td>sp + 0x48</td>
</tr>
<tr>
<td></td>
<td>r10</td>
<td>sp + 0x44</td>
</tr>
<tr>
<td></td>
<td>r9</td>
<td>sp + 0x40</td>
</tr>
<tr>
<td></td>
<td>r8</td>
<td>sp + 0x3C</td>
</tr>
<tr>
<td></td>
<td>r7</td>
<td>sp + 0x38</td>
</tr>
<tr>
<td></td>
<td>r6</td>
<td>sp + 0x34</td>
</tr>
<tr>
<td></td>
<td>r5</td>
<td>sp + 0x30</td>
</tr>
<tr>
<td></td>
<td>r4</td>
<td>sp + 0x2C</td>
</tr>
<tr>
<td></td>
<td>r3</td>
<td>sp + 0x28</td>
</tr>
<tr>
<td></td>
<td>r0</td>
<td>sp + 0x24</td>
</tr>
<tr>
<td>CR</td>
<td>CR</td>
<td>sp + 0x20</td>
</tr>
<tr>
<td>locals and padding</td>
<td>XER</td>
<td>sp + 0x1C</td>
</tr>
<tr>
<td></td>
<td>CTR</td>
<td>sp + 0x18</td>
</tr>
<tr>
<td></td>
<td>LR</td>
<td>sp + 0x14</td>
</tr>
<tr>
<td></td>
<td>SRR1</td>
<td>sp + 0x10</td>
</tr>
<tr>
<td></td>
<td>SRR0</td>
<td>sp + 0x0C</td>
</tr>
<tr>
<td></td>
<td>padding</td>
<td>sp + 0x08</td>
</tr>
<tr>
<td>LR area</td>
<td>LR placeholder</td>
<td>sp + 0x04</td>
</tr>
<tr>
<td>back chain</td>
<td>r1</td>
<td>sp</td>
</tr>
<tr>
<td>Step</td>
<td>Relevant Bit Fields</td>
<td>Pseudo Code</td>
</tr>
<tr>
<td>--------------</td>
<td>---------------------</td>
<td>------------------------------------</td>
</tr>
<tr>
<td>prolog</td>
<td>Create stack frame</td>
<td>e_stwu sp, -0x50 (sp)</td>
</tr>
<tr>
<td></td>
<td>Save SRR0:1 because nested interrupts will be allowed</td>
<td>store SRR0:1, r3 registers to stack frame</td>
</tr>
<tr>
<td></td>
<td>Re-enable external interrupts by setting MSR[EE]</td>
<td>EE = 1 wrteei 1</td>
</tr>
<tr>
<td></td>
<td>Save other appropriate registers for C ISR</td>
<td>store other registers to stack frame</td>
</tr>
<tr>
<td></td>
<td>Branch to ISR, saving return address</td>
<td>e_bl Pit1ISR</td>
</tr>
<tr>
<td>Pit1ISR</td>
<td>Increment Pit1Ctr</td>
<td>Pit1Ctr ++</td>
</tr>
<tr>
<td></td>
<td>If Pit1Ctr is even, invoke Software Interrupt 4</td>
<td>if ((Pit1Ctr&amp;1) ==0), INTC_SSCIR[4] = 2</td>
</tr>
<tr>
<td></td>
<td>Clear Pit1Ctr interrupt flag by writing 1 to it TIF1 or TIF=1</td>
<td>PIT_PITFLG[TIF1] = 1, if ((Pit1Ctr&amp;1) ==0), INTC_SSCIR[4] = 2</td>
</tr>
<tr>
<td></td>
<td>epilog</td>
<td>Restore registers from stack frame except SRR0:1 and two working registers</td>
</tr>
<tr>
<td></td>
<td>Restore registers from stack frame except SRR0:1 and two working registers</td>
<td>load most registers from stack frame</td>
</tr>
<tr>
<td></td>
<td>Disable external interrupts by clearing MSR[EE]</td>
<td>EE = 0 wrteei 0</td>
</tr>
<tr>
<td></td>
<td>Restore former INTC's Current Priority</td>
<td>write 0 to INTC_EOIR_PRC 0</td>
</tr>
<tr>
<td></td>
<td>Restore SRR0:1 and working registers</td>
<td>restore SRR0:1, working registers from stack frame</td>
</tr>
<tr>
<td></td>
<td>Restore stack frame space</td>
<td>e_add16i sp, sp, 0x50</td>
</tr>
<tr>
<td></td>
<td>Return</td>
<td>se_rfi</td>
</tr>
</tbody>
</table>
### Table 33. Software 4 Interrupt Handler (INTC in Hardware Vector Mode, VLE Instructions)

<table>
<thead>
<tr>
<th>Step</th>
<th>Relevant Bit Fields</th>
<th>Pseudo Code</th>
<th>MPC51x</th>
<th>MPC555x</th>
</tr>
</thead>
<tbody>
<tr>
<td>prolog</td>
<td>Create stack frame</td>
<td>e_stwu sp, -0x50 (sp)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Save SRR0:1 because nested interrupts will be allowed</td>
<td>store SRR0:1, r3 registers to stack frame</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Re-enable external interrupts by setting MSR[EE] EE = 1</td>
<td>wrteei 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Save other appropriate registers for C ISR</td>
<td>store other registers to stack frame</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Branch to ISR, saving return address</td>
<td>bl SWIrq4ISR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SWIrq4ISR</td>
<td>Increment SWIrq4Ctr</td>
<td>SWIrq4Ctr++</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Clear Software IRQ 4 interrupt flag by writing 1 to it</td>
<td>CLR = 1</td>
<td>INTC_SSCIR4 = 1</td>
<td></td>
</tr>
<tr>
<td>epilog</td>
<td>Restore registers from stack frame except SRR0:1 and two working registers</td>
<td>load most registers from stack frame</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Ensure interrupt flag in ISR has completed clearing before writing to INTC_EOIR</td>
<td>mbar</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Disable external interrupts by clearing MSR[EE] EE = 0</td>
<td>wrteei 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Restore former INTC's Current Priority</td>
<td>write 0 to INTC_EOIR_PRC0</td>
<td>write 0 to INTC_EOIR</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Restore SRR0:1 and working registers</td>
<td>restore SRR0:1, working registers from stack frame</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Restore stack frame space</td>
<td>e_add16i sp, sp, 0x50</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Return</td>
<td>se_rfi</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# 7.3 Code

## 7.3.1 main.c files

### 7.3.1.1 mainc.c (MPC551x shown with 8 MHz crystal)

/ * main.c - Hardware vector mode program using C isr */  
/ * Feb 12 2009 SM - Based on AN2865 INTC Hardware Vector Mode example*/  
/ * Aug 12 2009 SM - Changed PLL initial ERFD value, added 12MHz crystal numbers */  
/ * Copyright Freescale Semiconductor, In.c 2009 All rights reserved. */

#include "mpc5510.h" /* Use proper include file such as mpc5510.h or mpc5554.h */

extern uint32_t __IVPR_VALUE; /* Interrupt Vector Prefix value from link file*/
extern const vuint32_t IntcIsrVectorTable[];

uint32_t Pit1Ctr = 0; /* Counter for PIT 1 interrupts */
uint32_t SWirq4Ctr = 0; /* Counter for software interrupt 4 */

void initSysclk(void) { /* Initialize PLL and sysclk to 64 MHz */
    /* Use 2 of the next 4 lines: */
    FMPLL.ESYNCR2.R = 0x0000007; /* 8MHz xtal: ERFD to initial value of 7 */
    FMPLL.ESYNCR1.R = 0xF0000020; /* 8MHz xtal: CLKCFG=PLL, EPREDIV=0, EMFD=0x20*/
    /*FMPLL.ESYNCR2.R = 0x0000005; */  /* 12MHz xtal: ERFD to initial value of 5 */
    /*FMPLL.ESYNCR1.R = 0xF0020030; */  /* 12MHz xtal: CLKCFG=PLL, EPREDIV=2, EMFD=0x30*/
    CRP.CLKSRC.B.XOSCEN = 1; /* Enable external oscillator */
    while (FMPLL.SYNSR.B.LOCK != 1) {}; /* Wait for PLL to LOCK */
    /* Use 1 of the next 2 lines: */
    FMPLL.ESYNCR2.R = 0x00000005; /* 8MHz xtal: ERFD change for 64 MHz sysclk */
    /*FMPLL.ESYNCR2.R = 0x00000003; */  /* 12MHz xtal: ERFD change for 64 MHz sysclk */
    SIU.SYSCLK.B.SYSCLKSEL = 2; /* Select PLL for sysclk */
}

asm void initIrqVectors(void) {
    lis r3, __IVPR_VALUE@h /* IVPR value is passed from link file */
    ori r3, r3, __IVPR_VALUE@l
    mtivpr r3
}

void initINTC(void) {
    INTC.MCR.B.HVEN_PRC0 = 1; /* MPC551x Proc'r 0: initialize for HW vector mode*/
}

void initPIT(void) {
    SIU.SYSCLK.B.LPCLKDIV1 = 0; /* Divide sysclk by 1 for Group 1 modules */
    PIT.PITCTRL.R = 0; /* Ensure PIT is not disbaled */
    PIT.TLVAL[1].R = 64000; /* Timeout= 64K sysclks x 1sec/64M sysclks= 1 ms */
    PIT.PITINTEN.R = 0x00000002; /* Enable PIT 1 flag to request INTC or DMA */
    PIT.PITINTSEL.R = 0x00000006; /* Assign PIT 1 flag to select IRQ, not DMA req. */
    PIT.PITEN.B.PEN1 = 1; /* Start PIT counting */
    INTC.PSR[149].R = 0x01; /* PIT 1 interrupt selects proc'r 0 & priority 1 */
}

void initSwIrq4(void) {
    INTC.PSR[4].R = 2; /* Software interrupt 4 IRQ priority = 2 */
}

void enableIrq(void) {
    INTC.CPR_PRC0.B.PRI = 0; /* MPC551x Proc'r 0: Lower INTC's current priority */
    /* asm(" wrteei 1") */
    /* Enable external interrupts */
}
void main (void) {
    vuint32_t i = 0;    /* Dummy idle counter */

    initSysclk();  /* Set sysclk to 64 MHz */
    initIrqVectors();  /* Initialize exceptions: only need to load IVPR */
    initINTC();  /* Initialize INTC for software vector mode */
    initPIT();  /* Initialize PIT1 for 1kHz IRQ, priority 2 */
    initSwIrq4();  /* Initialize software interrupt 4 */
    enableIrq();  /* Ensure INTC current priority=0 & enable IRQ */

    while (1) {
        i++;
    }
}

void Pit1ISR(void) {
    Pit1Ctr++;
    if (((Pit1Ctr & 1)==0) {  /* If PIT1Ctr is even*/
        INTC.SSCIR[4].R = 2;  /* then invoke software interrupt 4 */
    }
    PIT.PITFLG.B.TIF1 = 1;  /* Clear PIT 1 flag by writing 1 */
}

void SwIrq4ISR(void) {
    SWirq4Ctr++;
    INTC.SSCIR[4].R = 1;  /* Clear channel's flag */
}
7.3.1.2 main.c (MPC555x MPC563xM shown with 8 MHz crystal)

/* main.c - Hardware vector mode program using C isr */
/* Feb 03 2009 SM - Based on AN2865 INTC Software Vector Mode example*/
/* Copyright Freescale Semiconductor, Inc. 2009 All rights reserved. */
#include "mpc563m.h" /* Use proper include file such as mpc5510.h or mpc5554.h */

extern uint32_t __IVPR_VALUE; /* Interrupt Vector Prefix value from link file*/
uint32_t Pit1Ctr = 0; /* Counter for PIT 1 interrupts */
uint32_t SWirq4Ctr = 0; /* Counter for software interrupt 4 */

void initSysclk(void) {
    /* Intialize sysclk to 64MHz for 8 MHz crystal*/
    FMPLL.ESYNCR1.B.CLKCFG = 0x7; /* Change clk to PLL normal mode from crystal */
    FMPLL.SYNCR.R = 0x16080000; /* Initial values: PREDIV=1, MFD=12, RFD=1 */
    while (FMPLL.SYNSR.B.LOCK != 1) {}; /* Wait for FMPLL to LOCK */
    FMPLL.SYNCR.R = 0x16000000; /* Final value for 64 MHz: RFD=0 */
}

asm void initIrgVectors(void) {
    lisr3, __IVPR_VALUE@h /* IVPR value is passed from link file */
    ori       r3, r3, __IVPR_VALUE@l /* Note: IVPR lower bits are unused in MPC555x */
    mtivprr3
}

void initINTC(void) {
    INTC.MCR.B.HVEN = 1; /* Single core: Initialize for HW vector mosde */
}

void initPIT(void) {
    PIT.MCR.R = 0x00000001; /* Enable PIT module & freeze count during debug */
    PIT.TIMER[1].LDVAL.R = 64000; /* Timeout= 64K sysclks x 1sec/64M sysclks= 1 ms */
    PIT.TIMER[1].TCTRL.R = 0x00000003; /* Start timer counting & freeze during debug */
    INTC.PSR[302].R = 0x1; /* PIT 1 interrupt has priority 1 */
}

void initSwIrq4(void) {
    INTC.PSR[4].R = 2; /* Software interrupt 4 IRQ priority = 2 */
}

void enableIrq(void) {
    INTC.CPR.B.PRI = 0; /* Lower INTC's current priority */
    asm(" wrteei 1"); /* Enable external interrupts */
}

void main (void) {
    vuint32_t i = 0; /* Dummy idle counter */
    initSysclk(); /* Set sysclk to 64 MHz */
    initIrgVectors(); /* Initialize exceptions: only need to load IVPR */
    initINTC(); /* Initialize INTC for software vector mode */
    initPIT(); /* Initialize PIT1 for 1kHz IRQ, priority 2 */
    initSwIrq4(); /* Initialize software interrupt 4 */
    enableIrq(); /* Ensure INTC current priority=0 & enable IRQ */
    while (1) {
        i++;
    }
}

void Pit1ISR(void) {
    PIT1Ctr++; /* Increment interrupt counter */
    if (((PIT1Ctr & 1)==0) { /* If PIT1Ctr is even*/
        INTC.SSCIR[4].R = 2; /* then invoke software interrupt 4 */
    }
    PIT.TIMER[1].TFLG.B.TIF = 1; /* Clear PIT 1 flag by writing 1 */
}

void SwIrq4ISR(void) {
    SWirq4Ctr++; /* Increment interrupt counter */
    INTC.SSCIR[4].R = 1; /* Clear channel's flag */
}
7.3.1.3 main.c (MPC56xB/P/S - MPC56xxS shown with 8 MHz crystal)

/* main.c - Hardware vector mode program using C isr */
/* Feb 12, 2009 S.Mihalik Initial version based on previous AN2865 example */
/* May 22 2009 S. Mihalik- Simplified by removing unneeded sysclk, PCTL code */
/* Jul 03 2009 S Mihalik - Simplified code */
/* Mar 15 2010 SM - modified initModesAndClks, updated header */
/* Copyright Freescale Semiconductor, Inc. 2009, 2010. All rights reserved. */
#include "56xxS_0204.h" /* Use proper include file */

extern uint32_t __IVPR_VALUE; /* Interrupt Vector Prefix value from link file*/
    uint32_t Pit1Ctr = 0; /* Counter for PIT 1 interrupts */
    uint32_t SWirq4Ctr = 0; /* Counter for software interrupt 4 */

void initModesAndClock(void) {
    ME.MER.R = 0x0000001D; /* Enable DRUN, RUN0, SAFE, RESET modes */
    CGM.FMPLL[0].CR.R = 0x02400100; /* Initialize PLL before turning it on: */
    ME.RUN[0].R = 0x001F0074; /* 8 MHz xtal: Set PLL0 to 64 MHz */
    ME.RUNPC[1].R = 0x00000010; /* Peri. Cfg. 1 settings: only run in RUN0 mode */
    ME.PCTL[92].R = 0x01; /* PIT, RTI: select ME_RUN_PC[1] */
    ME.MCTL.R = 0x40005AF0; /* Mode Transition to enter RUN0 mode: */
    ME.MCTL.R = 0x4000A50F; /* Enter RUN0 Mode & Key */
    while (ME.GS.B.S_MTRANS) {} /* Wait for mode transition to complete */
    while(ME.GS.B.S_CURRENTMODE != 4) {} /* Verify RUN0 is the current mode */
}

void disableWatchdog(void) {
    SWT.SR.R = 0x0000c520; /* Write keys to clear soft lock bit */
    SWT.SR.R = 0x0000d928;
    SWT.CR.R = 0x8000010A; /* Clear watchdog enable (WEN) */
}

asm void initIrqVectors(void) {
    lis   r3, __IVPR_VALUE@h /* IVPR value is passed from link file */
    ori      r3, r3, __IVPR_VALUE@l
    mtivpr   r3
}

void initINTC(void) { /* Single core: initialize for HW vector mode */
}

void initPIT(void) {
    PIT.MCR.R = 0x00000001; /* Enable PIT and configure to stop in debug mode */
    PIT.CH[1].LDVAL.R = 64000; /* Timeout= 64000 sysclks x 1sec/64M sysclks = 1 ms */
    PIT.CH[1].TCTRL.R = 0x00000003; /* Enable PIT1 interrupt & start PIT counting */
    INTC.PSR[60].R = 0x01; /* PIT 1 interrupt vector with priority 1 */
}

void initSwIrq4(void) { /* Software interrupt 4 IRQ priority = 2 */
    INTC.PSR[4].R = 2;
}

void enableIrq(void) { /* Single Core: Lower INTC's current priority */
    INTC.CPR.B.PRI = 0;
    asm(" wrteei 1"); /* Enable external interrupts */
}
void main (void) {
  vuint32_t i = 0;      /* Dummy idle counter */

  initModesAndClock(); /* MPC56xxP/B/S: Initialize mode entries, set sysclk=64 MHz*/
  disableWatchdog();   /* Disable watchdog */
  initIrqVectors();    /* Initialize exceptions: only need to load IVPR */
  initINTC();          /* Initialize INTC for software vector mode */
  initPIT();           /* Initialize PIT1 for 1kHz IRQ, priority 2 */
  initSwIrq4();        /* Initialize software interrupt 4 */
  enableIrq();         /* Ensure INTC current prority=0 & enable IRQ */

  while (1) {
    i++;
  }
}

void Pit1ISR(void) {
  Pit1Ctr++;                   /* Increment interrupt counter */
  if ((Pit1Ctr & 1)==0) {      /* If PIT1Ctr is even*/
    INTC.SSCIR[4].R = 2;       /* then nvoke software interrupt 4 */
  }
  PIT.CH[1].TFLG.B.TIF = 1;   /* MPC56xxP/B/S: CLean PIT 1 flag by writing 1 */
}

void SwIrq4ISR(void) {
  SWirq4Ctr++;                /* Increment interrupt counter */
  INTC.SSCIR[4].R = 1;        /* Clear channel's flag */
}
# handlers_vle.s file

## Description
Creates prolog, epilog for C ISR and enables nested interrupts

### Rev 1.0: April 23, 2004, S Mihalik,
### Rev 1.1 Aug 2, 2004 SM - delayed writing to EOIR until after disabling EE in epilog
### Rev 1.2 Sept 8 2004 SM - optimized & corrected r3,r4 restore sequence from rev 1.1
### Rev 1.2 Sept 21 2004 SM - optimized by minimizing time interrupts are disabled
### Rev 1.3 Jul 2 2007 SM - Changes for MPC55lx and mapped to .ivor handlers section
### Rev 1.4 Jan 22 2009 SM - Modified for VLE instructions, CodeWarrior 2.4 alpha
### Rev 1.5 Mar 09 2010 SM - Removed unneeded Epilogue instruction: se_mtlr r3
### Copyright Freescale Semiconductor, Inc. 2009, 2010. All rights reserved

### Stack Frame Design: Depth: 20 words (0xA0, or 80 bytes)

<table>
<thead>
<tr>
<th>Stack Frame Design</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x4C * GPR12</td>
</tr>
<tr>
<td>0x48 * GPR11</td>
</tr>
<tr>
<td>0x44 * GPR10</td>
</tr>
<tr>
<td>0x40 * GPR9</td>
</tr>
<tr>
<td>0x3C * GPR8</td>
</tr>
<tr>
<td>0x38 * GPR7</td>
</tr>
<tr>
<td>0x34 * GPR6</td>
</tr>
<tr>
<td>0x30 * GPR5</td>
</tr>
<tr>
<td>0x2C * GPR4</td>
</tr>
<tr>
<td>0x28 * GPR3</td>
</tr>
<tr>
<td>0x24 * GPR0</td>
</tr>
<tr>
<td>0x20 * CR</td>
</tr>
<tr>
<td>0x1C * XER</td>
</tr>
<tr>
<td>0x18 * CTR</td>
</tr>
<tr>
<td>0x14 * LR</td>
</tr>
<tr>
<td>0x10 * SRR1</td>
</tr>
<tr>
<td>0x0C * SRR0</td>
</tr>
<tr>
<td>0x08 * padding</td>
</tr>
<tr>
<td>0x04 * resvd- LR</td>
</tr>
<tr>
<td>0x00 * SP</td>
</tr>
</tbody>
</table>

### Section .ivor_handlers, text_vle

.equ INTC_EOIR_PRC0, 0xfff48018 # Dual Core: Proc 0 End Of Interrupt Reg. addr.
equ INTC_EOIR, 0xfff48018 # Single Core: End Of Interrupt Reg. addr.

.extern Pit1ISR
.extern SwIrq4ISR
.globl Pit1Handler
.globl SwIrq4Handler
Pit1Handler:

```assembly
# PROLOGUE

e_stwu r1, -0x50 (r1)       # Create stack frame and store back chain
se_stw r3, 0x28 (r1)       # Store a working register
mfsrr0 r3
se_stw r3, 0x0C (r1)
mfSrrl r3
se_stw r3, 0x10 (r1)

wrteei 1                      # Set MSR[EE]=1
e_stwu r12, 0x4C (r1)       # Store rest of gprs
  e_stwu r11, 0x48 (r1)
  e_stwu r10, 0x44 (r1)
  e_stwu r9, 0x40 (r1)
  e_stwu r8, 0x3C (r1)
  se_stwu r7, 0x38 (r1)
  se_stwu r6, 0x34 (r1)
  se_stwu r5, 0x30 (r1)
  se_stwu r4, 0x2C (r1)
  se_stwu r0, 0x24 (r1)
  e_stwu r12, 0x4C (r1)       # Store a working register
mfsrr0 r3
se_stw r3, 0x28 (r1)
e_add16i r1, r1, 0x50        # Delete stack frame
se_rfi                        # End of Interrupt
```

```
# EPILOGUE

se_lwz r3, 0x14 (r1)          # Restore LR
se_mtlr r3
se_lwz r3, 0x18 (r1)          # Restore CTR
se_lwz r3, 0x1C (r1)          # Restore XER
mfxer r3
se_lwz r3, 0x20 (r1)          # Restore CR
mfcfl 0xff, r3
se_lwz r0, 0x24 (r1)          # Restore other gprs except working registers
se_lwz r5, 0x30 (r1)
se_lwz r6, 0x3C (r1)
se_lwz r7, 0x38 (r1)
e_lwz r8, 0x3C (r1)
e_lwz r9, 0x40 (r1)
e_lwz r10, 0x44 (r1)
e_lwz r11, 0x48 (r1)
e_lwz r12, 0x4C (r1)
mbar 0                        # Ensure store to clear interrupt flag bit completed

# Use 1 or 2 of the following 2 lines:
# e_lis r3, INTC_EOIR_PRC0@ha # Dual Core: Load upper half proc 0 EIOR addr to r3
  e_lis r3, INTC_EOIR@ha       # Single Core: Load upper half of EIOR address to r3
  se_li r4, 0
  wrteei 0                    # Disable interrupts for rest of handler
# Use 1 or 2 of the next appropriate lines:
# e_stw r4, INTC_EOIR_PRC0@l(r3) # Dual Core - Write 0 to proc'r 0 INTC_EIOR
  e_stw r4, INTC_EOIR@l(r3)   # Single Core - Write 0 to proc'r 0 INTC_EIOR
se_lwz r3, 0x0C (r1)          # Restore SRR0
mtSrr0 r3
se_lwz r3, 0x10 (r1)          # Restore SRR1
mtSrrl r3
se_lwz r4, 0x2C (r1)          # Restore working registers
e_add16i r1, r1, 0x50        # Delete stack frame
se_rfi                        # End of Interrupt
```

Qorivva Simple Cookbook, Rev. 4
SwIrq4Handler:

```assembly
# PROLOGUE
e_stwu r1, -0x50 (r1)       # Create stack frame and store back chain
se_stw r3, 0x28 (r1)       # Store a working register
mfssr0 r3
se_stw r3, 0x0C (r1)
mfssr1 r3
se_stw r3, 0x10 (r1)

wrteei 1                   # Set MSR[EE]=1
e_stwu r12, 0x4C (r1)      # Store rest of gprs
e_stw r11, 0x48 (r1)
e_stw r10, 0x44 (r1)
e_stw r9, 0x40 (r1)
e_stw r8, 0x3C (r1)
se_stw r7, 0x38 (r1)
se_stw r6, 0x34 (r1)
se_stw r5, 0x30 (r1)
se_stw r4, 0x2C (r1)
se_stw r0, 0x24 (r1)

swIrq4ISR

# EPILOGUE
se_lwz r3, 0x14 (r1)      # Restore LR
se_mtlr r3
se_lwz r3, 0x18 (r1)      # Restore CTR
se_lwz r3, 0x1C (r1)      # Restore XER
mtXer r3
se_lwz r3, 0x20 (r1)      # Restore CR
mfcrr r3
se_lwz r3, 0x21 (r1)      # Restore other gprs except working registers
se_lwz r5, 0x30 (r1)
se_lwz r6, 0x34 (r1)
se_lwz r7, 0x38 (r1)
e_lwz r8, 0x3C (r1)
e_lwz r9, 0x40 (r1)
e_lwz r10, 0x44 (r1)
e_lwz r11, 0x48 (r1)
e_lwz r12, 0x4C (r1)
mBar 0

# Use 1 of the following 2 lines:
# e_lis r3, INTC_EOIR_PRC0@ha # Dual Core: Load upper half proc 0 EIOR addr to r3
# e_lis r3, INTC_EOIR@ha      # Single Core: Load upper half of EIOR address to r3
se_lwz r4, 0

wrteei 0                     # Disable interrupts for rest of handler
# Use 1 or 2 of the next appropriate lines:
# e_stw r4, INTC_EOIR_PRC0@l(r3) # Dual Core - Write 0 to proc'r 0 INTC_EIOR
# e_stw r4, INTC_EOIR@l(r3)     # Single Core - Write 0 to proc'r 0 INTC_EIOR

se_lwz r3, 0x00C (r1)       # Restore SRR0
mtssrr0 r3
se_lwz r3, 0x10 (r1)       # Restore SRR1
mtssrr1 r3
se_lwz r4, 0x2C (r1)       # Restore working registers
se_lwz r3, 0x28 (r1)
e_addi16i r1, r1, 0x50    # Delete stack frame
se_rfi                        # End of Interrupt
```

Qorivva Simple Cookbook, Rev. 4
7.3.3 intc_hw_branch_table.s file (MPC56xxB/P/S vectors shown)

# Rev 1.0 Jul 2, 2007 S Mihalik
# Rev 1.1 Aug 30 1007 SM - Made SwIrq4Handler, emiosCh0Handler .extern
# Rev 2.0 Jan 22 2009 SM - Modified for VLE and MPC56xxB/P/S INTC vector numbers
# Copyright Freescale Semiconductor, Inc. 2007. All rights reserved

.equ ALIGN_OFFSET, 4 # MPC551x,MPC56xxB/P/S: 4 byte branch alignments (Diab/GHS use 2; CW 4)
.equ ALIGN_OFFSET, 4 # MPC555x: 16 byte branch alignments (Diab/GHS use 4; CW 16)

IntcHandlerBranchTable: # Only 100 example vectors are implemented here
                             # MPC555x: This table must have 64 KB alignment
                             # MPC551x, MPC56xxB/P/S: Requires 2 KB alignment after 4KB boundary

    .align ALIGN_OFFSET
 hw_vect0:   e_b hw_vect0        #INTC HW vector 0
 .align ALIGN_OFFSET
 hw_vect1:   e_b hw_vect1        #INTC HW vector 1
 .align ALIGN_OFFSET
 hw_vect2:   e_b hw_vect2        #INTC HW vector 2
 .align ALIGN_OFFSET
 hw_vect3:   e_b hw_vect3        #INTC HW vector 3
 .align ALIGN_OFFSET
 hw_vect4:   e_b SwIrq4Handler   # SW IRQ 4
 .align ALIGN_OFFSET
 hw_vect5:   e_b hw_vect5        #INTC HW vector 5

... etc. for contiguous vectors ........................................................

    .align ALIGN_OFFSET
 hw_vect57:  e_b hw_vect57       #INTC HW vector 57
 .align ALIGN_OFFSET
 hw_vect58:  e_b hw_vect58       #INTC HW vector 58
 .align ALIGN_OFFSET
 hw_vect59:  e_b hw_vect59       #INTC HW vector 59
 .align ALIGN_OFFSET
 hw_vect60:  e_b Pit1Handler     #INTC HW vector 60
 .align ALIGN_OFFSET
 hw_vect61:  e_b hw_vect61       #INTC HW vector 61
 .align ALIGN_OFFSET
 hw_vect62:  e_b hw_vect62       #INTC HW vector 62
 .align ALIGN_OFFSET
 hw_vect63:  e_b hw_vect63       #INTC HW vector 63
 .align ALIGN_OFFSET
 hw_vect64:  e_b hw_vect64       #INTC HW vector 64

... etc. for other vectors .................................................................
7.3.4  ivor_branch_table.s file (MPC551x, MPC56xxB/P/S only)

# ivor_branch_table.s - for use with MPC551x, MPC56xxP, MPC56xxB, MPC56xxS only
# Description: Branch table for 16 ore interrupts
# Copyright Freescale 2007, 2009. All Rights Reserved
# Rev 1.0 Jul 6 2007 S Mihalik - Initial version
# Rev 1.1 Aug 30 2007 SM - Made IVOR4Handler extern (for SW vector mode)
# Rev 1.2 Sep 9 2008 SM - Converted assembly to VLE syntax

.extern IVOR4Handler
.section .ivor_branch_table,text_vle

.equ SIXTEEN_BYTES, 16   # 16 byte alignment required for table entries
# Diab compiler uses value of 4 (2**4=16)
# CodeWarrior, GHS, Cygnus use 16

.align SIXTEEN_BYTES
IVOR0trap:  e_b IVOR0trap # IVOR 0 interrupt handler
.align SIXTEEN_BYTES
IVOR1trap:  e_b IVOR1trap # IVOR 1 interrupt handler
.align SIXTEEN_BYTES
IVOR2trap:  e_b IVOR2trap # IVOR 2 interrupt handler
.align SIXTEEN_BYTES
IVOR3trap:  e_b IVOR3trap # IVOR 3 interrupt handler
.align SIXTEEN_BYTES
IVOR4trap:  e_b IVOR4trap # IVOR 4 interrupt handler
.align SIXTEEN_BYTES
IVOR5trap:  e_b IVOR5trap # IVOR 5 interrupt handler
.align SIXTEEN_BYTES
IVOR6trap:  e_b IVOR6trap # IVOR 6 interrupt handler
.align SIXTEEN_BYTES
IVOR7trap:  e_b IVOR7trap # IVOR 7 interrupt handler
.align SIXTEEN_BYTES
IVOR8trap:  e_b IVOR8trap # IVOR 8 interrupt handler
.align SIXTEEN_BYTES
IVOR9trap:  e_b IVOR9trap # IVOR 9 interrupt handler
.align SIXTEEN_BYTES
IVOR10trap: e_b IVOR10trap # IVOR 10 interrupt handler
.align SIXTEEN_BYTES
IVOR11trap: e_b IVOR11trap # IVOR 11 interrupt handler
.align SIXTEEN_BYTES
IVOR12trap: e_b IVOR12trap # IVOR 12 interrupt handler
.align SIXTEEN_BYTES
IVOR13trap: e_b IVOR13trap # IVOR 13 interrupt handler
.align SIXTEEN_BYTES
IVOR14trap: e_b IVOR14trap # IVOR 14 interrupt handler
.align SIXTEEN_BYTES
IVOR15trap: e_b IVOR15trap # IVOR15 interrupt handler
8  MMU: Create TLB Entry

8.1  Description

Task: Initialize translation lookaside buffer (TLB) entry number six to define a memory page defined below.

This program shows how MMU entries are created. However, there is not any memory or I/O on current MPC5500 devices for the effective address range of this entry, so nothing can be accessed yet. The cache program will later use this entry to allocate part of cache as SRAM. Accessing the MMU’s TLB entries is done using sprs MAS0:3, as shown below.

![Figure 18. MMU TLB Entry Example](image)

This program shows how MMU entries are created. However, there is not any memory or I/O on current MPC5500 devices for the effective address range of this entry, so nothing can be accessed yet. The cache program will later use this entry to allocate part of cache as SRAM. Accessing the MMU’s TLB entries is done using sprs MAS0:3, as shown below.

<table>
<thead>
<tr>
<th>Page Attribute</th>
<th>Value</th>
<th>Relevant Bit Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Starting Effective Address</td>
<td>0x4004 0000</td>
<td>EPN</td>
</tr>
<tr>
<td>Size</td>
<td>4 KB</td>
<td>SIZE</td>
</tr>
<tr>
<td>Translation</td>
<td>No translation</td>
<td>RPN = EPN</td>
</tr>
<tr>
<td>Access allowed</td>
<td>Data, both R + W for all</td>
<td>UR, SR, UW, SW = 1</td>
</tr>
<tr>
<td>Access not allowed</td>
<td>Executable</td>
<td>UX, SX = 0</td>
</tr>
<tr>
<td>Write Mode Policy(^1)</td>
<td>Not write through</td>
<td>W = 0</td>
</tr>
<tr>
<td>Cache Inhibit</td>
<td>Not inhibited</td>
<td>I = 0</td>
</tr>
<tr>
<td>Coherency, Guarded, Endianness</td>
<td>Default</td>
<td>M, G, E = 0</td>
</tr>
<tr>
<td>Global Process ID</td>
<td>Default</td>
<td>TID = 0</td>
</tr>
<tr>
<td>Translation Space</td>
<td>Default</td>
<td>TS = 0</td>
</tr>
</tbody>
</table>

\(^1\) To allow MMU entries to specify the write mode policy, the cache write mode, L1CSR0[CWM], must be set to 1.

![Table 34. TLB Entry 6 Summary](table)

Qorivva Simple Cookbook, Rev. 4
**Exercise:** If your debugger supports viewing the MMU TLB table, view the table entries before and after running the code. Create a new MMU entry seven that starts at address 0x4009 0000 and has a size of 16 KB.

### 8.2 Design

Below is a version of figure 6-6 of the *e200z6 PowerPC™ Core Reference Manual* Rev. 0 for reference.

**Figure 19. MMU Assist Registers 0–3 Summary**

<table>
<thead>
<tr>
<th>Step</th>
<th>Relevant Bit Fields</th>
<th>Pseudo Code</th>
</tr>
</thead>
</table>
| Initialize MMU TLB entry 6 | Select TLB entry number and define MMU R/W & Replacement Control:  
- Select TLB entry 6  
- Select TLB 1 (required)  
- Null replacement | | |
| | Define Descriptor Context and Configuration Control:  
- Set page as valid  
- No invalidation protection  
- Use global process ID  
- Use default translation space  
- Size = 4KB | | spr MAS0 = 0x1006 0000 |
| | Define EPN and Page Attributes:  
- EPN = 0x40 0400 for address 0x4004 0000  
- WIMAGE = all 0’s; use cache inhibit, no write-thru | | spr MAS2 = 0x4004 0000 |
| | Define RPN and Access Control (here as RW data)  
- RPN = 0x40 0400 for address 0x4004 0000  
- Set user bits to 0 (unused)  
- Disable both user & supervisor execution  
- Enable both user and supervisor data R and W | | spr MAS3 = 0x4004 000F |
| Write TLB entry fields in MAS1:3 to the entry per MAS0[ESEL] | | tlbwe |
8.3 Code

/* main.c - MMU - create TLB entry example */
/* Description: Creates a new entry to the TLB table in the MMU */
/* Rev 1.0 Sept 28, 2003 S.Mihalik, Copyright Freescale, 2004. All Rights Reserved */
/* Notes: */
/* 1. MMU not initialized; must be done by debug scripts or BAM */
/* 2. L2SRAM not initialized; must be done by debug scripts */

#include "mpc563m.h" /* Use proper include file such as mpc5510.h or mpc5554.h */

asm void MMU_init_TLB6(void) {
    lis r3, 0x1006       /* Select TLB entry #, define R/W replacment control */
    mtMAS0 r3            /* Load MAS0 with 0x1006 0000 for TLB entry #6 */
        /* Define description context and configuration control:*/
        /* VALID=1, IPROT=0, TID=0, TS=0, TSIZE=1 (4KB size) */
    lis r3, 0x8000       /* Load MAS 1 with 0x8000 0100 */
    ori r3, r3, 0x0100
    mtMAS1 r3
        /* Load MAS2 with 0x4004 0000 */
    lis r3, 0x4004       /* Load MAS2 with 0x4004 0000 */
    mtMAS2 r3
        /* Define RPN and access control for data R/W */
        /* RPN = 0x4004 0000, U0:3=0, UX/SX=0, UR/SR/UW/SW=1 */
    lis r3, 0x4004       /* Load MAS3 with 0x4004 0000 */
    ori r3, r3, 0x000F
    mtMAS3 r3
    tlbwe /* Write entry defined in MAS0 (entry 6 here) to MMU TLB */
}

void main (void) {
    int i = 0;           /* Dummy idle counter */

    MMU_init_TLB6();    /* Define 4KB R/W space starting at 0x4004 0000 */
    while (1) { i++; }   /* Loop forever */
}
9 Cache: Cache as RAM

9.1 Description

**Task:** Use 4 KB of cache as RAM. Use the memory page created in Section 8, “MMU: Create TLB Entry,” that uses memory starting at address 0x4004 0000 and that is 4 KB in size.

Note that cache is not available on all MPC555x devices. It is available on those with the e200z6 core.

This example shows how to turn the cache on (enable) based on information from the e200z6 PowerPC™ Core Reference Manual, Rev 0. After it is enabled, part of the cache is allocated to be used as RAM. The advantages of using cache as RAM is that it increases available memory to a program and is the fastest type of memory. Normally cache initialization is done shortly after reset, well before main and C code execution.

Although this example is implemented in a C program, the cache as well as MMU entries would normally be initialized in an assembler program as part of initialization. See AN2789, MPC5500 Configuration and Initialization, for further information.

Cache invalidation takes approximately 134 cycles for the 32 KB cache of the MPC5554. This design simply polls the INV bit to wait for an indication that invalidation completed, i.e., LISR0[CINV] = 0. However, during this time it would be possible to perform some other task, or use other resources such as DMA.

**Exercise:** Execute the code. View the MMU entry created (if your debugger supports this feature). View and modify memory created in cache. Add instructions to lock some data or code in cache. (Hint: Point a gpr to the address of data or code to be locked, then have LockingLoop use either `dcbtls` or `icbtls` instruction for each cache line.)

![Diagram of cache as RAM example](image)

**Figure 20. Cache as RAM Example**

---

1. Use the debugger's memory display window with the address of the new memory created.
Table 36. Cache: Configure Part as RAM

<table>
<thead>
<tr>
<th>Step</th>
<th>Relevant Bit Fields</th>
<th>Pseudo Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable cache</td>
<td>If cache is already being invalidated, wait until done.</td>
<td>CINV = 0</td>
</tr>
<tr>
<td></td>
<td>Invalidate cache.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Use sync instructions before writing to L1CSR0</td>
<td>CINV = 1</td>
</tr>
<tr>
<td></td>
<td>• Set cache invalidation bit</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Wait until cache invalidation completes.</td>
<td>CINV = 0</td>
</tr>
<tr>
<td></td>
<td>If cache invalidation was aborted, then attempt another cache invalidation.</td>
<td>ABT= 1</td>
</tr>
<tr>
<td></td>
<td>Enable cache in copyback mode, keeping other control bits at 0.</td>
<td>CWM = 1</td>
</tr>
<tr>
<td></td>
<td>• Use sync instructions before writing to L1CSR0</td>
<td>CE = 1</td>
</tr>
<tr>
<td></td>
<td>• Set mode to copyback</td>
<td></td>
</tr>
<tr>
<td>Initialize MMU TLB entry 6</td>
<td>(See previous example, Section 8, “MMU: Create TLB Entry.”)</td>
<td>spr MAS0 = 0x1006 0000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>spr MAS1 = 0x8000 0100</td>
</tr>
<tr>
<td></td>
<td></td>
<td>spr MAS2 = 0x4004 0000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>spr MAS3 = 0x4004 0000F tlbwe</td>
</tr>
<tr>
<td>Lock 4KB in cache</td>
<td>Initialize loop counter for 4 KB block size / 32 byte line size = 128</td>
<td>spr CTR = 0x80</td>
</tr>
<tr>
<td></td>
<td>Load a register with start address of new address space.</td>
<td>r3 = 0x4004 0000</td>
</tr>
<tr>
<td></td>
<td>For each 32-byte cache line in the 4 KB space:</td>
<td>Locking Loop:</td>
</tr>
<tr>
<td></td>
<td>• Establish line address in cache (and zero out line)</td>
<td>dcbz instruction for line’s address</td>
</tr>
<tr>
<td></td>
<td>• Lock line in cache</td>
<td>dcbtls instruction for line’s address</td>
</tr>
</tbody>
</table>

1 Per Book E, a context synchronizing instruction (CSI), such as an isync, would be needed before the tlbwe instruction if there were uncompleted preceding storage accesses. Similarly a CSI, such as isync or mbar, would be needed after the tlbwe to ensure that any immediate subsequent storage accesses use the updated TLB entry values.

**CAUTION:** Do not execute cache-based code that reconfigures the cache which could overwrite the cached code being executed! Example improper sequence is:
1. Out of reset, the flash MMU page has Cache Inhibit = 0, so when the cache is enabled code will be put in cache
2. Cache is enabled.
3. MMU TLB entry is initialized as above example.
4. Cache line addresses are established and locked into cache.

**Possible result:** If the code just happens to be in the cache line that is being converted to be used as SRAM, then that code could be erased.

**Solution:** Before enabling cache, change the flash MMU page from Cl = 0 to Cl = 1, so flash is temporarily cache inhibited. After the cache is enabled and configured, then change the flash MMU page back to Cl = 0.
9.3 Code

/* main.c - Configure part of cache as SRAM */
/* Rev 0.1 Sept 27, 2004 S.Mihalik, Copyright Freescale, 2004. All Rights Reserved */
/* Notes: */
/* 1. MMU not initialized; must be done by debug scripts or BAM */
/* 2. L2SRAM not initialized; must be done by debug scripts */

#include "mpc563m.h" /* Use proper include file such as mpc5510.h or mpc5554.h */

asm void cache_enable(void) {
  /* main.c - Configure part of cache as SRAM */
  mfspr r3, l1csr0 /* Get current status of cache from spr L1CSR0*/
  li r4, 0x0002 /* In a scratch register set bit 30 to 1, other bits 0 */
  and r3, r4 /* Mask out all bits except CINV*/
  cmp li r3, 0x0002 /* Compare if CINV, bit 30, =1 */
  beq WaitForInvalidationComplete1

  InvalidateCache: /* Before writing to L1CSR0, execute msync & isync */
  msync
  mtspr l1csr0, r4 /* Invalidate cache: Set L1CSR0[CINV]=1; other fields=0*/

  WaitForInvalidationComplete2:
  mfspr r3, l1csr0 /* Get current status of cache from spr L1CSR0*/
  and r3, r4 /* Mask out all bits except CINV*/
  beq InvalidateCache /* If there was an aborted invalidation, attempt again */

  /* Enable cache */
  lis r3, 0x0010 /* In a scratch register set bit 12 = 0 (used to set CWM)*/
  ori r3, r3, 0x01 /* Also set bit 31 =1 (used for setting CE) */
  msync /* Before writing to L1CSR0, execute msync & isync */
  mtspr l1csr0, r3 /* Enable cache with Cache others to 0 */
}

asm void MMU_init_TLB6(void) {
  lis r3, 0x1006 /* Select TLB entry #, define R/W replacment control */
  mtMAS0 r3 /* Load MAS0 with 0x1006 0000 for TLB entry #6 */
  lis r3, 0x8000 /* Define description context and configuration control: */
  ori r3, r3, 0x0100 /* VALID=1, IPROT=0, TID=0, TS=0, TSIZE=1 (4KB size) */
  mtMAS1 r3
  lis r3, 0x4004 /* Define EPN and page attributes: */
  ori r3, r3, 0x000F /* EPN = 0x4004 0000, WIMAGE = all 0's */
  mtMAS2 r3 /* Load MAS2 with 0x4004 0000 */
  lis r3, 0x4004 /* Define RPN and access control for data R/W */
  ori r3, r3, 0x0000 /* RPN = 0x4004 0000, UD:3=0, UX/SX=0, UR/SR/UW/SW=1 */
  mtMAS3 r3 /* Load MAS3 with 0x4004 0000F */
  tlbwe /* Write entry defined in MAS0 (entry 6 here) to MMU TLB */
}

asm void cache_lock_4KB(void) {
  li r3, 0x80 /* Load r3 with loop count = 4KB/32B = 128 = 0x80 */
  mtCTR r3 /* Move loop count to spr CTR */
  lis r3, 0x4004 /* Point r3 to start of desired address (r3=0x40040000) */

  LockingLoop:
void main (void) {
    int i = 0;        /* Dummy idle counter */
    cache_enable();   /* Invalidate then enable cache in copyback mode */
    MMU_init_TLB6();  /* Define 4KB R/W space starting at 0x4004 0000, no transl’n*/
    cache_lock_4KB(); /* Lock 4KB R/W space starting at 0x4004 0000 in cache */
    while (1) { i++; } /* Loop forever */
}

dcbz  r0, r3           /* Establish address in cache for 32B cache line of 0's */
dcbt1s 0, r0, r3        /* Lock that line in cache */
addi  r3, r3, 0x20      /* Increment address pointer by 32 B */
bdnz  LockingLoop       /* Decrement CTR, and loop back if it is not yet zero */
10 PLL: Initializing System Clock (MPC551x, MPC55xx)

10.1 Description

**Task**: Initialize the system clock to run at 64 MHz from the PLL (FMPLL on MPC555x) whose input is an 8 MHz crystal. Enable, if necessary, CLKOUT.

After initializing the PLL predivider, divider, and multiplier, the PLL LOCK is tested by simply polling for the desired status to occur. In a real system, a maximum timeout mechanism would be implemented. If LOCK was not achieved in a maximum time, then you can log an error message and reset the part.

CLKOUT has a maximum specified value, hence sysclk has an external bus division factor (EBDF) before sysclk reaches CLKOUT. For this example, 16 MHz CLKOUT will be used.

Attempting to increase the frequency in one step may cause overshoot of the PLL beyond the maximum sysclk specification and briefly cause a sharp increase in current demand from the power supply. Therefore two frequency increases are used. The second increase only changes the divider after the PLL feedback loop, which does not cause change of lock because the divider is after the feedback path.

**Exercise**: Measure CLKOUT frequency. Then modify code to produce a new frequency.

![Figure 21. PLL Example Block Diagram](image)

(sysclk reset default values:
16 MHz for MPC551x, 12 MHz for MPC555x with 8 MHz crystal, 8 MHz for MPC563x with 8 MHz crystal)

<table>
<thead>
<tr>
<th>Signal</th>
<th>Pin Name</th>
<th>SIU PCR No.</th>
<th>Package Pin No.</th>
<th>Function Name</th>
<th>SIU PCR No.</th>
<th>Package Pin No.</th>
<th>EVB</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLKOUT</td>
<td>PE6</td>
<td>70</td>
<td>67</td>
<td>83</td>
<td>P13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>eMIOS Ch 12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
10.2  Design

Loss of lock and loss of clock detection is not enabled in this example. Because changing the predivider or multiplier can cause loss of lock, the loss-of-lock circuitry and interrupts would not normally be enabled until after these steps are executed.

The preliminary specification for MPC5516 has a maximum CLKOUT frequency of 25 MHz, so EBDF will be used to divide the 64 MHz sysclk by 4, producing CLKOUT at 16 MHz.

MPC563x note: some packages do not have CLKOUT, so eMIOS channel 12 is configured as OPWFMB to verify sysclk frequency. eMIOS channel 12’s output frequency is sysclk divided by 16.

In devices that do not exit reset with PLL enabled as system clock, it is often good practice to initialize PLL registers with the multiplier and dividers before turning on the oscillator. Otherwise the reset default values may try to force the PLL to run beyond its frequency specifications.

Be careful not to select a system clock that is not enabled. For example, SIU_SYSCLK[SYSCLKSEL] on MPC551x or FMPLL_ESYNCR1[CLKCFG] on MPC563xM is used to specify the system clock. Do not initialize this field to a source, such as PLL or an internal reference clock, that is not turned on and ready.

10.2.1  MPC551x

MPC551x devices have a system clock frequency formula from the PLL of:

\[
F_{sys} = F_{ref} \times \frac{(EMFD + 16)}{(EPREDIV + 1)(ERFD+1)}
\]

For an 8 MHz crystal (F_{ref}) and a target frequency of 64 MHz (F_{sys}), the above formula is used below with values chosen for final MPC551x multiplier and dividers as shown. (See the MPC551x Reference Manual for allowed values for the multiplier and dividers.) Note: Prediv. Output Freq range is 4 MHz to 10 MHz.

\[
\frac{F_{sys}}{F_{ref}} = \frac{64 MHz}{8 MHz} = \frac{8}{1} = \frac{(32 + 16)}{(0+1)(5+1)} = \frac{(EMFD + 16)}{(EPREDIV + 1)(ERFD+1)}
\]

For this MPC551x example we have final values of EMFD = 32, EPREDIV = 0, and ERFD = 5.

We must be careful the maximum system clock frequency and VCO minimum/maximum frequencies are within the specification in the Data Sheet as the multiplier and dividers are changed. The steps below show the effect on these frequencies, and use the directions given in the PLL chapter of the MPC5510 Microcontroller Family Reference Manual (June 2007), to accommodate the frequency overshoot. These documented steps are used even though we are using conservative frequencies in this example.

Note: ERFD should values, per the reference manual, are mostly odd numbers.
10.2.2 MPC555x

MPC555x devices have a system clock frequency formula of:

\[
F_{\text{sys}} = F_{\text{ref}} \times \frac{(MFD + 4)}{((PREDIV + 1) \times 2^{\text{ERFD}})}
\]

For an 8 MHz crystal \((F_{\text{ref}})\) and a target frequency of 64 MHz \((F_{\text{sys}})\), the above formula is used below with values chosen for MPC555x multiplier and dividers as shown. (See the MPC555x Reference Manual for allowed values for the multiplier and dividers.)

---

**Table 38. MPC551x Steps for Programming Enhanced PLL**

<table>
<thead>
<tr>
<th>Crystal Freq.</th>
<th>Step</th>
<th>PLL_ESYNCR1</th>
<th>PLL_ESYNCR2</th>
<th>Predivider Output Frequency (4 MHz - 10 MHz)</th>
<th>VCO Freq. (192 MHz–680 MHz per prelim. Data Sheet)</th>
<th>(F_{\text{sys}}) (3 MHz–66 MHz per prelim. Data Sheet)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 MHz</td>
<td>Reset Default Values</td>
<td>1</td>
<td>83</td>
<td>5</td>
<td>4 MHz</td>
<td>(396 MHz)</td>
</tr>
<tr>
<td></td>
<td>Write &gt; final ERFD value to PLL_ESYNCR2 (0x000 0007)</td>
<td>1</td>
<td>83</td>
<td>7</td>
<td>4 MHz</td>
<td>(396 MHz)</td>
</tr>
<tr>
<td></td>
<td>Write final EPREDIV and EMFD values to PLL_ESYNCR1 (0xF000 0020)</td>
<td>0</td>
<td>32</td>
<td>7</td>
<td>8 MHz</td>
<td>(384 MHz)</td>
</tr>
<tr>
<td></td>
<td>Turn on OSC and wait for PLL to LOCK</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td></td>
<td>Write final ERFD value to PLL_ESYNCR2 (0x0000 0005)</td>
<td>0</td>
<td>32</td>
<td>5</td>
<td>8 MHz</td>
<td>384 MHz</td>
</tr>
<tr>
<td>12 MHz</td>
<td>Reset Default Values</td>
<td>1</td>
<td>83</td>
<td>5</td>
<td>6 MHz</td>
<td>(594 MHz)</td>
</tr>
<tr>
<td></td>
<td>Write &gt; final ERFD value to PLL_ESYNCR2 (0x0000 0005)</td>
<td>1</td>
<td>83</td>
<td>5</td>
<td>6 MHz</td>
<td>(594 MHz)</td>
</tr>
<tr>
<td></td>
<td>Write final EPREDIV and EMFD values to PLL_ESYNCR1 (0xF002 0030)</td>
<td>2</td>
<td>48</td>
<td>5</td>
<td>4 MHz</td>
<td>(256 MHz)</td>
</tr>
<tr>
<td></td>
<td>Turn on OSC and wait for PLL to LOCK</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td></td>
<td>Write final ERFD value to PLL_ESYNCR2 (0x0000 0003)</td>
<td>2</td>
<td>48</td>
<td>3</td>
<td>4 MHz</td>
<td>256 MHz</td>
</tr>
</tbody>
</table>
For this example with an 8 MHz crystal, we have final values of MFD = 12, PREDIV = 1, and RFD = 0.

If a 40 MHz crystal (\(F_{\text{ref}}\)) is used, such as on Freescale’s MPC5561 and MPC5567 EVBs, the formula for 64 MHz sysclk is shown below. (See the reference manual for that device to learn the allowed values for multiplier and dividers.)

\[
\frac{F_{\text{sys}}}{F_{\text{ref}}} = \frac{64 \text{ MHz}}{8 \text{ MHz}} = \frac{8}{1} = \frac{(12 + 4)}{(1 + 1) \times 2^0} = \frac{(MFD + 4)}{((\text{PREDIV} + 1) \times 2^{\text{RFD}})}
\]

For this example with a 40 MHz crystal, we have final values of MFD = 12, PREDIV = 4, and RFD = 1.

One must ensure the PLL specifications are not violated for the individual processor’s data sheet. For example, the MPC5554 Microcontroller Data Sheet, Rev 1.4, contains the specifications given here. (Be sure to check your microcontroller’s latest data sheet for actual specified values.)

- ICO frequency \((f_{ico})\): 48 MHz to the maximum \(F_{\text{sys}}\). (64 MHz is used here.)
- Predivider output frequency: 4 MHz to the maximum frequency.

We will use the same pattern of setting sysclk in two stages: the RFD value is initially set to the final RFD value + 1, then wait for LOCK before setting the final RFD value. By waiting to achieve LOCK before increasing to the final frequency, there is the benefit that the transient demand to the circuit board’s power is reduced and we can eliminate the potential risk of overclocking the CPU and peripherals, which could lead to unpredictable results.

CLKOUT is assigned to the pad by default after reset, so no writing to its SIU_PCR is required. However, its SIU_PCR does have controls for disabling the pad’s output buffer and drive strength.

10.2.2.1 MPC563xM Differences

MPC563xM implements both a legacy mode, where PLL output frequency uses the MPC555x formula, and an enhanced mode, which uses a different formula offering more frequency choices. This example uses the legacy mode, which is more compatible with existing MPC555x code.

Another difference is that the PLL exits reset in the bypass mode, so sysclk operates at the crystal frequency. To have sysclk be based on the PLL output, bit field FMPLL_ESYNCR1[CLKCFG] must be changed.
Table 39. MPC551x, MPC555x Steps for PLL Example

<table>
<thead>
<tr>
<th>Step</th>
<th>Relevant Bit Fields</th>
<th>Pseudo Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Init sysclk</td>
<td>Assign pad CLKOUT signal (MPC551x only). • Pad assignment — CLKOUT • Output buffer is enabled • Slew rate control = max. (fastest slew rate)</td>
<td>PA = 1 OBE = 1 SRC = 3 SIU_PCR[70] = 0x060C</td>
</tr>
<tr>
<td>Initialize External Bus Divider Factor from default divide by 2 value to desired divide by 4 value. Desired final CLKOUT = 16 MHz = 64 MHz / (3+1).</td>
<td>EBDF = 3 SIU_ECCR[EBDF] = 3</td>
<td></td>
</tr>
<tr>
<td>Change clock to normal mode with crystal reference. (MPC551x and MPC555x only)</td>
<td>CLKCFG = 7</td>
<td></td>
</tr>
<tr>
<td>Set initial multiplier and dividers for 64 MHz sysclk: MPC551x with an 8 MHz crystal input, use: • Predivider = 0 + 1 = 1 • Multiple = 32 + 16 = 48 • Divider+1 = 7 + 1 = 8</td>
<td>MPC551x: 8 MHz • EPREDIV = 0 • EMFD = 32 (0x20) • EFRD = 7 PLL_ESYNCR2 = 0x0000 0007 for 8 MHz crystal, = 0x0000 0005 for 12 MHz crystal MPC555x, 8 MHz: • PREDIV = 1 • MFD = 12 (0xC) • RFD = 1 FMPLL_ESYNCR1 = 0x0000 0000 for 8 MHz crystal, = 0x4610 0000 for 40 MHz crystal</td>
<td></td>
</tr>
<tr>
<td>MPC555x with a 12 MHz crystal input, use: • Predivider = 2 + 1 = 3 • Multiple = 48 + 16 = 4 • Divider+1 = 5 + 1 = 6</td>
<td>MPC555x, 12 MHz: • EPREDIV = 2 • EMFD = 48 (0x30) • EFRD = 5 PLL_ESYNCR1 = 0xF000 0020 for 8 MHz crystal, = 0x0000 0000 for 12 MHz crystal</td>
<td></td>
</tr>
<tr>
<td>MPC555x with an 8 MHz crystal input, use: • Predivider = 0 + 1 = 1 • Multipler = 12 + 4 = 16 • Divider = 2^1 = 2</td>
<td>MPC555x 40 MHz: • PREDIV = 4 • MFD = 12 (0xC) • RFD = 2</td>
<td></td>
</tr>
<tr>
<td>MPC555x with a 40 MHz crystal input, use: • Predivider = 4 + 1 = 5 • Multipler = 12 + 4 = 16 • Divider = 2^2 = 4</td>
<td>PLL_ESYNCR2 = 0x0000 0003 for 12 MHz crystal</td>
<td></td>
</tr>
<tr>
<td>Enable external oscillator (MPC551x only) [NOTE: Make sure there is a delay such as waiting to lock before changing clock to run on PLL.]</td>
<td>XOSCEN = 1 CRP_CLKSR [XOSCEN] = 1</td>
<td></td>
</tr>
<tr>
<td>Wait for PLL to lock</td>
<td>Wait for PLL_SYNCSR [LOCK] = 1 Wait for FMPLL_SYNCSR [LOCK] = 1</td>
<td></td>
</tr>
<tr>
<td>Set final divider after feedback loop: MPC551x with an 8 MHz crystal: • Divider = 1 + 5 = 6</td>
<td>MPC551x, 8 MHz: • EFRD = 5 PLL_ESYNCR2 = 0x0000 0005 for 8 MHz crystal, = 0x0000 0003 for 12 MHz crystal</td>
<td></td>
</tr>
<tr>
<td>MPC551x with a 12 MHz crystal: • Divider = 1 + 3 = 4</td>
<td>MPC551x 12 MHz: • EFRD = 3 FMPLL_SYNCR = 0x1600 0000 for 8 MHz crystal, = 0x4608 0000 for 40 MHz crystal</td>
<td></td>
</tr>
<tr>
<td>MPC555x with 8 MHz crystal: • Divider = 2^0 = 1</td>
<td>MPC555x, 8 MHz: • RFD = 0</td>
<td></td>
</tr>
<tr>
<td>MPC555x with 40 MHz crystal: • Divider = 2^1 = 2</td>
<td>MPC555x 40 MHz: • RFD = 1</td>
<td></td>
</tr>
<tr>
<td>Select PLL for sysclk [and optionally could divide sysclk to peripherals for lower power] (MPC551x only)</td>
<td>SYSLCLKSEL = 2 SIU_SYSLCLK [SYSLCLKSEL] = 2</td>
<td></td>
</tr>
</tbody>
</table>

Qorivva Simple Cookbook, Rev. 4
10.3 Code

10.3.1 MPC551x with 8 MHz crystal

```c
#include "mpc5510.h"

void main (void) {
    volatile uint32_t i=0; /* Dummy idle counter */
    SIU.PCR[70].R = 0x060C; /* Assign pad PortE[6] as CLKOUT signal */
    SIU.ECCR.B.EBDF = 3; /* Divide sysclk by 3+1 for CLKOUT */
    FMPLL.ESYNCR2.R = 0x00000007; /* 8MHz xtal: ERFD to initial value of 7 */
    FMPLL.ESYNCR1.R = 0xF0000020; /* 8MHz xtal: CLKCFG=PLL, EMFD=0x20*/
    /* Use 1 of the next 2 lines: */
    FMPLL.ESYNCR2.R = 0x00000005; /* 8MHz xtal: ERFD change for 64 MHz sysclk */
    FMPLL.ESYNCR1.R = 0xF0020030; /* 12MHz xtal: CLKCFG=PLL, EMFD=0x30*/
    CRP.CLKSRC.B.XOSCEN = 1; /* Enable external oscillator */
    while (FMPLL.SYNSR.B.LOCK != 1) {}; /* Wait for PLL to LOCK */
    /* Use 1 of the next 2 lines: */
    SIU.SYSCLK.B.SYSCLKSEL = 2; /* Select PLL for sysclk */
    SIU.PCR[70].R = 0x060C; /* Assign pad PortE[6] as CLKOUT signal */
    SIU.ECCR.B.EBDF = 3; /* Divide sysclk by 3+1 for CLKOUT */
    FMPLL.ESYNCR2.R = 0x00000000; /* Final value for 64 MHz: RFD=0 */
    SIU.PCR[70].R = 0x060C; /* Assign pad PortE[6] as CLKOUT signal */
    SIU.ECCR.B.EBDF = 3; /* Divide sysclk by 3+1 for CLKOUT */
    while (1) { i++; } /* Loop forever */
}
```

10.3.2 MPC555x with 8 MHz crystal

```c
#include "mpc5554.h"

void main (void) {
    volatile uint32_t i=0; /* Dummy idle counter */
    SIU.PCR[70].R = 0x060C; /* Assign pad PortE[6] as CLKOUT signal */
    SIU.ECCR.B.EBDF = 3; /* Divide sysclk by 3+1 for CLKOUT */
    FMPLL.SYNCR.R = 0x16080000; /* Initial values: PREDIV=1, MFDFD=12, RFDFD=1 */
    while (FMPLL.SYNSR.B.LOCK != 1) {}; /* Wait for FMPLL to LOCK */
    FMPLL.SYNCR.R = 0x16000000; /* Final value for 64 MHz: RFDFD=0 */
    SIU.PCR[70].R = 0x060C; /* Assign pad PortE[6] as CLKOUT signal */
    SIU.ECCR.B.EBDF = 3; /* Divide sysclk by 3+1 for CLKOUT */
    while (1) { i++; } /* Loop forever */
}
```
### 10.3.3 MPC563x with 8 MHz crystal

/* main.c - PLL-sysclk example */
/* Description: Set PLL to run at 64 MHz based on 8 MHz crystal for MPC563x */
/* For testing devices without CLKOUT, an eMIOS channel is used */
/* Copyright Freescale Semiconductor, 2008. All rights reserved. */
/* Rev 1.0 Jul 12 2007 SM- Initial version */
/* Rev 1.1 Aug 14 2007 SM- Changed sysclk to 64 MHz */
/* Notes: */
/* 1. MMU not initialized; must be done by debug scripts or BAM */
/* 2. L2SRAM not initialized; must be done by debug scripts or in a crt0 type file */

#include "mpc563m.h" /* Used for MPC563m devices */

void initEMIOS(void) {
    EMIOS.MCR.B.GPRE = 0x3; /* eMIOS clk= sysclk/(GPRE+1)= sysclk/4 */
    EMIOS.MCR.B.ETB = 0; /* Ext. time base is disabled; Ch 23 drives ctr bus A */
    EMIOS.MCR.B.GPREN = 1; /* Enable eMIOS clock */
    EMIOS.MCR.B.FRZ = 0; /* Disable freezing channel counters in debug mode */
}

void initEMIOSch12(void) {
    EMIOS.CH[12].CBDR.R = 4; /* Period= 4 eMios clocks= 16 sysclks */
    EMIOS.CH[12].CADR.R = 3; /* Duty cycle in emios clks */
    EMIOS.CH[12].CCR.B.UCPFRE = 0; /* Channel counter uses divide by (0+1) prescaler */
    EMIOS.CH[12].CCR.B.UCPREN = 1; /* Channel counter's prescaler is loaded & enabled */
    EMIOS.CH[12].CCR.B.MODE = 0x58; /* Mode= OPWFMB, flag on B match */
    SIU.PCR[191].B.PA = 1; /* Initialize pad for eMIOS channel. */
    SIU.PCR[191].B.OBE = 1; /* Initialize pad for output */
}

void main (void) {
    volatile uint32_t i=0; /* Dummy idle counter */
    initEMIOS(); /* Init. eMIOS to provide sysclk/4 to eMIOS channels */
    initEMIOSch12(); /* Init. eMIOS channel 12 for sysclk/16 OPWFMB */
    SIU.ECCR.B.EBDF = 3; /* Divide sysclk by 3+1 for CLKOUT */
    FMPLL.ESYNCR1.B.CLKCFG = 0x7; /* Change clk to PLL normal mode from crystal */
    FMPLL.SYNCR.R = 0x16080000; /* Initial values: PREDIV=1, MPD=12, RFD=1 */
    while (FMPLL.SYNCR.B.LOCK != 1) {}; /* Wait for FMPLL to LOCK */
    FMPLL.SYNCR.R = 0x16000000; /* Final value for 64 MHz: RFD=0 */
    while (1) { i++; } /* Loop forever */
}
11 PLL: Initializing System Clock (MPC56xxB/P/S)

11.1 Description

Task: Initialize the system clock to run at 64 MHz from the PLL whose input is an 8 MHz or 40 MHz crystal. Enable CLKOUT to observe frequency of 16 MHz IRC, external oscillator (crystal), then FMPLL0.

MPC56xxB/P/S mode transition logic simplifies software by not requiring checking the status bits for external oscillator stable and PLL locked. A mode transition is until not complete until:

1. the external oscillator is stable (if the XOSCON bit is set in the targeted mode’s configuration) &
2. the PLL is locked (if the PLLON bit was also set in the targeted mode’s configuration).

Exercise: Measure CLKOUT frequency while stepping through code and verify proper frequencies.

![Figure 22. PLL Example Block Diagram](image)

<table>
<thead>
<tr>
<th>Signal</th>
<th>Port</th>
<th>SIU PCR No.</th>
<th>Package Pin No.</th>
<th>Port</th>
<th>SIU PCR No.</th>
<th>Package Pin No.</th>
<th>Port</th>
<th>SIU PCR No.</th>
<th>Package Pin No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLKOUT</td>
<td>PA[0]</td>
<td>0</td>
<td>12</td>
<td>16</td>
<td>G4</td>
<td>22</td>
<td>96</td>
<td>138</td>
<td></td>
</tr>
<tr>
<td>eMIOS Ch 21</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(used on cut 1 MPC56xxS because cut 1 did not have CLKOUT)

Qorivva Simple Cookbook, Rev. 4
11.2 Design

11.2.1 Mode Use

Mode Transition is required for changing mode entry registers. Hence even enabling the crystal oscillator to be active in the current mode, for example default mode (DRUN), requires enabling the crystal oscillator in DRUN mode configuration register (ME_DRUN_MC) then initiating a mode transition to the same DRUN mode. This example changes from DRUN mode to RUN0 mode, which is the normal, expected use after power up.

This minimal example simply polls a status bit to wait for the targeted mode transition to complete. However, the status bit could instead be enabled to generate an interrupt request (assuming the INTC is initialized beforehand). This would allow software to complete other initialization tasks instead of brute force polling of the status bit.

It is normal to use a timer when waiting for a status bit to change. This example by default would have a watchdog timer expire if for some reason the mode transition never completed. One could also loop code on incrementing a software counter to some maximum value as a timeout. If a timeout was reached, then an error condition could be recorded in EEPROM or elsewhere.

It is good practice after a mode transition to verify the desired mode was entered by checking the ME_GS[S_CURRENTMODE] field. If there was a hardware failure, a SAFE mode transition could preempt the desired mode transition. There is an interrupt that, if enabled, can be used generate an interrupt request upon a SAFE mode transition.

Peripherals also have configurations to gate clocks on and off, enabling low power. The following table summarizes the peripheral configurations used in this example for MPC56xxB and MPC56xxS. MPC56xxP does not have peripheral configurations for XOSC0 nor PLL0, hence no code is needed for these. Because initial silicon for MPC56xxS did not include CLKOUT, an eMIOS channel is used to indicate final sysclk frequency. MPC56xxP does not require any peripheral clock gating configuration for this example.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Mode Config. Register</th>
<th>Mode Config. Register Value</th>
<th>Settings</th>
<th>Clock Sources</th>
<th>Memory Power Mode</th>
<th>Main Voltage Reg.</th>
<th>I/O Power Down Ctrl</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>16MHz IRC</td>
<td>XOSC0 PLL0 PLL1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Off</td>
<td>Off</td>
<td>Normal</td>
<td>On</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Off</td>
<td>PLL0</td>
<td>Normal</td>
<td>Off</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DRUN</td>
<td>ME_DRUN_MC</td>
<td>0x001F 0010 (default)</td>
<td></td>
<td>On</td>
<td>Off</td>
<td>Normal</td>
<td>On</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RUN0</td>
<td>ME_RUN0_MC</td>
<td>0x001F 0074</td>
<td>PLL0</td>
<td>On</td>
<td>On</td>
<td>Normal</td>
<td>Off</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Other modes are not used in example</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 41. Mode Configurations Summary for MPC56xxB/P/S PLL Example

Modes are enabled in ME_ME Register.
### 11.2.2 PLL Calculations

The formula for PLL output clock (also called PHI) in normal mode is:

\[
F_{PLL \, output\, clock} = \frac{F_{xtal} \times \text{idf}}{(\text{idf} \times \text{odf})}
\]

which in terms of the PLL’s Control Register field names is equivalent to:

\[
F_{PLL \, output\, clock} = \frac{F_{xtal} \times \text{NDIV}}{((\text{IDIF} + 1) \times 2^{\text{ODIF}+1})}
\]

Also \(F_{vco}\) must be within a range of 256 MHz and 512 MHz. \(F_{vco}\) is defined as:

\[
F_{vco} = \frac{F_{xtal} \times \text{NDIV}}{\text{(IDIF} + 1)}
\]

The following table summaries calculations to obtain desired 64 MHz frequency for different crystal frequencies.

#### Table 42. Peripheral Configurations for MPC56xxB/P/S PLL Example

Low power modes are not used in example.

<table>
<thead>
<tr>
<th>Peripheral Config.</th>
<th>Peri. Config. Register</th>
<th>Enabled Modes</th>
<th>Peripherals Selecting Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC1</td>
<td>ME_RUNPC_1</td>
<td>RUN3 RUN2 RUN1 RUN0 DRUN SAFE TEST RESET</td>
<td>SIUL (MPC56xxB/S) - used for enabling pad for output clock</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 0 0 1 0 0 0 0</td>
<td>68</td>
</tr>
</tbody>
</table>

Other peripheral configurations are not used in example

### 11.2.3 Progressive Clock Switching

Progressive clock switching is expected to be used on system clock. When changing the system clock to run on a PLL frequency, the PLL locks at a divided frequency, then gradually decreases the division until

#### Table 43. MPC56xxB/P/S PLL0 Calculations

(NDIV, IDF and ODF are bit field values in GCM FMPLL0 Control Register.
Verify range values in microcontroller reference manual.
Ranges shown are per MPC5604B Reference Manual, MPC5604B RM, Rev. 1, 4/2008.

<table>
<thead>
<tr>
<th>(F_{xtal})</th>
<th>NDIV</th>
<th>IDF</th>
<th>(F_{vco})</th>
<th>ODF</th>
<th>(F_{PLL , output, clock})</th>
<th>CR value</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 MHz</td>
<td>64</td>
<td>0</td>
<td>512 MHz</td>
<td>2</td>
<td>64 MHz</td>
<td>0x0240 0100</td>
</tr>
<tr>
<td>40 MHz</td>
<td>64</td>
<td>4</td>
<td>512 MHz</td>
<td>2</td>
<td>64 MHz</td>
<td>0x1240 0100</td>
</tr>
</tbody>
</table>
it is divided by 1. (See illustration below.) The effect is to gradually increase current consumption instead of a single large increase.

![Diagram](image)

In this example, the system clock changes from 16 MHz internal reference clock to a PLL output clock running at 64 MHz. When the PLL locks, the mode transition can complete. However with progressive clock switching, the PLL output frequency, sysclk in this example, is initially divided by 8 then changes gradually to the programmed frequency without losing lock. The table below illustrates how long this gradual increase takes.

**Table 44. Progressive Clock Switching for 64 MHz PLL operation**

<table>
<thead>
<tr>
<th>PLL lock frequency</th>
<th>Division Factor</th>
<th>PLL output clock frequency</th>
<th>Number of PLL output clock cycles during division</th>
<th>Number of PLL lock frequency cycles during division</th>
<th>Accumulated time (for 64 MHz PLL lock frequency)</th>
</tr>
</thead>
<tbody>
<tr>
<td>64 MHz</td>
<td>8</td>
<td>8 MHz</td>
<td>8</td>
<td>8 x 8 = 64</td>
<td>1 usec</td>
</tr>
<tr>
<td>64 MHz</td>
<td>4</td>
<td>16 MHz</td>
<td>16</td>
<td>4 x 16 = 64</td>
<td>2 usec</td>
</tr>
<tr>
<td>64 MHz</td>
<td>2</td>
<td>32 MHz</td>
<td>32</td>
<td>2 x 32 = 64</td>
<td>3 usec</td>
</tr>
<tr>
<td>64 MHz</td>
<td>1</td>
<td>64 MHz onward</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

As shown above, after the mode transition several microseconds are needed before the PLL output clock, (sysclk in this example) is running at full frequency. The number of system clock cycles before running at full frequency is $8 + 16 + 32 = 56$ sysclks, which takes 3 usec.

Progressive clock switching is enabled in the Clock Generation Module’s (MC_CGM’s) FMPLL Control Register (CR), en_pll_sw bit.

### 11.2.4 Clock Monitor Unit (CMU) — XOSC Frequency Monitor

One feature of the CMU is monitoring the XOSC frequency with respect to the 16 MHz FIRC. If the crystal frequency is less than $\text{FIRC} \div 2^{\text{RCDIV}}$, then, by reset default settings, a reset occurs. RCDIV is a programmable field in the CMU_CSR register.

For MPC56xxB/S, RCDIV defaults to a value of three, so as long as $\text{F}_{\text{XOSC}} > 16 \text{ MHz} \div 2^3$ (2 MHz) no action takes place.

However, MPC56xxP, which may have a 40 MHz crystal for FlexRAY operation, has a reset default of RCDIV = 0. Therefore the threshold is 16 MHz $\div 2^0$ (16 MHz). If, for example, an 8 MHz crystal is used instead of a 40 MHz crystal, CMU_CSR[RCDIV] must change before turning on XOSC in the mode configuration register.
11.2.5 Oscillator Stabilization Counter

Both FIRC and SIRC have counting periods that are used after reset and when the oscillator is switched on. This counting period ensures that the external oscillator clock signal is stable before it can be selected by the system.\(^1\) Depending on the crystal, the reset default value in CGM_FXOSC_CTL[EOCV] for FIRC or CGM_SXOSC_CTL[EOCV] for SIRC may need to be changed before the oscillator is turned on.

On MPC56xxS “cut 1,” CGM_FXOSC_CTL contained an oscillator enable bit which was disabled after reset. This control has been removed from MPC56xxS, and is also not on the other devices covered here.

Table 45. MPC5606B, MPC56xxP, MPC56xxS Steps for PLL Example

<table>
<thead>
<tr>
<th>Step</th>
<th>Relevant Bit Fields</th>
<th>Pseudo Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Init Modes and Clock</td>
<td>Enable desired modes</td>
<td>RUN0, DRUN=1 ME_ME = 0x0000 001D</td>
</tr>
<tr>
<td></td>
<td>If necessary, adjust XOSC clock monitor frequency:</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>• 8MHz Crystal: monitor Fxosc &gt; 4 MHz</td>
<td>RCDIV=4 -</td>
</tr>
<tr>
<td></td>
<td>• 40MHz Crystal: monitor Fxosc &gt; 16 MHz</td>
<td>RCDIV=0 40 MHz Crystal: CGM_CMU_CSR = 0x0000 0000 (default value for MPC56xxP)</td>
</tr>
<tr>
<td></td>
<td>Initialize PLL0 dividers to provide 64 MHz for input crystal before mode configuration, using progressive clock switching:</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>• 8MHz Crystal: FMPLL_CR=0x02400100</td>
<td>PD0=0 8 MHz Crystal: CGM_FMPLL_CR = 0x0240 0100</td>
</tr>
<tr>
<td></td>
<td>• 40MHz Crystal: FMPLL_CR=0x12400100</td>
<td>PDO=0 40 MHz Crystal: CGM_FMPLL[0]_CR = 0x1240 0100</td>
</tr>
<tr>
<td></td>
<td>Configure RUN0 Mode:</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>• I/O Output power-down: no safe gating</td>
<td>MVRON=1 ME_RUN0_MC = 0x001F 0070</td>
</tr>
<tr>
<td></td>
<td>• Main Voltage regulator is on (default)</td>
<td>DFLAON, ME_RUN_PC1 = 0x0000 0010</td>
</tr>
<tr>
<td></td>
<td>• Data, code flash in normal mode (default)</td>
<td>CFLAON= 3</td>
</tr>
<tr>
<td></td>
<td>• PLL0 is switched on</td>
<td>PLL0ON=1</td>
</tr>
<tr>
<td></td>
<td>• Crystal oscillator is switched on</td>
<td>XOSC0ON=1</td>
</tr>
<tr>
<td></td>
<td>• 16 MHz IRC is switched on (default)</td>
<td>16MHz_IRICON=1</td>
</tr>
<tr>
<td></td>
<td>• Select PLL0 (system pll) as sysclk</td>
<td>SYSCLK=0x4</td>
</tr>
<tr>
<td></td>
<td>MPC56xxB/S:</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>• Peri. Config.1: run in RUN0 mode only.</td>
<td>ME_PCTL68 = 0x01</td>
</tr>
<tr>
<td></td>
<td>Assign peripheral configuration to peripherals:</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>• SIUL (MPC5500B/S)</td>
<td>ME_RUN0_MC = 0x001F 0070</td>
</tr>
<tr>
<td></td>
<td>Initiate software mode transition to RUN0 mode</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>• Mode and key</td>
<td>TARGET_MODE= RUN0 ME_MCTL =0x4000 5AF0</td>
</tr>
<tr>
<td></td>
<td>• Mode and inverted key</td>
<td>S_TRANS ME_MCTL =0x4000 A50F</td>
</tr>
<tr>
<td></td>
<td>• Wait for transition complete status flag</td>
<td>wait for ME_GS[S_TRANS] = 0</td>
</tr>
<tr>
<td></td>
<td>• NOTE: if transition does not complete, check status flags such as ME_GS[XOSC] for cause.</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>• Verify current mode is desired mode</td>
<td>S_CURRENTMODE verify ME_GS[S_CURRENTMODE] = RUN0</td>
</tr>
<tr>
<td></td>
<td>• NOTE: This step ensures a SAFE mode transition did not occur at this point.</td>
<td>-</td>
</tr>
</tbody>
</table>

\(^1\)EOCV description, Table 3-9, MPC5604B/S Microcontroller Reference Manual, Rev.4 12 Aug 2009
### Initialize Output Clock /4

- **Step**: Enable Output Clock to pin
- **Relevant Bit Fields**: EN=1
- **Pseudo Code**: CGM_OCEN[EN] = 1

### Select division of output clock by 4

- **Step**: SELDIV = 2
- **Relevant Bit Fields**: CGM_OCDSSC[SELDIV] = 2

### Select 16 MHz IRC as output clock

- **Step**: SELCTL = 1 (MPC56xxB)
- **Relevant Bit Fields**: CGM_OCDSSC[SELCTL] = 1

### Assign output clock (CLKOUT) to pad:

- **Step**: MPC56xxB: PA[0] pad assignmt = alt func 2
- **Relevant Bit Fields**: SIU_PCR[0] = 0x0800
- **Pseudo Code**: CGM_OCDSSC[SELDIV] = 2

### Select crystal oscillator as output clock

- **Step**: SELCTL = 0 (MPC56xxB)
- **Relevant Bit Fields**: SIU_PCR[22] = 0x0400
- **Pseudo Code**: CGM_OCDSSC[SELCTL] = 0

### Select PLL0 as output clock

- **Step**: SELCTL = 2
- **Relevant Bit Fields**: SIU_PCR[103] = 0x0E00
- **Pseudo Code**: CGM_OCDSSC[SELCTL] = 2

### Disable Watchdog

- **Step**: Write keys to clear soft lock bit
- **Pseudo Code**: SWT_SR = 0x000 0C520
- **Step**: Clear watchdog enable bit
- **Pseudo Code**: SWT_SR = 0x0000 D928
- **Step**: Clear watchdog enable bit
- **Pseudo Code**: SWT_CR = 0x8000 010A

---

**Table 45. MPC5606B, MPC56xxP, MPC56xxS Steps for PLL Example** (continued)
11.3 Code

11.3.1 MPC56xxB with 8 MHz crystal

/* main.c - PLL example for MPC56xxB */
/* Description:  Set sysclk to FMPLL0 running at 64 MHz & enable CLKOUT */
/* Jan 14 2009 S Mihalik - Initial version */
/* May 22 2009 S Mihalik - simplified code */
/* Jun 24 2009 S Mihalik - Simplified code */
/* Mar 10 2010 S Mihalik - Modified initModesAndClock & updated header file */
/* Copyright Freescale Semiconductor, Inc 2009, 2010. All rights reserved. */
#include "MPC5604B_0M27V_0102.h" /* Use proper include file */

void initModesAndClock(void) {
    ME.MER.R = 0x0000001D;          /* Enable DRUN, RUN0, SAFE, RESET modes */
    /* Initialize PLL before turning it on: */
    /* Use 1 of the next 2 lines depending on crystal frequency: */
    CGM.FMPLL_CR.R = 0x02400000;    /* 8 MHz xtal: Set PLL0 to 64 MHz */
    /* 40 MHz xtal: Set PLL0 to 64 MHz */
    ME.RUN[0].R = 0x001F0074;       /* RUN0 cfg: 16MHzIRCON, OSC0ON, PLL0ON, syclk=PLL0 */
    ME.RUNPC[1].R = 0x00000010;   /* Peri.Cfg. 1 settings: only run in RUN0 mode */
    ME.PCTL[68].R = 0x01;           /* MPC56xxB/S: select ME.RUNPC[1] */
    /* Mode Transition to enter RUN0 mode: */
    ME.MCTL.R = 0x40005AF0;         /* Enter RUN0 Mode & Key */
    ME.MCTL.R = 0x4000A50F;         /* Enter RUN0 Mode & Inverted Key */
    while (ME.GS.B.S_MTRANS == 1) {} /* Wait for mode transition to complete */
    /* Notes: */
    /* 1. I_TC IRQ could be used here instead of polling */
    /* to allow software to complete other init. */
    /* 2. A timer could be used to prevent waiting forever. */
    while (ME.GS.B.S_CURRENTMODE != 4) {} /* Verify RUN0 is the current mode */
    /* Note: This verification ensures a SAFE mode * /
    /* transition did not occur. SW could instead */
    /* enable the safe mode transition interrupt */
}

void initOutputClock(void) {
    CGM.OCDLS_SC.B.SELDIV = 2;       /* Output Clock enabled (to go to pin) */
    CGM.OCDLS_SC.B.SELCTL = 1;       /* Output Clock’s selected division is 2**2 = 4 */
    SIU.PCR[0].R = 0x0800;            /* MPC56xxB: assign port PA[0] pad to Alt Func 2 */
    /* CLKOUT = 16 MHz IRC/4 = 4MHz */
    CGM.OCDLS_SC.B.SELCTL = 0;       /* MPC56xxB: Assign output clock to XTAL */
    /* CLKOUT = Fxtal/4 = 2 or 10 MHz for 8 or 40 MHx XTAL*/
    CGM.OCDLS_SC.B.SELCTL = 2;       /* Assign output clock to FMPLL[0] */
    /* CLKOUT = 64 MHz/4 = 16MHz */
}

void disableWatchdog(void) {
    SWT.SR.R = 0x0000c520;          /* Write keys to clear soft lock bit */
    SWT.SR.R = 0x0000d928;
    SWT.CR.R = 0x8000010A;          /* Clear watchdog enable (WEN) */
}

void main (void) {
    uint32_t i = 0;                 /* Dummy idle counter */
    initModesAndClock();            /* Initialize mode entries and system clock */
    initOutputClock();              /* Initialize Output Clock to 16 M, XOSC, then PLL */
    disableWatchdog();              /* Disable watchdog */
    while (1) {
        i++;
    }
}
11.3.2 MPC56xxP with 40 MHz crystal

/* main.c - PLL example for MPC56xxP */
/* Description: Set sysclk to FMPLL0 running at 64 MHz & enable CLKOUT */
/* Jan 14 2009 S. Mihalik - Initial version */
/* May 11 2009 S. Mihalik - Simplified code */
/* Jun 24 2009 S. Mihalik - Simplified code */
/* Mar 10 2010 S Mihalik - Modified initModesAndClock & updated header file */
/* Copyright Freescale Semiconductor, Inc 2010 All rights reserved. */

#include "Pictus_Header_v1_09.h" /* Use proper include file */

void initModesAndClock(void) {
  ME.MER.R = 0x0000001D; /* Enable DRUN, RUN0, SAFE, RESET modes */
  /* Initialize PLL before turning it on: */
  /* Use 2 of the next 4 lines depending on crystal frequency: */
  /*CGM.CMU_0_CSR.R = 0x000000004;*/ /*Monitor FXOSC > FIRC/4 (4MHz); no PLL monitor*/
  /*CGM.FMPLL[0].CR.R = 0x02400100;*/ /* 8 MHz xtal: Set PLL0 to 64 MHz */
  CGM.CMU_0_CSR.R = 0x000000000; /* Monitor FXOSC > FIRC/1 (16MHz); no PLL monitor*/
  CGM.FMPLL[0].CR.R = 0x12400100; /* 40 MHz xtal: Set PLL0 to 64 MHz */
  ME.RUN[0].R = 0x001F0074; /* RUN0 cfg: 16MHzIRCON,OSC0ON,PLL0ON,syclk=PLL0*/
  /* Mode Transition to enter RUN0 mode: */
  ME.MCTL.R = 0x40005AF0; /* Enter RUN0 Mode & Key */
  ME.MCTL.R = 0x4000A50F; /* Enter RUN0 Mode & Inverted Key */
  while (ME.GS.B.S_MTRANS == 1) {} /* Wait for mode transition to complete */
  /* Notes: */
  /* 1. I_TC IRQ could be used here instead of polling */
  /* to allow software to complete other init. */
  /* 2. A timer could be used to prevent waiting forever*/
  while(ME.GS.B.S_CURRENTMODE != 4) {} /* Verify RUN0 is the current mode */
  /* Note: This verification ensures a SAFE mode */
  /* transition did not occur. SW could instead */
  /* enable the safe mode transition interrupt */
}

void initOutputClock(void) {
  CGM.OCEN.B.EN = 1; /* Output Clock enabled (to go to pin) */
  CGM.OCDSSC.B.SELDIV = 2; /* Output Clock’s selected division is 2**2 = 4 */
  CGM.OCDSSC.B.SELCTL = 0; /* MPC56xxP/S: Output clock select 16 MHz int RC osc */
  SIU.PCR[22].R = 0x0400; /* MPC56xxP: assign port PB[6] pad to Alt Func 1 */
  /* CLKOUT = 16 MHz IRC/4 = 4MHz */
  CGM.OCDSSC.B.SELCTL = 1; /* MPC56xxP/S: Assign output clock to XTAL */
  /* CLKOUT= Fxtal/4 = 2 or 10 MHz for 8 or 40 MHx XTAL*/
  CGM.OCDSSC.B.SELCTL = 2; /* Assign output clock to FMPLL[0] */
  /* CLKOUT = 64 MHz/4 = 16 MHz */
}

void disableWatchdog(void) {
  SWT.SR.R = 0x0000c520; /* Write keys to clear soft lock bit */
  SWT.CR.R = 0x0000d928; /* Clear watchdog enable (WEN) */
}

void main (void) {
  vuint32_t i = 0; /* Dummy idle counter */

  initModesAndClock(); /* Initialize mode entries and system clock */
  initOutputClock(); /* Initialize Output Clock to 16 M, XOSC, then PLL */
  disableWatchdog(); /* Disable watchdog */
  while (1) {
    i++;
  }
}
# MPC56xxS with 8 MHz crystal

```c
#include "56xxS_0204.h" /* Use proper include file */

void initModesAndClock(void) {
    ME.MER.R = 0x0000001D;          /* Enable DRUN, RUN0, SAFE, RESET modes */
    /* Initialize PLL before turning it on: */
    /* Use 1 of the next 2 lines depending on crystal frequency: */
    CGM.FMPLL[0].CR.R = 0x02400100; /* 8 MHz xtal: Set PLL0 to 64 MHz */
    ME.RUN[0].R = 0x001F0074;       /* RUN0 cfg: 16MHzIRC/CON, OSC0ON, PLL0ON, syclk=PLL0 */
    ME.PCTL[68].R = 0x01;           /* MPC56xxB/S: select ME.RUNPC[1] */
    ME.MCTL.R = 0x40005AF0;         /* Enter RUN0 Mode & Key */
    while (ME.GS.B.S_MTRANS == 1) {}    /* Wait for mode transition to complete */
    /* Notes: */
    /* 1. I_TC IRQ could be used here instead of polling */
    /* 2. A timer could be used to prevent waiting forever*/
    while(ME.GS.B.S_CURRENTMODE != 4) {} /* Verify RUN0 is the current mode */
    /* Note: This verification ensures a SAFE mode */
    /* transition did not occur. SW could instead */
    /* enable the safe mode transition interrupt */
}

void initOutputClock(void) {
    CGM.OC_EN.B.EN = 1;       /* Output Clock enabled (to go to pin) */
    CGM.OCDSC.SC.B.SELDIV = 2; /* Output Clock’s selected division is 2**2 = 4 */
    SIU.PCR[103].R = 0x0E00;  /* MPC56xxS: assign port PH[4] pad to Opt 3 (untested)*/
    /* CLKOUT = 16 MHz IRC/4 = 4MHz */
    CGM.OCDSC.SC.B.SELCTL = 1; /* MPC56xxP/S: Assign output clock to XTAL */
    /* CLKOUT = Fxtal/4 = 2 or 10 MHz for 8 or 40 MHz XTL */
    CGM.OCDSCSC.B.SELCTL = 2; /* Assign output clock to FMPLL[0] */
    /* CLKOUT = 64 MHz/4 = 4MHz */
}

void disableWatchdog(void) {
    SWT.SR.R = 0x00000020; /* Write keys to clear soft lock bit */
    SWT.SR.R = 0x000034928;
    SWT.CR.R = 0x8000010A; /* Clear watchdog enable (WEN) */
}

void main (void) {
    /* Dummy idle counter */
    uint32_t i = 0;
    initModesAndClock(); /* Initialize mode entries and system clock */
    initOutputClock();   /* Initialize Output Clock to 16 M, XOSC, then PLL */
    disableWatchdog();   /* Disable watchdog */
    while (1) {
        i++;
    }
}
```
12 FMPLL: Frequency Modulation

12.1 Description

Task: Assuming an 8 MHz crystal, initialize the FMPLL to provide an 80 MHz system clock, ramping up in two steps. Enable frequency modulation (FM), with a depth of 1% and a rate of sysclk/40.

This example tests the PLL for LOCK and FM status. The program simply waits for the desired status to occur. In real life a maximum timeout mechanism would be implemented. If LOCK was not achieved in a maximum time, then one might log an error message and reset the part. If FM status is not correct then software could loop back and try again, for a selected number of iterations.

This implementation is done using C code in the main function for illustration. Normally the PLL would be initialized in assembly language soon after reset, to speed up the rest of initialization. Increasing the PLL frequency produces a brief current demand from the power supply, which varies with the amount of frequency change. This example uses two frequency increases and therefore has lower immediate current increase than if it were done in one step. The second increase changes only the RFD, which does not cause change of lock because the divider is after the feedback path.

Exercise: Measure CLKOUT frequency. CLKOUT default frequency is one half of sysclk frequency ($F_{sys}$) as defined by SIU_ECCR[EBDF]. Initially CLKOUT will be 1/2 of the 12 MHz reset default frequency (163 ns). CLKOUT at $F_{sys} = 40$ MHz is 50 ns; at $F_{sys} = 80$ MHz is 25 ns.

![Diagram of FMPLL Example](image)

Table 46. Signals for FMPLL Example

<table>
<thead>
<tr>
<th>Signal</th>
<th>Function Name</th>
<th>SIU PCR No.</th>
<th>Package Pin No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLKOUT</td>
<td>CLKOUT</td>
<td>AF25, AE24, AA20</td>
<td>–</td>
</tr>
</tbody>
</table>

Figure 23. FMPLL Example

Qorivva Simple Cookbook, Rev. 4
12.2 Design

This example applies to MPC555x devices, but not MPC563x. The formula for system clock is:

\[
F_{\text{sys}} = F_{\text{ref}} \times \frac{(MFD + 4)}{((\text{PREDIV} + 1) \times 2^{\text{RFD}})}
\]

NOTE

When using Frequency Modulation, the system frequency plus the 1% or 2% depth must not exceed the device's rated maximum system frequency.

For an 8 MHz crystal \(F_{\text{ref}}\) and a target frequency of 80 MHz \(F_{\text{sys}}\), derive target values as shown below using the formula for \(F_{\text{sys}}\):

\[
\frac{F_{\text{sys}}}{F_{\text{ref}}} = \frac{80 \text{ MHz}}{8 \text{ MHz}} = \frac{10}{1} = \frac{(6 + 4)}{((0 + 1) \times 2^{0})} = \frac{(MFD + 4)}{((\text{PREDIV} + 1) \times 2^{\text{RFD}})}
\]

From this we can see that in this case, for \(F_{\text{sys}}\) the target values are: \(MFD = 6\), \(\text{PREDIV}=0\), \(\text{RFD} = 0\). Initially software will set the \(\text{RFD}\) to the next higher value (\(\text{RFD} = 1\)), then it is lowered in a second step to the target frequency. These values provide an \(\text{ICO}\) frequency \(f_{\text{ico}}\) of 80 MHz, which is within the specification of the MPC5554 Data Sheet, Rev 1.4.

Now the bit field values for Frequency Modulation can be determined. The percent depth \((P)\) is 1. For a target \(MFD\) of 6, \(M = 480\). The expected difference value \((\text{EXP})\) is calculated per the formula in the MPC5554 Reference Manual, section 11.4.4.1:

\[
\text{EXP} = \frac{(MFD + 4) \times M \times P}{100} = \frac{(6 + 4) \times 480 \times 1}{100} = 48
\]

Loss of lock and loss of clock detection is not enabled in this example. The PLL will not exit reset until it has locked. Because changing \(\text{PREDIV}\) or \(MFD\), or enabling FM, can cause loss of lock, the loss-of-lock circuitry and interrupts would not be enabled until after these steps.
12.3  Code

/* main.c - FMPLL example */
/* Rev 1.0 Sept 21 2004 S. Mihalik */
/* Copyright Freescale Semiconductor, Inc. 2004 All rights reserved. */
/* Notes: */
/* 1. ECC in L2SRAM is not initialized here; must be done by debug scripts */
/* 2. Cache is not used */
#include "mpc5554.h"

void errorLog (void){
   while (1) {}                        /* Placeholder for FMPLL error - wait forever */
}

void initFMPLL(void) {
   FMPLL.SYNCR.R = 0x03080030;         /* Initial setting: 40 MHz for 8 MHz crystal*/
   while (FMPLL.SYNSR.B.LOCK != 1) {}; /* Wait for LOCK = 1 */
   FMPLL.SYNCR.R = 0x03080430;          /* Enable FM */
   while (FMPLL.SYNSR.B.LOCK != 1) {}; /* Wait for FMPLL to LOCK again */
   if ((FMPLL.SYNSR.B.CALDONE != 1) | (FMPLL.SYNSR.B.CALPASS != 1)) {
      errorLog();              /* Error if calibration is not done or did not pass */
   }
   FMPLL.SYNCR.R = 0x03000430;          /* Final setting: 80 MHz for 8 MHz crystal */
}

void main (void){
   int i=0;          /* Dummy counter */
   initFMPLL();       /* Initialize FMPLL for 80 MHz & 1% Frequency Modulation depth */
   while (1) {i++; } /* Wait forever */
}
13  **Modes: Low Power** (MPC56xxB/S)

13.1  **Description**

**Task:** Using lowest power modes, pulse an output pin to a 10% “on” duty cycle approximately every second (~0.9 seconds off, then ~0.1 seconds on). Repeat until a pin transition occurs.

There are two wakeup event timer choices: Autonomous Periodic Interrupt (API) for multiple regular (periodic) short timeouts, and Real Time Counter (RTC), a single longer timeout. This example uses RTC.

In STANDBY mode, outputs other than wakeup inputs with enabled pullups are always in high impedance state. Therefore, STANDBY should be used for the longer of the two wakeup timeouts (WT1 in this example) with possibly either a pullup or pulldown resistor externally connected to achieve the desired ON or OFF output state such as to an LED.

After the first wakeup timeout (WT1), STANDBY mode exits and the processor goes to the RESET vector or 0x4000 0000 in SRAM if RGM_STDBY[BOOTFROM_BKP_RAM] is set. At this point, the processor is in DRUN mode and software should verify that the reason for entering DRUN was the desired wakeup event. Software then turns on the LED and sets up the new wakeup timeout (WT2).

However, because the new output state must be maintained at the pad, you cannot go back to STANDBY, because the pad would enter high impedance and be pulled up or down depending on the external resistor. Therefore STOP mode is used instead of STANDBY for this second shorter wake up timeout (WT2). Software will configure STOP mode to maintain the I/O pin states (using MC_STOP_MC[PDO] field), then enter STOP mode from a RUNx mode.

After the WT2 wakeup timeout, STOP mode exits back to the previous RUNx mode with the instruction pointer unchanged. Software here simply re-initializes the wakeup timer to the first value and then re-enters STANDBY mode. By entering STANDBY, the output goes to the high impedance state so the pad is pulled back up or down according to the circuit.

**Exercise:** Change timeouts for a new period and duty cycle.

**Figure 24. Modes and Wakeup Timeouts WT1 & WT2**
13.2 Design

13.2.1 Modes vs. Powered Domains vs. Modules

Hardware is used to control switching power on and off to primary power domains (PD0 and PD1) based on the mode being entered to conserve current. When a domain is powered off, then of course all initialization or other data is lost for registers and memory in that domain. Unlike power domains PD0 and PD1, power for PD2 is configurable by software writing to the PCU_PCONF2 register.

From the table below we can see that going to STANDBY mode cuts power to core, most peripherals, mode entry configurations, clocks, etc. Hence we need to re-initialize registers in PD0 and PD1 when exiting STANDBY mode, but not other modes such as STOP mode.

Table 48. Summary of Powered Modules by Mode and Domain for MPC56xxB/S Mode Entry Low Power example
(See chapters in reference manual on “Power Control Unit and Voltage Regulators” and “Power Supplies”)

<table>
<thead>
<tr>
<th>Mode</th>
<th>Power Domain PD0</th>
<th>Power Domain PD1</th>
<th>Power Domain PD2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Configuration Register PCU_PCONF0</td>
<td>Configuration Register PCU_PCONF1</td>
<td>Configuration Register PCU_PCONF2</td>
</tr>
<tr>
<td>Domain Modules: PCU, RGM, WKPU, SIRC, FIRC, CAN Sampler, 8K SRAM, ME_DRUN_MC register</td>
<td>Domain Modules: MC_ME1, MC_CGM, SIUL, Core, Flash, FMPLL, other peripherals</td>
<td>Domain Modules: Additional SRAM</td>
<td></td>
</tr>
<tr>
<td>DRUN</td>
<td>Powered</td>
<td>Powered</td>
<td>Configurable</td>
</tr>
<tr>
<td>RUN3</td>
<td>Powered</td>
<td>Powered</td>
<td>Configurable</td>
</tr>
<tr>
<td>STOP</td>
<td>Powered</td>
<td>Powered</td>
<td>Configurable</td>
</tr>
<tr>
<td>STANDBY</td>
<td>Powered</td>
<td>Not Powered</td>
<td>Configurable</td>
</tr>
</tbody>
</table>

1 ME module exception: ME_DRUN_MC register is powered during STANDBY.

13.2.2 RTC/API Clock Selection and Configuration

One of three clock sources below can be selected for the RTC counter clock source:

- 128 kHz SIRC (Slow Internal Reference Clock) with optional divider in CGM_SIRC_CTL
- 16 MHz FIRC (Fast Internal Reference Clock) with optional divider in CGM_FIRC_CTL
- 32 kHz SXOSC (“Slow” External Oscillator)
- MPC56xxS only: 4–16 MHz FXOSC (“Fast” External Oscillator)

The selected clock can be also divided by 512 and/or 32 in RTC_RTCC before clocking the 32-bit RTC counter. (See RTC/API block diagram in the device reference manual chapter, “Real Time Clock / Autonomous Periodic Interrupt.”)

There are two timeouts that can occur and each can generate a wakeup event or interrupt request. Each timeout compares a programmable value to the 32-bit RTC counter.

- RTCVAL: for longer timeouts, is always enabled when RTC counter is running
APIVAL: for shorter regular timeouts, must be enabled separately

The following table provides useful calculations for various clock sources and dividers for RTC and API timeouts. Once we know our wakeup timeout(s), we can use this table to decide on a clock source, whether to use API or RTC wakeup, and whether to use the 512 and/or 32 clock dividers.

Our target wakeup timeouts are WT1 = 0.9 seconds and WT2 = 0.1 seconds. We will use the 128 kHz SIRC divided by 4 (32 kHz) for a clock source without other dividers. Wakeup source is RTC, since the range of min to max RTC timeouts meets our criteria of 0.1 and 0.9 seconds and SIRC uses less power than FIRC.

Note that the RTC Counter Register (RTCCNT) only clears on Power On Reset. Other resets do not change its value. When using the RTC, set the RTC value in RTC_RTCC[RTCVAL] to its current RTCVAL plus an offset, where the offset is the next desired delay in terms of appropriate RTCCNT units.

**Table 49. RTC/API Clock, Divider, and Timeout Examples for MPC56xxB/S**

<table>
<thead>
<tr>
<th>Clock Source</th>
<th>div512</th>
<th>div32</th>
<th>RTC Counter Input Clock Frequency</th>
<th>RTC Counter Input Clock Period (1 / (RTC Counter Input Clock Freq.))</th>
<th>Min. API Timeout (1 × RTC Counter input clock period)</th>
<th>Max. API Timeout (2^{10} – 1 × RTC Counter input clock period)</th>
<th>Min. RTC Timeout (2^{10} × RTC Counter input clock period)</th>
<th>Max. RTC Timeout (2^{22} – 1 × RTC Counter input clock period)</th>
<th>RTC Rollover Timeout (2^{32} × RTC Counter input clock period)</th>
</tr>
</thead>
<tbody>
<tr>
<td>128 kHz SIRC (^1) with SIRDIV = 0</td>
<td>0 0</td>
<td>128 kHz</td>
<td>~7.8 μsec</td>
<td>~7.8 μsec</td>
<td>~ 8 msec</td>
<td>~ 8 msec</td>
<td>~31 sec</td>
<td>~8.9 hrs</td>
<td></td>
</tr>
<tr>
<td>0 1</td>
<td>4 kHz</td>
<td>250 μsec</td>
<td>250 μsec</td>
<td>~ 256 msec</td>
<td>256 msec</td>
<td>~17 min</td>
<td>~12 days</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1</td>
<td>7.8125 kHz</td>
<td>128 msec</td>
<td>128 msec</td>
<td>~131 sec</td>
<td>~131 sec</td>
<td>~6 days</td>
<td>~17 yrs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16 MHz FIRC (^2) (125 x SIRC)</td>
<td>0 0</td>
<td>16 MHz</td>
<td>62.5 nsec</td>
<td>62.5 nsec</td>
<td>~64 μsec</td>
<td>64 μsec</td>
<td>0.25 sec</td>
<td>~4.3 min</td>
<td></td>
</tr>
<tr>
<td>0 1</td>
<td>500 kHz</td>
<td>2 μsec</td>
<td>2 μsec</td>
<td>~2 msec</td>
<td>2.048 msec</td>
<td>8 sec</td>
<td>~2.8 hrs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0</td>
<td>31.25 kHz</td>
<td>32 μsec</td>
<td>32 μsec</td>
<td>~33 msec</td>
<td>~33 msec</td>
<td>~2.1 min</td>
<td>~1.5 days</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1</td>
<td>~977 Hz</td>
<td>1.024 msec</td>
<td>1.024 msec</td>
<td>~1048 sec</td>
<td>~1049 sec</td>
<td>~1.1 hrs</td>
<td>~1.6 mo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32 kHz SXOSC (^3) or 128 kHz SIRC/4 (^4)</td>
<td>0 0</td>
<td>32 kHz</td>
<td>31.25 μsec</td>
<td>31.25 μsec</td>
<td>~32 msec</td>
<td>32 msec</td>
<td>~2.1 min</td>
<td>~1.5 days</td>
<td></td>
</tr>
<tr>
<td>0 1</td>
<td>1 kHz</td>
<td>1 msec</td>
<td>1 msec</td>
<td>1.023 sec</td>
<td>1.024 sec</td>
<td>~1.1 hrs</td>
<td>~1.6 mo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0</td>
<td>~2 Hz</td>
<td>16 msec</td>
<td>16 msec</td>
<td>~16 sec</td>
<td>~16 sec</td>
<td>~18 hrs</td>
<td>~2.1 yrs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1</td>
<td>~2 Hz</td>
<td>512 msec</td>
<td>512 msec</td>
<td>~524 sec</td>
<td>~524 sec</td>
<td>~24 days</td>
<td>~67 yrs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FXOSC (^5)</td>
<td>0 0</td>
<td>8 MHz</td>
<td>125 nsec</td>
<td>125 nsec</td>
<td>~128 sec</td>
<td>~128 sec</td>
<td>0.5 sec</td>
<td>~8.6 min</td>
<td></td>
</tr>
<tr>
<td>0 1</td>
<td>250 kHz</td>
<td>4 μsec</td>
<td>4 μsec</td>
<td>~4.1 msec</td>
<td>~4.1 msec</td>
<td>16 sec</td>
<td>~4.6 hrs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0</td>
<td>15.625 kHz</td>
<td>64 μsec</td>
<td>64 μsec</td>
<td>~65 msec</td>
<td>~66 msec</td>
<td>~4.3 min</td>
<td>~3 days</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1</td>
<td>~488 Hz</td>
<td>2.048 msec</td>
<td>2.048 msec</td>
<td>~2.1 sec</td>
<td>~2.1 sec</td>
<td>~2.3 hr</td>
<td>~3.2 mo</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\(^1\) SIRC is divided by 4 to 32 kHz using reset default value in CGM_SIRC_CTL[SIRCDIV] for MPC56xxB, or GGM_LPRC_CTL[LPRCDIV] for MPC56xxS.

\(^2\) FIRC is divided by 1 using reset default value in CGM_RC_CTL[RCDIV] for MPC56xxB, MPC56xxS.

\(^3\) 32 kHz SXOSC not available in STANDBY mode.

\(^4\) SIRC is divided by 4 to 32 kHz using reset default value in CGM_SIRC_CTL[SIRCDIV] for MPC56xxB, or GGM_LPRC_CTL[LPRCDIV] for MPC56xxS.

\(^5\) FXOSC as RTC/API option available only in MPC56xxS. Also, not available in STANDBY mode.
13.2.3 Pin Transition Selection and Configuration

One or more pins can be selected for generating a wakeup per the following table.

Table 50. Wakeup Sources
(Not all ports are available in all packages — see device reference manual)

<table>
<thead>
<tr>
<th>Wakeup Number</th>
<th>MPC560xB</th>
<th>MPC560xS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Port</td>
<td>SIU_PCR</td>
</tr>
<tr>
<td>WKUP0</td>
<td>API</td>
<td>n.a.</td>
</tr>
<tr>
<td>WKUP1</td>
<td>RTC</td>
<td>n.a.</td>
</tr>
<tr>
<td>WKUP2</td>
<td>PA1</td>
<td>PCR1</td>
</tr>
<tr>
<td>WKUP3</td>
<td>PA2</td>
<td>PCR2</td>
</tr>
<tr>
<td>WKUP4</td>
<td>PB1</td>
<td>PCR17</td>
</tr>
<tr>
<td>WKUP5</td>
<td>PC11</td>
<td>PCR43</td>
</tr>
<tr>
<td>WKUP6</td>
<td>PE0</td>
<td>PCR64</td>
</tr>
<tr>
<td>WKUP7</td>
<td>PE9</td>
<td>PCR73</td>
</tr>
<tr>
<td>WKUP8</td>
<td>PB10</td>
<td>PCR26</td>
</tr>
<tr>
<td>WKUP9</td>
<td>PA4</td>
<td>PCR4</td>
</tr>
<tr>
<td>WKUP10</td>
<td>PA15</td>
<td>PCR15</td>
</tr>
<tr>
<td>WKUP11</td>
<td>PB3</td>
<td>PCR19</td>
</tr>
<tr>
<td>WKUP12</td>
<td>PC7</td>
<td>PCR39</td>
</tr>
<tr>
<td>WKUP13</td>
<td>PC9</td>
<td>PCR41</td>
</tr>
<tr>
<td>WKUP14</td>
<td>PE11</td>
<td>PCR75</td>
</tr>
<tr>
<td>WKUP15</td>
<td>PF11</td>
<td>PCR91</td>
</tr>
<tr>
<td>WKUP16</td>
<td>PF13</td>
<td>PCR93</td>
</tr>
<tr>
<td>WKUP17</td>
<td>PG3</td>
<td>PCR99</td>
</tr>
<tr>
<td>WKUP18</td>
<td>PG5</td>
<td>PCR101</td>
</tr>
<tr>
<td>WKUP19</td>
<td>PA0</td>
<td>PCR0</td>
</tr>
<tr>
<td>WKUP20</td>
<td>(none)</td>
<td>(none)</td>
</tr>
</tbody>
</table>

These ports can be configured for rising and/or falling edge detection. An analog filter can be enabled in WKUP_WIFER which prevents glitches on those pins. When the filter is enabled, input pulses less than 40 ns ($W_{FI}$) are filtered (rejected), greater than 1000 ns ($W_{NFI}$) are not filtered, and those with widths in between may or may not get filtered.\(^1\)

---

1. Reference: MPC5604BC Data Sheet, Rev. 5, November 2009, Table 12, “I/O Input DC electrical characteristics, WFI and WNFI parameters.”
13.2.4 **STANDBY Wakeup: Execute from Flash or RAM?**

When exiting STANDBY mode by pin transition or timer wakeup, there are two options for code execution depending on the configuration of RGM_STDBY[BOOT_FROM_BKP_RAM]:

- System boots from flash on STANDBY exit (reset default)
- System boots from backup RAM on STANDBY exit

Configuring system boot from backup RAM has two advantages:

1. Enables lower power consumption. Reason: the flash does not need to be fully powered. If booting from SRAM and writing C code, there needs to be an initialization (such as initializing the stack pointer and small data areas), which is part of a normal compiler initialization on reset. However, if booting from SRAM and only writing a small amount of code, this could be written in assembly and have a shorter boot sequence.
2. Faster startup time. Reason: flash does not need to be power sequenced.

The amount of power savings when booting from RAM and STANDBY wakeup varies by application. This example does not use the feature to boot from backup RAM on standby wakeup, but boots from flash after reset and BAM code execution.

13.2.5 **Reset and Wakeup Strategy Summary**

After reset, the processor is in DRUN mode and software must determine the cause of reset. This example is implemented using the RTC and takes the key actions below based on the following reset causes:

- **Reset event**: Configure RTC wakeup of WT1 duration and enter STANDBY mode (outputs are in high impedance in STANDBY mode).
- **Wakeup event**: Set an output pin, configure RTC wakeup of WT2 duration and enter STOP mode. On STOP mode exit (from RTC timeout of WT2), configure RTC wakeup of WT1 and enter STANDBY mode (outputs return to high impedance in STANDBY mode).
- **Pin wakeup event** (final wakeup for this example): wait forever (do not enter more low power modes).

Wakeup from STOP mode exits to the RUNx MODE that was used previously, so instruction execution resumes in that mode from the next instruction, without any reset.

Alternative implementations could be based on using the API. Two options are:

1. **STANDBY exit with API 1/10th second wakeup**
   
   Simply increment an API wakeup counter in SRAM that is kept alive (in Power domain 0). At the 9th wakeup, go into STOP mode with the desired pad powered. On the next API wakeup, reset the counter and go back to STANDBY.

2. **STANDBY exit with API 1 second wakeup**
   
   On API wakeup from STANDBY, power the desired pad, set up a 1/10th second PIT interrupt and execute a WAIT instruction (instead of going back to a low power mode). At the PIT interrupt go back to STANDBY and wait for STANDBY wakeup.
NOTE

Some MPC56xx device documentation list more than one STOP0, HALT0, or STANDBY0 mode. The devices for this example only have one such mode each, so the “0” suffix is not used in this write up, but must be used for coding register names if applicable.

NOTE

Although STANDBY exit causes a reset, not all registers are reset as in a power-up reset. Power domain 0 registers are not reset, including: RGM registers, ME_DRUN_MC register and PCU registers, RTC registers, CAN SAMPLER registers, etc.

13.2.6 Debugging with Low Power Modes

By default, low power modes shut off as much power as possible, including to the debugger connection. To avoid the debugger losing connection, and therefore possibly asserting a reset, the Nexus Development Interface (NDI) module contains features to maintain a debugger connection. Debuggers may or may not implement this feature.

The controls to maintain the debugger connection are in the Port Configuration Register (PCR) of the Nexus Development Interface module. These registers are not memory mapped, and are only accessible through the debugger. Users should use appropriate debugger scripts to configure these settings.

To maintain a debugger connection, the debugger should set the following:

- **PCR[LPDBG_EN]** Low Power Debug Enable: Enables debug functionality to support exit from any low power mode. Debug functionality is immediately available for breakpoints, etc. When enabled, low power exit takes longer because the state machine adds steps to wait until after a handshake to the debugger and the debugger response to the handshake completes.
- **PCR[MCKO_EN]** MCKO Enable: Enables clock which will be used in debugger connection.
- **PCR[MCKO_DIV]** MCKO Division Factor: Determines MCK0 relative to system clock frequency. Options are SYSCLK divided by 1 (default), 2, 4, or 8.

13.2.7 Mode Use

This example simply polls a status bit to wait for the targeted mode transition to complete. However, the status bit could instead be enabled to generate an interrupt request (assuming the INTC is initialized beforehand). This would allow software to complete other initialization tasks instead of brute force polling of the status bit.

It is normal to use a timer when waiting for a status bit to change. This example by default would have a watchdog timer expire if for some reason the mode transition never completes. One could also loop code on incrementing a software counter to some maximum value as a timeout. If a timeout was reached, then an error condition could be recorded in EEPROM or elsewhere.
Peripherals also have configurations to gate clocks on and off, enabling low power. The following table summarizes the peripheral configurations used here.

Table 51. Mode Configurations Summary for MPC56xxB/S Mode Entry Low Power Example

<table>
<thead>
<tr>
<th>Mode &amp; Main Use</th>
<th>Mode Config. Register</th>
<th>Mode Config. Register Value</th>
<th>Settings</th>
<th>Clock Sources</th>
<th>Memory Power Mode</th>
<th>Main Voltage Reg.</th>
<th>I/O Power Down Ctrl</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRUN</td>
<td>ME_DRUN_MC</td>
<td>0x001F 0010 (default)</td>
<td></td>
<td>16 MHz IRC</td>
<td>Normal</td>
<td>On</td>
<td>Off</td>
</tr>
<tr>
<td>RUN3: Wakeup</td>
<td>ME_RUN3_MC</td>
<td>0x001F 0010</td>
<td></td>
<td>16 MHz IRC</td>
<td>Pwr Dn</td>
<td>Off</td>
<td>Off</td>
</tr>
<tr>
<td>STOP: Hold Pin States:</td>
<td>ME_STOP_MC</td>
<td>0x0005 000F</td>
<td>Disabled</td>
<td>Off</td>
<td>Off</td>
<td>—</td>
<td>Pwr Dn</td>
</tr>
<tr>
<td>STANDBY0</td>
<td>ME_STANDBY_MC</td>
<td>0x0085 000F</td>
<td>Disabled</td>
<td>Off</td>
<td>—</td>
<td>—</td>
<td>Pwr Dn</td>
</tr>
</tbody>
</table>

Other modes are not used in example.

Table 52. Run Peripheral Configurations for MPC56xxB/S Mode Entry Low Power Example

<table>
<thead>
<tr>
<th>Peripheral Config. &amp; Main Use</th>
<th>Peri. Config. &amp; Register</th>
<th>Enabled Modes</th>
<th>Peripherals Selecting Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC7: Low Pwr Wakeup</td>
<td>ME_RUNPC_7</td>
<td>RUN3 0 RUN2 0 RUN1 0 RUN0 1 DRUN 0 SAFE 0 TEST 0 RESET 0</td>
<td>SIUL WKPU RTC/API 68 69 91</td>
</tr>
</tbody>
</table>

Other peripheral configurations are not used in example.

Table 53. Low Power Peripheral Configurations for MPC56xxB/S Mode Entry Low Power Example

<table>
<thead>
<tr>
<th>Peripheral Config.</th>
<th>Peri. Config. Register</th>
<th>Enabled Modes</th>
<th>Peripherals Selecting Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC7: Low Pwr I/O</td>
<td>ME_LPPC_7</td>
<td>STANDBY0 0 STOP 1 HALT 0</td>
<td>SIUL WKPU RTC/API 68 69 91</td>
</tr>
</tbody>
</table>

Other peripheral configurations are not used in example.
### 13.2.8 Steps and Pseudo Code

#### Table 54. Initial Steps for MPC56xxB/S Mode Entry Low Power Example

<table>
<thead>
<tr>
<th>Step</th>
<th>Relevant Bit Fields</th>
<th>Pseudo Code</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Disable Watchdog</strong></td>
<td>• Write keys to clear soft lock bit</td>
<td>SWT_SR = 0x0000 0C520</td>
</tr>
<tr>
<td></td>
<td>• Clear watchdog enable bit</td>
<td>SWT_SR = 0x0000 D928</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SWT_CR = 0x8000 010A</td>
</tr>
<tr>
<td><strong>Init Modes and sysclk</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Enable desired modes: DRUN, RUN3, STANDBY, STOP</td>
<td>ME_ME = 0x0000 248D</td>
</tr>
<tr>
<td></td>
<td>Configure DRUN, RUN3 Modes:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• I/O Output power-down: outputs not disabled</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Main Voltage regulator is on</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Data, code flash in normal mode</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• PLL0 is switched off</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Crystal oscillator is switched off</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• 16 MHz IRC is switched on (default)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Select FIRC (16 MHz) as sysclk</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PDO=0</td>
<td>ME_DRUN_MC = 0x001F 0010</td>
</tr>
<tr>
<td></td>
<td>MVRON=1</td>
<td>ME_RUN3_MC = 0x001F 0010</td>
</tr>
<tr>
<td></td>
<td>DFLOAN = 3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CFLAON = 3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PLL0ON=0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>XOSC0ON=0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>16MHz_IRCON=1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SYSCLK=0x0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ME_RUN_PC7 = 0x0000 0088</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ME_LP_PC7 = 0x0000 0400</td>
<td></td>
</tr>
<tr>
<td><strong>Configure STOP Mode:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• I/O Output power-down: outputs not disabled</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Main Voltage regulator is off</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Data, code flash in power-down mode</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• 16 MHz IRC is switched off</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• sysclk is disabled</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PDO=0</td>
<td>ME_STOP0_MC = 0x0005 000F</td>
</tr>
<tr>
<td></td>
<td>MVRON=0</td>
<td>ME_STOP_MC = 0x0005 000F</td>
</tr>
<tr>
<td></td>
<td>DFLOAN=1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CFLAON=1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>16MHz_IRCON=0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SYSCLK=0xF</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ME_STOP_MC = 0x0085 0000F</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ME_STOP_MC = 0x0085 0000F</td>
<td></td>
</tr>
<tr>
<td><strong>Configure STANDBY Mode:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Main Voltage regulator is (always) off</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• 16 MHz IRC is switched off</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• sysclk is (always) disabled</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MVRON=0</td>
<td>ME_STANDBY0_MC = 0x0005 000F</td>
</tr>
<tr>
<td></td>
<td>16MHz_IRCON=0</td>
<td>ME_STANDBY_MC = 0x0005 000F</td>
</tr>
<tr>
<td></td>
<td>SYSCLK=0xF</td>
<td></td>
</tr>
<tr>
<td><strong>Peripheral Configurations:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Run Peri. Cfg. 7: run in DRUN, RUN3 modes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Low Pwr Peri. Cfg. 7: run in STOP</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DRUN, RUN3 = 1</td>
<td>ME_RUN_MC = 0x0000 0088</td>
</tr>
<tr>
<td></td>
<td>STOP = 1</td>
<td>ME_LP_PC7 = 0x0000 0400</td>
</tr>
<tr>
<td><strong>Assign peripheral configuration to peripherals:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• SIUL: select ME_RUN_PC7, ME_LP_PC7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• WKPU: select ME_RUN_PC7, ME_LP_PC7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• RTC/API: select ME_RUN_PC7, ME_LP_PC7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RUN_CFG = 7</td>
<td>ME_PCTL68 = 0x3F</td>
</tr>
<tr>
<td></td>
<td>LP_CFG = 7</td>
<td>ME_PCTL69 = 0x3F</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ME_PCTL91 = 0x3F</td>
</tr>
<tr>
<td><strong>If necessary, wait for mode transition from STANDBY to complete:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>S_MTRANS</td>
<td>while MS_GS[S_MTRANS] = 1</td>
</tr>
<tr>
<td><strong>Initiate software mode transition to same mode to enable PCTLs, RUN_PC, LP_PC to latch</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Mode and key</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Mode and inverted key</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Wait for mode transition complete status flag</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• NOTE: if transition does not complete, check status flags such as ME_GS[XOSC]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Verify desired target mode was entered</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TARGETMODE = RUN3</td>
<td>ME_MCTRL = 0x7000 5AF0</td>
</tr>
<tr>
<td></td>
<td>S_MTRANS</td>
<td>ME_MCTRL = 0x7000 A50F</td>
</tr>
<tr>
<td></td>
<td></td>
<td>wait for ME_GS[S_MTRANS] = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>verify ME_GS[S_CURRENT_MODE] = RUN3</td>
</tr>
</tbody>
</table>
### Table 54. Initial Steps for MPC56xxB/S Mode Entry Low Power Example (continued)

<table>
<thead>
<tr>
<th>Step</th>
<th>Relevant Bit Fields</th>
<th>Pseudo Code</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>MPC56xxB</td>
</tr>
<tr>
<td>Reset Source Determination and Action</td>
<td>If destructive reset event, call function: ResetEvent</td>
<td>RGM_DES flags</td>
</tr>
<tr>
<td></td>
<td>If functional reset event, call function: ResetEvent</td>
<td>RGM_FES flags</td>
</tr>
<tr>
<td></td>
<td>If RTC wakeup, call function: RTCWakeup function</td>
<td>56xxB: EIF1 56xxS: EIF20</td>
</tr>
<tr>
<td></td>
<td>If pin transition wakeup from: MPC56xxB: PE[0] MPC56xxS: PA[0] wait forever</td>
<td>56xxB: EIF6 56xxS: EIF0</td>
</tr>
<tr>
<td></td>
<td>else wait forever (should never get here!)</td>
<td></td>
</tr>
</tbody>
</table>
### Table 55. ResetEvent Function Steps for MPC56xxB/S Mode Entry Low Power Example

<table>
<thead>
<tr>
<th>Step</th>
<th>Relevant Bit Fields</th>
<th>Pseudo Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Init Wakeup Events</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Initialize SIRC clock used for wakeup: divide 128 kHz clock by 4</td>
<td>SIRCON_STDBY = 1 SIRCDIV = 3</td>
<td>MPC56xxB: CGM_SIRC_CTL = 0x0000 0301 MPC56xxS: CGM_LP0RC_CTL = 0x0000 0300</td>
</tr>
<tr>
<td>Clear RTC and enable writing to RTCVAL or APIVAL</td>
<td>CNTEN = 0</td>
<td>RTC_RTCC = 0x0000 0000</td>
</tr>
<tr>
<td>Initialize RTC wakeup timer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Enable RTC counter</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Enable counter freeze during debug</td>
<td>CNTEN = 1</td>
<td>RTC_RTCC = 0xA01B 1000</td>
</tr>
<tr>
<td>• Select clock source 128 kHz SIRC / 4</td>
<td>CLKSEL = 1</td>
<td></td>
</tr>
<tr>
<td>• Initialize initial RTCVAL to WT1 of 0.9 sec</td>
<td>RTCVAL = 27</td>
<td></td>
</tr>
<tr>
<td>Enable Wakeup Rising Edge Event from RTC and:</td>
<td>WKUP_WIREER = 0x0000 0002 WKUP_WIREER = 0x0010 0001</td>
<td></td>
</tr>
<tr>
<td>• MPC56xxB: Pin PE[0]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• MPC56xxS: Pin PA[0]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enable analog filter for pad:</td>
<td>WKUP_WIFER = 0x0000 0002 WKUP_WIFER = 0x0000 0001</td>
<td></td>
</tr>
<tr>
<td>• MPC56xxB: Pin PE[0]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• MPC56xxS: Pin PA[0]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enable wakeup event rom RTC and:</td>
<td>WKUP_WRER = 0x0000 0002 WKUP_WRER = 0x0010 0001</td>
<td></td>
</tr>
<tr>
<td>• MPC56xxB: Pin PE[0]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• MPC56xxS: Pin PA[0]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enable WKUP pin pullups to minimize leakage</td>
<td>WKUP_WIPUER = 0x0000 FFFF WKUP_WIPUER = 0x001F FFFF</td>
<td></td>
</tr>
<tr>
<td>Clear flags</td>
<td>WKUP_WISR = 0x0000 FFFF WKUP_WISR = 0x0001 FFFF</td>
<td></td>
</tr>
<tr>
<td>Clear reset's destructive &amp; functional event flags</td>
<td>clear RGM_DES &amp; RGM_FES flags</td>
<td></td>
</tr>
<tr>
<td>Transision to STANDBY</td>
<td>Target_MODE = STANDBY</td>
<td></td>
</tr>
<tr>
<td>Initiate software mode transition to STANDBY mode</td>
<td>ME_MCTL = 0xD000 5AF0 ME_MCTL = 0xD000 A50F</td>
<td></td>
</tr>
<tr>
<td>• Mode and key</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Mode and inverted key</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wait for low power mode transition to complete Note: Alternative is to implement a timeout interrupt during a mode instruction, then use WAIT instruction</td>
<td>while 1</td>
<td></td>
</tr>
</tbody>
</table>

Processor now in STANDBY mode
Table 56. RTC Wakeup Event Function Steps for MPC56xxB/S Mode Entry Low Power Example

<table>
<thead>
<tr>
<th>Step</th>
<th>Relevant Bit Fields</th>
<th>Pseudo Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pad config.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Configure pad GPIO68 to output 0 to turn on LED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• GPIO68 on 56xxB EVB: LED 1, port PE[4]</td>
<td>SIU_GPDO[68] = 0</td>
<td></td>
</tr>
<tr>
<td>• GPIO68 on 56xxS EVB: LED 3, port PE[6]</td>
<td>SIU_PCR[68] = 0x0200</td>
<td></td>
</tr>
<tr>
<td>Configure pad as input (to be used for wakeup):</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• MPC56xxB: Port E[0]</td>
<td>SIU_PCR[64] = 0x0103</td>
<td></td>
</tr>
<tr>
<td>• MPC56xxS: Port PA[0]. Can be wired to EVB key</td>
<td>SIU_PCR[0] = 0x0103</td>
<td></td>
</tr>
<tr>
<td>Re-init RTC wakeup</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clear RTC and enable writing to RTCVAL or APIVAL.</td>
<td>CNTEN = 0</td>
<td></td>
</tr>
<tr>
<td>Initialize RTC wakeup timer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Enable RTC counter</td>
<td>CNTEN = 1</td>
<td></td>
</tr>
<tr>
<td>• Enable counter freeze during debug</td>
<td>FRZEN = 1</td>
<td></td>
</tr>
<tr>
<td>• Select clock source 128 kHz SIRC / 4</td>
<td>CLKSEL = 1</td>
<td></td>
</tr>
<tr>
<td>• Initialize initial RTCVAL to WT1 of 0.9sec</td>
<td>RTCVAL = 0x1B</td>
<td></td>
</tr>
<tr>
<td>RTCVAL = 900 msec x 1 count/33 msec = ~27</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Re-init RTC wakeup</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clear RTC and enable writing to RTCVAL or APIVAL</td>
<td>CNTEN = 0</td>
<td></td>
</tr>
<tr>
<td>Initialize RTC wakeup timer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Enable RTC counter</td>
<td>CNTEN = 1</td>
<td></td>
</tr>
<tr>
<td>• Enable counter freeze during debug</td>
<td>FRZEN = 1</td>
<td></td>
</tr>
<tr>
<td>• Select clock source 128 kHz SIRC / 4</td>
<td>CLKSEL = 1</td>
<td></td>
</tr>
<tr>
<td>• Initialize initial RTCVAL to WT1 of 0.9sec</td>
<td>RTCVAL = 10</td>
<td></td>
</tr>
<tr>
<td>RTCVAL = 900 msec x 1 count/33 msec = ~10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clear flag</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clear RTC wakeup event flag (write 1 to clear)</td>
<td>WKUP_WISR = 0x0000 0002</td>
<td></td>
</tr>
<tr>
<td>Transition to STOP mode</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Initiate software mode transition to STOP mode</td>
<td>TARGET_MODE = STOP</td>
<td></td>
</tr>
<tr>
<td>• Mode and key</td>
<td>ME_MCTL = 0xA000 5AF0</td>
<td></td>
</tr>
<tr>
<td>• Mode and inverted key</td>
<td>ME_MCTL = 0xA000 A50F</td>
<td></td>
</tr>
<tr>
<td>• Wait for STOP mode transition to complete (could have timeout interrupt for timeout.)</td>
<td>Wait for ME_GS[S_MTRANS]</td>
<td></td>
</tr>
<tr>
<td>STOP mode active. Code continues on wakeup from STOP mode back into same RUN3 mode.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Verify RUN mode</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Verify mode transition from STOP mode completed back to RUN3 mode</td>
<td>Verify ME_GS[CURRENTMODE] = RUN3</td>
<td></td>
</tr>
<tr>
<td>Re-init RTC wakeup</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clear RTC and enable writing to RTCVAL or APIVAL</td>
<td>CNTEN = 0</td>
<td></td>
</tr>
<tr>
<td>Initialize RTC wakeup timer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Enable RTC counter</td>
<td>CNTEN = 1</td>
<td></td>
</tr>
<tr>
<td>• Enable counter freeze during debug</td>
<td>FRZEN = 1</td>
<td></td>
</tr>
<tr>
<td>• Select clock source 128 kHz SIRC / 4</td>
<td>CLKSEL = 1</td>
<td></td>
</tr>
<tr>
<td>• Initialize initial RTCVAL to WT1 of 0.9sec</td>
<td>RTCVAL = 10</td>
<td></td>
</tr>
<tr>
<td>RTCVAL = 900 msec x 1 count/33 msec = ~10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clear flag</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clear RTC wakeup event flag</td>
<td>WKUP_WISR = 0x0000 0002</td>
<td></td>
</tr>
<tr>
<td>Transition to STANDBY mode</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Initiate software mode transition to STANDBY mode</td>
<td>ME_MCTL = 0xD000 5AF0</td>
<td></td>
</tr>
<tr>
<td>• Mode and key</td>
<td>ME_MCTL = 0xD000 A50F</td>
<td></td>
</tr>
<tr>
<td>• Mode and inverted key</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wait to enter STANDBY (could have timeout)</td>
<td>while 1</td>
<td></td>
</tr>
<tr>
<td>STANDBY mode active. On exit, reset vector is taken.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

¹ After RTC wakeup timeout, software outputs a logic signal to turn on an Evaluation Board (EVB) LED, configures next wakeup timeout, and transitions to STOP mode. Upon wakeup timeout from STOP mode, the processor configures the next timeout, then transitions to STANDBY mode.
13.3 Code

The following includes the main.c code used in the example, plus two preliminary standalone test programs: one for testing only RTC, and the other for RTC with STOP mode.

13.3.1 main.c: Complete Example (MPC56xxB)

```c
/* main.c - Modes: Low Power example for MPC56xxB/S */
/* Description: Enters STANDBY, STOP modes and exits at timeout */
/* Mar 13 2010 S Mihalik - initial version */
/* Apr 07 2010 S Mihalik - Changed STOP mode config MVRON=0, not 1 */
/* Copyright Freescale Semiconductor, Inc 2010 All rights reserved. */

#include "MPC5604B_0M27V_0101.h"          /* Use proper header file */

void disableWatchdog();        /* Prototypes */
void ResetEvent();
void RTCWakeupEvent();

void main (void) {
  uint16_t ResetCause16 =0U;   /* Placeholder for reset cause status bits */

disableWatchdog();            /* Disable watchdog */
  ME.MER.R = 0x0000248D;       /* Enable RUN3, STANDBY, STOP, DRUN, other modes */
  ME.DRUN.R = 0x001F0010;       /* DRUN cfg: 16MHIRCON=1, syclk=16 MHz FIRC */
  ME.RUN[3].R = 0x001F0010;     /* RUN3 cfg: 16MHIRCON=1, syclk=16 MHz FIRC */
  ME.STOP0.R = 0x0015000F;     /* 56xxB STOP: FIRCON=0, flash LP, no sysclk*/
  ME.STANDBY0.R = 0x0085000F;  /* 56xxB STANDBY cfg: FIRCON = 0 */
  ME.RUNPC[7].R = 0x000000088; /* Run Peri. Cfg 7 settings: run in DRUN, RUN3 modes*/
  ME.LPPC[7].R = 0x000000400;  /* LP Peri. Cfg. 7 settings: run in STOP */
  ME.PCTL[68].R = 0x03F;       /* MPC56xxB/S SIU: select ME.RUNPC[7], ME.LPPC[7] */
  ME.PCTL[69].R = 0x03F;       /* MPC56xxB/S WKPU: select ME.RUNPC[7], ME.LPPC[7] */
  ME.PCTL[91].R = 0x03F;       /* MPC56xxB/S RTC/API: select ME.RUNPC[7], ME.LPPC[7] */

while (ME.GS.B.S_MTRANS) {}    /* Ensure any STANDBY to DRUN mode trans. completed*/
  ME.MCTRL.R = 0x70005AF0;     /* Enter RUN3 Mode & Key */
  ME.MCTRL.R = 0x7000A50F;     /* Enter RUN3 Mode & Inverted Key */
while (ME.GS.B.S_MTRANS) {}    /* Wait for RUN3 mode transition to complete */
  /* Note: could wait here using timer and/or I_TC IRQ */
while(ME.GS.B.S_CURRENTMODE != 7) {} /* Verify RUN3 (0x7) is the current mode */

ResetCause16 = RGM.DES.R;       /* Test for destructive reset event: */
  if (((uint16_t)ResetCause16 != 0) {  /* If destructive reset event */
    ResetEvent();                     /*   and execute ResetEvent function */
  }

ResetCause16 = RGM.FES.R;       /* Test for functional reset event: */
  if (((uint16_t)ResetCause16 != 0) { /* If functional reset event */
    ResetEvent();                    /*   and execute ResetEvent function */
  }

if (WKUP.WISR.R && 0x00000002) { /* MPC56xxB: If wake up event RTC causing reset*/
  RTCWakeupEvent();              /*   execute RTCWakeupEvent function */
}

if (WKUP.WISR.R && 0x000000040) { /*MPC56xxB: If wake up event PE[6] causing reset*/
  while(1);                      /*   wait forever */
}

while (1) { }                   /* else wait forever */
```

Qorivva Simple Cookbook, Rev. 4

Freescale Semiconductor 121
void ResetEvent(void) {
    CGM.SIRC_CTL.R = 0x00000301; /* MPC56xxB: Div. SIRC by 4, turn SIRC on in STANDBY */
    RTC.RTCC.R = 0x00000000; /* Clear CNTEN to reset RTC */
    RTC.RTCC.R = 0xA01B1000; /* CLKSEL=SIRC div. by 4, FRZEN=CNTEN=1, RTCVAL=27 */
    WKUP.WIREER.R = 0x00000042; /* MPC56xxB: Enable rising edge events on RTC, PE[0] */
    WKUP.WIFER.R = 0x00000040; /* MPC56xxB: Enable analog filters - , PE[0] */
    WKUP.WIFER.R = 0x00000042; /* MPC56xxB: Enable wakeup events for RTC, PE[0] */
    WKUP.WISR.R = 0x0000FFFF; /* MPC56xxB: Clear all wake up flags */
    RGM.DES.R = 0xFFFF; /* Clear destructive reset flags */
    RGM.FES.R = 0xFFFF; /* & clear functional reset flags */
    ME.MCTL.R = 0x0D0005AF0; /* Enter STANDBY mode and key */
    ME.MCTL.R = 0x0D000A50F; /* Enter STANDBY mode and inverted key */
    while (1) {} /* Wait to enter STANDBY mode (could use timeout) */
    /* ENTER STANDBY MODE. ON STANDBY EXIT, RESET VECTOR IS TAKEN IN THIS EXAMPLE */
}

void RTCWakeupEvent(void) {
    SIU.PCR[68].R = 0x0200; /* MPC56xxB/S EVB LED: enable as output */
    SIU.PCR[64].R = 0x0103; /* MPC56xxB: Cfg PE[0] input = KEY 1 on MPC56xxB EVB*/
    SIU.GPDO[68].R = 0; /* MPC56xxB/S EVB LED: data output: LED on */
    RTC.RTCC.R = 0x00000000; /* Clear CNTEN to reset RTC, enable reloading RTCVAL */
    RTC.RTCC.R = 0xA0031000; /* CLKSEL=SIRC div. by 4, FRZEN=CNTEN=1, RTCVAL=3 */
    WKUP.WISR.R = 0x00000002; /* MPC56xxB: Clear wake up flag RTC */
    ME.MCTL.R = 0xA0005AF0; /* Enter STOP mode and key */
    ME.MCTL.R = 0xA000A50F; /* Enter STOP mode and inverted key */
    while (ME.GS.B.S_MTRANS) {} /* Wait for STOP mode transition to complete */
    /* STOP HERE FOR STOP MODE! ON STOP MODE EXIT, CODE CONTINUES HERE: */
    while (ME.GS.B.S_CURRENTMODE != 7) {} /* Verify RUN3 (0x7) is still current mode */
    RTC.RTCC.R = 0xD0000000; /* Clear CNTEN to reset RTC, enable reloading RTCVAL */
    RTC.RTCC.R = 0xA01B1000; /* CLKSEL=SIRC div. by 4, FRZEN=CNTEN=1, RTCVAL=27 */
    WKUP.WISR.R = 0x00000002; /* MPC56xxB: Clear wake up flag RTC */
    ME.MCTL.R = 0xD0005AF0; /* Enter STANDBY mode and key */
    ME.MCTL.R = 0xD000A50F; /* Enter STANDBY mode and inverted key */
    while (1) {} /* Wait to enter STANDBY mode (could use timeout) */
    /* ENTER STANDBY MODE. ON STANDBY EXIT, RESET VECTOR IS TAKEN IN THIS EXAMPLE */
}

void disableWatchdog(void) {
    SWT.SR.R = 0x0000c520; /* Write keys to clear soft lock bit */
    SWT.SR.R = 0x0000d928;
    SWT.CR.R = 0x8000010A; /* Clear watchdog enable (WEN) */
}

Qorivva Simple Cookbook, Rev. 4
13.3.2 Test Program 1: Exercise RTC Function only (MPC56xxB)

/* main.c - Modes: Low Power example for MPC56xxB/S - RTC FUNCTION ONLY */
/* Description: Toggles output based on RTC timeouts */
/* NOTE: RTCNT only clears on power up reset, not debugger reset or RESET input */
/* Mar 12 2010 S Mihalik - Initial version */
/* Copyright Freescale Semiconductor, Inc 2010 All rights reserved. */

#include "MPC5604B_0M27V_0101.h" // Use proper header file

void disableWatchdog(); /* Prototypes */

uint32_t wakeupCtr = 0; /* RTC timeouts */

void main (void) {
    disableWatchdog(); /* Disable watchdog */
    ME.MER.R = 0x0000248D; /* Enable RUN3, STANDBY, STOP, DRUN, other modes */
    ME.DRUN.R = 0x001F0010; /* DRUN cfg: 16MHIRC=1, syclk=16 MHz FIRC */
    ME.RUN[3].R = 0x001F0010; /* RUN3 cfg: 16MHIRC, syclk=16 MHz FIRC */
    ME.RUNPC[7].R = 0x00000088; /* Run PeriCfg 7 settings: run in DRUN, RUN3 modes */
    ME.PCTRL[68].R = 0x03F; /* MPC56xxB/S SIU: select ME.RUNPC[7], ME.LPPC[7] */
    ME.PCTRL[91].R = 0x03F; /* MPC56xxB/S RTC/API: select ME.RUNPC[7], ME.LPPC[7] */
    ME.MCTL.R = 0x70005AF0; /* Mode Transition to enter RUN3 mode: */
    ME.MCTL.R = 0x7000A50F; /* Enter RUN3 Mode & Key */
    while (ME.GS.B.S_MTRANS) {} /* Wait for RUN3 mode transition to complete */
    while (ME.GS.B.S_CURRENTMODE != 7) {} /* Note: could wait here using timer and/or I_TC IRQ*/
    SIU.GPDO[68].R = 1; /* MPC56xxB/S EVB LED: data output: LED off */
    SIU.PCR[68].R = 0x0200; /* MPC56xxB/S EVB LED: enable as output */
    CGM.SIRC_CTL.R = 0x00000301; /* MPC56xxB: Div. SIRC by 4 & turn SIRC on in STANDBY*/
    RTC.RTCC.R = 0x00000000; /* Clear CNTEN to reset RTC (counter) */
    RTC.RTCC.R = 0xA01B1000; /* CLKSEL=SIRC (div. by 4), FRZEN=CNTEN=1, RTCVAL=27*/
    while (1) {
        while (RTC.RTCS.B.RTCF == 0) {} /* Wait for RTC timeout */
        wakeupCtr++; /* Increment wakeuup counter */
        SIU.GPDO[68].R = 0; /* MPC56xxB/S EVB LED: data output: LED on */
        RTC.RTCC.R = 0x00000000; /* Clear CNTEN to reset RTC, enable reloading RTCVAL*/
        RTC.RTCC.R = 0xA0031000; /* CLKSEL=SIRC div. by 4, FRZEN=CNTEN=1, RTCVAL=3 */
        RTC.RTCS.R = 0x20000000; /* Clear RTC flag */
        while (RTC.RTCS.B.RTCF == 0) {} /* Wait for RTC timeout */
        SIU.GPDO[68].R = 1; /* MPC56xxB/S EVB LED: data output: LED off */
        RTC.RTCC.R = 0x00000000; /* Clear CNTEN to reset RTC, enable reloading RTCVAL*/
        RTC.RTCC.R = 0xA01B1000; /* CLKSEL=SIRC div. by 4, FRZEN=CNTEN=1, RTCVAL=27*/
        RTC.RTCS.R = 0x20000000; /* Clear RTC flag */
    }
}

void disableWatchdog(void) {
    SWT.SR.R = 0x0000c520; /* Write keys to clear soft lock bit */
    SWT.SR.R = 0x0000d928;
    SWT.CR.R = 0x8000010A; /* Clear watchdog enable (WEN) */
}
13.3.3 Test Program 2: Exercise RTC Wakeup in STOP mode (MPC56xxB)

/* main.c - Modes: Low Power example for MPC56xxB/S - RTC, WAKEUP, STOP MODE ONLY */
/* Description: Toggles output on RTC timeouts */
/* NOTE: RTCCNT only clears on power up reset, not debugger reset or RESET input */
/* Mar 13 2010 S Mihalik - Initial version */
/* Copyright Freescale Semiconductor, Inc 2010 All rights reserved. */

#include "MPC5604B_0M27V_0101.h"    /* Use proper header file */

void disableWatchdog();   /* Prototypes */
uint32_t wakeupCtr = 0;    /* wake up counter incremented on STANDBY exit */
uint32_t temp = 0;

void main (void) {
  disableWatchdog();       /* Disable watchdog */
  ME.MER.R = 0x0000248D;  /* Enable RUN3, STANDBY, STOP, DRUN, other modes */
  ME.DRUN.R = 0x00010010; /* DRUN cfg: 16MHIRCON=1, syclk=16 MHz FIRC */
  ME.RUN[3].R = 0x00010010; /* RUN3 cfg: 16MHIRCON, syclk=16 MHz FIRC */
  ME.STOP0.R = 0x0015001F; /* 56xxB STOP cfg: FIRCON=MVRON=1, flashLP, no sysclk*/
  ME.RUNPC[7].R = 0x00000000; /* Run Peri. Cfg 7 settings: run in DRUN, RUN3 modes*/
  ME.LPPC[7].R = 0x00000000; /* LP Peri. Cfg 7 settings: run in STOP */
  ME.PCTL[68].R = 0x03F;    /* MPC56xxB/S SIU: select ME.RUNPC[7], ME>LPPC[7]*/
  ME.PCTL[69].R = 0x03F;    /* MPC56xxB/S WKPU: select ME.RUNPC[7], ME>LPPC[7]*/
  ME.PCTL[91].R = 0x03F;    /* MPC56xxB/S RTC/API: select ME.RUNPC[7], ME>LPPC[7]*/
  ME.MCTL.R = 0x70005AF0;  /* Enter RUN3 Mode & Key */
  ME.MCTL.R = 0x700005AF0; /* Enter RUN3 Mode & Inverted Key */
  while (ME.GS.B.S_CURRENTMODE != 7) {} /* Note: could wait here using timer and/or I_TC IRQ*/
  while (ME.GS.B.S_CURRENTMODE != 7) {} /* Verify RUN3 (0x7) is the current mode */

  CGM.SIRC_CTL.R = 0x00000301; /* MPC56xxB: Div SIRC by 4 & turn SIRC on in STANDBY*/
  RTC.RTCC.R = 0x00000000;    /* Clear CNTEN to reset RTC (counter) */
  RTC.RTCC.R = 0xA01B000;     /* CLKSEL=SIRC div. by 4, FRZEN=CNTEN=1, RTCVAL=27 */
  WKUP.WIREER.R = 0x00000000; /* MPC56xxB: Enable rising edge events on RTC */
  WKUP.WIFER.R = 0x00000000;  /* MPC56xxB: Enable analog filters - none */
  WKUP.WIFER.R = 0x00000000;  /* MPC56xxB: Enable wakeup events for RTC */
  ME.MCTL.R = 0xA0005AF0;    /* Enter STOP mode and key */
  ME.MCTL.R = 0xA00005AF0;   /* Enter STOP mode and inverted key */
  while (ME.GS.B.S_CURRENTMODE != 7) {} /* Wait for RUN3 mode transition to complete */
  while (ME.GS.B.S_CURRENTMODE != 7) {} /* Verify RUN3 (0x7) is the current mode */
  while (1) {
    while (ME.GS.B_S_MTRANS) {} /* Verify RUN3 mode transition to complete */
    while (ME.GS.B_S_MTRANS) {} /* Verify RUN3 (0x7) is the current mode */
    wakeupCtr++;               /* Increment wakeup counter */
    SIU.GPDO[68].R = 0;        /* MPC56xxB/S EVB LED: data output: LED on */
    SIU.PCR[68].R = 0x0200;    /* MPC56xxB/S EVB LED: enable as output */
    RTC.RTCC.R = 0x00000000;   /* Clear CNTEN to reset RTC, enable reloading RTCVAL*/
    RTC.RTCC.R = 0xA0031000;   /* CLKSEL=SIRC div. by 4, FRZEN=CNTEN=1, RTCVAL=3 */
    WKUP.WISR.R = 0x00000000;  /* MPC56xxB: Clear wake up flag RTC */
    ME.MCTL.R = 0xA0005AF0;    /* Enter STOP mode and key */
    ME.MCTL.R = 0xA00005AF0;   /* Enter STOP mode and inverted key */
    while (ME.GS.B_S_MTRANS) {} /* Wait for STOP mode transition to complete */
  } /* STOP HERE FOR STOP MODE! ON STOP MODE EXIT, CODE CONTINUES HERE!*/
while (1) {
  wakeupCtr++; /* Increment wakeup counter */
  SIU.GPDO[68].R = 0; /* MPC56xxB/S EVB LED: data output: LED on */
  SIU.PCR[68].R = 0x0200; /* MPC56xxB/S EVB LED: enable as output */
  RTC.RTCC.R = 0x00000000; /* Clear CNTEN to reset RTC, enable reloading RTCVAL*/
  RTC.RTCC.R = 0xA0031000; /* CLKSEL=SIRC div. by 4, FRZEN=CNTEN=1, RTCVAL=3 */
  WKUP.WISR.R = 0x00000000; /* MPC56xxB: Clear wake up flag RTC */
  ME.MCTL.R = 0xA0005AF0; /* Enter STOP mode and key */
  ME.MCTL.R = 0xA00005AF0; /* Enter STOP mode and inverted key */
  while (ME.GS.B_S_MTRANS) {} /* Wait for STOP mode transition to complete */
} /* STOP HERE FOR STOP MODE! ON STOP MODE EXIT, CODE CONTINUES HERE!*/

Qorivva Simple Cookbook, Rev. 4
while (ME.GS.B.S_CURRENTMODE != 7) {} /* Verify RUN3 (0x7) is the current mode */

SIU.GPDO[68].R = 1;       /* MPC56xxB/S EVB LED: data output: LED off */
RTC.RTCC.R = 0x00000000;   /* Clear CNTEN to reset RTC, enable reloading RTCVAL*/
RTC.RTCC.R = 0xA01B1000;   /* CLKSEL=SRC (div. by 4), FZEN=CNTEN=1, RTCVAL=27*/
WKUP.WISR.R = 0x00000002;   /* MPC56xxB: Clear wake up flag RTC */

ME.MCTL.R = 0xA0005AF0;  /* Enter STOP mode and key */
ME.MCTL.R = 0xA000A50F;  /* Enter STOP mode and inverted key */
while (ME.GS.B.S_MTRANS) {} /* Wait STOP mode transition to complete */

/* STOP HERE FOR STOP MODE! ON STOP MODE EXIT, CODE CONTINUES HERE:*/
}

void disableWatchdog(void) {
    SWT.SR.R = 0x0000c520;   /* Write keys to clear soft lock bit */
    SWT.SR.R = 0x0000d928;
    SWT.CR.R = 0x8000010A;   /* Clear watchdog enable (WEN) */
}
14 eDMA: Block Move

14.1 Description

Task: Initialize an eDMA channel’s Transfer Control Descriptor (TCD) to transfer a string of bytes (“Hello world”) from an array in SRAM to a single SRAM byte location. This emulates a common use of DMA, where a string of data or commands is transferred automatically under DMA control to an input register of a peripheral.

Only one byte of data will be transferred with each DMA service request. Hence the “minor loop” is simply one transfer, which transfers one byte. The “major loop” here consists of 12 minor loop iterations.

Because a peripheral is not involved, automatic DMA handshaking will not occur. Instead, the software handshaking given here must be implemented for each transfer:

- Start DMA service request (set a START bit).
- Poll when that request is done (check the CITER bit field).

These steps appear “messy” for every transfer, which is only a byte in this example. However, remember that when using actual peripherals, software never has to do these steps; they are done automatically by hardware. The purpose of this example is to illustrate how to set up a DMA transfer.

TCD0 will be used. On MPC555x devices, this TCD is assigned to the eQADC’s Command FIFO 0. On MPC551x devices, TCDs are not hard-assigned to any channel.

Because TCDs are in RAM, all fields will come up random. Hence all fields used or enabled must be initialized.

Exercise: Step through code, observing DMA transferring the “Hello world” string to the destination byte. Then modify the TCD so the destination is an array instead of a single byte location.

Figure 25. DMA Block Move Example
### 14.2 Design

#### Table 57. DMA: Block Move

<table>
<thead>
<tr>
<th>Step</th>
<th>Relevant Bit Fields</th>
<th>Pseudo Code</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data Init</strong></td>
<td>Initialize data source string.</td>
<td>SourceData = “Hello world”</td>
</tr>
</tbody>
</table>
| **Init TCD0** | Source Parameters:  
- Source address: address of SourceData  
- Source data transfer size: read 1 byte per transfer  
- Source address offset: increment source address by 1 after each transfer  
- Last source address adjustment: adjust source address by –12 at completion of major loop  
- Source address modulo: disabled |  
- TCD0[SADDR] = &SourceData[]  
- TCD0[SSIZE] = 0  
- TCD0[SOFF] = 1  
- TCD0[SLAST] = –12  
- TCD0[SMOD] = 0 |
| **Destination Parameters:** | Destination address: address of SourceData  
- Destination data transfer size: write one byte per transfer  
- Destination address offset: do not increment destination address after each transfer  
- Last destination address adjustment: do not adjust destination address after completion of major loop  
- Destination address modulo: disabled |  
- TCD0[DADDR] = &Dest.  
- TCD0[DSIZE] = 0  
- TCD0[DOFF] = 0  
- TCD0[DLAST_SGA] = 0  
- TCD0[DMOD] = 0 |
| **Control Parameters** | Inner “minor loop” byte transfer count: transfer 1 byte per service request  
- Number of minor loop iterations in major loop: 12  
- Disable request: at end of major loop, disable further requests for transfer  
- Interrupts: interrupts are not enabled here  
- Linking: channel linking is not enabled here  
- Dynamic programming: dynamic channel linking and scatter gather are not enabled  
- Bandwidth Control: no DMA stalls  
- Status flags: initialize to 0 |  
- TCD0[NBYTES] = 1  
- TCD0[BITER] = 12  
- TCD0[CITER] = 12  
- TCD0[D_REQ] = 1  
- TCD0[INT_HALF] = 0  
- TCD0[INT_MAJ] = 0  
- TCD0[CITERE_LINK] = 0  
- TCD0[BITERE_LINK] = 0  
- TCD0[MAJORE_LINK] = 0  
- TCD0[E_SG] = 0  
- TCD0[BWC] = 0  
- TCD0[START] = 0  
- TCD0[DONE] = 0  
- TCD0[ACTIVE] = 0 |
| **Init DMA Arbitration** | Use fixed priorities for groups and channels (default) | EDMA_CR = 0x0000 E400 |
| | Use defaults: Priorities (Gp = 0, Ch = 0), no preemption | EDMA_CPRO = 0x00 |
| **Enable Chan. 0** | Set enable request for channel 0 | SERQ = 0  
EDMA_SERQR = 0 |
| **Initiate DMA service by software** | Set channel 0 START bit to initiate first minor loop | SSB = 0  
EDMA_SSBR = 0 |
| | For each transfer (while not on last transfer)³:  
- Wait for last transfer to have started and minor loop is finished (is no longer active)  
- Set start bit to initiate next minor loop transfer | wait (START = 1 and ACTIVE = 0)  
SERQ = 0  
while (TCD0[CITER] != 1) {  
wait for ((TCD0[START] = 1) &  
(TCD0[ACTIVE] = 0))  
EDMA_SSBR = 0  
} |

³ The START bit is set on the last iteration here, but is not used. After completion, the channel's DONE bit will be set.
14.3 Code

14.3.1 main.c (All except MPC56xxS)

/* main.c - DMA-Block Move example */
/* Rev 0.1 Sept 30, 2004 S.Mihalik, Copyright Freescale, 2004. All Rights Reserved */
/* Rev 1.0 Jul 10 2007 SM - Changed from TCD18 to TCD0 to be MPC5510 compatible */
/* Notes:
*/
/* 1. MMU not initialized; must be done by debug scripts or BAM */
/* 2. L2SRAM not initialized; must be done by debug scripts */

#include "mpc563m.h" /* Use proper include file such as mpc5510.h or mpc5554.h */

const uint8_t SourceData[] = {"Hello World\r"}; /* Source data string */
uint8_t Destination = 0; /* Destination byte */

void initTCD0(void) {
    EDMA.TCD[0].SADDR = (vuint32_t) &SourceData; /* Load address of source data */
    EDMA.TCD[0].SSIZE = 0; /* Read 2**0 = 1 byte per transfer */
    EDMA.TCD[0].SOFF = 1; /* After transfer, add 1 to src addr*/
    EDMA.TCD[0].SLAST = -12; /* After major loop, reset src addr*/
    EDMA.TCD[0].SMOD = 0; /* Source modulo feature not used */

    EDMA.TCD[0].DADDR = (vuint32_t) &Destination; /* Load address of destination */
    EDMA.TCD[0].DSIZE = 0; /* Write 2**0 = 1 byte per transfer */
    EDMA.TCD[0].DOFF = 0; /* Do not increment destination addr*/
    EDMA.TCD[0].DLAST_SGA = 0; /* After major loop, no dest addr change*/
    EDMA.TCD[0].DMOD = 0; /* Destination modulo feature not used */

    EDMA.TCD[0].NBYTES = 1; /* Transfer 1 byte per minor loop */
    EDMA.TCD[0].BITER = 12; /* 12 minor loop iterations */
    EDMA.TCD[0].CITER = 12; /* Initialize current iteration count */
    EDMA.TCD[0].D_REQ = 1; /* Disable channel when major loop is done*/
    EDMA.TCD[0].INT_HALF = 0; /* Interrupts are not used */
    EDMA.TCD[0].INT_MAJ = 0;
    EDMA.TCD[0].CITERE_LINK = 0; /* Linking is not used */
    EDMA.TCD[0].BITERE_LINK = 0;
    EDMA.TCD[0].MAJORE_LINK = 0; /* Dynamic program is not used */
    EDMA.TCD[0].E_SG = 0;
    EDMA.TCD[0].BWC = 0; /* Default bandwidth control- no stalls */
    EDMA.TCD[0].DONE = 0; /* Initialize status flags */
    EDMA.TCD[0].ACTIVE = 0;
}

void main (void) {
    int i = 0; /* Dummy idle counter */

    initTCD0(); /* Initialize DMA Transfer Control Descriptor 0 */
    EDMA.CR.R = 0x0000E400; /* Use fixed priority arbitration for DMA groups and channels */
    EDMA.CPR[0].R = 0x12; /* Channel 0 priorities: group priority = 1, channel priority = 2 */
    EDMA.SERQR.R = 0; /* Enable EDMA channel 0 */
    EDMA.SSBR.R = 0; /* Set channel 0 START bit to initiate first minor loop transfer */

    /* Initiate DMA service using software */
    while (EDMA.TCD[0].CITER != 1) { /* while not on last minor loop */
        while ((EDMA.TCD[0].START == 1) | (EDMA.TCD[0].ACTIVE == 1)) {}
        EDMA.SSBR.R = 0; /* Set channel 0 START bit again for next minor loop transfer */
    }

    while (1) { i++; } /* Loop forever */
}
14.3.2 main.c (MPC56xxS)

/* main.c - DMA Block Move */
/* Oct 23 2008 S.Mihalik -inital version */
/* Mar 15 2010 S.Mihalik - updated for new eDMA version on MPC5606S */
/* Copyright Freescale Semiconductor, Inc 2008, 2010 All rights reserved. */
#include "56xxS_0204.h" /* Use proper include file such as mpc5510.h or mpc5554.h */

const uint8_t SourceData[] = {"Hello World\r"}; /* Source data string */
const uint8_t Destination = 0; /* Destination byte */

void initTCD0(void) {
    EDMATCD[0].SADDR = (vuint32_t)&SourceData; /* Load address of source data */
    EDMATCD[0].SSIZE = 0; /* Read 2**0 = 1 byte per transfer */
    EDMATCD[0].SLOOP = 1; /* After transfer, add 1 to src addr*/
    EDMATCD[0].SLAST = -12; /* After major loop, reset src addr*/
    EDMATCD[0].SMOD = 0; /* Source modulo feature not used */
    EDMATCD[0].DADDR = (vuint32_t)&Destination; /* Load address of destination */
    EDMATCD[0].DSIZE = 0; /* Write 2**0 = 1 byte per transfer */
    EDMATCD[0].DOFF = 0; /* Do not increment destination addr */
    EDMATCD[0].DLAST_SGA = 0; /* After major loop, no dest addr change*/
    EDMATCD[0].DMOD = 0; /* Destination modulo feature not used */
    EDMATCD[0].NBYTESu.R = 1; /* Transfer 1 byte per minor loop */
    EDMATCD[0].BITER = 12; /* 12 minor loop iterations */
    EDMATCD[0].D_REQ = 1; /* Initialize current iteration count */
    EDMATCD[0].INTHALF = 0; /* Disable channel when major loop is done*/
    EDMATCD[0].INT_HALF = 0; /* Interrupts are not used */
    EDMATCD[0].INT_MAJ = 0;
    EDMATCD[0].CITERE_LINK = 0; /* Linking is not used */
    EDMATCD[0].BITERE_LINK = 0;
    EDMATCD[0].MAJORE_LINK = 0; /* Dynamic program is not used */
    EDMATCD[0].E_SG = 0;
    EDMATCD[0].BWC = 0; /* Default bandwidth control- no stalls */
    EDMATCD[0].START = 0; /* Initialize status flags */
    EDMATCD[0].DONE = 0;
    EDMATCD[0].ACTIVE = 0;
}

void main (void) {
    volatile uint32_t i = 0; /* Dummy idle counter */
    initTCD0(); /* Initialize DMA Transfer Control Descriptor 0 */
    EDMACR.R = 0x0000E400; /* Use fixed priority arbitration for DMA groups and channels */
    EDMA.CPR[0].R = 0x0; /* Channel 0 priorities: group priority = 0, channel priority = 0 */
    EDMA.SREQ.R = 0; /* Enable EDMA channel 0 */
    EDMA.SSRT.R = 0; /* Set channel 0 START bit to initiate first minor loop transfer */
    while (1) {
        i++;
        while ((EDMA.TCD[0].CITER != 1) & (EDMA.TCD[0].ACTIVE == 1)) {}
        EDMA.SSRT.R = 0; /* Set channel 0 START bit to initiate first minor loop transfer */
    }
}
15 eSCI: Simple Transmit and Receive

15.1 Description

Task: Transmit the string of data “Hello World<CR>” to eSCI_A, then wait to receive a byte. The serial parameters are: 9600 baud, eight bits data, and no parity. Interrupts and DMA are not used. The program simply waits until flags are at the desired state before proceeding.

Overflow is not checked in this example, as we look for only one received character. If overflow is a concern on reception, the overflow flag would be checked after reading the received character.

Exercise: Connect a COM port of a PC to an MPC5500 evaluation board eSCI port. Use a terminal emulation program for communication from the PC side with settings of 9600 baud, 8 bits data, no parity, and no flow control. Make sure to have the correct SCI jumper settings for Tx/Rx connection for the particular EVB used. Step through the program and verify proper transmission.

![Diagram](image)

Figure 26. eSCI Simple Transmit and Receive Example

Table 58. Signals for eSCI Example

<table>
<thead>
<tr>
<th>Signal</th>
<th>Pin Name</th>
<th>SIU PCR No.</th>
<th>Package Pin No.</th>
<th>Function Name</th>
<th>SIU PCR No.</th>
<th>Package Pin No.</th>
<th>EVB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>144 QFP</td>
<td>176 QFP</td>
<td></td>
<td>496 BGA</td>
<td>416 BGA</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>208 BGA</td>
<td></td>
<td></td>
<td>208 BGA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TxDA</td>
<td>PD6</td>
<td>54</td>
<td>98</td>
<td>TXDA</td>
<td>89</td>
<td>V24</td>
<td>PJ1–14</td>
</tr>
<tr>
<td></td>
<td></td>
<td>122</td>
<td></td>
<td></td>
<td>U24</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>N20</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>J14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RxDA</td>
<td>PD7</td>
<td>55</td>
<td>97</td>
<td>RXDA</td>
<td>90</td>
<td>U26</td>
<td>PJ1–13</td>
</tr>
<tr>
<td></td>
<td></td>
<td>121</td>
<td></td>
<td></td>
<td>V24</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>P20</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>K14</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Qorivva Simple Cookbook, Rev. 4
15.2 Design

For the desired 9600 SCI baud rate, the ideal calculation for SBR bitfield `eSCIX_CR1[SBR]`, assuming a 64 MHz sysclk to the eSCI module, is:

\[
SBR = \frac{\text{eSCI system clock}}{16 \times \text{SCI Baud Rate}} = \frac{64 \text{ M}}{16 \times 9600} = 416.66...
\]

So after rounding it off to the nearest whole number, 417 (0x1A1) will be used for SBR in this example.

The time per bit at 9600 baud = 1 sec / 9600 bits, which rounds off to 104 microseconds. The oscilloscope shot below shows only one frame (character) because it was taken when stepping through code. It shows the bit time being met.

The scope shot also shows the transmission sequence of sending the start bit, followed by LSB through MSB bits. Here the data bits, LSB first, are 0001 0010, which after putting MSB first is 0x48, the ASCII code for “H.”

The scope shot below shows the program running without a breakpoint. There are contiguous transmissions of characters, and the first two character are shown here.
### Table 59. eSCI Simple Design Steps

<table>
<thead>
<tr>
<th>Step</th>
<th>Relevant Bit Fields</th>
<th>Pseudo Code</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data Init</strong></td>
<td></td>
<td>TransData = “Hello World!&lt;CR&gt;”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RecData = 0</td>
</tr>
<tr>
<td><strong>Init sysclk</strong></td>
<td></td>
<td>CRP_CLKSRC [XOSCEN] = 1;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PLL_ESYNCR2 = 0x0000 0006;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PLL_ESYNCR1 = 0xF000 0020;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Wait for FMPLL_SYNCR [LOCK] = 1;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PLL_ESYNCR2 = 0x0000 0005;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SIU_SYSCLK [SYSCLKSEL] = 2;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Note for 40 MHz Crystal used on MPC55xx — Replace FMPLL_SYNCR values:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x1608 0000 with 0x4610 0000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x1600 0000 with 0x4608 0000</td>
</tr>
<tr>
<td><strong>Init eSCI_A</strong></td>
<td>MDIS = 0 (default)</td>
<td>ESCIA_CR2 = 0x2000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SBR = 417 (0x1A1) M = 0 (default)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PE = 0 (default) TE = 1 RE = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SIU_PCR[54] = 0x0400 SIU_PCR[55] = 0x0400 SIU_PCR[89] = 0x0400 SIU_PCR[90] = 0x0400</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Configure pads: PA = primary PA = primary</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SIU_PCR[54] = 0x0400 SIU_PCR[55] = 0x0400 SIU_PCR[89] = 0x0400 SIU_PCR[90] = 0x0400</td>
</tr>
<tr>
<td><strong>Transmit Data</strong></td>
<td></td>
<td>Loop for # characters ( wait for ESCIA_SR[TDRE] = 1 ESCIA_SR[TDRE] = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>write TRDE = 1</td>
</tr>
<tr>
<td><strong>Read and Echo Back Data</strong></td>
<td></td>
<td>Loop for # characters ( wait for ESCIA_SR[RDREF] = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>write RDRF = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ESCIA_SR = 0x2000 0000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Read byte</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RecData = ESCIA_DR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Ensure Transmit Data Register Empty status flag</td>
</tr>
<tr>
<td></td>
<td></td>
<td>write TDRE = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ESCIA_SR = 0x8000 0000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Transmit back (echo) received byte</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ESCIA_DR = RecData</td>
</tr>
</tbody>
</table>
#include "mpc563m.h" /* Use proper include file such as mpc5510.h or mpc5554.h */

const uint8_t TransData[] = {"Hello World!\n\r"); /* Transmit string & CR*/
uint8_t RecData;                            /* Received byte from eSCI */

void initSysclk (void) {
    /* MPC551x: Use the next 6 lines */
    CRP.CLKSRC.B.XOSCEN = 1; /* Enable external oscillator */
    FMPLL.ESYNCR2.R = 0x00000006; /* Set ERFD to initial value of 6 */
    FMPLL.ESYNCR1.R = 0xF0000020; /* Set CLKCFG=PLL, EPREDIV=0, EMFD=0x20*/
    while (FMPLL.SYNSR.B.LOCK != 1) {}; /* Wait for PLL to LOCK */
    FMPLL.ESYNCR2.R = 0x00000005; /* Set ERFD to final value for 64 MHz sysclk */
    SIU.SYSCLK.B.SYSCLKSEL = 2; /* Select PLL for sysclk */
    FMPLL.ESYNCR1.B.CLKCFG = 0x7; /* Change clk to PLL normal mode from crystal */
    /* MPC563x: Use the next line */
    SIU.SYSCLK.B.SYCLKSEL = 2; /* Select PLL for sysclk */
    /* MPC555x including MPC563x: use the next 3 lines for either 8 or 40 MHz crystal */
    while (FMPLL.SYNSR.B.LOCK != 1) {}; /* Wait for PLL to LOCK */
    FMPLL.ESYNCR1.R = 0x46080000; /* 8 MHz xtal: 0x46080000; 40MHz: 0x46100000 */
    FMPLL.ESYNCR2.R = 0x00000005; /* Set ERFD to final value for 64 MHz sysclk */
}

void initESCI_A (void) {
    ESCI_A.CR2.R = 0x2000; /* Module is enabled (default setting ) */
    ESCI_A.CR1.R = 0x01A1000C; /* 9600 baud, 8 bits, no parity, Tx & Rx enabled */
    /* Use the following two lines for MPC551x */
    SIU.PCR[54].R = 0x400; /* Configure pad for primary func: TxDA */
    SIU.PCR[55].R = 0x400; /* Configure pad for primary func: RxDA */
    /* Use the following two lines for MPC555x */
    SIU.PCR[89].R = 0x400; /* Configure pad for primary func: TxDA */
    SIU.PCR[90].R = 0x400; /* Configure pad for primary func: RxDA */
}

void TransmitData (void) {
    uint8_t t_j;
    for (t_j = 0; t_j < sizeof (TransData); t_j++) { /* Loop for character string */
        while (ESCI_B.SR.B.TXRDY == 0) {}; /* Wait for LIN transmit ready = 1 */
        ESCI_B.SR.B.TXRDY = 0x00004000; /* Clear TXRDY flag */
        ESCI_B.LTR.R = FrameSpecAndData[t_j] << 24; /* Write byte to LIN Trans Reg. */
    }
}

void ReceiveData (void) {
    while (ESCI_A.SR.B.RDRF == 0) {}; /* Wait for receive data reg full = 1 */
    ESCI_A.SR.R = 0x02000000; /* Clear RDRF flag */
    while (ESCI_A.SR.B.TDRE == 0) {}; /* Wait for transmit data reg empty = 1 */
    ESCI_A.SR.R = 0x00000000; /* Clear TDRE flag */
    ESCI_A.DR.B.D = RecData; /* Echo back byte of Data read */
}

void main(void) {
    initSysclk(); /* Set sysclk = 64 MHz running from PLL */
    initESCI A(); /* Enable Tx/Rx for 9600 baud, 8 bits, no parity */
    TransmitData(); /* Transmit string of characters on eSCI A */
    ReceiveData(); /* Receive and echo character on eSCI A */
    while (1) {} /* Wait forever */
}
16  eSCI: LIN Transmit

16.1  Description

Task: Transmit the data string “Hello” via LIN using eSCI_B. When sending bytes to the LIN Transmit Register, use software to poll the TXRDY flag for each byte write, with software clearing the flag each time.

For the most efficiency, DMA would be used, which automatically waits for TXRDY flag and clears it after a write. The next most efficient method would be to have the TXRDY interrupt the processor. Parameters from Section 15, “eSCI: Simple Transmit and Receive,” are used, including 10417 baud, 8 bits data, and no parity. Interrupts and DMA are not used.

NOTE

The RxD receive pin must be able to monitor the TxD transmitted signal. Hence either (1) the LIN transceiver must be connected as shown and the transceiver must be powered, or (2) for debug purposes, the TxD and RxD can be shorted together without using the transceiver. The MPC5510 EVB and MPC563M EVB require changing the default configuration to implement one of these two options.

Exercise: Connect an oscilloscope or LIN tool and verify the transmit output. Alter frame ID and repeat.

Figure 27. eSCI LIN Example

Table 60. Signals for eSCI LIN Example

<table>
<thead>
<tr>
<th>Signal</th>
<th>Pin Name</th>
<th>SIU PCR No.</th>
<th>Package Pin No.</th>
<th>Function Name</th>
<th>SIU PCR No.</th>
<th>Package Pin No.</th>
<th>EVB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TxDB</td>
<td>PD8</td>
<td>56</td>
<td>94</td>
<td>118</td>
<td>G13</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>TXDB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RxDB</td>
<td>PD9</td>
<td>57</td>
<td>93</td>
<td>117</td>
<td>F16</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>RXDB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Qorivva Simple Cookbook, Rev. 4
16.2 Design

To transmit a LIN frame, first the frames data must be specified by writing the ID, length and control to the LIN Transmit Register. This specifies the LIN frame, and the frame’s header starts to transfer. Next, the data is written to the same LIN Transmit Register. The byte values used in the example are shown in the table below.

Per LIN 2.x and J2602 the a checksum is used (CSUM = 1), which includes the header (HDCHK = 1).

<table>
<thead>
<tr>
<th>Field</th>
<th>Byte Value</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parity (1:0) = 0 (Parity not used)</td>
<td>ID (5:0) = 0x35</td>
<td>0x35</td>
</tr>
<tr>
<td>Length (7:0) = 8 (for 8 bytes data)</td>
<td>Specifies LIN frame (which causes LIN frame’s header to start transmission)</td>
<td></td>
</tr>
<tr>
<td>HDCHK = 1 (header is included in checksum)</td>
<td>CSUM = 1 (checksum is appended to end of frame)</td>
<td>CRC = 0 (2 CRC bytes not appended to end of frame)</td>
</tr>
</tbody>
</table>

| Data 0 = ‘H’ | 0x48 |
| Data 1 = ‘e’ | 0x65 |
| Data 2 = ‘l’ | 0x6C |
| Data 3 = ‘l’ | 0x6C |
| Data 4 = ‘o’ | 0x6F |
| Data 5 = ‘ ’ | 0x20 |
| Data 6 = ‘ ’ | 0x20 |
| Data 7 = ‘ ’ | 0x20 |

Bit or physical bus errors can stop DMA transmission (by setting the BSTP bit in eSCIx_CR2). Although DMA is not used here, this bit will be set anyway.

This example shows a “good” case, where no errors occur or are checked. Normally one would enable error interrupts and/or check those status flags also. A partial list of errors includes:

- A physical bus error of a permanently low bit: sets framing error flag, ESCIx_SR[FE]
- RxD input stuck after transmission starts: sets physical bus error flag, ESCIx_LSR[PBERR]
- For receive frames, a slave does not respond in the specified timeout in ESCIx_LTR: sets slave timeout flag, ESCIx_LSR[STO]

The LIN state machine can automatically reset after an exception of a bit error, physical bus error, or wakeup. For debug purposes, it is useful to disable this feature, but we will allow the reset to occur by disabling the feature (accomplished by clearing the LDBG bit in eSCIx_LCR).
## 16.2.1 Design Steps

### Table 62. eSCI LIN Design Steps

<table>
<thead>
<tr>
<th>Step</th>
<th>Relevant Bit Fields</th>
<th>Pseudo Code</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td><strong>MPC551x</strong></td>
</tr>
<tr>
<td><strong>Data Init</strong></td>
<td>LIN frame specification and 8 bytes data for transmit</td>
<td>FrameSpecAndData = 0x35, 0x08, 0xD0, \texttt{'}H\texttt{'} , \texttt{'}e\texttt{'} , \texttt{'}l\texttt{'} , \texttt{'}l\texttt{'} , \texttt{'}\texttt{'} , \texttt{'}\texttt{'} , \texttt{'}\texttt{'}</td>
</tr>
<tr>
<td><strong>Init sysclk</strong></td>
<td>Set system clock = 64 MHz</td>
<td>CRP_CLKSRC [XOSCEN] = 1; PLL_ESYNCR2 = 0x0000 0006; PLL_ESYNCR1 = 0xF000 0020;</td>
</tr>
<tr>
<td>         (See Section 10, “PLL: Initializing System Clock (MPC551x, MPC555x)”</td>
<td></td>
<td></td>
</tr>
<tr>
<td>         Note for 40 MHz Crystal used on MPC55xx — Replace FMPLL_SYNCR values: 0x1608 0000 with 0x4610 0000 0x1600 0000 with 0x4608 0000</td>
<td></td>
<td>Also for MPC563x: FMPLL_ESYNCR1 = 0x4610 0000;</td>
</tr>
<tr>
<td><strong>Init eSCI B for LIN</strong></td>
<td>Initialize desired Control Register 2 settings:</td>
<td>MDIS = 0 (default) BRK13 = 1 BSTOP = 1</td>
</tr>
<tr>
<td>                    Turn on the eSCI module (in case it was off)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>                    Set break character length to 13 bits</td>
<td></td>
<td></td>
</tr>
<tr>
<td>                    Stop DMA (not used here) on bit/physical bus error</td>
<td></td>
<td></td>
</tr>
<tr>
<td>                    SCI bit errors cause stop on bit instead of frame end</td>
<td></td>
<td></td>
</tr>
<tr>
<td>                    Detect bit errors on each bit (“fast”) instead of byte</td>
<td></td>
<td></td>
</tr>
<tr>
<td>                    MDIS = 0 (default) BRK13 = 1 BSTOP = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>                    SBSTP = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>                    FBR = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Initialize desired Control Register 1 settings:</strong></td>
<td>SBR = 384 (0x180) M = 0 (default) PE = 0 (default) TE = 1 RE = 1 TIE = TCIE = RIE = 0</td>
<td>ESCIB_CR1 = 0x0180 000C</td>
</tr>
<tr>
<td>                    Baud Rate value = 64 M / (16x10417) == 384</td>
<td></td>
<td></td>
</tr>
<tr>
<td>                    Word length = 8 bits</td>
<td></td>
<td></td>
</tr>
<tr>
<td>                    Parity is not enabled</td>
<td></td>
<td></td>
</tr>
<tr>
<td>                    Enable transmitter</td>
<td></td>
<td></td>
</tr>
<tr>
<td>                    Enable receiver</td>
<td></td>
<td></td>
</tr>
<tr>
<td>                    Do not use normal eSCI interrupts</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Initialize desired LIN Control Register settings:</strong></td>
<td>LIN = 1 LDBG = 0</td>
<td>ESCIB_LCR = 0x0100 0000</td>
</tr>
<tr>
<td>                    Switch eSCI to LIN mode</td>
<td></td>
<td></td>
</tr>
<tr>
<td>                    Do not enable exception to reset state machine</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Configure pad:</strong></td>
<td>PA = primary</td>
<td>PA = primary</td>
</tr>
<tr>
<td>                    TxDB</td>
<td>SIU_PCR[56] = 0x0400</td>
<td>SIU_PCR[91] = 0x0400</td>
</tr>
<tr>
<td>                    RxDB (NOTE: Some EVB transceivers, such as TJA1020, require a pullup. In this case, at least the internal pullup should be used, so the PCR value should be 0x0403 instead of 0x400.)</td>
<td>SIU_PCR[57] = 0x0400</td>
<td>SIU_PCR[92] = 0x0400</td>
</tr>
<tr>
<td><strong>Transmit Data</strong></td>
<td>wait for TXRDY = 1</td>
<td>Loop for # characters {</td>
</tr>
<tr>
<td>                    Wait for Transmit Data Ready status flag</td>
<td>write TXRDY = 1</td>
<td>wait for ESCIB_SR[TXRDY] = 1</td>
</tr>
<tr>
<td>                    Clear status flag</td>
<td>write byte to LTR</td>
<td>ESCIB_SR = 0x0000 4000</td>
</tr>
<tr>
<td>                    Write one character</td>
<td></td>
<td>ESCIB_LTR = next char. }</td>
</tr>
</tbody>
</table>
16.2.2 Design Screenshots

Below is an oscilloscope trace of TxDB output showing the frame’s header. Normally the data would continue after the header, but code was deliberately stepped through here to show the header generation after the first three writes to the LIN Transmit Register.

![Figure 28. LIN Transmit Frame Header](image)

![Figure 29. Entire LIN Transmit Frame](image)
16.3 Code
/* main.c: Simple eSCI LIN program */
/* Rev 0.1 Oct 10, 2007 S.Mihalik- Initial version */
/* Rev 0.2 Jun 04 2008 SM - initSysclk changed for MPC5633M support */
/* Rev 0.3 Aug 16 2008 SM - changed baud rate to 10.417K */
/* Notes: 1 MMU not initialized; must be done by debug scripts or BAM */
/* 2 SRAM not initialized; must be done by debug scripts or in a startup file */
#include "mpc563m.h" /* Use proper include file such as mpc5510.h or mpc5554.h */
const uint8_t FrameSpecAndData[] = {0x35, 0x08, 0xD0, 'H', 'e', 'l', 'l', 'o', ' ', ' ', ' '};

void initSysclk (void) {
/* MPC551x: Use the next 6 lines */
CRP.CLSRC.B.XOSCEN = 1;    /* Enable external oscillator */
FMPLL.ESYNCR2.R = 0x00000006;    /* Set ERFD to initial value of 6 */
FMPLL.ESYNCR1.R = 0xF0000020;    /* Set CLKCFG=PLL, EPREDIV=0, EMFD=0x20 */
while (FMPLL.SYNSR.B.LOCK != 1) {};    /* Wait for PLL to LOCK */
FMPLL.ESYNCR2.R = 0x00000005;    /* Set ERFD to final value for 64 MHz sysclk */
SIU.SYSCLK.B.SYSLKSEL = 2;    /* Select PLL for sysclk */
/* MPC563x: Use the next line */
FMPLL.ESYNCR1.B.CLKCFG = 0x7;    /* Change clk to PLL normal mode from crystal */
/* MPC555x including MPC563x: use the next 3 lines for either 8 or 40 MHz crystal */
FMPLL.SYNCR.R = 0x16080000;    /* 8 MHz xtal: 0x16080000; 40MHz: 0x46100000 */
while (FMPLL.SYNSR.B.LOCK != 1) {};    /* Wait for PLL to LOCK */
FMPLL.SYNCR.R = 0x16000000;    /* 8 MHz xtal: 0x16000000; 40MHz: 0x46080000 */
}

void initESCI_B (void) {
ESCI_B.CR2.R = 0x6240;    /* Module is enabled, 13 bit break, stop on errors */
ESCI_B.CR1.R = 0x0180000C;    /* 10417 baud, 8 bits, no parity, Tx & Rx enabled */
ESCI_B.LCR.R = 0x01000000;    /* eSCI put in LIN mode */
/* Use the following two lines for MPC551x */
SIU.PCR[56].R = 0x400;    /* Configure pad for primary func: TxDB */
SIU.PCR[57].R = 0x400;    /* Configure pad for primary func: RxDB */
/* Use the following two lines for MPC555x */
SIU.PCR[91].R = 0x400;    /* Configure pad for primary func: TxDB */
SIU.PCR[92].R = 0x400;    /* Configure pad for primary func: RxDB */
}

void TransmitData (void) {
uint8_t j;
for (j=0; j< sizeof (FrameSpecAndData); j++) {
   while (ESCI_A.SR.B.TXRDY == 0) {} /* Wait for LIN transmit ready = 1 */
   ESCI_A.SR.R = 0x00000400;    /* Clear TXRDY flag */
   ESCI_A.LTR.R = FrameSpecAndData[j];    /* Write 8 byte to LIN Trans Reg. */
}
}

void main(void) {
   initSysclk();    /* Set sysclk = 64MHz running from PLL */
   initESCI_B();    /* Enable Tx for 10417 baud, 8 bits, no parity */
   TransmitData();    /* Transmit string of characters on eSCI B */
   while (1) {}    /* Wait forever */
}
17 LINFlex: LIN Transmit

17.1 Description

Task: Transmit the data string “Hello” and three spaces using LIN. Use parameters as in the table below.

Table 63. MPC56xxB/P/S LINFlex — LIN example parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Setting</th>
<th>Register[Bit field]</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td>Master or Slave</td>
<td>Master</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Master break length</td>
<td>13-bit break</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CKSUM done in HW</td>
<td>Yes (no CRC is used)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CKSUM enabled</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Baud rate</td>
<td>10417 Hz for 64 MHz clock</td>
<td></td>
</tr>
<tr>
<td>Frame</td>
<td>Data</td>
<td>String “Hello” and 3 spaces</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ID</td>
<td>0x35</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Data field length</td>
<td>8 bytes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Message direction</td>
<td>Transmit</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CKSYM type</td>
<td>Enhanced (LIN 2.0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NOTE

The RxD receive pin must be able to monitor the TxD transmitted signal. Hence either (1) the LIN transceiver must be connected as shown and the transceiver powered, or (2) for debug purposes, the TxD and RxD can be shorted together without using the transceiver.

Exercise: Connect an oscilloscope or LIN tool and verify the transmit output.

![Figure 30. LINFlex LIN Transmit Example Simplified Block Diagram](image-url)
### Table 64. MPC56xxB/P/S Signals for LINFlex LIN TransmitScan Example

<table>
<thead>
<tr>
<th>Signal</th>
<th>MPC56xxB Family</th>
<th>MPC56xxP Family</th>
<th>MPC56xxS Family</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Port</td>
<td>SIU Pad Configuration &amp; Selection Registers (values in hex)</td>
<td>Package Pin #</td>
</tr>
<tr>
<td></td>
<td>Port</td>
<td>100 QFP</td>
<td>144 QFP</td>
</tr>
<tr>
<td><strong>TxD</strong></td>
<td>B2</td>
<td>PCR18=0400</td>
<td>100</td>
</tr>
<tr>
<td><strong>RxD</strong></td>
<td>B3</td>
<td>PCR19=0103 no PSMI</td>
<td>1</td>
</tr>
</tbody>
</table>
17.2 Design

17.2.1 Mode Use

Mode Transition is required for changing mode entry registers. Hence even enabling the crystal oscillator to be active in the current mode (for example default mode (DRUN)) requires enabling the crystal oscillator in DRUN mode configuration register (ME_DRUN_MC), then initiating a mode transition to the same DRUN mode. This example changes from DRUN mode to RUN0 mode.

This minimal example simply polls a status bit to wait for the targeted mode transition to complete. However, the status bit could instead be enabled to generate an interrupt request (assuming the INTC is initialized beforehand). This would allow software to complete other initialization tasks instead of brute force polling of the status bit.

It is normal to use a timer when waiting for a status bit to change. This example by default would have a watchdog timer expire if for some reason the mode transition never completed. One could also loop code on incrementing a software counter to some maximum value as a timeout. If a timeout was reached, then an error condition could be recorded in EEPROM or elsewhere.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Mode Config. Register</th>
<th>Mode Config. Register Value</th>
<th>sysclk Selection</th>
<th>Clock Sources</th>
<th>Memory Power Mode</th>
<th>Main Voltage Reg.</th>
<th>I/O Power Down Ctrl</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRUN</td>
<td>ME_DRUN_MC</td>
<td>0x001F 0010 (default)</td>
<td>16 MHz IRC</td>
<td>On</td>
<td>Off</td>
<td>Normal</td>
<td>On</td>
</tr>
<tr>
<td>RUN0</td>
<td>ME_RUN0_MC</td>
<td>0x001F 0074</td>
<td>PLL0</td>
<td>On</td>
<td>On</td>
<td>Normal</td>
<td>Off</td>
</tr>
</tbody>
</table>

Other modes are not used in example.

Peripherals also have configurations to gate clocks on and off, enabling low power. The following table summarizes the peripheral configurations used here. ME_RUNPC_1 is selected, therefore peripherals to be used require a non-zero value in their respective ME_PCTL register.

<table>
<thead>
<tr>
<th>Peripheral Config.</th>
<th>Peripheral Config.</th>
<th>Enabled Modes</th>
<th>Peripherals Selecting Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>RUN3</td>
<td>RUN2</td>
<td>RUN1</td>
<td>RUN0</td>
</tr>
<tr>
<td>PC1</td>
<td>ME_RUNPC_1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Other peripheral configurations are not used in example.
### 17.2.2 Steps and Pseudo Code

**Table 67. MPC5606B, MPC56xxP, MPC56xxS Steps for LINFlex LIN Transmit Example**

<table>
<thead>
<tr>
<th>Step</th>
<th>Relevant Bit Fields</th>
<th>Pseudo Code</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Init Modes and Clock</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enable desired modes</td>
<td>RUN0, DRUN=1</td>
<td>ME_ME = 0x0000 001D</td>
</tr>
</tbody>
</table>
| Initialize PLL0 dividers to provide 64 MHz for input crystal before mode configuration:  
- 8 MHz xtal: FMPLL[0]_CR=0x02400100  
- 40 MHz xtal: FMPLL[0]_CR=0x12400100 | CGM_FMPLL_CR (MPC56xxB)  
CGM_FMPLL[0]_CR (MPC56xxS)  
= 0x0240 0100 (8 MHz crystal)  
or 0x1240 0100 (40 MHz crystal) |
| Configure RUN0 Mode:  
- I/O Output power-down: no safe gating  
- Main Voltage regulator is on (default)  
- Data, code flash in normal mode (default)  
- PLL0 is switched on  
- Crystal oscillator is switched on  
- 16 MHz IRC is switched on (default)  
- Select PLL0 (system pll) as sysclk | PDO=0  
MVRON=1  
DFLAON = 3  
CFLAON = 3  
PLL0ON=1  
XOSC0ON=1  
16MHz_IRCON = 1  
SYSCLK=0x4 |
| Peri. Config. 1: run in RUN0 mode only | RUN0=1 | ME_RUN0_MC = 0x001F 0074 |
| Assign peripheral configuration to peripherals:  
- LINFlex 0: select ME_RUN_PC0  
- LINFlex 1: select ME_RUN_PC0  
- SIUL: select ME_RUN_PC0 (56xxB/S) | RUN_CFG = 1  
ME_PCTL48 = 0x01 |
| Initiate software mode transition to RUN0 mode | TARGET_MODE=RUN0  
wait for I_TC = 1  
I_TC=1 | ME_MCTL =0x4000 5AF0  
ME_MCTL =0x4000 A50F |
| Disable Watchdog | | SWT_SR = 0x0000 0C520  
SWT_SR = 0x0000 D928  
SWT_CR = 0x8000 010A |
| init Peri Clk Gen | | |
| Initialize peripheral clock generation (See appendix: MPC56xxB/P/S Peripheral Clocks)  
- LINFlex (56xxB/S): peripheral set 1 – sysclk/1 | DE0=1  
SC_DC0 = 0x80 | CGM_SC_DC0 |
| | DIV0=0 | CGM_SC_DC0 = 0x80 |
### Table 67. MPC5606B, MPC56xxP, MPC56xxS Steps for LINFlex LIN Transmit Example

<table>
<thead>
<tr>
<th>Step</th>
<th>Relevant Bit Fields</th>
<th>Pseudo Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Put LINFlex hardware in init mode</td>
<td>INIT = 1</td>
<td>LINCR1[INIT] = 1</td>
</tr>
<tr>
<td>Initialize module:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Mode = LIN Master</td>
<td>MME = 1</td>
<td>LINCR1 = 0x0000 0311</td>
</tr>
<tr>
<td>• LIN master break length = 13</td>
<td>MBL = 3</td>
<td></td>
</tr>
<tr>
<td>• CKSUM done in hardware</td>
<td>CCD = 0</td>
<td></td>
</tr>
<tr>
<td>• CKSUM field (for LIN frame) enabled</td>
<td>CFD = 0</td>
<td></td>
</tr>
<tr>
<td>• Module HW put in INIT mode</td>
<td>INIT = 1</td>
<td></td>
</tr>
<tr>
<td>Set baud rate = 10417 for 64 MHz sysclk(^1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Put LINFlex hardware in normal mode</td>
<td>INIT = 0</td>
<td>LINCR1 = 0x0000 0310</td>
</tr>
<tr>
<td>SLEEP = 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Init pads for FlexLIN_0 TxD &amp; RxD</td>
<td>SIU_PCR18 = 0x0100</td>
<td>SIU_PCR18 = 0x0400</td>
</tr>
<tr>
<td>SIU_PCR19 = 0x0103</td>
<td>SIU_PCR19 = 0x0403</td>
<td>SIU_PCR19 = 0x0400</td>
</tr>
<tr>
<td></td>
<td>SIU_PCR19 = 0x0503</td>
<td>SIU_PCR19 = 0x0503</td>
</tr>
<tr>
<td>Transmit LIN Frame</td>
<td>Load buffer data with 8 bytes: “Hello” and 3 blank spaces (hex 48 65 6C 6C 6F 20 20 20)</td>
<td>BDRM = 0x4865 6C6C</td>
</tr>
<tr>
<td>BDRL = 0x6F20 2020</td>
<td>BIDR = 0x0000 1E35</td>
<td></td>
</tr>
<tr>
<td>Init header in Buffer ID register:</td>
<td>ID = 0x35</td>
<td></td>
</tr>
<tr>
<td>• ID = 0x35</td>
<td>DFL = 7</td>
<td></td>
</tr>
<tr>
<td>• Data field length = 8 bytes</td>
<td>DIR = 1</td>
<td></td>
</tr>
<tr>
<td>• Message direction = transmit</td>
<td>CCS = 0</td>
<td></td>
</tr>
<tr>
<td>• Cksum type = Enhanced (for LIN 2.0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Request header transmission (using default error controls)</td>
<td>HTRQ=1</td>
<td>LINCRC2[HTRQ] = 1</td>
</tr>
</tbody>
</table>

\(^1\) Per MPC5606S Microcontroller Reference Manual Rev 3 Table 22-1 for 10417 baud rate. Note: LINFBRR[DIV_F] field is 4 bits, so the value must be less than 16. Some documentation has a table incorrectly showing value of 16 for this field at 10417 baud.
### 17.3 Code

#### 17.3.1 MPC560xB

/* main.c: LINFlex program for MPC56xxB */
/* Description: Transmit one message from FlexCAN 0 buf. 0 to FlexCAN C buf. 1 */
/* Oct 30 2009 SM - initial version */
/* Mar 14 2010 SM - modified initModesAndClock, updated header file */
#include "MPC5604B_0M27V_0101.h" /* Use proper header file*/

void initModesAndClks(void) {
    ME.ER.R = 0x0000001D;          /* Enable DRUN, RUN0, SAFE, RESET modes */
    /* Initialize PLL before turning it on: */
    /* Use 1 of the next 2 lines depending on crystal frequency: */
    CGM.FMPLL.CR.R = 0x02400100;    /* 8 MHz xtal: Set PLL0 to 64 MHz */
    CGM.FMPLL.CR.R = 0x01200100;    /* 40 MHz xtal: Set PLL0 to 64 MHz */
    ME.RUN[0].R = 0x0001F0074;      /* RUN0 cfg: 16MHzIRCON,OSC0ON,FLL0ON,syclk=PLL */
    ME.RUNPC[1].R = 0x000000010;    /* Perc. Cfg. 1 settings: only run in RUN0 mode */
    ME.PCTL[48].R = 0x01;           /* MPC56xxB/P/S LINFlex 0: select ME.RUNPC[1] */
    ME.PCTL[68].R = 0x01;           /* MPC56xxB/S SIUL: select ME.RUNPC[0] */
    /* Mode Transition to enter RUN0 mode: */
    ME.MCTLR = 0x40005AF0;         /* Enter RUN0 Mode & Key */
    ME.MCTLR = 0x4000A50F;         /* Enter RUN0 Mode & Inverted Key */
    while (ME.GS.B.S_MTRANS) {}    /* Wait for mode transition to complete */
    /* Note: could wait here using timer and/or I TC IRQ */
    while(ME.GS.B.S_CURRENTMODE != 4) {} /* Verify RUN0 is the current mode */
}

```c
void initPeriClkGen(void) {
    CGM.SC_DC[0].R = 0x80; /* MPC56xxB/S: Enable peri set 1 sysclk divided by 1 */
}
```

```c
void disableWatchdog(void) {
    SWT.SR.R = 0x0000C520; /* Write keys to clear soft lock bit */
    SWT.SR.R = 0x0000D928;
    SWT.CR.R = 0x8000010A; /* Clear watchdog enable (WEN) */
}
```

```c
void initLINFlex_0 (void) {
    LINFLEX_0.LINCR1.B.INIT = 1; /* Put LINFlex hardware in init mode */
    LINFLEX_0.LINCR1.R= 0x00000311; /* Configure module as LIN master & header */
    LINFLEX_0.LINIBRR.B.DIV M = 383; /* Mantissa baud rate divider component */
    LINFLEX_0.LINFBRR.B.DIV F = 16; /* Fraction baud rate divider component */
    LINFLEX_0.LINCR2.R= 0x00000310; /* Configure module as LIN master & header */
    SIU.PCR[18].R = 0x0400;         /* MPC56xxB: Configure port B2 as LIN0TX */
    SIU.PCR[19].R = 0x00103;        /* MPC56xxB: Configure port B3 as LIN0RX */
}
```

```c
void transmitLINframe (void) {
    LINFLEX_0.BDRM.R = 0x2020206F; /* Load buffer data most significant bytes */
    LINFLEX_0.BDRL.R = 0x6C6C6548; /* Load buffer data least significant bytes */
    LINFLEX_0.BIDR.R = 0x00001E35; /* Init header: ID=0x35, 8 B, Tx, enh. cksum*/
    LINFLEX_0.LINCR2.B.HTRQ = 1; /* Request header transmission */
}
```

```c
void main(void) {
    volatile uint32_t IdleCtr = 0;
    initModesAndClks();          /* Initialize mode entries */
    initPeriClkGen();            /* Initialize peripheral clock generation for LINFlex */
    disableWatchdog();           /* Disable watchdog */
    initLINFlex_0();             /* Initialize FLEXCAN 0 as master */
    transmitLINframe();          /* Transmit one frame from master */
    while (1) {IdleCtr++;}       /* Idle loop: increment counter */
}
# 17.3.2 MPC560xP

/* main.c: LINFlex program for MPC56xxP */
/* Description: Transmit one message from FlexCAN 0 buf. 0 to FlexCAN C buf. 1 */
/* Rev Oct 30 2009 SM - initial version */
/* Rev Mar 14 1020 SM - Modified initModesAndClks, updated header */

#include "jdp_pictus_0106.h" /* Use proper include file */

void initModesAndClks(void) {
    ME.MER.R = 0x0000001D;          /* Enable DRUN, RUN0, SAFE, RESET modes */
    /* Initialize PLL before turning it on: */
    /* Use 2 of the next 4 lines depending on crystal frequency: */
    /*CGM.CMU_0_CSR.R = 0x000000004;*//* Monitor FXOSC > FIRC/4 (4MHz); no PLL monitor */
    /*CGM.FMPLL[0].CR.R = 0x02400100;*/ /* 8 MHz xtal: Set PLL0 to 64 MHz */
    CGM.CMU_0_CSR.R = 0x000000000;  /* Monitor FXOSC > FIRC/1 (16MHz); no PLL monitor*/
    CGM.FMPLL[0].CR.R = 0x12400100; /* 40 MHz xtal: Set PLL0 to 64 MHz */
    ME.RUN[0].R = 0x001F0074;       /* RUN0 cfg: 16MHzIRCON,OSC0ON,PPLL0ON,syclk=PLL */
    ME.RUNPC[1].R = 0x00000010;     /* Peri.Cfg. 1 settings: only run in RUN0 mode */
    ME.PCTL[48].R = 0x01;           /* MPC56xxB/P/S LINFlex 0: select ME.RUNPC[1] */
    /* Mode Transition to enter RUN0 mode: */
    ME.MCTL.R = 0x40005AF0;         /* Enter RUN0 Mode & Key */
    ME.MCTL.R = 0x4000A50F;         /* Enter RUN0 Mode & Inverted Key */
    while (ME.GS.B.S_MTRANS) {}     /* Wait for mode transition to complete */
    /* Note: could wait here using timer and/or I_TC IRQ */
    while(ME.GS.B.S_CURRENTMODE != 4) {} /* Verify RUN0 is the current mode */
}

void initPeriClkGen(void) {
}

void disableWatchdog(void) {
    SWT.SR.R = 0x0000c520;     /* Write keys to clear soft lock bit */
    SWT.SR.R = 0x0000d928;
    SWT.CR.R = 0x8000010A;     /* Clear watchdog enable (WEN) */
}

void initLINFlex_0 (void) {
    LINFLEX_0.LINCR1.B.INIT = 1;    /* Put LINFlex hardware in init mode */
    LINFLEX_0.LINCR1.R= 0x00000311; /* Configure module as LIN master & header */
    LINFLEX_0.LINIBRR.B.DIV_M= 383; /* Mantissa baud rate divider component */
    LINFLEX_0.LINFBRR.B.DIV_F = 16; /* Fraction baud rate divider component */
    LINFLEX_0.LINCR1.R= 0x00000310; /* Configure module as LIN master & header */
    SIU.PCR[18].R = 0x0400;         /* MPC56xxP: Configure port B2 as LIN0TX */
    SIU.PCR[19].R = 0x0503;         /* MPC56xxP: Configure port B3 as LIN0RX */
    SIU.PSMI[31].R = 0;             /* MPC56xxP: LIN0 Pad select mux port B3 */
}

void transmitLINframe (void) {
    LINFLEX_0.BDRM.R = 0x2020206F;  /* Load buffer data most significant bytes */
    LINFLEX_0.BDRL.R = 0x6C6C6548;  /* Load buffer data least significant bytes */
    LINFLEX_0.BIDR.R = 0x000001E35; /* Init header: ID=0x35, 8 B, Tx, enh. cksum*/
    LINFLEX_0.LINCR2.B.HTRQ = 1;    /* Request header transmission */
}

void main(void) {
    volatile uint32_t IdleCtr = 0;

    initModesAndClks();       /* Initialize mode entries */
    initPeriClkGen();         /* Initialize peripheral clock generation for LINFlex */
    disableWatchdog();        /* Disable watchdog */
    initLINFlex_0();          /* Initialize FLEXCAN 0 as master */
    transmitLINframe();       /* Transmit one Frame from master */
    while (1) {IdleCtr++;}     /* Idle loop: increment counter */
}
17.3.3 MPC56xxS

/* main.c: LINFlex program for MPC56xxS*/
/* Description: Transmit one message from FlexCAN 0 buf. 0 to FlexCAN C buf. 1 */
/* Oct 30 2009 SM - initial version */
/* Mar 15 2010 SM - modified initModesAndClks, updated header */

#include "56xxS_0204.h" /* Use proper header file */

void initModesAndClks(void) {
  ME.MER.R = 0x00000001D; /* Enable DRUN, RUN0, SAFE, RESET modes */
  CGM.FMPLL[0].CR.R = 0x002400100; /* Initialize PLL before turning it on: */
  ME.RUN[0].R = 0x001F0074; /* RUN0 cfg: 16MHzIRCON,OSC0ON,PLL0ON,syclk=PLL */
  ME.RUNPC[1].R = 0x000000010; /* Peri. Cfg. 1 settings: only run in RUN0 mode */
  ME.PCTL[48].R = 0x0001; /* MPC56xxB/S LINFlex 0: select ME.RUNPC[1] */
  ME.PCTL[68].R = 0x0001; /* Mode Transition to enter RUN0 mode: */
  ME.MCTL.R = 0x40005AF0; /* Enter RUN0 Mode & Key */
  ME.MCTL.R = 0x4000A50F; /* Enter RUN0 Mode & Inverted Key */
  while (ME.GS.B.S_MTRANS) {} /* Wait for mode transition to complete */
  while(ME.GS.B.S_CURRENTMODE != 4) {} /* Verify RUN0 is the current mode */
}

void initPeriClkGen(void) {
  CGM.SC_DC[0].R = 0x80; /* MPC56xxB/S: Enable peri set 1 sysclk divided by 1 */
}

void disableWatchdog(void) {
  SWT.SR.R = 0x000c520; /* Write keys to clear soft lock bit */
  SWT.SR.R = 0x000d928; /* Clear watchdog enable (WEN) */
  SWT.CR.R = 0x8000010A; /* Clear watchdog enable (WEN) */
}

void initLINFlex_0 (void) {
  LINFLEX_0.LINCR1.B.INIT = 1; /* Put LINFlex hardware in init mode */
  LINFLEX_0.LINCR1.R= 0x00000311; /* Configure module as LIN master & header */
  LINFLEX_0.LINIBRR.B.DIV_M= 383; /* Mantissa baud rate divider component */
  LINFLEX_0.LINFBRR.B.DIV_F = 16; /* Fraction baud rate divider component */
  LINFLEX_0.LINCR1.R= 0x00000310; /* Configure module as LIN master & header */
  SIU.PCR[18].R = 0x0400; /* MPC56xxS: Configure port B2 as LIN0TX */
  SIU.PCR[19].R = 0x0503; /* MPC56xxS: Configure port B3 as LIN0RX */
}

void transmitLINframe (void) {
  LINFLEX_0.BDRM.R = 0x2020206F; /* Load buffer data most significant bytes */
  LINFLEX_0.BDRL.R = 0x6C6C6548; /* Load buffer data least significant bytes */
  LINFLEX_0.BIDR.R = 0x06000001E35; /* Init header: ID=0x35, 8 B, Tx, enh. cksum*/
  LINFLEX_0.LINCR2.B.HTRQ = 1; /* Request header transmission */
}

void main(void) {
  volatile uint32_t IdleCtr = 0;

  initModesAndClks(); /* Initialize mode entries */
  initPeriClkGen(); /* Initialize peripheral clock generation for LINFlex */
  disableWatchdog(); /* Disable watchdog */
  initLINFlex_0(); /* Initialize FLEXCAN 0 as master */
  transmitLINframe(); /* Transmit one frame from master */
  while (1) {IdleCtr++;} /* Idle loop: increment counter */
}
18 eMIOS: Modulus Counter, OPWM Functions

18.1 Description

Task: Provide two Output Pulse Width Modulation (OPWM) signals that are synchronized to a common counter bus. The common counter bus is a separate eMIOS channel, configured as a modulus counter.

The original modulus counter and OPWM modes in early MPC555x devices were later replaced by modulus counter buffered and OPWM buffered modes in some MPC555x devices and in all MPC551x devices. Code is provided to manage both cases.

The MPC563x lacks OPWMB mode on channel 1, so channel 2 is used in the example code. MPC56xxB/S use eMIOS_0.

Exercise: If using the MPC5500 Evaluation Board, connect an eMIOS output to the speaker (if available) and/or an LED or oscilloscope. Observe the outputs, then alter code to change the frequency or duty cycle.

Alternate exercise: Wire an eMIOS channel to an LED, and slow down the oscillations using code changes below. Observe pulses on LED.

1. Increase eMIOS Prescaler. Example: EMIOS_0.MCR.B.GPRE = 254;
2. Increase Channel 23 Modulus Counter value. Example: EMIOS_0.CH[23].CADR.R = 64000;
3. Increase Channel 21 falling edge value. Example: EMIOS_0.CH[21].CBDR.R = 32000;

Figure 31. eMIOS OPWM Example
### Table 68. MPC551x, MPC55xx Signals for eMIOS OPWM Example

<table>
<thead>
<tr>
<th>Signal</th>
<th>Pin Name</th>
<th>SIU PCR No.</th>
<th>Package Pin No.</th>
<th>Function Name</th>
<th>SIU PCR No.</th>
<th>Package Pin No.</th>
<th>EVB</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMIOS[0]</td>
<td>PC0</td>
<td>32</td>
<td>122 146</td>
<td>B11</td>
<td>EMIOS[0]</td>
<td>179</td>
<td>AD17 AF15 AB10 T4</td>
</tr>
</tbody>
</table>

### Table 69. MPC56xxB/S Signals for eMIOS OPWM Example

<table>
<thead>
<tr>
<th>Signal</th>
<th>MPC56xxB Family</th>
<th>MPC56xxS Family</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Port</td>
<td>SiU Pad Configuration &amp; Selection Registers (values in hex)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EMIOS_0[21]</td>
<td>E5</td>
<td>PCR69=0600</td>
</tr>
<tr>
<td>EMIOS_0[22]</td>
<td>E6</td>
<td>PCR70=0600</td>
</tr>
</tbody>
</table>
18.2 Design

Timing resources used will include:

- sysclk: 64 MHz — assume 8 MHz crystal unless noted
- eMIOS internal clock: Choose 1 MHz (requires prescaling sysclk by 64)
- eMIOS Channel 23: initialize in modulus counter mode, which will be used as the global counter bus, up-counting 1000 eMIOS internal clocks (use value of 1000 – 1 = 999)
- eMIOS Channels 0 and 1: OPWM mode based on Time Bus A, each channel with different duty cycles; the signal polarity will be rising edge for the first match, falling edge for the second match

18.2.1 Mode Use Summary (MPC56xxB/S only)

Mode Transition is required for changing mode entry registers. Hence even enabling the crystal oscillator to be active in the default mode (DRUN) requires enabling the crystal oscillator in appropriate mode configuration register (ME_xxxx_MC) then initiating a mode transition. This example transitions from the default mode after reset (DRUN) to RUN0 mode.

Table 70. Mode Configurations for MPC56xxB/S DSPI SPI to SPI Example

<table>
<thead>
<tr>
<th>Modes are enabled in ME_ME Register.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mode</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>DRUN</td>
</tr>
<tr>
<td>RUN0</td>
</tr>
</tbody>
</table>

Peripherals also have configurations to gate clocks on or off for different modes, enabling low power. The following table summarizes the peripheral configurations used in this example.

Table 71. Peripheral Configurations for MPC56xxB/S DSPI SPI to SPI Example

<table>
<thead>
<tr>
<th>Low power modes are not used in example.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peripheral Config.</td>
</tr>
<tr>
<td>-------------------</td>
</tr>
<tr>
<td>RUN3 RUN2 RUN1 RUN0</td>
</tr>
<tr>
<td>PC1</td>
</tr>
</tbody>
</table>

Other peripheral configurations are not used in example.
### 18.2.2 Design Steps

**Table 72. eMIOS Modulus Counter and Output Pulse Width (OPWM) Example**

<table>
<thead>
<tr>
<th>Step</th>
<th>Relevant Bit Field</th>
<th>Pseudo Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>init Modes and Clock (MPC56xxPBS only)</td>
<td>DRUN=1, RUN0 = 1</td>
<td>ME_ME = 0x0000 001D</td>
</tr>
</tbody>
</table>
| Initialize PLL0 dividers to provide 64 MHz for input crystal before mode configuration:  
- 8 MHz xtal: FMPLL[0]_CR=0x02400100  
- 40 MHz xtal: FMPLL[0]_CR=0x12400100 (MPC56xxP & 8 MHz xtal requires changing default CMU_CSR value. See PLL example.) | | 8 MHz Crystal:  
CMG_FMPLL[0]_CR =0x02400100 |
| Configure RUN0 Mode:  
- I/O Output power-down: no safe gating  
- Main Voltage regulator is on (default)  
- Data, code flash in normal mode (default)  
- PLL0 is switched on  
- Crystal oscillator is switched on  
- 16 MHz IRC is switched on (default)  
- Select PLL0 (system pll) as sysclk | PDO=0  
MVRON=1  
DFLAN=1  
CFLAON= 3  
PLL0ON=1  
XOSC0ON=1  
16MHz_IRCON=1  
SYSCLK=0x4 | ME_RUN0_MC = 0x001F 0070 |
| Assign peripheral configuration to peripherals:  
- SIUL: select ME_RUN_PC1 (MPC56xxB/S)  
- eMIOS 0: select ME_RUN_PC1 | RUN0_CFG = 1  
RUN_CFG = 1 | MC_PCTL68 = MC_PCTL72  
= 0x01 |
| Initiate software mode transition to RUN0 mode:  
- Mode & key, then mode & inverted key  
- Wait for transition to complete  
- Verify current mode is RUN0 | TARGET_MODE = RUN0  
S_TRANS = CURRENTMODE | ME_MCTL  
=0x4000 5AF0,  
=0x4000 A50F  
wait ME_GS [S_TRANS] = 0  
verify 4 = ME_GS [CURRENTMODE] |
| init Sysclk | Initialize sysclk to 64 MHz, running from PLL | See PLL Initialization example |
| init Peri Clk Gen (MPC56xxPBS only) | | - |
| Initialize peripheral clock generation (See appendix: MPC56xxB/P/S Peripheral Clocks)  
- MPC56xxB: Enable Peri. Set 2- sysclk div. by 1  
- MPC56xxS: Enable Aux Clk 1- PLL div. by 1 | | MPC56xxB:  
CMG_SC_DC=0x0000 8000  
MPC56xxS:  
CMG_AC_DC2=0x80 |
| disable Watchdog (S6xxPBS) | | See PLL Initialization example |
Table 72. eMIOS Modulus Counter and Output Pulse Width (OPWM) Example (continued)

<table>
<thead>
<tr>
<th>Step</th>
<th>Relevant Bit Field</th>
<th>Pseudo Code</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>MPC551x</td>
</tr>
<tr>
<td>initEMIOS</td>
<td>Config eMIOS for 1 MHz internal clock:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Set global prescaler = divide by 64 (63 + 1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Do not use external time base</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Enable global prescaler &amp; internal clock</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Enable global time base</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Enable freezing counters during debug</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GPRE = 63(0x3F)</td>
<td>EMIOS_</td>
</tr>
<tr>
<td></td>
<td>ETB = 0 (default)</td>
<td>MCR =</td>
</tr>
<tr>
<td></td>
<td>GPREN = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GTBE = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FRZ = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GPREN = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GTBE = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FRZ = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>init EMIOS ch 23</td>
<td>Set Channel's A match data for 1000 counts (Match value used for modulus counter)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>A = 999</td>
<td>EMIOS_</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CH[23]CADR =</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Channel 23: Enable channel as up counter</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Mode (551x, 56xxB/S) = mod. up counter but'd</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Mode (555x) = mod. up counter</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Counter bus select = internal counter</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Channel prescaler = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Enable channel prescaler</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Enable freezing count in debug mode</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MODE= 0x50(551x)</td>
<td>EMIOS_</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CH[23]CCR =</td>
</tr>
<tr>
<td></td>
<td>MODE= 0x10(555x)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BSL = 3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>UCPR = 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>UCPR = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FREN = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BSL=0 (default)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EDPOL=1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mode=0x60 (551x)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>or (555x)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mode=0x20 (555x)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BSL=0 (default)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EDPOL=1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mode=0x60 (551x)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>or (555x)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mode=0x20 (555x)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>init EMIOS</td>
<td>Set Channel's A match Data value = 250</td>
<td></td>
</tr>
<tr>
<td>ch 0</td>
<td></td>
<td>A = 250</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Set Channel's B match Data value = 500</td>
<td></td>
</tr>
<tr>
<td></td>
<td>B = 500</td>
<td>EMIOS_</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CH[0]CBDR</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Set up Channel Control:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• BSL = Bus selected is counter bus A</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• EDPOL = leading edge sets; trailing clears</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Mode (551x, 563x, 56xxB/S) = Output PWMB</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Mode (MPC555x) = Output PWM</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BSL=0 (default)</td>
<td>EMIOS_</td>
</tr>
<tr>
<td></td>
<td>EDPOL=1</td>
<td>CH[0]CCR =</td>
</tr>
<tr>
<td></td>
<td>Mode=0x60 (551x)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>or (555x)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mode=0x20 (555x)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SIU_PCR[32] =</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x0600</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SIU_PCR[179] =</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x0E00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>See table:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MPC551x</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MPC56xxB/P/S Signals</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Signals</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Configure pad for eMIOS channel 0:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Pad assignment = EMIOS Ch 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Pad output buffer enabled</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PA = 1 (MPC551x)</td>
<td>EMIOS_</td>
</tr>
<tr>
<td></td>
<td>PA = 3 (MPC555x)</td>
<td>CH[0]CCR =</td>
</tr>
<tr>
<td></td>
<td>OBE = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Configure pad for eMIOS channel 1:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Pad assignment = EMIOS Ch 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Pad output buffer enabled</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PA = 1 (MPC551x)</td>
<td>EMIOS_</td>
</tr>
<tr>
<td></td>
<td>PA = 3 (MPC555x)</td>
<td>CH[1]CCR =</td>
</tr>
<tr>
<td></td>
<td>OBE = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>init EMIOS</td>
<td>Set Channel's A match Data value = 500</td>
<td></td>
</tr>
<tr>
<td>ch 1</td>
<td>(MPC563x channel 2 is used because channel 1 lacks OPWMB mode)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>A = 500</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Set Channel's B match Data value = 999</td>
<td></td>
</tr>
<tr>
<td></td>
<td>B = 999</td>
<td>EMIOS_</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Set up Channel Control:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• BSL = Bus selected is counter bus A</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• EDPOL = leading edge sets; trailing clears</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Mode (551x, 563x, 56xxB/S) = Output PWMB</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Mode (MPC555x) = Output PWM</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BSL=0 (default)</td>
<td>EMIOS_</td>
</tr>
<tr>
<td></td>
<td>Mode=0x60 (551x)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>or (555x)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mode=0x20 (555x)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SIU_PCR[32] =</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x0600</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SIU_PCR[180] =</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x0E00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>See table:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MPC56xxB/S Signals</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Signals</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
18.2.3 Design Screenshot

The screenshot below shows the two OPWM channels as a result of this design. Both channels schedule leading and trailing edges based on the counter of channel 23, which counts to 1000 μs. Hence the OPWM frequency is 1 kHz.
18.3 Code

18.3.1 MPC551x, MPC555x

/* main.c - eMIOS OPWM example */
/* Description: eMIOS example using Modulus Counter and OPWM modes */
/* Rev 1.0 Sept 9 2004 S.Mihalik */
/* Rev 1.1 April 13 2006 S.M.- corrected GPRE to be div by 12 instead of 13*/
/* Rev 1.2 June 26 1006 S.M. - updated comments & made i volatile uint32_t */
/* Rev 1.3 July 19 2007 SM- Changes for MPC551x, 50 MHz sysclk, Mod Ctr data value*/
/* Rev 1.4 Aug 10 2007 SM - Changed to use sysclk of 64 MHz */
/* Rev 1.5 Jun 04 2008 SM - initSysclk changed for MPC5633M support */
/* Notes: */
/* 1. MMU not initialized; must be done by debug scripts or BAM */
/* 2. SRAM not initialized; must be done by debug scripts or in a crt0 type file */
#include "mpc554.h" /* Use proper include file such as mpc5510.h or mpc554.h */

void initSysclk (void) {
    /* MPC551x: Use the next 6 lines */
    /* CRP.CLKSR.C.B.XOSCEN = 1; */
    /* FMPLL.EYNSCR.B=R = 0x00000006; */
    /* while (FMPLL.SYNSR.B.LOCK != 1) {}; */
    /* FMPLL.EYNSCR.B=R = 0x00000005; */
    /* SIU.SYSCLK.C.B.YSYCLKSEL = 2; */
    /* MPC563x: Use the next line */
    /* FMPLL.ESYNCR1.B.CLKCFG = 0X7; */
    /* MPC555x including MPC563x: use the next 3 lines for either 8 or 40 MHz crystal */
    /* FMPLL.SYNCR.R = 0x16080000; */
    /* 8 MHz xtal: 0x16080000; 40MHz: 0x46100000 */
    while (FMPLL.SYNSR.B.LOCK != 1) {}; /* Wait for PLL to LOCK */
    FMPLL.SYNCR.R = 0x16000000; /* 8 MHz xtal: 0x16000000; 40MHz: 0x46080000 */
}

void initEMIOS(void) {
    EMIOS.MCR.C.B.GPRE= 63; /* Divide 64 MHz sysclk by 63+1 = 64 for 1MHz eMIOS clk*/
    EMIOS.MCR.C.B.ETB = 0; /* External time base is disabled; Ch 23 drives ctr bus A */
    EMIOS.MCR.C.B.GPREN = 1; /* Enable eMIOS clock */
    EMIOS.MCR.C.B.GTBE = 1; /* Enable global time base */
    EMIOS.MCR.C.B.FRZ = 1; /* Enable stopping channels when in debug mode */
}

void initEMIOSch23(void) { /* EMIOS CH 23: Modulus Up Counter */
    EMIOS.CH[23].CADR.R = 999; /* Period will be 999+1 = 1000 clocks (1 msec) */
    EMIOS.CH[23].CCR.B.MODE = 0x10; /* MPC555x: Mode is OPWM */
    EMIOS.CH[23].CCR.B.BSL = 0x3; /* Use internal counter */
    EMIOS.CH[23].CCR.B.UCPRE=1; /* Set channel prescaler to divide by 1 */
    EMIOS.CH[23].CCR.B.UCPREN = 1; /* Freeze channel counting when in debug mode */
    EMIOS.CH[23].CCR.B.UCPREN = 1; /* Enable prescaler; uses default divide by 1 */
}

void initEMIOSch0(void) { /* EMIOS CH 0: Output Pulse Width Modulation */
    EMIOS.CH[0].CADR.R = 250; /* Leading edge when channel counter bus=250*/
    EMIOS.CH[0].CBDR.R = 500; /* Trailing edge when channel counter bus=500*/
    EMIOS.CH[0].CCR.B.BSL = 0x0; /* Use counter bus A (default) */
    EMIOS.CH[0].CCR.B.EDPOL = 0; /* Polarity-leading edge sets output/trailing clears */
    EMIOS.CH[0].CCR.B.MODE = 0x020; /* MPC555x: Mode is OPWM */
    EMIOS.CH[0].CCR.B.UCPRE=1; /* Use one of the following 2 lines */
    EMIOS.CH[0].CCR.B.UCPREN = 1; /* MPC551x: Initialize pad for eMIOS chan. 0 output */
    EMIOS.CH[179].R = 0x0E00; /* MPC555x: Initialize pad for eMIOS chan. 0 output */
}
void initEMIOSch1(void) { /* EMIOS CH 1: Output Pulse Width Modulation */
  EMIOS.CH[1].ADDR.R = 500; /* Leading edge when channel counter bus=500*/
  EMIOS.CH[1].CBDR.R = 999; /* Trailing edge when channel's counter bus=999*/
  EMIOS.CH[1].CCR.B.BSL = 0x0; /* Use counter bus A (default) */
  EMIOS.CH[1].CCR.B.EPDOL = 1; /*Polarity-leading edge sets output/trailing clears*/
  /* Use one of the following two lines for mode (Some MPC555x devices lack OPWMB) */
  /* EMIOS.CH[1].CCR.B.MODE = 0x60; */ /* MPC551x, MPC563x: Mode is OPWM Buffered */
  EMIOS.CH[1].CCR.B.MODE = 0x20; /* MPC555x: Mode is OPWM */
  /* Use one of the following 2 lines: */
  /* SIU.PCR[33].R = 0x0600; */ /* MPC551x: Initialize pad for eMIOS chan. 1 output */
  SIU.PCR[180].R = 0x0E00; /* MPC555x: Initialize pad for eMIOS chan. 1 output */
}

void main (void) {
  volatile uint32_t i = 0; /* Dummy idle counter */
  initSysclk(); /* Set sysclk = 50MHz running from PLL */
  initEMIOS(); /* Initialize eMIOS to provide 1 MHz clock to channels */
  initEMIOSch23(); /* Initialize eMIOS channel 23 as modulus counter*/
  initEMIOSch0(); /* Initialize eMIOS channel 0 as OPWM, using ch 23 as time base */
  initEMIOSch1(); /* Initialize eMIOS channel 1 as OPWM, using ch 23 as time base */
  while (1) {i++; } /* Wait forever */
}

18.3.2 MPC56xxB/S (MPC56xxB shown with 8 MHz crystal)

/* main.c - eMIOS OPWM example */
/* Description: eMIOS example using Modulus Counter and OPWM modes */
/* Rev 1.0 Sept 9 2004 S.Mihalik */
/* Rev 1.1 April 13 2006 S.M.- corrected GPRE to be div by 12 instead of 13*/
/* Rev 1.2 June 26 1006 S.M. - updated comments & made i volatile uint32_t t */
/* Rev 1.3 July 19 2007 SM- Changes for MPC551x, 50 MHz sysclk, Mod Ctr data value*/
/* Rev 1.4 Aug 10 2007 SM - Changed to use sysclk of 64 MHz */
/* Rev 1.5 Jun 04 2008 SM - initSysclk changed for MPC5633M support */
/* Rev 1.6 May 22 2009 SM - modified for MPC56xxB/S */
/* Rev 1.7 Jun 24 2008 SM - simplified code */
/* Rev 1.8 Mar 14 2010 SM - modified initModesAndClock, updated header file */
/* Copyright Freescale Semiconductor, Inc. 2004-2010 All rights reserved. */
#include "MPC5604B_0M27V_0102.h" /* Use proper include file */
  uint32_t i = 0; /* Dummy idle counter */
void initModesAndClock(void) { /* Enable DRUN, RUN0, SAFE, RESET modes */
  ME.MER.R = 0x0000001D; /* Initialize PLL before turning it on: */
  /* Use 1 of the next 2 lines depending on crystal frequency: */
  CGM.FMPLL_CR.R = 0x02400100; /* Set PLL0 to 64 MHz */
  /* CGM.FMPLL_R = 0x12400100; */
  /* 40 MHz xtal: Set PLL0 to 64 MHz */
  ME.RUN[0].R = 0x001F0074; /* RUN0 config: 16MHzIRCON,OSC0ON,PLL00N,syclk=PLL */
  ME.RUNPC[1].R = 0x000000010; /* Only run in RUN0 mode */
  ME.PCTL[68].R = 0x01; /* MPC56xxB/S SIUL: select ME.RUNPC[1] */
  ME.PCTL[72].R = 0x01; /* MPC56xxB/S EMIOS 0: select ME.RUNPC[1] */
  while (ME.GS.B.S_MTRANS) {} /* Mode Transition to enter RUN0 mode: */
  ME.MCTRL.R = 0x40005AF0; /* Enter RUN0 Mode & Key */
  ME.MCTRL.R = 0x4000A50F; /* Enter RUN0 Mode & Inverted Key */
  /* Note: could wait here using timer and/or I_TC IRQ */
  while(ME.GS.B.S_CURRENTMODE != 4) {} /* Verify RUN0 is the current mode */
}

void initPeriClkGen(void) { /* MPC56xxB: Enable peri set 3 sysclk divided by 1*/
  CGM.SC_DC[2].R = 0x80;
}
void disableWatchdog(void) {
void initEMIOS_0(void) {
    EMIOS_0.MCR.B.GPRE= 63;   /* Divide 64 MHz sysclk by 63+1 = 64 for 1MHz eMIOS clk*/
    EMIOS_0.MCR.B.GPREN = 1; /* Enable eMIOS clock */
    EMIOS_0.MCR.B.GTBE = 1;  /* Enable global time base */
    EMIOS_0.MCR.B.FRZ = 1; /* Enable stopping channels when in debug mode */
}

void initEMIOS_0ch23(void) {
    EMIOS_0.CH[23].CADR.R = 999;      /* Period will be 999+1 = 1000 clocks (1 msec)*/
    EMIOS_0.CH[23].CCR.B.MODE = 0x50; /* Modulus Counter Buffered (MCB) */
    EMIOS_0.CH[23].CCR.B.BSL = 0x3;   /* Use internal counter */
    EMIOS_0.CH[23].CCR.B.UCPRE=0;     /* Set channel prescaler to divide by 1 */
    EMIOS_0.CH[23].CCR.B.UCPEN = 1;   /* Enable prescaler; uses default divide by 1*/
    EMIOS_0.CH[23].CCR.B.FREN = 1;    /* Freeze channel counting when in debug mode*/
}

void initEMIOS_0ch21(void) {
    EMIOS_0.CH[21].CADR.R = 250;      /* Leading edge when channel counter bus=250*/
    EMIOS_0.CH[21].CBDR.R = 500;      /* Trailing edge when channel counter bus=500*/
    EMIOS_0.CH[21].CCR.B.BSL = 0x0;  /* Use counter bus A (default) */
    EMIOS_0.CH[21].CCR.B.EDPOL = 1; /* Polarity-leading edge sets output */
    EMIOS_0.CH[21].CCR.B.MODE = 0x60; /* Mode is OPWM Buffered */
    SIU.PCR[69].R = 0x0600;           /* MPC56xxS: Assign EMIOS_0 ch 21 to pad */
}

void initEMIOS_0ch22(void) {
    EMIOS_0.CH[22].CADR.R = 500;    /* Leading edge when channel counter bus=500*/
    EMIOS_0.CH[22].CBDR.R = 999;    /* Trailing edge when channel's counter bus=999*/
    EMIOS_0.CH[22].CCR.B.BSL = 0x0; /* Use counter bus A (default) */
    EMIOS_0.CH[22].CCR.B.EDPOL = 1; /* Polarity-leading edge sets output*/
    EMIOS_0.CH[22].CCR.B.MODE = 0x60; /* Mode is OPWM Buffered */
    SIU.PCR[70].R = 0x0600;          /* MPC56xxS: Assign EMIOS_0 ch 22 to pad */
}

void main (void) {
    volatile uint32_t i = 0; /* Dummy idle counter */
    initModesAndClock();    /* Initialize mode entries and system clock */
    initPeriClkGen();       /* Initialize peripheral clock generation for DSPIs */
    disableWatchdog();      /* Disable watchdog */
    initEMIOS_0();          /* Initialize eMIOS 0 to provide 1 MHz clock to channels */
    initEMIOS_0ch23();      /* Initialize eMIOS 0 channel 23 as modulus counter*/
    initEMIOS_0ch21();      /* Initialize eMIOS 0 channel 0 as OPWM, ch 23 as time base */
    initEMIOS_0ch22();      /* Initialize eMIOS 0 channel 1 as OPWM, ch 23 as time base */
    while (1) i++;           /* Wait forever */
}
19 eMIOS: PEC, OPWFM Functions

19.1 Description

**Task:** Count the number of input pulses in a time window using an eMIOS channel in pulse edge counting (PEC) mode. Input pulses will be generated by an eMIOS channel in output pulse width and frequency modulation (OPWFM) mode. The time base used for the time window is based on eMIOS channel 23 in modulus counter, as in Section 18, “eMIOS: Modulus Counter, OPWM Functions.” Also per that example, the window open and close times used for the PEC channel are replicated by a separate channel in OPWM mode, for observation purposes.

**Exercise:** Connect eMIOS channels 2 and 3 externally. Optionally, connect an oscilloscope to observe the pulses and their mirrored counting window, channel 2. Execute the code and verify four pulses were counted in the window. Then alter code to reduce the timing window 50% and verify the result.

**Figure 32. eMIOS PEC and OPWFM Example**

**Table 73. Signals for eMIOS PEC and OPWFM Example**

<table>
<thead>
<tr>
<th>Signal</th>
<th>Function Name</th>
<th>SIU PCR No.</th>
<th>Package Pin No.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>496 BGA</td>
</tr>
<tr>
<td>EMIOS[0]</td>
<td>EMIOS[0]</td>
<td>179</td>
<td>AD17</td>
</tr>
</tbody>
</table>
19.2 Design

Timing resources used will include:

- sysclk: 12 MHz: assume 8 MHz crystal and use default 1.5 multiplier
- eMIOS internal clock: Choose 1 MHz (requires prescaling sysclk by 12)
- eMIOS channel 2: Generate a 10 kHz signal using OPWFM mode
- eMIOS channel 3: Count input pulses using PEC mode; use eMIOS channel 23 for timebase
- eMIOS channel 23: Initialize in modulus counter mode, counting 1000 eMIOS internal clocks
- eMIOS channel 0: OPWM mode based on Time Bus A — the output pulse will have the same timing parameters as the PEC window of eMIOS channel 3

19.2.1 Steps and Pseudo Code

Table 74. eMIOS Pulse Edge Counting (PEC) and Output Pulse Width and Frequency Modulation (OPWFM)

<table>
<thead>
<tr>
<th>Step</th>
<th>Relevant Bit Field</th>
<th>Pseudo Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>initEMIOS</td>
<td>Config eMIOS for 1 MHz internal clock:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Set global prescaler (GPRE+1) = div. by 12</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Use channel 23 for Time Base A, not external</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Enable global prescaler and internal clock</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Disable freezing counters during debug</td>
<td>EMIOS_MCR = 0x0400 0B00</td>
</tr>
<tr>
<td></td>
<td>GPRE = 0xB</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ETB = 0 (default)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GPREN = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FRZ = 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EMIOS_MCR = 0x0400 0B00</td>
<td></td>
</tr>
<tr>
<td>init EMIOS ch 2</td>
<td>Set period = 100 eMIOS clocks (1 µs each)</td>
<td>EMIOS_CH[2]CBDR = 99</td>
</tr>
<tr>
<td>(10 kHz OPWFM)</td>
<td>Set duty cycle = 20 eMIOS clocks (1 µs each)</td>
<td>EMIOS_CH[2]CADR = 19</td>
</tr>
<tr>
<td></td>
<td>Set up Channel Control:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Channel counter's prescaler = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Enable counter's prescaler</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Set polarity to be active high</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Mode = OPWFM, next period update, flag at B</td>
<td>EMIOS_CH[2]CCR = 0x0200 0099</td>
</tr>
<tr>
<td></td>
<td>UCPRE = 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>UCPREN = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EDPOL = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MODE = 0x19</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EMIOS_CH[2]CCR = 0x0200 0099</td>
<td></td>
</tr>
<tr>
<td>Configure GPIO 181:</td>
<td>Pad assignment = EMIOS Ch</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Pad output buffer enabled</td>
<td>SIU_PCR[181] = 0xE00</td>
</tr>
<tr>
<td></td>
<td>PA = 3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OBE = 1</td>
<td></td>
</tr>
<tr>
<td>init EMIOS ch 23</td>
<td>Set modulus counter to match after 1000 clks (Match value used for modulus counter)</td>
<td>EMIOS_CH[23]CADR = 999</td>
</tr>
<tr>
<td>(Modulus Counter)</td>
<td>A = 999</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Set up Channel Control:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Mode = modulus up counter, using eMIOS internal clk</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Counter bus select = internal</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Set channel prescale to divide by</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Enable channel prescaler</td>
<td>EMIOS_CH[23]CCR = 0x0200 0610</td>
</tr>
<tr>
<td></td>
<td>MODE = 0x10</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BSL = 3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>UCPRE = 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>UCPREN = 1</td>
<td></td>
</tr>
</tbody>
</table>
Table 74. eMIOS Pulse Edge Counting (PEC) and Output Pulse Width and Frequency Modulation (OPWFM)

<table>
<thead>
<tr>
<th>Step</th>
<th>Relevant Bit Field</th>
<th>Pseudo Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>init EMIOS ch 3 (PEC)</td>
<td>Define end of pulse counting window = 650</td>
<td>EMIOS_CH[3]CBDR = 650</td>
</tr>
<tr>
<td></td>
<td>Set up Channel Control:</td>
<td>EMIOS_CH[3]CCR = 0x000C 000A</td>
</tr>
<tr>
<td></td>
<td>• Use counter bus A for defining window</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Count single edge trigger for counting</td>
<td>BSL = 0</td>
</tr>
<tr>
<td></td>
<td>• Count falling edges</td>
<td>EDSEL = 0</td>
</tr>
<tr>
<td></td>
<td>• Use main clock for input filtering</td>
<td>EDPOL = 0</td>
</tr>
<tr>
<td></td>
<td>• Use two clock periods for input filter</td>
<td>FCK = 1</td>
</tr>
<tr>
<td></td>
<td>• MODE = PEC, continuous mode</td>
<td>IF = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MODE = 0x0A</td>
</tr>
<tr>
<td></td>
<td>Define start of pulse counting window = 250</td>
<td>EMIOS_CH[3]CADR = 250</td>
</tr>
<tr>
<td></td>
<td>(note: writing to this register after mode is set)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Configure GPIO 182</td>
<td>SIU_PCR[182] = 0x0E00</td>
</tr>
<tr>
<td></td>
<td>• Pad assignment = EMIOS Ch 3</td>
<td>PA = 3</td>
</tr>
<tr>
<td></td>
<td>• Enable input buffer</td>
<td>IBE = 1</td>
</tr>
<tr>
<td></td>
<td>init EMIOS ch 0 (OPWM)</td>
<td>Set Channel’s A match data value = 250</td>
</tr>
<tr>
<td></td>
<td>Set Channel’s B match data value = 650</td>
<td>EMIOS_CH[0]CBDR = 650</td>
</tr>
<tr>
<td></td>
<td>Set up Channel Control:</td>
<td>EMIOS_CH[0]CCR = 0x0000 00A0</td>
</tr>
<tr>
<td></td>
<td>• BSL = Bus selected is counter bus A</td>
<td>BSL = 0 (default)</td>
</tr>
<tr>
<td></td>
<td>• EDPOL = leading edge sets; trailing clears</td>
<td>EDPOL = 1</td>
</tr>
<tr>
<td></td>
<td>• Mode = Output PWM, use immediate update</td>
<td>MODE = 0x20</td>
</tr>
<tr>
<td></td>
<td>Configure GPIO 179:</td>
<td>SIU_PCR[179] = 0x0E00</td>
</tr>
<tr>
<td></td>
<td>• Pad assignment = EMIOS Ch 0</td>
<td>PA = 3</td>
</tr>
<tr>
<td></td>
<td>• Enable output buffer</td>
<td>OBE = 1</td>
</tr>
<tr>
<td></td>
<td>Start timers</td>
<td>Start eMIOS (and eTPU) timers counting</td>
</tr>
<tr>
<td></td>
<td></td>
<td>EMIOS_MCR[GTBE] = 1</td>
</tr>
<tr>
<td></td>
<td>Read number of pulses</td>
<td>Wait for channel flag to indicate end of window</td>
</tr>
<tr>
<td></td>
<td>Read number of pulses counted</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Clear PEC flag</td>
<td>Write one to FLAG bit to clear it</td>
</tr>
<tr>
<td></td>
<td></td>
<td>EMIOS_CH[3]CCR[FLAG] = 1</td>
</tr>
</tbody>
</table>
19.2.2 Design Screenshots

Stopped 1K Samples 193 mV

eMIOS Chan 2 0PWFM: 10KHz, 20% duty cycle

eMIOS Chan 0 OPWM: Mirrors Ch. 3 Pulse Edge Counting window

Sa/Sec 250.0 KSa/s
T/Div 200 us
V/Div A 0.2 B 0.2
19.3 Code

/* main.c - eMIOS example focusing on PEC and OPWFM*/
/* Description: Configures 1MHz eMIOS clock, Ch2 OPWFM at 10kHz, */
/* Ch 23 as mod ctr of 1MHz in & counts to 1000, Ch 3 as PEC */
/* Rev 1.0 Apr 20 2006, S. Mihalik - Initial version */
/* Rev 1.1 Jul 16 2007 SM - Corrected OPWFM B, A values to 99, 19 from 100, 20 */
/* and OPWM period to 999 from 1000 */
/* Copyright Freescale Semiconductor, Inc. 2006 All rights reserved. */
/* Notes: */
/* 1. MMU not initialized; must be done by debug scripts or BAM */
/* 2. SRAM not initialized; must be done by debug scripts or in a crt0 type file */
#include "mpc5554.h"

void initEMIOS(void) {
    EMIOS.MCR.B.GPRE= 0xB;    /* eMIOS clk= sysclk/(GPRE+1)= 12 MHz/12= 1MHz */
    EMIOS.MCR.B.ETB = 0;      /* Ext. time base is disabled; Ch 23 drives ctr bus A */
    EMIOS.MCR.B.GPREN = 1;    /* Enable eMIOS clock */
    EMIOS.MCR.B.FRZ = 0;      /* Disable freezing channel counters in debug mode */
}

void initEMIOSch2(void) {       /* EMIOS CH 2: Output Pulse Width & Freq Modulation*/
    /* Input clock is eMIOS clk of 1MHz (1 usec period)*/
    EMIOS.CH[2].CBDR.R = 99;     /* Period = 1 usec x (99+1) = 100 usec, 10kHz*/
    EMIOS.CH[2].CADR.R = 19;      /* Duty cycle = 1 usec x (19+1) = 20 usec (20%) */
    EMIOS.CH[2].CCR.B.UCPRE = 0;  /* Channel counter uses divide by (0+1) prescaler */
    EMIOS.CH[2].CCR.B.UCPREN = 1; /* Channel counter's prescaler is loaded & enabled*/
    EMIOS.CH[2].CCR.B.EDPOL = 1;  /* Polarity is active high */
    EMIOS.CH[2].CCR.B.MODE= 0x19; /* Mode= 0PWFM, next period update, flag on B match*/
    SIU.PCR[181].B.PA = 3;        /* Initialize pad for eMIOS channel. */
    SIU.PCR[181].B.OBE = 1;       /* Initialize pad AC16 for output */
}

void initEMIOSch23(void) {        /* EMIOS CH 23: Modulus Up Counter: */
    /* Use internal counter */
    EMIOS.CH[23].CADR.R = 999;  /* Counter period= (999+1) clks x 10 usec/clk= 1 msec*/
    EMIOS.CH[23].CCR.B.MODE = 0x10; /* Mode is Modulus Counter, internal clock */
    EMIOS.CH[23].CCR.B.BSL = 0x3;   /* Use internal counter */
    EMIOS.CH[23].CCR.B.UCPRE=0;     /* Set channel prescaler to divide by 1 */
    EMIOS.CH[23].CCR.B.UCPREN = 1;  /* Enable prescaler; uses default divide by 1 */
}

void initEMIOSch3(void) {        /* EMIOS CH 3: Pulse Edge Counting, single shot */
    /* Count window closes when counter bus=650*/
    EMIOS.CH[3].CBDR.R = 650;      /* Count window closes when counter bus=650*/
    EMIOS.CH[3].CCR.B.BSL = 0x0;   /* Use counter bus A which is eMIOS Ch 23 */
    EMIOS.CH[3].CCR.B.EDSEL = 0;   /* Edge Select- Single edge trigger (count) */
    EMIOS.CH[3].CCR.B.FCK = 1;     /* Input filter will use main clock */
    EMIOS.CH[3].CCR.B.IF = 1;      /* Input filger uses 2 clock periods */
    EMIOS.CH[3].CCR.B.MODE = 0xA;  /* Mode is PEC, continuous */
    EMIOS.CH[3].CADR.R = 250;      /* Count window opens when counter bus=250*/
    SIU.PCR[182].B.PA = 3;         /* Initialize pad for eMIOS channel */
    SIU.PCR[182].B.OBE = 1;        /* Initialize pad for input */
}

void initEMIOSch0(void) {        /* EMIOS CH 0: Output Pulse Width Modulation */
    /* Mirror ch 3 PEC window for observation */
    EMIOS.CH[0].CBDR.R = 250;      /* Leading edge occurs when counter bus = 250 */
    EMIOS.CH[0].CCR.B.BSL = 0x0;   /* Trailing edge occurs when counter bus = 650 */
    EMIOS.CH[0].CCR.B.EDPOL = 1;   /* Polarity-leading edge sets output */
    EMIOS.CH[0].CCR.B.MODE = 0x20; /* Mode is OPWM */
    SIU.PCR[179].B.PA = 3;         /* Initialize pad for eMIOS channel */
    SIU.PCR[179].B.OBE = 1;        /* Initialize pad for output */
}
void main (void) {
    uint32_t i = 0;             /* Dummy idle counter */
    vuint32_t NoOfPulses = 0;   /* Number of pulses counted in PEC function of eMIOS */

    initEMIOS();          /* Init. eMIOS to provide 1 MHz clock to eMIOS channels */
    initEMIOSch2();       /* Init. eMIOS channel 2 for 10kHz OPWFM */
    initEMIOSch23();      /* Init. eMIOS channel 23 as 1K modulus counter*/
    initEMIOSch3();       /* Init. eMIOS channel 3 for PEC, using ch 23 as time base */
    initEMIOSch0();       /* Init. eMIOS channel 0 as OPWM to mirror ch 3 PEC window */

    EMIOS.MCR.B.GTBE = 1;  /* Start timers/counters by enabling global time base */

    while (EMIOS.CH[3].CSR.B.FLAG == 0) {} /* Wait for flag at end of window */
    NoOfPulses = EMIOS.CH[3].CCNTR.R;    /* Read number of pulses counted */
    EMIOS.CH[3].CSR.B.FLAG = 1;          /* Clear flag */
    while (1) {i++; }                     /* Wait forever */
}
eTPU: Set 1 PWM Function

20.1 Description

**Task:** Using the existing Set 1 eTPU functions from the Freescale web site, build an image that loads them to eTPU RAM. Then assign and use the PWM function on eTPU A channel 5, initially at 1 kHz with 25% duty cycle, then update to 2 kHz with 60% duty cycle.

This example uses the Freescale-provided eTPU utilities, eTPU code image, and eTPU application program interface for the PWM function. For further information, see application note AN2864, *General C Functions for the eTPU.*

**Exercise:** If using the MPC555x evaluation board, connect eTPU A channel 5 to the speaker, LED, or oscilloscope. Step through code and observe PWM frequency change. Then alter code to configure eTPU A channel 2 for PWM at a frequency of your choice.

---

**Figure 33. eTPU Set 1 PWM Function Example**

**Table 75. Signals for eTPU Set 1 PWM Example**

<table>
<thead>
<tr>
<th>Signal</th>
<th>Function Name</th>
<th>SIU PCR No.</th>
<th>Package Pin No.</th>
<th>EVB</th>
</tr>
</thead>
<tbody>
<tr>
<td>eTPUA5</td>
<td>ETPUA5</td>
<td>119</td>
<td>H9 L4 K4 M4</td>
<td>PJ9–6</td>
</tr>
<tr>
<td>eTPUA2</td>
<td>ETPUA2</td>
<td>116</td>
<td>M3 M3 K3 P2</td>
<td>PJ9–3</td>
</tr>
</tbody>
</table>

---

Qorivva Simple Cookbook, Rev. 4
20.2 Design

Timing resources used will include:

- sysclk = 64MHz: assume 8 MHz crystal, and use code from Section 10, “PLL: Initializing System Clock (MPC551x, MPC55xx),” to generate.
- eTPU TCR1 clock: Count at 1 MHz rate.

20.2.1 Steps and Pseudo Code

Table 76. Initialization: eTPU Using Set 1 Function Example

<table>
<thead>
<tr>
<th>Step</th>
<th>Relevant Bit Field or Structure</th>
<th>Pseudo Code / Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>init FMPLL</td>
<td>Set sysclk = 64&lt;br&gt;(See Section 10, “PLL: Initializing System Clock (MPC551x, MPC55xx)”)&lt;br&gt;Note for 40 MHz Crystal used on MPC555x — Replace FMPLL_SYNCR values:&lt;br&gt;0x1608 0000 with 0x4610 0000&lt;br&gt;0x1600 0000 with 0x4608 0000</td>
<td>FMPLL_SYNCR = 0x1608 0000&lt;br&gt;Wait for FMPLL_SYNSR [LOCK] = 1;&lt;br&gt;FMPPLL_SYNCR = 0x1600 0000&lt;br&gt;Also for MPC563x:&lt;br&gt;FMPPLL_ESYNCR[CLKCFG] = 7</td>
</tr>
<tr>
<td>init eTPU</td>
<td>Configure eTPU A for:&lt;br&gt;• MISC not used&lt;br&gt;• eTPUA Input filter clock divided by 8&lt;br&gt;• eTPUA Channel input filter uses 3 samples&lt;br&gt;• eTPUA TCR1 = sysclk/2 prescaled by 32&lt;br&gt;• eTPUA TCR2 = sysclk/8 prescaled by 8&lt;br&gt;• eTPUB configurations as desired</td>
<td>etpu_config_t&lt;br&gt;fs_etpu_init</td>
</tr>
</tbody>
</table>
| init eTPUA[5] PWM | • Channel = eTPUA[5]<br>• Priority = middle<br>• Frequency = 1000 Hz<br>• Duty cycle = 25%
• Timebase = TCR1<br>• Timebase frequency = 1 MHz | –<br>fs_etpu_pwm_init |
| Configure Pad | Configure Pad for eTPUA[5] output<br>Pad Assignment = eAPUA[5]<br>Output Buffer is enabled<br>Open Drain is not enabled | SIU_PCR[119] = 0x0E00
PA = 3
OBE = 1
ODE = 0 |
| Start timers | Start all eTPU timers and eMIOS timers | –<br>fs_timer_start |
| Update eTPU[5] | • Channel = eTPUA[5]<br>• Frequency = 2000 Hz<br>• Duty cycle = 60%
• Timebase = 1 MHz | –<br>fs_etpu_pwm_update |
20.2.2 Files Used For Example

Below is a summary of the files used for this project. All are available from the Freescale website, except for main.c and main.h, which are listed in Section 20.3, “Code.”

Note that the eTPU C compiler is not needed to make this example because the output files are available on the Freescale website at www.freescale.com/etpu. However, all the source files used to build the set 1 functions are available and may be useful for reference.

<table>
<thead>
<tr>
<th>Provider</th>
<th>Type</th>
<th>Filename</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
<td>“Application code”</td>
<td>main.c</td>
<td>Files written for this example</td>
</tr>
<tr>
<td></td>
<td></td>
<td>main.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Link and Make Files</td>
<td>link file</td>
<td>Link file and make file used for this example</td>
</tr>
<tr>
<td></td>
<td></td>
<td>make file</td>
<td></td>
</tr>
<tr>
<td>Freescale</td>
<td>eTPU Set 1 Library</td>
<td>etpu_pwm.c</td>
<td>Host application program interface for pwm function</td>
</tr>
<tr>
<td></td>
<td></td>
<td>etpu_pwm.h</td>
<td>Header file for pwm function</td>
</tr>
<tr>
<td></td>
<td></td>
<td>etpu_pwm_auto.h</td>
<td>Parameters automatically generated by eTPU compiler for pwm function</td>
</tr>
<tr>
<td></td>
<td></td>
<td>etpu_set1.h</td>
<td>Code image and globals generated by eTPU compiler for all of set 1 functions</td>
</tr>
<tr>
<td>eTPU Utilities</td>
<td></td>
<td>etpu_util.c</td>
<td>Host utilities to initialize eTPU, copy code image into code RAM, etc.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>etpu_util.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>etpu_struct.h</td>
<td></td>
</tr>
<tr>
<td>MPC5500 Headers</td>
<td>mpc5554.h</td>
<td></td>
<td>Headers for MPC5500 device</td>
</tr>
<tr>
<td></td>
<td>mpc5554_vars.h</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>typedefs.h</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 77. Files Used in Example
20.2.3 Design Screenshot

The screenshot below shows the output at the initial PWM setting of 1 kHz period and 25% duty cycle.
20.3 Code

20.3.1 main.c

#include "mpc563m.h" /* Use proper include file such as mpc5510.h or mpc5554.h */
#include "mpc563m_vars.h"       /* MPC563m specific variables */
#include "etpu_util.h"          /* useful utility routines */
#include "etpu_set1.h"          /* eTPU standard function set 1 */
#include "etpu_pwm.h"           /* eTPU PWM API */
#include "main.h"        /* include application specific defines. */

void initSysclk (void) {
    /* MPC563x: Use the next line */
    FMPLL.ESYNCR1.B.CLKCFG = 0X7;       /* Change clk to PLL normal mode from crystal */
    /* MPC555x including MPC563x: use the next 3 lines for either 8 or 40 MHz crystal */
    FMPLL.SYNCR.R = 0x16080000;         /* 8 MHz xtal: 0x16080000; 40MHz: 0x46100000 */
    while (FMPLL.SYNSR.B.LOCK != 1) {}; /* Wait for FMPLL to LOCK */
    FMPLL.SYNSR.R = 0x16000000;         /* 8 MHz xtal: 0x16000000; 40MHz: 0x46080000 */
}

int main ()
{
    int32_t error_code;               /* Returned value from eTPU API functions */
    initSysclk();                     /* Initialize PLL to 64 MHz */
    fs_etpu_init (             my_etpu_config,
        (uint32_t *) etpu_code,
        sizeof (etpu_code),
        (uint32_t *) etpu_globals,
        sizeof (etpu_globals));
    error_code = fs_etpu_pwm_init (5,  /* Channel ETPU_A[5] */
        FS_ETPU_PRIORITY_MIDDLE,
        1000,  /* Frequency = 1000 Hz*/
        2500,  /* Duty cycle = 2500/100 = 25% */
        FS_ETPU_PWM_ACTIVEHIGH,
        FS_ETPU_TCR1,
        1000000); /* Timebase (TCR1) freq is 1 MHz */
    SIU.PCR[119].R = 0x0E00;          /* Configure pad for signal ETPU_A[5] output */
    fs_timer_start ();                /* Enable all timebases */
    error_code = fs_etpu_pwm_update (5,  /* Channel ETPU_A[5] */
        2000,  /* New frequency = 2kHZ*/
        6000,  /* New duty cycle = 6000/100= 60% */
        1000000); /* Timebase (TCR1) freq = 1 MHz */
    while(1);
    /* Wait forever */
}
20.3.2 main.h

/* main.h based on gpio example.h below */
*******************************************************************************/
* FILE NAME: $RCSfile: gpio_example.h,v $ COPYRIGHT (c) FREESCALE 2004 *
* DESCRIPTION: All Rights Reserved *
* This file contains prototypes and definitions for the sample MPC5500 *
* program using the etPU GPIO function. *
*========================================================================*
* ORIGINAL AUTHOR: Jeff Loeliger (r12110) *
* $Log: gpio_example.h,v *
* Revision 1.1  2004/12/08 11:45:09  r47354 *
* Updates as per QOM API rel_2_1 *
*THEN *
*------------------------------------------------------------------------*
* 0.1 J. Loeliger 05/Sep/03 Initial version. *
* 0.2 K Terry 29/Apr/04 mod'd for GPIO function test *
* 0.3 Updated for new build structure. *
* 0.4 G. Emerson 2/Nov/04 Added etpu_config_t definition *
*------------------------------------------------------------------------*/
/* Rev 15/Mar/06 S. Mihalik: modified for etPU PWM example */
/* Rev 10/Aug/07 S. Mihalik: modified for 64 MHz sysclk, still 1 MHz TCR1 */
#include "etpu_util.h"

struct etpu_config_t my_etpu_config = {
  FS_ETPU_M Misc_DISABLE,  /*MCR register*/
  FS_ETPU_MISC,            /*MISC value from etPU compiler link file*/

  FS_ETPU_FILTER_CLOCK_DIV8 + FS_ETPU_CHAN_FILTER_3SAMPLE +
  FS_ETPU_ENTRY_TABLE,    /*Configure etPU engine A*/

  FS_ETPU_TCR2CTL_DIV8 +
  ( 7 << 16),             /*TCR2 prescaler of 8 (7+1)*/
  FS_ETPU_TCR1CTL_DIV2 +
  31,                     /*TCR1 prescaler of 32 (31+1) applied to sysclk/2*/
  0,                      /*Configure etPU engine B*/

  FS_ETPU_FILTER_CLOCK_DIV4 +
  FS_ETPU_CHAN_FILTER_3SAMPLE +
  FS_ETPU_ENTRY_TABLE,    /*Configure etPU engine B timebases*/

  FS_ETPU_TCR2CTL_DIV8 +
  ( 7 << 16),             /*TCR2 prescaler of 8 (7+1)*/
  FS_ETPU_TCR1CTL_DIV2 +
  3,                      /*TCR1 prescaler of 4 (3+1)*/
};
21 eQADC: Single Software Scan

21.1 Description

**Task:** Convert the analog signal on analog channel 5. Use CFIFO 0 in single-scan software-triggered mode, and use converter ADC0. Send results to RFIFO 0. The system clock will be the default of 16 MHz (MPC551x) or 12 MHz (MPC555x). The ADC_CLK, which must not exceed the 6 MHz for maximum resolution, will assume here the default system clock, which is not faster than 16 MHz.

This minimal example shows how to send configuration commands for ADC0, send conversion command, and read the result. It does not incorporate queues, DMA, interrupts, calibration, or time stamp, nor does it clear the last Command FIFO EOQ flag and Result FIFO drain flag.

Full accuracy may not be possible because calibration is not implemented here. For information, see AN2989, *Design, Accuracy, and Calibration of Analog to Digital Converters on the MPC5500 Family*.

**Exercise:** Jumper a voltage, such as from a variable resistor, to AN5 and read the converted result.

![Figure 34. eQADC Single Software Scan Example](image)

### Table 78. Signals for eQADC Single Software Scan Example

<table>
<thead>
<tr>
<th>Signal</th>
<th>Pin Name</th>
<th>SIU PCR No.</th>
<th>Package Pin No.</th>
<th>Function Name</th>
<th>SIU PCR No.</th>
<th>Package Pin No.</th>
<th>EVB</th>
</tr>
</thead>
<tbody>
<tr>
<td>AN5</td>
<td>PA[5]</td>
<td>5</td>
<td>4</td>
<td>4 D1</td>
<td>–</td>
<td>B9</td>
<td>A8</td>
</tr>
</tbody>
</table>

Qorivva Simple Cookbook, Rev. 4
21.2 Design

For MPC551x devices, the default system clock is 16 MHz. To achieve an ADC clock not exceeding 6 MHz, the prescaler will be 16 MHz systclk / 6 MHz = 8/3 ≈ 2.67. Because we need an even integer value that keeps the ADCCCLK under 6 MHz, we round up to a value of prescaler of 4. This provides an ADCCCLK = 16 MHz / 4 = 4 MHz.

(Note for the MPC5516 evaluation board: the variable resistor is hard-wired to AN0 that is on PA[0], so you can simply jumper this to AN5, which is on PA[5], and run this program.)

On MPC555x devices using the default system clock of 12 MHz, an ADC prescaler of 4 provides an ADCCCLK = 12 MHz / 4 = 3 MHz. On MPC563x devices using the default system clock of 8 MHz, an ADC prescaler of 4 provides an ADCCCLK = 8 MHz / 4 = 2 MHz.

<table>
<thead>
<tr>
<th>Table 79. eQADC Single Software Scan</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Step</strong></td>
</tr>
<tr>
<td>Init ADC0</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Send one write configuration command to CFIFO0:</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Trigger CFIFO0 using single scan SW mode</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Wait for End of Queue Flag for CFIFO0</td>
</tr>
<tr>
<td>Clear End of Queue Flag for CFIFO0</td>
</tr>
<tr>
<td>Send Conversion Command</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Trigger CFIFO0 using single scan SW mode</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Read Result</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
21.3 Code

/* main.c: performs a simple ADC conversion of channel 5 using ADC0 */
/* Rev 1.0 Sept 13, 2004 S. Mihalik. */
/* Rev 1.1 Jul 18 2007 SM - Changed ADCCLK prescaler for faster MPC551x default */
/* sysclk, added result in millivolts, used channel 5, conversion in loop */
/* Rev 1.2 Jun 12 2008 SM - Moved initADC0 out of while loop */
/* Copyright Freescale, 2007. All Rights Reserved */
/* Notes: */
/* 1. MMU not initialized; must be done by debug scripts or BAM */
/* 2. SRAM not initialized; must be done by debug scripts or in a crt0 type file */

#include "mpc563m.h" /* Use proper include file such as mpc5510.h or mpc5554.h */

static uint32_t Result = 0;         /* ADC conversion result */
static uint32_t ResultInMv = 0;     /* ADC conversion result in millivolts */

void initADC0(void) {
    EQADC.CFPR[0].R = 0x80801001;       /* Send CFIFO 0 a ADC0 configuration command */
    /* enable ADC0 & sets prescaler = divide by 2*/
    EQADC.CFCR[0].R = 0x0410;           /* Trigger CFIFO 0 using Single Scan SW mode */
    while (EQADC.FISR[0].B.EOQF != 1) {} /* Wait for End Of Queue flag */
    EQADC.FISR[0].B.EOQF = 1;           /* Clear End Of Queue flag */
}

void SendConvCmd (void) {
    EQADC.CFPR[0].R = 0x80000500; /* Conversion command: convert channel 5 */
    /* with ADC0, set EOQ, and send result to RFIFO 0*/
    EQADC.CFCR[0].R = 0x0410;       /* Trigger CFIFO 0 using Single Scan SW mode */
}

void ReadResult(void) {
    while (EQADC.FISR[0].B.RFDF != 1){}      /* Wait for RFIFO 0's Drain Flag to set*/
    Result = EQADC.RFPR[0].R;                /* ADC result */
    ResultInMv = (uint32_t)((5000*Result)/0x3FFC);  /* ADC result in millivolts */
    EQADC.FISR[0].B.EOQF = 1;                /* Clear CFIFO's End of Queue flag */
}

int main(void) {
    int i = 0;                    /* Dummy idle counter */
    initADC0();                   /* Enable ADC0 only on eQADC */
    while (1) {
        SendConvCmd();              /* Send one conversion command */
        ReadResult();               /* Read result */
        i++;                        /* Wait forever */
    }
}
22 ADC: Software Trigger, Continuous Scan

22.1 Description

Task: Convert a few standard ADC channel inputs by starting a normal conversion which is software triggered. Instead of One Shot Mode, Scan Mode is used where “a sequential conversions of N channels specified in the NCMR registers is continuously performed.”

Exercise: Connect an analog channel to the pot on the EVB. Jumper ATD’s VDD to 5 V, download program, and verify results. Add additional channels and connect to a known voltage. Use ANS7 on PC[7] pin for the Dashboard Cluster Demo’s AUX MOTOR pot. Add an injected channel using an external trigger of PIT or eMIOS.

![Figure 35. ADC Continuous SW Scan Example Simplified Block Diagram](image)

Table 80. MPC56xxB/P/S Signals for Continuous SW Scan Example

<table>
<thead>
<tr>
<th>Signal</th>
<th>MPC56xxB Family</th>
<th>MPC56xxP Family</th>
<th>MPC56xxS Family</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Port</td>
<td>SIU Pad Configuration &amp; Selection Registers (values in hex)</td>
<td>Package Pin #</td>
</tr>
<tr>
<td></td>
<td>B8</td>
<td>PCR24=2000</td>
<td>39 53  R9</td>
</tr>
<tr>
<td>ANSI0</td>
<td>PCS</td>
<td>100 QFP 144 QFP 176 BGA</td>
<td>40 54  P9</td>
</tr>
<tr>
<td></td>
<td>B9</td>
<td>PCR25=2000</td>
<td>38 52  T9</td>
</tr>
<tr>
<td>ANS1</td>
<td>ANS0</td>
<td>39 53  R9</td>
<td></td>
</tr>
<tr>
<td>ANS2</td>
<td>B10</td>
<td>PCR26=2000</td>
<td>40 54  P9</td>
</tr>
</tbody>
</table>

### 22.2 Design

#### 22.2.1 Mapping of ADC Channels to Analog Input Pins

Table 81. MPC56xxB/P/S Mapping of ADC Channels to ADC Input Pins

<table>
<thead>
<tr>
<th>ADC Channel Number</th>
<th>Pin Function Name</th>
<th>ADC 0 Pin Function Name</th>
<th>ADC 1 Pin Function Name</th>
<th>Pin Function Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ANP[0] ANP[1]</td>
<td>AN[0] AN[1]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>ANP[18] ANP[19]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>ANP[22] ANP[23]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>ANP[26] ANP[27]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>ANP[28] ANP[29]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>ANP[30] ANP[31]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>ANP[32] ANP[33]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>ANP[34] ANP[35]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>ANP[36] ANP[37]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>ANP[38] ANP[39]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>ANP[40] ANP[41]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>ANP[42] ANP[43]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>ANP[44] ANP[45]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>ANP[46] ANP[47]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>ANP[48] ANP[49]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>ANP[50] ANP[51]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>ANP[52] ANP[53]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>ANP[54] ANP[55]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>ANP[56] ANP[57]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>ANP[58] ANP[59]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>ANP[60] ANP[61]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>ANP[62] ANP[63]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>ANP[64] ANP[65]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>ANP[66] ANP[67]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>ANP[68] ANP[69]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>ANP[70] ANP[71]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>ANP[72] ANP[73]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>ANP[74] ANP[75]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>ANP[76] ANP[77]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>ANP[78] ANP[79]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>ANP[80] ANP[81]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>ANP[82] ANP[83]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>ANP[84] ANP[85]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>ANP[86] ANP[87]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>ANP[88] ANP[89]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>ANP[90] ANP[91]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>46</td>
<td>ANP[92] ANP[93]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>ANP[94] ANP[95]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>48,59</td>
<td>ANP[96] ANP[97]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The following channels use multiplex selector signals, MA[0:2]:

<table>
<thead>
<tr>
<th>ADC Channel Number</th>
<th>Pin Function Name</th>
<th>ADC 0 Pin Function Name</th>
<th>ADC 1 Pin Function Name</th>
<th>Pin Function Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>ANX[0], MA=0</td>
<td></td>
<td></td>
<td>ANS[10], MA=0</td>
</tr>
<tr>
<td>65</td>
<td>ANX[0], MA=1</td>
<td></td>
<td></td>
<td>ANS[10], MA=1</td>
</tr>
</tbody>
</table>

Qorivva Simple Cookbook, Rev. 4

Freescale Semiconductor
22.2.2 Mode Use

Mode Transition is required for changing mode entry registers. Hence even enabling the crystal oscillator to be active in the current mode (for example default mode (DRUN)) requires enabling the crystal oscillator in DRUN mode configuration register (ME_DRUN_MC), then initiating a mode transition to the same DRUN mode. This example changes from DRUN mode to RUN0 mode.

This minimal example simply polls a status bit to wait for the targeted mode transition to complete. However, the status bit could instead be enabled to generate an interrupt request (assuming the INTC is initialized beforehand). This would allow software to complete other initialization tasks instead of brute force polling of the status bit.

It is normal to use a timer when waiting for a status bit to change. This example by default would have a watchdog timer expire if for some reason the mode transition never completes. One could also loop code on incrementing a software counter to some maximum value as a timeout. If a timeout was reached, then an error condition could be recorded in EEPROM or elsewhere.

### Table 81. MPC56xxB/P/S Mapping of ADC Channels to ADC Input Pins

<table>
<thead>
<tr>
<th></th>
<th>ANX[0], MA=2</th>
<th>ANS[10], MA=2</th>
</tr>
</thead>
<tbody>
<tr>
<td>66</td>
<td>ANX[0], MA=3</td>
<td>ANS[10], MA=3</td>
</tr>
<tr>
<td>67</td>
<td>ANX[0], MA=4</td>
<td>ANS[10], MA=4</td>
</tr>
<tr>
<td>68</td>
<td>ANX[0], MA=5</td>
<td>ANS[10], MA=5</td>
</tr>
<tr>
<td>69</td>
<td>ANX[0], MA=6</td>
<td>ANS[10], MA=6</td>
</tr>
<tr>
<td>70</td>
<td>ANX[0], MA=7</td>
<td>ANS[10], MA=7</td>
</tr>
<tr>
<td>71</td>
<td>ANX[1], MA=0</td>
<td></td>
</tr>
<tr>
<td>72</td>
<td>ANX[1], MA=1</td>
<td></td>
</tr>
<tr>
<td>73</td>
<td>ANX[1], MA=2</td>
<td></td>
</tr>
<tr>
<td>74</td>
<td>ANX[1], MA=3</td>
<td></td>
</tr>
<tr>
<td>75</td>
<td>ANX[1], MA=4</td>
<td></td>
</tr>
<tr>
<td>76</td>
<td>ANX[1], MA=5</td>
<td></td>
</tr>
<tr>
<td>77</td>
<td>ANX[1], MA=6</td>
<td></td>
</tr>
<tr>
<td>78</td>
<td>ANX[1], MA=7</td>
<td></td>
</tr>
<tr>
<td>79</td>
<td>ANX[2], MA=0</td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>ANX[2], MA=1</td>
<td></td>
</tr>
<tr>
<td>81</td>
<td>ANX[2], MA=2</td>
<td></td>
</tr>
<tr>
<td>82</td>
<td>ANX[2], MA=3</td>
<td></td>
</tr>
<tr>
<td>83</td>
<td>ANX[2], MA=4</td>
<td></td>
</tr>
<tr>
<td>84</td>
<td>ANX[2], MA=5</td>
<td></td>
</tr>
<tr>
<td>85</td>
<td>ANX[2], MA=6</td>
<td></td>
</tr>
<tr>
<td>86</td>
<td>ANX[2], MA=7</td>
<td></td>
</tr>
<tr>
<td>87</td>
<td>ANX[3], MA=0</td>
<td></td>
</tr>
<tr>
<td>88</td>
<td>ANX[3], MA=1</td>
<td></td>
</tr>
<tr>
<td>89</td>
<td>ANX[3], MA=2</td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>ANX[3], MA=3</td>
<td></td>
</tr>
<tr>
<td>91</td>
<td>ANX[3], MA=4</td>
<td></td>
</tr>
<tr>
<td>92</td>
<td>ANX[3], MA=5</td>
<td></td>
</tr>
<tr>
<td>93</td>
<td>ANX[3], MA=6</td>
<td></td>
</tr>
<tr>
<td>94</td>
<td>ANX[3], MA=7</td>
<td></td>
</tr>
</tbody>
</table>
Peripherals also have configurations to gate clocks on and off, enabling low power. The following table summarizes the peripheral configurations used here. ME_RUNPC_1 is selected, so peripherals to be used require a non-zero value in their respective ME_PCTL register.

Table 82. Mode Configurations Summary for MPC56xxB/P/S ADC Continuous SW Scan Example
(modes are enabled in ME_ME Register)

<table>
<thead>
<tr>
<th>Mode</th>
<th>Mode Config. Register Value</th>
<th>sysclk</th>
<th>Clock Sources</th>
<th>PLL1 (MPC 56xxB/S only)</th>
<th>Memory Power Mode</th>
<th>Main Voltage Reg.</th>
<th>I/O Power Down Ctrl</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRUN</td>
<td>ME_DRUN_MC 0x001F 0010 (default)</td>
<td>16 MHz IRC</td>
<td>On Off Off Off</td>
<td>Normal Normal</td>
<td>On Off</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RUN0</td>
<td>ME_RUN0_MC 0x001F 0074</td>
<td>PLL0</td>
<td>On On On Off</td>
<td>Nomral Normal</td>
<td>On Off</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Other modes are not used in example.

Table 83. Peripheral Configurations for MPC56xxB/P/S ADC Continuous SW Scan Example
(low power modes are not used in example)

<table>
<thead>
<tr>
<th>Peripheral Config.</th>
<th>Peri. Config. Register</th>
<th>Enabled Modes</th>
<th>Peripherals Selecting Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>RUN3</td>
<td>RUN2</td>
<td>RUN1</td>
<td>RUN0</td>
</tr>
<tr>
<td>PC1</td>
<td>ME_RUNPC_1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Other peripheral configurations are not used in example.
## 22.2.3 Steps and Pseudo Code

### Table 84. MPC5606B, MPC56xxP, MPC56xxS Steps for ADC Continuous SW Scan Example

<table>
<thead>
<tr>
<th>Step</th>
<th>Relevant Bit Fields</th>
<th>Pseudo Code</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Init Modes and Clock</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enable desired modes</td>
<td>RUN0, DRUN=1</td>
<td>ME_ME = 0x0000 001D</td>
</tr>
<tr>
<td>Initialize PLL0 dividers to provide 64 MHz for input crystal before mode configuration:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• 8 MHz xtal: FMPLL[0]_CR=0x02400100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• 40 MHz xtal: FMPLL[0]_CR=0x12400100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(MPC56xxP &amp; 8 MHz xtal requires changing default CMU_CSR value. See PLL example.)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Relevant Bit Fields</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CGM_FMPLL_CR (MPC56xxB)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CGM_FMPLL[0]_CR (MPC56xxS)</td>
<td>= 0x02400100 (8 MHz crystal)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>or 0x12400100 (40 MHz crystal)</td>
<td></td>
</tr>
<tr>
<td><strong>Configuring RUN0 Mode:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• I/O Output power-down: no safe gating</td>
<td>PDO=0</td>
<td></td>
</tr>
<tr>
<td>• Main Voltage regulator is on (default)</td>
<td>MVRON=1</td>
<td>ME_RUN0_MC = 0x001F 0074</td>
</tr>
<tr>
<td>• Data, code flash in normal mode (default)</td>
<td>DFLAON = 3,</td>
<td></td>
</tr>
<tr>
<td>• PLL0 is switched on</td>
<td>CFLAON = 3</td>
<td></td>
</tr>
<tr>
<td>• Crystal oscillator is switched on</td>
<td>PLL0ON=1</td>
<td></td>
</tr>
<tr>
<td>• 16 MHz IRC is switched on (default)</td>
<td>XOSCO1N=1</td>
<td></td>
</tr>
<tr>
<td>• Select PLL0 (system pll) as sysclk</td>
<td>16MHz_IRCON=1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SYSCLK=0x4</td>
<td></td>
</tr>
<tr>
<td>Relevant Bit Fields</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CGM_FMPLL[0]_CR (MPC56xxS)</td>
<td>= 0x02400100 (8 MHz crystal)</td>
<td></td>
</tr>
<tr>
<td>CGM_FMPLL[0]_CR (MPC56xxS)</td>
<td>= 0x12400100 (40 MHz crystal)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>or 0x12400100 (40 MHz crystal)</td>
<td></td>
</tr>
<tr>
<td>Assign peripheral configuration to peripherals:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• ADC 0: select ME_RUN_PC0</td>
<td>RUN_CFG = 1</td>
<td></td>
</tr>
<tr>
<td>• SIUL: select ME_RUN_PC0 (56xxB/S)</td>
<td>RUN_CFG = 1</td>
<td></td>
</tr>
<tr>
<td>Initiate software mode transition to RUN0 mode</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Mode and key</td>
<td>TARGET_MODE= RUN0</td>
<td></td>
</tr>
<tr>
<td>• Mode and inverted key</td>
<td>S_MTRANS</td>
<td></td>
</tr>
<tr>
<td>• Wait for mode transition to complete</td>
<td>* MODE and * KEY and key</td>
<td></td>
</tr>
<tr>
<td>NOTE: if transition does not complete, check</td>
<td></td>
<td></td>
</tr>
<tr>
<td>status flags such as ME_GS[XOSC]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Verify desired target mode was entered</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Disable Watchdog</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Write keys to clear soft lock bit</td>
<td>SWT_SR = 0x0000 C520</td>
<td></td>
</tr>
<tr>
<td>• Clear watchdog enable bit</td>
<td>SWT_SR = 0x0000 D928</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SWT_CR = 0x8000 010A</td>
<td></td>
</tr>
<tr>
<td><strong>Init Peri Clk Gen</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Initialize peripheral clockgeneration (See appendix: MPC56xxB/P/S Peripheral Clocks)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• ADC (56xxB/S): peripheral set 3- sysclk/1</td>
<td>DE2=1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DIV2=0</td>
<td></td>
</tr>
<tr>
<td><strong>Init Pads</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Init pads for converting ANS0, ANS1, ANS2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Qorivva Simple Cookbook, Rev. 4

Freescale Semiconductor 175
Table 84. MPC5606B, MPC56xxP, MPC56xxS Steps for ADC Continuous SW Scan Example (continued)

<table>
<thead>
<tr>
<th>Step</th>
<th>Relevant Bit Fields</th>
<th>Pseudo Code</th>
<th>MPC56xxB</th>
<th>MPC56xxP</th>
<th>MPC56xxS</th>
</tr>
</thead>
</table>
| init ADC0 | Initialize ADC 0 module & start conversions:  
• Configure Analog Clock as sysclk/2 (PLL/2 = 32 MHz for this example)  
• Mode is Scan Mode (continuous)  
• Trigger: on chip not used  
• ADC in normal mode, not power down mode | ADCLKSEL = 0  
MODE = 1  
TRIGEN = 0  
PWDN = 0 | ADC_MCR = 0x2000 0000 (MPC56xxB)  
ADC0_MCR = 0x2000 0000 (MPC56xxS) |
| Enable normal sampling of desired channels for ADC 0 (Input pins ANS0, ANS1, ANS2) | MPC56xxP:  
CHAN0:2 = 1  
MPC56xxS:  
CHAN32:34 = 1 | ADC_5CMR1 = 0x0000 0007  
ADC0_5CMR0 = 0x0000 0007  
ADC0_5CMR1 = 0x0000 0007 | |
| Initialize conversion timings for 32 MHz ADCLK$^1$ | INPLATCH = 1  
INPCMP = 3  
INPSAMP = 6 | ADC_CTR1 = 0x0000 8606  
ADC0_CTR0 = 0x00008606  
ADC0_CTR1 = 0x00008606 | |
| Trigger ADC | Start Normal Conversions:  
• Start normal conversion (immediately) | NSTART = 1 | ADC_MCR [NSTART] = 1 (MPC56xxB)  
ADC0_MCR [NSTART] = 1 (MPC56xxS) |
| Loop: | Wait for completion of first chain (Note: VALID flag clears when read) | wait for EOC = 1,  
wait for first ADC_ISR[EOC] = 1  
wait for first ADC0_ISR[EOC] = 1  
wait for first ADC00_ISR[EOC] = 1 | |
| Read results | CDATA | Result0 = ADC_CDR32[CDATA]  
Result1 = ADC_CDR33[CDATA]  
Result2 = ADC_CDR34[CDATA]  
Result0 = ADC0_CDR0[CDATA]  
Result1 = ADC0_CDR1[CDATA]  
Result2 = ADC0_CDR2[CDATA]  
Result0 = ADC0_CDR32[CDATA]  
Result1 = ADC0_CDR33[CDATA]  
Result2 = ADC0_CDR34[CDATA] | Result0 = int16_t (5000*Result0 / 0x3FF)  
Result1 = int16_t (5000*Result1 / 0x3FF)  
Result2 = int16_t (5000*Result2 / 0x3FF) |
| Convert results to mv | | ResultInMv0 = int16_t (5000*Result0 / 0x3FF)  
ResultInMv1 = int16_t (5000*Result1 / 0x3FF)  
ResultInMv2 = int16_t (5000*Result2 / 0x3FF) |

#include "56xxS_0200.h" /* Use proper header file */

void initModesAndClock(void) {
    ME.MER.R = 0x0000001D; /* Enable DRUN, RUN0, SAFE, RESET modes */
    CGM.FMPLL[0].CR.R = 0x02400100; /* 8 MHz xtal: Set PLL0 to 64 MHz */
    ME.RUN[0].R = 0x001F0074; /* RUN0 cfg: 16MHzIRCON,OSC0ON,PLL0ON,sysclk=PLL0*/
    ME.RUNPC[1].R = 0x00000010; /* Peri. Cfg. 1 settings: only run in RUN0 mode */
    ME.PCTL[32].R = 0x01; /* MPC56xxB/P/S ADC 0: select ME.RUNPC[1] */
    ME.PCTL[68].R = 0x01; /* MPC56xxB/S SIU: select ME.RUNPC[1] */
    /* Mode Transition to enter RUN0 mode: */
    ME.MCTL.R = 0x40005AF0; /* Enter RUN0 Mode & Key */
    ME.MCTL.R = 0x4000A50F; /* Enter RUN0 Mode & Inverted Key */
    while (ME.GS.B.S_MTRANS) {} /* Wait for mode transition to complete */
    /* Note: could wait here using timer and/or I_TC IRQ */
    while(ME.GS.B.S_CURRENTMODE != 4) {} /* Verify RUN0 is the current mode */
}

void disableWatchdog(void) {
    SWT.SR.R = 0x0000c520; /* Write keys to clear soft lock bit */
    SWT.SR.R = 0x0000d928;
    SWT.CR.R = 0x8000010A; /* Clear watchdog enable (WEN) */
}

void initPeriClkGen(void) {
    CGM.SC_DC[2].R = 0x80; /* MPC56xxB/S: Enable peri set 3 sysclk divided by 1 */
}

void main (void) {
    uint32_t i = 0; /* Dummy idle counter */
    initModesAndClock(); /* Initialize mode entries and system clock */
    disableWatchdog(); /* Disable watchdog */
    initPeriClkGen(); /* Initialize peripheral clock generation for DSPIs */
    SIU.PCR[30].R = 0x2000; /* MPC56xxS: Initialize PC[0] as ANS0 */
    SIU.PCR[31].R = 0x2000; /* MPC56xxS: Initialize PC[1] as ANS1 */
    SIU.PCR[32].R = 0x2000; /* MPC56xxS: Initialize PC[2] as ANS2 */
    ADC_0.MCR.R = 0x20000000; /* Initialize ADC0 for scan mode */
    ADC_0.NCMR[1].R = 0x24000000; /* Select ANS0:2 inputs for conversion */
    ADC_0.CTR[1].R = 0x00002606; /* Conversion times for 32MHz ADClock */
    ADC_0.MCR.B.NSTART=1; /* Trigger normal conversions for ADC0 */

    while (1) {
        while (ADC_0.CDR[33].B.VALID != 1) {}; /* Wait for last scan to complete */
        Result[0]= ADC_0.CDR[32].B.CDATA; /* Read ANS0 conversion result data */
        Result[1]= ADC_0.CDR[33].B.CDATA; /* Read ANS1 conversion result data */
        Result[2]= ADC_0.CDR[34].B.CDATA; /* Read ANS2 conversion result data */
        ResultInMv[0] = (uint16_t) (5000*Result[0]/0x3FF); /* Converted result in mv */
        ResultInMv[1] = (uint16_t) (5000*Result[1]/0x3FF); /* Converted result in mv */
        ResultInMv[2] = (uint16_t) (5000*Result[2]/0x3FF); /* Converted result in mv */
        i++;
    }
}
23  ADC - CTU: eMIOS Trigger (MPC560xB)

23.1  Description

**Task:** While performing normal conversions on a few standard ADC channel inputs, use the Cross Triggering Unit (CTU) to trigger a conversion from an eMIOS channel event, where the channel is configured as Single Action Input Capture (SAIC).

In this example, ADC inputs ANS1 and ANS2 are continuously scanned. EMIOS and the CTU are configured so an input signal on eMIOS channel 2 will cause an event that “cross triggers” a conversion for ANS0. To generate an input signal, eMIOS channel 3 is configured for OPWM. Both channels use eMIOS channel 23 configured as modulus counter for their time base.

**Exercise:** Make the connections shown below. Connect ANS0 to a potentiometer or a known voltage. Connect ANS1:2 to other known voltages. Verify the eMIOS channel 2 injects an ADC command which reads the pot value.

![Figure 36. ADC CTU Example Simplified Block Diagram](image-url)
Table 85. MPC56xxB Signals for ADC CTU Example

<table>
<thead>
<tr>
<th>Signal</th>
<th>MPC56xxB Family</th>
<th>Port</th>
<th>SIU Pad Configuration &amp; Selection Registers (values in hex)</th>
<th>Package Pin #</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>100 QFP</td>
<td>144 QFP</td>
</tr>
<tr>
<td>ANS0</td>
<td>B8</td>
<td>PCR24=2000</td>
<td>39</td>
<td>53</td>
</tr>
<tr>
<td>ANS1</td>
<td>B9</td>
<td>PCR25=2000</td>
<td>38</td>
<td>52</td>
</tr>
<tr>
<td>ANS2</td>
<td>B10</td>
<td>PCR26=2000</td>
<td>40</td>
<td>54</td>
</tr>
<tr>
<td>eMIOS0[2]</td>
<td>A2</td>
<td>PCR2=0503</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>eMIOS0[3]</td>
<td>A3</td>
<td>PCR3=0600</td>
<td>68</td>
<td>90</td>
</tr>
</tbody>
</table>

23.2 Design

23.2.1 Channel Mappings

The following table provides the numbering used for analog input signal names, ADC channel numbers, and CTU channel numbers.

Table 86. MPC56xxB Mapping of Analog Signals, ADC channel numbers and CTU trigger channel numbers (per MPC5607B Microcontroller Reference Manual, Rev. 2)

<table>
<thead>
<tr>
<th>Signal Name (per Table 2-3)</th>
<th>ADC Channel # (per Fig. 23-1)</th>
<th>CTU_EVTCFTRx[CHANNEL_VALUE] (per Table 30-6)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ADC 0</td>
<td>ADC 1</td>
</tr>
<tr>
<td>ANP 0:15</td>
<td>0:15</td>
<td>0:15</td>
</tr>
<tr>
<td>ANS 0:15</td>
<td>32:47</td>
<td>—</td>
</tr>
<tr>
<td>ANS 16:27</td>
<td>48:59</td>
<td>—</td>
</tr>
<tr>
<td>ANX 0, MA 0:7</td>
<td>64:71</td>
<td>—</td>
</tr>
<tr>
<td>ANX 0, MA 0:7</td>
<td>72:79</td>
<td>—</td>
</tr>
<tr>
<td>ANX 0, MA 0:7</td>
<td>80:87</td>
<td>—</td>
</tr>
<tr>
<td>ANX 0, MA 0:7</td>
<td>88:95</td>
<td>—</td>
</tr>
</tbody>
</table>
23.2.2 Mode Use

Mode Transition is required for changing mode entry registers. Hence even enabling the crystal oscillator to be active in the current mode (for example default mode (DRUN)) requires enabling the crystal oscillator in DRUN mode configuration register (ME_DRUN_MC), then initiating a mode transition to the same DRUN mode. This example changes from DRUN mode to RUN0 mode.

This minimal example simply polls a status bit to wait for the targeted mode transition to complete. However, the status bit could instead be enabled to generate an interrupt request (assuming the INTC is initialized beforehand). This would allow software to complete other initialization tasks instead of brute force polling of the status bit.

It is normal to use a timer when waiting for a status bit to change. This example by default would have a watchdog timer expire if for some reason the mode transition never completes. One could also loop code on incrementing a software counter to some maximum value as a timeout. If a timeout was reached, then an error condition could be recorded in EEPROM or elsewhere.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Mode Config. Register</th>
<th>Mode Config. Register Value</th>
<th>Settings</th>
<th>Clock Sources</th>
<th>Memory Power Mode</th>
<th>Main Voltage Reg.</th>
<th>I/O Power Down Ctrl</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRUN</td>
<td>ME_DRUN_MC</td>
<td>0x001F 0010 (default)</td>
<td>16 MHz IRC</td>
<td>On</td>
<td>Off</td>
<td>Off</td>
<td>Off</td>
</tr>
<tr>
<td>RUN0</td>
<td>ME_RUN0_MC</td>
<td>0x001F 0074 PLL0</td>
<td>On</td>
<td>On</td>
<td>On</td>
<td>Off</td>
<td>Normal</td>
</tr>
</tbody>
</table>

Other modes are not used in example.

Peripherals also have configurations to gate clocks on and off, enabling low power. The following table summarizes the peripheral configurations used here. ME_RUNPC_1 is selected, so peripherals to be used require a non-zero value in their respective ME_PCTL register.

<table>
<thead>
<tr>
<th>Peripheral Config.</th>
<th>Peri. Config. Register</th>
<th>Enabled Modes</th>
<th>Peripherals Selecting Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>RUN3</td>
<td>RUN2</td>
<td>RUN1</td>
<td>RUN0</td>
</tr>
<tr>
<td>PC1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Other peripheral configurations are not used in example.
### 23.2.3 Steps and Pseudo Code

**Table 89. MPC5606B Steps for MPC56xxB ADC CTU Example**

<table>
<thead>
<tr>
<th>Step</th>
<th>Relevant Bit Fields</th>
<th>Pseudo Code</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Init Modes and Clock</strong></td>
<td>Enable desired modes</td>
<td>RUN0, DRUN=1</td>
</tr>
<tr>
<td></td>
<td>Initialize PLL0 dividers to provide 64 MHz for input crystal <strong>before</strong> mode configuration, using progressive clock switching:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• 8 MHz Crystal: FMPLL[0]_CR=0x02400100</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Configure RUN0 Mode:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• I/O Output power-down: no safe gating</td>
<td>PDO=0</td>
</tr>
<tr>
<td></td>
<td>• Main Voltage regulator is on (default)</td>
<td>MVRON=1</td>
</tr>
<tr>
<td></td>
<td>• Data, code flash in normal mode (default)</td>
<td>DFLAON = 3, CFLAON = 3</td>
</tr>
<tr>
<td></td>
<td>• PLL0 is switched on</td>
<td>PLL0ON=1</td>
</tr>
<tr>
<td></td>
<td>• Crystal oscillator is switched on</td>
<td>XOSC0ON=1</td>
</tr>
<tr>
<td></td>
<td>• 16 MHz IRC is switched on (default)</td>
<td>16MHz_IRCON=1</td>
</tr>
<tr>
<td></td>
<td>• Select PLL0 (system pll) as sysclk</td>
<td>SYSCLK=0x4</td>
</tr>
<tr>
<td></td>
<td>Assign peripheral configuration to peripherals:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• ADC 0: select ME_RUN_PC0</td>
<td>RUN_CFG = 1</td>
</tr>
<tr>
<td></td>
<td>• CTUL: select ME_RUN_PC0</td>
<td>RUN_CFG = 1</td>
</tr>
<tr>
<td></td>
<td>• SIUL: select ME_RUN_PC0</td>
<td>RUN_CFG = 1</td>
</tr>
<tr>
<td></td>
<td>• eMIOS 0: select ME_RUN_PC0</td>
<td>RUN_CFG = 1</td>
</tr>
<tr>
<td></td>
<td>Initiate software mode transition to RUN0 mode</td>
<td>TARGET_MODE=</td>
</tr>
<tr>
<td></td>
<td>• Mode and key</td>
<td>RUN0</td>
</tr>
<tr>
<td></td>
<td>• Mode and inverted key</td>
<td>S_MTRANS</td>
</tr>
<tr>
<td></td>
<td>• Wait for mode transition to complete</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Verify desired target mode was entered</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Disable Watchdog</td>
<td>Write keys to clear soft lock bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Clear watchdog enable bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>init Peri Clk Gen</strong></td>
<td>Initialize peripheral clock generation (See appendix: MPC56xxB/P/S Peripheral Clocks)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• ADC, CTU, eMIOS: peri. set 3- sysclk/1</td>
<td>DE2=1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DIV2=0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>init Pads</strong></td>
<td>Init pads:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• eMIOS0[2] as input</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• eMIOS0[3] as output</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• ADS0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• ADS1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Qorivva Simple Cookbook, Rev. 4

Freescale Semiconductor

181
### Table 89. MPC5606B Steps for MPC56xxB ADC CTU Example (continued)

<table>
<thead>
<tr>
<th>Step</th>
<th>Relevant Bit Fields</th>
<th>Pseudo Code</th>
</tr>
</thead>
</table>
| **init ADC** | Initialize ADC module but do not start conversions:  
| | • Configure Analog Clock as sysclk/2 (PLL/2 = 32 MHz for this example)  
| | • Mode is Scan Mode (continuous)  
| | • Trigger: on chip not used  
| | • ADC in normal mode, not power down mode  
| | • Enable CTU  
| | • Do not start conversions yet  
| | ADCLKSEL = 0  
| | MODE = 1  
| | TRIGEN = 0  
| | PWDN = 0  
| | CTUEN = 1  
| | NSTART = 0  
| | Enable normal sampling of desired channels for ADC 0 (Input pins ANS1, ANS2)  
| | CHAN33:34 = 0  
| | ADC_NCMR1 = 0x0000 0006  
| | Initialize conversion timings for 32 MHz ADCLK\(^1\)  
| | INPLATCH = 1  
| | INPCMP = 3  
| | INPSAMP = 6  
| | ADC_CTR1 = 0x0000 8606  
| | **init CTU** | Configure event on eMIOS ch. 2 to trigger ADC ch. 0  
| | • Enable trigger mask  
| | • Select input pint ANS0 (ADC ch 32=CTU ch 16)  
| | TM = 1  
| | CHANNEL_VALUE = 16 (0x10)  
| | CTU_EVTCFG2[2] = 0x0000 8010  
| | **init eMIOS** | eMIOS module: configure for 1 MHz internal clock  
| | (see Modulus Counter, OPWM example for details)  
| | Channel 23: initialize as modulus counter:  
| | Count 1000 × 1 usec clocks -> 1 kHz period  
| | (see Modulus Counter, OPWM example for details)  
| | Channel 3: initialize as OPWMB:  
| | Use Channel 23 as time base  
| | (see Modulus Counter, OPWM example for details)  
| | Channel 2: intialize as SAIC  
| | • Bus selected is ch 23  
| | • Edge Selected is a single edge  
| | • Edge Polarity is trigger on rising edge  
| | • Mode is SAIC  
| | • Flag enables IRQ or DMA request  
| | • DMA selected on flag  
| | (NOTE: CTU request generated instead of DMA request when CTU is enabled for that eMIOS channel)  
| | BSL = 0  
| | EDSEL = 0  
| | EDPOLY = 0  
| | MODE = 2  
| | FEN = 1  
| | DMA = 1  
| | EMIOS_0_CHAN[23]CADDR = 999  
| | EMIOS_0_CHAN[23]CCR = 0x8202 0650  
| | EMIOS_0_CHAN[3]CADDR = 250  
| | EMIOS_0_CHAN[3]CBDDR = 500  
| | EMIOS_0_CHAN[3]CCR = 0x0000 00E0  
| | EMIOS_0_CHAN[2]CCR = 0x0102 0082  
| | Normal trigger | Start Normal Conversions:  
| | • Start normal conversion (immediately)  
| | NSTART = 1  
| | ADC_MCR [NSTART] = 1  
| | Cross trigger | Start eMIOS module counting, which will generate PWM output for eMIOS channel 2 input  
| | GTPREN = 1  
| | EMIOS_0_MCR[GPREN] = 1  

# 23.3 Code

```c
/* main.c - ADC CTU example for MPC56xxB */
/* Description: Convert inputs ANS1:2 using normal scan mode and */
/* use eMIOS channel in SAIC mode with the CTU to trigger ANS0 */
/* Nov 11 2009 S Mihalik - initial version */
/* Mar 14 2010 S Mihalik - simplified initModesAndClock, updated header file */
/* Copyright Freescale Semiconductor, Inc 2009, 2010. All rights reserved. */

#include "MPC5604B_0M27V_0102.h" /* Use proper header file */
uint16_t Result; /* Read conversion result from ADC input ANS0 */

void initModesAndClock(void) {
    ME.MER = 0x0000001D; /* Enable DRUN, RUN0, SAFE, RESET modes */
    /* Initialize PLL before turning it on: */
    CGM.FMPLL_CR = 0x02400100; /* 8 MHz xtal: Set PLL0 to 64 MHz */
    ME.RUN[0].R = 0x001F0074; /* RUN0 cfg: 16MHzIRC,OSC0ON,PLOON,syclk=PLL0 */
    ME.RUNPC[1].R = 0x00000010; /* Peri. Cfg. 1 settings: only run in RUN0 mode */
    ME.PCTL[32].R = 0x01; /* MPC56xxB ADC 0: select ME.RUNPC[1] */
    ME.PCTL[57].R = 0x01; /* MPC56xxB CTUL: select ME.RUNPC[1] */
    ME.PCTL[68].R = 0x01; /* MPC56xxB SIU: select ME.RUNPC[1] */
    ME.PCTL[72].R = 0x01; /* MPC56xxB eMIOS 0: select ME.RUNPC[1] */
    /* Mode Transition to enter RUN0 mode: */
    ME.MCTL = 0x40005AF0; /* Enter RUN0 Mode & Key */
    ME.MCTL = 0x4000A50F; /* Enter RUN0 Mode & Inverted Key */
    while (ME.GS.B.S_MTRANS) {} /* Wait for mode transition to complete */
    while(ME.GS.B.S_CURRENTMODE != 4) {} /* Verify RUN0 is the current mode */
}

void disableWatchdog(void) {
    SWT.SR = 0x0000c520; /* Write keys to clear soft lock bit */
    SWT.SR = 0x0000d928;
    SWT.CR = 0x8000010A; /* Clear watchdog enable (WEN) */
}

void initPeriClkGen(void) {
    CGM.SC_DC[2].R = 0x80; /* MPC56xxB/S: Enable peri set 3 sysclk divided by 1 */
}

void initPads (void) {
    SIU.PCR[24].R = 0x2000; /* MPC56xxB: Initialize PB[8] as ANS0 */
    SIU.PCR[25].R = 0x2000; /* MPC56xxB: Initialize PB[9] as ANS1 */
    SIU.PCR[26].R = 0x2000; /* MPC56xxB: Initialize PB[10] as ANS2 */
}

void initADC(void) {
    ADC.MCR = 0x20020000; /* Initialize ADC */
    ADC.NCMR[1].R = 0x00000006; /* Select ANS1:2 inputs for normal conversion */
    ADC.CTR[1].R = 0x00008606; /* Conversion times for 32MHz ADClock */
}

void initCTU(void) {
    CTU.EVTCFGR[2].R = 0x00000810; /* Config event on eMIOS Ch 2 to trig ANS[0] */
}

void initEMIOS_0(void) {
    EMIOS_0.MCR.R = 0x30000000; /* Initialize eMIOS module for 1 MHz clock */
    EMIOS_0.CH[23].CADDR.R = 999; /* Ch 32: period will be 999+1 = 1K clks (1ms) */
    EMIOS_0.CH[23].CCR.R = 0x82020650; /* Ch 32: set mode as modulus counter */
    EMIOS_0.CH[3].CADDR.R = 250; /* Ch 3: Match "A" is 250 */
    EMIOS_0.CH[3].CBDR.R = 500; /* Ch 3: Match "B" is 500 */
    EMIOS_0.CH[3].CCR.R = 0x000000E0; /* Ch 3: Mode is OPWMB, time base = ch 23 */
    EMIOS_0.CH[2].CCR.R = 0x01020082; /* Ch 2: Mode is SAIC, time base = ch 23 */
}
```
void main (void) {
  uint32_t i = 0;      /* Dummy idle counter */

  initModesAndClock();   /* Initialize mode entries and system clock */
  disableWatchdog();     /* Disable watchdog */
  initPeriClkGen();      /* Initialize peripheral clock generation for DSPIs */
  initPads();            /* Initialize pads used in example */
  initADC();             /* Init. ADC for normal conversions but don't start yet */
  initCTU();             /* Configure desired CTU event(s) */
  initEMIOS_0();         /* Initialize eMIOS channels as counter, SAIC, OPWM */
  ADC.MCR.B.NSTART=1;    /* Trigger normal conversions for ADC0 */
  EMIOS_0.MCR.B.GPREN= 1; /* Start eMIOS counters to generate cross trigger */
  while (1) {
    while (ADC.CDR[33].B.VALID != 1) {};  /* Wait for cross trigger to complete */
    Result = ADC.CDR[32].B.CDATA;          /* Read ANS0 conversion result data */
    i++;
  }
}
24 DSPI: SPI to SPI

24.1 Description

Task: Transmit data from a DSPI configured as master to a DSPI configured as slave. The data sent by the master is 0x1234, and the slave will return 0x5678. This example shows how to configure a Clock and Transfer Attributes Register (CTAR) and other necessary items, then send a single command. Interrupts and DMA are not used.

Initialization and data is set up first before clearing the DSPI’s HALT bit, which immediately enables SPI operation. Setting the transmit command’s End Of Queue (EOQ) bit puts the DSPI in the stopped state at the end of that frame’s transmission. Clearing EOQ re-enables transmission if commands are present.

Exercise: (Note: external connections shown below are not required on MPC555x and MPC563x because of code that connects DSPIs internally in the SIU.) Change the data being sent and received. Debug tip: verify that SCK is at the desired baud rate by setting master DSPI MCR[CONT_SCK] with the debugger.

![Diagram of DSPI Single SPI Transfer Example](Signals shown use DSPI_C and DSPI_D)

Figure 37. DSPI Single SPI Transfer Example (Signals shown use DSPI_C and DSPI_D)
### Table 90. Signals for Single SPI Transfer Example using DSPI_C and DSPI_D

<table>
<thead>
<tr>
<th>Signal</th>
<th>Function Name</th>
<th>SIU PCR #</th>
<th>Package Pin #</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>496 BGA</td>
</tr>
<tr>
<td>PCS0_C</td>
<td>PCSC[0]</td>
<td>110</td>
<td>M26</td>
</tr>
<tr>
<td>SS_D</td>
<td>PCSD[0]</td>
<td>106</td>
<td>R28</td>
</tr>
<tr>
<td>SCK_C</td>
<td>SCKC</td>
<td>109</td>
<td>N27</td>
</tr>
<tr>
<td>SCK_D</td>
<td>SCKD</td>
<td>98</td>
<td>N26</td>
</tr>
<tr>
<td>SIN_C</td>
<td>SINC</td>
<td>108</td>
<td>M27</td>
</tr>
<tr>
<td>SOUT_D</td>
<td>SOUTD</td>
<td>100</td>
<td>U24</td>
</tr>
<tr>
<td>SOUT_C</td>
<td>SOUTC</td>
<td>107</td>
<td>T28</td>
</tr>
<tr>
<td>SIN_D</td>
<td>SIND</td>
<td>99</td>
<td>N24</td>
</tr>
</tbody>
</table>

### Table 91. Signals for Single SPI Transfer Example using DSPI_C and DSPI_B

<table>
<thead>
<tr>
<th>Signal</th>
<th>MPC51x Family</th>
<th>MPC55x Family</th>
<th>EVB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Pin Name</td>
<td>SIU PCR #</td>
<td>Package Pin #</td>
</tr>
<tr>
<td>PCS0_C</td>
<td>PB5</td>
<td>21</td>
<td>144 QFP</td>
</tr>
<tr>
<td>SS_B</td>
<td>PD12</td>
<td>60</td>
<td>90</td>
</tr>
<tr>
<td>SCK_C</td>
<td>PB6</td>
<td>22</td>
<td>128</td>
</tr>
<tr>
<td>SCK_B</td>
<td>PD13</td>
<td>61</td>
<td>89</td>
</tr>
<tr>
<td>SIN_C</td>
<td>PB8</td>
<td>24</td>
<td>126</td>
</tr>
<tr>
<td>SOUT_C</td>
<td>PB7</td>
<td>23</td>
<td>127</td>
</tr>
<tr>
<td>SIN_B</td>
<td>PD15</td>
<td>63</td>
<td>87</td>
</tr>
</tbody>
</table>
Table 92. MPC56xxB/P/S Signals for Single SPI Transfer Example using DSPI_0 and DSPI_1

<table>
<thead>
<tr>
<th>Signal</th>
<th>MPC56xxB Family</th>
<th></th>
<th>MPC56xxP Family</th>
<th></th>
<th>MPC56xxS Family</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Port</td>
<td>Package Pin #</td>
<td>Port</td>
<td>Package Pin #</td>
<td>Port</td>
<td>Package Pin #</td>
</tr>
<tr>
<td></td>
<td>SIU Pad Configuration &amp; Selection Registers (values in hex)</td>
<td>SIU Pad Configuration &amp; Selection Registers (values in hex)</td>
<td>SIU Pad Configuration &amp; Selection Registers (values in hex)</td>
<td>SIU Pad Configuration &amp; Selection Registers (values in hex)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Port</td>
<td>SIU Pad Configuration &amp; Selection Registers (values in hex)</td>
<td>SIU Pad Configuration &amp; Selection Registers (values in hex)</td>
<td>SIU Pad Configuration &amp; Selection Registers (values in hex)</td>
<td>SIU Pad Configuration &amp; Selection Registers (values in hex)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Package Pin #</td>
<td>100 QFP</td>
<td>144 QFP</td>
<td>176 BGA</td>
<td>100 QFP</td>
<td>144 QFP</td>
</tr>
<tr>
<td>SS_1</td>
<td>E5</td>
<td>PCR69=0903PSMI9=02</td>
<td>94 133</td>
<td>C6</td>
<td>A[5]</td>
<td>PCR5=0503</td>
</tr>
</tbody>
</table>
24.2 Design

Determining the timing clock and transfer attributes, as well as the SPI command itself, depends entirely on the connected device’s specification. In order to show an example of how to determine these items, specifications from an actual device, the MC33394, are used (except where noted).

24.2.1 Clock and Transfer Attribute Register Parameters

A 64 MHz sysclk is used in this example, which has a period of 15.625 nanoseconds. To keep the example simple, 15 nanoseconds will be used in the timing parameter calculations that follow.

The master SPI only needs one CTAR, since there is just one device for SPI communication. Slave SPIs must use CTAR0. The parameters below are based on the MC33394 data sheet, rev 2.5 11/2002, p. 15, pp. 21–22. Both master and slave SPIs will use the same CTAR values.

Frame Size: The MC33394 uses 16 bits for SPI communication.

Clock Polarity: The MC33394 uses SCK low in the inactive state.

Clock Phase: The MC33394 captures data on the leading edge, which is the rising edge since clock polarity is set to make SCK low when inactive.

LSB First: Although the MC33394 requires the least significant bit is sent first, the MSB will be sent first in this example to make it easier to decipher the logic analyzer trace shown in the end of this section.

PCS to SCK Delay ($t_{CSC}$): The MC33394 calls this parameter the “enable lead time.” The MC33394 minimum value is 105 ns. The formula for PCS to SCK delay in the MPC5554 is:

$$t_{CSC} = \text{(system clock period)} \times \text{(PCSSCK prescaler of 1,3,5 or 7)} \times \text{(CSSCK delay scaler of 2, 4, 8, etc.)}$$

With a system clock frequency of 64 MHz, the system clock period is about 15 nsec. This example will use:

PCSSCK (PCS to SCK delay prescaler) = 0 (for a prescaler value of 1)

CSSCK (PCS to SCK delay scaler) = 7 (for a scaler value of 256)

This gives a PCS to SCK delay of:

$$t_{CSC} = 15 \text{ ns} \times 1 \times 256 = 3.84 \mu\text{s}$$

After SCK Delay ($t_{ASC}$): This is the delay from the last edge of SCK to the negation of PCS. The MC33394 calls this parameter “enable lag time.” The MC33394 minimum value is 50 ns. The formula for after SCK delay in the MPC5554 is:

$$t_{ASC} = \text{(system clock period)} \times \text{(PASC prescaler of 1,3,5 or 7)} \times \text{(ASC delay scaler of 2, 4, 8, 16 etc.)}$$

With a system clock frequency of 64 MHz, the system clock period is about 15 ns. This example will use:

PASC (After SCK delay prescaler) = 0 (for a prescaler value of 1)

ASC (After SCK delay scaler) = 7 (for a scaler value of 256)

This gives an after SCK delay of:

$$t_{ASC} = 15 \text{ ns} \times 1 \times 256 = 3.84 \mu\text{s}$$
Delay after Transfer ($t_{DT}$): This is the length of time between the negation of PCS on the current frame and the assertion of PCS for the next frame. This example will only focus on a single transfer, but the calculation is shown here in case of a different situation. The MC33394 has a minimum time, called “CS Negated Time,” of 500 ns.

\[ t_{DT} = (\text{system clock period}) \times (\text{PDT prescaler of 1, 3, 5 or 7}) \times (\text{DT delay scaler of 2, 4, 8, 16 etc.}) \]

With a system clock frequency of 64 MHz, the system clock period is about 15 ns. This example will use:

- PDT (Delay after Transfer prescaler) = 2 (for a prescaler value of 5)
- DT (Delay after Transfer scaler) = 2 (for a prescaler value of 8)

This gives an after SCK delay of:

\[ t_{DT} = 15 \text{ ns} \times 5 \times 8 = 600 \text{ ns} \]

Baud Rate (BR): The baud rate maximum for the MC33394 is 5 MHz, but we will use close to 100 kHz here. The formula for DSPI baud rate in the MPC5554 is:

\[ \text{SCK Baud Rate} = (f_{SYS} / \text{PBR prescaler}) \times ((1+\text{DBR})/ (\text{BR scaler})) \]

With a system clock frequency of 64 MHz, this example will use:

- DBR = 0 (double baud rate feature not used)
- PBR = 2 (for a prescaler of 5)
- BR = 7 (for a scaler of 128)

Hence the baud rate is

\[ \text{SCK Baud Rate} = (64 \text{ MHz} / 5) \times ((1+0) / 128) = 100 \text{ kHz} \ (10 \mu\text{s SCK period}) \]

24.2.2 SPI Command

The SPI command will be written to the SPI’s push register, which will then automatically fall through the FIFO. The fields are listed below, again based on the MC33394 data sheet.

Continuous Peripheral Chip Select: Continuous PCS is not used and is inactive between transfers.

Clock and Transfer Register: CTAR0 is required for the slave DSPI. The master will use CTAR0 also.

End of Queue: Only one transfer is used in this example, so the EOQ bit will be set.

Peripheral Chip Selects used: PCS0 will be connected from the master to the SS of the slave DSPI.

Transfer Data: The slave DSPI (DSPI_D or DSPI_B) will have data 0x1234 in its shift register to respond to a transfer from the master DSPI (DSPI_C). The master will transmit 0x5678 for data.

24.2.3 Pad Slew Rate Control versus SPI Baud Rate

Pads must be driven harder for faster signals. This example’s baud rate of 100 kHz is relatively slow for a SPI. Faster baud rates require increasing the SRC field value in the respective pad configuration registers, SIU_PCR[SRC].

Qorivva Simple Cookbook, Rev. 4

Freescale Semiconductor 189
24.2.4 Mode Use Summary (MPC56xxB/P/S only)

Mode Transition is required for changing mode entry registers. Hence even enabling the crystal oscillator to be active in the default mode (DRUN) requires enabling the crystal oscillator in appropriate mode configuration register (ME_xxxx_MC) then initiating a mode transition. This example transitions from the default mode after reset (DRUN) to RUN0 mode.

Table 93. Mode Configurations for MPC56xxB/P/S DSPI SPI to SPI Example

Modes are enabled in ME_ME Register.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Mode Config. Register</th>
<th>Mode Config. Register Value</th>
<th>sysclk Selection</th>
<th>Clock Sources</th>
<th>Memory Power Mode</th>
<th>Main Voltage Reg.</th>
<th>I/O Power Down Ctrl</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRUN</td>
<td>ME_DRUN_MC</td>
<td>0x001F 0010 (default)</td>
<td>16 MHz IRC</td>
<td>On</td>
<td>Off</td>
<td>Normal</td>
<td>On</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Normal</td>
<td>Off</td>
</tr>
<tr>
<td>RUN0</td>
<td>ME_RUN0_MC</td>
<td>0x001F 007D</td>
<td>PLL0</td>
<td>On</td>
<td>On</td>
<td>Normal</td>
<td>On</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Normal</td>
<td>Off</td>
</tr>
</tbody>
</table>

Other modes are not used in example

Peripherals also have configurations to gate clocks on or off for different modes, enabling low power. The following table summarizes the peripheral configurations used in this example.

Table 94. Peripheral Configurations for MPC56xxB/P/S DSPI SPI to SPI Example

Low power modes are not used in example.

<table>
<thead>
<tr>
<th>Peripheral Config.</th>
<th>Peri. Config. Register</th>
<th>Enabled Modes</th>
<th>Peripherals Selecting Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>RUN3 RUN2 RUN1 RUN0 DRUN SAFE TEST RESET</td>
<td>Peripheral</td>
</tr>
<tr>
<td>PC1</td>
<td>ME_RUNPC_1</td>
<td>0 0 0 1 0 0 0 0</td>
<td>DSPI 0 DSPI 1 SIUL (MPC56xxB/S)</td>
</tr>
</tbody>
</table>

Other peripheral configurations are not used in example
### 24.2.5 Steps and Pseudo Code

Table 95. Steps and Pseudo Code Table (shown using DSPI_C as Master, DSPI_D as Slave)

<table>
<thead>
<tr>
<th>Step</th>
<th>Relevant Bit Fields</th>
<th>Pseudo Code MPC51x</th>
<th>Pseudo Code MPC55x</th>
<th>Pseudo Code MPC56xxB/P/S</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data Init.</strong></td>
<td>Initialize data to be received on master SPI</td>
<td>RecDataMaster = 0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Initialize data to be received on slave SPI</td>
<td>RecDataSlave = 0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>init Modes and Clock (MPC56xxPBS only)</strong></td>
<td>Enable desired modes</td>
<td>DRUN=1, RUN0 = 1</td>
<td>-</td>
<td>ME_ME = 0x0000 001D</td>
</tr>
<tr>
<td></td>
<td>Initialize PLL0 dividers to provide 64 MHz for input crystal before mode configuration:</td>
<td></td>
<td>8 MHz Crystal: CGM_FMPLL[0]_CR = 0x02400100</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- 8 MHz xtal: FMPLL[0]_CR=0x02400100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- 40 MHz xtal: FMPLL[0]_CR=0x12400100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(MPC56xxP &amp; 8 MHz xtal requires changing default CMU_CSR value. See PLL example.)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Configure RUN0 Mode:</td>
<td>PDO=0, MVRON=1, DFLAON, CFLAON= 3, PLL0ON=1, XOSSCON=0, 16MHz_IRCON=1, SYSCLK=0x4</td>
<td>-(MPC56xxP &amp; 8 MHz xtal requires changing default CMU_CSR value. See PLL example.)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- I/O Output power-down: no safe gating</td>
<td></td>
<td>ME_RUN_MC = 001F 0070</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Main Voltage regulator is on (default)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Data, code flash in normal mode (default)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- PLL0 is switched on</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Crystal oscillator is switched on</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- 16 MHz IRC is switched on (default)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Select PLL0 (system pll) as sysclk</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>MPC56xxB/S:</strong></td>
<td>RUN0=1</td>
<td>-</td>
<td>ME_RUN_PC1 = 0000 0010</td>
</tr>
<tr>
<td></td>
<td>- Peri. Config. 1: run in RUN0 mode only</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Assign peripheral config. to peripherals:</strong></td>
<td>RUN_CFG = 1</td>
<td>-</td>
<td>ME_PCTL4 = ME_PCTL5 = ME_PCTL68 = 0x01</td>
</tr>
<tr>
<td></td>
<td>- DSPI 0: select ME_RUN_PC1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- DSPI 1: select ME_RUN_PC1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- SIUL: select ME_RUN_PC1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(MPC56xxB/S)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Initiate software mode transition to RUN0 mode:</strong></td>
<td>TARGET_MODE = RUN0, S_TRANS, CURRENTMODE</td>
<td>-</td>
<td>ME_MCTL = 0x4000 5AF0,</td>
</tr>
<tr>
<td></td>
<td>- Mode &amp; key, then mode &amp; inverted key</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Wait for transition to complete</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Verify current mode is RUN0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Optional step - allows not using external connections for MPC555x, MPC563x only)</td>
<td></td>
<td>MPC555x: SIU_DISR = 0x0000 C0FC</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- MPC555x: Connect master DSPI C to slave DSPI D internally</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- MPC563x: Connect master DSPI C to slave DSPI D internally</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>init Sysclk</strong></td>
<td>Initialize sysclk to 64 MHz, running from PLL</td>
<td>See PLL Initialization example</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td><strong>init Peri Clk Gen (MPC56xxPBS)</strong></td>
<td>Initialize peripheral clock generation (See appendix: MPC56xxB/P/S Peripheral Clocks)</td>
<td>-</td>
<td>MPC56xxB/S: CGM_SC_DC1= 0x80</td>
<td></td>
</tr>
<tr>
<td>Step</td>
<td>Relevant Bit Fields</td>
<td>Pseudo Code</td>
<td>MPC551x</td>
<td>MPC555x</td>
</tr>
<tr>
<td>------</td>
<td>---------------------</td>
<td>-------------</td>
<td>---------</td>
<td>---------</td>
</tr>
<tr>
<td>Disable Watchdog</td>
<td>Disable watchdog by writing keys to Status Register, then clearing WEN (MPC56xxBPS only)</td>
<td></td>
<td>-</td>
<td>See PLL Initialization example</td>
</tr>
<tr>
<td>init</td>
<td>Initialize DSPI as master</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DSPI_C or DSPI_0 (master)</td>
<td>• SPI is master</td>
<td>DSPIC_MCR = 0x8001 0001</td>
<td></td>
<td>DSPI0_MCR = 0x8001 0001</td>
</tr>
<tr>
<td></td>
<td>• SCK will not be continuous</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• DSPI is configured for SPI only</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Debug Freeze does not halt transfers</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Modified transfer format is not used</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Peripheral chip select strobe is not used</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Ignore receive FIFO overflows</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Peripheral Chip Select 0 inactive state = high</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Doze, module disable not used</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Transmit FIFO not disabled</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Receive FIFO not disabled</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Transmit FIFO not cleared</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Receive FIFO not cleared</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Sample points not used</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Halt state enabled (STOPPED) for initialization</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Configure CTAR0 for MC33394</td>
<td>• Frame Size = 16 bits</td>
<td>DSPIC_CTAR0 = 0x780A 7727</td>
<td></td>
<td>DSPI0_CTAR0 = 0x780A 7727</td>
</tr>
<tr>
<td></td>
<td>• Clock polarity: low inactive state</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Data is captured on leading edge</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Most (not least) significant bit is first in frame</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• PCS to SCK delay = 3.84 usec (&gt;105 ns required)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• After SCK delay = 3.84 us (&gt;50 ns required)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Delay after transfer = 800 ns (&gt;500 ns required)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Double Baud Rate feature not used</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Baud Rate = 78.125 kHz (&lt;5 MHz required)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Change DSPI from STOPPED to RUNNING</td>
<td>HALT = 0</td>
<td>DSPIC_MCR = 0x8001 0000</td>
<td></td>
<td>DSPI0_MCR = 0x8001 0000</td>
</tr>
<tr>
<td>Configure pads for Master (DSPI_C or DSPI_0):</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• SOUT_x output</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• SIN_x input</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• SCK_x output</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• PCSC0_x output</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Note: Faster baud rates would require increasing the pad’s slew rate control in SIU_PCR[SRC].</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 95. Steps and Pseudo Code Table (shown using DSPI_C as Master, DSPI_D as Slave) (continued)
### Table 95. Steps and Pseudo Code Table (shown using DSPI_C as Master, DSPI_D as Slave) (continued)

<table>
<thead>
<tr>
<th>Step</th>
<th>Relevant Bit Fields</th>
<th>Pseudo Code</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong> DSPI_C or DSPI_D Response</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enable as slave (other fields same as master DSPI):</td>
<td>MSTR=0</td>
<td>DSPI1_MCR = 0x0001 0001</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DSPI1_MCR = 0x0001 0001</td>
</tr>
<tr>
<td>Configure CTAR0 same as for master DSPI:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DSPI1_CTAR0 = 0x780A 7727</td>
<td></td>
</tr>
<tr>
<td>Change DSPI from STOPPED to RUNNING:</td>
<td>HALT = 0</td>
<td>DSPI1_MCR = 0x8001 0000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DSPI1_MCR = 0x8001 0000</td>
</tr>
<tr>
<td>Configure pads for Slave (DSPI_B, DSPI_D, DSPI_1):</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• SCK_x input</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• SIN_x input</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• SOUT_x output</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• PCS0/SS_x input</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Note: Faster baud rates would require increasing the pad’s slew rate control in SIU_PCR[SRC].</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SIU_PCR[42] = 0x0B00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SIU_PCR[44] = 0x0B00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SIU_PCR[43] = 0x0D00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SIU_PCR[45] = 0x1100</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SIU_PCR[98] = 0x0900</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SIU_PCR[99] = 0x0900</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SIU_PCR[100] = 0x0A00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SIU_PCR[106] = 0x0900</td>
<td></td>
</tr>
<tr>
<td>See table: MPC56xxB/P/S Signals for SIU_PCR &amp; SIU_PSMI Registers and Values.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Init</strong> DSPI_C or DSPI_D Response</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Initialize response data from slave DSPI:</td>
<td>TxDATA=0x1234</td>
<td>DSPI1_PUSHR = 0x0000_1234</td>
</tr>
<tr>
<td>• Data = 0x1234</td>
<td></td>
<td>DSPI1_PUSHR = 0x0000_1234</td>
</tr>
<tr>
<td><strong>Transmit</strong> DSPI_C or DSPI_D Response</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transmit a single SPI command from DSPI:</td>
<td>CONT=0</td>
<td>DSPI0_PUSHR = 0x0801_5678</td>
</tr>
<tr>
<td>• Continuous PCS disabled</td>
<td>CTAS=0</td>
<td>DSPI0_PUSHR = 0x0801_5678</td>
</tr>
<tr>
<td>• Clock and Transfer Register 0 used</td>
<td>EOQ=1</td>
<td></td>
</tr>
<tr>
<td>• End of Queue is set (only one command sent).</td>
<td>CTCNT=0</td>
<td></td>
</tr>
<tr>
<td>• Do not Clear SPI Transfer Count</td>
<td>PCS0=1, other PCSx=0</td>
<td></td>
</tr>
<tr>
<td>• Transfer Data = 0x5678</td>
<td>TxDATA=0x5678</td>
<td></td>
</tr>
</tbody>
</table>
### Design Screenshot

The signals for this example are shown in the screenshot below for master DSPI_C and slave DSPI_D. The data is MSB first, and the frame size is 16 bits wide.

![Design Screenshot](image)

*Figure 38. DSPI Example Signals (Captured on MPC555x EVB)*
### 24.3 Code

#### 24.3.1 MPC551x (DSPI_C master, DSPI_B slave)

```c
/* main.c: performs a single transfer from DSPI_C to DSPI_B on MPC551x */
/* Rev 1.0 Sept 14 2004 S.Mihalik */
/* Rev 2.0 Jan 3 2007 S. Mihalik - Modified to use two SPIs */
/* Rev 2.1 July 20 2007 SM - Modified for MPC551x, changed sysclk (50 MHz) */
/* Rev 2.2 Aug 13 2007 SM - Modified for sysclk of 64 MHz & lengthened CSSCK, ASC */
/* Rev 2.3 Jun 04 2008 SM - initSysclk changed for MPC5633M support */
/* Rev 2.4 Aug 14 2008 SM - Switched slave DSPI to DSPI_B, removed 555x lines */
/* Rev 2.5 Aug 18 2008 D McKenna - Kept DSPI_MCR[HALT] set during DSPI initialization */
/* Rev 2.6 Aug 12 2009 SM - Changed PLL initial ERFD value, added 12MHz crystal */
/* Copyright Freescale Semiconductor, Inc. 2007 All rights reserved. */
#include "mpc5510.h" /* Use proper include file like mpc5510.h or mpc5554.h */

#define _32_T i = 0;                      /* Dummy idle counter */
#define _16_T RecDataMaster = 0;           /* Data recievied on master SPI */
#define _16_T RecDataSlave  = 0;           /* Data received on slave SPI */

void initSysclk(void) { /* Initialize PLL and sysclk to 64 MHz */
    /* Use appropriate code for crystal*/
    FMPLL.ESYNCR2.R = 0x00000007;     /* 8MHz xtal: ERFD to initial value of 7 */
    FMPLL.ESYNCR1.R = 0xF0000020;     /* 8MHz xtal: CLKCFG=PLL, EPREDIV=0, EMFD=0x20 */
    FMPLL.ESYNCR1.R = 0x00000000;     /* 12MHz xtal: ERFD to initial value of 5 */
    FMPLL.ESYNCR2.R = 0xF0020030;     /* 12MHz xtal: CLKCFG=PLL, EPREDIV=2, EMFD=0x30 */
    CRP.CLKSR.B.XOSCEN = 1;          /* Enable external oscillator */
    while (FMPLL.SYNSR.B.LOCK != 1) {}; /* Wait for PLL to LOCK */
    FMPLL.ESYNCR2.R = 0x00000005;     /* 8MHz xtal: ERFD change for 64 MHz sysclk */
    SIU.SYSCLK.B.SYSCLKSEL = 2;       /* Select PLL for sysclk */
}

void initDSPI_C(void) {
    DSPI_C.MCR.R = 0x80010001;     /* Configure DSPI_C as master */
    DSPI_C.CTAR[0].R = 0x780A7727; /* Configure CTAR0 */
    DSPI_C.MCR.B.HALT = 0x0;     /* Exit HALT mode: go from STOPPED to RUNNING state*/
    SIU.PCR[23].R = 0x0A00;       /* MPC551x: Config pad as DSPI_C SOUT output */
    SIU.PCR[24].R = 0x0900;       /* MPC551x: Config pad as DSPI_C SIN input */
    SIU.PCR[22].R = 0x0A00;       /* MPC551x: Config pad as DSPI_C SCK output */
    SIU.PCR[21].R = 0x0A00;       /* MPC551x: Config pad as DSPI_C PCS0 output */
}

void initDSPI_B(void) {
    DSPI_B.MCR.R = 0x00010001;     /* Configure DSPI_B as slave */
    DSPI_B.CTAR[0].R = 0x780A7727; /* Configure CTAR0 */
    DSPI_B.MCR.B.HALT = 0x0;     /* Exit HALT mode: go from STOPPED to RUNNING state*/
    SIU.PCR[61].R = 0x0500;       /* MPC551x: Config pad as DSPI_B SCK input */
    SIU.PCR[63].R = 0x0500;       /* MPC551x: Config pad as DSPI_B SIN input */
    SIU.PCR[62].R = 0x0600;       /* MPC551x: Config pad as DSPI_B SOUT output*/
    SIU.PCR[60].R = 0x0500;       /* MPC551x: Config pad as DSPI_B PCS0/SS input */
}

void ReadDataDSPI_B(void) {
    while (DSPI_B.SR.B.RFDF != 1){}  /* Wait for Receive FIFO Drain Flag = 1 */
    RecDataSlave = DSPI_B.POPR.R;    /* Read data received by slave SPI */
    DSPI_B.SR.R = 0x80020000;        /* Clear TCF, RDRF flags by writing 1 to them */
}

void ReadDataDSPI_C(void) {
    while (DSPI_C.SR.B.RFDF != 1){}  /* Wait for Receive FIFO Drain Flag = 1 */
    RecDataMaster = DSPI_C.POPR.R;   /* Read data received by master SPI */
    DSPI_C.SR.R = 0x90020000;        /* Clear TCF, RDRF, EOQ flags by writing 1 to them */
}

int main(void) { /* Set sysclk = 64MHz running from PLL */
    initSysclk();        /* Initialize DSPI C as master SPI and init CTAR0 */
    initDSPI_C();        /* Initialize DSPI B as Slave SPI and init CTAR0 */
    DSPI_B.PUSHR.R = 0x00001234; /* Transmit data from master to slave SPI with EOQ */
    DMSPI.C.PUSHR.R = 0x08015678; /* Transmit data from master to slave SPI with EOQ */
    while (1) i++;       /* Wait forever */
}
```
# 24.3.2 MPC555x (DSPI_C master, DSPI_D slave)

/* main.c: performs a single transfer from DSPI_C to DSPI_D on MPC555x*/
/* Rev 1.0 Sept 14 2004 S.Mihalik */
/* Rev 2.0 Jan 3 2007 S. Mihalik - Modified to use two SPIs */
/* Rev 2.1 July 20 2007 SM - Modified for MPC551x, changed sysclk (50 MHz) */
/* Rev 2.2 Aug 13 2007 SM - Modified for sysclk of 64 MHz & lengthened CSSCK, ASC*/
/* Rev 2.3 Jun 04 2008 SM - initSysclk changed for MPC5633M support */
/* Rev 2.4 Aug 15 2008 SM - removed lines for MPC551x, MPC563x */
/* Rev 2.5 Aug 18 2008 D McKenna- Kept DSPI_MCR[HALT] set during initialization*/
/* Copyright Freescale Semiconductor, Inc. 2007 All rights reserved. */
/* Notes: */
/* 1. MMU not initialized; must be done by debug scripts or BAM */
/* 2. SRAM not initialized; must be done by debug scripts or in a crt0 type file */

#include "mpc5554.h" /* Use proper include file like mpc5510.h or mpc5554.h */

vuint32_t i = 0;                      /* Dummy idle counter */
uint16_t RecDataMaster = 0;           /* Data received on master SPI */
uint16_t RecDataSlave  = 0;           /* Data received on slave SPI */

void initSysclk (void) {
  FMPLL.SYNCR.R = 0x16080000;         /* 8 MHz xtal: 0x16080000; 40MHz: 0x46100000 */
  while (FMPLL.SYNSR.B.LOCK != 1) {}; /* Wait for FMPLL to LOCK */
  FMPLL.SYNCR.R = 0x16000000;         /* 8 MHz xtal: 0x16000000; 40MHz: 0x46080000 */
}

void initDSPI_C(void) {
  DSPI_C.MCR.R = 0x80010001;     /* Configure DSPI_C as master */
  DSPI_C.CTAR[0].R = 0x780A7727; /* Configure CTAR0 */
  DSPI_C.MCR.B.HALT = 0x0;    /* Exit HALT mode: go from STOPPED to RUNNING state*/
  SIU.PCR[107].R = 0x0A00;       /* MPC555x: Config pad as DSPI_C SOUT output */
  SIU.PCR[109].R = 0x0A00;       /* MPC555x: Config pad as DSPI_C SCK output */
  SIU.PCR[110].R = 0x0A00;       /* MPC555x: Config pad as DSPI_C PCS0 output */
}

void initDSPI_D(void) {
  DSPI_D.MCR.R = 0x00010001;     /* Configure DSPI_D as slave */
  DSPI_D.CTAR[0].R = 0x780A7727; /* Configure CTAR0 */
  DSPI_D.MCR.B.HALT = 0x0;     /* Exit HALT mode: go from STOPPED to RUNNING state*/
  SIU.PCR[98].R = 0x0900;        /* MPC555x: Config pad as DSPI_D SCK input */
  SIU.PCR[99].R = 0x0900;        /* MPC555x: Config pad as DSPI_D SIN input */
  SIU.PCR[100].R = 0x0A00;       /* MPC555x: Config pad as DSPI_D SOUT output*/
  SIU.PCR[106].R = 0x0900;       /* MPC555x: Config pad as DSPI_D PCS0/SS input */
}

void ReadDataDSPI_D(void) {
  while (DSPI_D.SR.B.RFDF != 1) {}/* Wait for Receive FIFO Drain Flag = 1 */
  RecDataSlave = DSPI_D.POPR.R;    /* Read data received by slave SPI */
  DSPI_D.SR.R = 0x80020000;        /* Clear TCF, RDRF flags by writing 1 to them */
}

void ReadDataDSPI_C(void) {
  while (DSPI_C.SR.B.RFDF != 1) {}/* Wait for Receive FIFO Drain Flag = 1 */
  RecDataMaster = DSPI_C.POPR.R;   /* Read data received by master SPI */
  DSPI_C.SR.R = 0x90020000;        /* Clear TCF, RDRF, EOQ flags by writing 1 to them */
}

int main(void) {
  SIU.DISR.R = 0x0000C0FC;         /* MPC555x only: Connect DSPI_C, DSPI_D internally */
  initSysclk();                   /* Set sysclk = 64MHz running from PLL */
  initDSPI_C();                   /* Initialize DSPI_C as master SPI and init CTAR0 */
  initDSPI_D();                   /* Initialize DSPI_D as Slave SPI and init CTAR0 */
  DSPI_D.PUSHR.R = 0x00001234;     /* Initialize slave DSPI_D's response to master */
  DSPI_C.PUSHR.R = 0x00001234;     /* Transmit data from master to slave SPI with EOQ */
  ReadDataDSPI_D();               /* Read data on slave DSPI */
  ReadDataDSPI_C();               /* Read data on master DSPI */
  while (1) {i++; }               /* Wait forever */
}
24.3.3 MPC563x (DSPI_C master, DSPI_B slave)

/* main.c: Performs a single transfer from DSPI C to DSPI B */
/* Rev 1.0 Jun 2 2008 SM - Ported from AN2865 example Rev Z.2 for DSPI C, DSPI B */
/* and used POPR[RXDATA] for RecDataMaster, RecDataSlave */
/* Rev 1.2 Aug 10 2008 D McKenna- Kept DSPI MCR[HALT] set during initialization */
/* Copyright Freescale Semiconductor, Inc. 2007 All rights reserved. */
/* Notes: */
/* 1. MMU not initialized; must be done by debug scripts or BAM */
/* 2. SRAM not initialized; must be done by debug scripts or in a crt0 type file */
#include "mpc563m.h" /* Use proper include file such as mpc5510.h or mpc5554.h */

uint32_t i = 0;                /* Dummy idle counter */
uint32_t RecDataMaster = 0;     /* Data received on master SPI */
uint32_t RecDataSlave  = 0;     /* Data received on slave SPI */

void initSysclk(void) {
  /* MPC563x: Use the next line */
  FMPLL.ESYNCR.B.CLKCFG = 0x7;  /* MPC563x: Change clk to PLL normal from crystal*/
  FMPLL.SYNCR.R = 0x16080000;    /* 8 MHz xtal: 0x16080000; 40MHz: 0x46100000 */
  while (FMPLL.SYNSR.B.LOCK != 1) {}; /* Wait for FMPLL to LOCK */
  FMPLL.SYNCR.R = 0x16000000;    /* 8 MHz xtal: 0x16000000; 40MHz: 0x46080000 */
}

void initDSPI_C(void) {
  DSPI_C.MCR.R = 0x80010001;    /* Configure DSPI C as master */
  DSPI_C.CTAR[0].R = 0x780A7727; /* Configure CTAR0 */
  DSPI_C.MCR.B.HALT = 0x0;      /* Exit HALT mode: go from STOPPED to RUNNING state*/
  SIU.PCR[107].R = 0x0A00;      /* MPC555x: Config pad as DSPI C SOUT output */
  SIU.PCR[108].R = 0x0900;      /* MPC555x: Config pad as DSPI C SIN input */
  SIU.PCR[109].R = 0x0A00;      /* MPC555x: Config pad as DSPI C SCK output */
  SIU.PCR[110].R = 0x0A00;      /* MPC555x: Config pad as DSPI C PCS0 output */
}

void initDSPI_B(void) {
  DSPI_B.MCR.R = 0x00010001;    /* Configure DSPI B as slave */
  DSPI_B.CTAR[0].R = 0x780A7727; /* Configure CTAR0 */
  DSPI_B.MCR.B.HALT = 0x0;      /* Exit HALT mode: go from STOPPED to RUNNING state*/
  SIU.PCR[102].R = 0x0500;      /* MPC555x: Config pad as DSPI B SCK input */
  SIU.PCR[103].R = 0x0500;      /* MPC555x: Config pad as DSPI B SIN input */
  SIU.PCR[104].R = 0x0600;      /* MPC555x: Config pad as DSPI B SOUT output*/
  SIU.PCR[105].R = 0x0500;      /* MPC555x: Config pad as DSPI B PCS0/SS input */
}

void ReadDataDSPI_B(void) {
  while (DSPI_B.SR.B.RFDF != 1){}      /* Wait for Receive FIFO Drain Flag = 1 */
  RecDataSlave = DSPI_B.POPR.B.RXDATA; /* Read data received by slave SPI */
  DSPI_B.SR.R = 0x80020000;          /* Clear TCF, RDRF flags by writing 1 to them */
}

void ReadDataDSPI_C(void) {
  while (DSPI_C.SR.B.RFDF != 1){}  /* Wait for Receive FIFO Drain Flag = 1 */
  RecDataMaster = DSPI_C.POPR.B.RXDATA; /* Read data received by master SPI */
  DSPI_C.SR.R = 0x90020000;        /* Clear TCF, RDRF, EOQ flags by writing 1 */
}

int main(void) {
  /* Optional: Use one of the next two lines for internal DSPI connections: */
  /* SIU.DISR.R = 0x00000C0FC; */ /* MPC55xx except MPC563x: Connect DSPI C, DSPI_D */
  SIU.DISR.R = 0x00000A00;        /* MPC563x only: Connect DSPI C, DSPI_B */
  initSysclk();                  /* Set sysclk = 64MHz running from PLL */
  initDSPI_C();                  /* Initialize DSPI C as master SPI and init CTAR0 */
  initDSPI_B();                  /* Initialize DSPI B as Slave SPI and init CTAR0 */
  DSPI_B.PUSHR.R = 0x00001234;    /* Initialize slave DSPI B's response to master */
  DSPI_C.PUSHR.R = 0x08015678;    /* Transmit data from master to slave SPI with EOQ */
  ReadDataDSPI_B();              /* Read data on slave DSPI */
  ReadDataDSPI_C();              /* Read data on master DSPI */
  while (1) (i++);               /* Wait forever */
}
#include "Pictus_Header_v1_09.h" /* Use proper include file */
uint32_t i = 0; /* Dummy idle counter */
uint16_t RecDataMaster = 0; /* Data received on master SPI */
uint16_t RecDataSlave  = 0; /* Data received on slave SPI */

void initModesAndClks(void) {
  ME.MER.R = 0x0000001D; /* Enable DRUN, RUN0, SAFE, RESET modes */
  /* Initialize XOSC, PLL before turning them on: */
  /* Use 2 of the next 4 lines depending on crystal frequency: */
  CGM.CMU_0_CSR.R = 0x000000004; /* Monitor FXOSC > FIRC/4 (4MHz); no PLL monitor*/
  CGM.FMPLL[0].CR.R = 0x02400100; /* 8 MHz xtal: Set PLL0 to 64 MHz */
  CGM.CMU_0_CSR.R = 0x000000000; /* Monitor FXOSC > FIRC/1 (16MHz); no PLL monitor*/
  CGM.FMPLL[0].CR.R = 0x12400100; /* 40 MHz xtal: Set PLL0 to 64 MHz */
  ME.RUN[0].R = 0x001P0074; /* RUN0 cfg: 16MHzIRCON,OSC0ON,PLL0ON,syclk=PLL */
  ME.RUNPC[1].R = 0x00000010; /* Peri. Cfg. 1 settings: only run in RUN0 mode */
  ME.PCTL[4].R = 0x00000010; /* MPC56xxB/P/S DSPI0: select ME.RUNPC[1] */
  ME.PCTL[5].R = 0x00000010; /* MPC56xxB/P/S DSPI1: select ME.RUNPC[1] */
  ME.PCTL[68].R = 0x00000010; /* MPC56xxB/S SIUL: select ME.RUNPC[1] */
  ME.MCTL.R = 0x40005AF0; /* Mode Transition to enter RUN0 mode: */
  ME.MCTL.R = 0x4000A50F; /* Enter RUN0 Mode & Inverted Key */
  while (ME.GS.B.S_MTRANS) {} /* Wait for mode transition to complete */
  /* Note: could wait here using timer and/or I TC IRQ */
  while(ME.GS.B.S_CURRENTMODE != 4) {} /* Verify RUN0 is the current mode */
}

void initPeriClkGen(void) { /* Use the following code as required for MPC56xxB or MPC56xxS:*/
  CGM.SC_DC[1].R = 0x80; /* MPC56xxB/S: Enable peri set 2 sysclk divided by 1*/
}

void disableWatchdog(void) { /* MPC56xx/P: Enable peri set 2 sysclk divided by 1*/
  SWT.SR.R = 0x00000c520; /* Write keys to clear soft lock bit */
  SWT.SR.R = 0x0000d928;
  SWT.CR.R = 0x0000010A; /* Clear watchdog enable (WEN) */
}

void initDSPI_0(void) { /* Configure DSPI 0 as master */
  DSPI_0.MCR.R = 0x80010001; /* Exit HALT mode: go from STOPPED to RUNNING state*/
  SIU.PCR[38].R = 0x0604; /* MPC56xxP: Config pad as DSPI 0 SOUT output */
  SIU.PCR[39].R = 0x0103; /* MPC56xxP: Config pad as DSPI 0 SIN input */
  SIU.PCR[37].R = 0x0604; /* MPC56xxP: Config pad as DSPI 0 SCK output */
  SIU.PCR[36].R = 0x0604; /* MPC56xxP: Config pad as DSPI 0 PCS0 output */
}

void initDSPI_1(void) { /* Configure DSPI 1 as slave */
  DSPI_1.MCR.R = 0x00001001; /* Exit HALT mode: go from STOPPED to RUNNING state*/
  SIU.PCR[6].R = 0x0503; /* MPC56xxP: Config pad as DSPI 1 SCK input */
  SIU.PCR[8].R = 0x0103; /* MPC56xxP: Config pad as DSPI1 SIN input */
  SIU.PCR[7].R = 0x0604; /* MPC56xxP: Config pad as DSPI1 SOUT output */
  SIU.PCR[5].R = 0x0503; /* MPC56xxP: Config pad as DSPI1 PCS0/SS input */
}
void ReadDataDSPI_1(void) {
    while (DSPI_1.SR.B.RFDF != 1){} /* Wait for Receive FIFO Drain Flag = 1 */
    RecDataSlave = DSPI_1.POPR.R;    /* Read data received by slave SPI */
    DSPI_1.SR.R = 0x80020000;        /* Clear TCF, RDRF flags by writing 1 to them */
}

void ReadDataDSPI_0(void) {
    while (DSPI_0.SR.B.RFDF != 1){} /* Wait for Receive FIFO Drain Flag = 1 */
    RecDataMaster = DSPI_0.POPR.R;   /* Read data received by master SPI */
    DSPI_0.SR.R = 0x90020000;        /* Clear TCF, RDRF, EOQ flags by writing 1 */
}

int main(void) {
    initModesAndClks();           /* Initialize mode entries and system clock */
    initPeriClkGen();             /* Initialize peripheral clock generation for DSPIs*/
    disableWatchdog();           /* Disable watchdog */
    initDSPI_0();                 /* Initialize DSPI 0 as master SPI and init CTAR0 */
    DSPI_0.PUSHR.R = 0x00001234;   /* Initialize slave DSPI 1's response to master */
    DSPI_0.FUSHR.R = 0x08015678;   /* Transmit data from master to slave SPI with EOQ */
    ReadDataDSPI_1();            /* Read data on slave DSPI */
    ReadDataDSPI_0();            /* Read data on master DSPI */
    while (1) {i++; }            /* Wait forever */
}
25 FlexCAN Transmit and Receive

25.1 Description

Task: Initialize two FlexCAN modules for 100 kHz bit time based on an 8 MHz crystal. Send a message from FlexCAN A (FlexCAN 0 on MPC56xxB/P/S). The modules will be connected externally in an open drain circuit, before the transceiver. The CAN reference will be based on the crystal, not the PLL.

It is common for two CAN modules to be connected to the same transceiver. This allows more buffers to be available for a node. In this case, only one module is used for transmit and the other FlexCAN module’s transmit is not connected — otherwise the node might acknowledge itself.

However, because there are not any other nodes connected in this example, we want the second CAN module to respond to the first. The connections needed are shown below. The transmit outputs must be configured as open drain and use a pullup resistor. Note this extra resistance on the transmit line may slow rise and fall times, which could limit the CAN frequency.

Exercise: On a Evaluation Board (EVB), externally connect FlexCAN modules. EVBs typically have the pullup resistor. Be sure to use appropriate CAN_SEL jumpers on the EVB. Verify proper operation, then modify to send a second message.
### Table 96. MPC551x, MPC55xx Signals for FlexCAN Transmit and Receive Example

<table>
<thead>
<tr>
<th>Signal</th>
<th>Pin Name</th>
<th>SIU PCR No.</th>
<th>Package Pin No.</th>
<th>Function Name</th>
<th>SIU PCR No.</th>
<th>Package Pin No.</th>
<th>EVB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>144 QFP</td>
<td>176 QFP</td>
<td>208 BGA</td>
<td>496 BGA</td>
<td>416 BGA</td>
<td>324 BGA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>496 BGA</td>
<td></td>
<td>416 BGA</td>
<td>324 BGA</td>
<td>208 BGA</td>
<td>xPC 563M</td>
</tr>
<tr>
<td>CNTxA</td>
<td>PD0</td>
<td>48</td>
<td>104</td>
<td>128</td>
<td>D15</td>
<td>CNTXA</td>
<td>83</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>AF22</td>
<td>AD21</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Y17</td>
<td>P12</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PJ1–5</td>
<td></td>
</tr>
<tr>
<td>CNTxC</td>
<td>PD4</td>
<td>52</td>
<td>100</td>
<td>124</td>
<td>E</td>
<td>CNTXC</td>
<td>87</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>W24</td>
<td>V23</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>P19</td>
<td>K13</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PJ1–6</td>
<td></td>
</tr>
<tr>
<td>CNRxA</td>
<td>PD1</td>
<td>49</td>
<td>103</td>
<td>127</td>
<td>D16</td>
<td>CNRXA</td>
<td>84</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>AG22</td>
<td>AE22</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>AA18</td>
<td>R12</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PJ1–3</td>
<td></td>
</tr>
<tr>
<td>CNRxC</td>
<td>PD5</td>
<td>53</td>
<td>99</td>
<td>123</td>
<td>F13</td>
<td>CNRXC</td>
<td>88</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Y26</td>
<td>W24</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>R20</td>
<td>L14</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PJ1–4</td>
<td></td>
</tr>
</tbody>
</table>

### Table 97. MPC56xxB/P/S Signals for FlexCAN Transmit and Receive Example

(MPC56xxP: Safety Port is used for CAN1)

<table>
<thead>
<tr>
<th>Signal</th>
<th>MPC56xxB Family</th>
<th>MPC56xxP Family</th>
<th>MPC56xxS Family</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Port</td>
<td>SIU Pad</td>
<td>Port</td>
</tr>
<tr>
<td></td>
<td>Configuration</td>
<td>Configuration</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&amp; Selection</td>
<td>&amp; Selection</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Registers</td>
<td>Registers</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(values in hex)</td>
<td>(values in hex)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CAN0TX</td>
<td>B0</td>
<td>PCR16=0624</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CAN1TX</td>
<td>C10</td>
<td>PCR42=0624</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CAN0RX</td>
<td>B1</td>
<td>PCR17=0100</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CAN1RX</td>
<td>C3</td>
<td>PCR35=0100</td>
<td>77</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Connection Notes:

1. MPC56XX EVB main board: needs CAN_SEL jumpers, J27, connecting pins 1–3 and 2–4. In addition to connecting both Rx pins together and both Tx pins together, at least one FlexCAN needs to be connected to the transceiver. This allows (1) the CAN bus to sync by allowing the Rx to see an idle bus and (2) the receiving FlexCAN to transmit an acknowledge from its Tx pin, through the transceiver, which routes it to the transmitting FlexCAN’s Rx pin. CANH_SEL (J27) jumpers are connected as shown and CANL_SEL (J29) jumpers are not connected.

2. MPC56XX EVB main board: I/O power (J4) should be jumpered to 5 V as shown so transceiver pullup gets 5 V.

3. MPC56XX main board with XPC560S: Trimmer jumper (J40) on main board connects Port B0 to an potentiometer hooked to 5V. Disconnect this jumper to allow Port B0 to drive it’s output properly.
4. MPC56xxS EVB’s expansion card: make sure VDDE_B (J23) and VDDE_E (J25) are jumpered to 5V for the FlexCAN port pins used in this example. Transceiver’s need 5V on most EVBs.

5. MPC5510DEMO EVB needs CAN_EN jumpers (J9) connected by the CAN transceiver. Connect pins 1-2 and 3-4 to connect Tx and Rx to the transceiver as shown below.

6. MPC553x and MPC555x EVBs Tx and Rx pins need be connected to the CAN transceiver with jumpers. MPC5534EVB’s CAN_EN jumpers are shown below.
Debug Tips:

1. Observing a Repeating CAN Frame
   When debugging CAN, a simple test is only to connect Tx and Rx of one CAN together, and not to the transceiver. This direct connection allows the Rx pin to listen to the Tx without a transceiver, so that when a CAN frame starts to transmit then an error frame will not be generated. In addition, since there is no acknowledge from any receiving CAN module, the transmitting CAN module will keep repeating the transmitting of the frame, making it easy to examine on an oscilloscope or other tool.

   This example code uses an open drain on transmit pins, so if you are doing this test be sure either that an external pullup is connected on the board, or else disable the open drain in the transmitting CAN’s Pad Configuration Register (SIU_PCR).

2. Inadvertent Locking/Unlocking of Message Buffers
   Sometimes a message will not be received because a debugger window is reading the buffer’s Control and Status (CS), hence that buffer is locked and cannot receive a message.

   Conversely, a debugger window that constantly reads the TIMER or another message buffer may cause premature unlocking of a message buffer in your program.
25.2 Design

25.2.1 Timing Calculations

These common guidelines are used in this example:

- CAN bit rate period is typically subdivided into 12–20 time quanta units.
- The sample point is normally chosen around 75%–80% through the bit rate period.
- The remaining 20–25% will be the value for Phase_Seg2.
- The value of Phase_Seg1 will be the same as Phase_Seg2.
- The Sync_Seg is 1 time quanta.
- Resync Jump Width (RJW+1) = Phase_Seg2 (if Phase_Seg2 < 4; otherwise (RJW +1) = 4.

For this example and within the above guidelines, these are the values selected for CAN modules:

- number of time quanta units per bit rate period = 16
- Sample point = 75%, which is 12 time quanta units into the 16 time quanta period

Hence,

\[
\text{Phase}_\text{Seg}2 = (100\% - 75\%) \times 16 \text{ time quanta} = 25\% \times 16 \text{ time quanta} = 4 \text{ time quanta}; \text{PSEG}2 = 3 \\
\text{Phase}_\text{Seg}1 = \text{Phase}_\text{Seg}2 = 4 \text{ time quanta}; \text{PSEG}1 = 3 \\
\text{Prop}_\text{Seg} = 16 - \text{Phase}_\text{Seg}1 - \text{Phase}_\text{Seg}2 - \text{SYNCSEG} = 16 - 4 - 4 - 1 = 7; \text{PROPSEG} = 6 \\
\text{Resync Jump Width (RJW +1) = 4}
\]

Also for this example, the following applies for an 8 MHz crystal. (Note: If a 40 MHz crystal is used, the bit rate will increase five times.)

\[
\text{f}_{\text{CANCLK}} = 8 \text{ MHz (EVB oscillator)} \\
\text{Desired bit rate} = 100 \text{ kHz}
\]

Hence,

\[
\text{f}_{\text{iq}} (\text{time quanta frequency}) = (16 \text{ time quanta/bit rate period}) \times (100 \text{ K bit rate periods/sec}) = 1.6 \text{ MHz} \\
\text{Prescaler Value (PRESDIV +1) = f}_{\text{CANCLK}} / \text{f}_{\text{iq}} = 8 \text{ MHz} / 1.6 \text{ MHz} = 5 \\
\text{PRESDIV = 5} - 1 = 4
\]

<table>
<thead>
<tr>
<th>Number of Time Quanta</th>
<th>SYNCSEG</th>
<th>PROP_SEG</th>
<th>PHASE_SEG1</th>
<th>PHASE_SEG2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Bit Fields</td>
<td>1</td>
<td>7</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>1 bit time = (16 time quanta) \times (1 sec / 1.6M time quanta) = 10 \mu s</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Qorivva Simple Cookbook, Rev. 4
25.2.2 Message Buffers

The message buffer structure is shown below, which is from Freescale’s MPC5554 header files version 1.2. Remember that buffers are in FlexCAN RAM, so they are random in value on power-up. Hence all buffers must all have their CODE field set inactive before negating the halt state.

```c
struct canbuf_t {
    union {
        vuint32_t R;
        struct {
            vuint32_t:4;
            vuint32_t CODE:4;
            vuint32_t:1;
            vuint32_t SRR:1;
            vuint32_t IDE:1;
            vuint32_t RTR:1;
            vuint32_t LENGTH:4;
            vuint32_t TIMESTAMP:16;
        } B;
    } CS;

    union {
        vuint32_t R;
        struct {
            vuint32_t:3;
            vuint32_t STD_ID:11;
            vuint32_t EXT_ID:18;
        } B;
    } ID;

    union {
        vuint8_t B[8]; /* Data buffer in Bytes (8 bits) */
        vuint16_t H[4]; /* Data buffer in Half-words (16 bits) */
        vuint32_t W[2]; /* Data buffer in words (32 bits) */
        vuint32_t R[2]; /* Data buffer in words (32 bits) */
    } DATA;

} BUF[64];
```
25.2.3 Mode Use Summary (MPC56xxB/P/S only)

Mode Transition is required for changing mode entry registers. Hence even enabling the crystal oscillator to be active in the default mode (DRUN) requires enabling the crystal oscillator in appropriate mode configuration register (ME_xxxx_MC) then initiating a mode transition. This example transitions from the default mode after reset (DRUN) to RUN0 mode.

Table 99. Mode Configurations for MPC56xxB/P/S FlexCAN Transmit and Receive Example
 Modes are enabled in ME_ME Register.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Mode Config. Register</th>
<th>Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mode Config. Register Value</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sysclk Selection</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Clock Sources</td>
<td></td>
</tr>
<tr>
<td></td>
<td>16MHz IRC</td>
<td>XOSC0</td>
</tr>
<tr>
<td>DRUN</td>
<td>ME_DRUN_MC 0x001F 0010 (default)</td>
<td>16 MHz IRC</td>
</tr>
<tr>
<td>RUN0</td>
<td>ME_RUN0_MC 0x001F 007D</td>
<td>PLL0</td>
</tr>
</tbody>
</table>

Other modes are not used in example

Peripherals also have configurations to gate clocks on or off for different modes, enabling low power. The following table summarizes the peripheral configurations used in this example.

Table 100. Peripheral Configurations for MPC56xxB/P/S FlexCAN Transmit and Receive Example
 Low power modes are not used in example.

<table>
<thead>
<tr>
<th>Peripheral Config.</th>
<th>Peri. Config. Register</th>
<th>Enabled Modes</th>
<th>Peripherals Selecting Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RUN3 RUN2 RUN1 RUN0 DRUN SAFE TEST RESET</td>
<td>Peripheral PCTL Reg. #</td>
<td></td>
</tr>
<tr>
<td>PC1</td>
<td>ME_RUNPC_1 1</td>
<td>0 0 0 1 0 0 0 0</td>
<td>FlexCAN0 (MPC56xxB/P/S)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>FlexCAN1 (MPC56xxB/S)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SafetyPort (MPC56xxP use as FlexCAN)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SIUL (MPC56xxB/S)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>16 17 26 68</td>
</tr>
</tbody>
</table>

Other peripheral configurations are not used in example.
## 25.2.4 Steps and Pseudo Code

**Table 101. Steps and Pseudo Code Table for FlexCAN Transmit and Receive Example**

<table>
<thead>
<tr>
<th>Step</th>
<th>Relevant Bit Fields</th>
<th>Pseudo Code</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data Init</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Global variables for a CAN receive buffer</td>
<td>RxCODE byte, RxID word, RxLENGTH byte, RxDATA 8 byte string, RxTIMESTAMP word,</td>
<td></td>
</tr>
<tr>
<td><strong>init Modes and Clock (MPC 56xxPBS only)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enable desired modes</td>
<td>DRUN=1, RUN0 = 1</td>
<td>—</td>
</tr>
<tr>
<td>Initialize PLL0 dividers to provide 64 MHz for input crystal before mode configuration:</td>
<td>—</td>
<td>8 MHz Crystal: CGM_ FMPLL[0]_CR = 0x2400100</td>
</tr>
<tr>
<td>• 8 MHz xtal: FMPLL[0]_CR=0x02400100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• 40 MHz xtal: FMPLL[0]_CR=0x12400100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(MPC56xxP &amp; 8 MHz xtal requires changing default CMU_CSR value. See PLL example.)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Configure RUN0 Mode:</td>
<td>PDO=0, MVRON=1, DFLOAD, CFLAON= 3, PLL0ON=1, XOSCOON=1, 16MHz_IRCON=1, SYSCLK=0x4</td>
<td>—</td>
</tr>
<tr>
<td>• I/O Output power-down: no safe gating</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Main Voltage regulator is on (default)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Data, code flash in normal mode (default)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• PLL0 is switched on</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Crystal oscillator is switched on</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• 16 MHz IRC is switched on (default)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Select PLL0 (system pll) as sysclk</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPC56xxB/S:</td>
<td>RUN0=1</td>
<td>—</td>
</tr>
<tr>
<td>• Peri. Config.1: run in DRUN mode only</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Assign peripheral configuration to peripherals:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• FlexCAN 0: select ME_RUN_PC0</td>
<td>RUN_CFG = 1</td>
<td>—</td>
</tr>
<tr>
<td>• FlexCAN 1: select ME_RUN_PC0</td>
<td>RUN_CFG = 1</td>
<td></td>
</tr>
<tr>
<td>(56xxB/S)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Safety Port: select ME_RUN_PC0 (56xxP)</td>
<td>RUN_CFG = 1</td>
<td></td>
</tr>
<tr>
<td>• SIUL: select ME_RUN_PC0 (56xxB/S)</td>
<td>RUN_CFG = 1</td>
<td></td>
</tr>
<tr>
<td>Initiate software mode transition to RUN0 mode</td>
<td>TARGET_MODE = RUN0</td>
<td>-</td>
</tr>
<tr>
<td>• Mode &amp; key, then mode &amp; inverted key</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Wait for transition to complete</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Verify current mode is RUN0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>init Peri Clk Gen 56xxB/S</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Initialize peripheral clock generation (See appendix: MPC56xxB/P/S Peripheral Clocks)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• MPC56xxB/S CANs: Peri Set 2- sysclk div. 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• MPC56xxP Safety Port: Aux Clk 2- PLL0 div. 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 101. Steps and Pseudo Code Table for FlexCAN Transmit and Receive Example (continued)

<table>
<thead>
<tr>
<th>Step</th>
<th>Relevant Bit Fields</th>
<th>Pseudo Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disable Watchdog</td>
<td>Disable watchdog by writing keys to Status Reg, then clearing WEN (MPC56xxBPS only)</td>
<td>—</td>
</tr>
<tr>
<td>Put FlexCAN module in freeze mode</td>
<td></td>
<td>HALT = 1, FRZ = 1</td>
</tr>
<tr>
<td>When in freeze mode, enable all 64 buffers (32 message buffers for MPC56xxP)</td>
<td>MAXMB = 63 or 31</td>
<td>CANC_MCR = 0x5000 003F</td>
</tr>
<tr>
<td>(or FlexCAN 1 or Safety Port)</td>
<td></td>
<td>CAN1_MCR = 0x5000 003F</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MPC56xxP only: 1F</td>
</tr>
<tr>
<td>Configure bit timing parameters, bit rate, arbitration (same as FlexCAN C or FlexCAN 1)</td>
<td>See other CAN</td>
<td>CANC_CR = 0x04DB 0006</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CAN1_CR = 0x04DB 0006</td>
</tr>
<tr>
<td>Inactivate 32 or 64 Message Buffers, Initialize Message Buffer 4 for receive:</td>
<td>CODE = b0000</td>
<td>CANC_MB0:63[CODE] = 0</td>
</tr>
<tr>
<td>• Extended ID is not used</td>
<td>IDE = 0</td>
<td>CANC_MB4[CODE] = 4</td>
</tr>
<tr>
<td>• Desired std. ID for incoming message is 555 (0x22B)</td>
<td>std. ID = 555</td>
<td>CANC_MB4[ID] = 555 &lt;&lt;18</td>
</tr>
<tr>
<td>• Set MB as RX EMPTY</td>
<td>CODE = b0100</td>
<td>CANC_MB4[CODE] = 4</td>
</tr>
<tr>
<td>Initialize global acceptance mask: exact ID matches on all message buffers except 14, 15</td>
<td>MI = 0x1FFFF FFFF</td>
<td>CANC_RXGMASK = 0x1FFFFFFF</td>
</tr>
<tr>
<td>Initialize interrupt mask bits as needed</td>
<td></td>
<td>CAN1_RXGMASK = 0x1FFFFFFF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MPC56xxB/P/S Signals</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See table:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MPC56xxB/P/S</td>
</tr>
<tr>
<td></td>
<td></td>
<td>—</td>
</tr>
<tr>
<td>Configure pad as CNTXC, open drain, max slew rate</td>
<td>PA = 3, OBE = 1, SRC = 3</td>
<td>SIU_PCR[52] = 0x062C</td>
</tr>
<tr>
<td></td>
<td>Configure pad as CNRXC</td>
<td>PA = 3, IBE = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SIU_PCR[87] = 0x0E2C</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SIU_PCR[88] = 0xD03</td>
</tr>
<tr>
<td>Negate HALT state, while enabling the maximum number of msg. buffers</td>
<td>HALT = 0, FRZ = 0</td>
<td>CANC_MCR = 0x0000 003F</td>
</tr>
<tr>
<td></td>
<td>MAXMB = 63 or 31</td>
<td>CAN1_MCR = 0x0000 003F or 1F</td>
</tr>
</tbody>
</table>
### Table 101. Steps and Pseudo Code Table for FlexCAN Transmit and Receive Example (continued)

<table>
<thead>
<tr>
<th>Step</th>
<th>Relevant Bit Fields</th>
<th>Pseudo Code</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init FlexCAN</strong>&lt;br&gt;(or FlexCAN0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Put FlexCAN module in freeze mode</td>
<td>HALT=1, FRZ=1</td>
<td>CANA_MCR = 0x5000 003F</td>
</tr>
<tr>
<td>When in freeze mode, enable all 64 buffers (32 message buffers for MPC56xxP)</td>
<td>MAXMB = 63 (31 for MPC56xxP)</td>
<td>CAN0_MCR = 0x5000 003F&lt;br&gt;MPC56xxP only: 0x5000 001F&lt;br&gt;CANA_MCR = 0x5000 003F &lt;br&gt;CAN0_MCR = 0x5000 001F</td>
</tr>
<tr>
<td>Configure bit timing parameters</td>
<td>PROPSEG = 6&lt;br&gt;PSEG1 = 3&lt;br&gt;PSEG2 = 3&lt;br&gt;RJW = 3</td>
<td>CANA_CR = 0x04DB 0006</td>
</tr>
<tr>
<td>Configure bit rate for 8 MHz OSC, 100 kHz bit rate, and 16 time quanta</td>
<td>CLK_SRC = 0&lt;br&gt;PRESDIV = 4</td>
<td>CAN0_CR = 0x04DB 0006</td>
</tr>
<tr>
<td>Configure internal arbitration, others to default</td>
<td>LBUF = 0</td>
<td></td>
</tr>
<tr>
<td>Inactivate all 64 message buffers (reduce to 32 message buffers for MPC56xxP)</td>
<td>CODE = b0000&lt;br&gt;CANA_MB0[CODE] = 0&lt;br&gt;CAN0_MB0[CODE] = 8</td>
<td></td>
</tr>
<tr>
<td>Message Buffer 0: set as TX INACTIVE</td>
<td>CODE = b1000&lt;br&gt;CANA_MB0[CODE] = 8&lt;br&gt;CAN0_MB0[CODE] = 8</td>
<td></td>
</tr>
<tr>
<td>Initialize acceptance masks as needed</td>
<td>(none used here)</td>
<td>–</td>
</tr>
<tr>
<td>Initialize interrupt mask bits as needed</td>
<td>(none used here)</td>
<td>–</td>
</tr>
<tr>
<td>Configure pad as CNTXA, open drain, max slew rate</td>
<td>PA = 1, OBE = 1, SRC = 3&lt;br&gt;PA = 1, OBE = 1</td>
<td>CANA_MCR = 0x0000 003F&lt;br&gt;CAN0_MCR = 0x0000 003F or 1F</td>
</tr>
<tr>
<td>Configure pad as CNRXA</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>Negate HALT state, while enabling the maximum number of message buffers</td>
<td>HALT=0, FRZ=0&lt;br&gt;MPC551x&lt;br&gt;MPC555x&lt;br&gt;MPC56xxB/P/S</td>
<td>CANA_MCR = 0x0000 003F&lt;br&gt;CAN0_MCR = 0x0000 003F or 1F</td>
</tr>
</tbody>
</table>

### Transmit Message <br>(Assume CANA MB 0 is inactive)<br>Use standard ID, not extended ID<br>Standard ID = 555 (0x22B) | IDE = 0<br>std. ID = 555 | CANA_MBO[IDE] = 0<br>CAN0_MBO[IDE] = 0<br>MPC56xxB/P/S | CANA_MBO[IDE] = 0<br>CAN0_MBO[IDE] = 0<br>MPC56xxB/P/S |
<p>| Frame is a data frame, not remote Tx request&lt;br&gt;Length of data is five bytes | RTR = 0&lt;br&gt;LENGTH = 5 | CANA_MBO[RTR] = 0&lt;br&gt;CAN0_MBO[RTR] = 0&lt;br&gt;MPC56xxB/P/S | CANA_MBO[RTR] = 0&lt;br&gt;CAN0_MBO[RTR] = 0&lt;br&gt;MPC56xxB/P/S |
| Data is string “Hello” | DATA = ‘Hello’ | CANA_MBO[DATA] = ‘Hello’&lt;br&gt;CAN0_MBO[DATA] = ‘Hello’&lt;br&gt;MPC56xxB/P/S | CANA_MBO[DATA] = ‘Hello’&lt;br&gt;CAN0_MBO[DATA] = ‘Hello’&lt;br&gt;MPC56xxB/P/S |
| SRR = 1 for transmit; only req’d in exd’d frames | SRR = 1 | CANA_MBO[SRR] = 1&lt;br&gt;CAN0_MBO[SRR] = 1&lt;br&gt;MPC56xxB/P/S | CANA_MBO[SRR] = 1&lt;br&gt;CAN0_MBO[SRR] = 1&lt;br&gt;MPC56xxB/P/S |
| Activate MB to transmit a data frame | CODE = b1100 | CANA_MBO[CODE] = 0xC&lt;br&gt;CAN0_MBO[CODE] = 0xC&lt;br&gt;MPC56xxB/P/S | CANA_MBO[CODE] = 0xC&lt;br&gt;CAN0_MBO[CODE] = 0xC&lt;br&gt;MPC56xxB/P/S |</p>
<table>
<thead>
<tr>
<th>Step</th>
<th>Relevant Bit Fields</th>
<th>Pseudo Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Receive Message</td>
<td>wait for BUF04I = 1</td>
<td>wait for CANC_IFLAG1 [BUF04I] = 1</td>
</tr>
<tr>
<td>Read CODE (activates internal buffer lock)</td>
<td>CODE will be b0010</td>
<td>RxCODE = CANC_MB4[CODE] RxID = CANC_MB4[ID] RxDATA[ ] = CANC_MB4[DATA] RxTIMESTAMP = CANC_MB4 [TIMESTAMP]</td>
</tr>
<tr>
<td>Read ID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read DATA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read TIMESTAMP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read TIMER to unlock buffer</td>
<td>CODE = b0100</td>
<td>Read CANC TIMER</td>
</tr>
<tr>
<td>Clear CANC MB 4 flag by writing a 1 to it</td>
<td>BUF04I = 1</td>
<td>CANC_IFLAG1 = 0x0000 0010</td>
</tr>
</tbody>
</table>
25.2.5 Design Results

The pictures below show results for this design based on an 8 MHz crystal.

- Oscilloscope screenshot of the start of the transmit output line, changing at the 100 kHz bit time rate
- CAN tool screenshot of the transmitted message as seen on the CAN bus
25.3 Code

25.3.1 MPC551x, MPC555x

/* main.c: FlexCAN program */
/* Description: Transmit one message from FlexCAN A buf. 0 to FlexCAN C buf. 4 */
/* Rev 0.1 Jan 16, 2006 S.Mihalik, Copyright Freescale, 2006. All Rights Reserved */
/* Rev 0.2 Jun 6 2006 SM - changed Flexcan A to C & enabled 64 msg buffers */
/* Rev 0.3 Jun 15 2006 SM - 1. Made globals uninitialized */
/* 2. receiveMsg function: read CANx TIMER, removed setting buffer's CODE*/
/* 3. added idle loop code for smoother Nexus trace */
/* 4. modified for newer Freescale header files (r 16) */
/* Rev 0.4 Aug 11 2006 SM - Removed redundant CAN A.MCR init */
/* Rev 0.5 Jan 31 2007 SM - Removed other redundant CAN C.MCR init */
/* Rev 0.6 Mar 08 2007 SM - Corrected init of MBs: cleared 64 MBs, instead of 63 */
/* Rev 0.7 Jul 20 2007 SM - Changes for MPC5510 */
/* Rev 0.8 May 15 2008 SM - Changes for new MPC5510 header file symbols */
/* Rev 1.0 Jul 10 2009 SM - Cleared CAN Msg Buf flag by writing to reg. not bit, */
/* increased Tx pads slew rate, changed RxCODE, RxLENGTH, dummy data types and */
/* init receiving CAN C first to allow CAN bus sync time before receiving first msg*/
/* Notes: */
/* 1. MMU not initialized; must be done by debug scripts or BAM */
/* 2. SRAM not initialized; must be done by debug scripts or in a crt0 type file */
#include "mpc563m.h" /* Use proper include file such as mpc5510.h or mpc5554.h */

uint32_t RxCODE;              /* Received message buffer code */
uint32_t RxID;                 /* Received message ID */
uint32_t RxLENGTH;            /* Received message number of data bytes */
uint8_t RxDATA[8];            /* Received message data string*/
uint32_t RxTIMESTAMP;          /* Received message time */

void initCAN_A (void) {
    uint8_t i;
    CAN_A.MCR.R = 0x5000003F;       /* Put in Freeze Mode & enable all 64 msg buffers*/
    CAN_A.CR.R = 0x04DB0006;        /* Configure for 8MHz OSC, 100kHz bit time */
    for(i=0; i<64; i++) {
        CAN_A.BUF[i].CS.B.CODE = 0;   /* Inactivate all message buffers */
    }
    CAN_A.BUF[0].CS.B.CODE = 8;     /* Message Buffer 0 set to TX INACTIVE */
    /* Use 1 pair of the next four lines of code for MPC551x or MPC555x */
    SIU.PCR[48].R = 0x062C;         /* MPC551x: Configure pad as CNTXA, open drain */
    SIU.PCR[49].R = 0x0500;         /* MPC551x: Configure pad as CNRXC */
    SIU.PCR[83].R = 0x062C;         /* MPC555x: Configure pad as CNTXA, open drain */
    SIU.PCR[84].R = 0x0500;         /* MPC555x: Configure pad as CNRXC */
    CAN_A.MCR.R = 0x0000003F;       /* Negate FlexCAN A halt state for 64 MB */
}

void initCAN_C (void) {
    uint8_t i;
    CAN_C.MCR.R = 0x5000003F;       /* Put in Freeze Mode & enable all 64 msg buffers*/
    CAN_C.CR.R = 0x04DB0006;        /* Configure for 8MHz OSC, 100kHz bit time */
    for(i=0; i<64; i++) {
        CAN_C.BUF[i].CS.B.CODE = 0;   /* Inactivate all message buffers */
    }
    CAN_C.BUF[4].CS.B.IDE = 0;      /* MB 4 will look for a standard ID */
    CAN_C.BUF[4].ID.B.STD_ID = 555; /* MB 4 will look for ID = 555 */
    CAN_C.BUF[4].CS.B.CODE = 4;     /* MB 4 set to RX EMPTY */
    CAN_C.RXGMASK.R = 0x1FFFFFFF;   /* Global acceptance mask */
    /* Use 1 pair of the next four lines of code for MPC551x or MPC555x */
    SIU.PCR[52].R = 0x062C;         /* MPC551x: Configure pad as CNTXC, open drain */
    SIU.PCR[53].R = 0x0500;         /* MPC551x: Configure pad as CNRXC */
    SIU.PCR[87].R = 0x0E2C;         /* MPC555x: Configure pad as CNTXC, open drain */
    SIU.PCR[88].R = 0x0D00;         /* MPC555x: Configure pad as CNRXC */
    CAN_C.MCR.R = 0x0000003F;       /* Negate FlexCAN C halt state for 64 MB */
}
void TransmitMsg (void) {
    void TransmitMsg (void) {
        uint8_t i;
        /* Assumption: Message buffer CODE is INACTIVE */
        const uint8_t TxData[] = {"Hello"); /* Transmit string*/
        CAN_A.BUF[0].CS.B.IDE = 0; /* Use standard ID length */
        CAN_A.BUF[0].ID.B.STD_ID = 555; /* Transmit ID is 555 */
        CAN_A.BUF[0].CS.B.RTR = 0; /* Data frame, not remote Tx request frame */
        CAN_A.BUF[0].CS.B.LENGTH = sizeof(TxData) - 1; /* # bytes to transmit w/o null */
        for (i=0; i<sizeof(TxData); i++) {
            CAN_A.BUF[0].DATA.B[i] = TxData[i]; /* Data to be transmitted */
        }
        CAN_A.BUF[0].CS.B.SRR = 1; /* Tx frame (not req'd for standard frame)*/
        CAN_A.BUF[0].CS.B.CODE = 0xC; /* Activate msg. buf. to transmit data frame */
    }

    void receiveMsg (void) {
        void receiveMsg (void) {
            uint8_t j;
            uint32_t dummy;
            /* Use 1 of the next 2 lines:
             */
            /* Use 1 of the next 2 lines:
             */
            /*while (CAN_C.IFLAG1.B.BUF04I == 0) {}; /* MPC551x: Wait for CAN C MB 4 flag */
            while (CAN_C.IFFRL.B.BUF04I == 0) {}; /* MPC555x: Wait for CAN C MB 4 flag */
            RxCODE = CAN_C.BUF[4].CS.B.CODE; /* Read CODE, ID, LENGTH, DATA, TIMESTAMP*/
            RxID = CAN_C.BUF[4].ID.B.STD_ID;
            RxLENGTH = CAN_C.BUF[4].CS.B.LENGTH;
            for (j=0; j<RxLENGTH; j++) {
                RxDATA[j] = CAN_C.BUF[4].DATA.B[j];
            }
            RxTIMESTAMP = CAN_C.BUF[4].CS.B.TIMESTAMP;
            dummy = CAN_C.TIMER.R; /* Read TIMER to unlock message buffers */
            /* Use 1 of the next 2 lines: */
            /* Use 1 of the next 2 lines: */
            /*while (CAN_C.IFLAG1.R = 0x00000010); /* MPC551x: Clear CAN C MB 4 flag */
            /*while (CAN_C.IFFRL.R = 0x00000010); /* MPC555x: Clear CAN C MB 4 flag */
        }

    void main(void) {
        void main(void) {
            volatile uint32_t IdleCtr = 0;
            initCAN_C(); /* Initialize FLEXCAN C & one of its buffers for receive*/
            initCAN_A(); /* Initialize FlexCAN A & one of its buffers for transmit*/
            TransmitMsg(); /* Transmit one message from a FlexCAN A buffer */
            receiveMsg(); /* Wait for the message to be received at FlexCAN C */
            while (1) {
                /* Idle loop: increment counter */
                IdleCtr++;
            }
        }
    }
}
25.3.2  MPC56xxB/S (MPC56xxB shown)

/* main.c: FlexCAN program */
/* Description: Transmit one message from FlexCAN 0 buf. 0 to FlexCAN C buf. 1 */
/* Rev 0.1 Jan 16, 2006 S.Mihalik, Copyright Freescale, 2006. All Rights Reserved */
/* Rev 0.2 Jun 6 2006 SM - changed Flexcan A to C & enabled 64 msg buffers */
/* Rev 0.3 Jul 15 2006 SM - 1. Made globals uninitialized */
/* 2. ReceiveMsg function: read CANx TIMER, removed setting buffer's CODE*/
/* 3. added idle loop code for smoother Nexus trace */
/* 4. modified for newer Freescale header files (r 16) */
/* Rev 0.4 Aug 11 2006 SM - Removed redundant CAN A.MCR init */
/* Rev 0.5 Jan 16 2006 SM - Removed other redundant CAN C.MCR init */
/* Rev 0.6 Mar 08 2007 SM - Corrected init of MBs- cleared 64 MBs, instead of 63 */
/* Rev 0.7 Jul 20 2007 SM - Changes for MPC5510 */
/* Rev 0.8 May 15 2008 SM - Changes for new header file symbols */
/* Rev 0.9 May 29 2009 SM - Changes for MPC56xxB/P/S */
/* Rev 1.0 Jul 10 2009 SM - Simplified, cleared CAN Msg Buf flag by writing to reg */
/* not bit, increased Tx pads slew rate, chgd RxCODE, RxLENGTH, dummy data types */
/* 6 init receiving CAN first to allow CAN bus sync time before receiving 1st msg */
/* Rev 1.1 Mar 14 2010 SM - modified initModesAndClock, updated header file */
/* NOTE!! structure canbuf_t in jdp.h header file modified to allow byte addressing */
#include "MPC5604B_0M27V_0T02.h"  /* Use proper include file */

uint32_t  RxCODE;              /* Received message buffer code */
uint32_t RxID;                 /* Received message ID */
uint32_t RxLENGTH;             /* Recieved message number of data bytes */
uint8_t  RxDATA[8];            /* Received message data string*/
uint32_t RxTIMESTAMP;          /* Received message time */

void initModesAndClks(void) {
    ME.MER.R = 0x0000001D;          /* Enable DRUN, RUN0, SAFE, RESET modes */
    /* Initialize PLL before turning it on: */
    /* Use 1 of the next 2 lines depending on crystal frequency: */
    CGM.FMPLL.CR.R = 0x02400100;    /* 8 MHz xtal: Set PLL0 to 64 MHz */
    CGM.FMPLL.CR.R = 0x12400100;    /* 40 MHz xtal: Set PLL0 to 64 MHz */
    ME.RUN[0].R = 0x0001F0074;      /* RUN0 cfg: 16MHzIRCON,OSC0ON,PLL0ON,syclk=PLL */
    ME.RUNPC[1].R = 0x000000010;    /* Perl. Cfg. 1 settings: only run in RUN0 mode */
    ME.PCTL[16].R = 0x00;           /* MPC56xxB/P/S FlexCAN0: select ME.RUNPC[1] */
    ME.PCTL[17].R = 0x00;           /* MPC56xxB/S FlexCAN1: select ME.RUNPC[1] */
    ME.PCTL[68].R = 0x00;           /* MPC56xxB/S SIUL: select ME.RUNPC[1] */
    ME.MCTL.R = 0x40005AF0;         /* Enter RUN0 Mode & Key */
    ME.MCTL.R = 0x4000A50F;         /* Enter RUN0 Mode & Inverted Key */
    while (ME.GS.B.S_MTRANS) {}    /* Wait for mode transition to complete */
    while (ME.GS.B.S_CURRENTMODE != 4) {}  /* Verify RUN0 is the current mode */
}

void initPeriClkGen(void) {
    CGM.SC_DC[1].R = 0x880;        /* MPC56xxB/S: Enable peri set 2 sysclk divided by 1 */
}

void disableWatchdog(void) {
    SWT.SR.R = 0x00000c520;       /* Write keys to clear soft lock bit */
    SWT.SR.R = 0x00000d928;       /* Enter RUN0 Mode & Key */
    SWT.CR.R = 0x8000010A;        /* Clear watchdog enable (WEN) */
}

void initCAN_1 (void) {
    uint8_t  I;
    CAN_1.MCR.R = 0x50000003F;    /* Put in Freeze Mode & enable all 64 msg bufs */
    CAN_1.CR.R = 0x04DB0006;       /* Configure for 8MHz OSC, 100kHz bit time */
    for (I=0; I<64; I++) {
        CAN_1.BUF[I].CS.B.CODE = 0;  /* Inactivate all message buffers */
    }
    CAN_1.BUF[4].CS.B.IDE = 0;     /* MB 4 will look for a standard ID */
    CAN_1.BUF[4].ID.B.STD_ID = 555; /* MB 4 will look for ID = 555 */
    CAN_1.BUF[4].CS.B.CODE = 4;   /* MB 4 set to RX EMPTY */

Qorivva Simple Cookbook, Rev. 4
void initCAN_0 (void) {
  uint8_t i;

  CAN_0.MCR.R = 0x5000003F;     /* Put in Freeze Mode & enable all 64 msg bufs */
  CAN_0.CR.R = 0x04DB0006;      /* Configure for 8MHz OSC, 100kHz bit time */
  for (i=0; i<64; i++) {
    CAN_0.BUF[i].CS.B.CODE = 0;  /* Inactivate all message buffers */
  }
  CAN_0.BUF[0].CS.B.CODE = 8;   /* Message Buffer 0 set to TX INACTIVE */
  SIU.PCR[16].R = 0x0024;       /* MPC56xxB: Config port B0 as CAN0TX, open drain */
  SIU.PCR[17].R = 0x0100;       /* MPC56xxB: Configure port B1 as CAN0RX */
  CAN_0.MCR.R = 0x0000003F;     /* Negate FlexCAN 0 halt state for 64 MB */
}

void TransmitMsg (void) {
  uint8_t i;

  const uint8_t TxData[] = {"Hello"};  /* Transmit string*/
  CAN_0.BUF[0].CS.B.IDE = 0;           /* Use standard ID length */
  CAN_0.BUF[0].ID.B.STD_ID = 555;      /* Transmit ID is 555 */
  CAN_0.BUF[0].CS.B.RTR = 0;           /* Data frame, not remote Tx request frame */
  CAN_0.BUF[0].CS.B.LENGTH = sizeof(TxData) -1; /* # bytes to transmit w/o null */
  for (i=0; i<sizeof(TxData); i++) {
    CAN_0.BUF[0].DATA.B[i] = TxData[i];  /* Data to be transmitted */
  }
  CAN_0.BUF[0].CS.B.SRR = 1;           /* Tx frame (not req'd for standard frame)*/
  CAN_0.BUF[0].CS.B.CODE =0xC;         /* Activate msg. buf. to transmit data frame */
}

void RecieveMsg (void) {
  uint8_t j;
  uint32_t dummy;

  while (CAN_1.IFRL.B.BUF04I == 0) {};  /* Wait for CAN 1 MB 4 flag */
  RxCODE = CAN_1.BUF[4].CS.B.CODE;    /* Read CODE, ID, LENGTH, DATA, TIMESTAMP */
  RxID = CAN_1.BUF[4].ID.B.STD_ID;
  RxLENGTH = CAN_1.BUF[4].CS.B.LENGTH;
  for (j=0; j<RxLENGTH; j++) {
    RxDATA[j] = CAN_1.BUF[4].DATA.B[j];
  }
  RxTIMESTAMP = CAN_1.BUF[4].CS.B.TIMESTAMP;
  dummy = CAN_1.TIMER.R;               /* Read TIMER to unlock message buffers */
  CAN_1.IFRL.R = 0x00000010;          /* Clear CAN 1 MB 4 flag */
}

void main(void) {
  volatile uint32_t IdleCtr = 0;

  initModesAndClks();               /* Initialize mode entries */
  initPeriClkGen();                 /* Initialize peripheral clock generation for DSPIs */
  disableWatchdog();                /* Disable watchdog */
  initCAN_1();                      /* Initialize FLEXCAN 1 & one of its buffers for receive*/
  initCAN_0();                      /* Initialize FlexCAN 0 & one of its buffers for transmit*/
  TransmitMsg();                    /* Transmit one message from a FlexCAN 0 buffer */
  RecieveMsg();                     /* Wait for the message to be recieved at FlexCAN 1 */
  while (1) {
    IdleCtr++;
  }
}
25.3.3 MPC56xxP

/* main.c: FlexCAN program */
/* Description: Transmit one message from FlexCAN 0 buf. 0 */
/* Rev 0.1 Jan 16, 2006 S.Mihalik, Copyright Freescale, 2006. All Rights Reserved */
/* Rev 0.2 Jun 6 2006 SM - changed Flexcan A to C & enabled 64 msg buffers */
/* Rev 0.3 Jun 15 2006 SM - 1. Made globals uninitialized */
/* 2. ReceiveMsg function: read CANx TIMER, removed setting buffer's CODE */
/* 3. added idle loop code for smoother Nexus trace */
/* 4. modified for newer Freescale header files (r 16) */
/* Rev 0.4 Aug 11 2006 SM - Removed redundant CAN A.MCR init */
/* Rev 0.5 Jan 31 2007 SM - Removed other redundant CAN C.MCR init */
/* Rev 0.6 Mar 08 2007 SM - Corrected init of MBs- cleaFed MBs 64 MBs, instead of 63 */
/* Rev 0.7 Jul 20 2007 SM - Changes for MPC5510 */
/* Rev 0.8 May 15 2008 SM - Changes for new header file symbols */
/* Rev 0.9 May 29 2009 SM - Changes for MPC56xxP, simplified. */
/* Cleared CAN Msg Buf flag by writing to reg. not bit, */
/* increased Tx pads slew rate, changed RxCODE, RxLENGTH, dummy data types and */
/* init receiving CAN first to allow CAN bus sync time before receiving first msg */
/* Rev 1.0 Mar 14 2009 SM - Modified initModesAndClks, updated header */
/* NOTE!! structure canbuf_t DATA in header file modified to allow byte addressing*/
#include "Pictus_Header_v1_09.h" /* Use proper include file */

#include "Pictus_Header_v1_09.h" /* Use proper include file */

void initModesAndClks(void) {
    ME.MER.R = 0x0000001D;  /* Enable DRUN, RUN0, SAFE, RESET modes */
    /* Initialize PLL before turning it on: */
    /* Use 2 of the next 4 lines depending on crystal frequency: */
    /*CGM.CMU_0_CSR.R = 0x0000000004; /* Monitor FXOSC > FIRC/4 (4MHz); no PLL monitor*/
    /*CGM.FMPLL[0].CR.R = 0x02400100; /* 8 MHz xtal: Set PLL0 to 64 MHz */
    /*CGM.FMPLL[0].CR.R = 0x012400100; /* 40 MHz xtal: Set PLL0 to 64 MHz */
    ME.RUN[0].R = 0x001F0074;    /* RUN0 cfg: 16MHzIRCON,OSC0ON,PLL0ON,syclk=PLL */
    ME.RUNPC[1].R = 0x00000010;  /* Peri. Cfg. 1 settings: only run in RUN0 mode */
    ME.PCTL[16].R = 0x01;       /* MPC56xxB/P/S FlexCAN0: select ME.RUNPC[1] */
    ME.PCTL[26].R = 0x01;       /* MPC56xxP SafetyPort: select ME.RUNPC[1] */
    ME.MCTL.R = 0x40005AF0;    /* Mode Transition to enter RUN0 mode: */
    ME.MCTL.R = 0x4000A50F;    /* Enter RUN0 Mode & Key */
    while (ME.GS.B.S_MTRANS) {}  /* Wait for mode transition to complete */
    /* Note: could wait here using timer and/or I_TC IRQ */
    while (ME.GS.B.S_CURRENTMODE != 4) {}  /* Verify RUN0 is the current mode*/
}

void initPeriClkGen(void) {
    CGM.AC2SC.R = 0x04000000;  /* MPC56xxP Safety Port: Select PLL0 for aux clk 0 */
    CGM.AC2DC.R = 0x80000000;  /* MPC56xxP Safety Port: Enable aux clk 0 div by 1 */
}

void disableWatchdog(void) {
    SWT.SR.R = 0x00005c20;    /* Write keys to clear soft lock bit */
    SWT.SR.R = 0x0000928;     /* Enter RUN0 Mode & Key */
    SWT.CR.R = 0x8000010A;    /* Clear watchdog enable (WEN) */
}

void initSAFEPORT (void) {
    uint8_t i;
    SAFEPORT.MCR.R = 0x5000001F; /* Put in Freeze Mode & enable all 32 msg bufs */
    /* Use 1 of the next 2 lines depending on crystal frequency: */
    /*SAFEPORT.CR.R = 0x04DB0006; *//* Configure for 8MHz OSC, 100kHz bit time */
    SAFEPORT.CR.R = 0x18DB0006;  /* Configure for 40MHz OSC, 100kHz bit time */
    for (i=0; i<32; i++) {
        SAFEPORT.BUF[i].CS.B.CODE = 0; /* Inactivate all message buffers */
    }
}

Qorivva Simple Cookbook, Rev. 4
Freescale Semiconductor 217
SAFEPORT.BUF[4].CS.B.IDE = 0; /* MB 4 will look for a standard ID */
SAFEPORT.BUF[4].ID.B.STD_ID = 555; /* MB 4 will look for ID = 555 */
SAFEPORT.BUF[4].CS.B.CODE = 4; /* MB 4 set to RX EMPTY */
SAFEPORT.RXGMASK.R = 0x1FFFPFPF; /* Global acceptance mask */
SIU.PCR[14].R = 0x0624; /* MPC56xxxP: Config port A14 as CAN1TX, open drain */
SIU.PCR[15].R = 0x0900; /* MPC56xxxP: Config port A15 as CAN1RX */
SAFEPORT.MCR.R = 0x0000001F; /* Negate SAFETY PORT halt state for 32 MB */

void initCAN_0 (void) {
    uint8_t i;
    CAN_0.MCR.R = 0x5000001F; /* Put in Freeze Mode & enable all 32 msg bufs */
    /* Use 1 of the next 2 lines depending on crystal frequency: */
    /* CAN_0.CR.R = 0x04DB0006; */ /* Configure for 8MHz OSC, 100kHz bit time */
    CAN_0.CR.R = 0x18DB0006; /* Configure for 40MHz OSC, 100kHz bit time */
    for(i=0; i<32; i++) {
        CAN_0.BUF[i].CS.B.CODE = 0; /* Inactivate all message buffers */
    }
    CAN_0.BUF[0].CS.B.CODE = 8; /* Message Buffer 0 set to TX INACTIVE */
    SIU.PCR[16].R = 0x0624; /* MPC56xxxP: Config port B0 as CAN0TX, open drain */
    SIU.PCR[17].R = 0x0500; /* MPC56xxxP: Configure port B1 as CAN0RX */
    CAN_0.MCR.R = 0x0000001F; /* Negate FlexCAN 0 halt state for 21 MB */
}

void TransmitMsg (void) {
    uint8_t i;
    /* Assumption: Message buffer CODE is INACTIVE */
    const uint8_t TxData[] = {"Hello"}; /* Transmit string */
    CAN_0.BUF[0].CS.B.IDE = 0; /* Use standard ID length */
    CAN_0.BUF[0].ID.B.STD_ID = 555; /* Transmit ID is 555 */
    CAN_0.BUF[0].CS.B.RTR = 0; /* Data frame, not remote Tx request frame */
    CAN_0.BUF[0].CS.B.LENGTH = sizeof(TxData) - 1; /* # bytes to transmit w/o null */
    for(i=0; i<sizeof(TxData); i++) {
        CAN_0.BUF[0].DATA.B[i] = TxData[i]; /* Data to be transmitted */
    }
    CAN_0.BUF[0].CS.B.SRR = 1; /* Tx frame (not req'd for standard frame) */
    CAN_0.BUF[0].CS.B.CODE = 0xC; /* Activate msg. buf. to transmit data frame */
}

void RecieveMsg (void) {
    uint8_t j;
    uint32_t dummy;
    while (SAFEPORT.IFRL.B.BUF04I == 0) {} /* Wait for SAFETY PORT MB 4 flag */
    RxCODE = SAFEPORT.BUF[4].CS.B.CODE; /* Read CODE, ID, LENGTH, DATA, TIMESTAMP */
    RxID = SAFEPORT.BUF[4].ID.B.STD_ID;
    RxLENGTH = SAFEPORT.BUF[4].CS.B.LENGTH;
    for (j=0; j<RxLENGTH; j++) {
        RxDATA[j] = SAFEPORT.BUF[4].DATA.B[j]; /* Data to be transmitted */
    }
    RxTIMESTAMP = SAFEPORT.BUF[4].CS.B.TIMESTAMP;
    dummy = SAFEPORT.TIMER.R; /* Read TIMER to unlock message buffers */
    SAFEPORT.IFRL.R = 0x00000010; /* Clear SAFETY PORT MB 4 flag */
}

void main(void) {
    volatile uint32_t IdleCtr = 0;
    initModesAndClks(); /* Initialize mode entries */
    initPerIclkGen(); /* Initialize peripheral clock generation for DSPIs */
    disableWatchdog(); /* Disable watchdog */
    initSAFEPORT(); /* Initialize SafetyPort & one of its buffers for receive*/
    initCAN_0(); /* Initialize FlexCAN 0 & one of its buffers for transmit*/
    TransmitMsg(); /* Transmit one message from a FlexCAN 0 buffer */
    RecieveMsg(); /* Wait for the message to be recieved at FlexCAN 1 */
    while (1) { /* Idle loop: increment counter */
        IdleCtr++;
    }
}
26 Flash: Configuration

26.1 Description

Task: Configure the flash performance parameters for a 64 MHz system clock and increase performance for branch instructions by enabling branch target buffers and branch prediction.

Key elements of the flash module include the flash array and its line buffers. Line buffers allow overlapping fast access between the crossbar and a flash module line buffer with a slower access between a line buffer and the flash array. Line buffer bus interface width is 32 or 64 bits, but the actual buffer width is the width of the flash array: either 128 or 256 bits. Hence one line buffer fill from the array will provide multiple bus interface transfers to, for example, the core. Parameters affecting transfers between the flash array and the line buffers, such as read wait states, are not optimal out of reset and for best performance should be configured for desired target a system clock frequency.

When modifying characteristics for a memory, such as in this example, it is good practice not to execute code in the same memory that is having its characteristics modified. Hence code to modify the flash performance parameters of the module’s configuration register will be executed from SRAM.

For a more complete checklist of performance items, see application note AN3519, “Optimizing Performance for the MPC5500 Family.”

Exercise: Change configuration parameters and attempt to determine performance differences.
26.2 Design

Flash configuration consists of optimizing the flash bus interface for the following areas:

- Enabling line buffers
- Number of wait states and pipeline hold cycles for a given frequency
- Line buffer prefetch controls
- Line buffer configurations (MPC551x and MPC56xxB/P/S only)
- Read-While-Write control (MPC56xxB/P/S only)

**NOTE:** Be sure to verify flash configuration parameters with the latest data sheet for each device.

26.2.1 Line Buffer Enable

**BFEN** or **BFE** (FBIU Line Read Buffers Enable): Enables or disables line read buffer hits.

*Recommendation:* Set BFEN = 1 to enable the use of line buffers.

26.2.2 Line Buffer Wait State and Hold Cycles

Wait states and hold cycles are defined to be *between the flash line buffers and flash array*, not between flash and core or crossbar.

**RWSC** (Read Wait State Control): Defines the number of wait states to be added to the flash array access time for reads.

**WWSC** (Write Wait State Control): Defines the number of wait states to be added to the flash array access time for writes.

**APC** (Address Pipelining Control): Address pipelining drives a subsequent access address and control signals while waiting for the current access to complete. APC defines the number of hold cycles between flash array access requests.

*Recommendation:* Use the parameter values in the reference manuals for the desired system clock frequency to get fastest performance.

26.2.3 Line Buffer Prefetch Controls

When one line buffer is being accessed from the crossbar, another line buffer can be prefetching the next sequential address from the flash array at the same time. This overlapping operation allows the flash module to provide sequential address accesses without any delay.

Prefetching does not provide any benefit if accesses are not sequential beyond a flash line. Therefore as a general rule, prefetching should be enabled when accesses are expected to be sequential, such as instructions. Generally data access has addresses more random than sequential, so prefetching will not be a benefit.

**IPFEN** or **IPFE** (Instruction Prefetch Enable): Enables line buffers to prefetch instructions.
Recommendation: Since most instruction accesses are sequential, instruction prefetching should be enabled for any access.

**DPFEN or DPFE** (Data Prefetch Enable): Enables line buffers to prefetch data.

Recommendation: Determining the setting is not as obvious as for instruction prefetch enable. Many data accesses may be sequential, such as table data, filter coefficients, and for MPC5606S, large graphic objects. For engine control, start with enabled for any read access. For MPC56xxB/P, start with DPFEN = 0 and for MPC5606S with graphics in internal flash, start with DPFEN = 1.

**PFLIM** (Prefetch Limit): Defines the maximum number of prefetches done on a buffer miss.

Recommendation: Prefetch the next sequential line on a hit or a miss.

### 26.2.4 Master Prefetch Enables / Disables

**MnPFE** (Master n Prefetch Enable) and **MnPFD** (MPC56xxB/P/S — Disable): Enables (disables on MPC56xxB/P/S) line buffers to prefetch an entire line in the flash array to a line buffer for that master.

The following table summarizes values for given devices from the respective device documentation.

Recommendation: Most of the time, only enable prefetching for core instructions. This is based on the rationale that, most of the time, prefetching the next sequential address usually benefits only instruction fetching. Few prefetch benefits occur from other masters, such as data fetching, external bus master, and DMA. Most of the time DMA is simply moving a byte, half word, or word for MPC5500 and MPC5600 devices. (However, this may not be true for bus masters such as a display controller, FlexRAY, or FEC. Settings may be optimized differently in these cases.) Recommend only prefetching for master 0 (core).

### 26.2.5 Line Buffer Configuration (MPC551x and MPC56xxB/P/S)

Some devices have two interfaces (ports) to the flash module. Each port has its own set of line buffers; hence there are two configuration registers to initialize. Line buffers for each port can be organized as a “pool” available for data and instructions, or with fixed partitions.

**LBCFG or BCFG** (Line Buffer Configuration): Specifies configuration of how many buffers are used for instruction fetches and data fetches.

Recommendation: Generally the optimal configuration would be to have a fixed partition of mostly instructions fetch line buffers, or a pool for both instruction and data fetches. On the MPC551x, the line buffer control value is loaded initially from the shadow block by BAM code.

### 26.2.6 Read-While-Write (RWW) Control (MPC56xxB/P/S)

Low cost flash arrays such as on MPC56xxB/P/S do not support RWW as in MPC55xx devices.

**RWWC** (Read-While-Write Control): Defines the flash controller response to flash reads while the array is busy with a program (write) erase operation.

Likely Recommendation: Terminate RWW or generate bus stall with abort and abort interrupt enabled so the CPU knows it occurred.
### Table 102. MPC56xxB/P/S Flash Configuration Example Values for 64 MHz System Clock

<table>
<thead>
<tr>
<th>Access</th>
<th>Parameter</th>
<th>General Recommendations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Code Flash (MPC56xxB/P: Bank 0, MPC56xxS: Banks 0 &amp; 2) 4 Line Buffers Per Port</td>
<td>Data Flash (Bank 1) 1 Line Buffer Per Port</td>
</tr>
<tr>
<td></td>
<td>Parameter Symbol in register PFCR0</td>
<td>Result value: 0x1084_126F</td>
</tr>
<tr>
<td>Port 0 (Connected to all masters on MPC56xxB/P, connected to core only on MPC56xxS)</td>
<td>Page Buffer Enable</td>
<td>B0_P0_BFE = 1</td>
</tr>
<tr>
<td></td>
<td>Instruction Prefetch Enable</td>
<td>B0_P0_IPFE = 1</td>
</tr>
<tr>
<td></td>
<td>Data Prefetch Enable</td>
<td>B0_P0_DPFE = 0</td>
</tr>
<tr>
<td></td>
<td>Prefetch Limit</td>
<td>B0_P0_PFLIM = 3</td>
</tr>
<tr>
<td></td>
<td>Page Buffer Configuration</td>
<td>B0_P0_BCFG = 3</td>
</tr>
<tr>
<td>Port 1 (Connected to DCU, DMA on MPC56xxS)</td>
<td>Page Buffer Enable</td>
<td>B0_P1_BFE = 1</td>
</tr>
<tr>
<td></td>
<td>Instruction Prefetch Enable</td>
<td>B0_P1_IPFE = 0</td>
</tr>
<tr>
<td></td>
<td>Data Prefetch Enable</td>
<td>B0_P1_DPFE = 1</td>
</tr>
<tr>
<td></td>
<td>Prefetch Limit</td>
<td>B0_P1_PFLIM = 1</td>
</tr>
<tr>
<td></td>
<td>Page Buffer Configuration</td>
<td>B0_P1_BCFG = 0</td>
</tr>
<tr>
<td>Array Access (for 64 MHz)</td>
<td>Read Wait States</td>
<td>BK0_RWSC = 2</td>
</tr>
<tr>
<td></td>
<td>Write Wait States</td>
<td>BK0_VWSC = 2</td>
</tr>
<tr>
<td></td>
<td>Adv. Pipeline Ctl.</td>
<td>BK0_APC = 2</td>
</tr>
<tr>
<td></td>
<td>Read While Write Ctl.</td>
<td>BK0_RRWC = 0</td>
</tr>
</tbody>
</table>
**Table 103.** MPC56xxB/P/S Crossbar Master Assignments.

<table>
<thead>
<tr>
<th>Crossbar Physical Master ID</th>
<th>MPC56xxB</th>
<th>MPC56xxP</th>
<th>MPC56xxS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>e200z0h core instructions</td>
<td>e200z0h core instructions</td>
<td>e200z0h core instructions</td>
</tr>
<tr>
<td>1</td>
<td>e200z0h core data</td>
<td>e200z0h core data</td>
<td>e200z0h core data</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>eDMA</td>
<td>eDMA</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>FlexRAY</td>
<td>DCU</td>
</tr>
<tr>
<td>4 to 7</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**Table 104.** MPC56xxB/P/S Access and Protection Example Settings

(Result value for MPC56xxS in PFAPR = 0x03F2 005D)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Symbol</th>
<th>MPC56xxB Value in Register PFAPR</th>
<th>MPC56xxP Value in Register PFAPR</th>
<th>MPC56xxS Value in Register PFAPR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arbitration Mode</td>
<td>ARBM</td>
<td>3 (start with round-robin and change after application testing as needed)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Master n Prefetch Disable</td>
<td>M0PFD</td>
<td>0 (since core instructions are mostly sequential, so prefetching should help)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>M1PFD</td>
<td>1 (assuming CPU data is not normally accessed sequentially)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>M2PFD</td>
<td>-</td>
<td>1 (assuming eDMA will not have significant consecutive accesses)</td>
<td>0 (assuming eDMA will have significant consecutive data used for graphics)</td>
</tr>
<tr>
<td></td>
<td>M3PFD</td>
<td>-</td>
<td>1 (assuming FlexRAY will not have significant consecutive accesses)</td>
<td>0 (assuming DCU will have significant consecutive data used for graphics)</td>
</tr>
<tr>
<td>Master n Access Protection</td>
<td>M0AP</td>
<td>1 (R only for core instructions)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>M1AP</td>
<td>3 (R+W access allowed for core data)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>M2AP</td>
<td>-</td>
<td>1 (R only for eDMA)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>M3AP</td>
<td>-</td>
<td>1 (R only unless flashing over FlexRay)</td>
<td>1 (R only for DCU)</td>
</tr>
</tbody>
</table>
Table 105. MPC551x, MPC55xx Flash Configuration Example Values
(MPC551x port 0 configuration register also uses values LBCF=0, ARB=1 and PRI=0 here and MPC5634M configuration register also uses GCE = 0.)

<table>
<thead>
<tr>
<th>Device &amp; Reference for APC, RWSC, WWSC</th>
<th>Max. Target Frequency</th>
<th>MnPFE Core (others 0x0)</th>
<th>APC</th>
<th>RWSC</th>
<th>WWSC</th>
<th>DPFEN</th>
<th>IPFEN</th>
<th>PFLIM</th>
<th>BFEN</th>
<th>Resulting Configuration Register and Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPC5510 - port 0 (Reference: MPC5510 Data Sheet, Rev. 3 &amp; MPC5510 Ref Manual, Rev. 1)</td>
<td>Reset Default</td>
<td>0</td>
<td>7</td>
<td>7</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>PFCRP0 = 0x0800 FF00</td>
</tr>
<tr>
<td></td>
<td>Up to 25 MHz</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td></td>
<td>PFCRP0 = 0x0801 0815</td>
</tr>
<tr>
<td></td>
<td>Up to 50 MHz</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td></td>
<td>PFCRP0 = 0x0801 2915</td>
</tr>
<tr>
<td></td>
<td>Up to 80 MHz</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td></td>
<td>PFCRP0 = 0x0801 4A15</td>
</tr>
<tr>
<td>MPC5533, MPC5534 (Reference: MPC5534 Rev. 1 Addendum, Rev. 1) (Note on up to 25 MHz row: This APC/RWSC/WWSC combination requires setting the Flash MCR register bit PRD=1.)</td>
<td>Reset Default</td>
<td>0</td>
<td>7</td>
<td>7</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>BUICR = 0x0000 FF00</td>
</tr>
<tr>
<td></td>
<td>Up to 25 MHz</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>BUICR = 0x0001 0855</td>
</tr>
<tr>
<td></td>
<td>Up to 50 MHz</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>BUICR = 0x0001 2955</td>
</tr>
<tr>
<td></td>
<td>Up to 75 MHz</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>BUICR = 0x0001 4A55</td>
</tr>
<tr>
<td></td>
<td>Up to 80 MHz</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>BUICR = 0x0001 6B55</td>
</tr>
<tr>
<td>MPC5553, MPC5554, MPC5565, MPC5566, MPC5567 (Reference: published data sheets as of April 2009)</td>
<td>Reset Default</td>
<td>0</td>
<td>7</td>
<td>7</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>BUICR = 0x0000 FF00</td>
</tr>
<tr>
<td></td>
<td>Up to 82 MHz</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>BUICR = 0x0001 28FD</td>
</tr>
<tr>
<td></td>
<td>Up to 102 MHz</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>BUICR = 0x0001 2AFD</td>
</tr>
<tr>
<td></td>
<td>Up to 132 MHz (MPC555x) or up to 135 MHz (MPC556x)</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>BUICR = 0x0001 4BFD</td>
</tr>
<tr>
<td></td>
<td>Up to 147 MHz (MPC5566 only)</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>BUICR = 0x0001 6CFD</td>
</tr>
<tr>
<td>MPC5632M, MPC5634M, MPC5634M (Reference: MPC5634M Ref. Manual, Rev. 2)</td>
<td>Reset Default</td>
<td>0</td>
<td>7</td>
<td>7</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>PFCR1 = 0x0000 FF00</td>
</tr>
<tr>
<td></td>
<td>Up to 40 MHz</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>PFCR1 = 0x0001 2955</td>
</tr>
<tr>
<td></td>
<td>Up to 62 MHz</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>PFCR1 = 0x0001 4A55</td>
</tr>
<tr>
<td></td>
<td>Up to 82 MHz</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>PFCR1 = 0x0001 6B55</td>
</tr>
</tbody>
</table>

Qorivva Simple Cookbook, Rev. 4
<table>
<thead>
<tr>
<th>Step</th>
<th>Relevant Bit Fields</th>
<th>Pseudo Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Init</td>
<td>Determine desired configuration data. In this case, assume an MPC555x running up to 82 MHz.</td>
<td>FLASH_CONFIG_DATA = 0x0001 29FD, FLASH_CONFIG_REG = FLASH_BIUCR</td>
</tr>
<tr>
<td>Configure Flash</td>
<td>In RAM, create array structure of machine code to write 32 bits data in r3 to address in r4, ensure the instruction completes, then return. Instruction code is: — stw r3, 0 (r4) — mbar — blr</td>
<td>uint32_t mem_write_code[] = {0x90640000, 0x7C0006AC, 0x4E800020 }</td>
</tr>
<tr>
<td></td>
<td>Create a new typedef for function pointer that does not return anything (void) and passes two integers (in r4, then r5 per EABI).</td>
<td>typedef void (*mem_write_code_ptr_t) (int, int)</td>
</tr>
<tr>
<td></td>
<td>Call memory write code function: Cast memory_write_code as a function pointer — then de-reference it, which converts it to a function which passes, in order: — Value to be written (goes into r3 per EABI) — Memory address used for write (goes into r4 per EABI)</td>
<td>(*((mem_write_code_ptr_t) mem_write_code)) (FLASH_CONFIG_DATA, &amp;FLASH_CONFIG_ADDR)</td>
</tr>
<tr>
<td>Enable Branch Acceleration and Branch Target Buffers</td>
<td>Enable branch acceleration for forward and backward branches (reset default = enabled)</td>
<td>HID0[PBRED]=0 spr HID0 = 0x0 (reset default)</td>
</tr>
<tr>
<td></td>
<td>Invalidate branch target buffers and Enable branch target buffers</td>
<td>BUCSR[BBFI]=1 spr BUCSR = 0x0000 0201</td>
</tr>
</tbody>
</table>
26.3 Code (Shown for MPC5554 with 64 MHz sysclk)

/* main.c - Example of flash configuration plus using branch target buffers */
/* Copyright Freescale Semiconductor, Inc 2008 All rights reserved. */
/* Rev 0.1 Jun 18 2008 Dan and Steve Mihalik */
/* Rev 0.2 Oct 30 2008 S. Mihalik - added mbar to mem_write_code */
/* Rev 0.3 May 20 2009 S. Mihalik - updated Config. value */
/* Rev 0.4 Sep 11 2009 S. Mihalik - corrected BUCSR value to 0x0201 */

#include "mpc5554.h" /* Include appropriate header file like mpc5554.h */

#define FLASH_CONFIG_DATA 0x001029FD    /* MPC554 config. value for up to 82 MHz */
#define FLASH_CONFIG_REG FLASH.BIUCR.R    /* Flash config. register address */

asm void enable_accel_BTB(void) {
    li      r0, 0      /* Enable branch acceleration (HID[PBRED]=0) */
    mtHID0  r0
    li      r0, 0x0201 /* Invalidate Branch Target Buffers and enable them */
    mtBUCSR r0
}

int main(void) {

    uint32_t i=0; /* Dummy idle counter */

    uint32_t mem_write_code [] = {
        0x90640000, /* stw r3,(0)r4 machine code; writes r3 contents to addr in r4 */
        0x7C0006AC, /* mbar machine code: ensure prior store completed */
        0x4E800020  /* blr machine code: branches to return address in link register */
    };

typedef void (*mem_write_code_ptr_t)(uint32_t, uint32_t);
    /* create a new Type def: a func pInter called mem_write_code_ptr_t */
    /* which does not return a value (void) */
    /* and will pass two 32 bit unsigned integer values */
    /* (per EABI, the first parameter will be in r3, the second r4) */

    (*((mem_write_code_ptr_t)mem_write_code)) /* cast mem_write_code as func ptr*/
        (FLASH_CONFIG_DATA,            /* which passes integer (in r3) */
         (uint32_t)&FLASH_CONFIG_REG); /* and address to write integer (in r4) */

    enable_accel_BTB();            /* Enable branch accel., branch target buffers */
    while(1) {i++;}                /* Wait forever */
}
26.4 Code (Shown for MPC56xxS with 64 MHz sysclk)

/* main.c - Example of flash configuration plus using branch target buffers */
/* Copyright Freescale Semiconductor, Inc 2009 All rights reserved. */
/* Rev 0.1 May 22 2009 S. Mihalik - Initial version based on AN2865 example */
/* Rev 0.2 Sep 11 2009 S. Mihalik - corrected BUCSR value to 0x0201 */
/* Rev 0.3 Mar 11 2010 S. Mihalik - corrected FLASH_CONFIG_DATA to 0x1084126F */
#include "jdp.h" /* Include appropriate header file like mpc5554.h */

#define FLASH_CONFIG_DATA 0x1084126F /* MPC56xxS flash config value for 64 MHz */
#define FLASH_CONFIG_REG CFLASH.PFCR0.R /* Flash config. register address */
#define FLASH_ACCESS_PROT_DATA 0x03F2005D /* MPC56xxS flash access prot. value */
#define FLASH_ACCESS_PROT_REG CFLASH.FAPR.R /* Flash Access Prot. Reg. address */

asm void enable.accel.BTB(void) {
  li r0, 0      /* Enable branch acceleration (HID[PBRED]=0) */
  mtHID0 r0
  li r0, 0x0201 /* Invalidate Branch Target Buffers and enable them */
  mtBUCSR r0
}

int main(void) {
  uint32_t i=0; /* Dummy idle counter */

  /* NOTE: Structures are default aligned on a boundary which is a multiple of */
  /* the largest sized element, 4 bytes in this case. The first two */
  /* instructions are 4 bytes, so the last instruction is duplicated to */
  /* avoid the compiler adding padding of 2 bytes before the instruction. */
  uint32_t mem_write_code_vle [] = {
    0x54640000, /* e_stw R3,(0)R4 machine code: writes R3 contents to addr in R4 */
    0x7C0006AC, /* mbar machine code: ensure prior store completed */
    0x00040004  /* 2 se_blr's machine code: branches to return address in link reg.*/
  };

typedef void (*mem_write_code_ptr_t)(uint32_t, uint32_t);
/* create a new type def: a func pointer called mem_write_code_ptr_t */
/* which does not return a value (void) */
/* and will pass two 32 bit unsigned integer values */
/* (per EABI, the first parameter will be in r3, the second r4) */

  (*((mem_write_code_ptr_t)mem_write_code_vle)) /* cast mem_write_code as func ptr*/
  (FLASH_CONFIG_DATA,            /* which passes integer (in r3) */
   (uint32_t)&FLASH_CONFIG_REG); /* and address to write integer (in r4) */

  (*((mem_write_code_ptr_t)mem_write_code_vle)) /* cast mem_write_code as func ptr*/
  (FLASH_ACCESS_PROT_DATA,        /* which passes integer (in r3) */
   (uint32_t)&FLASH_ACCESS_PROT_REG); /* and address to write integer (in r4) */

  enable_accel.BTB();            /* Enable branch accel., branch target buffers */

  while(1) {i++;}                /* Wait forever */
# Appendix A Interrupt Alignment Summary

The table below shows the alignment requirements for interrupts, interrupt tables, and interrupt handlers. Table names used in examples are listed for reference.

**Table 107. Address Locations and Alignment for Interrupt Tables, ISR’s, and Handlers**

<table>
<thead>
<tr>
<th>Interrupt Type</th>
<th>Memory Section in Examples</th>
<th>MPC551x, MPC55xxB/P/S</th>
<th>MPC555x</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Address</td>
<td>Alignment</td>
</tr>
<tr>
<td>Core Interrupts</td>
<td>ivor_branch_table (MPC551x, MPC56xxB/P/S only)</td>
<td>IVPR&lt;sub&gt;0:19&lt;/sub&gt;</td>
<td>4 KB (0x1000)</td>
</tr>
<tr>
<td></td>
<td>ivor_handlers</td>
<td>N.A.</td>
<td>N.A.</td>
</tr>
<tr>
<td>INTC SW mode interrupts</td>
<td>Software Vector Mode: intc_sw_isr_vector_table [VTBA]</td>
<td>INTC_IACKR&lt;sub&gt;0:20&lt;/sub&gt;</td>
<td>2 KB (0x800)</td>
</tr>
<tr>
<td>INTC HW mode interrupts</td>
<td>Hardware Vector Mode: intc_hw_branch_table</td>
<td>IVPR&lt;sub&gt;0:19 + 0x800&lt;/sub&gt; above a 4 KB boundary</td>
<td>2 KB (0x800)</td>
</tr>
</tbody>
</table>
Appendix B  Single Core Build Files

This section describes the build files used in this application note’s examples that use only a single core.

NOTE: These examples assume 32 KB of internal SRAM. Some devices, such as MPC5604S, only have 24 KB internal SRAM, so adjustments are needed for link files in that case.

B.1 Memory Map for Executing from Internal RAM

To execute from SRAM, the memory map in Figure 40 is used. Only 32 KB RAM is assumed, so the program can run on the MPC5500 family member with the least amount of memory. From this build, there is no special boot code — the debugger scripts must initialize internal RAM and MMU, as well as setting the instruction pointer to the beginning of code.

This memory map is intended as a general example for a single-core processor for both MPC551x and MPC555x devices. Therefore some memory section names are defined that do not exist for some builds. For example, the Decrementer example does not use the section “intc_hw_branch_table.” Similarly, section “ivor_branch_table” is only used for interrupts on MPC551x, and does not exist for MPC555x applications.

![Figure 40. Memory Map for Executing from Internal SRAM (IVPR Value is Passed as 0x4000 0000)](image-url)
B.1.1 CodeWarrior Link File for Execution from Internal RAM

/* 5500_ram.lcf - Simple minimal MPC5500 link file using 32 KB SRAM */
/* Aug 30 2007 initial version */
/* May 09 2008 SM: Put stack in it's own 1KB (0x400) memory segment */

MEMORY {
    interrupts_RAM: org = 0x40000000, len = 0x3000
    internal_RAM:   org = 0x40003000, len = 0x4C00
    stack_RAM:      org = 0x40007C00, len = 0x0400
}

SECTIONS {
    GROUP : {
        .ivor_branch_table    : {}              /* For MPC5516 core interrupts */
        .intc_hw_branch_table ALIGN (2048) : {} /* For INTC in HW Vector Mode */
        .ivor_handlers        : {}              /* Handlers for core interrupts */
    } > interrupts_RAM
    GROUP : {
        .intc_sw_isr_vector_table : {} /* For INTC in SW Vector Mode */
        .text : {
            *(.text)
            *(.rodata)
            *(.ctors)
            *(.dtors)
            *(.init)
            *(.fini)
            *(.eini)
            . = (.+15);
        }
        .sdata2       : {}
        extab      : {}
        extabindex : {}
    } > internal_RAM
    GROUP : {
        .data (DATA) : {}
        .sdata (DATA) : {}
        .sbss (BSS) : {}
        .bss (BSS) : {}
    } > internal_RAM
}

/* Freescale CodeWarrior compiler address designations */
_stack_addr = ADDR(stack_RAM)+SIZEOF(stack_RAM);
_stack_end  = ADDR(stack_RAM);
/* These are not currently being used
_heap_addr  = ADDR(.bss)+SIZEOF(.bss);
_heap_end   = ADDR(internal_RAM)+SIZEOF(internal_RAM);
*/
__IVPR_VALUE = ADDR(interrupts_RAM);
/* L2 SRAM Location (used for L2 SRAM initialization) */
L2SRAM_LOCATION = 0x40000000;
### B.1.2 Diab Link File for Execution from Internal RAM

/* 5500_ram.lin - Simple minimal MPC5500 link file for 32 KB SRAM */
/* Jul 05 2007 S.M. Initial version. */
/* May 09 2008 S.M. Put stack in separate memory segment
*/
/* Notes: 1. assumption: debug scripts will init SRAM, MMU, start vector */

```c
MEMORY {
   /*************************************************************************
   /*          Address          Length          Use                        */
   /*************************************************************************/
   /* 0x4000_0000 - 0x4000_2FFF  12 KB RAM - Interrupt area */
   /* 0x4000_3000 - 0x4000_7BFF  15 KB RAM - Code and data */
   /* 0x4000_7C00 - 0x4000_7FFF  1 KB RAM - Stack */
   /*************************************************************************/
   interrupts_ram:org = 0x40000000, len = 0x3000
   internal_ram:org = 0x40003000, len = 0x4C00
   stack_ram: org = 0x40007C00, len = 0x0400
}

SECTIONS {
   GROUP : {
      .ivor_branch_table : {} /* For MPC5516 core interrupts */
      .intc_hw_branch_table ALIGN (2048): {} /* For INTC in HW Vector Mode */
      .ivor_handlers : {} /* Handlers for core interrupts */
   } > interrupts_ram

   GROUP : {
      .intc_sw_isr_vector_table ALIGN (2048): {} /* For INTC SW Vector Mode */
      .text (TEXT) : {
         *(.text)
         *(.rodata)
         *(.ctors)
         *(.dtors)
         *(.init)
         *(.fini)
         *. = (.+15) & ~15;
      }
      .sdata2 (TEXT) : {}
   } > internal_ram

   GROUP : {
      .data (DATA) LOAD(ADDR(.sdata2)+SIZEOF(.sdata2)) : {}
      .sdata (DATA) LOAD(ADDR(.sdata2)+SIZEOF(.sdata2)+SIZEOF(.data)) : {}
      .sbss (BSS) : {}
      .bss (BSS) : {}
   } > internal_ram
}

__IVPR_VALUE = ADDR(interrupts_ram); /* Pass address to be loaded to spr IVPR */
```

Qorivva Simple Cookbook, Rev. 4
__SP_INIT = ADDR(stack_ram)+SIZEOF(stack_ram);
__SP_END = ADDR(stack_ram);
__DATA_ROM = ADDR(.sdata2)+SIZEOF(.sdata2);
__DATA_RAM = ADDR(.data);
__DATA_END = ADDR(.sdata)+SIZEOF(.sdata);
__BSS_START = ADDR(.sbss);
__BSS_END = ADDR(.bss)+SIZEOF(.bss);
__HEAP_START= ADDR(.bss)+SIZEOF(.bss);
__HEAP_END = ADDR(internal_ram)+SIZEOF(internal_ram);
B.1.3  Green Hills Link File for Execution from Internal RAM

/* 5500_ram.ld - Example minimal MPC5500 link file- 512 KB flash, 32 KB SRAM */
/* May 09 2008 S.M., G.L. Initial version. */
/* Notes: 1. assumption: debug scripts will init SRAM, MMU, start vector */

MEMORY
{
    /******************************************************************
    /*         Address      Length      Use                            */
    /******************************************************************
    /* 4000_0000-4000_2fff  12 KB RAM  Interrupt area                 */
    /* 4000_3000-4000_7BFF  19 KB RAM  Code and data except stack     */
    /* 4000_7C00-4000_7FFF  1 KB RAM  Stack                          */
    /******************************************************************
    interrupts_ram    :  ORIGIN = 0x40000000, LENGTH = 12K
    internal_ram      :  ORIGIN = .         , LENGTH = 19K
    stack_ram         :  ORIGIN = .         , LENGTH =  1K
}

CONSTANTS
{
    stack_reserve = 1K
    heap_reserve = 1K
}

SECTIONS
{

    // RAM SECTIONS
    .ivor_branch_table : > interrupts_ram    /* For MPC5516 core interrupts */
    .intc_hw_branch_table  ALIGN(2048) : > .  /* For INTC in HW Vector Mode */
    .ivor_handlers                      : > .  /* For core interrupt handlers */

    .init                        : > internal_ram
    .text   : > .
    .syscall     : > .  /* For GHS hostio support */
    .rodata   : > .
    .sdata2   : > .
    .secinfo  : > .  /* For GHS startup code */
    .fixaddr : > .
    .fixtype  : > .

    .PPC.EMB.sdata0                  ABS  : > .
    .PPC.EMB.sbss0                   CLEAR ABS : > .
    .sdabase   ALIGN(8) : > .
    .sdata     : > .
    .sbss      : > .
    .data      : > .
    .bss       : > .
    .heap     ALIGN(16) PAD(heap_reserve) : > .

Qorivva Simple Cookbook, Rev. 4
.stack ALIGN(16) PAD(stack_reserve) : > stack_ram
     /* SP init value addr stack_ram + pad */
__STACK_SIZE   = stack_reserve;
__SP_END       = ADDR(.stack);
__IVPR_VALUE   = MEMADDR(interrupts_ram); /* Pass address to be loaded to IVPR */

//
// These special symbols mark the bounds of RAM and ROM memory.
// They are used by the MULTI debugger.
//
__ghs_ramstart  = MEMADDR(interrupts_ram);
__ghs_ramend    = MEMENDADDR(stack_ram);
__ghs_romstart  = 0;
__ghs_romend    = 0;

//
// These special symbols mark the bounds of RAM and ROM images of boot code.
// They are used by the GHS startup code (_start and __ghs_ind_crt0).
//
__ghs_rambootcodestart  = MEMADDR(interrupts_ram);
__ghs_rambootcodeend    = ADDR(.fixtype);
__ghs_rombootcodestart  = 0;
__ghs_rombootcodeend    = 0;
}
B.2 Memory Map for Executing from Internal Flash

To execute from flash, the memory map in Figure 41 is used. Only 512 KB flash is assumed, so the program can run on the MPC5500 family member with the least amount of memory. When executing from flash, we must add some “boot” code, including a Reset Configuration Half Word (RCHW), a starting vector, and code to initialize SRAM. This is put in the “boot” memory section. After reset, the BAM code inside the device will initialize the MMU.

This memory map is intended as a general example for a single-core processor for both MPC551x and MPC555x devices. Therefore some memory section names are defined that do not exist for some builds. For example, the Decrementer example does not use the section “intc_hw_branch_table.” Similarly, section “ivor_branch_table” is only used for interrupts on MPC551x, and does not exist for MPC555x applications.

NOTE: These examples assume 32 KB of internal SRAM. Some devices, such as MPC5604S, only have 24 KB internal SRAM, so adjustments are needed for link files in that case.

![Figure 41. Memory Map for Executing from Internal Flash (IVPR Value is Passed as 0x0001 0000)](image)

- **Memory Segment Names**
  - boot_flash (64 KB)
  - interrupts_flash (64 KB)
  - internal_flash (384 KB)

- **Memory Section Names**
  - boot
  - ivor_branch_table
  - intc_hw_branch_table
  - ivor_handlers
  - intc_sw_isr_vector_table
  - (code, etc.)
  - (data, etc.)
  - stack_ram (1 KB)
  - (stack)

Figure 41. Memory Map for Executing from Internal Flash (IVPR Value is Passed as 0x0001 0000)
B.2.1 CodeWarrior Link File for Execution from Internal Flash

/* 5500_flash.lcf - Simple minimal MPC5500 link file using 32 KB SRAM */
/* Sept 20 2007 SM, DF initial version */
/* May 09 2008 SM: Put stack in it's own 1KB (0x400) memory segment */

MEMORY
{
    boot_flash: org = 0x00000000, len = 0x00010000
    interrupts_flash: org = 0x00010000, len = 0x00010000
    internal_flash: org = 0x00020000, len = 0x00060000
    internal_ram: org = 0x40000000, len = 0x00007C00
    stack_ram: org = 0x40007C00, len = 0x00400
}

/* This will ensure the rchw and reset vector are not stripped by the linker */
FORCEACTIVE { "bam_rchw" "bam_resetvector"

SECTIONS
{
    .boot LOAD (0x00000000) : {} > boot_flash  /* LOAD (0x0) prevents relocation by ROM copy during startup */

    GROUP : {
        /* Note: _e_ prefix enables load after END of that specified section */
        .ivor_branch_table (TEXT) LOAD (ADDR(interrupts_flash)) : {}
        .intc_hw_branch_table (TEXT) LOAD (_e_ivor_branch_table) ALIGN (0x800) : {}
        .ivor_handlers (TEXT) LOAD (_e_intc_hw_branch_table) : {}
        /* Each MPC555x handler require 16B alignmt */
    } > interrupts_flash

    GROUP : {
        .intc_sw_isr_vector_table ALIGN (2048) : {} /* For INTC in SW Vector Mode */
        .text : {
            *(.text)
            *(.rodata)
            *(.ctors)
            *(.dtors)
            *(.init)
            *(.fini)
            *(.eini)
            . = (.+15);
        }
        .sdata2 : {}
        extab : {}
        extabindex : {}
    } > internal_flash

Qorivva Simple Cookbook, Rev. 4
GROUP :
    .data  (DATA) : {}
    .sdata (DATA) : {}
    .sbss  (BSS)  : {}
    .bss   (BSS)  : {}
    .PPC.EMB.sdata0 : {}
    .PPC.EMB.sbss0  : {}
  ) > internal_ram

/* Freescale CodeWarrior compiler address designations */

_stack_addr = ADDR(stack_ram)+SIZEOF(stack_ram);
_stack_end = ADDR(stack_ram);

/* These are not currently being used
_heap_addr = ADDR(.bss)+SIZEOF(.bss);
_heap_end = ADDR(internal_ram)+SIZEOF(internal_ram);
*/
__IVPR_VALUE = ADDR(interrupts_flash);

/* L2 SRAM Location (used for L2 SRAM initialization) */

L2SRAM_LOCATION = 0x40000000;
B.2.2  Diab Link File for Internal Flash Execution

/* 5500_flash.lin - Simple minimal MPC5500 link file for 512 KB flash, 32 KB SRAM */
/* Jul 05 2007 S.M. Initial version. */
/* May 09 2008 S.M. Put stack in separate memory segment */

MEMORY
{
  /**************************************************************************/
  /* Address      Length      Use                            */
  /**************************************************************************/
  /* 0000_0000-0000_FFFF  64 KB Flash- RCHW & start vector */
  /* 0001_0000-0001_FFFF  64 KB Flash- Interrupt area */
  /* 0002_0000-0007_FFFF 384 KB Flash- Available for code, etc */
  /* 4000_0000-4000_7BFF  31 KB Internal RAM except stack */
  /* 4000_7C00-4000_7FFF  1 KB Internal RAM - stack */
  /**************************************************************************/
  
  boot_flash:  org = 0x00000000, len = 0x10000
  interrupts_flash: org = 0x00010000, len = 0x10000
  internal_flash: org = 0x00020000, len = 0x60000
  internal_ram: org = 0x40000000, len = 0x7C00
  stack_ram: org = 0x40007C00, len = 0x0400
}

SECTIONS
{
  .boot : {} > boot_flash

  GROUP :
    .ivor_branch_table : {} /* For MPC5516 core interrupts */
    .intc_hw_branch_table ALIGN (2048) : {} /* For INTC in HW Vector Mode */
    .ivor_handlers : {} /* For core interrupt handlers */
  } > interrupts_flash

  GROUP :
    .intc_sw_isr_vector_table ALIGN (2048) : {} /* For INTC SW Vector Mode */
    .text (TEXT) : {
      *(.text)
      *(.rodata)
      *(.ctors)
      *(.dtors)
      *(.init)
      *(.fini)
      *(.eini)
      . = (.+15) & ~15;
    }
  } > internal_flash

  GROUP :
    .data (DATA) LOAD(ADDR(.sdata2)+SIZEOF(.sdata2)) : {}
    .sdata (DATA) LOAD(ADDR(.sdata2)+SIZEOF(.sdata2)+SIZEOF(.data)) : {}
    .sbss (BSS) : {}
    .bss (BSS) : {}
  } > internal_ram
__IVPR_VALUE = ADDR(interrupts_flash); /* Pass address to to load to IVPR */

__SP_INIT = ADDR(stack_ram)+SIZEOF(stack_ram);
__SP_END = ADDR(stack_ram);
__DATA_ROM = ADDR(.sdata2)+SIZEOF(.sdata2);
__DATA_RAM = ADDR(.data);
__DATA_END = ADDR(.sdata)+SIZEOF(.sdata);
__BSS_START = ADDR(.sbss);
__BSS_END = ADDR(.bss)+SIZEOF(.bss);
__HEAP_START = ADDR(.bss)+SIZEOF(.bss);
__HEAP_END = ADDR(internal_ram)+SIZEOF(internal_ram);
B.2.3 Green Hills Link File for Execution from Internal Flash

/* 5500_flash.ld - Example minimal MPC5500 link file- 512 KB flash, 32 KB SRAM */
/* May 09 2008 S.M., G.L. Initial version. */

MEMORY
{
  /******************************************************************************/
  /*         Address      Length      Use                            */
  /******************************************************************************/
  /*  0000_0000-0000_FFFF  64 KB      Flash- RCHW & start vector        */
  /*  0001_0000-0001_FFFF  64 KB      Flash- Interrupt area          */
  /*  0002_0000-0007_FFFF 384 KB      Flash- Available for code, etc   */
  /*  4000_0000-4000_7BFF  31 KB      Internal RAM except stack      */
  /*  4000_7C00-4000_7FFF  1 KB      Internal RAM - Stack            */
  /******************************************************************************/
  boot_flash : ORIGIN = 0x00000000, LENGTH = 64K
  interrupts_flash : ORIGIN = . , LENGTH = 64K
  internal_flash : ORIGIN = . , LENGTH = 384K
  internal_ram : ORIGIN = 0x40000000, LENGTH = 31K
  stack_ram : ORIGIN = . , LENGTH = 1K
}

CONSTANTS {
  stack_reserve = 1K
  heap_reserve = 1K
}

SECTIONS {
  // FLASH SECTIONS
  .boot : > boot_flash
  .ivor_branch_table : > interrupts_flash /* For MPC5516 core interrupts */
  .intc_hw_branch_table ALIGN(2048) : > . /* For INTC in HW Vector Mode */
  .ivor_handlers : > . /* For core interrupt handlers */
  .init : > internal_flash
  .text : > . /* For GHS hostio support */
  .syscall : > .
  .rodata : > .
  .sdata2 : > .
  .secinfo : > . /* For GHS startup code */
  .fixaddr : > .
  .fixtype : > .
  .CROM.PPC.EMB.sdata0 CROM(.PPC.EMB.sdata0) : > . /* compressed initialized data */
  .CROM.sdata CROM(.sdata) : > . /* compressed initialized data */
  .CROM.data CROM(.data) : > . /* compressed initialized data */

Qorivva Simple Cookbook, Rev. 4
// RAM SECTIONS

.PPC.EMB.sdata0 ABS : > internal_ram
.PPC.EMB.sbss0 CLEAR ABS : > .
.sdabase ALIGN(8) : > .
.sdata : > .
.sbss : > .
.data : > .
.bss : > .
.heap ALIGN(16) PAD(heap_reserve) : > .

.stack ALIGN(16) PAD(stack_reserve) : > stack_ram
    /* SP init = addr stack_ram + pad */

__STACK_SIZE = stack_reserve;
__SP_END = ADDR(.stack);

__IVPR_VALUE = MEMADDR(interrupts_flash); /* Pass address to load to IVPR */

//
// These special symbols mark the bounds of RAM and ROM memory.
// They are used by the MULTI debugger.
//
// __ghs_ramstart = MEMADDR(internal_ram);
// __ghs_ramend = MEMENDADDR(stack_ram);
// __ghs_romstart = MEMADDR(boot_flash);
// __ghs_romend = MEMENDADDR(internal_flash);

//
// These special symbols mark the bounds of RAM and ROM images of boot code.
// They are used by the GHS startup code (_start and __ghs_ind_crt0).
//
// __ghs_rambootcodestart = 0;
// __ghs_rambootcodeend = 0;
// __ghs_rombootcodestart = ADDR(.boot);
// __ghs_rombootcodeend = ENDADDR(.fixtype);

}
B.3  Additions to Startup File for Flash Executable

The startup file, often called the “crt0” file, has two modifications below for a flash executable. RAM executable example programs perform these functions by executing a debugger script file, typically when the debugger starts up.

1. Add a .boot section that contains:
   — the Reset Configuration Half Word (RCHW)
   — the vector to the first line of code, typically labelled “_start”

2. Code to initialize all of internal RAM. This must be done before using RAM, to avoid ECC errors.

Below is example .boot code that is added to the startup file of these flash examples. The file is then renamed, such as “crt0new.s,” and included in the build. MPC563x devices have a second watchdog that can be enabled in the RCHW. For these examples, watchdog(s) are disabled by the RCHW for flash targets.

```assembly
.section .boot

.LONG 0x005A0000 # RCHW: WTE = SWT = PS = VLE = 0, BOOTID = 0x5A
.LONG _start
```

Sample code to initialize RAM is listed in the MPC55xx reference manuals. The code below is from the MPC5553/MPC5554 Reference Manual to initialize 64 KB SRAM. It is inserted in the startup file.

```assembly
init_l2RAM:
    lis r11, 0x4000 # base address of RAM, 64-bit word aligned
    li r12, 512 # loop counter for 64 bits x 512 = 64 KB RAM
    mtctr r12

init_l2ram_loop:
    stmw r0, 0(r11) # write all 32 GPRs to RAM
    addi r11, r11, 128 # increment the ram pointer; 32 GPRs x 4 B = 128
    bdnz init_l2ram_loop # loop for 64 KB of RAM
```
# "Makefile" for Building Flash & RAM Executable Programs

```makefile
# makefile : Sample makefile for MPC5500 to work with Diab v5.5.3.1
# Rev 1 - Jan, 2003 - based on example from S.M. & P.S.
# Rev 2 - March, 2006 - updated for DWARF 2.0 output
# Rev 3 - Feb, 2007 - modified to provide both RAM and FLASH output files

OBJS-RAM= main.o handlers.o ivor_branch_table.o       # For MPC551x only
OBJS-FLASH= main.o handlers.o crt0new.o ivor_branch_table.o    # For MPC551x only
#OBJS-RAM= main.o handlers.o      # For MPC555x only
#OBJS-FLASH= main.o handlers.o crt0new.o       # For MPC555x only

CC      = dcc
AS      = das
LD      = dcc
DUMP    = ddump

COPTS = -tPPC5554ES:cross -@E+err.log -g -O -c -Xdebug-dwarf2-extensions-off /
        -I ..\MPC5500
AOPTS = -tPPC5554ES:cross -@E+err.log -g
LOPTS-RAM = -tPPC5554ES:cross -@E+err.log -Ws -m6 -lc -l:crt0.o
LOPTS-FLASH = -tPPC5554ES:cross -@E+err.log -Ws -m6 -lc

EXECUTABLE-RAM = DEC-ram
EXECUTABLE-FLASH = DEC-flash

.SUFFIXES: .c .s
.c.o :
  $(CC) $(COPTS) -o $*.o $<
.s.o :
  $(AS) $(AOPTS) $<


$(EXECUTABLE-RAM).elf: makefile $(OBJS-RAM)
  $(LD) $(LOPTS-RAM) $(OBJS-RAM) -o $(EXECUTABLE-RAM).elf -Wm 5500_ram.lin /
  > $(EXECUTABLE-RAM).map
  $(DUMP) -tv $(EXECUTABLE-RAM).elf >>$(EXECUTABLE-RAM).map

# Generate s record file
$(EXECUTABLE-RAM).s19: $(EXECUTABLE-RAM).elf
  $(DUMP) -Rv -o $(EXECUTABLE-RAM).s19 $(EXECUTABLE-RAM).elf

$(EXECUTABLE-FLASH).elf: makefile $(OBJS-FLASH)
  $(LD) $(LOPTS-FLASH) $(OBJS-FLASH) -o $(EXECUTABLE-FLASH).elf /
  -Wm 5500_flash.lin > $(EXECUTABLE-FLASH).map
  $(DUMP) -tv $(EXECUTABLE-FLASH).elf >>$(EXECUTABLE-FLASH).map

# Generate s record file
$(EXECUTABLE-FLASH).s19: $(EXECUTABLE-FLASH).elf
  $(DUMP) -Rv -o $(EXECUTABLE-FLASH).s19 $(EXECUTABLE-FLASH).elf

clean:
  del *.o
```

Qorivva Simple Cookbook, Rev. 4

Freescale Semiconductor
B.5 VLE Implementation - CodeWarrior Project

CodeWarrior’s project wizard allows selection of the project to be VLE. If converting an existing project to use VLE, the following must be done.

In the project’s settings, change the following:

1. **Target — Build Extras:** If “Use External Debugger” is checked, the debugger may need to know code is VLE. For example, the PEMicro debugger would have the Argument:
   ```
   LOADTORAM -RESETFILE “mpc5516vle.mac”
   ```

2. **Language Settings — C/C++ PreProcessor:** Set the preprocessor macro: “#define VLE_IS_ON 1”. This gets used in MPC55xx_init.c when setting up the RCHW.

3. **Code Generation — EPPC Processor:** In the “e500/Zen Options” box, check the item “Generate VLE Instructions.”

The linker files must indicate code sections to be VLE. Example:
```
SECTIONS
{
   .__bam_bootarea LOAD (0x00000000): {} > resetvector
GROUP : {
   .ivor_branch_table_p0 (VLECODE) LOAD (0x00001000) :
   .intc_hw_branch_table_p0 LOAD (0x00001800): {}
   __exception_handlers_p0 (VLECODE) LOAD (0x00001100) :
} > exception_handlers_p0
GROUP : {
   .intc_sw_isr_vector_table_p0 ALIGN (2048) :
   .init : {}
   .init_vle (VLECODE) : {
      *(.Init)
      *(.init_vle)
   }
   .text : {}
   .text_vle (VLECODE) ALIGN(0x1000): {
      *(.text)
      *(.text_vle)
   }
   .rodata (CONST) : {
      *(.rdata)
      *(.rodata)
   }
   .ctors : {}
   .dtors : {}
   extab : {}
   extabindex : {}
} > internal_flash
```

Any assembly function must either be encapsulated in a C function or use VLE mnemonics, which start with “e_” or “se_”.

Qorivva Simple Cookbook, Rev. 4

244 Freescale Semiconductor
Appendix C  MPC56xxB/P/S Peripheral Clocks

Clocks to peripherals are sometimes not connected after reset. If a peripheral does not have a clock, there will be a bus error when attempting to access any of its registers. Software must initialize in two ways:

C.1 Peripheral Clock Gating on a Mode Basis

Clocks are gated to peripherals based on ME_RUN_PCx, ME_LP_PCx registers and each peripheral’s ME_PCTLx register. This allows peripherals to be clocked in some modes, unclocked in others. The table below lists the ME_PCTLx registers and the peripherals they control.

Table 108. Peripheral Control Registers

<table>
<thead>
<tr>
<th>ME_PCTL #</th>
<th>Peripheral</th>
<th>MPC56xxB</th>
<th>MPC56xxP</th>
<th>MPC56xxS</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 - 5</td>
<td>DSPI 0:1</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>6</td>
<td>DSPI 2</td>
<td>Y</td>
<td>Y</td>
<td>—</td>
</tr>
<tr>
<td>7</td>
<td>DSPI 3</td>
<td>—</td>
<td>Y</td>
<td>—</td>
</tr>
<tr>
<td>10</td>
<td>QuadSPI</td>
<td>—</td>
<td>—</td>
<td>Y</td>
</tr>
<tr>
<td>16</td>
<td>FlexCAN 0</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>17</td>
<td>FlexCAN 1</td>
<td>Y</td>
<td>—</td>
<td>Y</td>
</tr>
<tr>
<td>18</td>
<td>FlexCAN 2</td>
<td>Y</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>19 - 21</td>
<td>FlexCAN 3:5</td>
<td>Y</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>23</td>
<td>DMA Mux</td>
<td>—</td>
<td>—</td>
<td>Y</td>
</tr>
<tr>
<td>24</td>
<td>FlexRAY</td>
<td>—</td>
<td>Y</td>
<td>—</td>
</tr>
<tr>
<td>26</td>
<td>Safety Port</td>
<td>—</td>
<td>Y</td>
<td>—</td>
</tr>
<tr>
<td>32</td>
<td>ADC 0</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>33</td>
<td>ADC 1</td>
<td>—</td>
<td>Y</td>
<td>—</td>
</tr>
<tr>
<td>35</td>
<td>CTU 0</td>
<td>Y</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>38 - 39</td>
<td>eTimer 0:1</td>
<td>Y</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>41</td>
<td>FlexPWM 0</td>
<td>—</td>
<td>Y</td>
<td>—</td>
</tr>
<tr>
<td>44</td>
<td>I2C DMA 0</td>
<td>Y</td>
<td>—</td>
<td>Y</td>
</tr>
<tr>
<td>45 - 47</td>
<td>I2C DMA 1:3</td>
<td>—</td>
<td>—</td>
<td>Y</td>
</tr>
<tr>
<td>48 - 49</td>
<td>LIN FLEX 0:1</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>50 - 51</td>
<td>LIN FLEX 2:3</td>
<td>Y</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>56</td>
<td>Gauge Driver</td>
<td>—</td>
<td>—</td>
<td>Y</td>
</tr>
<tr>
<td>57</td>
<td>CTUL</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>60</td>
<td>CAN Sampler</td>
<td>Y</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>61</td>
<td>LCD</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>62</td>
<td>Sound Gen.</td>
<td>—</td>
<td>—</td>
<td>Y</td>
</tr>
<tr>
<td>63</td>
<td>DCU</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>
C.2 Clock Generation on a Peripheral or Peripheral Set Basis

In addition to the peripheral clock gating, some peripherals also require generating a clock to them. This clock generation is done on a “peripheral set” basis where all peripherals in a group run at the same frequency. Clocks to those peripherals may be disabled after reset and must be enabled with a clock divider value. Sometimes you must specify the clock source such as PLL, external oscillator, etc. The following table lists the sets and the registers that must be configured to use any peripheral in that set.

<table>
<thead>
<tr>
<th>Peripheral Set</th>
<th>MPC56xxB</th>
<th>MPC56xxP (after cut 1)</th>
<th>MPC56xxS (after cut 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Registers to Enable and Generate Clock</td>
<td>Registers to Enable and Generate Clock</td>
<td>Registers to Enable and Generate Clock</td>
</tr>
<tr>
<td>1</td>
<td>LINFlex I2C</td>
<td>CGM_SC_DC0</td>
<td>Motor Control</td>
</tr>
<tr>
<td>2</td>
<td>FlexCAN DSPI</td>
<td>CGM_SC_DC1</td>
<td>CMU Monitor</td>
</tr>
<tr>
<td>3</td>
<td>eMIOS ADC CTU</td>
<td>CGM_SC_DC2</td>
<td>Safety Port</td>
</tr>
<tr>
<td>—</td>
<td>FlexRay</td>
<td>CGM_AC3_SC</td>
<td>DCU</td>
</tr>
<tr>
<td>—</td>
<td>eMIOS_0</td>
<td>CGM_AC1_SC</td>
<td>CGM_AC1_DC0</td>
</tr>
<tr>
<td>—</td>
<td>eMIOS_1</td>
<td>CGM_AC2_SC</td>
<td>CGM_AC2_DC0</td>
</tr>
<tr>
<td>—</td>
<td>QuadSPI</td>
<td>CGM_AC3_SC</td>
<td></td>
</tr>
</tbody>
</table>

Table 109. Peripheral Set Clock Generation Registers