Freescale Semiconductor
Application Note

Document Number: AN5191
Rev. 0, 09/2015

MPC5777C Software Initialization and Optimization

by: Steven Becerra and David Erazmus

Contents

1 Introduction....1
2 Overview....2
3 Startup code....3
4 MCU optimization....14
5 Conclusion....16
6 Code....16

1 Introduction

This application note describes a recommended software initialization procedure for the MPC5777C 32-bit Power Architecture® automotive microcontroller. This covers starting both Power Architecture cores, memory management units (MMU), clock frequency (PLL), watchdog timers, flash memory controller, and internal static RAM. Recommended configuration settings for these modules will be given for the purpose of optimizing system performance.

The MPC5777C is a high-performance 32-bit Power Architecture Microcontroller for powertrain applications. The three e200z7 host processor cores, two of which run in lockstep, of the MPC5777C are compatible with the Power Architecture Book E architecture. They are 100% user-mode compatible (with floating point library) with the classic PowerPC instruction set. The Book E architecture has enhancements that improve the architecture's fit in embedded applications. In addition to the standard and VLE Power Architecture instruction sets, both cores have additional instruction support for digital signal processing (DSP).

The MPC5777C has two levels of memory hierarchy; separate 16 K instruction and 16 K data caches for each of the two cores and 512 KB of on-chip SRAM including 48 KB standby RAM. 8 MB of internal flash memory is provided. An external bus interface is also available for special packaged parts to support application development and calibration.

© 2015 Freescale Semiconductor, Inc.

freescale

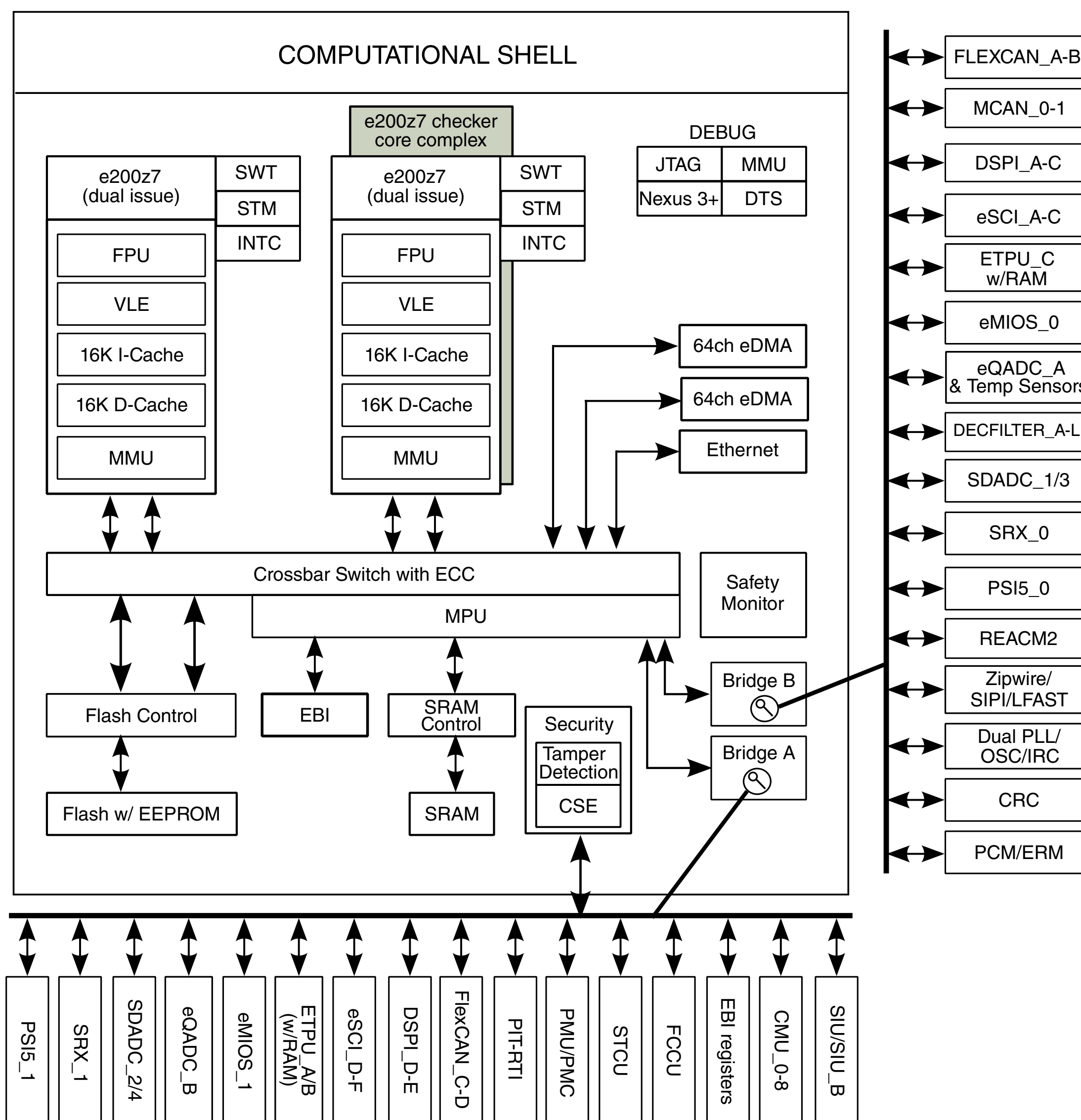

Figure 1. MPC5777C block diagram

2 Overview

There are several options to consider when discussing the structure of our embedded software application. The first is how it will execute. The application can be stored in internal flash memory or it can be downloaded from an external device, such as a debugger, or via a serial communications link. This affects certain steps in the initialization process and where applicable, this will be noted. Another option is choosing Variable Length Encoding instructions (VLE) vs. PowerPC Book E instructions. The assembly code examples shown in this application note will use VLE mnemonics and syntax, but can easily be translated into the Book E variant.

3 Startup code

The first part of the initialization procedure executes on the primary core (core 0) from the reset vector or program entry point and performs the minimal setup needed to prepare for C code execution. Another goal of this stage is to optimize the startup procedure's execution time. This involves taking certain initialization steps in a particular order:

1. Reset configuration and watchdog
2. Program PLL
3. Configure memory management unit
4. Enable instruction and data caches
5. Initialize SRAM
6. Initialize C runtime environment
7. Start core 1

Steps 1-6 are performed first by core 0. Core 1 will not be started until execution enters the main C routine. It is possible to start core 1 slightly earlier but core 0 should at least complete PLL and SRAM initialization first.

3.1 Reset configuration and watchdog

There are several ways to begin software execution after device reset. These are controlled by the Boot Assist Module (BAM).

- Boot from internal flash
- Serial boot via SCI or CAN interface with optional baud-rate detection
- Boot from a memory connected to the MCU development bus (EBI) with multiplexed or separate address and data lines (not available on all packages)

When using a hardware debugger connected via the JTAG or Nexus ports, the BAM can be bypassed. The debugger can download software to RAM via the debug interface and specify a start location for execution. In this case, much of the low-level device initialization is accomplished by the debugger using configuration scripts.

This application note will focus on the internal flash boot case because it performs all initialization tasks either in the BAM or explicitly in the application code. During any power-on, external, or internal reset event, except for software reset, the BAM begins by searching for a valid Reset Configuration Half Word (RCHW) in internal flash memory at one of the following pre-defined addresses.

Table 1. Possible RCHW locations in the internal flash memory

Priority	Address
0	0x0080_0000
1	0x0002_0000
2	0x0003_0000
3	0x0000_0000
4	0x0001_0000

The RCHW is a collection of control bits that specify a minimal MCU configuration after reset. If a valid RCHW is not found, the BAM will attempt a serial boot. Here is the format for the RCHW:

Table 2. Reset configuration half word

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	SWT	WTE	PS0	VLE	Boot Identifier							
Reserved				x[1]	x[1]	x[1]	x[1]	0	1	0	1	1	0	1	0

1. x = User-defined

Where:

- SWT = Software watchdog timer enable
- WTE = MCU core watchdog timer enable
- PS0 = Port size
- VLE = VLE Code Indicator

See the MPC5777C reference manual for more details on these bit-field definitions.

The RCHW occupies the most significant 16 bits of the first 32-bit internal memory word at the boot location. The next 32 bits contain the boot vector address. After applying the RCHW, the BAM will branch to this boot vector. During software initialization, reserve space for both of these 32-bit locations in the linker directive file as follows:

```
MEMORY
{
    flash_rcw : org = FLASH_BASE_ADDR,   len = 0x8
    …
}

SECTIONS
{
    .rcw              : {} > flash_rcw
    …
}
```

In the initialization code file, these two locations are generated with a valid RCHW encoding and the start address symbol for code entry point.2

```
.section .rcw
    .LONG 0x015A0000    # RCHW
    .LONG _start        # Code entry point
```

In the example above, the core (WTE) and software (SWT) watchdog timers are both disabled. These can operate independently, but it is common to use just one or the other in an application. When debugging, the RCHW is not applied when the BAM does not execute, so the debugger must disable these timers so that they do not interfere with application debug sessions. Disabling the core watchdog is necessary, because it cannot be disabled by software once it is enabled. The software watchdog starts out in an unlocked state, so the control register is still writable. If desired, the enable bit can be cleared to prevent watchdog operation during a debug session, if the debug tool does not handle this with its own configuration scripts.

NOTE

If either watchdog timer is enabled, there may be points within the initialization procedure that require watchdog service, depending on the timeout period of the watchdog.

3.2 Programming the PCS

The Progressive Clock Switch (PCS) module is used to prevent sudden voltage drops and overshoots when the system clock frequency is switched from one clock source to another. The PCS module can be programmed for both ramp-up and ramp-down and automatically monitors changes of the system clock source to trigger the progressive clock switching feature. Since the PCS assumes that the Internal Reference Clock (IRC) is the lowest clock frequency of the device, the user should not use PCS when switching to or from a frequency less than the IRC frequency.

The PCS module can detect if the system clock source is changing to and from the XOSC, PLL1, and PLL0. A set of three registers need to be programmed for each of the system clock sources that will be used. These registers include the PCS Divider Change Register (PCS_DIVCn), PCS Divider End Register (PCS_DIVEn), and PCS Divider Start Register (PCS_DIVSn). PCS_DIVC1, PCS_DIVE1, and PCS_DIVS1 need to be programmed if to configure the PCS module for XOSC. There also exists a single PCS Switch Duration Register (PCS_SDUR) that applies to all three of the clock sources.

NOTE

To use the PCS module, SIU_SYSDIV[PCSEN] must be set to 1 before changing SIU_SYSDIV[SYSCLKSEL] to ensure correct behavior. The PCS module will not function correctly if SIU_SYSDIV[PCSEN] is written to 1 in the same write as a change to SIU_SYSDIV[SYSCLKSEL].

To determine the values for these registers, see the PCS section in the reference manual.

The following example configures the PCS module for PLL1 given PLL1 will operate at 264 MHz, maximum allowable current change (mA/µs), and step duration (µs) of 1. This values used in the code below are just an example. The user should change these values based on the user's power supply. See the MPC5777C reference manual for more details on configuring the PCS.

```
#******************************
# configure PCS
#******************************

# Configure PCS Switch Duration
# SDUR = 0x10
    e_lis   r3,0xFFE7
    e_lis   r4,0x1000
    e_stw   r4,0x700(r3)

# Configure PLL1 Divider Change Register
# INIT = 0x4DF
# RATE = 0x30
    e_lis   r3,0xFFE7
    e_lis   r4,0x04DF
    e_or2i  r4,0x0030
    e_stw   r4,0x710(r3)

# Configure PLL1 Divider End Register
# DIVE = 0x4073
    e_lis   r3,0xFFE7
    e_li    r4,0x4073
    e_stw   r4,0x714(r3)

# Configure PLL1 Divider Start Register
# DIVS = 0x45B7
    e_lis   r3,0xFFE7
    e_li    r4,0x45B7
    e_stw   r4,0x718(r3)
```

3.3 Programming the PLL

The PLL module contains two clock domains coming from separate PLLs, PLL0 and PLL1. PLL1 feeds the system clock of the cores, cross-bar switch, peripheral bridges, memories, debug logic, and the memory mapped portion of peripherals. PLL0 feeds the state machines and protocol engines of communication and timer peripheral modules. PLL1 can be configured to be frequency modulated to reduce electromagnetic emissions. PLL0 is intended to provide an accurate reference clock for the peripheral timer and communication functions and cannot tolerate frequency modulation. The two outputs of PLL0 are PHI and PHI1. The PHI output clock drives various peripheral clocks and the system clock when PLL1 is not locked or active. The PHI1 output clock provides one of the input references for PLL1.

The user can configure both PLLs by writing to the Dual PLL Digital Interface (PLLDIG) registers. In normal operation, the following equations can be used to calculate the programming values for the two PLLs:

$F_{pll0_phi} = F_{pll0_ref}$ **x PLL0DV[MFD] / ((PLL0DV[PREDIV]) x (PLL0DV[RFDPHI]))**

$F_{pll0_phi1} = F_{pll0_ref}$ **x PLL0DV[MFD] / ((PLL0DV[PREDIV]) x (PLL0DV[RFDPHI1]))**

$F_{pll1_phi} = F_{pll1_ref}$ **x PLL1DV[MFD] / (2 x PLL1DV[RFDPHI])**

Where:

- F_{pll0_phi} = PLL0 PHI output frequency
- F_{pll0_phi1} = PLL0 PHI1 output frequency
- F_{pll1_phi} = PLL1 PHI output frequency
- **PLL0DV[MFD]** = PLL0 Loop multiplication factor divider
- **PLL0DV[PREDIV]** = PLL0 Input clock predivider
- **PLL0DV[RFDPHI]** = PLL0 PHI reduced frequency divider
- **PLL0DV[RFDPHI1]** = PLL0 PHI1 reduced frequency divider
- **PLL1DV[MFD]** = PLL1 Loop multiplication factor divider
- **PLL1DV[RFDPHI]** = PLL1 PHI reduced frequency divider

Table 3. Example PLL settings

F_{pll0_phi}	F_{pll0_ph1}	F_{pll1_ph}	F_{pll0_ref}	PLL0DV[MFD]	PLL0DV[PREDIV]	PLL0DV[RFDPHI]	PLL0DV[RFDPHI1]	PLL1DV[MFD]	PLL1DV[RFDPHI]
192 MHz	48 MHz	264 MHz	40 MHz	72	5	3	12	22	2
200 MHz	50 MHz	262.5 MHz	40 MHz	40	4	2	8	21	2

The following example sets up PLL0 to produce a 192 MHz peripheral clock and PLL1 to produce a 264 MHz system clock assuming a 40 MHz reference crystal.

```
#******************************
# configure PLL0 to 192 MHz (40 MHz XOSC reference)
# configure PLL1 to 264 MHz (48 MHz PLL0 PHI1 output reference)
# progressive clock switching enabled
#******************************

# Disable PLL0
    e_lis   r3,0xFFE7
    e_or2i  r3,0x4000
    e_li    r4,0
    e_stw   r4,0(r3)
# Disable PLL1
    e_stw   r4,0x20(r3)

# Select XOSC as PLL0 source
# Select PHI1 output of PLL0 as PLL1 source
    e_lis   r3,0xC3F9
    e_lis   r4,0x0400
    e_or2i  r4,0x0010
    e_stw   r4,0x9A0(r3)
```

```
# Program PLL0
# RFDPHI1 = 0x0C
# RFDPHI = 0x03
# PREDIV = 0x5
# MFD = 0x48
    e_lis   r3,0xFFE7
    e_or2i  r3,0x4000
    e_lis   r4,0x6003
    e_or2i  r4,0x5048
    e_stw   r4,0x8(r3)

# Wait for stable XOSC
xosc_wait:
    e_lis   r3,0xC3F9
    e_lwz   r4,0xC(r3)      # load SIU_RSR
    e_andi. r4,r4,0x80
    se_beq  xosc_wait

# Turn on PLL0 and wait for lock
    e_lis   r3,0xFFE7
    e_or2i  r3,0x4000
    e_li    r4,0x0300
    e_stw   r4,0(r3)
wait_for_pll0:
    e_lwz   r4,0x4(r3)      # load PLLDIG_PLL0SR
    e_andi. r4,r4,0x4
    se_beq  wait_for_pll0

# Program PLL1
# RFDPHI = 0x02
# MFD = 0x16
    e_lis   r4,0x0002
    e_or2i  r4,0x0016
    e_stw   r4,0x28(r3)

# Turn on PLL1 and wait for lock
    e_li    r4,0x0300
    e_stw   r4,0x20(r3)
wait_for_pll1:
    e_lwz   r4,0x24(r3)
    e_andi. r4,r4,0x4
    se_beq  wait_for_pll1
```

At this point, though the PLL is locked on the desired clock rate, the device is still being clocked by the internal RC oscillator. Select the PLL1 as the new system clock source and other clock dividers used in this example.

```
#******************************
# Select clock dividers and sources
#   PLL0 ref = XOSX
#   PLL1 ref = PLL0 PHI1
#   PERCLKSEL = PLL1
#   PERDIV = Divide-by-2
#   MCANSEL = XOSC
#   SYSCLKSEL = PLL1
#   ETPUDIV = Divide-by-1
#   SYSCLKDIV = Divide-by-1
#   PCS = Enabled
#******************************
    e_lis   r3,0xC3F9
    e_lis   r4,0x0500
    e_or2i  r4,0x2111
    e_stw   r4,0x9A0(r3)
```

3.4 Memory management unit (MMU)

The BAM includes a default setup for the MMU. The first thing the BAM program does is configure the MMU to allow access to all MCU internal resources. Below is an example of MMU configuration that enables Variable-Length Encoding, alters some default Table Lookup Buffer (TLB) sizes, and adds an entry for stack-in cache. The resulting MMU table is as follows:

Table 4. Example MMU table configuration

TLB	Space	Address	Size	Attributes
0	Peripheral bridge B	0xFFE0_0000	2 MB	Cache inhibit, Guarded
1	Flash	0x0000_0000	8 MB	VLE
2	External bus	0x2000_0000	16 MB	VLE
3	SRAM	0x4000_0000	512 KB	Write-through cache, VLE
4	Peripheral bridge A	0xC3E0_0000	2 MB	Cache inhibit, Guarded
5	Stack	0x4008_0000	8 KB	-

The folowing code configures the MMU with the parameters from Table 4.

```
#TLB1 = internal flash @ 0x0000_0000, VLE
    e_lis   r3,0x1001
    mtspr   mas0,r3
    e_lis   r3,0xC000
    e_or2i  r3,0x0700
    mtspr   mas1,r3
    e_lis   r3,0x0000
    e_or2i  r3,0x0020
    mtspr   mas2,r3
    e_lis   r3,0x0000
    e_or2i  r3,0x003F
    mtspr   mas3,r3
    msync     # Synchronize in case running from flash
    tlbwe
    se_isync  # Synchronize in case running from flash

#TLB3 = internal SRAM @ 0x4000_0000, VLE, Write-Through Cache
    e_lis   r3,0x1003
    mtspr   mas0,r3
    e_lis   r3,0xC000
    e_or2i  3,0x0480
    mtspr   mas1,r3
    e_lis   r3,0x4000
    e_or2i  r3,0x0030
    mtspr   mas2,r3
    e_lis   r3,0x4000
    e_or2i  r3,0x003F
    mtspr   mas3,r3
    msync     # Synchronize in case running from SRAM
    tlbwe
    se_isync  # Synchronize in case running from SRAM

#TLB0 = pbridgeB @ 0xFFE0_0000, Cache inhibited, Guarded
    e_lis   r3,0x1000
    mtspr   mas0,r3
    e_lis   r3,0xC000
    e_or2i  r3,0x0580
    mtspr   mas1,r3
    e_lis   r3,0xFFE0
    e_or2i  r3,0x000A
    mtspr   mas2,r3
    e_lis   r3,0xFFE0
```

```
    e_or2i  r3,0x003F
    mtspr   mas3,r3
    tlbwe

#TLB2 = external bus @ 0x2000_0000, VLE
    e_lis   r3,0x1002
    mtspr   mas0,r3
    e_lis   r3,0xC000
    e_or2i  r3,0x0700
    mtspr   mas1,r3
    e_lis   r3,0x2000
    e_or2i  r3,0x0020
    mtspr   mas2,r3
    e_lis   r3,0x0000
    e_or2i  r3,0x003F
    mtspr   mas3,r3
    tlbwe

#TLB4 = pbridgeA @ 0xC3E0_0000, Cache inhibited, Guarded
    e_lis   r3,0x1004
    mtspr   mas0,r3
    e_lis   r3,0xC000
    e_or2i  r3,0x0580
    mtspr   mas1,r3
    e_lis   r3,0xC3E0
    e_or2i  r3,0x000A
    mtspr   mas2,r3
    e_lis   r3,0xC3E0
    e_or2i  r3,0x003F
    mtspr   mas3,r3
    tlbwe

#TLB5 = 4k stack for each core (will be locked in cache)
# (Note: located just after 512k TLB3 entry for SRAM)
    e_lis   r3,0x1005
    mtspr   mas0,r3
    e_lis   r3,0xC000
    e_or2i  r3,0x0180
    mtspr   mas1,r3
    e_lis   r3,0x4008
    e_or2i  r3,0x0000
    mtspr   mas2,r3
    e_lis   r3,0x4008
    e_or2i  r3,0x003F
    mtspr   mas3,r3
    tlbwe
```

Note that in this example, tlbwe is preceded by msync and followed by se_isync for TLB1 and TLB3. These synchronization steps are taken in case the code being executed is from the region being modified.

3.5 Enable caches

The core instruction and data caches are enabled through the L1 Cache Control and Status Registers 0 & 1 (L1CSR0 and L1CSR1). The instruction cache is invalidated and enabled by setting the ICINV and ICE bits in L1CSR1. The data cache is enabled by setting DCINV and DCE in L1CSR0. The cache invalidate operation takes some time and can be interrupted or aborted. Because the cache invalidate operation occurs, in the code example below, during the boot-up precedure, this operation will not be interrupted or aborted.

```
    #-------------------------------------------------------#
    # invalidate and enable the data and instruction caches #
    #-------------------------------------------------------#
    # data cache
    e_lis   r3,0x0010
    e_or2i  r3,0x0003
    mtspr   l1csr0,r3
```

```
    # inst cache
    e_lis   r3,0x0
    e_or2i  r3,0x0003
    mtspr   l1csr1,r3
```

The following code represents a more robust cache enable routine that may be used elsewhere in the application, if desired. This code checks to ensure the invalidation has successfully completed and if not, retries the operation before enabling the cache. This code may be used with interrupts enabled, provided that those interrupts are properly handled and cleared. If the invalidate operation cannot complete without being interrupted due to a heavy interrupt load in the system, it is better to disable interrupts first.

```
cfg_ICACHE:

    #--------------------------------------------#
    # Invalidate Instruction Cache - Set ICINV   #
    # bit in L1CSR1 Register                     #
    #--------------------------------------------#
    e_lis   r5, 0x0000
    e_or2i  r5, 0x0002
    mtspr   l1csr1,r5
    se_isync

    #--------------------------------------------#
    # Mask out ICINV and ICABT to see if         #
    # invalidation is complete (i.e. ICINV=0,    #
    # ICABT=0)                                   #
    #--------------------------------------------#
label_ICINV_check:
    #--------------------------------------------#
    # Load Registers with Masks:                 #
    # Load ICINV mask into R4                    #
    # Load ICABT mask into R6                    #
    # Load ICABT clear mask into R7              #
    #--------------------------------------------#
    e_lis   r4, 0x0000
    e_or2i  r4, 0x0002
    e_lis   r6, 0x0000
    e_or2i  r6, 0x0004
    e_lis   r7, 0xFFFF
    e_or2i  r7, 0xFFFB
CHECK_ICINV:

    #--------------------------------------------#
    # Read L1CSR1 register, store in r3          #
    #--------------------------------------------#
    mfspr   r3, l1csr1
    #--------------------------------------------#
    # check for an ABORT of the cache invalidate #
    # operation                                  #
    #--------------------------------------------#
    se_and. r6, r3
    se_beq   NO_ABORT
    #--------------------------------------------#
    # If abort detected, clear ICABT bit and     #
    # re-run invalidation                        #
    #--------------------------------------------#
    se_and. r7, r3
    mtspr   l1csr1, r7
    se_isync
    se_b    cfg_ICACHE

NO_ABORT:
    #--------------------------------------------#
    # Check that invalidation has completed -    #
    # (ICINV=0). Branch if invalidation not      #
    # complete.                                  #
    #--------------------------------------------#
    se_and. r4, r3
    se_bne   CHECK_ICINV
```

```
#--------------------------------------------#
# Enable the ICache by performing a          #
# read/modify/write of the ICE bit in the    #
# L1CSR1 register                            #
#--------------------------------------------#
mfspr   r5, l1csr1
e_or2is r5, 0x0000
e_or2i  r5, 0x0001  # Store L1CSR1 value to R5 (ICE=1)
mtspr l1csr1, r5    # Write R5 to L1CSR1 register
se_isync

se_blr
```

3.6 SRAM initialization

The internal SRAM features Error Correcting Code (ECC). Because these ECC bits can contain random data after the device is turned on, all SRAM locations must be initialized before being read by application code. Initialization is done by executing 64-bit writes to the entire SRAM block. The value written does not matter at this point, so the Store Multiple Word instruction will be used to write 32 general-purpose registers with each loop iteration. The following code initializes 512 KB of SRAM.

```
# Store number of 128Byte (32GPRs) segments in Counter
    e_lis   r5, _SRAM_SIZE@h   # Initialize r5 to size of SRAM (Bytes)
    e_or2i  r5, _SRAM_SIZE@l
    e_srwi  r5, r5, 0x7        # Divide SRAM size by 128
    mtctr   r5                 # Move to counter for use with "bdnz"

# Base Address of the internal SRAM
    e_lis   r5, _SRAM_BASE_ADDR@h
    e_or2i  r5, _SRAM_BASE_ADDR@l

# Fill SRAM with writes of 32GPRs
sram_loop:
    e_stmw  r0,0(r5)            # Write all 32 registers to SRAM
    e_addi  r5,r5,128           # Increment the RAM pointer to next 128bytes
    e_bdnz  sram_loop           # Loop for all of SRAM
```

3.7 C runtime register setup

The Power Architecture Enhanced Application Binary Interface (EABI) specifies certain general purpose registers as having special meaning for C code execution. At this point in the initialization code, the stack pointer, small data, and small data 2 base pointers are set up. EABI-conformant C compilers will generate code that makes use of these pointers later on.

```
    e_lis   r1, __SP_INIT@h     # Initialize stack pointer r1 to
    e_or2i  r1, __SP_INIT@l     # value in linker command file.

    e_lis   r13, _SDA_BASE_@h   # Initialize r13 to sdata base
    e_or2i  r13, _SDA_BASE_@l   # (provided by linker).

    e_lis   r2, _SDA2_BASE_@h   # Initialize r2 to sdata2 base
    e_or2i  r2, _SDA2_BASE_@l   # (provided by linker).
```

As noted in the comments above, these values are defined in the linker command file for this project.

```
    __DATA_SRAM_ADDR  = ADDR(.data);
    __SDATA_SRAM_ADDR = ADDR(.sdata);

    __DATA_SIZE    = SIZEOF(.data);
```

```
__SDATA_SIZE  = SIZEOF(.sdata);

__DATA_ROM_ADDR  = ADDR(.ROM.data);
__SDATA_ROM_ADDR = ADDR(.ROM.sdata);
```

These values in the internal flash boot case will be used to copy initialized data from flash to SRAM, but first the SRAM must be initialized.

This runtime setup procedure may vary depending on the compiler. Consult your compiler's documentation. There may also be additional setup required for initializing the C standard library.

3.8 Copy initialized data

When booting from flash, the program image stored in flash will contain the various data segments created by the C compiler and linker. Initialized read-write data must be copied from read-only flash to read-writable SRAM before branching to the C main routine.

The following example assumes the initialized data values are stored uncompressed in the flash. Some compilers compress this data to save space in the flash image. The example code to this application note invokes the compiler-dependent _start routine to accomplish the C runtime setup and data copy for core 0. This example is provided as a reference.

```
##--------- Initialized Data - ".data" --------------------------------------
DATACOPY:
    e_lis       r9, __DATA_SIZE@ha      # Load upper SRAM load size
    e_or2i      r9, __DATA_SIZE@l       # Load lower SRAM load size into R9
    e_cmp16i    r9,0                    # Compare to see if equal to 0
    se_beq      SDATACOPY               # Exit cfg_ROMCPY if size is zero
    mtctr       r9                      # Store no. of bytes to be moved in counter

    e_lis       r10, __DATA_ROM_ADDR@h  # Load address of first SRAM load into R10
    e_or2i      r10, __DATA_ROM_ADDR@l  # Load lower address of SRAM load into R10
    se_subi     r10, 1                  # Decrement address

    e_lis       r5, __DATA_SRAM_ADDR@h  # Load upper SRAM address into R5
    e_or2i      r5, __DATA_SRAM_ADDR@l  # Load lower SRAM address into R5
    se_subi     r5, r5, 1               # Decrement address

DATACPYLOOP:
    e_lbzu      r4, 1(r10)             # Load data byte at R10 into R4
    e_stbu      r4, 1(r5)              # Store R4 data byte into SRAM at R5
    e_bdnz      DATACPYLOOP            # Branch if more bytes to load from ROM

##--------- Small Initialised Data - ".sdata" --------------------------------
SDATACOPY:
    e_lis       r9, __SDATA_SIZE@ha     # Load upper SRAM load size
    e_or2i      r9, __SDATA_SIZE@l      # Load lower SRAM load size into R9
    e_cmp16i    r9,0                    # Compare to see if equal to 0
    e_beq       ROMCPYEND               # Exit cfg_ROMCPY if size is zero
    mtctr       r9                      # Store no. of bytes to be moved in counter

    e_lis       r10, __SDATA_ROM_ADDR@h # Load address of first SRAM load into R10
    e_or2i      r10, __SDATA_ROM_ADDR@l # Load lower address of SRAM load into R10
    e_subi      r10,r10, 1              # Decrement address

    e_lis       r5, __SDATA_SRAM_ADDR@h # Load upper SRAM address into R5
    e_or2i      r5, __SDATA_SRAM_ADDR@l # Load lower SRAM address into R5
    e_subi      r5, r5, 1               # Decrement address
SDATACPYLOOP:
    e_lbzu      r4, 1(r10)              # Load data byte at R10 into R4
    e_stbu      r4, 1(r5)               # Store R4 data byte into SRAM at R5
    e_bdnz      SDATACPYLOOP            # Branch if more bytes to load from ROM
ROMCPYEND:
##--------------------------------------------------------------------------
```

3.9 Start core 1

While core 0 begins execution immediately after device reset, core 1 remains held in reset. Its reset input is controlled by the Halt register (SIU_HLT) and the Core 1 Reset Vector register (SIU_RSTVEC1). Clearing either the core 1 HLT bit in SIU_HLT or the RST bit in SIU_RSTVEC1 will de-assert reset to core 1 and allow it to begin execution. When core 1 comes out of reset, it begins executing code at the address specified in the RSTVEC field of register SIU_RSTVEC1. By default, this field points to the address 0xFFFF_FFFC, which is the address of the Boot Assist Module (BAM). [1] SIU_RSTVEC also contains a field to specify whether this start code should be executed in Book E Power Architecture or Variable Length Encoding (VLE) format. If the user wants to supply a different start address, the RSTVEC field must be set before clearing either the HLT or RST bits.

The following example main C routine is used by both cores. It first checks the core's Processor ID Register (PIR) to determine which core is executing the code. PIR will read 0 for core 0 and 1 for core 1. The main routine then branches to core-specific code depending upon the value in PIR. Core 0 begins by starting core 1 and then enters an infinite loop to toggle a GPIO. Core 1 simply toggles a different GPIO.

```
#define RSTVEC_VLE 1
#define RSTVEC_RESET 0x2
#define RSTVEC_RST_MASK 0xFFFFFFFC

extern unsigned long __start;

int main (void)
{
  int pid;

  asm ("li      r0, 0x4000");
  asm ("mthid0  r0");    /* enable TB and decrementer */

  pid = __MFSPR(286);

  if (pid == 0)
  {
    volatile int i;

    SIU.PCR[144].R = ALT0 | OBE; /* PA=0, OBE=1 for GPIO[0] PA[1] */

    /* Start Core 1 in VLE mode */
    SIU.RSTVEC1.R = ((unsigned long)&__start & RSTVEC_RST_MASK) | RSTVEC_VLE;

    while (1)/* loop forever (core 0) */
    {
      i++;
      if (i % 1000000 == 0)
      {
        SIU.GPDO[144].R  ^= 1;
      }
    }
  } else {
    volatile int i;

    SIU.PCR[113].R = ALT0 | OBE;  /* PA=0, OBE=1 for GPIO[0] PA[0] */

    while (1)/* loop forever (core 1) */
    {
```

1. While it is possible to start core 1 from the Boot Assist Monitor address, it is more efficient to supply the start routine address in RSTVEC even if it is the same address as used by core 0. Doing so avoids having BAM perform the Reset Configuration Half-Word search in flash again as well as the setup of the MMU which must be updated anyway in the initialization routine. As noted above, if you do not want the BAM to be executed again for core 1 then you must set a new start address value in the RSTVEC field of register SIU_RSTVEC1 before clearing either the core 1 HLT or RST bits.

```
        i++;
        if (i % 1000000 == 0)
        {
          SIU.GPDO[113].R  ^= 1;
        }
      }
    }
}
```

Because core 1 has been pointed at the same startup procedure as core 0, it is necessary to add similar PIR checks and branches to skip initialization tasks that do not need to be repeated for core 1, such as PLL configuration and SRAM ECC initialization. For example:

```
mfpir    r5          # Check core id
se_cmpi  r5,0
bne      pll_end     # Skip pll init if this is core 1
```

Another important consideration is MMU setup. By default, core 1 will have a single MMU entry out of reset. TLB0 will allow access to the 4 K address space beginning at the start address provided in RSTVEC. The startup code must program the remaining MMU table entries to cover all other address spaces, expanding TLB0 as necessary, or using a different entry to cover additional code execution space.

4 MCU optimization

In this section, the following areas for potential optimization will be discussed:

- Wait states, prefetch, and BIU settings for the flash controller
- Branch target buffer
- Crossbar switch

Other areas of optimization to consider but will not be discussed in this section include:

- using an optimizing compiler.
- making use of the signal processing extension (SPE1.1) instruction support for digital signal processing (DSP).
- using the Enhanced Time Processing Unit (eTPU) for timing.
- making use of the floating point unit (FPU) for floating point calculations.
- enabling the instruction and data caches.

4.1 Flash optimization

The on-chip flash array controller comes out of reset with fail-safe settings. Wait states are set to maximum and performance features like prefetch, read buffering, and pipelining are disabled. These settings can typically be optimized based on the operating frequency, using the information specified in the MPC5777C data sheet. The following code can be modified to select the appropriate value for the flash array's Platform Flash Configuration Registers (PFCRn).

The following example selects the 132 MHz operating settings which accomplish the following optimizations:

- Enable instruction prefetch for both cores on buffer hits and misses
- Enable read buffer
- Reduce read wait states to 3
- Pipelined access to flash is disabled and one wait state is inserted before a subsequent access can be initiated

Because, in this example, the program is executing from flash memory, the user needs to load instructions to perform the update of PFCR1 and PFCR2 into SRAM and temporarily execute from there.

```
#******************************
# Optimize Flash
#******************************
# Code is copied to RAM first, then executed, to avoid executing code from flash
# while wait states are changing.

    se_b copy_to_ram

# settings for 132MHz platform
#
# PFCR1 = 0x00018317
# P0_M8PFE = 0b0        (Core 0 Nexus master pre-fetch disabled)
# P0_M0PFE = 0b1        (Core 0 master pre-fetch enabled)
# P0_MxPFE = 0b0        (All other masters N/A port 0, disable)
# APC = 0b100            (APC disabled and 1 wait state added per Errata 7422)
# RWSC = 0b011          (3 additional hold cycles)
# P0_DPFEN = 0b0        (Port0 data pre-fetch disabled)
# P0_IPFEN = 0b1        (Port0 instruction pre-fetch enabled)
# P0_PFLIM = 0b1x       (prefetch on miss or hit)
# P0_BFEN = 0b1         (read line buffer enabled)
#
# PFCR2 = 0x00020017
# P1_MxPFE = 0b0        (All other masters N/A port 1, disable)
# P1_M9PFE = 0b0        (Core 1 Nexus master pre-fetch disabled)
# P1_M8PFE = 0b0        (Core 0 Nexus pre-fetch N/A port 1, disable)
# P1_M6PFE = 0b0        (SIPI  master pre-fetch disabled)
# P1_M5PFE = 0b0        (eDMA_B master pre-fetch disabled)
# P1_M4PFE = 0b0        (eDMA_A master pre-fetch disabled)
# P1_M3PFE = 0b0        (CSE master pre-fetch disabled)
# P1_M2PFE = 0b0        (FEC master pre-fetch disabled)
# P1_M1PFE = 0b1        (Core 1 master pre-fetch enabled)
# P1_M0PFE = 0b0        (Core 0 master pre-fetch N/A port 1, disable)
# P1_DPFEN = 0b0        (data pre-fetch disabled)
# P1_IPFEN = 0b1        (instruction pre-fetch enabled)
# P1_PFLIM = 0b1x       (prefetch on miss or hit)
# P1_BFEN = 0b1         (read line buffer enabled)
#
flash_opt:
    e_lis   r3,0x0001
    e_or2i  r3,0x8317
    e_lis   r4,0xFFF6
    e_or2i  r4,0x8000
    e_stw   r3,0(r4)        # PFCR1
    e_lis   r3,0x0002
    e_or2i  r3,0x0017
    e_stw   r3,4(r4)        # PFCR2
    se_isync
    msync
    se_blr

copy_to_ram:
    e_lis    r3,flash_opt@h
    e_or2i   r3,flash_opt@l
    e_lis    r4,copy_to_ram@h
    e_or2i   r4,copy_to_ram@l
    subf     r4,r3,r4
    se_mtctr r4
    e_lis    r5,0x4000
    se_mtlr  r5
copy:
    e_lbz   r6,0(r3)
    e_stb   r6,0(r5)
    e_addi  r3,r3,1
    e_addi  r5,r5,1
    e_bdnz  copy
    se_isync
    msync
    se_blrl
```

NOTE

These settings are currently preliminary and subject to change pending characterization of the device.

4.2 Branch target buffer

The MPC5777C Power Architecture cores feature a branch prediction optimization which can be enabled to improve overall performance by storing the results of branches and using those results to predict the direction of future branches at the same location. To initialize the branch target buffer, it is necessary to flash invalidate the buffer and enable branch prediction. This can be accomplished with a single write to the Branch Unit Control and Status Register (BUCSR) in each core.

```
cfg_BTB:
    #---------------------------------------------#
    # Flush and Enable BTB - Set BBFI and BPEN    #
    #---------------------------------------------#
    e_li    r3, 0x0201
    mtspr   1013, r3
    se_isync
```

NOTE

If the application modifies instruction code in memory after this initialization procedure, the branch target buffer may need to be flushed and re-initialized as it may contain branch prediction for the code that previously existed at the modified locations.

4.3 Crossbar switch

In most cases, the crossbar settings can be left at their reset defaults. By knowing certain things about the application behavior and use of different masters on the crossbar, it is possible to customize priorities and use algorithms accordingly to obtain some slight performance improvements. For example, DMA transfers may benefit from a higher priority setting than the CPU load/store when communicating with the peripheral bus. This would prevent DMA transfers from stalling if the CPU were to poll a status register in a peripheral. However, this is a specific case which may not apply for all applications. For details on how to configure the crossbar and the relevant registers, see the Crossbar Switch chapter of the MPC5777C reference manual.

5 Conclusion

This application note has presented some specific recommendations for initializing this device and optimizing some of the settings from their reset defaults. This is a starting point only. Other areas to consider include compiler optimization and efficient use of system resources such as DMA and cache. Consult the MPC5777C reference manual for additional information.

6 Code

The code examples in this section can be found on the Freescale website as a software download.

6.1 init.s file

```
#*************************************************************************
#* FILE: init.s
#*
#* DESCRIPTION:
#* Example init code for MPC5777C. Performs following setup tasks:
#*     1) Sets PLL0 to 200MHz and PLL1 to 264MHz. (may require modification
#.        for desired operating frequency)
#*     2) Configure MMU
#*     3) Add MMU entry to support C stack in cache.
#*     4) Invalidate and enable both instruction and data caches.
#*     5) Enable SPE instructions (GHS compiler will use SPE by default)
#*     6) Intialize ECC bits on all 384K of internal SRAM.
#*     7) Reduce flash wait states. (may require modification of wait
#*        state parameters for desired operating frequency)
#*     8) Enables branch target buffer for performance increase.
#*     9) Lock the stack in cache memory. (included linker file required)
#*     10)Branch to _start in GHS provided crt0.s file to finish setup
#*        of the C environment. _start in crt0.s will call main().
#*=======================================================================
#* UPDATE HISTORY
#* Revision    Author      Date          Description of change
#* 1.0         B. Terry    11/12/2009    Initial version for MPC5674F.
#* 1.1         D. Erazmus  12/14/2010    Ported to MPC5676R.
#* 1.2         D. Erazmus  07/27/2011    Converted to VLE instruction set.
#* 1.3         D. Erazmus  11/08/2011    Fixed error in BIUCR value.
#* 2.0         D. Erazmus  12/12/2013    Ported to MPC5777C.
#  2.1         S. Becerra  06/09/2015    Progressive clock switching enabled.
#*=======================================================================
#* COPYRIGHT (c) Freescale Semiconductor, Inc. 2013
#* All Rights Reserved
#*************************************************************************

    .vle
    .globl __start
    .section .rcw, ax
    .long 0x015a0000
    .long __start

    .section .init,avx             # The "ax" generates symbols for debug

__start:

    mfpir    r5                    # Check core id
    se_cmpi    r5,0
    se_bne  pll_end                # Skip pll init if this is core 1

#****************************
# configure PLL0 to 192MHz (40MHz XOSC reference)
# configure PLL1 to 264MHz (48MHz PLL0 PHI1 output reference)
#****************************

# Disable PLL0
    e_lis   r3,0xFFE7
    e_or2i    r3,0x4000
    e_li    r4,0
    e_stw    r4,0(r3)
# Disable PLL1
    e_stw    r4,0x20(r3)

# Select XOSC as PLL0 source
# Select PHI1 output of PLL0 as PLL1 source
    e_lis    r3,0xC3F9
    e_lis    r4,0x0400
    e_or2i  r4,0x0010
    e_stw    r4,0x9A0(r3)

# Program PLL0
```

```
# RFDPHI1 = 0x0C
# RFDPHI = 0x03
# PREDIV = 0x5
# MFD = 0x48
    e_lis    r3,0xFFE7
    e_or2i    r3,0x4000
    e_lis    r4,0x6003
    e_or2i    r4,0x5048
    e_stw    r4,0x8(r3)

# Wait for stable XOSC
xosc_wait:
    e_lis    r3,0xC3F9
    e_lwz   r4,0xC(r3)      # load SIU_RSR
    e_andi. r4,r4,0x80
    se_beq xosc_wait

# Turn on PLL0 and wait for lock
    e_lis    r3,0xFFE7
    e_or2i    r3,0x4000
    e_li    r4,0x0300
    e_stw    r4,0(r3)
wait_for_pll0:
    e_lwz   r4,0x4(r3)      # load PLLDIG_PLL0SR
    e_andi. r4,r4,0x4
    se_beq  wait_for_pll0

# Program PLL1
# RFDPHI = 0x02
# MFD = 0x16
    e_lis    r4,0x0002
    e_or2i    r4,0x0016
    e_stw    r4,0x28(r3)

# Turn on PLL1 and wait for lock
    e_li    r4,0x0300
    e_stw    r4,0x20(r3)
wait_for_pll1:
    e_lwz    r4,0x24(r3)
    e_andi.    r4,r4,0x4
    se_beq    wait_for_pll1

#******************************
# configure PCS
#******************************

# Configure PCS Switch Duration
# SDUR = 0x10
    e_lis   r3,0xFFE7
    e_lis   r4,0x1000
    e_stw   r4,0x700(r3)

# Configure PLL1 Divider Change Register
# INIT = 0x4DF
# RATE = 0x30
    e_lis   r3,0xFFE7
    e_lis   r4,0x04DF
    e_or2i  r4,0x0030
    e_stw   r4,0x710(r3)

# Configure PLL1 Divider End Register
# DIVE = 0x4073
    e_lis   r3,0xFFE7
    e_li    r4,0x4073
    e_stw   r4,0x714(r3)

# Configure PLL1 Divider Start Register
# DIVS = 0x45B7
    e_lis   r3,0xFFE7
    e_li    r4,0x45B7
```

```
    e_stw   r4,0x718(r3)

#******************************
# Select clock dividers and sources
#   PLL0 ref = XOSX
#   PLL1 ref = PLL0 PHI1
#   PERCLKSEL = PLL1
#   PERDIV = Divide-by-2
#   MCANSEL = XOSC
#   SYSCLKSEL = PLL1
#   ETPUDIV = Divide-by-1
#   SYSCLKDIV = Divide-by-1
#   PCS = Enabled
#******************************
    e_lis    r3,0xC3F9
    e_lis    r4,0x0500
    e_or2i    r4,0x2111
    e_stw    r4,0x9A0(r3)

pll_end:

#*****************************
# configure the MMU
#*****************************
# Note 1: When core 1 is running this same code it may not have executed BAM at all.
#
# Note 2: configure TLB1 and TLB3 first before TLB0 since core 1 may be running
# from RAM or Flash with the default 4k TLB0 page out of reset.
#
# Note 3: Flash TLB covers non-contiguous flash array space and so is larger than 8M.
#
#TLB1 = internal flash @ 0x0000_0000, VLE
    e_lis   r3,0x1001
    mtspr   mas0,r3
    e_lis   r3,0xC000
    e_or2i  r3,0x0700
    mtspr   mas1,r3
    e_lis   r3,0x0000
    e_or2i  r3,0x0020
    mtspr   mas2,r3
    e_lis   r3,0x0000
    e_or2i  r3,0x003F
    mtspr   mas3,r3
    msync     # Synchronize in case running from flash
    tlbwe
    se_isync  # Synchronize in case running from flash

#TLB3 = internal SRAM @ 0x4000_0000, VLE, Write-Through Cache
    e_lis   r3,0x1003
    mtspr   mas0,r3
    e_lis   r3,0xC000
    e_or2i  3,0x0480
    mtspr   mas1,r3
    e_lis   r3,0x4000
    e_or2i  r3,0x0030
    mtspr   mas2,r3
    e_lis   r3,0x4000
    e_or2i  r3,0x003F
    mtspr   mas3,r3
    msync     # Synchronize in case running from SRAM
    tlbwe
    se_isync  # Synchronize in case running from SRAM

#TLB0 = pbridgeB @ 0xFFE0_0000, Cache inhibited, Guarded
    e_lis   r3,0x1000
    mtspr   mas0,r3
    e_lis   r3,0xC000
    e_or2i  r3,0x0580
    mtspr   mas1,r3
    e_lis   r3,0xFFE0
```

```
    e_or2i  r3,0x000A
    mtspr   mas2,r3
    e_lis   r3,0xFFE0
    e_or2i  r3,0x003F
    mtspr   mas3,r3
    tlbwe

#TLB2 = external bus @ 0x2000_0000, VLE
    e_lis   r3,0x1002
    mtspr   mas0,r3
    e_lis   r3,0xC000
    e_or2i  r3,0x0700
    mtspr   mas1,r3
    e_lis   r3,0x2000
    e_or2i  r3,0x0020
    mtspr   mas2,r3
    e_lis   r3,0x0000
    e_or2i  r3,0x003F
    mtspr   mas3,r3
    tlbwe

#TLB4 = pbridgeA @ 0xC3E0_0000, Cache inhibited, Guarded
    e_lis   r3,0x1004
    mtspr   mas0,r3
    e_lis   r3,0xC000
    e_or2i  r3,0x0580
    mtspr   mas1,r3
    e_lis   r3,0xC3E0
    e_or2i  r3,0x000A
    mtspr   mas2,r3
    e_lis   r3,0xC3E0
    e_or2i  r3,0x003F
    mtspr   mas3,r3
    tlbwe

#TLB5 = 4k stack for each core (will be locked in cache)
# (Note: located just after 512k TLB3 entry for SRAM)
    e_lis   r3,0x1005
    mtspr   mas0,r3
    e_lis   r3,0xC000
    e_or2i  r3,0x0180
    mtspr   mas1,r3
    e_lis   r3,0x4008
    e_or2i  r3,0x0000
    mtspr   mas2,r3
    e_lis   r3,0x4008
    e_or2i  r3,0x003F
    mtspr   mas3,r3
    tlbwe

#******************************
# invalidate and enable the data and instruction caches
#******************************
# data cache
    e_lis   r3,0x0010
    e_or2i  r3,0x0003
    mtspr   l1csr0,r3
# inst cache
    e_lis   r3,0x0
    e_or2i  r3,0x0003
    mtspr   l1csr1,r3

#*****************************
# Enable SPE
#*****************************
    mfmsr    r6
    e_or2is    r6, 0x0200
    mtmsr    r6

#*****************************
```

```
# initialize 512k SRAM
# (core 0 only)
#*****************************
    mfpir    r5                    # Check core id
    se_cmpi    r5,0
    se_bne    flashopt_end        # Skip sram init and flash optimization if this is core 1

# Store number of 128Byte (32GPRs) segments in Counter
    e_lis    r5, _SRAM_SIZE@h    # Initialize r5 to size of SRAM (Bytes)
    e_or2i   r5, _SRAM_SIZE@l
    e_srwi   r5, r5, 0x7           # Divide SRAM size by 128
    mtctr    r5                   # Move to counter for use with "bdnz"

# Base Address of the internal SRAM
    e_lis    r5, _SRAM_BASE_ADDR@h
    e_or2i   r5, _SRAM_BASE_ADDR@l

# Fill SRAM with writes of 32GPRs
sram_loop:
    e_stmw   r0,0(r5)            # Write all 32 registers to SRAM
    e_addi   r5,r5,128            # Increment the RAM pointer to next 128bytes
    e_bdnz   sram_loop            # Loop for all of SRAM

sram_end:

#*******************************
# Optimize Flash
#*******************************
# Code is copied to RAM first, then executed, to avoid executing code from flash
# while wait states are changing.

    se_b copy_to_ram

# settings for 132MHz platform
#
# PFCR1 = 0x00018317
# P0_M8PFE = 0b0        (Core 0 Nexus master pre-fetch disabled)
# P0_M0PFE = 0b1        (Core 0 master pre-fetch enabled)
# P0_MxPFE = 0b0        (All other masters N/A port 0, disable)
# APC = 0b100            (APC disabled and 1 wait state added per Errata 7422)
# RWSC = 0b011          (3 additional hold cycles)
# P0_DPFEN = 0b0        (Port0 data pre-fetch disabled)
# P0_IPFEN = 0b1        (Port0 instruction pre-fetch enabled)
# P0_PFLIM = 0b1x       (prefetch on miss or hit)
# P0_BFEN = 0b1         (read line buffer enabled)
#
# PFCR2 = 0x00020017
# P1_MxPFE = 0b0        (All other masters N/A port 1, disable)
# P1_M9PFE = 0b0        (Core 1 Nexus master pre-fetch disabled)
# P1_M8PFE = 0b0        (Core 0 Nexus pre-fetch N/A port 1, disable)
# P1_M6PFE = 0b0        (SIPI  master pre-fetch disabled)
# P1_M5PFE = 0b0        (eDMA_B master pre-fetch disabled)
# P1_M4PFE = 0b0        (eDMA_A master pre-fetch disabled)
# P1_M3PFE = 0b0        (CSE master pre-fetch disabled)
# P1_M2PFE = 0b0        (FEC master pre-fetch disabled)
# P1_M1PFE = 0b1        (Core 1 master pre-fetch enabled)
# P1_M0PFE = 0b0        (Core 0 master pre-fetch N/A port 1, disable)
# P1_DPFEN = 0b0        (data pre-fetch disabled)
# P1_IPFEN = 0b1        (instruction pre-fetch enabled)
# P1_PFLIM = 0b1x       (prefetch on miss or hit)
# P1_BFEN = 0b1         (read line buffer enabled)
#
flash_opt:
    e_lis    r3,0x0001
    e_or2i   r3,0x8317
    e_lis    r4,0xFFF6
    e_or2i   r4,0x8000
    e_stw    r3,0(r4)        # PFCR1
    e_lis    r3,0x0002
    e_or2i   r3,0x0017
```

```
    e_stw   r3,4(r4)        # PFCR2
    se_isync
    msync
    se_blr

copy_to_ram:
    e_lis    r3,flash_opt@h
    e_or2i   r3,flash_opt@l
    e_lis    r4,copy_to_ram@h
    e_or2i   r4,copy_to_ram@l
    subf     r4,r3,r4
    se_mtctr r4
    e_lis    r5,0x4000
    se_mtlr  r5
copy:
    e_lbz   r6,0(r3)
    e_stb   r6,0(r5)
    e_addi  r3,r3,1
    e_addi  r5,r5,1
    e_bdnz  copy
    se_isync
    msync
    se_blrl

flashopt_end:

#******************************
# enable BTB
#******************************
    e_li    r3, 0x0201
    mtspr   1013, r3
    se_isync

#*******************************
# lock the stack into cache and set stack pointer (core 1)
#*******************************
    mfpir    r5                    # Check core id
    se_cmpi     r5,0
    se_beq     stack_cache_0

stack_cache_1:
    e_lis      r3,__STACK_SIZE_1@h
    e_or2i     r3,__STACK_SIZE_1@l
    se_srwi    r3,5              # Shift the contents of R5 right by 5 bits (size/32)
    se_mtctr   r3
    e_lis      r3,__SP_END_1@h
    e_or2i     r3,__SP_END_1@l

lock_cache_loop_1:
    dcbz       r0,r3             # Establish address in cache for 32 bytes and zero
    dcbtls     0,r0,r3           # Lock the address into the cache
    se_addi    r3,32             # Increment to start of next cache line (+32 bytes)
    e_bdnz     lock_cache_loop_1 # Decrement the counter (CTR), branch if nonzero
    e_lis      r1,(__SP_INIT_1-0x10)@h
    e_or2i     r1,(__SP_INIT_1-0x10)@l

##--------------- Set up stack and run time environment Core 1 ---------
    e_lis   r1, __SP_INIT_1@h   # Initialize stack pointer r1 to
    e_or2i  r1, __SP_INIT_1@l   # value in linker command file.

    e_lis   r13, _SDA_BASE_@h   # Initialize r13 to sdata base
    e_or2i  r13, _SDA_BASE_@l   # (provided by linker).

    e_lis   r2, _SDA2_BASE_@h   # Initialize r2 to sdata2 base
    e_or2i  r2, _SDA2_BASE_@l   # (provided by linker).

    e_stwu  r0,-64(r1)            # Terminate stack.

    e_b     main
```

```
#******************************
# lock the stack into cache and set stack pointer (core 0)
#******************************
stack_cache_0:
    e_lis     r3,__STACK_SIZE_0@h
    e_or2i    r3,__STACK_SIZE_0@l
    se_srwi   r3,5              # Shift the contents of R5 right by 5 bits (size/32)
    se_mtctr  r3
    e_lis     r3,__SP_END_0@h
    e_or2i    r3,__SP_END_0@l

lock_cache_loop_0:
    dcbz      r0,r3             # Establish address in cache for 32 bytes and zero
    dcbtls    0,r0,r3           # Lock the address into the cache
    se_addi   r3,32             # Increment to start of next cache line (+32 bytes)
    e_bdnz    lock_cache_loop_0 # Decrement the counter (CTR), branch if nonzero
    e_lis     r1,(__SP_INIT_0-0x10)@h
    e_or2i    r1,(__SP_INIT_0-0x10)@l

#******************************
# call ghs init code (_start in crt0.s) This
# call to the GHS code insures heap etc. are
# configured and intialized correctly.
#******************************
    e_b _start

# hang if here
loop_forever:
    se_b loop_forever
```

6.2 main.c file

```
/*************************************************************************/
/* FILE: main.c                                                          */
/*                                                                       */
/* DESCRIPTION:                                                          */
/* Example project for MPC5777C.                                         */
/*=======================================================================*/
/* UPDATE HISTORY                                                        */
/* Revision    Author      Date          Description of change           */
/* 1.0         D. Erazmus  12/13/2010    Initial version for MPC5676R.   */
/* 2.0         D. Erazmus  12/13/2013    Adapted for MPC5777C.           */
/*=======================================================================*/
/* COPYRIGHT (c) Freescale Semiconductor, Inc. 2013                      */
/* All Rights Reserved                                                   */
/*************************************************************************/

/**************************************************
INCLUDE FILES
**************************************************/
#include <ppc_ghs.h>

#include "MPC577xC.h"
#include "siu.h"

#define RSTVEC_VLE         1
#define RSTVEC_RESET     0x2
#define RSTVEC_RST_MASK 0xFFFFFFFC

/**************************************************
Global vars
**************************************************/
extern unsigned long __start;

/**************************************************/
/*                      MAIN                      */
```

```
/*************************************************/
int main (void)
{
    int pid;

    asm ("e_li    r0, 0x4000");
    asm ("mthid0  r0");                 /* enable TB and decrementer */

    pid = __MFSPR(286);

    if (pid == 0)
    {
        volatile int i;

        SIU.PCR[144].R = ALT0 | OBE;    //PA=0, OBE=1 for GPIO[0] PA[1]

        /* Start Core 1 in VLE mode */
        SIU.RSTVEC1.R = ((unsigned long)&__start & RSTVEC_RST_MASK) | RSTVEC_VLE;

        while (1)    /* loop forever (core 0) */
        {
            i++;
            if (i % 1000000 == 0)
            {
                SIU.GPDO[144].R  ^= 1;
            }
        }
    } else {
        volatile int i;

        SIU.PCR[113].R = ALT0 | OBE;    //PA=0, OBE=1 for GPIO[0] PA[0]

        while (1)    /* loop forever (core 1) */
        {
            i++;
            if (i % 1000000 == 0)
            {
                SIU.GPDO[113].R  ^= 1;
            }
        }
    }

}   /* end of main */
```

6.3 Linker definition file

```
//*************************************************************************
// FILE: standalone_romrun.ld                                             *
//                                                                        *
// DESCRIPTION:                                                           *
// Linker definition file for MPC5777C project                           *
//=======================================================================*
// UPDATE HISTORY                                                         *
// Revision    Author      Date          Description of change            *
// 1.0         D. Erazmus  12/13/2010    Initial version forMPC5676R.     *
// 2.0         D. Erazmus  12/13/2013    Adapted for MPC5777C.           *
//=======================================================================*
// COPYRIGHT (c) Freescale Semiconductor, Inc. 2013                       *
// All Rights Reserved                                                    *
//*************************************************************************

DEFAULTS {
    SRAM_SIZE       = 512K
    SRAM_MMU_SIZE   = 512K
    SRAM_BASE_ADDR  = 0x40000000
```

```
    STACK_SIZE_0    = 4K                        // 4KB Stack for core 0
    STACK_SIZE_1    = 4k                        // 4KB Stack for core 1

    stack_reserve = 4k
    heap_reserve  = 4k
}

MEMORY {

//  6M Internal Flash
    flash_rsvd1   : ORIGIN = 0x00000000, LENGTH = 8
    flash_memory1 : ORIGIN = .,          LENGTH = 256K-8
    flash_rsvd2   : ORIGIN = 0x00800000, LENGTH = 8
    flash_memory2 : ORIGIN = .,          LENGTH = 8M-8


// 512KB of internal SRAM starting at 0x40000000
    dram_rsvd1   : ORIGIN = 0x40000000, LENGTH = 0x200
    dram_reset   : ORIGIN = .,          LENGTH = 0
    dram_memory  : ORIGIN = .,          LENGTH = SRAM_SIZE-0x200
    dram_rsvd2   : ORIGIN = .,          LENGTH = 0

// 4k of stack per core to be locked in cache
    stack_ram0   : ORIGIN = SRAM_BASE_ADDR+SRAM_MMU_SIZE,              LENGTH = STACK_SIZE_0
    stack_ram1   : ORIGIN = SRAM_BASE_ADDR+SRAM_MMU_SIZE+STACK_SIZE_0, LENGTH = STACK_SIZE_1
}

//
// Program layout for starting in ROM, copying data to RAM,
// and continuing to execute out of ROM.
//

SECTIONS
{

//
// RAM SECTIONS
//
    .PPC.EMB.sdata0                                 ABS : > dram_memory
    .PPC.EMB.sbss0                            CLEAR ABS : > .

    .sdabase                                  ALIGN(16) : > dram_memory
    .sdata                                              : > .
    .sbss                                               : > .
    .data                                               : > .
    .bss                                                : > .
    .heap                   ALIGN(16) PAD(heap_reserve) : > .
    .__exception_handlers                     ALIGN(4k) : > .

    .stack ALIGN(16) PAD(STACK_SIZE_0)  : {} > stack_ram0        // Stack Area
    .stack1 ALIGN(16) PAD(STACK_SIZE_1) : {} > stack_ram1        // Stack Area

//
// ROM SECTIONS
//

    .rcw                          NOCHECKSUM   : > flash_rsvd2
    .init                                      : { *(.init) } > flash_memory2
    .text                                      : > .
    .vletext                                   : > .
    .syscall                                   : > .

    .rodata                                    : > .
    .sdata2                                    : > .

    .secinfo                                   : > .
    .fixaddr                                   : > .
    .fixtype                                   : > .

    .CROM.PPC.EMB.sdata0   CROM(.PPC.EMB.sdata0) : > .
```

```
    .CROM.sdata                     CROM(.sdata) : > .
    .CROM.data                       CROM(.data) : > .

    /* Stack Address Parameters */
    __SP_INIT       = ADDR(.stack)  + SIZEOF(stack_ram0);
    __SP_INIT_0     = ADDR(.stack)  + SIZEOF(stack_ram0);
    __SP_INIT_1     = ADDR(.stack1) + SIZEOF(stack_ram1);

    __SP_END        = ADDR(.stack);
    __SP_END_0      = ADDR(.stack);
    __SP_END_1      = ADDR(.stack1);

    __STACK_SIZE    = SIZEOF(stack_ram0);
    __STACK_SIZE_0  = SIZEOF(stack_ram0);
    __STACK_SIZE_1  = SIZEOF(stack_ram1);

    _SRAM_BASE_ADDR = ADDR(dram_rsvd1);
    _SRAM_SIZE      = SIZEOF(dram_rsvd1) + SIZEOF(dram_reset) + SIZEOF(dram_memory) +
SIZEOF(dram_rsvd2);

//
// These special symbols mark the bounds of RAM and ROM memory.
// They are used by the MULTI debugger.
//
    __ghs_ramstart  = MEMADDR(dram_rsvd1);
    __ghs_ramend    = MEMENDADDR(dram_memory);
    __ghs_romstart  = MEMADDR(flash_rsvd1);
    __ghs_romend    = MEMENDADDR(flash_rsvd2);

//
// 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(.text);
    __ghs_rombootcodeend    = ENDADDR(.fixtype);
}
```

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Information in this document is provided solely to enable system and software implementers to use Freescale products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document. Freescale reserves the right to make changes without further notice to any products herein.

Freescale makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters that may be provided in Freescale data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including "typicals," must be validated for each customer application by customer's technical experts. Freescale does not convey any license under its patent rights nor the rights of others. Freescale sells products pursuant to standard terms and conditions of sale, which can be found at the following address: freescale.com/SalesTermsandConditions.

Freescale and the Freescale logo are trademarks of Freescale Semiconductor, Inc., Reg. U.S. Pat. & Tm. Off. The Power Architecture and Power.org word marks and the Power and Power.org logos and related marks are trademarks and service marks licensed by Power.org. All other product or service names are the property of their respective owners.

© 2015 Freescale Semiconductor, Inc.

Document Number AN5191
Revision 0, 09/2015