Freescale Semiconductor
Application Note

Document Number: AN4097
Rev. 0, 04/2010

MPC5674F Software Initialization and Optimization

by: David Erazmus
32-bit Automotive Applications Engineering
Microcontroller Solutions Group

1 Introduction

This application note describes a recommended software initialization procedure for the MPC5674F 32-bit Power Architecture® automotive microcontroller. This covers the Power Architecture core, memory management unit (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.

Contents

freescale™ semiconductor

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 BookE Instructions. The assembly code examples shown in this application note will be using VLE mnemonics and syntax but can easily be translated into the BookE variant.

3 Startup Code

The first part of our initialization procedure executes from the reset vector or program entry point and performs the minimal setup needed to prepare for C code execution later on. 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. Enable Cache
3. Program PLL
4. Initialize SRAM
5. Initialize C Runtime Environment

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 if desired. 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 typically accomplished by the debugger using configuration scripts.

We will focus on the internal flash boot case in this application note since 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

Block	Address
0	0x0000_0000
1	0x0000_4000
2	0x0001_0000
3	0x0001_C000
4	0x0002_0000
5	0x0003_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
				SWT	WTE	PS0	VLE	0	1	0	1	1	0	1	0
								Boot Identifier = 0x5A							

The RCHW occupies the most significant 16 bits of the first 32-bit internal memory word at the boot location. The word immediately following this contains the boot vector address. After applying the RCHW, the BAM will branch to this address. During software initialization we can 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 is generated with a valid RCHW encoding and the start address symbol for code entry point.

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

In the above example, the core and software watchdog timers are both disabled. These can both operate independently, but it is typical to use just one or the other in an application. When debugging, the RCHW is not applied as the BAM does not execute, so it is up to the debugger to disable these timers so that they do not interfere with application debug sessions. This is necessary for the core watchdog as software cannot be disabled it 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 Instruction Cache and PLL

Enabling the instruction cache will reduce execution time for the remainder of the initialization procedure and, of course, improve application execution speed later on. Enabling the cache after the PLL is programmed, can increase peak current draw during the setup routine, so it is recommended that the cache be enabled first. However, when we are executing code from internal flash it is also a good idea to inhibit the flash space, initially, until after PLL programming. The reason for this is the polling loop that tests for PLL lock. With the cache enabled this loop can execute very quickly and draw more current. There is not a real need for the polling loop to execute that quickly at this point, so the Memory Management Unit will be used to first inhibit the flash region from caching and then enable the cache.

3.2.1 Memory Management Unit

The BAM includes a default setup for the MMU which allows access to all device internal resources with no address translation. Variable Length Encoding (VLE) for instructions will be enabled for SRAM and Flash regions if enabled in the RCHW. This default configuration may be sufficient for most applications but as mentioned above one change is made to inhibit caching in the flash until after PLL lock. By default, the BAM sets up MMU table index 1 for the internal flash region with a size of 16MB. The cache-inhibit control bit is set and during this, reduce the size to 4MB to cover just the user flash region for this device.

```
# Set up MMU
    # MAS0 : ESEL=1
    # MAS1 : TSIZ=4Mbytes
    # MAS2 : EPN=0x000000028, VLE=1, W=0, I=1, M=0, G=0, E=big
    # MAS3 : RPN=0x000000000, PERMIS=all

    e_lis   r3,0x1001
    mtspr   mas0,r3

    e_lis   r4,0xC000
    e_or2i  r4,0x0600
    mtspr   mas1,r4

    e_lis   r5,0x0000
    e_or2i  r5,0x0028
    mtspr   mas2,r5

    e_lis   r6,0x0000
    e_or2i  r6,0x003f
    mtspr   mas3,r6

    msync     # Synchronize for running out of flash
    tlbwe
    se_isync  # Synchronize for running out of flash
```

Note that in this example, tlbwe is preceded by msync and followed by isync. These synchronization steps are taken when we are executing code from the region being modified.

3.2.2 Enable Cache

The core instruction and data caches are enabled through the L1 Cache Control and Status Registers 1 & 2 (L1CSR0 and L1CSR1). At this point just the instruction cache is enabled since nothing is being done in the initialization routines that would benefit from data cache yet. The instruction cache is invalidated and enabled by setting the ICINV and ICE bits in L1CSR1. The cache invalidate operation takes some time and can be interrupted or aborted. Since nothing else is going on at this point in the boot-up procedure this is not going to happen here and we could just get by setting these bits and moving on. For future reference, however, the code shown here 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.

```
cfg_ICACHE:

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

    #-------------------------------------------#
    # 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
```

```
    e_beq   NO_ABORT
    #-------------------------------------------#
    # If abort detected, clear ICABT bit and    #
    # re-run invalidation                       #
    #-------------------------------------------#
    se_and. r7, r3
    mtspr l1csr1, r10
    se_b    cfg_ICACHE

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

    #-------------------------------------------#
    # Enable cache 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)
    se_isync
    msync
    mtspr l1csr1, r5    # Write R5 to L1CSR1 register

    se_blr
```

3.2.3 Programming the PLL

The FMPLL module contains the frequency modulated phase lock loop (FMPLL), enhanced frequency divider (ERFD), enhanced synthesizer control registers (ESYNCR1 and ESYNCR2), synthesizer status register (SYNSR), and clock/PLL control logic. The block also contains a reference frequency pre-divider controlled by the EPREDIV bits in the ESYNCR1. This enables the use of a high frequency crystal or external clock generator to obtain finer frequency synthesis resolution than would be available if the raw input clock were used directly by the analog loop.

The FMPLL on this device can synthesize clock frequencies ranging from 48 to 148 times the reference frequency of the predivider output. The post-divider can reduce this output frequency without forcing a re-lock. In normal operation, the following equation can be used to calculate the programming values for the FMPLL:

$$F_{sys} = F_{extal} \times (EMFD + 16) / ((EPREDIV + 1)(EFRD + 1)) \qquad \textit{Eqn. 1}$$

The following example sets up the PLL to produce a 264MHz system clock assuming a 40MHz reference crystal.

```
# ESYNCR1
        e_lis   r3, 0xC3F8
        e_lis   r4, 0x0004    # EPREDIV
```

```
        e_or2i  r4, 0x0032    # EMFD
        e_stw   r4, 8(r3)

# ESYNCR2
        e_lis   r4, 0x0000
        e_or2i  r4, 0x0001    # ERFD
        e_stw   r4, 12(r3)

wait_for_lock:
        e_lwz   r5, 4(r3)     # load SYNSR
        e_andi. r5, r5, 0x8
        beq wait_for_lock
```

Now that we've locked on our new clock rate, we can enable caching for the flash region.

```
    # Enable caching of this region
    # MAS0 : ESEL=1
    # MAS1 : TSIZ=4Mbytes
    # MAS2 : EPN=0x000000020, VLE=1, W=0, I=0, M=0, G=0, E=big
    # MAS3 : RPN=0x000000000, PERMIS=all

    e_lis   r3,0x1001
    mtspr   mas0,r3

    e_lis   r4,0xC000
    e_or2i  r4,0x0600
    mtspr   mas1,r4

    e_lis   r5,0x0000
    e_or2i  r5,0x0020
    mtspr   mas2,r5

    e_lis   r6,0x0000
    e_or2i  r6,0x003f
    mtspr   mas3,r6

    msync     # Synchronize for running out of flash
    tlbwe
    se_isync  # Synchronize for running out of flash
```

3.3 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. In the initialization code at this point 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 our 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);
```

The 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.

3.4 SRAM Initialization

The internal SRAM features Error Correcting Code (ECC). Because these ECC bits can contain random data after the device is powered on, all SRAM locations must be initialized before being read by application code. This 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 at a time.

```
# 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
```

3.5 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 we branch to our C main routine.

```
##--------- 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
    e_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
    e_subi      r10,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
    e_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:
##-------------------------------------------------------------------------
```

3.6 C Code Execution

At this point in the procedure C code execution can be started. Before branching to our application main routine, however, there are some additional microcontroller setup and optimization steps to perform. These are described in the next sections.

```
##--------- Start of Main Code-------------------------------------------------
# Start of MCU initialization code
    e_bl      init_MCU
# Start of optimisation code
    e_bl      optimise_MCU
# Start of main code
    e_bl      main
##-------------------------------------------------------------------------
```

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
- Data Cache
- Branch Target Buffer
- Crossbar Switch

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 MPC5674F data sheet. The following code can be modified to select the appropriate value for the flash array's Bus Interface Unit Control Register (BIUCR).

The example below selects the 264MHz operating settings which accomplish the following optimizations:

- Enable instruction prefetch for all masters on buffer hits and misses
- Enable read buffer
- Reduce read wait states to 3
- Enable pipelining with 3 hold cycles between access requests
- Reduce write wait states to 1

```
cfg_FLASH:
    #-------------------------------------------#
    # Save Link Register as this will be modified#
    #-------------------------------------------#
    mflr  r3
    #-------------------------------------------#
    # Load Flash BIUCR Setting into R7           #
    ############################################
    ## CHANGE FOR DIFFERENT Fsys/Fplat:         ##
    ##  Up to 264MHz/132MHz : 0x01716B15        ##
    ##  Up to 200MHz/100MHz : 0x01714A15       ##
    ##  Up to 180MHz/90MHz  : 0x01714A15        ##
    ##  Up to 132MHz/132MHz : 0x01716B15        ##
    ############################################
    #-------------------------------------------#
    e_lis   r7, 0x0171
    e_or2i  r7, 0x6B15
    #-------------------------------------------#
    # Load Flash BIUCR Address into R6           #
    #-------------------------------------------#
    e_lis   r6, 0xC3F8
    e_or2i  r6, 0x801C
```

Since in this example we are executing from flash memory, we need to load instructions to perform the update of BIUCR into SRAM and then execute from there temporarily.

If desired, SIU_SYSDIV[IPCLKDIV] can also be changed here to affect the platform/peripheral frequency at which the flash array operates. This should be done before changing BIUCR settings. The resulting flash clock rate should be checked against the MPC5674F data sheet to determine appropriate BIUCR values. Here, the example assumes the default divider of 2, so the system clock is 264MHz and the platform clock is 132MHz.

```
SRAMLOAD:
    #-------------------------------------------#
    # Load BIUCR write instruction into R8, R9 & #
    # R10                                                                  #
    #-------------------------------------------#
    e_lis   r8, 0x54E6
```

```
e_or2i  r8, 0x0000    # R8 = "e_stw r7, 0x0(r6)"
e_lis   r9, 0x4C00
e_or2i  r9, 0x012C    # R9 = "isync"
e_lis   r10, 0x0004
e_or2i  r10, 0x0004   # R10 = ""se_blr,se_blr"

#-------------------------------------------#
# Load RAM address into R11                 #
#-------------------------------------------#
e_lis     r11, _BIUCR_RAM_ADDR@h
e_add16i  r11,r11, _BIUCR_RAM_ADDR@l

#-------------------------------------------#
# Store Instructions in RAM, then branch and #
# execute instructions to setup BIUCR       #
#-------------------------------------------#
e_stw   r8, 0x0(r11);
e_stw   r9, 0x4(r11);
e_stw   r10, 0x8(r11);
mtlr  r11
se_blrl
```

NOTE

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

4.2 Data Cache

Earlier the instruction cache was enabled to speed up execution of the initialization procedure. Before the user application is executed, the data cache will be enabled. The procedure for this is more or less the same as for the instruction cache, taking care to ensure the invalidate operation does not abort. One additional step is selecting the cache write mode (DCWM). When set to write-through mode, the “W” page attribute from the MMU is ignored and all write accesses write through the cache. When set to copy-back mode, write accesses only write through the cache for MMU regions marked with the “W” page attribute. Otherwise the write data is stored in the cache and remains there until that cache line is flushed to memory.

Copy-back mode is generally recommended for performance, however care must be taken when sharing data buffers between CPU, DMA, and peripherals. Write-through mode will eliminate coherency issues when writing data from the CPU to these shared buffers, but would still require invalidation of the buffer’s addresses from the CPU’s data cache when accepting data back from DMA or peripheral modules. The MMU can be used to create a region of write-through or cache-inhibited space for such shared memory needs.

4.3 Branch Target Buffer

MPC5674F Power Architecture core features a branch prediction optimization which can be enabled to improve overall performance by storing the results of branches and using that to predict the direction of future branches at the same location. To initialize it, we need 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 the core.

```
cfg_BTB:
```

```
#-------------------------------------------#
# Flush and Enable BTB - Set BBFI and BPEN  #
#-------------------------------------------#
e_lis   r3, 0x0
e_or2i  r3, 0x0201
mtspr 1013, r3
```

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.

4.4 Crossbar Switch

For the most part, the crossbar settings can be left at their reset defaults. It is possible, knowing certain things about the application behavior and use of different masters on the crossbar, to customize priorities and using algorithms accordingly and 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. Again, however, this is a specific case which may not apply for all applications.

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 look at include compiler optimization and efficient use of system resources such as DMA and cache. Consult the MPC5674F reference manual for additional information.

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor, Inc.
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
+1-800-521-6274 or +1-480-768-2130
www.freescale.com/support

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor China Ltd.
Exchange Building 23F
No. 118 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 10 5879 8000
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Document Number: AN4097
Rev. 0
04/2010

freescale™
semiconductor