Freescale Semiconductor
Application Note

Document Number: AN4239
Rev. 0, 12/2010

Getting started with the MPC564xB/C Microcontroller

by: Alasdair Robertson
Applications Engineering, Microcontroller Solutions Group
East Kilbride
UK

1 Introduction

The MPC564xB/C family of devices are targeted for automotive body applications, specifically gateway. These dual core architecture devices contain an e200z4 and e200z0 core, compliant with the Power architecture® standard.

The dual core architecture offers increased performance and flexibility while maintaining low operating frequencies and power consumption.

These devices feature up to 3 MB of internal flash and up to 256 KB of internal SRAM memory.

The MPC564xB/C differs from previous MPC56xx devices by being the first such device to feature the e200z4 core and the e200z0 in a dual core configuration. These differences mean that the initialization and configuration is different to the previous parts in this device family.

This application note details the steps required to correctly initialize the MPC564xB/C from reset as well as how to control the second core. Example code is used throughout the application note to explain the steps. Additionally, the full example code is listed in the appendix.

It is intended that this application note is read along with the MPC564xB/C Reference manual which can be obtained from the Freescale website at http://www.freescale.com.

Contents

2 Power on sequence

Before looking at detailed configuration steps, it is important to understand what happens when the device boots from a power on reset.

As with similar devices within the MPC56xx family, there are two hardware boot configuration pins, FAB and ABS. These pins are used to determine the boot sequence as per Table 1. In this application note, the focus will be on the default operating mode, flash boot mode.

Table 1. Boot mode selection

Mode	FAB	ABS
Flash Boot (default mode)	0	x
Serial Boot (LIN)	1	0
Serial Boot (CAN)	1	1

When the device is powered on, the SSCM (System Status Configuration Module) searches the pre-determined locations in flash for a valid RCHW (Reset Configuration Half Word). If this is found, then the e200z4 core reset vector is set to the address contained in the location after the RCHW. If a valid RCHW is not located, then the e200z4 core reset vector is set to the start of the BAM (Boot Assist Module) code at address 0xFFFF_C000. The BAM then starts to configure the device for static mode by enabling the SWT (Software Watchdog Timer) and then putting the core into Wait mode. In all these conditions, the e200z0 core remains held in reset. Note that static mode differs from safe mode. Static mode is a non-operational mode which can only be entered via the BAM.

The SSCM also configures the MMU (Memory Management Unit) to validate a small 4 KB section of memory to be accessible at the e200z4 reset vector address in order to allow a successful boot. This is discussed in more detail in MMU Considerations.

Figure 1 shows a diagram of the initialization sequence which will be further discussed in this application note. The sequence is broken down into automated tasks carried out by the hardware, tasks that are carried out in software initialization code as well as in the main user application.

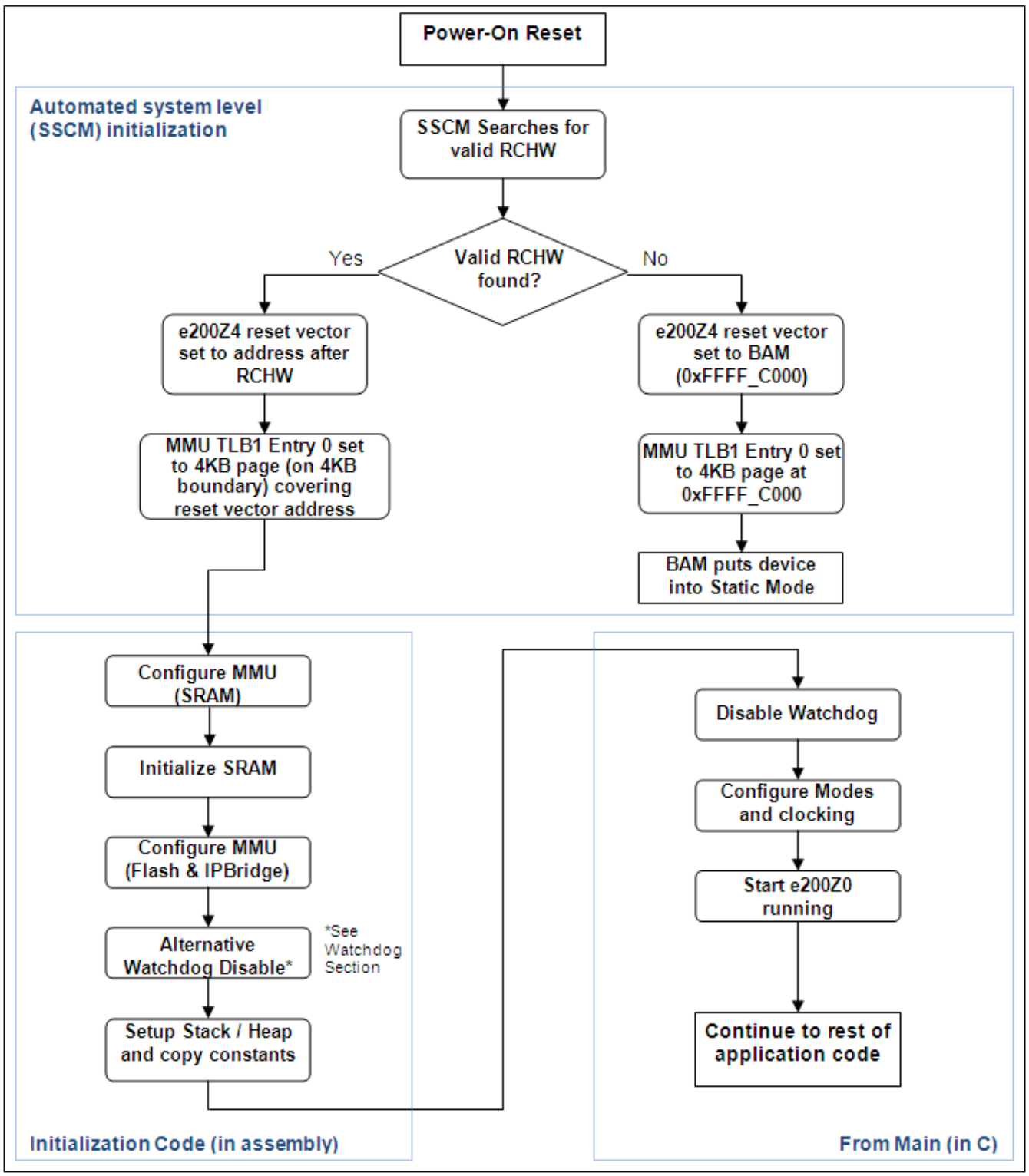


Figure 1. Startup flow

3 Initialization Code

3.1 MMU Considerations

The e200z4 core has a Memory Management Unit (MMU) which prohibits access from the e200z4 core to the crossbar unless a valid MMU entry is configured for that access.

Unlike some of the other Freescale MPC5xxx devices that you may have used, the MPC564xB/C Boot Assist Module (BAM) does not run when the device performs a normal flash boot. This means that the MMU is not configured automatically apart from the small 4 KB block at the reset vector. Therefore, all the MMU configuration must be done manually by the user before any of the additional memory outside the 4 KB block or the peripherals are accessed.

As mentioned previously, the SSCM searches the flash for a valid RCHW and if found, will configure a 4 KB MMU TLB entry covering the boot vector address. The MMU entry is 4 KB aligned, so in order to maximize the available space inside the 4 KB block, the reset vector should be set close to the start of a 4 KB boundary.

For example, if the reset vector is set to address 0x0000_0020, then the SSCM will align the 4 KB MMU page to the start of the 4 KB aligned block containing address 0x0000_0020 – an MMU page starting at address 0x0000_0000.

Similarly, if the reset vector is set to address 0x0000_0FF0 which is towards the end of an aligned 4 KB block, the MMU page still has to start at 0x0000_0000 leaving very little room in the 4 KB MMU page for code execution.

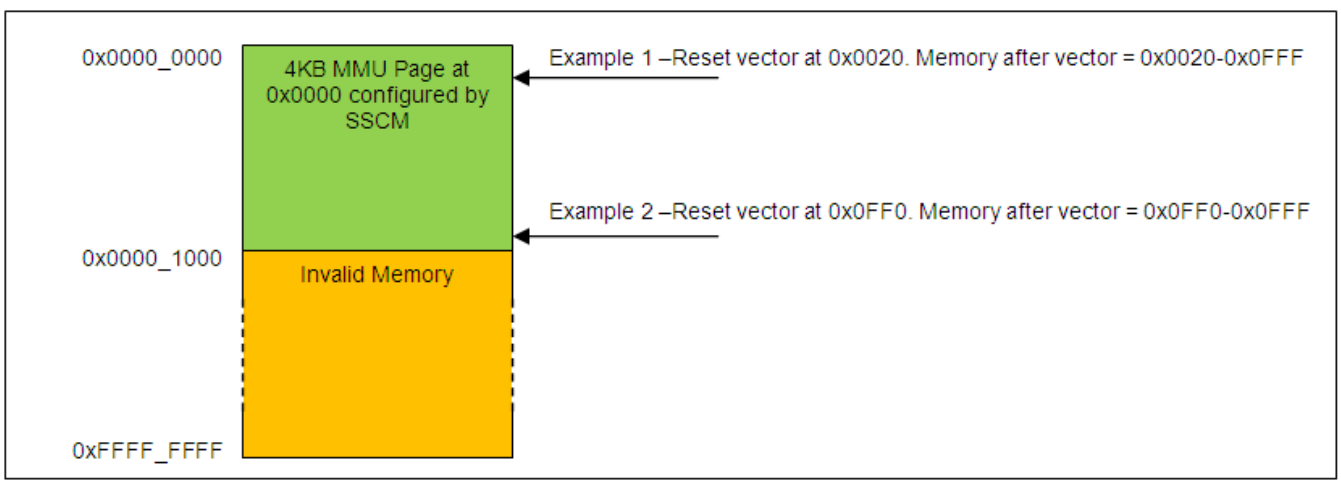


Figure 2. Reset vector and default MMU page

In order to configure the MMU for a typical application, MMU access needs to be granted so that the e200z4 core can access flash, RAM and the peripheral blocks; an example of this is shown in the table given below. In a real application the MMU blocks could be split to provide more protection granularity if desired. Note that the MMU only protects access from the e200z4 core to the crossbar so does not affect any of the other crossbar masters such as the e200z0 core or the DMA. There are 16 MMU TLB entries possible on the MPC564xB/C device.

Table 2. Example MMU generic configuration

Area	Memory range	Actual size	Closest MMU size
SRAM	0x40000000 – 0x4003FFFF	256 KB	256 KB
Flash incl shadow	0x00000000 – 0x00FFFFFF	16 MB	16 MB
Peripherals	0xC0000000 – 0xFFFFFFFF	1 GB	1 GB

The MMU SRAM entry must be configured prior to performing any RAM accesses including ECC initialization and setting up the stack. Similarly no register writes can be performed before the peripheral's MMU entry is configured. This means that the MMU regions need to be set up at the start of the initialization files.

The 4 KB MMU page configured by the SSCM uses TLB entry 0 by default. MMU pages must not overlap and must also be configured on a boundary matching their size (e.g. 4 KB MMU page must sit on a 4 KB boundary). It is strongly recommended that you do not change the MMU configuration of a memory location currently being accessed, otherwise, there will be potential for errors. The order of device initialization is therefore defined as follows:

- Configure the MMU TLB entry for RAM
- Initialize the RAM ECC (by performing 32-bit writes to the full RAM array)
- Copy code to RAM which will configure the MMU TLB entry for flash and execute it. Transfer execution back to the flash.
- Configure the MMU TLB entry for the IPBridge (peripherals)

3.1.1 Configure MMU for RAM

In order to configure MMU TLB entries, there are four MMU Assist registers (MAS) which are written with the TLB entry number, start address and size of the MMU entry and other information such as whether the page will be VLE or BookE instructions.

The code snippet below shows the MMU configuration for RAM using TLB entry 1. Note how the MAS registers are written and then the TLB is validated with a single “tlbwe” (TLB Write Entry) instruction.

Figure 3. MMU TLB configuration for SRAM

```
#/*****************************************************************************/
#/* (3) Setup MMU TLB Entry 1 for RAM at 0x4000_0000 to 0x4003_FFFF           */
#/*****************************************************************************/

    e_lis      r3, 0x1001     #/* MAS0, Configure TLB1, Entry 1                */
    mtmas0     r3

    e_lis      r3, 0xC000     #/* MAS1 = 0xC0000400 (256Kb)                    */
    e_or2i     r3, 0x0400
    mtmas1     r3

    e_lis      r3, 0x4000     #/* MAS2 = 0x40000028                            */
    e_or2i     r3, 0x0028
    mtmas2     r3

    e_lis      r3, 0x4000     #/* MAS3 = 0x4000003F                            */
    e_or2i     r3, 0x003F
    mtmas3     r3

    tlbwe                   #/* Comit to TLB                                 */
```

3.1.2 Initialize the RAM ECC

As with all other MPC5xxx devices, the RAM ECC needs to be initialized, after a reset by performing a 32-bit write to all of the words in RAM. The most efficient way to do this is to use the Store Multiple Word instruction to copy all of the 32-bit General Purpose Registers (GPRS) to RAM. Failure to initialize the RAM ECC prior to reading a RAM location will result in an ECC error and therefore an exception being raised.

Figure 4. SRAM ECC Initialization

```
#/*****************************************************************************/
#/* (4) Initialise SRAM by copying all 32GPR's to RAM (fast)                  */
#/*     Counter defines number of 32 x 32-bit words needed to write to RAM    */
#/*     256K = 262144 bytes -> 65536 words -> 2048 x 32 register writes       */
#/*****************************************************************************/
    e_li      r5, 2048
    mtctr     r5
    e_lis     r5,0x4000
sram_loop:
    e_stmw    r0, 0x0 (r5)
    e_addi    r5, r5, 128
    e_bdnz    sram_loop
```

3.1.3 Run code from RAM to reconfigure flash MMU

As mentioned previously, it is advised not to re-configure an MMU entry for memory that is currently being used. Doing so can cause stability issues at the point when the MMU TLB region is re-validated. In order to avoid this, the initialization code copies a section of code from flash to RAM which will re-configure the flash MMU entry. Code execution is then transferred from flash to RAM so that the flash MMU entry can be safely re-configured.

When the code is executed from flash, all of the code between the labels “copy_start” and “copy_end” will be copied to and executed from the SRAM. Note that, this code takes the same format as previous MMU configuration examples except, this time TLB entry 0 is being addressed. Note also, there is an “se_blr” instruction which gives the link back to the flash once the code has executed from SRAM.

Figure 5. Code to copy to RAM to re-configure flash MMU settings

```
#/*************************************************************************/
#/* (5) MMU configuration code for flash (TLB1 entry 0) -> copy to RAM    */
#/*     TLB1 entry 0, 0x0000_0000 to 0x00FF_FFFF overwriting existing TLB */
#/*************************************************************************/

    e_b copy_to_ram   #/* Copy code between copy_start and copy_end -> RAM */

#/* ---- Start of block that will be copied to RAM  */
copy_start:

    e_lis     r3, 0x1000    #/* MAS0, Configure TLB1, Entry 0              */
    mtmas0    r3

    e_lis     r3, 0xC000    #/* MAS1 = 0xC0000700 (16MB)                   */
    e_or2i    r3, 0x0700
    mtmas1    r3

    e_lis     r3, 0x0000    #/* MAS2 = 0x00000020                          */
    e_or2i    r3, 0x0020
    mtmas2    r3

    e_lis     r3, 0x0000    #/* MAS3 = 0x0000003F                          */
    e_or2i    r3, 0x003F
    mtmas3    r3

    tlbwe                    #/* Comit to TLB                             */

    se_blr                   #/* Return to flash (when running from RAM)  */

copy_end:
#/* ---- End of block that will be copied to RAM  */
```

The next section of the code calculates how many bytes of data to copy (between “copy_start” and “copy_end” and the copy routine then copies this data to the SRAM at address 0x4000_0000. Finally execution is transferred to start of RAM with the return address stored in the link register.

Figure 6. Perform RAM copy and jump to execute code in RAM

```
   #/* Calculate number of bytes to copy (data between labels)             */
copy_to_ram:
    e_lis      r3,  copy_start@h
    e_or2i     r3, copy_start@l
    e_lis      r4,  copy_end@h
    e_or2i     r4, copy_end@l
    subf       r4, r3, r4
    mtctr      r4
    e_lis      r5, 0x4000
    mtlr       r5

   #/* And now copy code from flash to RAM                                 */
copy:
    e_lbz      r6, 0(r3)
    e_stb      r6, 0(r5)
    e_addi     r3, r3, 1
    e_addi     r5, r5, 1
    e_bdnz     copy

  #/* Finally jump to RAM to execute code. Return to flash when complete   */
  se_blrl
```

3.1.4 Configure MMU for peripheral bridge

Before any of the peripherals (including the SWT) can be addressed, a valid MMU entry must be created as shown below – this time using TLB entry 2.

Figure 7. Configuring MMU TLB entry for peripherals (and BAM)

```
#/*************************************************************************/
#/* (6) MMU configuration code for Peripheral Area (IPBridge)             */
#/*     TLB1, entry 2 1GB 0xC000_0000 to 0xFFFF_FFFF                      */
#/*************************************************************************/

    e_lis      r3, 0x1002     #/* MAS0, Configure TLB1, Entry 2            */
    mtmas0     r3

    e_lis      r3, 0xC000     #/* MAS1 = 0xC0000A00                        */
    e_or2i     r3, 0x0A00
    mtmas1     r3

    e_lis      r3, 0xC000     #/* MAS2 = 0xC000002A                        */
    e_or2i     r3, 0x002A
    mtmas2     r3

    e_lis      r3, 0xC000     #/* MAS3 = 0xC000003F                        */
    e_or2i     r3, 0x003F
    mtmas3     r3

    tlbwe                   #/* Comit to TLB                             */
```

3.2 Other Initialization

The stack / heap can now be setup as well as constants and pre-initialized variables being copied from flash to RAM. These initialization steps are sometimes performed by a pre-built compiler initialization script hidden from the user. In any case these initialization steps are tightly coupled to the linker file and is compiler specific so is not detailed in the application note or example code. This code will also typically be the same as used on other MPC56xx devices.

4 Configuration from Main program

4.1 Disable watchdog

The Software Watchdog Timer (SWT) is clocked from the Slow Internal Reference Clock (SIRC) with a nominal frequency of 128 KHz. The default SWT timeout is set to 0x0000_0500 which correlates to approximately 10 ms, however the timeout period will be determined by the actual SIRC frequency which will vary over temperature according to the device specification. In order to prevent a system reset, the watchdog must be serviced or disabled prior to the initial expiry of the timer.

In the flow chart shown at the start of this application note, there are two potential options for when to disable or refresh and configure the watchdog timer. The default option is to do this at the start of the main program once all of the initialization code has run. This works fine in this particular example, however the user must ensure that there is a sufficient time window in the time taken for the initialization code to complete and get to the SWT code in main versus the SWT timeout period. If there is any doubt over the time taken to run the initialization code, the watchdog can be addressed as soon as the MMU page is defined for the peripherals.

In this example the watchdog is totally disabled however it is expected that in a real application the watchdog would be serviced and re-configured to suit the application rather than being disabled. For more information on configuring and using the SWT, consult the Reference manual.

In order to disable the watchdog in software you need to:

- Write the sequence of 0xC520 followed by 0xD928 to the service register. This clears the soft lock bit enabling the next step in the process
- Clear the WEN bit in the Control register

Figure 8. Disabling the SWT

```
SWT.SR.R = 0xC520;          /* Clear Soft lock bit in CR by writing     */
SWT.SR.R = 0xD928;          /*  0xC520 followed by 0xD928 to WSC field  */
SWT.CR.B.WEN = 0x0;         /* Clear Watchdog Enable Bit -> 0xC000010A  */
```

There is an additional way to disable the SWT at device reset by writing to the Non Volatile User Options (NVUSRO) register located in the shadow row at address 0x00FF_FE18. By disabling the watchdog in this manner it can then be re-enabled at a later stage if desired.

- If bit-0 (MSB) of NVUSRO is set, as with an erased shadow flash, then the SWT is enabled
- If bit-0 of NVUSRO is cleared, then the SWT is disabled

Note that if no valid RCHW is found by the SSCM, the watchdog will be re-enabled (even if the SWT is disabled in the NVUSRO register) before the device enters Static mode.

NOTE

When you are connecting to the MPC564xB/C device using a debugger, it is likely that the debugger itself will disable the watchdog to allow debug to be carried out. This can result in a fairly common problem when attempting to run the code in a standalone configuration where a periodic device reset is observed, caused by the watchdog timeout.

4.2 Mode configuration and clocking

The MPC564xB/C device has several operating modes in line with other devices within the MPC5xxx family portfolio. Out of reset the device leaves the Reset mode and enters DRUN mode. In order to use all of the available modes, they must be enabled in the Mode Enable register.

Figure 9. Enabling all device operation modes

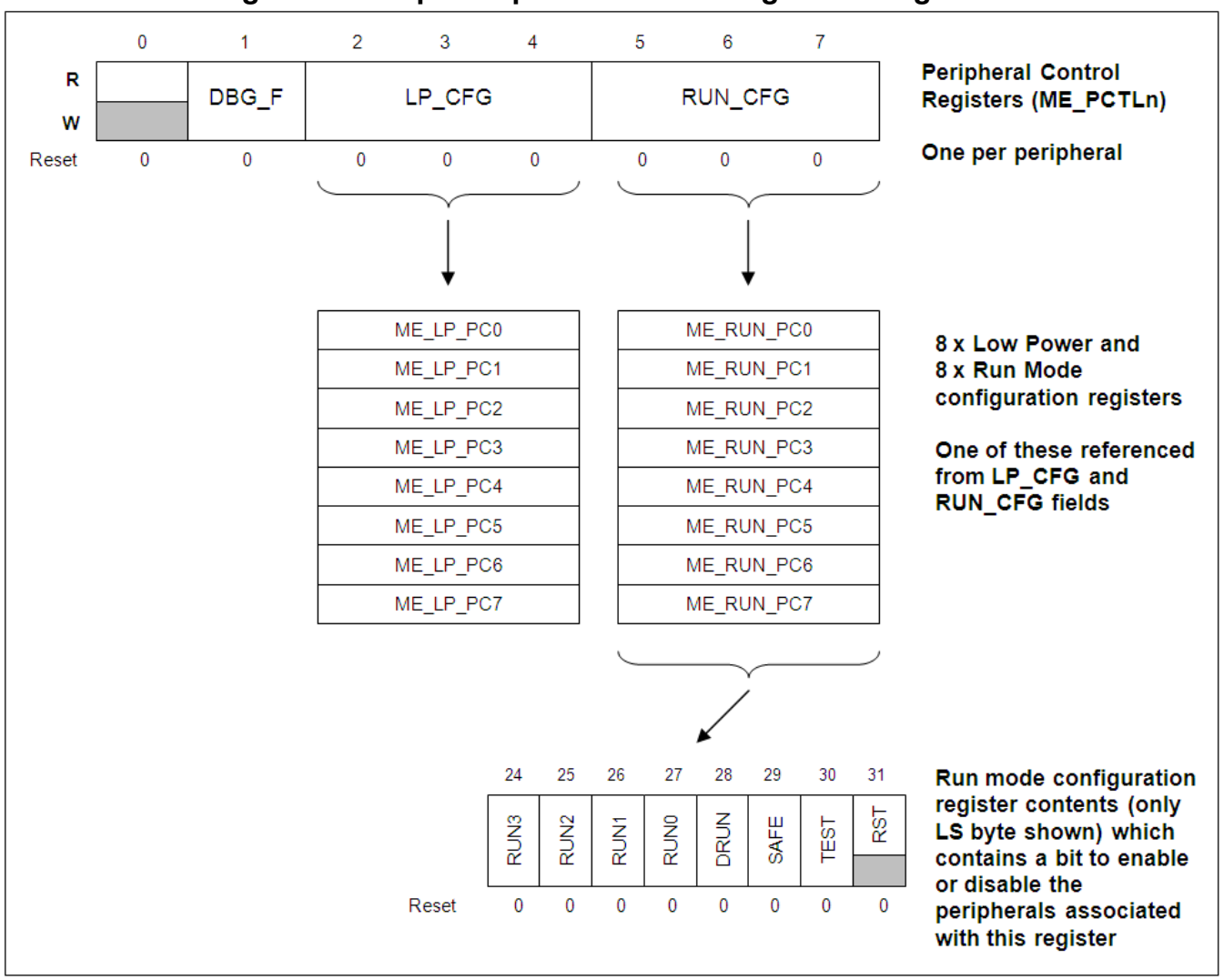
```
ME.MER.R = 0x000025FF;          /* Enable all modes                    */
```

Every peripheral has an associated control register which has 3 fields to determine what happens when the MCU is in the following modes:

- Debug mode (DBG_F)
- Low Power mode (LP_CFG)
- Run Mode (RUN_CFG)

For Low Power and Run modes there are 8 sets of configuration registers that are referenced by the LP_CFG and RUN_CFG fields from the peripheral control registers. Each of these configuration registers have a bit for each Run or Low Power mode which determines whether the peripherals referencing this register are clock gated or available in that mode. This allows up to 8 different power schemes to be created with a different mix of peripherals to be available in each Low Power or Run mode. The block diagram below shows how these registers are referenced and associated with the individual peripheral control registers.

Figure 10. Peripheral power mode configuration registers



Out of reset, all of the peripheral control registers have value 0x0 which means they are associated with ME_RUN_PC0 for Run modes (and ME_LP_PC0 for Low Power modes). Setting ME_RUN_PC0 to 0x0000_00FE, enables all peripherals in all Run modes which is the baseline configuration used in this example.

Figure 11. Enabling all peripherals in Run modes

```
ME.RUNPC[0].R = 0x000000FE;      /* Enable all peripherals in all modes   */
```

There are 3 peripheral clock groups on the MPC564xB/C with associated clock dividers. These allow all 3 peripheral clock groups independent clock control so groups can be slowed down or even clock gated which helps with power saving. For more details and for peripheral assignments, refer to the device Reference manual. Peripheral group 1 can be clocked at a maximum of 32 kHz, peripheral groups 2 and 3 can run at a maximum of 64 kHz. Care needs to be taken not to exceed these limits.

Note that a peripheral clock group has to be specifically enabled or the peripherals will not be accessible. For the purpose of this example, all of the peripheral clock groups will be enabled by writing to the associated system clock divider registers even though all are not required.

```
/* Enable system clock for all peripherals assuming 120MHz system clock  */
 CGM.SC_DC[0].R = 0x83;          /* Max 32MHz. Closest is 30MHz, Div+1=3 */
 CGM.SC_DC[1].R = 0x81;          /* Max 64MHz. Closest is 60MHz, Div+1=2 */
 CGM.SC_DC[2].R = 0x81;          /* Max 64MHz. Closest is 60MHz, Div+1=2 */
```

Figure 12. Enabling the system clock for all peripherals

Finally, in line with all clock and mode configuration, a mode transition must be made for all the above changes to take effect. In this case, DRUN mode is re-entered by writing a mode and key command sequence as shown below.

Figure 13. Re-entering DRUN mode to activate the changes

```
/* Re-enter DRUN mode to update the clock configuration                  */
 ME.MCTL.R = 0x30005AF0;           /* DRUN Mode & Key                    */
 ME.MCTL.R = 0x3000A50F;           /* DRUN Mode & Key                    */
 while (ME.IS.B.I_MTC != 1) {}     /* Wait Until transition completed    */
 ME.IS.B.I_MTC = 1;                /* Clear flag                         */
```

4.3 Clock & PLL configuration

By default the MPC564xB/C is clocked from the 16 MHz IRC. To attain the maximum performance of the device, the system clock speed needs to be increased, up to 120 Mhz (consult the MPC564xB/C datasheet for the maximum system clock). Before the system clock is changed there are some system speed considerations that need to be adhered to as shown below. These all have individual clock dividers to allow the system clock to run up to the maximum of 120 MHz.

- The e200z0 core has a maximum clock speed of 80 Mhz
- The FEC (Fast Ethernet Controller) requires its clock divider to be /2 if system clock is > 80 MHz
- The flash register interface has a maximum operating frequency of 80 Mhz
- The RAM needs an additional wait state if the system clock is above 64 Mhz

All of these functional blocks mentioned above have individual clock dividers. In the example given in this application note, the system clock will be set to 120 Mhz so before the clock is changed, the clock dividers must be set accordingly. The FEC is not used so in this case the divider is left at it's default value.

Figure 14. Configuring the system clock dividers

```
/* (3) - Configure system clock dividers for 120Mhz Fsys                 */
 CGM.ZO_DCR.B.DIV = 0x1;         /* Z0 clock divider to divide by 2       */
 CGM.FLASH_DCR.B.DIV = 0x1;      /* Flash register interface /2 (default) */
 ECSM.MUDCR.B.RAM_WS=0x1;        /* RAM Wait states to divide by 2        */
```

Now that the clock dividers have been configured, the PLL can be used to change the system clock to the desired level. As with all clocking changes, the current Run mode must be re-entered to allow the change to take effect. The diagram below describes the steps needed to change the clock using the PLL. Note that all of the clock and PLL re-configuration can be achieved with a single mode re-entry as shown below but there is also an option to effect a mode re-entry for each critical step in the process. This allows easy debug in case any of the steps fail because, it allows the problem to be identified easily. With a new project and new hardware it is probably advisable to use the multi mode re-entry method so that any potential problems are easily identified.

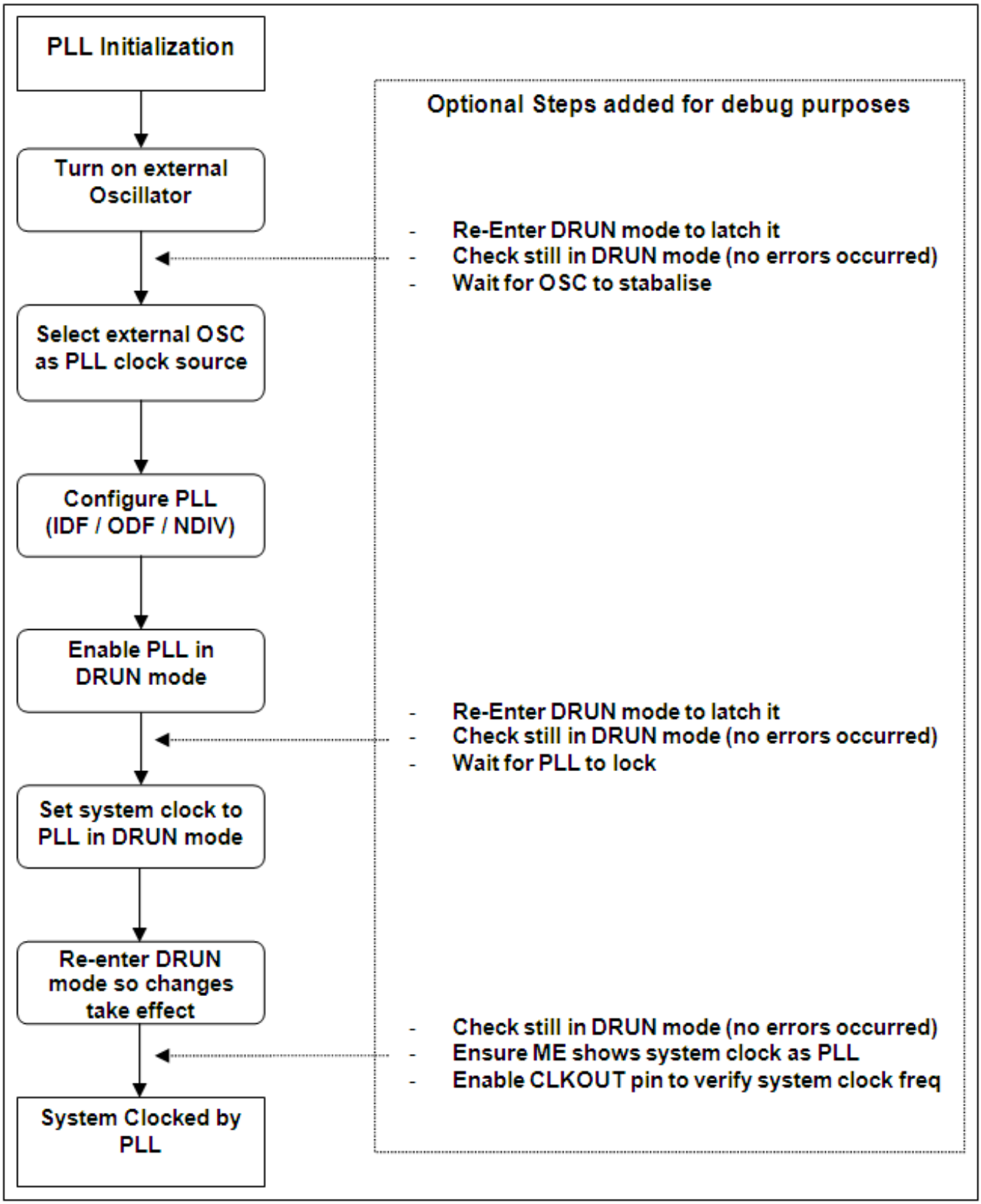


Figure 15. Using the PLL as the system clock

The code below shows the single mode re-entry method. The full application code for this apps note shows the alternative version with multiple mode re-entry steps.

Figure 16. Configuring the PLL as the system clock

```
/* Switch on external osc in DRUN mode                                    */
 ME.DRUN.B.FXOSCOON=1;

/* Select External OSC as the FMPLL Reference Clock Source                */
 CGM.ACO_SC.B.SELCTL = 0x1;

/* Configure PLL for 120MHz with 40MHz xtal                               */
 CGM.FMPLL_CR.B.IDF=0x4;           /* Divide by 5                         */
 CGM.FMPLL_CR.B.ODF=0x1;           /* Divide by 4                         */
 CGM.FMPLL_CR.B.NDIV=60;           /* Divide by 60                        */

/* Enable PLL in DRUN mode.                                               */
 ME.DRUN.B.FMPLLON  = 1;

/* Finally set system clock to be PLL in DRUN mode                        */
 ME.DRUN.B.SYSCLK=0x4;

/* Re-Enter DRUN mode (mode=0x3) to activate change                       */
 ME.MCTL.R = 0x30005AF0;           /* Mode & Key                          */
 ME.MCTL.R = 0x3000A50F;           /* Mode & Key inverted                 */
 while(ME.GS.B.S_MTRANS == 1);     /* Wait for mode transition complete */

/* Mode Check - if entered mode other than DRUN (eg SAFE), then loop      */
 while(ME.GS.B.S_CURRENTMODE != 3);

/* Wait for PLL to lock                                                   */
 while(CGM.FMPLL_CR.B.S_LOCK==0);

/* Final check - ensure ME_GS reports clock as system PLL (0x4)           */
 while(ME.GS.B.S_SYSCLK != 4){};

/* Enable CLKOUT pin so clock frequency can be verified                   */
 CGM.OC_EN.B.EN=1;          /* Enable Output clock                        */
 CGM.OCDS_SC.R =0x23;       /* And seclect output as system clock / 4     */
 SIU.PCR[0].R = 0x0A04;     /* PA0 ALT2 function (Clkout), MAX SRC        */
```

4.4 Starting the e200z0 core

After a reset, the e200z0 core (if available on your device) is held in reset until it is released by writing a sequence of registers from the e200z4 core:

- Write the e200z0 boot (start) address to the SSCM DPM Boot. Note that this address has to be 4 byte aligned as the lower 2 bits in the BPM boot register are reserved and set to 0b00.

- Write 0x0000_5AF0 to the SSCM DPM Boot Key register
- Write 0x0000_A50F to the SSCM DPM Boot Key register

Figure 17. Starting the e200z0 running

```
/* (5) - Start Z0 running                                                  */
 SSCM.DPMBOOT.R = 0x00180000;  /* Start address of Z0 (2nd flash block)   */
 SSCM.DPMKEY.R  = 0x00005AF0;  /* Write key 1                             */
 SSCM.DPMKEY.R  = 0x0000A50F;  /* Write key 2                             */
```

The e200z0 will now start executing code from the start address defined in the DPM Boot register. Any subsequent reset of the MCU will result in the e200z0 being held in reset until the sequence is carried out again by the e200z4 core.

The memory partition and the crossbar architecture have been optimized for dual core operation.

- The SRAM is split over 2 slave ports, 0x4000_0000 to 0x4001_FFFF on slave port 2 and 0x4002_0000 to 0x4003_FFFF on slave port 3
- There are 2 flash ports, z4 instruction port on slave port 0 and a second flash port for everything else on slave port 1.

By splitting the SRAM in the linker file(s) such that the e200z4 has access to one block and the e200z0 has access to the other block, crossbar contentions are generally avoided between the cores for RAM access. If there are large amounts of DMA transfers to/from RAM it may be possible to use the RAM allocated to the processor which performs fewer accesses to RAM. Note that none of this prevents either core from writing the full RAM array so care has to be taken in software (using semaphores and / or software interrupts) to ensure memory coherency.

5 Other considerations

The initialization described above is intended to get you started with the MPC564xB/C device but does not determine settings for optimum performance. There are many more system level configurations that can be customized to match your application for optimum performance.

- Signal Processing Extension (SPE) for fixed point and single precision calculations.
- Branch Target Buffer (BTB) providing target address pre-fetching.
- Flash Port Configuration allowing flash wait states to be configured as well as line buffer allocation and pre-fetch behaviour.
- Highly configurable crossbar (XBAR) architecture allowing tailored priorities and arbitration scheme on a per slave port basis.
- 4 KB cache on e200z4 core.

For more details on how to configure these features and on the MPC564xB/C microcontroller, consult the Reference manual which can be downloaded from http://www.freescale.com.

Appendix A Application Code Example

The example application code discussed in this application note is available for you to use directly from the following pages. Note that the formatting and alignment is configured for a normal text editor so the formatting should be correctly aligned when pasted into a standard text editor with normal spacing.

A.1 Application Code

Refer to the following sections for the example code.

Init Code (excluding stack and memcopy)

Main.c

typedefs.h

project.h

A.1.1 Init Code (excluding stack and memcopy)

```
#/*****************************************************************************/
#/* FILE NAME: flash_init_VLE_Z4.s       COPYRIGHT (c) Freescale 2010    */
#/*                                                    All Rights Reserved     */
#/* DESCRIPTION:                                                              */
#/* This is the initialization (crt0) file for MPC564xB/C devices running  */
#/*  from flash in VLE mode and performs the following configuration       */
#/*                                                                           */
#/*  (1) Configures the Reset Config Half word for VLE                      */
#/*  (2) Optionally enables SPE (required for some compiler functions)     */
#/*  (3) Configures MMU TLB entry 1 for 256KB RAM at 0x4000_0000           */
#/*  (4) Initialises the ECC in the SRAM by writing all words              */
#/*  (5) Copies FLASH MMU Code to RAM, runs it and return back to flash    */
#/*  (6) Configures IPBridge (peripherals) MMU TLB entry 2                 */
#/*                                                                           */
#/*  Resultant MMU configuration is:                                       */
#/*   0 - 0x0000_0000 to 0x00FF_FFFF (16MB  VLE) FLASH                     */
#/*   1 - 0x4000_0000 to 0x4003_FFFF (256KB VLE) RAM                       */
#/*   2 - 0xC000_0000 to 0xFFFF_FFFF (1GB   VLE) Platform Peripherals      */
#/*                                                                           */
#/*****************************************************************************/
#/*                                                                           */
#/*   Revision Information:                                                   */
#/*   0.1  A. Robertson  24Sep10   Created specifically for MPC564xB/C     */
#/*                                                                           */
#/*****************************************************************************/


#/*****************************************************************************/
#/* (1) RCHW Configuration (and referencing of start label for linker)     */
#/*****************************************************************************/
    .section  ".rcw"
    .LONG     0x015A0000     #/* Set VLE bit                        */
    .LONG     _start_Z4

    .file     "flash_init_vle_Z4.s"
    .section  .start ,avx
    .vle
    .global   _start_Z4

_start_Z4:


#/*****************************************************************************/
#/* (2) Enable Signal Processing extension (SPE) in Machine State Register */
#/*****************************************************************************/
    mfMSR     r3
    e_or2is   r3, 0x0200
    mtMSR     r3


#/*****************************************************************************/
#/* (3) Setup MMU TLB Entry 1 for RAM at 0x4000_0000 to 0x4003_FFFF        */
```

```
#/*********************************************************************/

    e_lis     r3, 0x1001    #/* MAS0, Configure TLB1, Entry 1             */
    mtmas0    r3

    e_lis     r3, 0xC000    #/* MAS1 = 0xC0000400 (256Kb)                 */
    e_or2i    r3, 0x0400
    mtmas1    r3

    e_lis     r3, 0x4000    #/* MAS2 = 0x40000028                         */
    e_or2i    r3, 0x0028
    mtmas2    r3

    e_lis     r3, 0x4000    #/* MAS3 = 0x4000003F                         */
    e_or2i    r3, 0x003F
    mtmas3    r3

    tlbwe                  #/* Comit to TLB                              */


#/*********************************************************************/
#/* (4) Initialize SRAM by copying all 32GPR's to RAM (fast)          */
#/*     Counter defines number of 32 x 32-bit words needed to write to RAM */
#/*     256K = 262144 bytes -> 65536 words -> 2048 x 32 register writes    */
#/*********************************************************************/
    e_li      r5, 2048
    mtctr     r5
    e_lis     r5,0x4000
sram_loop:
    e_stmw    r0, 0x0 (r5)
    e_addi    r5, r5, 128
    e_bdnz    sram_loop

#/*********************************************************************/
#/* (5) MMU configuration code for flash (TLB1 entry 0) -> copy to RAM     */
#/*     TLB1 entry 0, 0x0000_0000 to 0x00FF_FFFF overwriting existing TLB  */
#/*********************************************************************/

    e_b copy_to_ram   #/* Copy code between copy_start and copy_end -> RAM */

#/* ---- Start of block that will be copied to RAM  */
copy_start:

    e_lis     r3, 0x1000    #/* MAS0, Configure TLB1, Entry 0             */
    mtmas0    r3

    e_lis     r3, 0xC000    #/* MAS1 = 0xC0000700 (16MB)                  */
    e_or2i    r3, 0x0700
    mtmas1    r3

    e_lis     r3, 0x0000    #/* MAS2 = 0x00000020                         */
    e_or2i    r3, 0x0020
    mtmas2    r3

    e_lis     r3, 0x0000    #/* MAS3 = 0x0000003F                         */
    e_or2i    r3, 0x003F
    mtmas3    r3

    tlbwe                    #/* Comit to TLB                           */

    se_blr                   #/* Return to flash (when running from RAM)   */

copy_end:
#/* ---- End of block that will be copied to RAM  */

   #/* Calculate number of bytes to copy (data between labels)           */
copy_to_ram:
    e_lis     r3,  copy_start@h
    e_or2i    r3, copy_start@l
```

```
    e_lis     r4,  copy_end@h
    e_or2i    r4, copy_end@l
    subf      r4, r3, r4
    mtctr     r4
    e_lis     r5, 0x4000
    mtlr      r5

   #/* And now copy code from flash to RAM                                  */
copy:
    e_lbz     r6, 0(r3)
    e_stb     r6, 0(r5)
    e_addi    r3, r3, 1
    e_addi    r5, r5, 1
    e_bdnz    copy

  #/* Finally jump to RAM to execute code. Return to flash when complete   */
  se_blrl



#/*****************************************************************************/
#/* (6) MMU configuration code for Peripheral Area (IPBridge)                 */
#/*     TLB1, entry 2 1GB 0xC000_0000 to 0xFFFF_FFFF                          */
#/*****************************************************************************/

    e_lis     r3, 0x1002    #/* MAS0, Configure TLB1, Entry 2                 */
    mtmas0    r3

    e_lis     r3, 0xC000    #/* MAS1 = 0xC0000A00                             */
    e_or2i    r3, 0x0A00
    mtmas1    r3

    e_lis     r3, 0xC000    #/* MAS2 = 0xC000002A                             */
    e_or2i    r3, 0x002A
    mtmas2    r3

    e_lis     r3, 0xC000    #/* MAS3 = 0xC000003F                             */
    e_or2i    r3, 0x003F
    mtmas3    r3

    tlbwe                  #/* Comit to TLB                                  */
```

A.1.2 Main.c

```
/* ************************************************************************** */
/* MPC564xB/C simple startup Flash VLE template (e200Z4).                     */
/* Run this code alongside corresponding Z0 code which also needs to be       */
/*  programmed into the Flash.                                                */
/*                                                                            */
/* Copyright Freescale Semiconductor 2010. MCD Applicaitons East Kilbride     */
/*                                                                            */
/* Revision Information:                                                      */
/* v0.1   Apr  2010    Alasdair Robertson     Initial Version                 */
/* v0.2   Sept 2010    Alasdair Robertson     Tweaked post silicon            */
/*                                                                            */
/* ************************************************************************** */

#include "..\header\project.h"

/* Function Prototypes */
void DisableWatchdog();
void initMODE();
void setPLL();


int main(void)
```

```
{
    uint32_t count;               /* temp counter used in delay loop        */

    /* (1) - Disable Software Watchdog Timer                                */
    DisableWatchdog();            /* (Can also be disabled in Shadow row)   */

    *((vuint16_t *) 0x40020000) = 0x0;
    *((vuint16_t *) 0x40020002) = SWT.TO.R;
     *((vuint32_t *) 0x40020004) = 0x0;


   /* (2) - Configure modes and activate all clock for all peripherals      */
    initMODE();

   /* (3) - Configure system clock dividers for 120Mhz Fsys                 */
    CGM.Z0_DCR.B.DIV = 0x1;       /* Z0 clock divider to divide by 2        */
    CGM.FLASH_DCR.B.DIV = 0x1;    /* Flash register interface /2 (default)  */
    ECSM.MUDCR.B.RAM_WS=0x1;      /* RAM Wait states to divide by 2         */

   /* (4) - Set system clock to 120MHz based on 40Mhz XTAL                  */
    setPLL();                     /* Also enables CLKOUT On PA[0]           */

   /* (5) - Start Z0 running                                                */
    SSCM.DPMBOOT.R = 0x00180000;  /* Start address of Z0 (2nd flash block)  */
    SSCM.DPMKEY.R  = 0x00005AF0;  /* Write key 1                            */
    SSCM.DPMKEY.R  = 0x0000A50F;  /* Write key 2                            */


   /* Loop forever flashing an LED connected to Port PA[1]                  */
    SIU.PCR[1].R = 0x0200;        /* PA[0] to GPIO mode, output             */
    while(1)
     {
       SIU.GPDO[1].R = (~(SIU.GPDO[1].R) & 0x01);  /* Invert LED output    */
       for (count=0; count<1000000; count++);      /* wait a while         */
     };

} /* End Of Main */


/* ----------------------------- */
/* Disable Software Watchdog     */
/* ----------------------------- */
void DisableWatchdog(void)
{
    SWT.SR.R = 0xC520;          /* Clear Soft lock bit in CR by writing     */
    SWT.SR.R = 0xD928;          /*  0xC520 followed by 0xD928 to WSC field  */
    SWT.CR.B.WEN = 0x0;         /* Clear Watchdog Enable Bit -> 0xC000010A  */
}

/* ----------------------------- */
/* Initialize Modes              */
/* ----------------------------- */
void initMODE(void)
{
    ME.MER.R = 0x000025FF;          /* Enable all modes                     */
    ME.RUNPC[0].R = 0x000000FE;     /* Enable all peripherals in all modes  */

   /* Enable system clock for all peripherals assuming 120MHz system clock  */
    CGM.SC_DC[0].R = 0x83;          /* Max 32MHz. Closest is 30MHz, Div+1=3 */
    CGM.SC_DC[1].R = 0x81;          /* Max 64MHz. Closest is 60MHz, Div+1=2 */
    CGM.SC_DC[2].R = 0x81;          /* Max 64MHz. Closest is 60MHz, Div+1=2 */

   /* Re-enter DRUN mode to update the clock configuration                  */
    ME.MCTL.R = 0x30005AF0;           /* DRUN Mode & Key                    */
    ME.MCTL.R = 0x3000A50F;           /* DRUN Mode & Key                    */
    while (ME.IS.B.I_MTC != 1) {}     /* Wait Until transition completed    */
    ME.IS.B.I_MTC = 1;                /* Clear flag                         */
}

/* ----------------------------- */
```

```
/* PLL to 120Mhz (40Mhz xtal)    */
/* ----------------------------- */
void setPLL(void)
{

   /* Note - in example code below the flow is:                              */
   /*  Switch on osc, change mode and wait for osc ON                        */
   /*  Configure and enable PLL, change mode and wait for PLL to lock        */
   /*  Set clock source as PLL, change mode and check clock is PLL           */
   /*                                                                        */
   /*  However, do not actually have to do all 3 mode changes. Can           */
   /*  switch on osc, enable PLL and Set PLL as clock source THEN do a       */
   /*  single mode change. The ME module must be smart enough to look        */
   /*  at which bits are set and see if it's a valid combination of          */
   /*  bits. However, if there is an issue, there is no way of seeing        */
   /*  what caused the problem!                                              */


   /* Switch on external osc in DRUN mode                                    */
    ME.DRUN.B.FXOSC0ON=1;

   /* Re-Enter DRUN mode (mode=0x3) to activate change                       */
    ME.MCTL.R = 0x30005AF0;            /* Mode & Key                        */
    ME.MCTL.R = 0x3000A50F;            /* Mode & Key inverted               */
    while(ME.GS.B.S_MTRANS == 1);      /* Wait for mode transition complete */

   /* Error trap - if current mode is not DRUM (eg in safe mode), then loop */
    while(ME.GS.B.S_CURRENTMODE != 3);


   /* Wait for external OSC to stabilize                                     */
    while(ME.GS.B.S_FXOSC != 1);

   /* Select External OSC as the FMPLL Reference Clock Source                */
    CGM.AC0_SC.B.SELCTL = 0x1;


   /* Configure PLL for 120MHz with 40MHz xtal                               */
   /*  PLL frequency = (40 * NDIV) / (IDF * ODF)                             */
   /*  VCO (PLL * ODF) must be between 256 and 512MHz                        */
   /*                                                                        */
   /* For 120Mhz Output:                                                     */
   /*  ODF diviers are 2, 4, 8, 16. /4 gives VCO of 480 (in range)           */
   /*  With ODF = 2, NDIV = 12xIDF. Chose IDF =5, therefore NDIV = 60        */

    CGM.FMPLL_CR.B.IDF=0x4;           /* Divide by 5                         */
    CGM.FMPLL_CR.B.ODF=0x1;           /* Divide by 4                         */
    CGM.FMPLL_CR.B.NDIV=60;           /* Divide by 60                        */


   /* Enable PLL in DRUN mode.                                               */
    ME.DRUN.B.FMPLLON  = 1;

   /* Re-Enter DRUN mode (mode=0x3) to activate change                       */
    ME.MCTL.R = 0x30005AF0;            /* Mode & Key                        */
    ME.MCTL.R = 0x3000A50F;            /* Mode & Key inverted               */
    while(ME.GS.B.S_MTRANS == 1);      /* Wait for mode transition complete */

   /* Error trap - if current mode is not DRUM (eg safe mode), then loop     */
    while(ME.GS.B.S_CURRENTMODE != 3);


   /* wait for PLL to lock (will not lock until re-enter DRUN mode           */
    while(CGM.FMPLL_CR.B.S_LOCK==0);


   /* Finally set system clock to be PLL in DRUN mode                        */
    ME.DRUN.B.SYSCLK=0x4;

   /* Re-Enter DRUN mode (mode=0x3) to activate change                       */
```

```
    ME.MCTL.R = 0x30005AF0;            /* Mode & Key                          */
    ME.MCTL.R = 0x3000A50F;            /* Mode & Key inverted                 */
    while(ME.GS.B.S_MTRANS == 1);      /* Wait for mode transition complete */

   /* Error trap - if current mode is not DRUM (eg safe mode), then loop    */
    while(ME.GS.B.S_CURRENTMODE != 3);


   /* Final check - ensure ME_GS reports clock as system PLL (0x4)          */
    while(ME.GS.B.S_SYSCLK != 4){};     /* fail if stuck here            */


   /* Enable CLKOUT pin so clock frequency can be verified                  */
    CGM.OC_EN.B.EN=1;          /* Enable Output clock                       */
    CGM.OCDS_SC.R =0x23;       /* And seclect output as system clock / 4    */
    SIU.PCR[0].R = 0x0A04;     /* PA0 ALT2 function (Clkout), MAX SRC       */

}
```

A.1.3 typedefs.h

```
#ifdef DEBUG
/*****************************************************************************/
/* FILE NAME: typedefs.h                     COPYRIGHT (c) Freescale 2010 */
/* VERSION:  0.6                                  All Rights Reserved     */
/*                                                                           */
/* DESCRIPTION:                                                              */
/* This file defines all of the data types for the Freescale header file.  */
/*===========================================================================*/
/* AUTHOR: Jeff Loeliger                                                     */
/*                                                                           */
/* UPDATE HISTORY                                                            */
/* REV      AUTHOR      DATE       DESCRIPTION OF CHANGE                     */
/* ---   -----------  ---------    ---------------------                     */
/* 0.1   J. Loeliger  17/Feb/03    Initial version of file.                  */
/* 0.2   J. Loeliger  06/Mar/03    Added DCC support.                        */
/* 0.3   J. Loeliger  07/May/03    Change to fully use ISO data types.       */
/* 0.4   J. Loeliger  17/Jun/03    Change name to motint.h and merge         */
/*                                  MPC5500 and MAC7100 files.               */
/* 0.5   J. Loeliger  04/Nov/03    Changed name to typedefs.h.               */
/* 0.6   J. Loeliger  09/May/04    Changed to support GHS and GCC.           */
/*****************************************************************************/
#endif

#ifndef _TYPEDEFS_H_
#define _TYPEDEFS_H_

#ifdef __MWERKS__     //Metrowerk CodeWarrior
    #include <stdint.h>

    // Standard typedefs used by header files, based on ISO C standard
    typedef volatile int8_t vint8_t;
    typedef volatile uint8_t vuint8_t;

    typedef volatile int16_t vint16_t;
    typedef volatile uint16_t vuint16_t;

    typedef volatile int32_t vint32_t;
    typedef volatile uint32_t vuint32_t;

#else
#ifdef __GHS__     //GreenHills
    #include <stdint.h>

    // Standard typedefs used by header files, based on ISO C standard
    typedef volatile int8_t vint8_t;
```

```
    typedef volatile uint8_t vuint8_t;

    typedef volatile int16_t vint16_t;
    typedef volatile uint16_t vuint16_t;

    typedef volatile int32_t vint32_t;
    typedef volatile uint32_t vuint32_t;

#else

    // This is needed for compilers that don't have a stdint.h file

    typedef signed char int8_t;
    typedef unsigned char uint8_t;
    typedef volatile signed char vint8_t;
    typedef volatile unsigned char vuint8_t;

    typedef signed short int16_t;
    typedef unsigned short uint16_t;
    typedef volatile signed short vint16_t;
    typedef volatile unsigned short vuint16_t;

    typedef signed int int32_t;
    typedef unsigned int uint32_t;
    typedef volatile signed int vint32_t;
    typedef volatile unsigned int vuint32_t;

#endif
#endif
#endif
```

A.1.4 project.h

```
/********************************** File Includes *************************************/
#include "MPC564xBC_V0.3_WIP.h"
#include "typedefs.h"
```

How to Reach Us:

Home Page:
www.freescale.com

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

USA/Europe or Locations Not Listed:
Freescale Semiconductor
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 +1-303-675-2140
Fax: +1-303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Document Number: AN4239
Rev. 0, 12/2010