Freescale Semiconductor
Application Note

Document Number: AN3627
Rev. 0, 04/2008

Basic Multicore Initialization

For the MPC5516G/E and MPC5514G/E Devices

by: Bill Terry
32-Bit Automotive Applications
Microcontroller Solutions Group

Contents

1 Overview

Several devices in the MPC551x family implement a dual-core architecture, using the e200z1 and e200z0 cores. This application note describes the minimal software requirements for initializing the device and for bringing both cores to an operating status.

Simple example code is provided to illustrate the initialization procedure — however, the specifics of a particular application may require unique initializations that are not discussed in this example. The intent is to provide the engineer with an understanding of the boot process, the initialization process, and their requirements. The example code included in this application note is intended as a guide, and may not be appropriate for an actual application.

The example given in this application note is for a typical application booting from flash memory. Serial boot mode is not discussed.

2 MPC551x Dual-Core Architecture

The MPC5516G/E and MPC5514G/E devices incorporate two cores from the eSys family of cores — the e200z1 (main core) and the e200z0 (auxiliary or I/O core). This dual-core architecture has the benefit of providing lower power consumption, increased processing power, and reduced electromagnetic interference. Here are the key concepts of this architecture that the software engineer must understand:

- The e200z1 core is the main core. At reset, it is the core that executes the Boot Assist Module (BAM) and typically manages the primary initialization of system resources.
- The e200z0 is the secondary or I/O core. At initial power on reset (POR), it is held in reset by default. It must be specifically enabled by code that is executing on the e200z1 core.
- The e200z1 core may execute the Variable Length Encoded (VLE) instruction set, or classic Power Architecture instruction set. However, the e200z0 only executes the VLE instruction set. Regardless of which instruction set each core is executing, the process of bringing both cores out of reset and up to operating status does not change.
- Each of the two cores can access all system memory and peripherals, unless otherwise configured by the Memory Protection Unit (MPU). See the MPC551x Reference Manuals for details of the MPU.

Figure 1 illustrates the high level system architecture for the MPC5516G/E dual-core microcontroller.

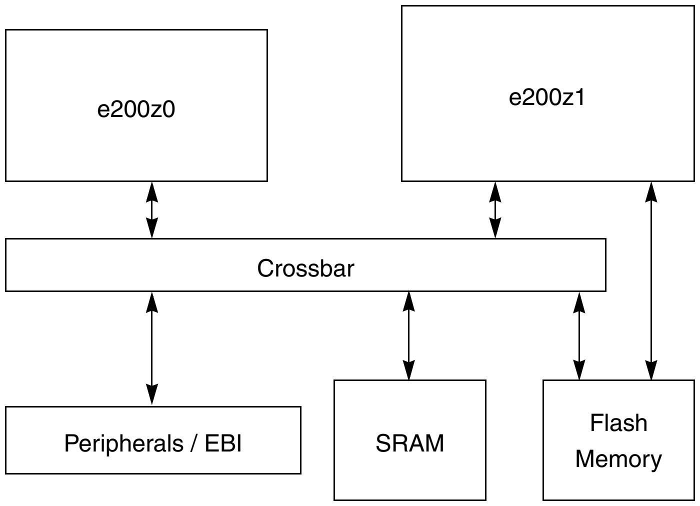

Figure 1. MPC5516G Architecture

2.1 The e200z1 Core

Here is a list of some of the key features of the e200z1 core:

- 32-bit Power Architecture Book E programming model
- Single-issue 32-bit CPU
- VLE APU for reduced code footprint
- In-order execution and retirement
- Precise exception handling

- Branch processing unit
  - Dedicated branch address calculation adder
  - Branch acceleration using Branch Target Buffer
- Support for independent instruction and data accesses to different memory subsystems, such as SRAM and flash memory via independent instruction and data BIUs.
- Load/store unit
  - 1-cycle load latency
  - Fully pipelined
  - Big- and little-endian support
  - Misaligned access support
  - Zero load-to-use pipeline bubbles for aligned transfers

2.2 The e200z0 Core

Here is a list of some of the key features of the e200z0:

- 32-bit Power Architecture Book E VLE-only programmer model
- Single-issue 32-bit CPU
- VLE APU for reduced code footprint
- In-order execution and retirement
- Precise exception handling
- Branch processing unit
  - Dedicated branch address calculation adder
- Support for instruction and data access via a unified 32-bit Instruction/Data BIU
- Load/store unit
  - 1-cycle load latency
  - Fully pipelined
  - Big-endian support only
  - Misaligned access support
  - Zero load-to-use pipeline bubbles for aligned transfers

2.3 Typical Applications

The multi-core architecture used in the MPC551x family is an asymmetric implementation. In other words, the two cores are not identical, and typically the e200z0 core will be used as a co-processor, or I/O processor. In a symmetric multi-core architecture both cores (assuming a two-core architecture) are typically identical. Many times there is an Operating System (OS) in place that may delegate tasks to each core dynamically during execution, depending on the current status of resources and the requirements of the application at any given time.

The MPC551x dual-core devices do not require an OS for efficient operation. In a typical application, the e200z1 core would run the main control loop, and the e200z0 core is configured for specific tasks, such as generating multiple pulse width modulation (PWM) channels, or collecting multiple channels of input data from an external device.

3 Flash Boot Sequence

3.1 Review of the Reset Control Half Word (RCHW) and Boot Assist Module (BAM)

As in other members of the MPC55xx family, the MPC551x devices incorporate a reset control half word (RCHW) and the Boot Assist Module (BAM). The BAM is ROM code that executes during a power-on reset (POR) sequence. The RCHW is a programmed memory location that controls how the BAM operates during the POR. The RCHW bit definition is shown in Figure 2. A brief discussion of the these two features is given here. For complete details refer to the MPC551x RM.

ADDRESS	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0x0000_0000	Reserved					WTE	PS0	VLE	Boot ID							

Figure 2. Reset Configuration Half Word

At reset the BAM executes and searches for a valid RCHW at the lowest 32-bit address in certain defined flash memory spaces. If the BAM locates a valid RCHW as indicated by an RCHW[BOOTID] value of 0x5A, the WTE, PS0, and VLE bits are read to determine further conditional BAM program operation.

Based on the RCHW value, the BAM configures the memory management unit (MMU) to a default configuration that is suitable for use by many applications without modification. This configuration can optionally perform other initializations as well, such as configuring VLE mode and enabling the watchdog timer. When the BAM completes execution, program control branches to the low-level user initialization code.

Only the e200z1 core executes BAM code. At a POR, the e200z0 core is held in reset until enabled by the application code running on the e200z1 core.

3.2 Basic Dual-Core Flash Boot — Program Flow

This section contains a description of the basic flow of control during a dual-core boot. The key concept to understand is that in the case of the MPC551x devices, a dual-core boot is nothing more than a typical single core boot on the e200z1 core, with additional code that prepares the e200z0 for operation and releases it from reset. A flow chart of the boot sequence is shown in Figure 3.

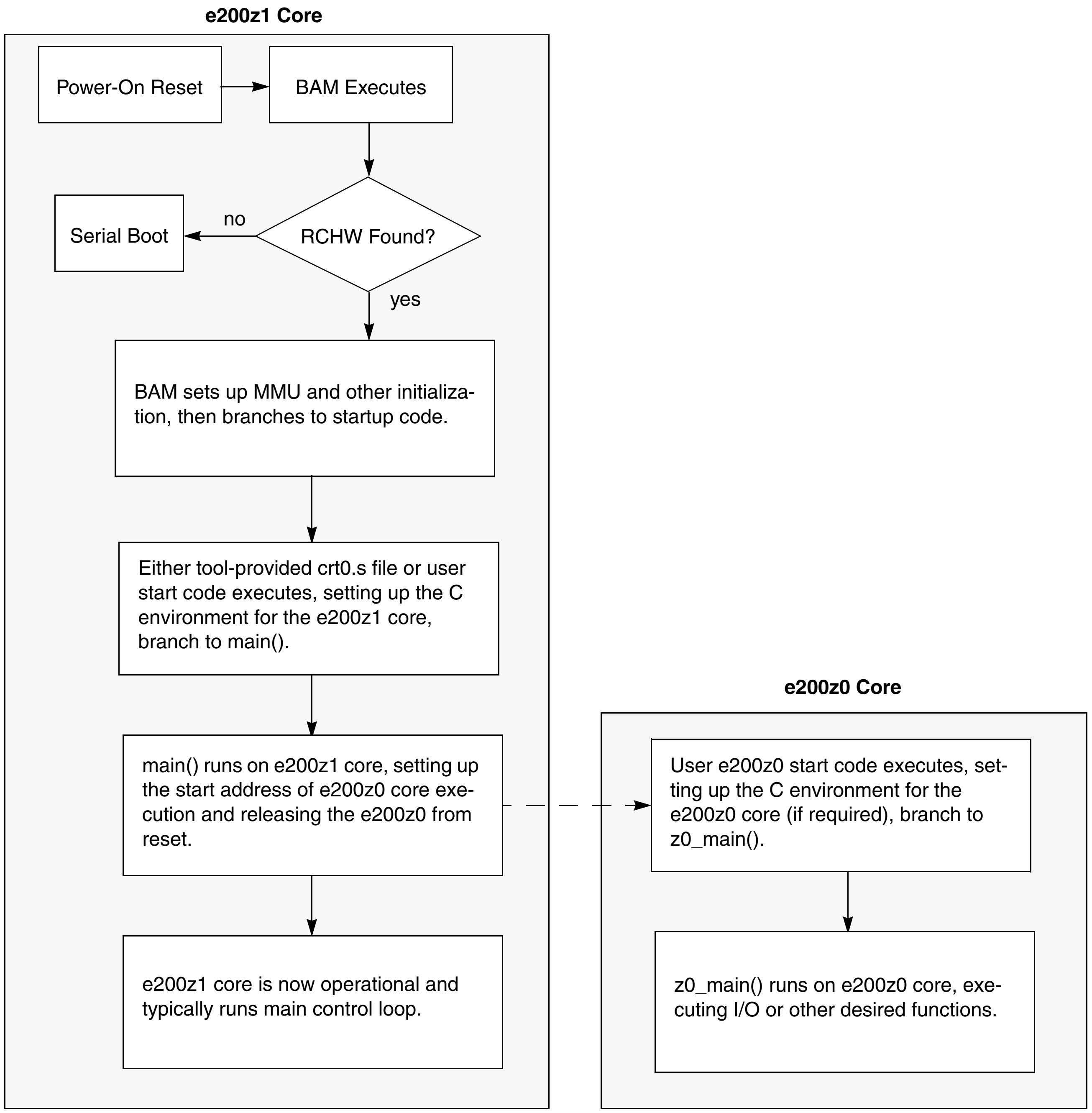
e200z1 Core
Power-On Reset
BAM Executes
Serial Boot
no
RCHW Found?
yes
BAM sets up MMU and other initialization, then branches to startup code.
Either tool-provided crt0.s file or user start code executes, setting up the C environment for the e200z1 core, branch to main().
main() runs on e200z1 core, setting up the start address of e200z0 core execution and releasing the e200z0 from reset.
e200z1 core is now operational and typically runs main control loop.
e200z0 Core
User e200z0 start code executes, setting up the C environment for the e200z0 core (if required), branch to z0_main().
z0_main() runs on e200z0 core, executing I/O or other desired functions.

Figure 3. Flash Boot Sequence — Dual-Core Operation

4 Reset Vector Registers

There are two Reset Vector registers, Z1 Reset Vector Register (CRP_Z1VEC) and Z0 Reset Vector Register (CRP_Z0VEC). These two registers respectively control the starting address of code execution at a core reset for the e200z1 and e200z0 cores. The function of these registers, and how they affect a boot into dual-core operation, is described in the next subsections.

4.1 Z1 Reset Vector Register (CRP_Z1VEC)

The CRP_Z1VEC bit definitions are shown in the register diagram below.

Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R W	Z1 START ADDRESS																														Z1RST	VLE
Reset	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1

Figure 4. CRP_Z1VEC

The Z1VEC value determines the initial program counter for the Z1 on exiting reset. On POR, the value contained in the register defaults to 0xFFFF_FFFD, so that the Z1 fetches VLE code from the BAM starting at address 0xFFFF_FFFC. The Z1RST bit is cleared by default at POR, allowing the Z1 core to exit reset.

4.2 Z0 Reset Vector Register (CRP_Z0VEC)

The CRP_Z0VEC bit definitions are shown in the register diagram below.

Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R	Z0 START ADDRESS																														Z0RST	0
W																																▨
Reset	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0

Figure 5. CRP_Z0VEC

The Z0VEC value determines the initial program counter for the Z0 upon exiting reset. On POR, the value contained in the register defaults to 0xFFFF_FFFE. The Z0RST bit is set by default at POR, so the Z0 core remains in a reset state until initially released from reset by the Z1 core.

When the e200z1 core is ready to start the e200z0 core, the e200z0 start address is written to the CRP_Z0VEC register, and the Z0RST bit is cleared. These two operations can be performed as a single 32-bit write.

Note that the two LSBs on both the CRP_Z0VEC and the CRP_Z1VEC registers are not used as part of the reset vector. This means that the actual address where execution begins must be word-aligned in both cases.

5 Examples of Dual-Core Initialization Code

This section contains a linker descriptor file and source code that will perform an initialization and startup of the e200z1 and e200z0 cores. This example code has been assembled, compiled, and linked using the Green Hills Systems MULTI 5.0.3 development environment, and tested on the MPC5516 EVB available from Freescale. The compiler, assembler, and linker file syntax may differ slightly depending on the tool set used — however, the essential structure and function are applicable.

Note also that this example is linked as a single .elf image. With most compilers that support the dual-core MPC5516 devices (and other dual-core devices) it is also possible to create separate .elf files for each core. This example does not discuss that method, and the software engineer should consult the documentation for his compiler if multiple .elf files are a requirement of the application or development process.

5.1 Linker Configuration

This is an example of a linker configuration file. Note that in this example stack space has been designated in the MEMORY section for both cores. Depending on the intended function of the z0 core, z0 stack space may not be required.

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

     FREESCALE HEADER??????

   ---------------------------------------------------------------------*/

MEMORY
{
     rchw:          org = 0x0,        len = 8
     flash_memory:  org = .,          len = 1M - 8
     dram_memory:   org = 0x40000000, len = 64k-8k /* 64k - 8k bytes of stack space RAM */
     stack_z0:      org = 0x4000e000, len = 4k     /* 4k z0 stack */
     stack_z1:      org = .,          len = 4k     /* 4k z1 stack */

}

DEFAULTS {
     heap_reserve  = 4k
     stack_reserve = 4K

}

SECTIONS
{
    /* RAM SECTIONS */
    .PPC.EMB.sdata0                         ABS  : > dram_memory
    .PPC.EMB.sbss0                     CLEAR ABS : > .
    .sdabase                           ALIGN(16) : > .
    .sdata                                       : > .
    .sbss                                        : > .
    .data                                        : > .
    .bss                                         : > .
    .heap       ALIGN(16) PAD(heap_reserve)      : > .
    .stack_z1   ALIGN(16) PAD(stack_reserve)     : > .
    .stack_z0   ALIGN(16) PAD(stack_reserve)     : > .

    /* ROM SECTIONS */
    .text                                        : > flash_memory
    .vletext                                     : > .
    .syscall                                     : > .
    .rodata                                      : > .
    .sdata2                                      : > .
    .secinfo                                     : > .
    .fixaddr                                     : > .
    .fixtype                                     : > .
    .CROM.PPC.EMB.sdata0 CROM(.PPC.EMB.sdata0): > .
    .CROM.sdata                 CROM(.sdata) : > .
    .CROM.data                  CROM(.data)  : > .

/* Definitions of identifiers used by the z1_crt0.s and z0_crt0.s  */
     __HEAP_START= ADDR(.bss)+SIZEOF(.bss);
     __SP_INIT= ADDR(stack_z1)+SIZEOF(stack_z1);
     __HEAP_END= ADDR(dram_memory)+SIZEOF(dram_memory);
     __SP_END= ADDR(stack_z1);
     __DATA_ROM= ADDR(.sdata2)+SIZEOF(.sdata2);
     __DATA_RAM= ADDR(.data);
     __DATA_END= ADDR(.sdata)+SIZEOF(.sdata);
     __BSS_START= ADDR(.sbss);
     __BSS_END= ADDR(.bss)+SIZEOF(.bss);
     __SP_INIT_Z0 = ADDR(stack_z0) + SIZEOF(stack_z0);
     __SP_END_Z0  = ADDR(stack_z0);
}
```

5.2 e200z1 Initialization

This section contains a very simple example of the typical function of the crt0.s file that is included as part of most compilers. This code illustrates the basic function required to configure the Power Embedded Application Binary Interface (EABI) environment. In some cases the tool vendor will recommend the use of the vendor provided crt0.s initialization code, to ensure that the C environment is configured correctly for that vendor's compiler. Consult the documentation provided with your development tools.

```
# define RCHW
     .globl         __start
     .section       .rchw
     .long          0x015a0000             # define the RCHW value, VLE bit is set
     .long          __start                # define the __start address

     .section       .vletext, "avx"        # syntax to locate this code in .vletext section
     .vle                                   # tell assembler this is VLE code
# z1 start point, RCHW vector points here
__start:
     e_lis           r1, __SP_INIT@ha      # Initialize stack pointer r1 to
     e_or2i          r1, __SP_INIT@l        # value in linker command file.
     e_lis           r13, _SDA_BASE_@ha    # Initialize r13 to sdata base
     e_or2i          r13, _SDA_BASE_@l     # (provided by linker).
     e_lis           r2, _SDA2_BASE_@ha    # Initialize r2 to sdata2 base
     e_or2i          r2, _SDA2_BASE_@l     # (provided by linker).

     e_stwu          r0,-64(r1)             # Terminate z1 stack.

     e_bl           init_SRAM              # initialize SRAM (this code is not included
                                           # in this example
#
# Insert other initialize code here.
#

     e_b            main
```

5.3 e200z1 Main Loop

This is sample code that illustrates the procedure for releasing the z0 core from reset and pointing it to the appropriate initialization code. The while(1) loop contains code to flash an LED on the MPC5516 EVB to indicate that the z1 core is running the main loop code.

```
#include "mpc5516.h"

extern void _start_z0( void );         /* this is the z0 start up code */

void main( void ){
     unsigned int temp = 0;

     /* start z0 core by writing address of z0 start code to CRP.Z0VEC.R.
        This write also clears the Z0RST bit which releases the z0 core from
        reset. */
     CRP.Z0VEC.R = (unsigned int)_start_z0;

     /* Flash LED on PB0 of EVB to indicate z1 core is functioning in main loop */
     SIU.PCR[16].R =0x0200;
        while(1){
                 for(temp = 0; temp <100000; temp++);
              SIU.PGPDO0.R = 0x0000;
           for(temp = 0; temp <100000; temp++);
           SIU.PGPDO0.R = 0x8000;
       };

} /* END of main */
```

5.4 e200z0 Initialization

This section contains a very simple example of the basic function required to configure the Power Embedded Application Binary Interface (EABI) environment for the z0 core. Depending on the intended function of the z0 core, some parts of this configuration may not be required.

```
# z0 core start code
     .globl          _start_z0

     .section  .vletext, "axv"             # syntax to locate this code in .vletext section
     .vle                                  # tell assembler this is VLE code
     .align 2                               # this function must be word aligned
                                            # See Section 4.2, on page 4.2

# main.c on z1 core writes the address of this label to the Z0
_start_z0:
        e_lis          r1, __SP_INIT_Z0@ha   # Initialize stack pointer r1 to
        e_or2i          r1, __SP_INIT_Z0@l    # value in linker command file.
        e_lis           r13, _SDA_BASE_@ha    # Initialize r13 to sdata base
        e_or2i          r13, _SDA_BASE_@l    # (provided by linker).
        e_lis           r2, _SDA2_BASE_@ha   # Initialize r2 to sdata2 base
        e_or2i          r2, _SDA2_BASE_@l     # (provided by linker).

        e_stwu          r0,-64(r1)            # Terminate z0 stack.

#
# Insert other initialization code here.
#

        e_b             z0_main
```

5.5 e200z0 Main Loop

This is sample code that illustrates a typical z0_main loop. In this example, the while(1) loop contains code to flash an LED on the MPC5516 EVB to indicate that the z0 core is running the main loop code.

```
#include "mpc5516.h"

void z0_main(){

     unsigned int temp = 0;

     /* Flash LED on PC0 of EVB to indicate z0 core is functioning in main loop */
     SIU.PCR[32].R =0x0200;

     while(1){
          for(temp = 0; temp <100000; temp++);
          SIU.PGPDO1.R = 0x80000000;
          for(temp = 0; temp <100000; temp++);
          SIU.PGPDO1.R = 0x00000000;
     };

} /* END of z0_main */
```

6 Summary

Initializing and starting both cores on the MPC5516G/E devices involves minimal modification and additions to the code required for a normal single core boot. This application note has described the initialization flow during a POR reset, and illustrated the software requirements with simple, but functional, sample initialization code and a linker descriptor file.

The example shown here is not intended to be a template for an application, but rather to illustrate in an easy-to-understand manner the process of a dual-core boot. Many of the Freescale tool partners support the dual-core devices in the MPC5516 family. The software engineer should always refer to the documentation for his or her particular compiler for complete information regarding configuring and building applications for dual-core devices.

THIS PAGE IS INTENTIONALLY BLANK

Document Number: AN3627
Rev. 0
04/2008