MPC8641 Bare-Metal Application Template (BMAT) Guide

Tennessee Carmel-Veilleux <tennessee.carmelveilleux@gmail.com>

October 6, 2010

1 Introduction

The MPC8641 bare-metal application template (BMAT) allows quick prototyping and testing of drivers and interrupt handling code with Simics on the mpc8641-simple target.

The following features are provided by the template:

- Startup code for C-language applications (equivalent to crt0.S, calls `main()`);
- Free-standing implementation (not reliant on any standard library files or headers);
- Minimal C library provided for "printf" and memory copy operations;
- Linker configuration script with static addressing;
- Compatibility with U-Boot:
  - ▷ "GO" command argument processing with the familiar `int main(int argc, char **argv)` prototype;
  - ▷ board data structure access if available;
  - ▷ console integration if desired.
- Full exception handling support:

Table 1: Supported environments for BMAT building

Environment	Version
Linux	≥ 2.0
Cygwin	≥ 1.7

  - ▷ handler programmable per exception;
  - ▷ C or assembly language handlers possible;
  - ▷ default handlers to prevent crashing on unhandled exceptions.
- PIC support:
  - ▷ handler programmable per interrupt source;
  - ▷ C language handlers possible;
  - ▷ interrupt nesting managed by external interrupt exception handler;
  - ▷ constants and tables to access all VPR and DR registers.

2 Build

2.1 Toolchain and build environment

The BMAT was built and tested under Linux and Cygwin. Supported versions are shown in table 1.

The toolchain used was Sourcery G++, which is a commercially supported version of GCC. The GCC target was `powerpc-eabi`. Tool versions used are shown in table 2.

2.2 Building

The BMAT build is based on a single Makefile. The following make targets are provided:

- "all" (or "bare-metal-app"): builds the "`bare-metal-app`" binary file from sources.
- "clean": removes output files, temporary files and objects.

The default configuration of the Makefile should be sufficient for most applications. By default, the resulting binary will be located for loading at 0x0010_0000.

Table 2: Tool versions for BMAT building

Function	Tool	Version
PowerPC C cross-compiler	GNU powerpc-eabi-gcc	4.4.1 (Sourcery G++ 4.4-73)[1]
PowerPC cross-linker	GNU powerpc-eabi-ld	2.19.51 (Sourcery G++ 4.4-73)
PowerPC cross-assembler	GNU powerpc-eabi-as	2.19.51 (Sourcery G++ 4.4-73)
PowerPC debugger	GNU powerpc-eabi-gdb	6.8.50 (Sourcery G++ 4.4-73)
Make	GNU Make	3.81
Virtual platform	Wind River Simics core	4.0.60
PowerPC e600 model	Wind River Simics mpc8641-simple	4.0.12

[1]Sourcery G++ is a commercial distribution of GNU GCC. It is available at: `https://www.codesourcery.com/sgpp`.

2.3 Build configuration

The source code is configured through the "`conf86xx.h`" header file (see section 4.3). It is included implicitly on every invokation of the compiler or assembler. The following constants are defined therein:

CONFIG_START_ADDRESS Static start address of binary (must be aligned to 1 MBytes (2^{20} bytes)).

CONFIG_CCSR_BASE CCSR block base address as configured by U-Boot.

CONFIG_STACK_SIZE Stack size in KBytes.

CONFIG_NO_HWIRQS Number of hardware IRQ sources supported by the PIC (including the internal sources).

CONFIG_PIC_NUM_MSIR Number of Message Signaled Interrupt Registers supported by the PIC.

KERNEL_MSR MSR value to use for calling main().

3 Boot process

3.1 Loading the BMAT binary into Simics

Loading of the BMAT binary can be accomplished with the "`load-binary`" Simics command. It is that easy. Since the binary is in ELF format and the Makefile selects debugging

during compilation, users can also load a symbol table from the binary.

See the "`Simics/bare_metal_test.simics`" file for a complete usage example.

3.2 Boot process overview

The BMAT boot process depends on U-Boot for machine initialization. Therefore, a Simics script or checkpoint must be used to ensure U-Boot initialized the platform prior to running the BMAT "`bare-metal-app`" binary. The use of the GO command by U-Boot is however not necessary. Neither is it necessary to use any features of U-Boot other than basic system bring-up.

Once the system has been brought-up by U-Boot, the Simics simulation can be stopped. The BMAT binary is then loaded and run with one of two methods:

1. Running the entrypoint (possibly with parameters) using the U-Boot "GO" command
    - Example: "`go 100100 param1 param2`", where 100100 is the hexadecimal address of the entrypoint (see next section for an explanation).
2. Setting the PC to the entrypoint and starting the simulation
    - Example: "`cpu0->pc = 0x00100100 ; run`", where 0x100100 is the hexadecimal address of the entrypoint

3.3 Boot process details

The boot process used by BMAT is a simple C environment bring-up based on the assumption that the binary image was fully loaded into memory either by bootloading or by direct memory writes. The linker script ensures that the first 8KBytes of the image is a valid PowerPC exception table. The entrypoint of BMAT is the reset vector of that exception table (offset 0x0100 from the start). This is why the address used to start BMAT in the previous section is 0x0010_0100 instead of 0x0010_0000.

Note that you can reconfigure the memory placement of BMAT (see parameter "CONFIG_START_ADDRESS" in section 2.3). Since the exception table is remapped using the Boot Page Translation (BPTR) mecanism of the MPC8641 SoC, the CONFIG_START_ADDRESS value must respect the BPTR requirement of 1 Megabytes boundary alignment.

The boot process defined in `start.S` follows these steps:

1. Disable interrupts and reset the decrementer
2. Zero-out the BSS section
3. Save U-Boot argc and argv data
4. Setup a valid stack with SP in GPR1
5. Setup EABI registers GPR2 and GPR13 to point to small data areas
6. Setup the interrupts subsystem by calling SetupIrqs from irqs.c
    a) Fill the interrupt and exception handler tables with default values
    b) Reset the PIC
    c) Initialize every PIC source VPR register with a preset vector and masked flag
7. Setup call to `main()`
    a) Set parameters `argc` and `argv`
    b) Set SRR0 to address of `main()`
    c) Set SRR1 to KERNEL_MSR (configuration of MSR for the duration of the test)
    d) Set LR to `_halt_system()` so that returning for `main()` enters an endless loop
8. Branch to main with context-synchronizing RFI

After step 8 above, `main()` is entered and the user can use the API described in section 5.

4 Files

The BMAT is composed of a minimal number of files so that only the strict minimum required functionality for basic applications is preserved. Apart from libgcc.a (compiler internal stubs), no other library is linked and no C standard library headers are used.

The following sections list the files and briefly describes their contents.

4.1 Build-related files

./Makefile Main makefile to build the BMAT.

src/bare-metal-app.lds.in Linker script input file. This linker script file is preprocessed with the C macro preprocessor to yield `./bare-metal-app.lds`, which is used for final linking.

4.2 Simics-related files

Simics/bare-metal-booted.ckpt* Checkpoint of a booted mpc8641-simple system, having gone through U-Boot and ready for loading and execution of the BMAT

Simics/bare_metal_test.simics Simics script to load and test the BMAT sample application. See the file comments for details.

4.3 Header files

include/conf86xx.h Main build configuration parameters (see section 2.3).

include/immap_86xx.h Internal memory map of the MPC86xx SoCs. Every hardware register is accessible through the `immr` volatile global structure (see section 5.1).

include/irqs.h Interrupts and exception related prototypes and constants

include/ppc_asm.h Minimal GPR and CR definitions as well as Simics magic breakpoint macro for inclusion in assembly-language files.

include/ppc_asm_handlers.h Helper constants and macros for exception table construction in `start.S`.

include/ppc_reg.h SPR accessor macros and SPR bit definitions for the most common PowerPC SPRs.

include/processor.h Miscellaneous PowerPC I/O and synchronization macros (see section 5.4).

include/stdc.h Minimal C standard library prototypes (see section 5.3).

include/uboot.h Definitions for U-Boot 1.3.0 global data structures access.

4.4 Source files

src/irqs.c PIC and exception handling constants, tables and functions (see section 5.2).

src/main.c Sample mainline file with API usage example (see section 6).

src/stdc.c Minimal standard library support (see section 5.3).

src/start.S Exception table and startup code.

5 API reference

5.1 Internal memory map access (include/immap_86xx.h)

The `include/immap_86xx.h` file provides a single-point access to the internal register space of the MPC8641 SoC. The main type is `immap_t` (lines 1-26 in the listing below) and the accessor is the `immr` macro (line 28).

```
1  typedef struct immap {
2      ccsr_local_mcm_t  im_local_mcm;
3      ccsr_ddr_t        im_ddr1;
4      ccsr_i2c_t        im_i2c;
5      ccsr_duart_t      im_duart;
6      ccsr_lbc_t        im_lbc;
7      ccsr_ddr_t        im_ddr2;
8      uint8_t           res1[4096];
9      ccsr_pex_t        im_pex1;
10     ccsr_pex_t        im_pex2;
11     ccsr_ht_t         im_ht;
12     uint8_t           res2[90112];
13     ccsr_dma_t        im_dma;
14     uint8_t           res3[8192];
15     ccsr_tsec_t       im_tsec1;
16     ccsr_tsec_t       im_tsec2;
17     ccsr_tsec_t       im_tsec3;
18     ccsr_tsec_t       im_tsec4;
19     uint8_t           res4[98304];
20     ccsr_pic_t        im_pic;
21     uint8_t           res5[389120];
22     ccsr_rio_t        im_rio;
```

```
23      ccsr_gur_t        im_gur;
24      uint8_t           res6[12288];
25      ccsr_wdt_t        im_wdt;
26  } immap_t;
27
28  #define immr ((volatile immap_t *)(CONFIG_CCSR_BASE))
```

For instance, to access the Feature Reporting Register (FRR) register of the PIC module, one could write:

```
uint32_t value = immr->im_pic.frr;
```

To write to the UMCR1 register of the DUART module, one could write:

```
immr->im_duart.umcr1 = (uint8_t)value;
```

The `immr` macro is an explicit `volatile` cast, so values read and written through it are not cached by the compiler.

5.2 Interrupt and exception handling (include/irqs.h)

A simple API is provided to hook handlers into the exception table or onto PIC interrupt sources. Macros and functions are also provided to enable and disable the external interrupts globally through manipulation of the MSR[EE] bit.

5.2.1 Exception and interrupt context structure

A pointer to saved context (`irqCtxt_t *p_ctxt`) is provided using a parameter to every interrupt and exception handler called. The irqCtxt_t structure is defined below:

```
typedef struct irqCtxt {
    uint32_t gpr[32]; /* General purpose registers */
    uint32_t nip; /* Return address (from SRR0) */
    uint32_t msr; /* Saved MSR (from SRR1) */
    /* Cause of exception/interrupt:
     * - For exceptions: number of handler, ie: 6 (instead of 0x0600) for alignment exception)
     * - For external interrupts coming for PIC: number of the interrupt (0 through PIC->FRR.
         NIRQ)
     */
```

```
    uint32_t irqNr;
    /* Saved SPRs */
    uint32_t ctr;
    uint32_t lr;
    uint32_t xer;
    uint32_t ccr;
    /* Fault registers */
    uint32_t dar;
    uint32_t dsisr;
    uint32_t result; /* Result of a system call (unused and cleared) */
    uint32_t error; /* Error condition to be reported if necessary */
} irqCtxt_t;
```

5.2.2 Exception handling

The exception table defined in start.S calls an exception dispatch function for all exceptions except the following:

- Reset, obviously
- Alignment exception, which calls `DoAlignment()`, see below
- Program check exception which calls `DoProgramCheck()`, see below
- External interrupts, which are dealt with separately (see section 5.2.3)
- Decrementer interrupt, which call `DoDecrementer()` (see section 5.2.3)

The following are documentation entries pulled from the source code:

```
typedef void (*exceptionHandler_t)(irqCtxt_t *);

/**
 * Alignment exception handler.
 *
 * @param p_ctxt - pointer to saved interrupt context
 * @param dar - value of DAR on exception entry
 * @param dsisr - value of DSISR on exception entry
 */
void __attribute__ ((weak)) DoAlignment(irqCtxt_t *p_ctxt, uint32_t dar, uint32_t dsisr);

/**
```

```
 * Default exception handler called when no handler is set.
 *
 * @param p_ctxt - pointer to saved exception context
 */
void __attribute__ ((weak)) UnexpectedExceptionHandler(irqCtxt_t *p_ctxt);

/**
 * Handle a program check exception on the PowerPC.
 *
 * @param p_ctxt - pointer to interrupt saved context.
 */
void __attribute__ ((weak)) DoProgramCheck(irqCtxt_t *p_ctxt);

/**
 * Set the exception handler for one of the internal exceptions.
 *
 * @param exceptionNr - Exception number (0-31)
 * @param exceptionHandler - Exception handler function pointer
 * @return a pointer to the previously set exception handler.
 */
exceptionHandler_t SetExceptionHandler(int32_t exceptionNr, exceptionHandler_t exceptionHandler
    );
```

As can be seen from the code, the default versions of the predefined handlers can be redefined in the user's own code since they are weak references.

An example of using the `SetExceptionHandler()` function to hook onto the System Call exception is provided in the example main.c file that comes with BMAT (see section 6).

5.2.3 PIC IRQ handling

The IRQ handling subsystem of the BMAT is programmed to support full nesting of prioritized interrupts as per the OpenPIC specification.

For each interrupt source, an interrupt handler can be registered. A single data pointer can also be registered and it will be passed as parameter to the handler every time the interrupt occurs.

The following are documentation entries pulled from the source code:

```
typedef void (*irqHandler_t)(irqCtxt_t *, void *);

/**
 * Default IRQ handler called when no handler is set.
 *
 * @param p_ctxt - pointer to saved interrupt context
 * @param p_data - pointer to IRQ-specific data (always NULL in this case)
 */
void __attribute__ ((weak)) DefaultIrqHandler(irqCtxt_t *p_ctxt, void *p_data);

/**
 * Decrementer interrupt handler.
 *
 * @param p_ctxt - pointer to saved interrupt context
 */
void __attribute__ ((weak)) DoDecrementer(irqCtxt_t *p_ctxt);

/**
 * Spurious IRQ handler called when a PIC spurious IRQ occurs.
 *
 * @param p_ctxt - pointer to saved interrupt context
 * @param p_data - pointer to IRQ-specific data (always NULL in this case)
 */
void __attribute__ ((weak)) SpuriousIrqHandler(irqCtxt_t *p_ctxt, void *p_data);

/**
 * Set the IRQ handler for one of the PIC's interrupt sources.
 *
 * @param irqNr - PIC IRQ source number (0-127)
 * @param irqHandler - IRQ handler function pointer
 * @param p_data - Pointer to IRQ-specific data structure that gets passed at every handler
     invokation
 * @return a pointer to the previously set IRQ handler
 */
irqHandler_t SetIrqHandler(int32_t irqNr, irqHandler_t irqHandler, void *p_data);
/**
 * Disable (mask) the specified IRQ at the controller.
 * @param irqNr - Hardware IRQ number
 */
void DisableIrq(uint32_t irqNr);
```

```
/**
 * Enable (unmask) the specified IRQ at the controller.
 * @param irqNr - Hardware IRQ number
 */
void EnableIrq(uint32_t irqNr);

/**
 * Set the priority of the specified IRQ at the controller.
 *
 * The priority value is architecture-specific.
 *
 * @param irqNr - Hardware IRQ number
 * @param priority - Priority to set for the IRQ
 */
void SetPriorityIrq(uint32_t irqNr, uint32_t priority);

/**
 * Disable External Interrupts by setting MSR[EE] to 0
 */
static inline void GlobalIrqDisable(void);

/**
 * Enable External Interrupts by setting MSR[EE] to 1
 */
static inline void GlobalIrqEnable(void);

/**
 * Disable Interrupts, returning state of MSR[EE] before the disabling
 *
 * @return value containing at least a valid MSR[EE] bit in the right position
 */
static inline uint32_t __SaveFlagsIrqDis(void);

/**
 * Saves interrupt enable flag before disabling interrupts.
 * Equivalent to "flags = __SaveFlagsIrqDis()".
 */
#define GlobalIrqDisableSaveFlags(flags) ((flags)=__SaveFlagsIrqDis())
```

```
/**
 * Restores MSR[EE] from a previously saved MSR
 *
 * @param flags - value containing MSR[EE] in its proper bit place
 */
static inline void GlobalIrqRestoreFlags(uint32_t flags);

/** Sets "flags" to interrupt enable state */
#define GlobalIrqSaveFlags(flags) ((flags) = (mfmsr() & MSR_EE))

/** @return non-zero if interrupts are enabled */
static inline uint32_t IsGlobalIrqEnabled(void);
```

IRQ source numbers are defined with constants starting with "HWIRQ_". There are several tables that can help in working with PIC interrupt sources:

```
/** MSIR registers table */
uint32_t *g_p_PIC_msir[CONFIG_PIC_NUM_MSIR];

/** IRQ number to VPR register table */
uint32_t *g_p_PIC_irq2vpr[CONFIG_NO_HWIRQS];

/** IRQ number to IDR register table */
uint32_t *g_p_PIC_irq2idr[CONFIG_NO_HWIRQS];

/** IRQ source names */
char *g_p_PIC_irq_names[CONFIG_NO_HWIRQS];
```

An example of using these function and table to hook onto the PIC's messaging tnterrupts is provided in the example main.c file that comes with BMAT (see section 6).

5.3 Minimal C standard library (include/stdc.h)

This include file defines minimal functionality for memory block copying and a simple kprintf() implementation. The code from stdc.c is the hack of a hack of a hack. I did not write it :) It comes from a long line of adaptations for embedded code. The original version was written by Volker Oth as a test for AVR-GCC.

The following memory-related functions are defined:

- `void *memset(void *dst, int32_t s, uint32_t count)`
- `void *memcpy(void *dst, const void *src, uint32_t count)`
- `int32_t memcmp(const void *dst, const void *src, uint32_t count)`
- `uint32_t strlen(const int8_t *s)`

These functions are pretty much identical in nature to their C standard library cousins, apart from their slow speeds (generic implementations) and odd parameter types.

The `printf()` implementation has this prototype:

- `int32_t kprintf(const char *fmt, ...);`

You must provide an implementation of kputchar if you want a console output.

- `void kputchar(int32_t c);`

The sample application (see section 6) provides an implementation of `kputchar()` that outputs on the console configured by U-Boot.

5.4 PowerPC SPR access (include/ppc_reg.h)

This include file contains macros to read and set SPRs, as well as constants for common SPR numbers and their contents. Look at the contents of the file (or its outline in your IDE) for a full list of constants.

The SPR access macros follow the equivalent assembly mnemonic:

- `mfmsr()`
- `mtmsr(value)`
- `mfdec()`
- `mtdec(value)`
- `mftbu()`
- `mttbu(value)`
- `mftb()`
- `mttbl(value)`
- `mfspr(sprn)`
- `mtspr(sprn, value)`

5.5 I/O and synchronization (include/processor.h)

5.5.1 I/O functions

These functions perform input or output of 8, 16 or 32 bit values in memory-mapped I/O regions with standard (big-endian) byte ordering.

The input functions are crafted such that a read is forced and that a barrier occurs for the value returned. A call to any of these functions thus insures that the value is immediately read with a bus transaction and that ordering is total:

- `uint32_t io_in8(const volatile uint8_t *addr)`
- `uint32_t io_in16be(const volatile uint16_t *addr)`
- `uint32_t io_in32be(const volatile uint32_t *addr)`

The output functions are similar to the read functions in that the transaction is forced and synchronized. The output functions are as follows:

- `void io_out8(volatile uint8_t *addr, uint32_t val)`
- `void io_out16be(volatile uint16_t *addr, uint32_t val)`
- `void io_out32be(volatile uint32_t *addr, uint32_t val)`

5.5.2 Synchronization functions

The following PowerPC synchronization barriers are provided for easy use in C code:

- `void eieio(void)`
- `void sync(void)`
- `void isync(void)`

6 Sample application

A sample application is provided with BMAT. It demonstrates the use of the API presented in section 5. The source code is in the `src/main.c` file. Users can replace the entire contents of that file with their own code, or extend the sample application.

6.1 Basic example flow

The sample application executes the following step starting from the main() function:

1. Setup the U-Boot console for use with kprintf().
2. Setup an IRQ test which tests the PIC IRQ subsystem.
3. Print "Hello World !" on the console.
4. Trigger the IRQ test.
5. Trigger a trap.
6. Trigger a system call prior to setting a handler for it, thus generating an unhandled exception call.
7. Set the exception handler for the system call so that the next system call will be handled correctly.
8. Trigger a system call that gets handled properly.
9. Trigger an illegal instruction exception.
10. Print "Done !" on the console.
11. Exit main() with exit code 6502 (which gets printed by the Simics script).

6.2 IRQ test flow

The IRQ test shows the use of nested interrupts and how to handle unexpected interrupts. It is setup using the "`setup_irq_test()`" function call from the mainline.

The idea behind the test is that we can trigger some PIC messaging IRQs manually and show what happens. In this case, we trigger messaging IRQ 0, for which we have set a valid handler. This handler in turn triggers messaging IRQ 1, which has a higher priority and no handler set. This causes a nested interrupt response as well as a call to the default exception handler. See figure 1 for a sequence diagram of the test.

See the source code in `src/main.c` for more details about the test.

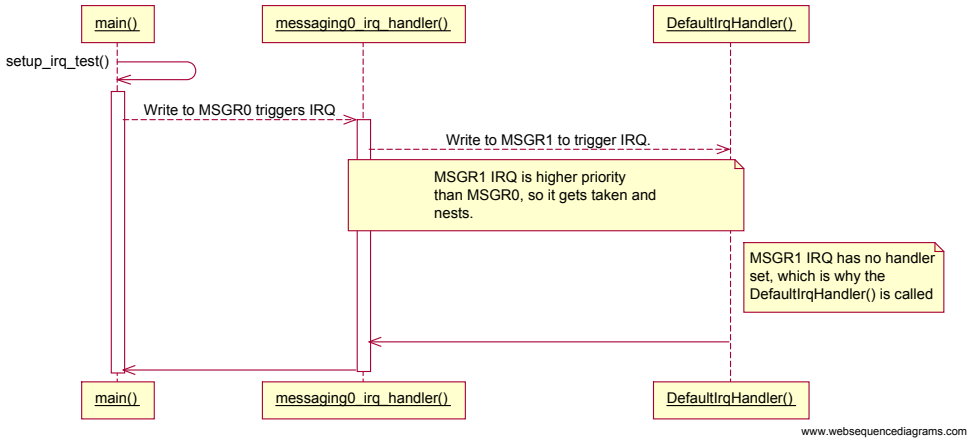

Figure 1: Sequence diagram of the IRQ test

7 Conclusion

The BMAT is a simple MPC8641 bare-metal application template that allows users to easily develop low-level code on the MPC8641-simple Simics model. Exceptions, PIC interrupts and U-Boot console support ease the testing of driver and low-level code.