EDK 9.1i January 8, 2007

Standalone Board Support Package

Summary

The Board Support Package (BSP) is the lowest layer of software modules used to access processor specific functions. The standalone BSP is used when an application accesses board/processor features directly and is below the operating system layer.

This document contains the following sections.

- "MicroBlaze BSP"
- "PowerPC BSP"
- "Program Profiling"
- "Configuring the Standalone BSP"

MicroBlaze BSP

When your system contains a MicroBlaze™ processor and no operating system, the Library Generator automatically builds the standalone BSP in the project library `libxil.a`.

Function Summary

The following table contains a list of all MicroBlaze BSP functions.

Table 1: **Function Summary**

Functions
`void microblaze_enable_interrupts(void)`
`void microblaze_disable_interrupts(void)`
`void microblaze_register_handler (XInterruptHandler Handler, void *DataPtr)`
`void microblaze_disable_exceptions(void)`
`void microblaze_enable_exceptions(void)`
`void microblaze_register_exception_handler (Xuint8 ExceptionId, XExceptionHandler Handler, void *DataPtr)`
`void microblaze_enable_icache(void)`
`void microblaze_disable_icache(void)`
`void microblaze_init_icache_range (int cache_addr, int cache_size)`
`void microblaze_enable_dcache(void)`
`void microblaze_disable_dcache(void)`
`void microblaze_init_dcache_range (int cache_addr, int cache_size)`

Interrupt Handling

The following functions help manage interrupt handling on MicroBlaze devices. You must include the header file `mb_interface.h` in your source code to use these functions.

`void` **`microblaze_enable_interrupts`** `(void)`

This function enables interrupts on the MicroBlaze. When the MicroBlaze starts up, interrupts are disabled. Interrupts must be explicitly turned on using this function.

`void` **`microblaze_disable_interrupts`** `(void)`

This function disables interrupts on the MicroBlaze. This function can be called when entering a critical section of code where a context switch is undesirable.

`void` **`microblaze_register_handler`** `(XInterruptHandler Handler, void` *`*DataPtr`*`)`

This function allows you to register the interrupt handler for the MicroBlaze processor. This handler is invoked in turn, by the first level interrupt handler that is present in the BSP. The first level interrupt handler takes care of saving and restoring registers, as necessary for interrupt handling and hence the function that you register with this handler can concentrate on the other aspects of interrupt handling, without concern about saving registers.

Exception Handling

This section describes the exception handling functionality available on the MicroBlaze processor.

Note: This feature and hence the corresponding interfaces are not available on versions of MicroBlaze older than v3.00.a.

The handlers for the various exceptions can be specified in the parameter `microblaze_exception_vectors` in the Xilinx® Platform Studio (XPS) software platform settings. In the field corresponding to the exception type, specify the name of the routine that you wish to handle a particular exception. You can pass in callback parameters (can only be literal C constants) in the second field.

The following functions help manage exceptions on MicroBlaze. You must include the header file `mb_interface.h` in your source code to use these functions.

`void` **`microblaze_disable_exceptions`** `(void)`

Disable hardware exceptions from the MicroBlaze processor. This routine clears the appropriate “exceptions enable” bit in the model-specific register (MSR) of the processor.

`void` **`microblaze_enable_exceptions`** `(void)`

Enable hardware exceptions from the MicroBlaze processor. This routine sets the appropriate “exceptions enable” bit in the MSR of the processor.

void **microblaze_register_exception_handler** (Xuint8 *ExceptionId*, XExceptionHandler *Handler*, void **DataPtr*)

Register a handler for the specified exception type. `Handler` is the function that handles the specified exception. `DataPtr` is a callback data value that is passed to the exception handler at run-time. By default the exception ID of the corresponding exception is passed to the handler. The valid exception IDs, which are defined in microblaze_exceptions_i.h, are described in the following table.

Table 2: **Valid Exception IDs**

Exception ID	Value	Description
XEXC_ID_UNALIGNED_ACCESS	1	Unaligned access exceptions
XEXC_ID_IOPB_EXCEPTION	2	Exception due to a timeout from the IOPB bus
XEXC_ID_ILLEGAL_OPCODE	3	Exception due to an attempt to execute an illegal opcode
XEXC_ID_DOPB_EXCEPTION	4	Exception due to a timeout on the DOPB bus
XEXC_ID_DIV_BY_ZERO	5	Divide by zero exceptions from the hardware divide
XEXC_ID_FPU	6	Exceptions from the floating point unit on MicroBlaze. ***Note:*** This exception is valid only on v4.00.a and newer versions of MicroBlaze.

By default, the BSP provides empty handlers for all the exceptions except the unaligned exceptions. A default, fast unaligned access exception handler is provided for by the BSP. An unaligned exception can be handled by making the corresponding aligned access on or to the appropriate bytes in memory. User software is not required to be aware of the unaligned access at all and it is invisibly handled by the default handler. However, software that involves a significant amount of unaligned accesses will see its effects at run-time. This is because the software exception handler takes longer to satisfy the unaligned access request as compared to a purely aligned one. Therefore, in some cases, user software might want to use the provision for unaligned exceptions, just to trap the exception and be aware of software causing the exception. In this case, you should specify your own exception handler for unaligned exceptions.

Note: Because of the way the first level handler stores volatile and temporary registers on the stack, by the time your custom unaligned access handler is invoked, critical information is not directly available to the handler. Therefore, it is not recommended to specify your own recovering (perform alignment and return) handler for unaligned exceptions. Rather, you should use a custom handler, just to trap the faulting access (for example, by setting a breakpoint or printing a message in your handler). Similarly, your custom handlers for the other exceptions must be aware of the fact that the first level exception handler would have saved some state on the stack, before invoking your handler.

Note: A Handler for unaligned exception (XEXC_ID_UNALIGNED_ACCESS) cannot be specified/changed dynamically at run-time by using the microblaze_register_exception_handler function if the software platform was built with the default handler for unaligned exceptions. This is an enforced limitation. The microblaze_register_exception_handler function silently performs a “nop” when invoked within XEXC_ID_UNALIGNED_ACCESS as the ExceptionId. If you do need to specify a custom handler for unaligned exceptions, you must set a handler value for the “microblaze_exception_handlers” parameter in the Standalone BSP software settings. Thus, when the software platform is built with a custom handler specified for unaligned exceptions, then the microblaze_register_exception_handler function is fully functional for XEXC_ID_UNALIGNED_ACCESS.

The other exceptions are very useful in conjunction with an OS of some kind. For example, on a divide by zero, the operating system (OS) might determine the process that caused the exception and then terminate it. The same applies for the other exceptions.

Nested exceptions are allowed by MicroBlaze. The exception handler, in its prologue, re-enables exceptions. Thus, exceptions within exception handlers are allowed and handled.

The parameter *predecode_fpu_exceptions* when set to true, causes the low-level exception handler to decode the faulting floating point instruction, figure out the operand registers and store their values into two global variables. You can register a handler for floating point exceptions and retrieve the values of the operands from the global variables. You can use the *microblaze_getfpex_operand_a()* and *microblaze_getfpex_operand_b()* macros.

Note: These macros return the operand values of the last floating point (FP) exception. If there are nested exceptions, you cannot retrieve the values of outer exceptions. An FP instruction might have one of the source registers being the same as the destination operand. In this case, the faulting instruction overwrites the input operand value and it is again irrecoverable.

Instruction Cache Handling

The following functions help manage instruction caches on MicroBlaze. You must include the header file `mb_interface.h` in your source code to use these functions.

Note: These functions work correctly only when the parameters that determine the caching system are configured appropriately in the MicroBlaze Microprocessor Hardware Specification (MHS) hardware block. Refer to the *MicroBlaze Reference Guide* for information on how to configure these cache parameters.

`void` **`microblaze_enable_icache`**`(void)`

This function enables the instruction cache on MicroBlaze. When MicroBlaze starts up, the instruction cache is disabled. The ICache must be explicitly turned on using this function.

`void` **`microblaze_disable_icache`**`(void)`

This function disables the instruction cache on MicroBlaze.

`void` **`microblaze_init_icache_range`** `(int` *`cache_addr`*`, int` *`cache_size`*`)`

The icache can be invalidating using the function `microblaze_init_icache_range`. This function can be used for invalidating the entire icache or only a part of it. The parameter `cache_addr` indicates the beginning of the cache location, which is to be invalidating. The `cache_size` represents the number of bytes from `cache_addr`, which needs to be invalidating.

For example, `microblaze_init_icache_range` (0x00000300, 0x100) invalidates the instruction cache region between 0x300 to 0x3ff (0x100 bytes of cache memory is cleared starting from 0x300).

Data Cache Handling

The following functions help manage data caches on MicroBlaze. You must include the header file `mb_interface.h` in your source code to use these functions.

Note: These functions work correctly only when the parameters that determine the caching system are configured appropriately in the MicroBlaze MHS hardware block. Refer to the *MicroBlaze Reference Guide* for information on how to configure these cache parameters.

`void` **`microblaze_enable_dcache`**`(void)`

This function enables the data cache on MicroBlaze. When MicroBlaze starts up, the data cache is disabled. The Dcache must be explicitly turned on using this function.

`void` **`microblaze_disable_dcache`**`(void)`

This function disables the data cache on MicroBlaze.

`void` **`microblaze_init_dcache_range`** `(int` *`cache_addr`*`, int` *`cache_size`*`)`

The dcache can be invalidated using the function `microblaze_init_dcache_range`. This function can be used for invalidating the entire dcache or only a part of it. The parameter `cache_addr` indicate the beginning of the cache location, which is to be invalidated. The `cache_size` represents the size from `cache_addr`, which needs to be invalidated. Both the `cache_addr` and the `cache_size` parameters are aligned to a cache line boundary before the invalidation starts. Interrupts are disabled while the cache is being invalidated.

For example, *`microblaze_init_dcache_range`* (0x00000300, 0x100) invalidates the data cache region between 0x300 to 0x3ff (0x100 bytes of cache memory is cleared starting from 0x300).

Software Initialization Sequence for Instruction and Data Caches

Typically, before using the cache, your program should do a particular sequence of cache operations, to ensure that invalid/dirty data in the cache is not being used by the processor. This would typically happen, during repeated downloads of program(s) and executing them.

The required sequence is the following:

1. The cache should be disabled.
2. The cache should be fully invalidated.
3. The cache should be enabled.

Here is an example snippet that assumes that both I and D caches are present and are 8 kB in size. The values should be the C_CACHE_BYTE_SIZE and C_DCACHE_BYTE_SIZE parameters on MicroBlaze.

```
/* Cache sizes
*
* ICACHE_SIZE should be set to C_CACHE_BYTE_SIZE
* DCACHE_SIZE should be set to C_DCACHE_BYTE_SIZE
*/

#define ICACHE_SIZE 8192
#define DCACHE_SIZE 8192

/* Initialize ICache *//
microblaze_disable_icache ();
microblaze_init_icache_range(0, ICACHE_SIZE);
microblaze_enable_icache ();

/* Initialize DCache */
microblaze_disable_dcache ();
microblaze_init_dcache_range(0, DCACHE_SIZE);
microblaze_enable_dcache ();
```

Fast Simplex Link Interface Macros

The BSP includes macros to provide convenient access to accelerators connected to the Fast Simple Link (FSL) Interfaces of MicroBlaze. The macros are listed below. In the macros, `val` refers to a variable in your program which can be the source or sink of the FSL operation. You must include `fsl.h` in your source files to make these macros available.

getfsl(val, id)

This macro performs a blocking data get function on an input FSL of MicroBlaze; `id` is the FSL identifier and can range from 0 to 7. This macro is uninterruptible.

putfsl(val, id)

This macro performs a blocking data put function on an output FSL of MicroBlaze; `id` is the FSL identifier and can range from 0 to 7. This macro is uninterruptible.

ngetfsl(val, id)

This macro performs a non-blocking data get function on an input FSL of MicroBlaze; `id` is the FSL identifier and can range from 0 to 7.

nputfsl(val, id)

This macro performs a non-blocking data put function on an output FSL of MicroBlaze; `id` is the FSL identifier and can range from 0 to 7.

`cgetfsl(val, id)`

This macro performs a blocking control get function on an input FSL of MicroBlaze; `id` is the FSL identifier and can range from 0 to 7. This macro is uninterruptible.

`cputfsl(val, id)`

This macro performs a blocking control put function on an output FSL of MicroBlaze; `id` is the FSL identifier and can range from 0 to 7. This macro is uninterruptible.

`ncgetfsl(val, id)`

This macro performs a non-blocking control get function on an input FSL of MicroBlaze; `id` is the FSL identifier and can range from 0 to 7.

`ncputfsl(val, id)`

This macro performs a non-blocking control put function on an output FSL of MicroBlaze; `id` is the FSL identifier and can range from 0 to 7.

`getfsl_interruptible(val, id)`

This macro performs repeated non-blocking data get operations on an input FSL of MicroBlaze until valid data is actually fetched; `id` is the FSL identifier and can range from 0 to 7. Since the FSL access is non-blocking, interrupts will be serviced by the processor.

`putfsl_interruptible(val, id)`

This macro performs repeated non-blocking data put operations on an output FSL of MicroBlaze until valid data is sent out; `id` is the FSL identifier and can range from 0 to 7. Since the FSL access is non-blocking, interrupts will be serviced by the processor.

`cgetfsl_interruptible(val, id)`

This macro performs repeated non-blocking control get operations on an input FSL of MicroBlaze until valid data is actually fetched; `id` is the FSL identifier and can range from 0 to 7. Since the FSL access is non-blocking, interrupts are serviced by the processor.

`cputfsl_interruptible(val, id)`

This macro performs repeated non-blocking control put operations on an output FSL of MicroBlaze until valid data is sent out; `id` is the FSL identifier and can range from 0 to 7. Since the FSL access is non-blocking, interrupts are serviced by the processor.

`fsl_isinvalid(invalid)`

This macro is used to check if the last FSL operation returned valid data or not. This macro is applicable after invoking a non-blocking FSL put or get instruction. If there was no data on the FSL channel on a get, or if the FSL channel was full on a put, then `invalid` is set to 1. Otherwise, `invalid` is set to 0.

`fsl_iserror(error)`

This macro is used to check if the last FSL operation set an error flag. This macro is applicable after invoking a control FSL put or get instruction. If the control bit was set `error` is set to 1. Otherwise, `error` is set to 0.

Deprecated FSL Macros

The following FSL macros are deprecated. They have been replaced by the ones described above.

`microblaze_nbread_cntlfsl(val, id)`

This macro performs a non-blocking control get function on an input FSL of MicroBlaze; `id` is the FSL identifier and can range from 0 to 7.

`microblaze_nbwrite_cntlfsl(val, id)`

This macro performs a non-blocking data control function on an output FSL of MicroBlaze; `id` is the FSL identifier and can range from 0 to 7.

`microblaze_bread_datafsl(val, id)`

This macro performs a blocking data get function on an input FSL of MicroBlaze; `id` is the FSL identifier and can range from 0 to 7.

`microblaze_bwrite_datafsl(val, id)`

This macro performs a blocking data put function on an output FSL of MicroBlaze; `id` is the FSL identifier and can range from 0 to 7.

`microblaze_nbread_datafsl(val, id)`

This macro performs a non-blocking data get function on an input FSL of MicroBlaze; `id` is the FSL identifier and can range from 0 to 7.

`microblaze_nbwrite_datafsl(val, id)`

This macro performs a non-blocking data put function on an output FSL of MicroBlaze; `id` is the FSL identifier and can range from 0 to 7.

`microblaze_bread_cntlfsl(val, id)`

This macro performs a blocking control get function on an input FSL of MicroBlaze; `id` is the FSL identifier and can range from 0 to 7.

`microblaze_bwrite_cntlfsl(val, id)`

This macro performs a blocking control put function on an output FSL of MicroBlaze; `id` is the FSL identifier and can range from 0 to 7.

`microblaze_nbread_cntlfsl(val, id)`

This macro performs a non-blocking control get function on an input FSL of MicroBlaze; `id` is the FSL identifier and can range from 0 to 7.

`microblaze_nbwrite_cntlfsl(val, id)`

This macro performs a non-blocking data control function on an output FSL of MicroBlaze; `id` is the FSL identifier and can range from 0 to 7.

Pseudo-asm Macros

The BSP includes macros to provide convenient access to various registers in MicroBlaze. Some of these macros are very useful within exception handlers for retrieving information about the exception. You must include the header file `mb_interface.h` in your source code to use these APIs.

`mfgpr(rn)`

Return value from the general purpose register (GPR) `rn`.

`mfmsr()`

Return the current value of the MSR.

`mfesr()`

Return the current value of the equivalent series resistance (ESR).

`mfear()`

Return the current value of the exception address register (EAR).

`mffsr()`

Return the current value of the feedback shift register (FSR).

`mtmsr(v)`

Move the value `v` to MSR.

`mtgpr(rn,v)`

Move the value `v` to GPR `rn`.

`microblaze_getfpex_operand_a()`

Return the saved value of operand A of the last faulting floating point instruction.

`microblaze_getfpex_operand_b()`

Return the saved value of operand B of the last faulting floating point instruction.

Note: Because of the way some of these macros have been written, they cannot be used as parameters to function calls and other such constructs.

Processor Version Register Access Routines

MicroBlaze v5.00.a and later versions have configurable Processor Version Registers (PVRs). The contents of the PVR are captured using the `pvr_t` data structure. It is defined to be an array of 32-bit words, with each word corresponding to a PVR register on hardware. The number of PVR words is determined by the number of PVRs configured in the hardware. You should not attempt to access PVR registers that are not present in hardware, as the `pvr_t` data structure is resized to hold only as many PVRs as are present in hardware. The following routines and pre-processor macros are used to access the PVR. You must include `pvr.h` to make these routines and macros available.

int microblaze_get_pvr(pvr_t *pvr)

Populate the PVR data structure pointed to by `pvr` with the values of the hardware PVR registers. This routine populates only as many PVRs as are present in hardware and the rest are zeroed. This routine is not available if C_PVR is set to NONE in hardware.

Table 3: **PVR Access Macros**

Macro	Description
MICROBLAZE_PVR_IS_FULL(pvr)	Return non-zero integer if PVR is of type FULL, 0 if basic.
MICROBLAZE_PVR_USE_BARREL(pvr)	Return non-zero integer if hardware barrel shifter present.
MICROBLAZE_PVR_USE_DIV(pvr)	Return non-zero integer if hardware divider present.
MICROBLAZE_PVR_USE_HW_MUL(pvr)	Return non-zero integer if hardware multiplier present.
MICROBLAZE_PVR_USE_FPU(pvr)	Return non-zero integer if hardware floating point unit (FPU) present.
MICROBLAZE_PVR_USE_ICACHE(pvr)	Return non-zero integer if I-cache present.
MICROBLAZE_PVR_USE_DCACHE(pvr)	Return non-zero integer if D-cache present
MICROBLAZE_PVR_MICROBLAZE_VERSION (pvr)	Return MicroBlaze version encoding. Refer to the *MicroBlaze Hardware Reference Manual* for mappings from encodings to actual hardware versions.
MICROBLAZE_PVR_USER1(pvr)	Return the USER1 field stored in the PVR.
MICROBLAZE_PVR_USER2(pvr)	Return the USER2 field stored in the PVR.
MICROBLAZE_PVR_D_OPB(pvr)	Return non-zero integer if Data side on-chip peripheral bus (OPB) interface present.
MICROBLAZE_PVR_DLMB(pvr)	Return non-zero integer if Data side local memory bus (LMB) interface present.
MICROBLAZE_PVR_I_OPB(pvr)	Return non-zero integer if Instruction side OPB interface present.
MICROBLAZE_PVR_I_LMB(pvr)	Return non-zero integer if Instruction side LMB interface present.
MICROBLAZE_PVR_INTERRUPT_IS_EDGE(pvr)	Return non-zero integer if interrupts are configured as edge-triggered.
MICROBLAZE_PVR_EDGE_IS_POSITIVE(pvr)	Return non-zero integer if interrupts are configured as positive edge triggered.
MICROBLAZE_PVR_USE_MUL64(pvr)	Return non-zero integer if MicroBlaze supports 64-bit products for multiplies.
MICROBLAZE_PVR_OPCODE_OxO_ILLEGAL(pvr)	Return non-zero integer if opcode 0x0 is treated as an illegal opcode.
MICROBLAZE_PVR_UNALIGNED_EXCEPTION(pvr)	Return non-zero integer if unaligned exceptions are supported.

Table 3: **PVR Access Macros**

Macro	Description
MICROBLAZE_PVR_ILL_OPCODE_EXCEPTION(pvr)	Return non-zero integer if illegal opcode exceptions are supported.
MICROBLAZE_PVR_IOPB_EXCEPTION(pvr)	Return non-zero integer if I-OPB exceptions are supported.
MICROBLAZE_PVR_DOPB_EXCEPTION(pvr)	Return non-zero integer if D-OPB exceptions are supported.
MICROBLAZE_PVR_DIV_ZERO_EXCEPTION(pvr)	Return non-zero integer if divide by zero exceptions are supported
MICROBLAZE_PVR_FPU_EXCEPTION(pvr)	Return non-zero integer if FPU exceptions are supported.
MICROBLAZE_PVR_DEBUG_ENABLED(pvr)	Return non-zero integer if debug is enabled.
MICROBLAZE_PVR_NUM_PC_BRK(pvr)	Return the number of hardware PC breakpoints available.
MICROBLAZE_PVR_NUM_RD_ADDR_BRK(pvr)	Return the number of read address hardware watchpoints supported.
MICROBLAZE_PVR_NUM_WR_ADDR_BRK(pvr)	Return the number of write address hardware watchpoints supported.
MICROBLAZE_PVR_FSL_LINKS(pvr)	Return the number of FSL links present.
MICROBLAZE_PVR_ICACHE_BASEADDR(pvr)	Return the base address of the I-cache.
MICROBLAZE_PVR_ICACHE_HIGHADDR(pvr)	Return the high address of the I-cache.
MICROBLAZE_PVR_ICACHE_ADDR_TAG_BITS(pvr)	Return the number of address tag bits for the I-cache.
MICROBLAZE_PVR_ICACHE_USE_FSL(pvr)	Return non-zero if I-cache uses FSL links.
MICROBLAZE_PVR_ICACHE_ALLOW_WR(pvr)	Return non-zero if writes to I-caches are allowed.
MICROBLAZE_PVR_ICACHE_LINE_LEN(pvr)	Return the length of each I-cache line in bytes.
MICROBLAZE_PVR_ICACHE_BYTE_SIZE(pvr)	Return the size of the D-cache in bytes.
MICROBLAZE_PVR_DCACHE_BASEADDR(pvr)	Return the base address of the D-cache.
MICROBLAZE_PVR_DCACHE_HIGHADDR(pvr)	Return the high address of the D-cache.
MICROBLAZE_PVR_DCACHE_ADDR_TAG_BITS(pvr)	Return the number of address tag bits for the D-cache.
MICROBLAZE_PVR_DCACHE_USE_FSL(pvr)	Return non-zero if the D-cache uses FSL links.

Table 3: **PVR Access Macros**

Macro	Description
MICROBLAZE_PVR_DCACHE_ALLOW_WR(pvr)	Return non-zero if writes to D-cache are allowed.
MICROBLAZE_PVR_DCACHE_LINE_LEN(pvr)	Return the length of each line in the D-cache in bytes.
MICROBLAZE_PVR_DCACHE_BYTE_SIZE(pvr)	Return the size of the D-cache in bytes.
MICROBLAZE_PVR_TARGET_FAMILY	Return the encoded target family identifier.
MICROBLAZE_PVR_MSR_RESET_VALUE	Refer to the *MicroBlaze Hardware Reference Manual* for mappings from encodings to target family name strings.

Accessing the information in the PVRs is done through a two-step process. In the first step, you must use the `microblaze_get_pvr ()` function to populate the PVR data into a `pvr_t` data structure. In subsequent steps, you may use anyone of the PVR access macros listed in to get individual data stored in the PVR.

Note: The PVR access macros take a parameter, which must be of type `pvr_t`.

File Handling

`int fcntl(int fd, int cmd, long arg);`

A dummy implementation of `fcntl`, which always returns 0, is provided. `fcntl` is intended to manipulate file descriptors according to the command specified by `cmd`. Since the standalone BSP does not provide a file system, this function does not do anything.

Errno

`int errno();`

Return the global value of errno as set by the last C library call.

PowerPC BSP

When the user system contains a PowerPC™ processor, and no Operating System, the Library Generator automatically builds the BSP in the project library libxil.a.

The BSP contains boot code, cache, file and memory management, configuration, exception handling, time and processor specific include functions.

Function Summary

The following table contains a list of all PowerPC BSP functions.

Table 4: **Function Summary**

Functions
`void XCache_WriteCCR0(unsigned int val);`
`void XCache_EnableDCache(unsigned int regions);`
`void XCache_DisableDCache(void);`
`void XCache_FlushDCacheLine(unsigned int adr);`
`void XCache_InvalidateDCacheLine(unsigned int adr);`
`void XCache_StoreDCacheLine(unsigned int adr);`
`void XCache_EnableICache(unsigned int regions);`
`void XCache_DisableICache(void);`
`void XCache_InvalidateICache(void);`
`void XCache_InvalidateICacheLine(unsigned int adr);`
`void XExc_Init(void);`
`void XExc_RegisterHandler(Xuint8 ExceptionId, XExceptionHandler Handler, void *DataPtr);`
`void XExc_RemoveHandler(Xuint8 ExceptionId)`
`void XExc_mEnableExceptions (EnableMask)`
`void XExc_mDisableExceptions (DisableMask);`
`int read(int fd, char *buf, int nbytes);`
`int write(int fd, char *buf, int nbytes);`
`int isatty(int fd);`
`int fcntl(int fd, int cmd, long arg);`
`int errno();`
`char *sbrk(int nbytes);`
`void XTime_SetTime(XTime xtime);`
`void XTime_GetTime(XTime *xtime);`
`void XTime_TSRClearStatusBits(unsigned long Bitmask);`
`void XTime_PITSetInterval(unsigned long interval);`
`void XTime_PITEnableInterrupt(void);`
`void XTime_PITDisableInterrupt(void);`
`void XTime_PITEnableAutoReload(void);`
`void XTime_PITDisableAutoReload(void);`
`void XTime_PITClearInterrupt(void);`
`unsigned int usleep(unsigned int __useconds);`
`unsigned int sleep(unsigned int __seconds);`
`int nanosleep(const struct timespec *rqtp, struct timespec *rmtp);`

Boot Code

The `boot.S` file contains a minimal set of code for transferring control from the processor's reset location to the start of the application. Code in the `boot.S` consists of the two sections **`boot`** and **`boot0`**. The boot section contains only one instruction that is labeled with **`_boot`**. During the link process, this instruction is mapped to the reset vector and the **`_boot`** label marks the application's entry point. The boot instruction is a jump to the **`_boot0`** label. The **`_boot0`** label must reside within a ±23-bit address space of the **`_boot`** label. It is defined in the **`boot0`** section. The code in the **`boot0`** section calculates the 32-bit address of the **`_start`** label and jumps to it.

Cache

The `xcache_l.c` file and corresponding `xcache_l.h` include file provide access to cache and cache-related operations.

void **XCache_WriteCCR0**(unsigned int *val*);

The XCache_WriteCCR0( ) function writes an integer value to the CCR0 register. Below is a sample code sequence. Before writing to this register, the instruction cache must be enabled to prevent a lockup of the processor core. After writing the CCR0, the instruction cache can be disabled, if not needed.

```
...
XCache_EnableICache(0x80000000) /* enable instruction cache for first 128
MB memory region */
XCache_WriteCCR0(0x2700E00) /* enable 8 word pre-fetching */
XCache_DisableICache() /* disable instruction cache */
...
```

void **XCache_EnableDCache**(unsigned int *regions*);

The XCache_EnableDCache( ) function enables the data cache for a specific memory region. Each bit in the *regions* parameter represents 128 MB of memory.

A value of `0x80000000` enables the data cache for the first 128 MB of memory (`0 - 0x7FFFFFF`). A value of `0x1` enables the data cache for the last 128 MB of memory (`0xF8000000 - 0xFFFFFFFF`).

void **XCache_DisableDCache**(void);

The XCache_DisableDCache( ) function disables the data cache for all memory regions.

void **XCache_FlushDCacheLine**(unsigned int *adr*);

The XCache_FlushDCacheLine( ) function flushes and invalidates the data cache line that contains the address specified by the *adr* parameter. A subsequent data access to this address results in a cache miss and a cache line refill.

`void XCache_InvalidateDCacheLine(unsigned int adr);`

The XCache_InvalidateDCacheLine( ) function invalidates the data cache line that contains the address specified by the *adr* parameter. If the cache line is currently dirty, the modified contents are lost and are **not** written to system memory. A subsequent data access to this address results in a cache miss and a cache line refill.

`void XCache_StoreDCacheLine(unsigned int adr);`

The XCache_StoreDCacheLine( ) function stores in memory the data cache line that contains the address specified by the *adr* parameter. A subsequent data access to this address results in a cache hit if the address was already cached; otherwise, it results in a cache miss and cache line refill.

`void XCache_EnableICache(unsigned int regions);`

The XCache_EnableICache( ) function enables the instruction cache for a specific memory region. Each bit in the *regions* parameter represents 128 MB of memory.

A value of `0x80000000` enables the instruction cache for the first 128 MB of memory (`0 - 0x7FFFFFF`). A value of `0x1` enables the instruction cache for the last 128 MB of memory (`0xF8000000 - 0xFFFFFFFF`).

`void XCache_DisableICache(void);`

The XCache_DisableICache( ) function disables the instruction cache for all memory regions.

`void XCache_InvalidateICache(void);`

The XCache_InvalidateICache( ) function invalidates the whole instruction cache. Subsequent instructions produce cache misses and cache line refills.

`void XCache_InvalidateICacheLine(unsigned int adr);`

The XCache_InvalidateICacheLine( ) function invalidates the instruction cache line that contains the address specified by the *adr* parameter. A subsequent instruction to this address produces a cache miss and a cache line refill.

Exception Handling

This section documents the exception handling API that is provided in the Board Support Package. For an in-depth explanation on how exceptions and interrupts work on the PowerPC405, refer to the chapter “Exceptions and Interrupts” in the *PowerPC Processor Reference Guide*.

Note: Exception handlers do not automatically reset (disable) the wait state enable bit in the MSR when returning to user code. You can force exception handlers to reset the Wait-Enable bit to zero on return from all exceptions by compiling the BSP with the preprocessor symbol `PPC405_RESET_WE_ON_RFI` defined. You can add this to the compiler flags associated with the libraries. This pre-processor define turns the behavior on.

The exception handling API consists of a set of the files `xvectors.S`, `xexception_l.c`, and the corresponding header file `xexception_l.h`.

For additional information on interrupt handing see the *XPS Help* and the "Interrupt Management" appendix in the *Embedded System Tools Reference Manual* (available in the `/doc` directory of your EDK installation).

void **XExc_Init**(void);

This function sets up the interrupt vector table and registers a “do nothing” function for each exception. This function has no parameters and does not return a value.

This function must be called before registering any exception handlers or enabling any interrupts. When using the exception handler API, this function should be called at the beginning of your main( ) routine.

IMPORTANT: If you are not using the default linker script, you need to reserve memory space for storing the vector table in your linker script. The memory space must begin on a 64k boundary. The linker script entry should look like this example:

```
.vectors :
  {
    . = ALIGN(64k);
    *(.vectors)
  }
```

For further information on linker scripts, refer to the Linker documentation.

`void XExc_RegisterHandler(Xuint8 ExceptionId, XExceptionHandler Handler, void *DataPtr);`

This function is used to register an exception handler for a specific exception. It does not return a value. Refer to the table below for a list of parameters.

Table 5: **Exception Handler Parameters**

Parameter Name	Parameter Type	Description
ExceptionId	Xuint8	Exception to which this handler should be registered. The type and the values are defined in the header file `xexception_l.h`. Refer to the table below for possible values.
Handler	XExceptionHandler	Pointer to the exception handling function.
DataPtr	void *	User value to be passed when the handling function is called.

Table 6: **Registered Exception Types and Values**

Exception Type	Value
XEXC_ID_JUMP_TO_ZERO	0
XEXC_ID_MACHINE_CHECK	1
XEXC_ID_CRITICAL_INT	2
XEXC_ID_DATA_STORAGE_INT	3
XEXC_ID_INSTUCTION_STORAGE_INT	4
XEXC_ID_NON_CRITICAL_INT	5
XEXC_ID_ALIGNMENT_INT	6
XEXC_ID_PROGRAM_INT	7
XEXC_ID_FPU_UNAVAILABLE_INT	8
XEXC_ID_SYSTEM_CALL	9
XEXC_ID_APU_AVAILABLE	10
XEXC_ID_PIT_INT	11
XEXC_ID_FIT_INT	12
XEXC_ID_WATCHDOG_TIMER_INT	13
XEXC_ID_DATA_TLB_MISS_INT	14
XEXC_ID_INSTRUCTION_TLB_MISS_INT	15
XEXC_ID_DEBUG_INT	16

The function provided as the *Handler* parameter must have the following function prototype:

```
typedef void (*XExceptionHandler)(void * DataPtr);
```

This prototype is declared in the `xexception_l.h` header file.

When this exception handler function is called, the parameter *DataPtr* contains the same value as you provided when you registered the handler.

`void` **`XExc_RemoveHandler`**`(Xuint8` *`ExceptionId`*`)`

This function is used to deregister a handler function for a given exception. For possible values of parameter *ExceptionId*, refer to Table 6.

`void` **`XExc_mEnableExceptions`** `(`*`EnableMask`*`)`

This macro is used to enable exceptions. It must be called after initializing the vector table with function exception_Init and registering exception handlers with function XExc_RegisterHandler. The parameter *EnableMask* is a bitmask for exceptions to be enabled. The *EnableMask* parameter can have the values XEXC_CRITICAL, XEXC_NON_CRITICAL or XEXC_ALL.

`void` **`XExc_mDisableExceptions`** `(`*`DisableMask`*`);`

This macro is called to disable exceptions. The parameter *DisableMask* is a bitmask for exceptions to be disabled.The *DisableMask* parameter can have the values XEXC_CRITICAL, XEXC_NON_CRITICAL or XEXC_ALL.

Files

File support is limited to the **stdin** and **stdout** streams. In such an environment, the following functions do not make much sense:

- open( ) (in `open.c`)
- close( ) (in `close.c`)
- fstat( ) (in `fstat.c`)
- unlink( ) (in `unlink.c`)
- lseek( ) (in `lseek.c`)

These files are included for completeness and because they are referenced by the C library.

`int` **`read`**`(int` *`fd, char *buf, int nbytes`*`);`

The read( ) function in `read.c` reads *nbytes* bytes from the standard input by calling inbyte( ). It blocks until all characters are available, or the end of line character is read. Read( ) returns the number of characters read. The parameter *fd* is ignored.

`int **write**(int *fd*, char **buf*, int *nbytes*);`

The write( ) function in `write.c` writes *nbytes* bytes to the standard output by calling outbyte( ). It blocks until all characters have been written. Write( ) returns the number of characters written. The parameter *fd* is ignored.

`int **isatty**(int *fd*);`

The isatty( ) function in `isatty.c` reports if a file is connected to a tty. This function always returns 1, since only the `stdin` and `stdout` streams are supported.

`int **fcntl**(int *fd*, int *cmd*, long *arg*);`

A dummy implementation of `fcntl`, which always returns 0, is provided. `fcntl` is intended to manipulate file descriptors according to the command specified by `cmd`. Since the standalone BSP does not provide a file system, this function does not do anything.

Errno

`int **errno**();`

Return the global value of errno as set by the last C library call.

Memory Management

`char ***sbrk**(int *nbytes*);`

The sbrk( ) function in the `sbrk.c` file allocates `nbytes` of heap and returns a pointer to that piece of memory. This function is called from the memory allocation functions of the C library.

Process

The functions getpid() in `getpid.c` and kill() in `kill.c` are included for completeness and because they are referenced by the C library.

Processor-Specific Include Files

The `xreg405.h` include file contains the register numbers and the register bits for the PPC 405 processor.

The `xpseudo-asm.h` include file contains the definitions for the most often used inline assembler instructions, available as macros. These can be very useful for doing things such as setting or getting special purpose registers, synchronization, or cache manipulation.

These inline assembler instructions can be used from drivers and user applications written in C.

Time

The `xtime_l.c` file and corresponding `xtime_l.h` include file provide access to the 64-bit time base counter inside the PowerPC core. The counter increases by one at every processor cycle.

The `sleep.c` file and corresponding `sleep.h` include file implement functions for tired programs. All sleep functions are implemented as busy loops.

typedef unsigned long long XTime;

The `XTime` type in `xtime_l.h` represents the Time Base register. This struct consists of the Time Base Low (TBL) and Time Base High (TBH) registers, each of which is a 32-bit wide register. The definition of `XTime` is as follows:

```
typedef unsigned long long XTime;
```

---

void **XTime_SetTime**(XTime *xtime*);

The XTime_SetTime( ) function in `xtime_l.c` sets the time base register to the value in *xtime*.

---

void **XTime_GetTime**(XTime **xtime*);

The XTime_GetTime( ) function in `xtime_l.c` writes the current value of the time base register to variable *xtime*.

---

void **XTime_TSRClearStatusBits**(unsigned long *Bitmask*);

The XTime_TSRClearStatusBits() function in `xtime_l.c` is used to clear bits in the Timer Status Register (TSR). The parameter *Bitmask* designates the bits to be cleared. A value of 1 in any position of the Bitmask parameter clears the corresponding bit in the TSR. This function does not return a value. The header file `xreg405.h` defines the following values for the *Bitmask* parameter.

Table 7: **Bitmask Parameter Values**

Name	Value	Description
XREG_TSR_WDT_ENABLE_NEXT_WATCHDOG	`0x80000000`	Clearing this bit disables the watchdog timer event.
XREG_TSR_WDT_INTERRUPT_STATUS	`0x40000000`	Clears the Watchdog Timer Interrupt Status bit. This bit is set after a watchdog interrupt occurred, or could have occurred had it been enabled.
XREG_TSR_WDT_RESET_STATUS_11	`0x30000000`	Clears the Watchdog Timer Reset Status bits. These bits Specify the kind of reset that occurred as a result of a watchdog timer event.
XREG_TSR_PIT_INTERRUPT_STATUS	`0x08000000`	Clears the Programmable Interval Timer (PIT) Status bit. This bit is set after a PIT interrupt has occurred.
XREG_TSR_FIT_INTERRUPT_STATUS	`0x04000000`	Clears the Fixed Interval Timer Status (FIT) bit. This bit is set after a FIT interrupt has occurred.
XREG_TSR_CLEAR_ALL	`0xFFFFFFFF`	Clears all bits in the TSR. After a Reset, the content of the TSR is not specified. Use this Bitmask to clear all bits in the TSR.

Example:

XTime_TSRClearStatusBits(TSR_CLEAR_ALL);

`void` **`XTime_PITSetInterval`** `(unsigned long` *`interval`*`);`

The XTime_PITSetInterval( ) function in `xtime_l.c` is used to load a new value into the Programmable-Interval Timer Register. This register is a 32-bit decrementing counter clocked at the same frequency as the time-base register. Depending on the AutoReload setting the PIT is automatically reloaded with the last written value or has to be reloaded manually. This function does not return a value.

Example:

XTime_PITSetInterval(0x00ffffff);

`void` **`XTime_PITEnableInterrupt`** `(void);`

The XTime_PITEnableInterrupt() function in `xtime_l.c` enables the generation of PIT interrupts. An interrupt occurs when the PIT register contains a value of 1, and is then decremented. This function does not return a value. XExc_Init() must be called, the PIT interrupt handler must be registered, and exceptions must be enabled before calling this function.

Example:

XTime_PITEnableInterrupt();

void **XTime_PITDisableInterrupt**(void);

The XTime_PITDisableInterrupt() function in `xtime_l.c` disables the generation of PIT interrupts. It does not return a value.

Example:

XTime_PITDisableInterrupt();

void **XTime_PITEnableAutoReload**(void);

The XTime_PITEnableAutoReload( ) function in `xtime_l.c` enables the auto-reload function of the PIT Register. When auto-reload is enabled the PIT Register is automatically reloaded with the last value loaded by calling the `XTime_PITSetInterval` function when the PIT Register contains a value of 1 and is decremented. When auto-reload is enabled, the PIT Register never contains a value of 0. This function does not return a value.

Example:

```
XTime_PITEnableAutoReload();
```

void **XTime_PITDisableAutoReload**(void);

The XTime_PITDisableAutoReload() function in `xtime_l.c` disables the auto-reload feature of the PIT Register. When auto-reload is disabled the PIT decrements from 1 to 0. If it contains a value of 0 it stops decrementing until it is loaded with a non-zero value. This function does not return a value.

Example:

```
XTime_PITDisableAutoReload();
```

void **XTime_PITClearInterrupt**(void);

The `XTime_PITClearInterrupt()` function in `xtime_l.c` is used to clear PIT-Interrupt-Status bit in the Timer-Status Register. This bit specifies whether a PIT interrupt occurred. You must call this function in your interrupt-handler to clear the Status bit, otherwise another PIT interrupt occurs immediately after exiting the interrupt handler function. This function does not return a value. Calling this function is equivalent to calling
`XTime_TSRClearStatusBits(XREG_TSR_PIT_INTERRUPT_STATUS.`

Example:

```
XTime_PITClearInterrupt();
```

unsigned int **usleep**(unsigned int *__useconds*);

The usleep() function in sleep.c delays the execution of a program by *__useconds* microseconds. It always returns zero. This function requires that the processor frequency (in Hz) is defined. The default value of this variable is 400MHz. This value can be overwritten in the MSS file as follows:

```
BEGIN PROCESSOR
PARAMETER HW_INSTANCE = PPC405_i
PARAMETER DRIVER_NAME = cpu_ppc405
PARAMETER DRIVER_VER = 1.00.a
PARAMETER CORE_CLOCK_FREQ_HZ = 20000000
END
```

The file xparameters.h can also be modified with the correct value, as follows:

```
#define XPAR_CPU_PPC405_CORE_CLOCK_FREQ_HZ 20000000
```

unsigned int **sleep**(unsigned int *__seconds*);

The sleep() function in sleep.c delays the execution of a program by *__seconds* seconds. It always returns zero.This function requires that the processor frequency (in Hz) is defined. The default value of this variable is 400MHz. This value can be overwritten in the Microprocessor Software Specification (MSS) file as follows:

```
BEGIN PROCESSOR
PARAMETER HW_INSTANCE = PPC405_i
PARAMETER DRIVER_NAME = cpu_ppc405
PARAMETER DRIVER_VER = 1.00.a
PARAMETER CORE_CLOCK_FREQ_HZ = 20000000
END
```

The file xparameters.h can also be modified with the correct value, as follows:

```
#define XPAR_CPU_PPC405_CORE_CLOCK_FREQ_HZ 20000000
```

int **nanosleep**(const struct timespec **rqtp*, struct timespec **rmtp*);

The nanosleep() function in sleep.c is currently not implemented. It is a placeholder for linking applications against the C library. It always returns zero.

Fast Simplex Link Interface Macros

The BSP includes macros to provide convenient access to accelerators connected to the PowerPC Auxiliary Processing Unit over the FSL Interfaces. The macros are listed below. In the macros, val refers to a variable in your program which can be the source or sink of the FSL operation. You must include the header file fsl.h in your source files to make these macros available.

getfsl(val, id)

This macro performs a blocking data get function on an input FSL interface; id is the FSL identifier and can range from 0 to 31. This macro is interruptible.

`putfsl(val, id)`

This macro performs a blocking data put function on an output FSL interface; `id` is the FSL identifier and can range from 0 to 31. This macro is interruptible.

`ngetfsl(val, id)`

This macro performs a non-blocking data get function on an input FSL interface; `id` is the FSL identifier and can range from 0 to 31.

`nputfsl(val, id)`

This macro performs a non-blocking data put function on an output FSL interface; `id` is the FSL identifier and can range from 0 to 31.

`cgetfsl(val, id)`

This macro performs a blocking control get function on an input FSL interface; `id` is the FSL identifier and can range from 0 to 31. This macro is interruptible.

`cputfsl(val, id)`

This macro performs a blocking control put function on an output FSL interface; `id` is the FSL identifier and can range from 0 to 31. This macro is interruptible.

`ncgetfsl(val, id)`

This macro performs a non-blocking control get function on an input FSL interface; `id` is the FSL identifier and can range from 0 to 31.

`ncputfsl(val, id)`

This macro performs a non-blocking data control function on an output FSL interface; `id` is the FSL identifier and can range from 0 to 31.

`getfsl_interruptible(val, id)`

This macro is aliased to *`getfsl(val,id)`*.

`putfsl_interruptible(val, id)`

This macro is aliased to *`putfsl(val,id)`*.

`cgetfsl_interruptible(val, id)`

This macro is aliased to *`cgetfsl(val,id)`*.

`cputfsl_interruptible(val, id)`

This macro is aliased to *`cputfsl(val,id)`*.

`fsl_isinvalid(invalid)`

This macro is used to check the if the last FSL operation returned valid data or not. This macro is applicable after invoking a non-blocking FSL put or get instruction. If there was no data on the FSL channel on a get, or if the FSL channel was full on a put, then `invalid` is set to 1. Else, `invalid` is set to 0.

`fsl_iserror(error)`

This macro is used to check the if the last FSL operation set an error flag. This macro is applicable after invoking a control FSL put or get instruction. If the control bit was set `error` is set to 1. Else, `error` is set to 0.

Pseudo-asm Macros

The BSP includes macros to provide convenient access to various registers on the PPC405. You must include the header file `xpseudo_asm.h` in your source code to use these APIs.

`mfgpr(rn)`

Return value from GPR `rn`.

`mfspr(rn)`

Return the current value of the special purpose register (SPR) `rn`.

`mfmsr()`

Return value from MSR.

`mfdcr(rn)`

Return value from the device control register (DCR) `rn`.

`mtdcr(rn,v)`

Move the value `v` to DCR `rn`.

`mtevpr(addr)`

Move the value `addr` to the exception vector prefix register (EVPR).

`mtspr(rn,v)`

Move the value `v` to SPR `rn`.

`mtgpr(rn,v)`

Move the value `v` to GPR `rn`.

`iccci`

Invalidate the instruction cache congruence class (entire cache).

`icbi(adr)`

Invalidate the instruction cache block at effective address `adr`.

`icbt(adr)`

Touch the instruction cache block at effective address `adr`.

`isync`

Execute the `isync` instruction.

`dccci(adr)`

Invalidate the data cache congruence class represented by effective address `adr`.

`dcbi(adr)`

Invalidate the data cache block at effective address `adr`.

`dcbst(adr)`

Store the data cache block at effective address `adr`.

`dcbf(adr)`

Flush the data cache block at effective address `adr`.

`dcread(adr)`

Read from data cache address `adr`.

`eieio`

Execute the `eieio` instruction.

`sync`

Execute the `sync` instruction.

`lbz(adr)`

Execute a load and return the byte value from address `adr`.

`lhz(adr)`

Execute a load and return the word half-word value from address `adr`.

`lwz(adr)`

Execute a load and return the word value from address `adr`.

`stb(adr,val)`

Store the byte value in `val` into address `adr`.

`sth(adr,val)`

Store the half-word value in `val` into address `adr`.

`stw(adr,val)`

Store the word value in `val` into address `adr`.

`lhbrx(adr)`

Execute a Load Halfword Byte-Reversed Indexed instruction on effective address `adr` and return the value.

`lwbrx(adr)`

Execute a Load Word Byte-Reversed Indexed instruction on effective address `adr and return the value`.

`sthbrx(adr,val)`

Execute a Store Halfword Byte-Reversed Indexed instruction on effective address `adr`, on value `val`.

`stwbrx(adr,val)`

Execute a Store Word Byte-Reversed Indexed instruction on effective address `adr`, on value `val`.

Macros for APU FCM User-Defined Instructions

Macros are provided for using the user-defined instructions supported by the PowerPC APU Fabric Coprocessor Module (FCM). There are a total of 16 user-defined instruction mnemonics provided — 8 for instructions that modify the Condition Register (CR) and 8 for the instructions that do not modify the CR. Since the meaning of the operands that these instructions take can be dynamically redefined, macros are provided for all combinations of operands. The user program must use the macros appropriately, in conjunction with higher level program flow.

`UDI<n>FCM(a, b, c, fmt)`

This macro inserts the mnemonic for user defined fcm instruction `n` (that does not modify CR) into the user's program. The user defined instruction, has `a`, `b`, `c` as operands to it in that order. The way the operands are interpreted by the compiler, is determined by the format specifier given by `fmt`. The format specifier is explained further below. `n` can range from 0 to 7. The mnemonic inserted is, **`udi<n>fcm`**.

`UDI<n>FCMCR(a, b, c, fmt)`

This macro inserts the mnemonic for user defined fcm instruction (that modifies CR) `n` into the user's program. The user defined instruction, has `a`, `b`, `c` as operands to it in that order. The way the operands are interpreted by the compiler, is determined by the format specifier given by `fmt`. The format specifier is explained further below. `n` can range from 0 to 7. The mnemonic inserted is, **`udi<n>fcm.`** (note the period at the end).

The format specifier can have the values described in the table below.

Table 8: **Format Specifier for UDI Instructions**

Identifier	Meaning
FMT_GPR_GPR_GPR	Operands `a`, `b` and `c` are general purpose registers.
FMT_GPR_GPR_IMM	Operands `a` and b are general purpose registers, while operand `c` is an immediate value representing an immediate constant or an FCM register.
FMT_GPR_IMM_IMM	Operand `a` is a general purpose registers, while operands `b` and **`c`** are immediate values representing an immediate constant or an FCM register.
FMT_IMM_GPR_GPR	Operands `b` and `c` are general purpose registers, while operand `a` is an immediate value representing an immediate constant or an FCM register.
FMT_IMM_IMM_GPR	Operand `c` is a general purpose registers, while operands `a` and `b` are immediate values representing an immediate constant or an FCM register.
FMT_IMM_IMM_IMM	All three operands are immediate values, representing an immediate constant or an FCM register.

Program Profiling

The Standalone BSP supports program profiling in conjunction with the GNU compiler tools and the Xilinx Microprocessor Debugger (XMD). Profiling a program running on hardware (board), provides insight into program execution and identifies where it spends most time. The interaction of the program with memory and other peripherals can be more accurately captured.

Program running on hardware target is profiled using *software intrusive* method. In this method, the profiling software code is instrumented in the user program. The profiling software code is a part of the **libxil.a** library and is generated when software intrusive profiling is enabled in Standalone-BSP. For more details on the Profiling flow, refer to the "Profiling Embedded Designs" section of the *XPS Help.*

When the profile option "**-pg**" is specified to the compiler (mb-gcc and powerpc-eabi-gcc), the profiling functions are automatically linked with the application to profile. The generated executable file contains code to generate profile information.

On program execution, this instrumented profiling function stores information on the hardware. Xilinx® Microprocessor Debugger (XMD) collects the profile information and generates the output file, which can be read by the GNU *gprof* tool. The program functionality remains unchanged but it slows down the execution.

Note: The profiling functions do not have any explicit application API. The library is linked due to profile calls (_mcount) introduced by gcc for profiling.

Profiling Requirements

- Software intrusive profiling requires memory for storing profile information. You can use any memory in the system for profiling.
- A timer is required for sampling instruction address. *opb_timer* is the supported profile timer. For PowerPC™ systems, *Programmable Interrupt Timer (PIT)* can also be used as profile timer.

Profiling Function Source Code Description

Some of the important functions are:

- **_profile_init** — This function is called before function *main* of an application. This function initializes the profile timer routine, registers timer handler accordingly based on timer used and connection to processor and starts the timer. The TCL routine of standalone library figures the timer type and connection to processor and generates the *#defines* in `profile_config.h` file. Refer to the "Microprocessor Library Definition (MLD)" chapter in the *Embedded System Tools Reference Manual*.
- **mcount** — This function is called by *_mcount* function, which is inserted at every function start by gcc. This function records the *caller* and *callee* information (Instruction address), which is used to generate call graph information.
- **profile_intr_handler** — This is the interrupt handler for the profiling timer. The timer is set to sample the executing application for PC values at fixed intervals and increment the Bin count. This is used to generate the histogram information.

Configuring the Standalone BSP

The Standalone-BSP is configured through XPS. From the tree view of XPS, right-click on any peripheral and select **Software Platform Settings**. This launches the Software Platform Settings dialog box. The bottom half of the Software Platform panel displays the Kernel and Operating Systems available in EDK. Select **standalone** as the OS. Click on the Library/OS Parameters tab to configure. The table below lists the configurable parameters for standalone BSP.

Table 9: **Configuration Parameters**

Parameter	Type	Default Value	Description
enable_sw_intrusive_profiling	Bool	false	Enable software intrusive profiling functionality. Select **true** to enable.
profile_timer	Peripheral Instance	none	Specify the timer to use for profiling. Select a opb_timer from the list of instance displayed. For a PowerPC system select "**none**" to use in-built PIT timer.
stdin	Peripheral Instance	none	Specify the STDIN peripheral from the drop down list
stdout	Peripheral Instance	none	Specify the STDOUT peripheral from the drop down list

Table 9: **Configuration Parameters *(Continued)***

Parameter	Type	Default Value	Description
microblaze_exception_vectors	Array	For the handler parameter, a *'default'* built-in handler is assigned for unaligned exceptions and XNullHandler is assigned for the other exceptions. The callback parameter is assigned the corresponding exception ID values, by default.	This parameter is valid only for MicroBlaze. Specify the exception handling routines for each hardware exception that is available on MicroBlaze.
predecode_fpu_exception	Bool	false	This parameter is valid only for MicroBlaze when FPU exceptions are enabled in the hardware. Setting this to true will include extra code to be included, that decodes and stores the faulting FP instruction's operands in global variables.