Interrupt Manager Documentation

Garrett Lail

April 2023

1 The Hardware Abstraction Layer

The interrupt manager fits into the larger Hardware Abstraction Layer (HAL) designed for AFTx06 and AFTx07. This documentation pertains only to the interrupt manager.

The HAL is meant to simplify programming by "wrapping" peripheral devices such as GPIO, SPI, PWM, and the interrupt manager in accessible and conventional software interfaces. Standard "bare-metal" programming requires assembly or embedded-C manipulation of hardware registers to set-up and control peripheral devices. This process itself requires explicit understanding of the underlying hardware. The HAL seeks to provide well-documented, easily-understood tools for software developers unfamiliar with the hardware to program AFTx07 to a similar effect.

1.1 Interrupts Background

Assembly instructions are stored on-chip, uploaded to read-only-memory (ROM) during the linking and loading phase of code compilation. During execution in a program, these instructions are fetched, decoded, and run in real-time to alter the state of the processor. Thus, the results of a computer program are determined while writing the program, not during run-time.

This is **synchronous programming**, in-which all instructions are executed one-by-one according to a predetermined schedule. However, the world of embedded programming frequently requires asynchronous responses to unexpected events. For example, using a DMA to transfer memory from one address to another may take an unexpected amount of time; however, a flag can be altered to show the completion of the process and avoid **polling** (the unnecessary devotion of computer resources to repeatedly check whether a task has been completed). This is the purpose of an interrupt: to warn the processor of unexpected events that it must address.

1.2 Asynchronous Programming

An interrupt enables an asynchronous response to any unexpected event. In the embedded world, there are many examples of unexpected events: the push of a

button, the turn of a dial, the completion of an external communication (using SPI/I2C/UART), or the periodic pulse of a timer.

If an interrupt occurs, it tells the processor that a more important task needs its attention. First, the processor pushes all its necessary registers onto the memory stack for storage. These registers represent the current workload – including the instruction the processor is executing and the data necessary for computation. Then, with a clean slate, the processor jumps to an **interrupt service routine (ISR)**, which is software specifying how the processor should address the current issue. Once the issue has been addressed, a **register reload** will initiate, popping the stored register values from the stack and restarting the execution of sequential tasks from exactly where the processor left off.

The hardware interrupt manager, the **Platform-Level Interrupt Controller (PLIC)**, assigns an identification to each interrupt source, binding an interrupt to a specific numeric id. Consequently, whenever an interrupt is handled by the processor, it knows which ISR it should "jump" to. Thus, specific peripheral devices can be bound to unique ISR handlers. This enables the processor to avoid wasteful polling while waiting for a peripheral to complete a task. Instead, the peripheral may indicate to the processor the completion of its duty using an interrupt.

With asynchronous programming, empowered by interrupts, a processor can now efficiently jump between different interrupt service routine handlers to address unexpected issues whenever they arise. This enables a single hardware thread to control a number of different hardware peripherals instead of being devoted to a single function.

2 The Interrupt Manager

The interrupt manager is the primary interface between developers and hardware interrupts. Three classes of interrupts are exposed to software developers in RISC-V (the instruction set architecture of the system-on-chip): software, timer, and external.

2.1 Interrupts in Hardware

The hardware behind interrupts differs between each hardware source. Furthermore, the AFTx07 chip has two different interrupt managers: the **Core-Local Interrupt Controller (CLINT)** and the **Peripheral-Level Interrupt Controller (PLIC)**. The "core-local" in the name of the CLINT indicates that it processes interrupts local to a single core of the processor. As such, it controls **timer interrupts** and **software interrupts**. The PLIC controls all interrupts arising from peripheral devices external to the processor itself. These interrupt sources are called **external interrupts**.

These interrupt managers contain register-blocks controlling the flow of interrupts. Using software, these registers can be manipulated to enable, disable, trigger, clear, and prioritize interrupts of each class. The HAL accesses these

registers by declaring a structure starting at the base address of the interrupt manager.

The register structure of the CLINT:

Register	Function
msip	trigger or clear a machine-mode software interrupt
mtime	counts the number of cycles of the system clock from startup
mtimecmp	if `mtime > mtimecmp`, triggers a machine-mode timer interrupt

The register structure of the CLINT as defined in the HAL:

```
// CLINT register block
typedef struct {
    __IO uint32_t msip;
    __IO uint32_t mtime;
    __IO uint32_t mtimeh;
    __IO uint32_t mtimecmp;
    __IO uint32_t mtimecmph;
} CLINTRegBlk;
```

The register structure of the PLIC:

Register	Function
iprior[N]	block of N registers storing each interrupt priority
ipndgr	each bit refers to whether that interrupt is pending or not
ier	each bit refers to whether that interrupt is enabled or not
ptr	sets a priority threshold, interrupts of lesser priority are ignored
ccr	claim and complete interrupts by reading id, then writing back id

The register structure of the PLIC as defined in the HAL:

```
// PLIC register block
#define N_INTS 32
typedef struct {
    uint32_t reserved0;
    __IO uint32_t iprior[N_INTS];
    __IO uint32_t ipndgr;
    __IO uint32_t ier;
    uint32_t reserved1;
    __IO uint32_t ptr;
    __IO uint32_t ccr;
} PLICRegBlk;
```

2.2 Interrupts in Software

Once the hardware has determined the origin of the interrupt and its corresponding interrupt source, it is able to branch to a corresponding interrupt service routine (ISR). There are three classes of interrupts – software, timer, and external. Thus, there are three base ISRs available for a developer to define.

The `mcause` register is the primary register in determining the source of the interrupt. Upon the launch of a trap – a synchronous or asynchronous exception/interrupt to the regular flow of the program – it provides a 32-bit code uniquely identifying the type of interrupt or exception.

`mcause` register codes:

`mcause` register codes (exceptions excluded):

Interrupt	Code	Result
1	0	user software interrupt
1	1	supervisor software interrupt
1	2	reserved
1	3	machine software interrupt
1	4	user timer interrupt
1	5	supervisor timer interrupt
1	6	reserved
1	7	machine timer interrupt
1	8	user external interrupt
1	9	supervisor external interrupt
1	10	reserved
1	11	machine external interrupt

Once the interrupt source has been identified, the hardware must know the next instruction to decode. In other words, it must identify where to jump to in its read-only-memory. The trap handler register `mtvec` assists with this. The register is given an address and a mode of operation. The address refers to where the processor should "jump" to once an interrupt occurs. The mode of operation is set to *vectored* on HAL initialization. A table can be created at this address and, when set to vectored mode, the hardware will use the code from `mcause` to leap to a certain instruction in the **vector table**.

Vector table initialization code:

```
// Sets up interrupt vector table
void __attribute((naked)) __attribute((aligned(4))) vtable( ) {
    asm volatile(".option push");
    asm volatile(".option norvc");
    asm volatile("j exception_handler");
    asm volatile("j default_handler");
    asm volatile("j default_handler");
    asm volatile("j m_sw_handler");
    asm volatile("j default_handler");
    asm volatile("j default_handler");
    asm volatile("j default_handler");
    asm volatile("j m_timer_handler");
    asm volatile("j default_handler");
    asm volatile("j default_handler");
    asm volatile("j default_handler");
```

```
    asm volatile("j m_ext_handler");
    asm volatile(".option pop");
}
```

This table establishes that all exceptions go the the `exception_handler`, that all non-machine mode interrupts go to `default_handler`, and that machine-mode software, timer, and external handlers go to `m_sw_handler`, `m_timer_handler`, and `m_ext_handler` respectively.

At this point, these functions can be declared and written. Whenever a corresponding interrupt is triggered, the respective ISR handler will be called to address the interrupt.

2.3 Structure

The HAL is implemented using C++ object-orientated principles. Each hardware component has its functionality defined as a class meanwhile the HAL is defined as the overarching namespace.

For example, to access a function in the HAL, the HAL namespace must be declared by using `HAL::` in front of the function name.

```
#include "hal.h"
int main() {
   HAL::hal_init();
}
```

The `hal_init()` function initializes the modules inside of the HAL such as the interrupt manager.

To use a function inside of the IntMgr class, the HAL namespace must be declared and the IntMgr class name must also be declared using `HAL::IntMgr::` in front of the function name.

```
#include "hal.h"
int main() {
   HAL::hal_init();
   HAL::IntMgr* intmgr = HAL::IntMgr::get();
}
```

The `IntMgr *intmgr` variable is a pointer to an instantiated object of the IntMgr class, on which class methods (or functions) can be used. This object must be instantiated to use the IntMgr. The `get()` function retrieves and returns this object pointer. Once these functions have been executed, the IntMgr is ready to use.

2.4 Software Interrupts

2.4.1 What is a Software Interrupt

A software interrupt is the only synchronous interrupt, which can be triggered by setting the `msip` register equal to 1. The `msip` stands for machine software

interrupt pending register and indicates whether a software interrupt is waiting to be handled or not. The ISR `m_sw_handler()` should reset the `msip` register to 0 to ensure the interrupt is not immediately re-invoked.

2.4.2 Using the HAL

Enabling and disabling software interrupts:

```
void IntMgr::enable_sw_interrupts()
void IntMgr::disable_sw_interrupts()
```

These functions enable and disable machine-mode software interrupts. By default, software interrupts are enabled on HAL initialization. Thereafter, they can be disabled and enabled by the user.

Triggering and clearing software interrupts:

```
void IntMgr::trig_sw_int()
void IntMgr::clr_sw_int()
```

These functions trigger and clear machine-mode software interrupts. The program must trigger software interrupts for them to occur. Then, once triggered, the ISR for software interrupts must clear the `msip` register so that the interrupt is not immediately re-invoked. The HAL includes this automatically.

Setting and getting software interrupt handlers:

```
void IntMgr::set_swISR(void (*handler)())
void (*IntMgr::get_swISR())()
```

These functions set and get the user-defined software ISR in the form of a function pointer. The user must define a function they want to run during every software interrupt. The name is not important, nor does it need to clear the `msip` register. A pointer to this function must be sent to the interrupt manager using the `set_swISR()` method. The HAL handles the hardware-specific code and calls the user-defined function every interrupt.

The HAL ISR

```
void __attribute__((interrupt)) m_sw_handler() {
    void (*user_swISR)() = INTMGR.get_swISR();
    if (user_swISR != NULL) {
        (user_swISR)();
    }

    INTMGR.clr_sw_int();
}
```

2.5 Timer Interrupts

2.5.1 What is a Timer Interrupt

A timer interrupt is a periodic or single-pulse interrupt based on the system clock of the chip. The `mtime` register is a simple upcounter holding the total number of clock cycles since reset. The `mtimecmp` register is the compare value for `mtime`. If the value in `mtime` becomes greater or equal to the value in `mtimecmp`, an interrupt will occur. When an interrupt does occur, the timer interrupt pending bit must be cleared by writing to the `mtimecmp` register.

2.5.2 Using the HAL

Enabling and disabling timer interrupts:

```
void IntMgr::enable_tm_interrupts()
void IntMgr::disable_tm_interrupts()
```

These functions enable and disable machine-mode timer interrupts. By default, timer interrupts are *not* enabled on HAL initialization.

Setting and getting timer interrupt handlers:

```
void IntMgr::set_timerISR(void (*handler)(uint32_t), uint32_t interval)
void (*IntMgr::get_timerISR(uint32_t *increment))(uint32_t)
```

These functions set and get the user-defined timer ISR in the form of a function pointer. The user must define a function they want to run during every timer interrupt. The name is not important, nor does it need to clear the pending bit by writing to `mtimecmp`. A pointer to this function must be sent to the interrupt manager using the `set_timerISR(...)` method. The HAL handles the hardware-specific code and calls the user-defined function every interrupt.

The user handler must accept a `uint32_t` (unsigned 32-bit integer) parameter. The HAL will pass this function the value of `mtime` at which the interrupt was triggered. Alongside the handler, the user must pass an interval to `set_timerISR(...)` which represents the amount `mtimecmp` will be offset by. This starts the countdown to the next timer interrupt.

Adjusting mtime and mtimecmp registers:

```
uint32_t IntMgr::read_mtime()
uint32_t IntMgr::read_mtimecmp()
void IntMgr::write_mtimecmp(uint32_t value)
void IntMgr::increment_mtimecmp(uint32_t increment)
void IntMgr::increment_mtimecmp_past_mtime(uint32_t increment)
```

These functions adjust the values in the timer interrupt related registers. The first two return 32-bit integer values read from `mtime` and `mtimecmp`. The third writes to `mtimecmp`. The fourth adds an increment to the current value of `mtimecmp`. The fifth takes the current value in `mtime` and adds an increment

before writing this value to `mtimecmp`. The fifth function is the most useful for the user to start the countdown to a timer interrupt at any point in the program (where `mtime` may be non-zero).

The HAL ISR

```
void __attribute__((interrupt)) m_timer_handler() {
    uint32_t user_mtimecmp_interval;
    void (*user_timerISR)(uint32_t) =
              INTMGR.get_timerISR(&user_mtimecmp_interval);

    // increment the mtimecmp register forward
    if (user_mtimecmp_interval > 0) {
       INTMGR.increment_mtimecmp(user_mtimecmp_interval);
    } else {
       INTMGR.increment_mtimecmp(0xFFFF);
       INTMGR.disable_tm_interrupts();
    }

    if (user_timerISR != NULL) {
       (*user_timerISR)(INTMGR.read_mtime());
    }
}
```

2.6 External Interrupts

2.6.1 What is an External Interrupt

An external interrupts originates from some peripheral device *external* to the processor itself. It is the primary method of communication between a peripheral device and the processor itself, indicating when certain tasks are complete or when errors have occurred. The PLIC governs the functioning of external interrupts, and they are the most complicated form of interrupt.

The are 32 external interrupt sources on AFTx07. The first (id 0) is reserved and the remaining 31 (id 1-31) are free to be "mapped" to certain peripherals. This mapping is done in the hardware design process. For example, certain pins of GPIO may be tied to specific interrupt sources. If interrupts are enabled in GPIO, a change on pin 1 may incite an interrupt while a subsequent change on pin 3 could trigger a different interrupt. This enables writing complex and case-specific interrupt handlers for certain peripherals.

Since there are several types of external interrupts, the PLIC allows users to alter the priority of each interrupt id. If an interrupt with higher priority is pending while another interrupt is in the middle of executing, the processor will initiate another "save" and then begin working immediately on the more important task. This process can repeat until stack overflow occurs or all interrupts are handled.

Furthermore, a priority threshold can be set. Interrupts with priority values beneath this threshold cannot trigger. Specific external interrupts can also be selectively enabled and disabled (disabled by default) using the interrupt enable register.

2.6.2 Using the HAL

Enabling and disabling external interrupts:

```
void IntMgr::enable_ext_interrupts()
void IntMgr::disable_ext_interrupts()
```

These functions enable and disable machine-mode external interrupts. By default, external interrupts are *not* enabled on HAL initialization.

Setting and getting external interrupt handlers:

```
void IntMgr::set_extISR(void (*handler)(uint32_t), uint32_t int_id)
void (*IntMgr::get_extISR(uint32_t int_id))(uint32_t)
```

These functions set and get the user-defined external ISR in the form of a function pointer. The user must define a function they want to run during each unique external interrupt. The name is not important, nor does it need to claim or complete the interrupt in the PLIC. A pointer to this function must be sent to the interrupt manager using the `set_extISR(...)` method. This user-defined function must also accept an unsigned 32-bit integer as a parameter (the interrupt id). This value does not need to be used in any specific way, but can be useful to identify the interrupt source. The HAL handles the hardware-specific code and calls the user-defined function every interrupt.

Since there are several external interrupts, an interrupt id (1-31) must be sent alongside the function pointer. The HAL stores a block of function pointers and the function will set the pointer corresponding to the index (id).

Claiming and completing external interrupt handlers:

```
uint32_t IntMgr::claim_extISR()
void IntMgr::complete_extISR(uint32_t id)
```

These functions claim and complete an external interrupt. Claiming an external interrupt must occur so that only one core of the processor operates on a specific interrupt (since the PLIC is device wide). It is done by reading the `ccr` register of the PLIC when an interrupt is pending. In such a case, if a non-zero id is read, an interrupt must be addressed.

Completing a register is done once the ISR has completed, indicating to the PLIC that the interrupt has been handled. This is similar to clearing the pending bit of a software or timer interrupt. The HAL handles both claiming and completing the external interrupt.

Enabling and disabling specific external interrupt handlers:

```
void IntMgr::enable_extISR(uint32_t int_id)
void IntMgr::enable_extISR(uint32_t int_id)
```

Specific external interrupts can be enabled or disabled based on their ids (1-31). These functions manipulate the PLIC `ier` register.

Prioritizing specific external interrupt handlers:

```
void IntMgr::set_priority_extISR(uint32_t priority, uint32_t int_id)
void IntMgr::set_threshold_extISR(uint32_t priority_threshold)
```

Specific external interrupts can be given higher or lower priority based on their ids (1-31). The first function sets the `iprior` register, changing the priority of one interrupt. The second function sets the `ptr` register, changing the priority threshold of the PLIC. This priority threshold sets the minimum priority value that an external interrupt must have to be triggered (effectively disabling interrupts of lower priority).

The HAL ISR

```
void __attribute__((interrupt)) m_ext_handler() {
   uint32_t int_id = INTMGR.claim_extISR();
   void (*user_extISR)(uint32_t) = INTMGR.get_extISR(int_id);
   if (user_extISR != NULL) {
      (*user_extISR)(int_id);
   }

   INTMGR.complete_extISR(int_id);
}
```

3 Appendix

3.1 IntMgr Class

```
class IntMgr {
   public:
      IntMgr();
      static IntMgr* get();
      IntMgr(IntMgr const&)=delete;
      void operator=(IntMgr const&)=delete;

      void trig_sw_int();
      void clr_sw_int();
      void entr_critsec();
      void exit_critsec();
      void reg_int_cb(void (*cb)(), IRQMap irq);
```

```
    // MTIME and MTIMECMP read/write commands
    uint32_t read_mtime();
    uint32_t read_mtimecmp();
    void write_mtimecmp(uint32_t);
    void increment_mtimecmp(uint32_t);
    void increment_mtimecmp_past_mtime(uint32_t);

    // ENABLE/DISABLE interrupts
    void enable_tm_interrupts();
    void enable_sw_interrupts();
    void enable_ext_interrupts();
    void disable_tm_interrupts();
    void disable_sw_interrupts();
    void disable_ext_interrupts();

    // SET and GET user-callback handlers
    void set_timerISR(void (*)(uint32_t), uint32_t);
    void set_swISR(void (*)());
    void set_extISR(void (*)(uint32_t), uint32_t);
    void (*get_timerISR(uint32_t *))(uint32_t);
    void (*get_swISR())();
    void (*get_extISR(uint32_t))(uint32_t);

    // EXTERNAL interrupts functions
    void enable_extISR(uint32_t);
    void disable_extISR(uint32_t);
    void set_priority_extISR(uint32_t, uint32_t);
    void set_threshold_extISR(uint32_t);

    // CLAIM and COMPLETE external interrupts
    uint32_t claim_extISR();
    void complete_extISR(uint32_t);

private:
    static bool intmgr_init;
    uint32_t mstatus_last;
    void (*irq_cb[5])() = { nullptr };
    void setup_vtable();

    CLINTRegBlk* clint_reg_blk;
    PLICRegBlk* plic_reg_blk;

    // User callback ISR handlers
    uint32_t user_mtimecmp_interval = 0;
    void (*user_timerISR)(uint32_t) = NULL;
    void (*user_swISR)() = NULL;
```

```
        void (*user_extISR[N_INTS])(uint32_t);
};
```