71M65XX Energy Meter ICs

TERIDIAN SEMICONDUCTOR CORP.

AB_65XX_006 **APPLICATION BRIEF**

AUGUST 2007

Precautions for Interrupts

Meter code for both the 71M651X and 71M652X Energy Meter chip family services a multitude of interrupts while not losing data. Carefully balancing the interrupt priorities and interrupt timing will achieve this goal. This Application Brief describes the main concepts for interrupt processing.

Interrupts Serviced by the 80515

A brief look at the interrupt structure of the 80515, as shown in Figure 1, confirms the complexity of the interrupt system required in a modern meter.

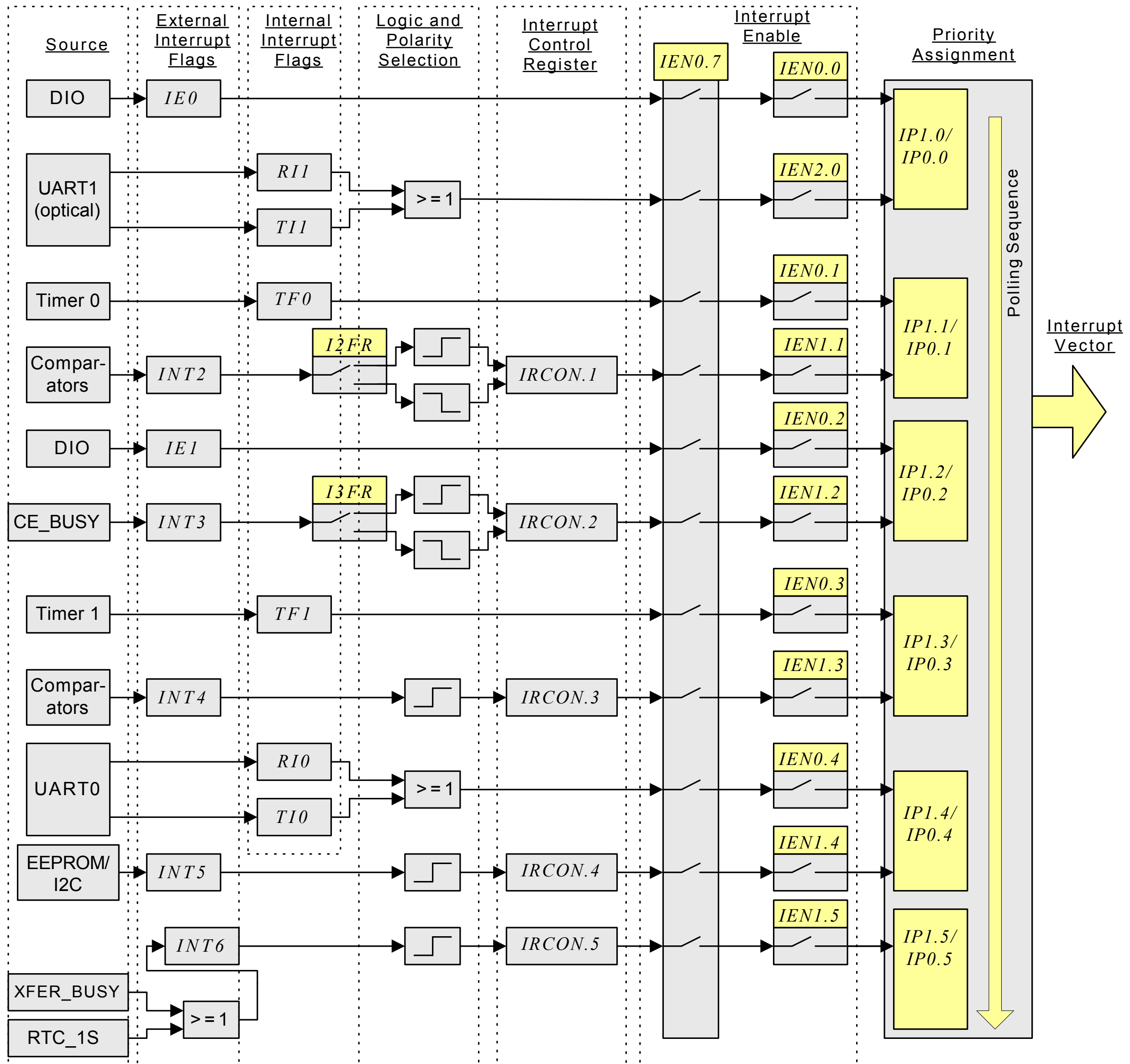

Figure 1: Interrupt Structure of the 80515

Internal and external interrupt sources have to be served, while no loss of data is permitted. It is easy to imagine how the processing of one interrupt could “starve” other interrupts and lead to data loss. Careful assignment of interrupt priorities and good interrupt processing techniques prevent lock-ups and data loss.

The following interrupts, as appearing in Figure 1 from top to bottom, are used in a typical meter based on the TERIDIAN 71M651X and 71M652X architectures:

1) DIO (IE0 flag): This interrupt occurs when a state change happens to a DIO pin configured to generate high-priority interrupts via its associated DIO_R register. Pulse counting is a typical application for this interrupt, and pulses can appear with frequencies up to several hundred per second.
2) UART1 (RI1, TI1 flags): An interrupt occurs when a character has been sent or received via UART1, which is typically used for optical communication at fairly low baud-rates (300bd).
3) TIMER0 (TF0 flag): This timer can be used to support so called software timers. A timer “tick” interrupt may occur every milli-second.
4) Comparators (INT2 flag): Only the 71M6513 offers the V2 and V3 comparators, which might be used for analog measuring or monitoring purposes. Events on these comparators are usually not very frequent.
5) DIO (IE1 flag): This is another interrupt caused by a transition of a DIO pin. In this case, a low-priority interrupt is selected by the DIO_R register.
6) CE_BUSY (INT3 flag): This interrupt will occur every 396μs (with a default setting of MUX_DIV). The MPU needs to service this interrupt in order to gain information on sag and other conditions contained in the CE STATUS word.
7) TIMER1 (TF1 flag): Applications can have a separate timer, e.g. for delay functions required when controlling EEPROM or RTC.
8) UART0 (RI0, TI0 flags): An interrupt occurs when a character has been sent or received via UART0, which is typically used for communication with AMR, logging, calibration systems or other equipment. The baud-rate used on UART0 may be fairly high (typically 9600bd).
9) EEPROM/I2C (INT5 flag): An interrupt occurs when an I2C operation, such as write, read, or reset has been completed. Meters use this interface to store revenue data and other information in EEPROMs. Many meters store revenue data routinely, e.g. once per day, while still metering. This means that the EEPROM interrupts have to be processed in the presence of all other interrupts.
10) XFER_BUSY/RTC_1S (INT6 flag): These two sources trigger the most important interrupt of the meter. XFER_BUSY signals to the MPU that the CE has collected energy information in the accumulation interval that was just completed. The MPU cannot afford to lose the energy data, but fortunately this interrupt occurs typically only every 999.76ms (with default settings for *PRE_SAMPS* and *SUM_CYCLES*). Secondly, the CE holds the data valid until the end of the next accumulation interval, so XFER_BUSY does not have to be served immediately. RTC_1S occurs when the RTC has completed one second. RTC_1S and XFER_BUSY are not synchronized in any way, but they are OR-ed together into the INT6 flag. It is important to check both interrupt conditions in the interrupt service routine. Otherwise, a pending RTC_1S interrupt can be overlooked when servicing the XFER_BUSY interrupt. Since the interrupt is edge-sensitive, the next XFER_BUSY event would not generate an interrupt due to RTC_1S being 1 (see TERIDIAN Application Brief AB_651X_001 for details on this issue).

Interrupt Priority Settings

Priorities can be assigned to interrupts in groups, as shown in Table 2, using the IP0 (SFRA9) and IP1 (SFRB9) registers of the 80515 MPU shown in Table 1.

IP1.x	IP0.x	Priority Level
0	0	Level0 (lowest)
0	1	Level1
1	0	Level2
1	1	Level3 (highest)

Table 1: Priority Levels

IP0/IP1 Bits	Group	Group Members		
IP1.0, IP0.0	0	EXT0	RI1/TI1	
IP1.1, IP0.1	1	TF0	-	EXT2
IP1.2, IP0.2	2	EXT1	-	EXT3
IP1.3, IP0.3	3	TF1	-	EXT4
IP1.4, IP0.4	4	RI0/TI0	-	EXT5
IP1.5, IP0.5	5	-	-	EXT6

Table 2: Groups of Priority

When examining the IP0 and IP1 registers in typical TERIDIAN Demo Code, we observe the bit pattern shown in Table 3.

Group	IP1 Bit	Value	IP0 Bit	Value	Resulting Level
0	IP1.0	1	IP0.0	0	2
1	IP1.1	0	IP0.1	0	0
2	IP1.2	1	IP0.2	1	3
3	IP1.3	0	IP0.3	0	0
4	IP1.4	1	IP0.4	0	2
5	IP1.5	0	IP0.5	1	1

Table 3: Bit Assignments for IP0 and IP1

This bit assignment reflects the interrupt priorities shown in Table 4.

Interrupt Routine	Interrupt	Group	Priority
io_high_priority_isr	EXT0	0	2
io_low_priority_isr	EXT1	2	3
compare_falling_isr	EXT2	1	0
ce_busy_isr	EXT3	2	3
compare_rising_isr	EXT4	3	0
eeprom_isr	EXT5	4	2
xfer_busy_isr	EXT6 (shared w/ RTC)	5	1
timer0_isr	TF0	1	0
timer1_isr	TF1	3	0
rtc_isr	EXT6 (shared w/ XFER)	5	1
es0_isr	RI0/TI0	0	2
es1_isr	RI1/TI1	4	2

Table 4: Interrupt Service Routines

The highest priority is assigned to CE_BUSY, along with EXT1, which happens to be in the same group. EXT0 (I/O), EXT5 (EEPROM/I2C), and the UARTs are next in line with priority 2, while XFER_BUSY and RTC_1S have the next lower priority. To sum it up, fast or urgent interrupts need higher priority, but they also have to be processed quickly.

Details on Interrupt Processing

In addition to selecting the proper interrupt priorities, it is important to observe the following precautions for the processing of interrupts:

1) Since the memory transfers and comparisons associated with the processing of the CE STATUS word (sag processing) tend to load down the MPU, the TERIDIAN Demo Code executes this type of processing only once per every eight CE_BUSY interrupts. This means that sag bits are in the worst case discovered after 3.1ms, which is still a very acceptable time frame for the processing of sag events.
2) While executing interrupt service routines, the TERIDIAN Demo Code uses the `irq_disable()` function to disable all interrupts. Similarly, soon after completing an interrupt service routine, the Demo Code uses the `irq_enable()` function to enable interrupts. This technique works efficiently and prevents “starving“ of interrupts.
3) IP0 and IP1 should be set only once, preferably near the start of initialization, and then never changed again. Changing IP0 and IP1 during meter program execution can have undesired effects.
4) “Critical” code regions should be protected from being interrupted. Critical regions are code sequences that fail if they are interrupted. For example, in
   `variable |= mask;`
   the instructions that read `variable` and write `variable` could be interrupted by code that overwrites `variable` causing incorrect operation. Another example is the code that accesses CE RAM, as shown below in the `ce_busy_isr()` service routine. CE RAM is accessed in multiples of four bytes. If an interrupt separates the operation (read or write) before the fourth byte is completed, the data will be corrupted, because the unprocessed bytes may have changed when the unfinished code resumes.
5) All interrupt service routines and the routines called by them must be reentrant.
6) Priority inversion happens when a lower-priority interrupt prevents a higher-priority interrupt from finishing in a timely manner. One common cause in meter code is the attempt to protect critical regions by disabling interrupts individually. It can occur that the code that enables a high-priority interrupt has to wait in the main loop for a low-priority (but enabled) interrupt to complete. The cure in this case is to leave individual interrupts enabled, and disable all interrupts (wtih EA) briefly (less than 100μs) for critical regions. The listing of `irq.c` below shows an example for this technique.

Example Source Files

irq.c is listed below:

```
#include "options.h"
#include "api.h"

/*** Private variables declared within this module ***/
// This uses less space than a system that places the interrupt state on the stack.
// This is almost as fast, when reentrant variables are used (as they often must be).
Bbool irq_state = 1;    // the "outer" state of a nested set of states.
U08 nest_cnt = 0;      // counts nesting of interrupt-disable calls

// This routine should be called at the start of a critical region.
// It disables the interrupt state, and can be called from nested subroutine calls.
#pragma save
#pragma NOAREGS
void irq_disable (void) small reentrant
{
    // local variable of a reentrant fn is mutex-safe
    register U08 ea_temp;
    ea_temp = EA;        // copy the interrupt state
    EA = FALSE;          // disable all interrupts
    // The increment is done while interrupts are disabled,
    // and is therefore reentrant and mutex-safe, even though it uses
    // a global variable.
    if (nest_cnt++ == 0) // if at the outer call, save the state
    {
        irq_state = ea_temp;
    }
    // otherwise, the interrupt state is managed- discard it.
}
#pragma restore

// This routine should be called at the end of a critical region.
// It restore the interrupt state, and can be called
// from nested subroutine calls.
#pragma save
#pragma NOAREGS
void irq_enable (void) small reentrant
{
    // The decrement is done while interrupts are disabled,
    // and is therefore reentrant and mutex-safe.
    if (--nest_cnt == 0) // if at the outer call, restore the state
    {
        EA = irq_state;  // restore the saved interrupt state
    }
    // otherwise, the interrupt state is nested, and should
    // remain disabled.
}
#pragma restore

void irq_init (void)
{
    EA = FALSE;        // assure that interrupts start disabled
    irq_state = TRUE;  // and will resume as enabled
    nest_cnt = 1;      // from an unbalanced first call of irq_enable ()
}
```

Listing of irq.h:

```
//  Disable interrupts; nestable and reentrant
void irq_disable (void) small reentrant;
//  Enable interrupts; nestable and reentrant
void irq_enable (void) small reentrant;
//  Sets up for the above; should be called early in initialization.
void irq_init (void);

// Use less time, more space, an incompatible scheme
#define IRQ_DEFINES  U08 ea = EA
#define IRQ_DISABLE() EA = FALSE
#define IRQ_ENABLE() EA = ea
```

`ce_busy_isr()` Interrupt Routine from `ce.c`:

```
// Runs 396us after the XREF_BUSY_INT, to alternate the polarity of the chop on VRef.
// If sag detection is enabled, runs every 396us
#pragma save
#pragma NOAREGS
void ce_busy_isr (void) small reentrant
{
    CE2 = (CE2 & ~CHOP_EN) | CHOP_EN;   // start hardware chop on next cycle
    #if 1 == SAG_DETECT // 1 = rapid sag detection, flag from reg651x.h
    // this code is run less often in order to reduce the real-time
    // cost of running this interrupt.
    if (sag_decimation <= 0)
    {
        #define ea 1
        U08 ck = CKCON;
        sag_decimation = 6;                // run every 6*396us = 2.4ms
        // read the compute engine's sag bits from its status register
        CE_BEGIN_CRITICAL_SECTION;         // Disable CE interrupts.
        CLK_STRETCH;                       // Change stretch to '6' equivalent.
        // read the CE status's sag bits
        sag_data  = *((U08x *)&CE.Outputs.O_cestatus);
        CLK_RELAX;                         // Back to default value.
        CE_END_CRITICAL_SECTION;
        #undef ea
        *(U08 xdata *)&Status = sag_data;  // save to the meter's status word
        // Are the compute-engine's sag detection bits set for all the phases
        // that power the meter?  (POWERED_PHASE is in options.h)
        if ((sag_data & POWERED_PHASE) == POWERED_PHASE)
        {
            // Save when the power fails, not when it is off.
            if (meter_had_power)
            {
                EA = 0;  // disable interrupts
                // mark the data save with hardware
                // OSCOPE_INIT;
                // OSCOPE_TOGGLE;
                RESET_WD(); // push off the hardware watchdog
                meter_had_power = FALSE;
                // save the data to EEPROM (this operation is not reentrant)
                memcpy_px (
                    EEPROM_REGISTERS,
                    (U08x*)&Totals.Acc, // The start of the data
                    (2*sizeof(struct Accumulators_t)));  // save two copies
                // OSCOPE_TOGGLE;
                Soft_Reset(); // recover from nonreentrant code
                for(;;);  // make it impossible to pass this point
            }
        }
    }
    else
    {
        sag_decimation--;
    }
    #else // if no rapid sag detection is needed
    EX_CE_BUSY = 0;  // disable this interrupt until the next xfer_busy ISR
    #endif
```

This product is sold subject to the terms and conditions of sale supplied at the time of order acknowledgment, including those pertaining to warranty, patent infringement and limitation of liability. TERIDIAN Semiconductor Corporation (TSC) reserves the right to make changes in specifications at any time without notice. Accordingly, the reader is cautioned to verify that the information is current before placing orders. TERIDIAN assumes no liability for applications assistance.

TERIDIAN Semiconductor Corp., 6440 Oak Canyon Rd., Irvine, CA 92618

TEL (714) 508-8800, FAX (714) 508-8877, http://www.teridian.com

© 2004-2007 TERIDIAN Semiconductor Corporation