Software Interrupts - SWI

Introduction

In this chapter the second BIOS thread type – the Software Interrupt, or “SWI” will be investigated. Comparisons and contrasts to the previously covered HWI will be made. A variety of options for posting SWIs will be considered including examples of when each type might be preferred.

Objectives

At the conclusion of this module, you should be able to:

• Describe the basic concepts of SWIs
• Demonstrate how to post a SWI
• Describe the SWI object
• List several SWI posting options
• Define the benefit of each SWI posting method
• Add a SWI to an HWI-based system

Module Topics

Software Interrupts - SWI .............................................................. 4-1

  Concepts ................................................................. 4-2
  Posting a SWI ......................................................... 4-5
  The SWI Object ...................................................... 4-9
  SWI API Review ....................................................... 4-11
  Queues (QUE) .......................................................... 4-12
  Block FIR Concepts .................................................. 4-14
  Lab 4: SWI-Based System ........................................... 4-17
    A. SWI Based procBuf() ............................................. 4-18
    B. Passing Buffers Via QUEues ................................. 4-19
    C. (Optional) Interrupt Keyword / Dispatcher Conflict .... 4-20
Concepts

New Paradigm: DSP/BIOS Scheduler

- SWI_post is equivalent to setting ISR flag
- Scheduler replaces the while loop
- SWI manager is like ‘if’ test with no overhead

Hardware and Software Interrupt System

Execution flow for flexible real-time systems:

- HWI
- SWI
- INT ! Hard R/T Process Post SWI Cleanup, RETURN

HWI
- Fast response to interrupts
- Minimal context switching
- High priority for CPU
- Limited number of HWI possible

SWI
- Latency in response time
- Context switch performed
- Selectable priority levels
- Execution managed by scheduler

- DSP/BIOS provides for HWI and SWI management
- DSP/BIOS allows the HWI to post an SWI to the ready queue
DSP/BIOS Preemptive Scheduler

**Hardware Interrupts (HWI)**
- Urgent response time
- Often at “sample rate”
- Microseconds duty cycle
- Preemptive or non-preemptive

**Software Interrupts (SWI)**
- Flexible processing time
- Often at “frame rate”
- Milliseconds duty cycle
- Preemptive

**Idle (IDL)**
- Best Effort
- Sequential Execution

**Legend**
- Collect Samples
- Post SWI
- Process Buffer

**HWI** Collect data into frame/buffer, perform minimum processing
**SWI** Process each datum in buffer
**IDL** Runs when no real-time events are active
**HWI preempt SWI** - new data is not inhibited by processing of frame
State Diagrams: IDL, HWI, SWI

- **IDL**
  - Lowest priority - soft real-time - no deadline
  - Idle functions executes sequentially
  - Priority at which real-time analysis is passed to host

- **HWI & SWI**
  - Encapsulations of functions with priorities managed by DSP/BIOS kernel
  - Run to completion (cannot be suspended or terminated prior to completion)
  - Runs only once regardless of how many times posted prior to execution
Posting a SWI

Scheduling Rules

Highest Priority

- HWI
- SWI_b (p2)
- SWI_a (p1)

Lowest Priority

- IDL

Legend

- Running
- Ready

- SWI_post(&mySwi) : Unconditionally post a software interrupt (in the ready state)
- If a higher priority thread becomes ready, the running thread is preempted
- SWI priorities from 1 to 14
- Automatic context switch (uses system stack)

- Processes of same priority are scheduled first-in first-out
Posting a SWI from an HWI

- Problem: Scheduler not aware of interrupt!
- If ISR posts a higher priority SWI, the scheduler will run that SWI in the context of the HWI - not usually desired

Using the Dispatcher with HWI

- Solution: Use the Dispatcher
- Some APIs that may affect scheduling: SWI_post, SWI_andn, SWI_dec, SWI_inc, SWI_or, SEM_post, PIP_alloc, PIP_free, PIP_get, PIP_put, PRD_tick
Scheduling Strategies

- Most important “Deadline Monotonic”
  - Assign higher priority to the most important process

- Rate monotonic analysis
  - Assign higher priority to higher frequency events
  - Events that execute at the highest rates are assigned highest priority
  - An easy way to assign priorities in a system!
  - Systems under 70% loaded guaranteed to run successfully (proofs for this in published papers)
  - Also allows you to determine scheduling bounds

- Dynamic priorities
  - Raise process priority as deadline approaches

DSP/BIOS: Priority-Based Scheduling

```
int1
post2 rtn
post3 rtn
SWI_post(&swi_name);
```

```
int2
post1 rtn
```

```
MAIN
```

```
IDLE
```

```
```

```
```

```
```

```
```

```
```
Another Scheduling Example

The BIOS Execution Graph provides this kind of information to assist in temporal debugging.
The SWI Object

Creation of SWI with Configuration Tool

Creating a new SWI
1. right click on SWI mgr
2. select "Insert SWI"
3. type SWI name
4. right click on new SWI
5. select "Properties"
6. indicate desired
   • function
   • priority
   • mailbox value

SWI Attributes : Manage SWI Properties

- Allows programmer to inspect and modify key SWI object values
- Do not modify fields on preempted or ready to run SWI recommended: implement during lower priority thread
- Priority range is 1 to 14, inclusive
- Example - changing a SWI's priority to 5:

```c
extern SWI_Obj swiProcBuf;
SWI_Attrs attrs;
SWI_getattrs (&swiProcBuf, &attrs);
attrs.priority = 5;
SWI_setattrs (&swiProcBuf, &attrs);
```
The SWI Object

SWI Structures (from swi.h and fxn.h)

- SWI_Attrs contains the most commonly used SWI object elements
- SWI_getattrs and SWI_setattrs allow well defined access to these elements

```c
typedef struct SWI_Attrs {
  SWI_Fxn fxn;
  Arg arg0;
  Arg arg1;
  Int priority;
  Uns mailbox;
} SWI_Attrs;
```

```c
typedef struct SWI_Obj {
  Int lock;
  Ptr ready;
  Uns mask;
  Ptr link;
  Uns initkey;
  FXN_Obj *fxnobj;
  Int stslck;
  STS_Obj *sts;
} SWI_Obj;
```

- SWI object can be directly access also, if desired, as per these examples:

  ```c
  myValue = mySwi.fxnobj.arg1;
  mySwi.fxnobj.arg0 = 7;
  ```

Managing Thread Priorities via GCONF

- Drag-and-drop SWIs in list to vary priority
- Priorities range from 1–14

- Scheduler is invoked when SWI is posted
- When scheduler runs, control is passed to the highest priority thread
- Equal priority SWIs run in the order posted
SWI API Review

SWI Post and SWI Mailbox Overview

<table>
<thead>
<tr>
<th>API</th>
<th>Allows you to:</th>
</tr>
</thead>
<tbody>
<tr>
<td>SWI_inc</td>
<td>Know how many times the SWI was posted before it ran</td>
</tr>
<tr>
<td>SWI_dec</td>
<td>Post N times before the SWI is scheduled – a countdown</td>
</tr>
<tr>
<td>SWI_or</td>
<td>Send a single value to the SWI when posting - signature</td>
</tr>
<tr>
<td>SWI_andn</td>
<td>Only post the SWI when multiple posters all have posted</td>
</tr>
</tbody>
</table>

If the value of the mailbox is needed by the SWI, use SWI_getmbox() which returns the value of the mailbox when the SWI was posted.

Note: this is a ‘shadow’ value for use within the SWI – BIOS manages a second mailbox for the next posting of the SWI.

After each posting, the mailbox is reset to the initial condition specified in the SWI object.

---

SWI API Summary

<table>
<thead>
<tr>
<th>SWI API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SWI_post</td>
<td>Post a software interrupt</td>
</tr>
<tr>
<td>SWI_andn</td>
<td>Clear bits from SWI’s mailbox; post if becomes 0</td>
</tr>
<tr>
<td>SWI_or</td>
<td>Or mask with value contained in SWI’s mailbox field</td>
</tr>
<tr>
<td>SWI_inc</td>
<td>Increment SWI’s mailbox value</td>
</tr>
<tr>
<td>SWI_dec</td>
<td>Decrement SWI’s mailbox value; post if becomes 0</td>
</tr>
<tr>
<td>SWI_getatrts</td>
<td>Copy SWI attribute from SWI object to a structure</td>
</tr>
<tr>
<td>SWI_setatrts</td>
<td>Update SWI object attributes from specified structure</td>
</tr>
<tr>
<td>SWI_getmbox</td>
<td>Obtain the value in the mailbox prior to SWI run</td>
</tr>
<tr>
<td>SWI_create</td>
<td>Create a SWI</td>
</tr>
<tr>
<td>SWI_delete</td>
<td>Delete a SWI</td>
</tr>
<tr>
<td>SWI_disable</td>
<td>Disable software interrupts</td>
</tr>
<tr>
<td>SWI_enable</td>
<td>Enable software interrupts</td>
</tr>
<tr>
<td>SWI_getpri</td>
<td>Return a SWI’s priority mask</td>
</tr>
<tr>
<td>SWI_raisex</td>
<td>Raise a SWI’s priority</td>
</tr>
<tr>
<td>SWI_restorepri</td>
<td>Restore a SWI’s priority</td>
</tr>
<tr>
<td>SWI_self</td>
<td>Return current SWI’s object handle</td>
</tr>
</tbody>
</table>

Mod 11

Mod 7
Queues (QUE)

Queue Concepts

- QUE message is anything you like, starting with QUE_Elem
- QUE_Elem is a set of pointers that BIOS uses to manage a double linked list
- Items queued are **NOT** copied – only the QUE_Elem ptrs are managed!

```c
struct MyMessage {
    QUE_ELEM elem;
    Int x[1000];
} Message1;
```

```c
typedef struct QUE_ELEM {
    struct QUE_ELEM *next;
    struct QUE_ELEM *prev;
} QUE_ELEM;
```

**How do you synchronize reader and writer?**

Queue Usage

- QUE Properties
  - any number of messages can be passed
  - atomic API assure correct sequencing
  - no intrinsic semaphore

- Using QUE
  - Declare the QUE via the config tool
  - Define (typedef) the structure to queue – 1st element must be “QUE_ELEM”
  - Fill the message(s) to QUE with the desired data
  - Send the data to the queue via `QUE_put(&myQue, msg);`
  - Acquire data from the queue via `info=QUE_get(&myQue);`

- Application Considerations
  - Two queues are needed to circulate messages between two threads

```
typedef struct MsgObj {
    QUE_ELEM elem;
    short *pInBuf;
    short *pOutBuf;
} MsgObj, *Msg;
```
Queue Lab Steps

GCONF:
1. Declare 2 QUEues called toDev and toProc

ABOVE MAIN:
2. Declare 2 Input Buffers and 2 Output Buffers
3. Define a message structure to send over the QUEs:
   - first element of a QUE message must be of type QUE_Elem
   - 'payload' in this lab will be 2 pointers ( *pIn, *pOut )
4. Declare, as globals, an array of two messages of the type created above

IN MAIN:
5. Initialize the pointers in message 1 to point to in/out buffers 1
6. Initialize the pointers in message 2 to point to in/out buffers 2
7. "Prime" the toDev QUE with the two messages

HWI CODING:
8. At the top of the HWI, if the current buffer size drops to 0:
   - get a new message from the toDev QUE
   - extract the input buffer pointer from the message
   - extract the output buffer pointer from the message
   - as before, each time the HWI is posted:
     - fill 1 new input buffer word
     - output 1 output buffer word
9. At the bottom of the HWI, if the current buffer is full:
   - send the message with the current pointer set to the SWI via the toProc QUE
   - zero the buffer size counter

SWI CODING:
10. At the top of the SWI, get a message from the toProc QUE
    - extract the input buffer pointer from the message
    - extract the output buffer pointer from the message
    - as before, FIR filter the input buffer & store results to the output buffer
11. At the end of the SWI, return the message to the HWI via the toDev QUE

QUE API Summary

<table>
<thead>
<tr>
<th>QUE API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>QUE_put</td>
<td>Add a message to end of queue – atomic write</td>
</tr>
<tr>
<td>QUE_get</td>
<td>Get message from front of queue – atomic read</td>
</tr>
<tr>
<td>QUE_enqueue</td>
<td>Non-atomic QUE_put</td>
</tr>
<tr>
<td>QUE_dequeue</td>
<td>Non-atomic QUE_get</td>
</tr>
<tr>
<td>QUE_head</td>
<td>Returns ptr to head of queue (no de-queue performed)</td>
</tr>
<tr>
<td>QUE_empty</td>
<td>Returns TRUE if queue has no messages</td>
</tr>
<tr>
<td>QUE_next</td>
<td>Returns next element in queue</td>
</tr>
<tr>
<td>QUE_prev</td>
<td>Returns previous element in queue</td>
</tr>
<tr>
<td>QUE_insert</td>
<td>Inserts element into queue in front of specified element</td>
</tr>
<tr>
<td>QUE_remove</td>
<td>Removes specified element from queue</td>
</tr>
<tr>
<td>QUE_new</td>
<td>….</td>
</tr>
<tr>
<td>QUE_create</td>
<td>Create a queue</td>
</tr>
<tr>
<td>QUE_delete</td>
<td>Delete a queue</td>
</tr>
</tbody>
</table>
Block FIR Concepts

Block FIR Filter Overview

- Read block of data to input buffer
- Convolve 1st N samples with coefficients
- Store result to 1st location of output buffer

```
  0 x c0 +
  0 x c1 +
  0 x c2 +
  0 x c3 +

A/D   SP
HWI
...
...
95
96
97
98
99
```

- Repeat convolution advanced by 1 sample
- Store result to 2nd location of output buffer
- Repeat for BLOCKSIZE iterations
- Send output buffer to DAC
- Copy last N-1 samples to history pre-buffer
- Repeat above steps...

```
  96 x c0 +
  97 x c1 +
  98 x c2 +
  99 x c3 +

A/D   SP
HWI
...
...
195
196
197
198
199
```

```
  100 =
100
```

```
D/A
SP
HWI
```
Block FIR Filter Overview

- Read block of data to input buffer
- Convolve 1st N samples with coefficients
- Store result to 1st location of output buffer
- Repeat convolution advanced by 1 sample
- Store result to 2nd location of output buffer
- Repeat for BLOCKSIZE iterations
- Send output buffer to DAC
- Copy last N-1 samples to history pre-buffer
- Repeat above steps...

\[ 96 \times c_0 + 97 \times c_1 + 98 \times c_2 + 99 \times c_3 + 100 \times c_4 = 100 \]
**Double Buffer Management**

1. Get block of data
2. Start collecting next block while processing data
3. Prime for next block
4. Start collecting next block while processing data
5. Prime for next block
6. ...

- Make data buffers size of block plus history
- Collect data in last 'blocksize' locations
- After buffer is processed, copy last 'history' values to top of other buffer

**Interlaced Stereo Double Buffers**

```c
#define FIRSZ 64
#define HIST (2*FIRSZ-2)
short in[2][2*BUF+HIST];
short out[2][2*BUF];

for (i = 0; i < HIST; i++)
    pln[i] = pPriorIn[i+2*BUF-HIST];
pPriorin = pln;

fir(pln-HIST, &coeffs[cSet][0], pOut, FIRSZ, BUF);
fir(pln+1-HIST, &coeffs[cSet][0], pOut+1, FIRSZ, BUF);
```

*Note: the driver will be passed the address where new data is to be collected. Only the SWI will be aware of the history segment that precedes it.*
Lab 4: SWI-Based System

This lab begins with a block based version of the audio filtering system. FIR processing is performed on a block (“buffer”) of data instead of being called with each new sample. Block processing is often desirable in multi-threaded systems, since context switching can be reduced to the block rate rather than the (often much) higher sample rate. This allows for more of the DSPs power to be applied to more complex and/or numerous activities.

In this lab, you will:
• begin with the solution from lab 3
• replace audio-3.c with isr.c, which does buffer IO, and proc.c, for the block FIR filtering
• modify the config file to make procBuf a SWI
• build and test the system and consider its range of usability
• modify isr.c and proc.c to circulate buffer ownership via queues (QUE)
• test this version and compare it to the prior version

All the files needed as a starting point for this lab procedure will be found in various directories under C:\BIOS\Labs. If this particular chapter is of lesser interest, or if there is a time constraint, you may elect to skip over the authoring steps and use the solution files stored in directory C:\BIOS\Sols\04b, and proceed directly to seeing how the completed code looks and works.
A. SWI Based procBuf()

In the steps below the HWI-only solution will be upgraded to one employing a SWI which implements block data FIR filtering.

1. **Open CCS** and the solution project from Lab 3. Verify the lab builds and runs properly.

2. **Remove audio-3.c** from the project and from the `Work` directory. It will be replaced by two files which separate the ISR and block data processing functions.

3. **Copy** from `C:\BIOS\Labs\Algos` to `C:\BIOS\Labs\Work, isr.c` and `proc.c`.

4. **Add isr.c and proc.c** to the project. Open `isr.c` and observe how the HWI collects a buffer of data from the serial port and passes it to the SWI once full. Open `proc.c`, and note how the delay line is managed and the offset of --HIST applied to the input buffers.

5. **Create a SWI**: In `myWork.tcf`, add a new SWI object named `swiProcBuf` which calls function `_procBuf`.

6. **Build, load, run, and test** the system. Note the CPU load in debug / release, filter on / off.

7. **Open proc.h in C:\BIOS\Labs\Algos**. Test how different buffer sizes affects the CPU load. Restore the original buffer size of 200 when done testing.

8. If desired, save the contents of the `Work` folder to a new folder, eg: `C:\BIOS\mySols\04a`.

```c
void isrAudio(void) {
    static short bkCnt = 0;
    static short *pInBuf, *pOutBuf;
    static Bool N = 0;
    if (bkCnt == 0) {
      pInBuf = &in[N][HIST];
      pOutBuf = &out[N][0];
      dataIn = MCBSP1_DRR_32BIT;
      pInBuf[bkCnt] = (short)dataIn;
      dataOut = 0x0000FFFF & (pOutBuf[bkCnt]*1<<16);
      bkCnt = (bkCnt+2) & 0xF;
      if (bkCnt == 2) {
        dataOut &= 0xFFFF0000;
        dataOut = 0x0000FFFF & pOutBuf[bkCnt];
        bkCnt = 0;
      }
    }
    else { // if bkCnt % 2 == 1
      pIn = &in[N][HIST];
      pOut = &out[N][0];
      SWI_post(&swiProcBuf);
      bkCnt = 0;
    }
}

void procBuf() {
    short i;
    static short *pPriorIn=&in[1][0];
    for (i = 0; i < HIST; i++)
      pPriorIn[i] = &in[1][0];
    pPriorIn = &in[1][0];
    if (sw0 == 1) {
      firHIST, &coeffs[sw1][0], pOut, FIRSZ, BUF;
      firHIST, &coeffs[sw1][0], pOut, FIRSZ, BUF;
    } else {
      for (i = 0; i < 2*BUF; i++)
        pOut[i] = pIn[i];
    }
}
```
B. Passing Buffers Via QUEues

1. Create two queues: In the config tool, under synchronization, add 2 QUE objects – qDev and qProc

2. Create two messages for the queues: In proc.h note the typedef of the message that will be passed by the queues. In proc.c, declare as globals, an array of two MsgObj’s called bufPtrs

3. Initialize the messages with the addresses of the two in and out buffers: In the main() function, initialize the first bufPtr message with the addresses of input buffer 0, offset HIST and the base address of output buffer 0. (Hint: the line of C code required to load an element of the message structure would be bufPtrs[0].pInBuf=&in[0][HIST]; ). Repeat the process with the second message referencing buffer set 1. Note: the input buffers were offset by HIST so that the HWI would only fill the new data portion of the array, and not consider the preceding area reserved for procBuf to use for the requisite history buffer

4. Prime the queue to the HWI with the two buffer pointer messages: Once the messages have been initialized, place them into qDev for the HWI to acquire when it runs

5. Add a pointer to receive messages from the queue: In isr.c, create a static local variable intBufs of type Msg

6. Use the queue message to determine which buffer set to use in the ISR: Modify the if bkCnt=0 activity to begin by getting a message from the qDev queue. Then replace the loads of pInBuf and pOutBuf with the pointer information received from the queue. (Hint: the line of C code required to load the pointer from the message pointer would be pln = procBufs->pInBuf; )

7. Give the first set of buffers to the processing thread via the 2nd queue: Modify the if bkCnt >= 2*BUF activity to replace the loads of the in and out pointers with a write to the qProc of the buffers now ready for processing prepared by the HWI

8. Clean up prior references to the ‘ping/pong’ operator: Remove all references to the variable N no longer required by the HWI

9. Add a pointer to receive messages from the queue: In procBuf, create a local variable procBufs of type Msg

10. Use the queue message to determine which buffer set to use in the process: Begin procBuf with a read of the message from qProc and initialize pln and pOut with the information received there

11. When finished, give back the buffer ownership: At the end of the procBuf function, return the pointers back to the HWI via qDev

12. Try out the new solution: Build, download and run the code. Verify the performance

13. Save the contents of C:\BIOS\Labs\Work to C:\BIOS\mySols\Lab4b

14. Optional: time permitting, expand the solution to a 3 buffer system – something not readily possible with the pre-queue solution in part A of this lab
C. (Optional) Interrupt Keyword / Dispatcher Conflict

Optional (but valuable) : put back the interrupt keyword, rebuild and run again. What happens? Why? Remove the dispatcher check and try again. Note that in these configurations, the system fails, and it is very hard to figure out why using normal diagnostic efforts. What went wrong?

The use of the interrupt keyword allows the lowest overhead in launching an ISR, however this efficiency is had by not telling the BIOS scheduler about the HWI being run. This works fine as long as the HWI does not enable any other threads to run - such as we did with the $SWI\_post()$ operation. Since BIOS received a $SWI\_post()$, and didn't know of the HWI running at that moment, it launches the SWI, and destabilizes the system, leading to the mysterious failure observed.

In the second case, where both the interrupt keyword and the dispatcher are used, a different problem occurs - two sets of context switches are invoked, but only one can be properly implemented, leaving an that is very hard to observe without painstakingly watching the system run line-by-line in assembly. All these headaches can be avoided, however, by following one of two simple disciplines. The first is foolproof: never use the interrupt keyword and always use the dispatcher. This method cost extra cycles of HWI context switch, but will always work as expected, and never requires further thought on the subject. The second method allows for the extra context cycles to be avoided when desired, but requires the user to be more cautious in authoring a system. Use interrupt keywords on HWIs that do not use BIOS APIs. On an HWI that uses a BIOS API, remove the interrupt keyword and enable the dispatcher for this HWI. Both methods work fine - as long as care is taken. Finally, if you are confronted with a system that fails mysteriously, remember to check for this potential problem early on in order to potentially save a lot of time. Another option to consider is to use only the dispatcher during development, and only switch to the use of the interrupt keyword during final system optimization, where appropriate per the above rules.