CS/ECE 6780/5780

Al Davis

Today's topics:

- Threads
  - restart code from last lecture
  - move on to scheduling & semaphores
- Midterm (next Tues) covers Chaps & Labs 1-5
  - sample on the web

Interrupts on the 6812

- ISR's
  - declare ISR's and the vector that they're associated
    
    ```c
    void interrupt n IsrFcn () { ...code...}
    ```

  - Interrupt n is mapped to a particular branch PC
    - maps n to IsrFcn
    - compiler reference manual pg. 538-539 shows what happens

<table>
<thead>
<tr>
<th>Vector Number</th>
<th>Vector Address</th>
<th>Vector Address Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0xFFFF, 0xFFFF</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0xFFFF, 0xFFFFD</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>0xFFFFA, 0xFFFF</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>n</td>
<td>0xFFFF, (n*2)</td>
<td>2</td>
</tr>
</tbody>
</table>
Mapping HW Events to Interrupt #'s

- In more complex microcontrollers
  - this mapping can be set up in software
- For the 6812
  - the map is fixed
    » see MC9S12C128V reference manual section 1.6.1 pg. 61-62

First 18 Interrupts

<table>
<thead>
<tr>
<th>Int #</th>
<th>Interrupt Source</th>
<th>CCR Mask</th>
<th>Local Enable</th>
<th>HPRO Value to Elevate</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFF, 0xFFF</td>
<td>External reset, power on reset, or low voltage reset (see CRG flags register to determine reset source)</td>
<td>None</td>
<td>None</td>
<td>—</td>
</tr>
<tr>
<td>0xFFF0, 0xFFF1</td>
<td>Clock monitor fail reset</td>
<td>None</td>
<td>CCRCTL (CME, FOME)</td>
<td>—</td>
</tr>
<tr>
<td>0xFFF2, 0xFFF3</td>
<td>COP failure reset</td>
<td>None</td>
<td>COP rate select</td>
<td>—</td>
</tr>
<tr>
<td>0xFFF4, 0xFFF5</td>
<td>Unimplemented instruction trap</td>
<td>None</td>
<td>None</td>
<td>—</td>
</tr>
<tr>
<td>0xFFF6, 0xFFF7</td>
<td>SWI</td>
<td>None</td>
<td>None</td>
<td>—</td>
</tr>
<tr>
<td>0xFFF8, 0xFFF9</td>
<td>XIRQ</td>
<td>X-Bit</td>
<td>None</td>
<td>—</td>
</tr>
<tr>
<td>0xFFFF2, 0xFFFF3</td>
<td>IRQ</td>
<td>1-bit</td>
<td>INTCR (IPEN)</td>
<td>0x00F2</td>
</tr>
<tr>
<td>0xFFFF4, 0xFFFF5</td>
<td>Real time interrupt</td>
<td>1-bit</td>
<td>CRGINT (RTIE)</td>
<td>0x00F0</td>
</tr>
<tr>
<td>0xFFFF6, 0xFFFF7</td>
<td>Standard timer channel 0</td>
<td>1-bit</td>
<td>TIE (C0)</td>
<td>0x00EE</td>
</tr>
<tr>
<td>0xFFFF8, 0xFFFF9</td>
<td>Standard timer channel 1</td>
<td>1-bit</td>
<td>TIE (C1)</td>
<td>0x00EC</td>
</tr>
<tr>
<td>0xFFFFA, 0xFFFFB</td>
<td>Standard timer channel 2</td>
<td>1-bit</td>
<td>TIE (C2)</td>
<td>0x00EA</td>
</tr>
<tr>
<td>0xFFFFC, 0xFFFFD</td>
<td>Standard timer channel 3</td>
<td>1-bit</td>
<td>TIE (C3)</td>
<td>0x00E8</td>
</tr>
<tr>
<td>0xFFFE</td>
<td>Standard timer channel 4</td>
<td>1-bit</td>
<td>TIE (C4)</td>
<td>0x00E6</td>
</tr>
<tr>
<td>0xFFFFF</td>
<td>Standard timer channel 5</td>
<td>1-bit</td>
<td>TIE (C5)</td>
<td>0x00E4</td>
</tr>
<tr>
<td>0xFFE2, 0xFFE3</td>
<td>Standard timer channel 6</td>
<td>1-bit</td>
<td>TIE (C6)</td>
<td>0x00E2</td>
</tr>
<tr>
<td>0xFFE4, 0xFFE5</td>
<td>Standard timer channel 7</td>
<td>1-bit</td>
<td>TIE (C7)</td>
<td>0x00C0</td>
</tr>
</tbody>
</table>
Remaining Interrupts

<table>
<thead>
<tr>
<th>Vector Address</th>
<th>Interrupt Source</th>
<th>Code</th>
<th>Local Enable</th>
<th>NPRO Value in Diverts</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,0000,0000</td>
<td>Standard timer overflow</td>
<td>113</td>
<td></td>
<td>0x000E</td>
</tr>
<tr>
<td>0,0000,0001</td>
<td>Pulse accumulator A overflow</td>
<td>113</td>
<td></td>
<td>0x000C</td>
</tr>
<tr>
<td>0,0000,0002</td>
<td>Pulse accumulator B overflow</td>
<td>113</td>
<td></td>
<td>0x002A</td>
</tr>
<tr>
<td>0,0000,0097</td>
<td>SPI (SPI, SPIE)</td>
<td>113</td>
<td></td>
<td>0x000D</td>
</tr>
<tr>
<td>0,0000,0098</td>
<td>8259</td>
<td>113</td>
<td></td>
<td>0x000E</td>
</tr>
<tr>
<td>0,0000,0099</td>
<td>8259</td>
<td>113</td>
<td></td>
<td>0x0012, 13, 16, 18, 19</td>
</tr>
<tr>
<td>0,0000,0100</td>
<td>ATD</td>
<td>113</td>
<td></td>
<td>0x002C</td>
</tr>
<tr>
<td>0,0000,0101</td>
<td>Port 2</td>
<td>113</td>
<td></td>
<td>0x002D</td>
</tr>
<tr>
<td>0,0000,0102</td>
<td>Port 2</td>
<td>113</td>
<td></td>
<td>0x002E</td>
</tr>
<tr>
<td>0,0000,0103</td>
<td>Port 5</td>
<td>113</td>
<td></td>
<td>0x002F</td>
</tr>
<tr>
<td>0,0000,0104</td>
<td>Port 5</td>
<td>113</td>
<td></td>
<td>0x0030</td>
</tr>
<tr>
<td>0,0000,0105</td>
<td>Port 5</td>
<td>113</td>
<td></td>
<td>0x0031</td>
</tr>
<tr>
<td>0,0000,0106</td>
<td>Port 5</td>
<td>113</td>
<td></td>
<td>0x0032</td>
</tr>
<tr>
<td>0,0000,0107</td>
<td>Port 5</td>
<td>113</td>
<td></td>
<td>0x0033</td>
</tr>
<tr>
<td>0,0000,0108</td>
<td>Port 5</td>
<td>113</td>
<td></td>
<td>0x0034</td>
</tr>
<tr>
<td>0,0000,0109</td>
<td>Port 5</td>
<td>113</td>
<td></td>
<td>0x0035</td>
</tr>
<tr>
<td>0,0000,0110</td>
<td>Port 5</td>
<td>113</td>
<td></td>
<td>0x0036</td>
</tr>
<tr>
<td>0,0000,0111</td>
<td>Port 5</td>
<td>113</td>
<td></td>
<td>0x0037</td>
</tr>
</tbody>
</table>

not needed for Lab5 or this lecture

Thread Code

• Admittedly somewhat silly & review from last lecture

```c
int Sub(int j) { int i;
    PTM = 1; // Port M
    i = j+1;
    return(i); }

void ProgA() { int i;
    i=5;
    while(1) {
        PTM = 2;
        i = Sub(i); }}

void ProgB() { int i;
    i=6;
    while(1) {
        PTM = 4;
        i = Sub(i); }}
```

PTM assignment used to provide external visibility of the running thread

Use of one-hot code on PortM pins is just a random choice

Key is that both ProgA & ProgB threads run forever

Hence preemptive scheduler is needed
### Setting up the TCB

```c
struct TCB
{
    struct TCB *Next;    /* Link to Next TCB */
    unsigned char *SP;   /* Stack Pointer when idle */
    unsigned short Id;   /* output to PortT */
    unsigned char MoreStack[49]; /* more stack */
    unsigned char CCR;   /* Initial CCR */
    unsigned char RegB;  /* Initial RegB */
    unsigned char RegA;  /* Initial RegA */
    unsigned short RegX; /* Initial RegX */
    unsigned short RegY; /* Initial RegY */
    void (*PC)(void);    /* Initial PC */
};
typedef struct TCB TCBType;
typedef TCBType *TCBPtr;
```

see anything fishy so far?

### Port M vs. Port T

- **Essential difference between program & thread**
  - **program is just the code**
    - note that code has no state
    - it's just a specification of what will happen if it is executed
  - **thread is an execution instance**
    - inherently has state
      - in this case initial state can be seen in the code
      - subsequent state will depend
        - TCB values if the thread isn't running
        - TCB values and registers if the thread is running

- **In this simple example**
  - Port M is used to show which Program is being executed
  - Port T is used to show which Thread is being executed
  - in this case
    - M will be the same for threads 1 & 2
  - in general
    - a thread could run more than 1 program in different thread phases
Defining 3 Threads

Thread n = sys[n]

threads 1 & 2 are the same code but work on different local data

CCR = 0x40
XIRQ disabled
IRQ enabled

Note all TCB variables values here influence only what happens the FIRST time the thread is executed

Why will these variables need to be changed for subsequent executions?

Preemptive Thread Scheduler in C

TCBPtr RunPt; /* Pointer to current thread */

void main(void) {
    DHR = 0xPF; /* Output running thread on Port T */
    DORM = 0xPF; /* Output running program on Port M */
    RunPt = &ys[0]; /* Specify first thread */
    asm sei
    TFLG1 = 0x20; /* Clear G5F */
    TIE = 0x20; /* Arm G5F */
    TSCR1 = 0x80; /* Enable TCNT+ */
    TSCR2 = 0x01; /* 2MHz TCNT */
    TIOS = 0x20; /* Output compare */
    TC5 = TCNT+20000;
    PTF = RunPt->Id;
    asm ldx RunPt
    asm lda 2,x
    asm cli
    asm rti
    } /* Launch First Thread */
Preemptive Thread Switch

```c
void interrupt 13 ThreadSwitch() {
    asm ldx RunPt
    asm sts 2,x
    RunPt = RunPt->Next;
    PTT = RunPt->Id; /* PortH=active thread */
    asm 1dx RunPt
    asm 1ds 2,x
    TC5 = TCNT+20000; /* Thread runs for 10 ms */
    TFLG1 = 0x20; } /* ack by clearing C5F */
```

see any mistakes?
what does “interrupt 13” mean?

Dynamic Thread Allocator

```c
int create(void (*startFunc)(void), int TheId) {
    TCBPtr NewPt; // pointer to new thread control block
    NewPt = (TCBPtr)malloc(sizeof(TCBType)); // new TCB
    if(NewPt==0)return FAIL;
    NewPt->SP = &NewPt->OCR; /* Stack Pointer when not running */
    NewPt->Id = TheId; /* Visualize active thread */
    NewPt->CCR = 0x00; /* Initial CCR, I=0 */
    NewPt->RegB = 0; /* Initial RegB */
    NewPt->RegA = 0; /* Initial RegA */
    NewPt->RegX = 0; /* Initial RegX */
    NewPt->RegY = 0; /* Initial RegY */
    NewPt->PC = startFunc; /* Initial PC */
    if(RunPt) {
        NewPt->Next = RunPt->Next;
        RunPt->Next = NewPt; } /* will run Next */
    else
        RunPt = NewPt; /* the first and only thread */
    return SUCCESS;
}
```
Concluding Remarks

• Implementation of a very simple thread system
  • e.g. round robin preemptive
  • it’s not that hard
    » but note the tricks for setting the PC to the appropriate thread
    code start
• Preemptive scheduling
  • lies at the heart of an RTOS
    » but in this case we didn’t consider real time issues
      • making things significantly easier
• The hard part
  • designing correct embedded codes that use threads

• Note
  • this code shows the general idea
  • there are parts missing that will need to be coded for a full
    solution – future lab?

So Far

• We’ve talked about
  • thread scheduling
  • synchronization
    » between main & ISR’s
• Next
  • synchronization
    » between threads
  • table driven scheduling
2 Threads & a communication FIFO

```c
int Fifo_Put(char data)
{
    char *Ppt;
    // BEGIN CRITICAL
    Ppt=PutPt;
    *(Ppt++)=data;
    if (Ppt == &Fifo[FIFOSIZE]) Ppt = &Fifo[0];
    if (Ppt == GetPt ) {
        // END CRITICAL
        return(0);
    } else {
        PutPt=Ppt;
        // END CRITICAL
        return(1);
    }
}
```

what's missing?

Semaphores

- Used to implement mutual exclusion (MUTEX)
  - useful for sharing, synchronization, & communication
- 2 basic operations
  - classic terminology
    - P \rightarrow wait (Dijkstra's Dutch "probeer te verlagen" ::- try to grab"
    - V \rightarrow signal ("verhogen" ::= increase)
  - semaphore is binary value
    - 1 \rightarrow free (resource available)
    - 0 \rightarrow busy (resource owned by some other thread)
- Numerous semaphore implementations
  - simplest is a “Spin-Lock” version
    - thread calls wait to wait (spins) for semaphore to be free
      - when semaphore is free, wait sets semaphore to busy
      - and then return
    - critical
      - enable Interrupts during spin or preemption can't happen
      - read modify write on semaphore value must be atomic
        - otherwise an interrupt might switch threads and chaos will result
      - once thread is done it calls signal to return the semaphore to free
2 Common Semaphore Types

- **Binary**
  - simple lock

- **Counter**
  - useful when multiple resources are available
    - say tables in a restaurant
      - >0 there is something available
      - <= 0 the place is busy
      - hence the semantics

- **Note earlier lecture error**
  - **tst instruction is not “Test and Set”**
    - this instruction exists on a lot of machines
      - useful for binary semaphores
  - **6812 TST, TSTA, TSTB → test M,A, or B for 0 or minus**
    - does not change M, A, or B value – just changes the CC flags
    - hence not useful for semaphore implementation

MINA & MINM Instructions

- **MINM**
  - stores minimum value
    - MINM [opndA, opndB] → min(A, B) stored in B
      - CC bits N,Z,V,C set based on A-B
      - opndA can be
        - indexed addressing postbyte code
          - e.g. preincrement similar to “test and set”
        - integer in various ranges
      - opndB can be
        - X, Y, SP
  - useful for semaphore implementation

- **MINA does the same thing**
  - but result goes to register A rather than memory
  - only one REG A however so unlikely choice for semaphore implementation
Remember the Atomicity Functions

```c
unsigned char begin_critical (void)
{
    unsigned char SaveSP;
    asm tpa
    asm staa SaveSP
    asm sei
    return SaveSP;
}

void end_critical (unsigned char SaveSP)
{
    asm ldaa SaveSP
    asm tpa
}
```

Key idea

- begin - save the predicate register in SaveSP
- disable interrupts

on end – restore the predicate register

why no cli to re-enable interrupts?
Spin-Lock Semaphore Wait

// decrement and spin if less than 0
// input: pointer to a semaphore
// output: none
void OS_Wait(short *semaPt) {
    unsigned char SaveSP = begin_critical();
    while(*semaPt <= 0) {
        end_critical (SaveSP);
        asm nop
        SaveSP = begin_critical();
    }
    (*semaPt)--;
    end_critical (SaveSP);
}

key point: semaphore access is in critical section & MUTEX

Spin-Lock Semaphore Signal

// increment semaphore
// input: pointer to a semaphore
// output: none
void OS_Signal(short *semaPt) {
    unsigned char SaveSP = begin_critical();
    (*semaPt)++;
    end_critical (SaveSP);
}
Spin-Lock Binary Semaphore

```c
void bWait(char *semaphore) {
    asm clra
    asm loop: minm [2,x]
    asm bcc loop
}
void bSignal(char *semaphore) {
    (*semaphore) = 1; // compiler makes this atomic
}
```

clra sets new value for the semaphore
minm [2,X] is test and set in ICC version 5

not sure why this works – anybody know?

Counting Semaphore from Binary Semaphores

```c
struct sema4
{
    short value; // semaphore value
    char s1;     // binary semaphore
    char s2;     // binary semaphore
    char s3;     // binary semaphore
};
typedef struct sema4 sema4Type;
typedef sema4Type * sema4Ptr;
void Initialize(sema4Ptr semaphore, short initial) {
    semaphore->s1 = 1; // first one to bWait(s1) continues
    semaphore->s2 = 0; // first one to bWait(s2) spins
    semaphore->s3 = 1; // first one to bWait(s3) continues
    semaphore->value=initial;
}
```
void Wait(sema4Ptr semaphore) {
    bWait(&semaphore->s3); // wait if other caller here first
    bWait(&semaphore->s1); // mutual exclusive access to value
    (semaphore->value)--; // basic function of Wait
    if((semaphore->value)<0) {
        bSignal(&semaphore->s1); // end of exclusive access
        bWait(&semaphore->s2); // wait for value to go above 0
    }
    else
        bSignal(&semaphore->s1); // end of exclusive access
    bSignal(&semaphore->s3); // let other callers in
}

void Signal(sema4Ptr semaphore) {
    bWait(&semaphore->s1); // exclusive access
    (semaphore->value)++; // basic function of Signal
    if((semaphore->value)<=0)
        bSignal(&semaphore->s2); // allow S2 spinner to continue
    bSignal(&semaphore->s1); // end of exclusive access
}
Blocking Semaphore

- Useful when multiple threads are blocked waiting on a resource

Blocking Semaphore Stages

Initialize:
Set the counter to its initial value.
Clear associated blocked tcb linked list.

Wait:
Disable interrupts to make atomic
Decrement the semaphore counter, S=S-1
If semaphore counter < 0, then block this thread.
Restore interrupt status.

Signal:
Disable interrupts to make atomic
Increment the semaphore counter, S=S+1
If counter ≤ 0, wakeup one thread.
Restore interrupt status
Initialize

S  rmb 1 ; semaphore counter
BlockPt rmb 2 ; Pointer to threads blocked on S
Init  
  tpa
  psha ; Save old value of I
  sei ; Make atomic
  ldaa #1
  staa S ; Init semaphore value
  ldx #Null
  stx BlockPt ; empty list
  pula
  tap ; Restore old value of I
  rts

Block a Thread

; To block a thread on semaphore S, execute SWI
SWIhan ldx  RunPt ; running process "to be blocked"
    sts  SP,x ; save Stack Pointer in its TCB
; Unlink "to be blocked" thread from RunPt list
    ldy Next,x ; find previous thread
    sty RunPt ; next one to run
    look cpx Next,y ; search to find previous
    beq found
    ldy Next,y
    bra  look
found  ldd RunPt ; one after blocked
    std Next,y ; link previous to next to run
Block and Launch Next

; Put "to be blocked" thread on block list
ldy BlockPt
sty Next,x  ;link "to be blocked"
stx BlockPt

; Launch next thread
ldx RunPt
lds SP,x    ;set SP for this new thread
ldd TCNT    ;Next thread gets a full 10ms time slice
add #20000 ;interrupt after 10 ms
std TC5
ldaa #$20
staa TFLG1  ;clear C5F
rti

Linked Lists

![Diagram of linked lists with TCB entries and blocking status](image-url)
**Thread Rendezvous**

Synchronize two threads at a *rendezvous* location.

- **S1**  S2  Meaning
- 0   0  Neither thread at rendezvous location
- -1  +1  Thread 2 arrived first, waiting for thread 1
- +1  -1  Thread 1 arrived first, waiting for thread 2

```c
Thread 1
signal(&S1);  signal(&S2);
wait(&S2);    wait(&S1);
Thread 2
```

This only works for 2 threads

How do you make a general *n* thread barrier?

---

**Mutex Sharing or Non-reentrant Code**

Guarantee mutual exclusive access to a critical section.

```c
Thread 1    Thread 2    Thread 3
bwait(&S);  bwait(&S);  bwait(&S);
printf("bye"); printf("tchau"); printf("ciao");
bsignal(&S); bsignal(&S); bsignal(&S);
```
2 Thread Mailbox

Thread 1 sends mail to thread 2.

<table>
<thead>
<tr>
<th>Send</th>
<th>Ack</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>No mail available, consumer not waiting</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td>No mail available, consumer is waiting</td>
</tr>
<tr>
<td>+1</td>
<td>-1</td>
<td>Mail available and producer is waiting</td>
</tr>
</tbody>
</table>

Producer thread
Mail=4;
wait(&send);
signal(&send);
wait(&ack);

Consumer thread
read(Mail);
signal(&ack);

Bounded FIFO

• Multiple consumer and producer threads

PutFifo
wait(&RoomLeft);
wait(&mutex);
put data in FIFO
signal(&mutex);
signal(&CurrentSize);

GetFifo
wait(&CurrentSize);
wait(&mutex);
remove data from FIFO
signal(&mutex);
signal(&RoomLeft);

Could disable interrupts instead of using mutex, but would lock out threads that don’t affect the FIFO.
Concluding Remarks

• **Threads introduce concurrency**
  - decoupling makes thinking about complex tasks easier
  - there is a cost however
    - scheduling is required
    - shared resources requires additional control
      - semaphores are the control mechanism
      - but access must be mutually exclusive

• **Reminder**
  - midterm Tuesday
  - don't be late