Lecture 5: Debugging, FSMs Reprise, & Interfacing Methods
The debugging process is defined as testing, stabilizing, localizing, and correcting errors.

Research in program monitoring and debugging has not kept pace with developments in other areas of software.

In embedded systems, debugging is further complicated by concurrency and real-time requirements.

Although monitoring and debugging tools exist, many still use manual methods such as print statements.

Print statements are highly intrusive especially in a real-time system because they can take too much time.
Debugging with Hardware

![Diagram showing a 6811 processor, Logic analyzer, Memory, and address/data bus connections.](image)

<table>
<thead>
<tr>
<th>R/W</th>
<th>Address</th>
<th>Data</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>8000</td>
<td>B6</td>
<td>ldaa $1003</td>
</tr>
<tr>
<td>R</td>
<td>8001</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>8002</td>
<td>03</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>1003</td>
<td>55</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>8003</td>
<td>B7</td>
<td>staa $1004</td>
</tr>
<tr>
<td>R</td>
<td>8004</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>8005</td>
<td>04</td>
<td></td>
</tr>
<tr>
<td>W</td>
<td>1004</td>
<td>55</td>
<td></td>
</tr>
</tbody>
</table>

**Registers**
- A = $55
- B = $31
- X = $1234
- Y = $5678
- SP = $0BF0
- PC = $F103

**I/O Ports**
- PortH = $93
- PortJ = $00
- PortS = $55
- PortT = $0F
- PortR = $21
- TCNT = $A010

**Address contents interpretation**

<table>
<thead>
<tr>
<th>Address</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E000</td>
<td>$B6 ldaa $1003</td>
</tr>
<tr>
<td>$E001</td>
<td>$10</td>
</tr>
<tr>
<td>$E002</td>
<td>$03</td>
</tr>
<tr>
<td>$E003</td>
<td>$B7</td>
</tr>
<tr>
<td>$E004</td>
<td>$10</td>
</tr>
<tr>
<td>$E005</td>
<td>$04</td>
</tr>
<tr>
<td>$E000</td>
<td>$10 staa $1004</td>
</tr>
</tbody>
</table>

**Diagram showing an embedded system with microcomputer and I/O, 6811 processor, ROM socket, and RAM connections.**
A \textit{debugging instrument} is code that is added to a program for the purpose of debugging.

A print statement is a common example.

When adding print statements, use one of the following:

- Place all print statements in a unique column.
- Define instruments with specific pattern in their name.
- Define all instruments to test a run-time global flag.
- Use conditional compilation (assembly) to turn on/off.
Functional debugging means: checking if the code implements the correct computation
Inputs are supplied, system is run, outputs are checked.
There exist many functional debugging methods:

Single stepping or tracing.
Breakpoints without filtering.
Conditional breakpoints.
Instrumentation: print statements.
Instrumentation: dump into array without filtering.
Instrumentation: dump into array with filtering.
Monitor using fast displays.
// global variables in RAM
#define size 20
unsigned char buffer[size][2];
unsigned int cnt=0;
// dump happy and sad
void Save(void){
   if(cnt<size){
      buffer[cnt][0] = happy;
      buffer[cnt][1] = sad;
      cnt++;
   }
}

// dump happy and sad
void Save(void){
    if(sad>100){
        if(cnt<size){
            buffer[cnt][0] = happy;
            buffer[cnt][1] = sad;
            cnt++;
        }
    }
}
**Performance debugging** is debugging of timing behavior. System is run and dynamic behaviors of I/O checked.

Count bus cycles using the assembly listing.
Instrumentation: measuring with a counter.

```c
unsigned short before, elasped;
void main(void) {
    ss=100;
    before=TCNT;
    tt=sqrt(ss);
    elasped=TCNT-before;
}
```
Set bset PORTB,#$40
t
Clr bclr PORTB,#$40
t
loop jsr Set
jsr Calculate ; function under test
jsr Clr
bra loop
Performance (Dynamic) Debugging

; Assembly listing from TExaS of the sqrt subroutine.

$F019  org * ;reset cycle counter
$F019  35 [ 2]( 0) sqrt pshy
$F01A  B776 [ 1]( 2) tsy
$F01C  1B9C [ 2]( 3) leas -4,sp ;allocate t,oldt,s16
$F01E  C7 [ 1]( 5) clrb
$F01F  A644 [ 3]( 6) ldaa s8,y
$F021  2723 [ 3]( 9) beq done
$F023  C610 [ 1]( 12) ldab #16
$F025  12 [ 3]( 13) mul ;16*s
$F026  6C5C [ 2]( 16) std s16,y ;s16=16*s
$F028  18085F20 [ 4]( 18) movb #32,t,y ;t=2.0, initial guess
$F02C  18085E03 [ 4]( 22) movb #3,cnt,y
$F030  A65F [ 3]( 26) next ldaa t,y ;RegA=t
$F032  180E [ 2]( 29) tab ;RegB=t
$F034  B705 [ 1]( 31) tfr a,x ;RegX=t
$F036  12 [ 3]( 32) mul ;RegD=t*t
$F037  E35C [ 3]( 35) addd s16,y ;RegD=t*t+16*s
$F039  1810 [12]( 38) idiv ;RegX=(t*t+16*s)/t
$F03B  B754 [ 1]( 50) tfr x,d
$F03D  49 [ 1]( 51) lsrd ;RegB=((t*t+16*s)/t)/2
$F03E  C900 [ 1]( 52) adcb #0
$F040  6B5F [ 2]( 53) stab t,y
$F042  635E [ 3]( 55) dec cnt,y
$F044  26EA [ 3]( 58) bne next
$F046  B767 [ 1]( 61) done tys
$F048  31 [ 3]( 62) puly
$F049  3D [ 5]( 65) rts
$F04A  183E [16]( 70) stop
Testing

Exercise the capabilities of a program by stimulating it with a wide range of input values.
Test:

Normal conditions.
Special cases and boundary conditions.
Execution of every branch.

Any peculiar performance or result should be investigated.

Stabilization

Predictably reproduce the bug.
Most bugs are easily stabilized.
Memory, concurrency, and timing related bugs may be more difficult to stabilize.
Localization & Correction

Localization

Isolate the a specific variable or code segment.

Methods:

Create an experiment to alternately hide/show the bug.
Single-step through suspect code.
Collect and examine program traces.

Correction

Eradicate the bug.
Usually this is the easy step.
Bugs involving design flaws may be more complicated to correct.
Integration bugs:

- Function return-value type mismatch bugs.
- Problems with global variables may surface during integration.
- Interrupt related bugs (improperly saved code, non-reentrant code).
- Shared resource bugs.

Machine-specific tricks become bugs when porting code.
FSM code: A more traditional approach

```c
#define GO_N 110
#define WAIT_N 111
#define GO_E 112
#define WAIT_E 113
#define NO_CARS 0x00 // 0
#define CAR_E 0x01 // 1
#define CAR_N 0x02 // 2
#define CAR_N_E 0x03 // 3
#define OUT_GO_N 0x21 // 33
#define OUT_WAIT_N 0x22 // 34
#define OUT_GO_E 0x0C // 12
#define OUT_WAIT_E 0x14 // 20

void main() {
    int state;
    unsigned char inPTB;
    Timer_Init();
    DDRB = 0xFF;
    DDRA &= ~0x03;
    state = GO_N;
}```
while(1) {
    switch(state) {
    case GO_N:
        PORTB = OUT_GO_N;
        Timer_Wait_1s(30);
        inPTA = PORTA&0x03;
        if(inPTB == CAR_E || inPTB == CAR_N_E) {
            state = WAIT_N;
        }        
        break;
    case WAIT_N:
        PORTB = OUT_WAIT_N;
        Timer_Wait_1s(5);
        inPTA = PORTA&0x03;
        state = GO_E;
        break;
    ...

    Pros and cons?
Embedded systems often have many special I/O devices, so I/O interfacing is a critical task. I/O interfacing includes both physical connections and software routines that affect information exchange. Chapter 3 introduces basic interfacing methods. Chapter 4 introduces interfacing using interrupts.
Latency is the delay from when an I/O device needs service until the service is initiated. It includes both hardware and software delays. Real-time systems guarantee a worst-case latency. Throughput or bandwidth is maximum rate data can be processed. Can be limited by I/O device or the software. Priority determines order of service when two or more requests are made at the same time.
Key problem: I/O devices operate in parallel with the CPU
  This is good: parallelism is efficient
  This is bad: parallelism is difficult for humans
Common case: Hardware is in 1 of 3 states: idle, busy, or done.
  When working, device alternates between busy and done.
I/O devices usually much slower than software, so synchronization is required for proper operation.
When an I/O device is slower than software, it is I/O bound, otherwise it is CPU bound.
Interface can be buffered or unbuffered.
There is very wide variation in the amount of work that the HW can do without babysitting from the SW
  We start out looking at interfaces that require maximal babysitting
Synchronizing Software with an Input Device

- **flag=0**
  - Waiting for input
  - Service provided
  - Software reads data, asks for another

- **flag=1**
  - New input is ready
  - Input device creates new data
  - Causes gadfly loop to complete

---

**Input device**

- Busy
- Done
- Busy
- Done

**Software**

- Waiting for new input
- Read data
- Process data
- Waiting
- Read data

---

Time
Synchronizing Software with an Output Device

flag=0
Busy performing last output

Service provided
Software writes new data, asks device to output it

flag=1
Output is done

Output device completes output operation
Causes gadfly loop to complete

Output device
<table>
<thead>
<tr>
<th>Done</th>
<th>Busy</th>
<th>Done</th>
<th>Busy</th>
<th>Done</th>
</tr>
</thead>
</table>

Software
| Generate | Write | Generate | Waiting | Write | Generate | Waiting |

Time
Synchronization Mechanisms

*Blind cycle* - software waits a fixed amount of time for the I/O to complete.

*Gadfly or busy waiting* - software loops checking the I/O status waiting for the done state.

*Interrupt* - I/O device causes software to execute upon request.

*Periodic polling* - periodic interrupts check the I/O status.

*Direct memory access* - I/O device directly transfers data to/from memory.
Blind Cycle Printer Interface

MC9S12C32
PM0
PT7-0

Printer
GO
DATA

GO
10 ms
DATA required
void Init(void){
    DDRT = 0xFF;   // outputs
    DDRM|= 0x01;
    PTM |= 1;     // GO=1
    Timer_Init();}
void Out(unsigned char value){
    PTT = value;
    PTM&=~0x01;   // GO=0
    PTM|=0x01;    // GO=1
    Timer_MsWait(10);    // 10ms
Blind Cycle ADC Interface

MC9S12C32
PM0
PT7-0

ADC
GO
DATA
In

Analog signal

GO

DATA available

5 μs
void Init(void){
    DDRT = 0x00; // input DATA
    DDRM|= 0x01;
    PTM &=~0x01; // GO=0
    Timer_Init();}

unsigned char In(void){
    PTM |= 0x01; // GO=1
    PTM &=~0x01; // GO=0
    Timer_UsWait(5);
    return(PTT);}
Blind Cycle Evaluation

Pros

Simple
Predictable

Cons

Inflexible
Inefficient if the delay is long
If done wrong, code breaks when speed of CPU changes

Works well for simple, high-speed devices.
In fact, nearly 100% of embedded systems contain a few examples of blind cycle interfacing
Gadfly or Busy Waiting Synchronization

Input

No

Ready?

Yes

Read (input) data

Return

Output

No

Ready?

Yes

Write data begin output process

Return

Output

Write data begin output process

No

Ready?

Yes

Return
Multiple Gadfly Outputs

Gadfly before output

No

Ready 1?

Yes

Write data1
begin output1 process

No

Ready 2?

Yes

Write data2
begin output2 process

No

Ready 3?

Yes

Write data3
begin output3 process

Gadfly after output

Write data1
begin output1 process

No

Ready 1?

Yes

Write data2
begin output2 process

No

Ready 2?

Yes

Write data3
begin output3 process

No

Ready 3?

Yes
Multiple Gadfly Inputs and Outputs
Gadfly Evaluation

Pros
Simple
A bit more flexible than blind cycle

Cons
Inefficient if devices are not fast
Potentially unpredictable

Use with caution
But again, almost 100% of embedded systems will contain some examples of busy-waiting on devices
Gadfly Keyboard Interface Using Latched Input
void Init(void){ // PJ7=STROBE
    DDRJ = 0x00;    // PT6-0 DATA
    DDRT = 0x80;    // PT7 unused output
    PPSJ = 0x80;    // rise on PJ7
    PIFJ= 0x80;}    // clear flag7

unsigned char In(void){
    while((PIFJ&0x80)==0); // wait
    PIFJ = 0x80;    // clear flag7
    return(PTT);
}
void Init(void){ // PJ7=DONE in
    DDRJ = 0x40;  // PJ6=GO out
    PPSJ = 0x80;  // rise on PJ7
    DDRT = 0x00;  // PT7-0 DATA in
    PTJ &=~0x40; } // GO=0

unsigned char In(void){
    PIFJ = 0x80;  // clear flag7
    PTJ |= 0x40;  // GO=1
    PTJ &=~0x40; // GO=0
    while((PIFJ&0x80)==0);
    return(PTT);}
Gadfly External Sensor Interface Using Input Handshake

[Diagram showing the connection between MC9S12C32, PJ7, PJ6, PT7-0, READY, ACK, DATA, and Sensor.]
void Init(void){
    // PJ7=READY in
    DDRJ = 0x40;  // PJ6=ACK out
    PPSJ = 0x80;  // rise on PJ7
    DDRT = 0x00;  // PT7-0 DATA in
    PIFJ = 0x80;  // clear flag7
    PTJ |= 0x40;} // ACK=1
unsigned char In(void){
    unsigned char data;
    while(((PIFJ&0x80)==0));
    PTJ &=~0x40;  // ACK=0
    data = PTT;  // read data
    PIFJ = 0x80;  // clear flag7
    PTJ |=0x40;  // ACK=1
    return(data);}
Gadfly Printer Interface Using Output Handshake

- **MC9S12C32**
- **PJ7**
- **PJ6**
- **PT6-0**

**READY**

**START**

**DATA required**

- Set up = 100 ns
- Hold = 20 ns

**READY**

**START**

Gadfly loop

- 10 ms
void Init(void){// PJ7=READY in
    DDRJ = 0x40; // PJ6=START out
    PPSJ = 0x80; // rise on PJ7
    DDRT = 0xFF; // PT7-0 DATA out
    PTJ |= 0x40;} // START=1
void Out(unsigned char data){
    PIFJ = 0x80; // clear flag
    PTJ &=~0x40; // START=0
    PTT = data; // write data
    PTJ |= 0x40; // START=1
    while((PIFJ&0x80)==0);}
Range: -55 to 125°C with a resolution of 0.5°C.
Data is encoded using a 9-bit 2’s complement number.
Basis elements for the number are: -128, 64, 32, 16, 8, 4, 2, 1, 0.5.

<table>
<thead>
<tr>
<th>Temperature</th>
<th>Binary Value</th>
<th>Hex Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>+125.0°</td>
<td>011111010</td>
<td>$0FA</td>
</tr>
<tr>
<td>+64.0°</td>
<td>010000000</td>
<td>$080</td>
</tr>
<tr>
<td>0.5°</td>
<td>000000001</td>
<td>$001</td>
</tr>
<tr>
<td>0°</td>
<td>000000000</td>
<td>$000</td>
</tr>
<tr>
<td>-0.5°</td>
<td>111111111</td>
<td>$1FF</td>
</tr>
<tr>
<td>-16.0°</td>
<td>111100000</td>
<td>$1E0</td>
</tr>
<tr>
<td>-55.0°</td>
<td>110010010</td>
<td>$192</td>
</tr>
</tbody>
</table>
Initialization of the DS1620

```c
void DS_Init(void){ // PT7=RST=0
    DDRT = 0xE0; // PT6=CLK=1
    PTT = 0x60;} // PT5=DQ=1
```
void out8(char code){ int n;
    for(n=0;n<8;n++){
        PTT &= 0xBF; // PT6=CLK=0
        if(code&0x01)
            PTT |= 0x20; // PT5=DQ=1
        else
            PTT &= 0xDF; // PT5=DQ=0
        PTT |= 0x40; // PT6=CLK=1
        code = code>>1;}}
void DS_Start(void){
    PTT |= 0x80;  // PT7=RST=1
    out8(0xEE);
    PTT &= 0x7F;  // PT7=RST=0
}

void DS_Stop(void){
    PTT |= 0x80;  // PT7=RST=1
    out8(0x22);
    PTT &= 0x7F;  // PT7=RST=0
}
void DS_Config(char data){
    PTT |= 0x80;  // PT7=RST=1
    out8(0x0C);
    out8(data);
    PTT &= 0x7F;  // PT7=RST=0
void out9(short code){  short n;
    for(n=0;n<9;n++){
        PTT &= 0xBF;  // PT6=CLK=0
        if(code&0x01)
            PTT |= 0x20;  // PT5=DQ=1
        else
            PTT &= 0xDF;  // PT5=DQ=0
        PTT |= 0x40;  // PT6=CLK=1
        code = code>>1;}}
Set Threshold Registers on the DS1620

```c
void DS_WriteTH(short data){
    PTT |= 0x80; // PT7=RST=1
    out8(0x01);
    out9(data);
    PTT &= 0x7F; // PT7=RST=0
}

void DS_WriteTL(short data){
    PTT |= 0x80; // PT7=RST=1
    out8(0x02);
    out9(data);
    PTT &= 0x7F; // PT7=RST=0
}
```
Read 8-bits from the DS1620

```c
unsigned char in8(void)
{
    short n;
    unsigned char result;
    DDRT &= 0xDF; // PT5=DQ input
    for(n=0;n<8;n++){
        PTT &= 0xBF; // PT6=CLK=0
        result = result>>1;
        if(PTT&0x20)
            result |= 0x80; // PT5=DQ=1
        PTT |= 0x40; // PT6=CLK=1
    }
    DDRT |= 0x20; // PT5=DQ output
    return result;
}
```
unsigned char DS_ReadConfig(void) {
    unsigned char value;
    PTT |= 0x80;  // PT7=RST=1
    out8(0xAC);
    value = in8();
    PTT &= 0x7F;  // PT7=RST=0
    return value;
}
unsigned short in9(void){ short n;
unsigned short result=0;
    DDRT &= 0xDF;  // PT5=DQ input
    for(n=0;n<9;n++){
        PTT &= 0xBF;  // PT6=CLK=0
        result = result>>1;
        if(PTT&0x20)
            result |= 0x0100; // PT5=DQ=1
        PTT |= 0x40;  // PT6=CLK=1
    }  // PT6=CLK=1
DDRT |= 0x20;  // PT5=DQ output
return result;}
unsigned short DS_ReadTH(void){
  unsigned short value;
  PTT |= 0x80;  // PT7=RST=1
  out8(0xA1);
  value = in9();
  PTT &= 0x7F;  // PT7=RST=0
  return value;}

Read the Temperatures from the DS1620
unsigned short DS_ReadTL(void){
    unsigned short value;
    PTT |= 0x80;  // PT7=RST=1
    out8(0xA2);
    value = in9();
    PTT &= 0x7F;  // PT7=RST=0
    return value;}

Read the Temperatures from the DS1620 (cont)
unsigned short DS_ReadT(void){
    unsigned short value;
    PTT |= 0x80;  // PT7=RST=1
    out8(0xAA);
    value = in9();
    PTT &= 0x7F;  // PT7=RST=0
    return value;}

Read the Temperatures from the DS1620 (cont)
Most devices look like a state machine
What are the inputs to these state machines?
Can think of your code as a state machine too
A protocol defines a way for these state machines to interact
The protocol is like a legal contract: it specifies exactly what each side of the interface must provide and can expect
What happens if, though some glitch, the CPU state machine becomes desynchronized from the device state machine?