Debugging: Another perspective

- An embedded.com article by Robin Knoke entitled "Debugging embedded C" provides another perspective.
- Debugging consumes much of the programmer’s time.
- 4 steps to the debugging process:
  - Testing.
  - Stabilization.
  - Localization.
  - Correction.

Testing & Stabilization

- 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.

## Types of bugs

- **Four classes of bugs:**
  - Lexical: Compile time bug found by the lexer.
  - Syntactic: Compile time bug found by the parser.
  - Intent: Run-time error without catastrophic consequences.
    - One-liner, typematch, boundary, macro, intent.
  - Execution: Run-time error resulting in abnormal termination.
    - Divide by zero, out-of-bounds arrays, incorrectly implemented interrupts, uninitialized pointer assignment.
- **Viral bugs** corrupt code, data, or stack.
- Can be difficult to stabilize and localize.

## Subtle bugs

- **Integration bugs:**
  - Function return-value typematch bugs.
  - Problems with global variables may surface during integration.
  - Interrupt related bugs (improperly saved code, non-reentrant code).
  - Shared resource bugs.
- **Dynamic memory allocation and file handles** must allocate and free resources carefully.
- **Machine specific tricks** become bugs when porting code.
- **Expression of compiler bugs** often indicate poor coding practice.

## Suggestions for writing better C

- Be extremely careful when using pointers; uninitialized-pointer bugs and boundary bugs can be very time consuming to correct.
- Look for typematch bugs using a linter/checker, or check each function call for proper argument and return types by hand.
- Be careful when using macros, especially those containing parameter substitutions.
- Use parentheses liberally to guarantee associativity.
- Think about portability while writing code.
- Use casting when converting types.
- Avoid global variables.
- Use header files for function prototyping and argument definition.
- Use return codes for modules that interface with each other.
- Design tests to exercise all branches in a program. Include tests for boundary conditions, especially for code using pointers. Keep the test suite to aid in modification/porting of the code.
Introduction to interfacing

- 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.

Performance Measures

- **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.
- **Soft real time** system is one that supports priority.
Synchronizing Software with the State of I/O

- 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 transmission.
- 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.

Synchronizing Software with an Input Device

- 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

```
void Init(void){
    DDRT = 0xFF;  // outputs
    DDRM|= 0x01;
    PTM |= 1;     // GO=1
    Timer_Init();
}
void Out(unsigned char value){
    PTT = value;
    PMT&=~0x01;  // GO=0
    PMT|=0x01;   // GO=1
    Timer_MSWait(10);} // 10ms
```

Blind Cycle ADC Interface

```
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_MSWait(5);
    return(PTT);}
```
Blind Cycle Evaluation

- Pros
  - Simple
  - Predictable
- Cons
  - Inflexible
  - Inefficient if the delay is long.
  - Works well for simple, high-speed devices.

Gadfly or Busy Waiting Synchronization

Multiple Gadfly Outputs

Multiple Gadfly Inputs and Outputs
Gadfly Evaluation

**Pros**
- Simple
- A bit more flexible than blind cycle.

**Cons**
- Inefficient
- Potentially unpredictable.
- Doesn’t support priority.
- Use with caution.

Gadfly Key Evaluation

**Pros**
- Simple
- A bit more flexible than blind cycle.

**Cons**
- Inefficient
- Potentially unpredictable.
- Doesn’t support priority.
- Use with caution.

Key Wakeup Interrupts in the 6812

- Available on PJ7 and PJ6 as well as all 8 pins of Port P.
- Only PP5 is available on your module.
- Direction register, DDR and DDRP, sets input or output.
- Active input edge sets a flag in PIFJ and PIFP.
- Either edge (rising or falling) can be configured to be active using PPSJ and PPSP.
- Each has a separate interrupt arm bit in PIEJ and PIEP.
- Key wakeup interrupt generated if flag bit set, arm bit set, and interrupts enabled (I=0).
- Ports also have built in pull up or pull down resistors which are configured using PPSJ/PPSP and PERJ/PERP registers.
- RDRJ and RDRP determine drive strength, if bit is 1 then uses 1/3 drive current to save power.

Initialize and Read from a Keyboard

```c
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);
}
```
**Gadfly ADC Interface Using Simple Input**

```plaintext
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**

```plaintext
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

![Diagram of MC9S12C32 and PJ7, PJ6, PT6-0 connections]

- Initialize and Write to a Printer

```c
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);}
```

Gadfly Sync Serial I/F to Temperature Sensor

- 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°C</td>
<td>011111010</td>
<td>$0FA</td>
</tr>
<tr>
<td>+64.0°C</td>
<td>010000000</td>
<td>$080</td>
</tr>
<tr>
<td>0.5°C</td>
<td>000000001</td>
<td>$001</td>
</tr>
<tr>
<td>0°C</td>
<td>000000000</td>
<td>$000</td>
</tr>
<tr>
<td>-0.5°C</td>
<td>111111111</td>
<td>$1FF</td>
</tr>
<tr>
<td>-16.0°C</td>
<td>111100000</td>
<td>$1E0</td>
</tr>
<tr>
<td>-55.0°C</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
```
Send 8-bits Out to the DS1620

```c
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;
    }
}
```

Start/Stop the DS1620

```c
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
}
```

Config the DS1620

```c
void DS_Config(char data){
    PTT |= 0x80;  // PT7=RST=1
    out8(0x0C);
    out8(data);
    PTT &= 0x7F;  // PT7=RST=0
}
```

Send 9-bits Out to the DS1620

```c
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 the Configuration Register on the DS1620

```c
unsigned char DS_ReadConfig(void){
    unsigned char value;
    PTT |= 0x80; // PT7=RST=1
    out8(0xAC);
    value = in8();
    PTT &= 0x7F; // PT7=RST=0
    return value;
}
```

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;
}
```

Read 9-bits from the DS1620

```c
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 |= 0x100; // PT5=DQ=1
        PTT |= 0x40;} // PT6=CLK=1
    DDRT |= 0x20; // PT5=DQ output
    return result;
}
```
Read the Temperatures from the DS1620

```
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 (cont)

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