Introduction

- Success of an embedded system project depends on both hardware and software.
- Real-time embedded systems are usually not very large, but are often quite complex.
- Needed software skills include: modular design, layered architecture, abstraction, and verification.
- Writing good software is an art that must be developed and cannot be added on at the end of a project.
- Good software with average hardware will always outperform average software with good hardware.

Memory Allocation

(See Figure 2.2)

Pseudo Instructions

- `org` set location to ORiGin
- `fcc` Form Constant Character string
- `fcb` Form Constant Byte
- `fdb` Form Double Byte
- `equ` EQUate symbol to a value
- `rmb` Reserve Memory Bytes
Memory Allocation Example

```
org $0000 ;RAM
cnt rmb 1 ;global
org $B600 ;EEPROM
const fcb 5 ;amount to add
org $E000 ;ROM
init ldaa #$FF
staa DDRC ;outputs
clr cnt
rts
main lds #$00FF ;sp => RAM
bsr init
loop ldaa cnt
staa PORTC ;output
adda const
staa cnt
bra loop
org $FFFF ;ROM
fdb main ;reset vector
```

Golden Rule of Software Development

Write software for others as you wish they would write for you.

- Quantitative performance measurements:
  - Dynamic efficiency - number of CPU cycles required.
  - Static efficiency - number of memory bytes required.
  - Are given design constraints satisfied?

- Qualitative performance measurements:
  - Easy to debug (fix mistakes)
  - Easy to verify (prove correctness)
  - Easy to maintain (add features)

- Sacrificing clarity in favor of execution speed often results in software that runs fast but doesn’t work and can’t be changed.

- You are a good programmer if (1) you can understand your own code 12 months later and (2) others can change your code.

Software Maintenance

- Maintenance is most important phase of development.
- Includes fixing bugs, adding features, optimization, porting to new hardware, configuring for new situations.
- Documentation should assist software maintenance.
- Most important documentation is in the code itself.
- Client comments describe inputs, outputs, and errors.
- Colleague comments focus on software mechanisms.

Good Comments

- Comments that simply restate the operation do not add to the overall understanding.
  - BAD  X=X+4; /* add 4 to X */
  - Flag=0; /* set Flag=0 */
  - GOOD X=X+4; /* 4 is added to correct for the offset (mV) in the transducer */
  - Flag=0; /* means no key has been typed */

- When variable defined, should explain how used.
  - int SetPoint; /* Desired temperature, 16-bit signed value with resolution of 0.5C,
    a range of -55C to +125C,
    a value of 25 means 12.5C */

- When constant defined, should explain what it means.
  - V=999; /* 999mV is the maximum possible voltage */
Good Comments (cont)

- When a subroutine defined, two types of comments:
  - Client comments explain how the function is to be used, how to pass parameters, and what errors and results are possible. (in header or start of subroutine)
  - Colleague comments explain how the function works (within the body of the function).

Self-Documenting Code

- Software written in a simple and obvious way such that its purpose and function are self-apparent.
- Use descriptive names for var, const, and functions.
- Formulate & organize into well-defined subproblems.
- Liberal use of `#define` and `equ` statements.
- Assembly language style issues:
  - Begins and ends with a line of `*`
  - States the purpose of the function
  - Gives the I/O parameters, what they mean, and how they are passed
  - Different phases of code delineated by a line of `-`

Software Documentation

- Purpose of the module
- Input parameters
  - How passed (call by value, call by reference)
  - Appropriate range
  - Format (8 bit/16 bit, signed/unsigned, etc.)
- Output parameters
  - How passed (return by value, return by reference)
  - Format (8 bit/16 bit, signed/unsigned, etc.)
- Example inputs and outputs if appropriate
- Error conditions
- Example calling sequence
- Local variables and their significance

Abstraction

- *Software abstraction* is when we define a complex problem with a set of basic abstract principles.
- Advantages of abstraction:
  - Faster to develop because some building blocks exist,
  - Easier to debug (prove correct) because it separates conceptual issues from implementation, and
  - Easier to change.
- *Finite state machine* (FSM) is a good abstraction.
- Consists of inputs, outputs, states, and state transitions.
- An FSM software implementation is easy to understand, debug, and modify.
Mealy FSM in C

const struct State
{
  unsigned char Time; /* Time to wait in each state */
  unsigned char Out[2]; /* Output if input=0,1 */
} StateType

StateType fsm[4] = {{100, {0,0}, {SB,SD}},
                    {100, {0,8}, {SC,SA}},
                    {15, {0,0}, {SB,SD}},
                    {15, {8,8}, {SC,SA}}};

Mealy FSM in C (cont)

void Wait(unsigned int delay)
{
  int Endt;
  Endt = TCNT + delay; /* Time (125ns cycles) to wait */
  while((Endt-(int)TCNT)>0) /* wait */
  
  
}

void main(void)
{
  StatePtr *Pt; /* Current State */
  unsigned char Input;

  Pt = SA; /* Initial State */

  DDRC = 0x08; /* PortC bit 3 is output */

  while(1)
  {
    Wait(Pt->Time); /* Time to wait in this state */
    Input = PORTC << 7; /* Input = 0 or 1 */
    PORTC = Pt->Out[Input]; /* Perform output */
    Pt = Pt->Next[Input]; /* Move to the next state */
  }
}

Mealy FSM in Assembly

org $B600

Put in EEPROM so it can be changed

* Finite State Machine

Time equ 0 Index for time to wait in this state
Out0 equ 1 Index for output pattern if input=0
Out1 equ 2 Index for output pattern if input=1
Next0 equ 3 Index for next state if input=0
Next1 equ 5 Index for next state if input=1
IS fdb SA Initial state
SA fcb 100 Time to wait
fdb 0,0 Outputs for inputs 0,1
fdb SB Next state if Input=0
fdb SD Next state if Input=1

(See Figure 2.4)
Mealy FSM in Assembly (cont)

<table>
<thead>
<tr>
<th>SB</th>
<th>fcb 100</th>
<th>Time to wait</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>fcb 0,8</td>
<td>Outputs for inputs 0,1</td>
</tr>
<tr>
<td></td>
<td>fdb SC</td>
<td>Next state if Input=0</td>
</tr>
<tr>
<td></td>
<td>fdb SA</td>
<td>Next state if Input=1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SC</th>
<th>fcb 15</th>
<th>Time to wait</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>fcb 0,0</td>
<td>Outputs for inputs 0,1</td>
</tr>
<tr>
<td></td>
<td>fdb SB</td>
<td>Next state if Input=0</td>
</tr>
<tr>
<td></td>
<td>fdb SD</td>
<td>Next state if Input=1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SD</th>
<th>fcb 15</th>
<th>Time to wait</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>fcb 8,8</td>
<td>Outputs for inputs 0,1</td>
</tr>
<tr>
<td></td>
<td>fdb SC</td>
<td>Next state if Input=0</td>
</tr>
<tr>
<td></td>
<td>fdb SA</td>
<td>Next state if Input=1</td>
</tr>
</tbody>
</table>

Mealy FSM in Assembly (cont)

<table>
<thead>
<tr>
<th>is1</th>
<th>ldaa Out1,X</th>
<th>Get desired output from structure</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>staa $1003</td>
<td>Set PC3=output</td>
</tr>
<tr>
<td></td>
<td>ldx Next1,X</td>
<td>Input is 1</td>
</tr>
<tr>
<td></td>
<td>bra LL</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>is0</th>
<th>ldaa Out0,X</th>
<th>Get desired output from structure</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>staa $1003</td>
<td>Set PC3=output</td>
</tr>
<tr>
<td></td>
<td>ldx Next0,X</td>
<td>Input is 0</td>
</tr>
<tr>
<td></td>
<td>bra LL</td>
<td>Infinite loop</td>
</tr>
</tbody>
</table>

Moore FSM

(See Figure 2.4)

Mealy FSM in Assembly (cont)

<table>
<thead>
<tr>
<th>org $E000</th>
<th>Place assembly program in ROM</th>
</tr>
</thead>
<tbody>
<tr>
<td>* 6811 program</td>
<td></td>
</tr>
<tr>
<td>* Initialization of 6811 and linked structure</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GO</th>
<th>lds #80FF</th>
<th>Initialize stack</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ldaa #80</td>
<td>PC3=output, rest are input</td>
</tr>
<tr>
<td></td>
<td>staa $1007</td>
<td>Set DDRC</td>
</tr>
<tr>
<td></td>
<td>ldx IS</td>
<td>Reg X =&gt; current state</td>
</tr>
</tbody>
</table>

*Linked structure interpreter

<table>
<thead>
<tr>
<th>LL</th>
<th>ldaa Time,X</th>
<th>Time to wait</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>bsr WAIT</td>
<td>Reg A is call by value</td>
</tr>
<tr>
<td></td>
<td>ldaa $1003</td>
<td></td>
</tr>
<tr>
<td></td>
<td>bira #$80</td>
<td>Test input PC7</td>
</tr>
<tr>
<td></td>
<td>bpl is0</td>
<td>Go to is0 if Input=0</td>
</tr>
</tbody>
</table>

fdb GD reset vector
Modular Software Development

- Modular programming breaks software problems in distinct and independent modules.
- Modular software development provides:
  - Functional abstraction to allow software reuse.
  - Complexity abstraction (i.e., divide and conquer).
  - Portability.
- A program module is a self-contained software task with clear entry and exit points.
- Can be a collection of subroutines or functions that in their entirety perform a well-defined set of tasks.

Stack Contents

movb ss,1,-sp ;push parameter onto stack
jsr sqrt ;call sqrt subroutine
ins
stab tt ;save result

(See Figure 2.7)

Software Module

(See Figure 2.6)

Example Module in Assembly

sqrt pshy
tsy
leas -4,sp ;allocate t,oldt,s16
cirb
ldaa s8,y
...
stab t,y
dec cnt,y
bne next
done tys
pul
rts
Returning Multiple Parameters in Assembly 1

module: ldax #1
ldab #2
ldx #3
ldy #4
rts ;returns 4 parameters in 4 registers

*******calling sequence******
sjr module
* Reg A,B,X,Y have four results

Returning Multiple Parameters in Assembly 2

data1 equ 2
data2 equ 3
module movb #1, data1, sp ;1st parameter onto stack
module movb #2, data2, sp ;2nd parameter onto stack
rts

*******calling sequence******
leas -2,sp ;allocate space for results
jsr module
pula ;1st parameter from stack
staa first
pula ;2nd parameter from stack
staa second

More Issues in Modular Software

- All exit points in an assembly routine must balance the stack and return parameters in the same way.
- Performing unnecessary I/O in a subroutine makes it harder to reuse at a later time.
- I/O devices must be considered global, and the number of modules that can access them should be restricted.
- Information hiding means to separate mechanism from policies (i.e., hiding the inner workings from the user).

Dividing a Software Task into Modules

- Coupling is influence one module’s behavior has another, and is typically caused by shared variables.
- When dividing into modules have these goals:
  - Make the software project easier to understand.
  - Increase the number of modules.
  - Decrease the interdependency (minimize coupling).
- Develop and connect modules in a hierarchical manner.
  - Top-down - “Write no software until every detail is specified.”
  - Bottom-up - “one brick at a time.”
### Simple Calling Graph

(See Figure 2.10)

### Rules for Modular Software in Assembly

- The single entry point is at the top.
- The single exit point is at the bottom.
- Write structured programs.
- The registers must be saved.
- Use high-level languages when possible.
- Minimize conditional branching.

### Layered Software Systems

- Software undergoes many changes due to as better hardware or algorithms become available.
- Layered software facilitates these changes.
- The top layer is the main program.
- The lowest layer, the *hardware abstraction layer*, includes all modules that access the I/O hardware.
- Each layer can only call modules in its layer or lower.
- A *gate* (also known as an application program interface (API)) is used to call from a higher-to a lower layer.
- The main advantage is that one layer can be replaced without affecting the other layers.

### Layered Approach for Parallel Port

(See Figure 2.11)
Three-Layer Software System (High Level)

```
org $E000
main:
lds #800FF
bsr Initialize ; simple call to a function
loop:
bsr PrintInfo ; simple call to a function
bra loop
Initialize: ldaa #1 ; function code for InitPrinter
swi ; call to middle level
rts
org $FFFE ; high level vector (reset)
```

```
fdb main
```

Three-Layer Software System (Middle Level)

```
org $F400
swihandler:
cmpa #1 ; function code for InitIEEE
bne notInit
bsr InitIEEE
bra swidone
notInit:
cmpa #2 ; function code for SetDAV
bne notDAV
bsr SetDAV
bra lowdone
notDAV: ; rest of the functions
lowdone: rts
InitIEEE: ; access to hardware
rts
org $FF80 ; lower level vector
```

```
fdb swihandler
```

Three-Layer Software System (Low Level)

```
org $F800
lowhandler:
cmpa #1 ; function code for InitIEEE
bne notInit
bsr InitIEEE
bra swidone
notInit:
cmpa #2 ; function code for SetDAV
bne notDAV
bsr SetDAV
bra lowdone
notDAV: ; rest of the functions
lowdone: rts
InitIEEE: ; access to hardware
rts
org $FF80 ; lower level vector
```

```
fdb lowhandler
```

Layered Software Rules

1. A module may make simple call to modules in same layer.
2. A module may call a lower-level module only using gate.
3. A module may not directly access any function or variable in another layer (w/o going through a gate).
4. A module may not call a higher-level routine.
5. A module may not modify the vector address of another level’s handler(s).
6. (Optional) A module may not call farther than one level.
7. (Optional) All I/O hardware access is in lowest level.
8. (Optional) All user interface I/O is in highest level unless it is the purpose of the module to do such I/O.