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.
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

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.

Memory Allocation Example

Memory Allocation Example

Slide 5

org $0000 ;RAM
cnt rmb 1 ;global
org $B600 ;EEPROM
cst 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 $FFFE ;ROM
fdb main ;reset vector

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  
  \begin{verbatim}
  X=X+4;  /* add 4 to X */
  Flag=0;  /* set Flag=0 */
  \end{verbatim}

  GOOD  
  \begin{verbatim}
  X=X+4;  /* 4 is added to correct for the offset (mV) in the transducer */
  Flag=0;  /* means no key has been typed */
  \end{verbatim}

- When variable defined, should explain how used.
  \begin{verbatim}
  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 */
  \end{verbatim}

- When constant defined, should explain what it means.
  \begin{verbatim}
  V=999;  /* 999mV is the maximum possible voltage */
  \end{verbatim}

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 \texttt{define} and \texttt{equ} statements.
- Assembly language style issues:
  - Begins and ends with a line of *'s
  - 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 '-'s

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).

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

- **Mealy FSM in C**

```c
const struct State
{ unsigned char Time; /* Time to wait in each state */
  unsigned char Out[2]; /* Output if input=0,1 */
  const struct State *Next[2]; /* Next state if inp=0,1*/
typedef const struct State StateType;
#define SA &fsm[0]
#define SB &fsm[1]
#define SC &fsm[2]
#define SD &fsm[3]
StateType fsm[4] = {{100, {0,0}, {SB,SD}},
                   {100, {0,8}, {SC,SA}},
                   { 15, {0,0}, {SB,SD}},
                   { 15, {8,8}, {SC,SA}}};
```

```c
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 bit3 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

Slide 17

```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
   fcb 0,0 Outputs for inputs 0,1
   fdb SB  Next state if Input=0
   fdb SD  Next state if Input=1
```

Slide 18

Mealy FSM in Assembly (cont)

```assembly
SB  fcb 100 Time to wait
   fcb 0,8 Outputs for inputs 0,1
   fdb SC  Next state if Input=0
   fdb SA  Next state if Input=1
SC  fcb 15 Time to wait
   fcb 0,0 Outputs for inputs 0,1
   fdb SB  Next state if Input=0
   fdb SD  Next state if Input=1
SD  fcb 15 Time to wait
   fcb 8,8 Outputs for inputs 0,1
   fdb SC  Next state if Input=0
   fdb SA  Next state if Input=1
```

Slide 19

Mealy FSM in Assembly (cont)

```assembly
org $E000  Place assembly program in ROM
* 6811 program
* Initialization of 6811 and linked structure
GO  lds #$00FF  Initialize stack
    ldaa #$08  PC3=output, rest are input
    staa $1007  Set DDRC
    ldx IS  Reg X => current state
    fdb SA  Initial state
    fcb 100 Time to wait
    fdb SB  Next state if Input=0
    fdb SD  Next state if Input=1
```

Slide 20

Mealy FSM in Assembly (cont)

```assembly
is1  ldaa Out1,X Get desired output from structure
    staa $1003  Set PC3=output
    ldx Next1,X Input is 1
    bra LL
is0  ldaa Out0,X Get desired output from structure
    staa $1003  Set PC3=output
    ldx Next0,X Input is 0
    bra LL Infinite loop
```
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
Example Module in Assembly

```
sqrt pshy
tsy
leas -4,sp ;allocate t,oldt,s16
cr
lda s8,y
...
stab t,y
dec cnt,y
bne next
done tys
puly
rts
```

Returning Multiple Parameters in Assembly 1

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

*****calling sequence*****
jsr 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.”

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.

Simple Calling Graph

```
void main(void){
    unsigned char n:
    SCInport();
    while(1)
    SCOutport(n++);
}
```

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

Three-Layer Software System (High Level)

```
org $E000
main:
  lds #$00FF
  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 $FFF6 ; high level vector (reset)
  fdb main
```

Three-Layer Software System (Middle Level)

```
org $F400
swihandler:
  cmpa #1 ; function code for InitPrinter
  bne notIP
  bsr InitPrinter
  bra swidone
notIP:
  cmpa #2 ; function code for PrintInfo
  bne notPI
  bsr PrintInfo
  bra swidone
notPI:
  ; **** error
swidone:
  rti
InitPrinter:
  ldaa #1 ; function code for InitIEEE
  ldx $FF80 ; vector into low level
  jsr 0,x ; call lower level
  rts
  org $FFF6 ; middle level vector (SWI)
  fdb swihandler
```

Three-Layer Software System (Low Level)

```
org $F800
lowhandler:
  cmpa #1 ; function code for InitIEEE
  bne notInit
  bsr InitIEEE
  bra lowdone
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.