

Lecture 3 objectives

- Quality software
- Debugging concepts
- Programming style guidelines for this class

Quality Programming

- easy to debug (fix mistakes)
- easy to verify (prove correctness)
- easy to maintain (add features)

Dynamic efficiency**Bandwidth**

- calculations performed per second
- data transferred per second

Latency or response time (real time means bounded latency)

- time from a request to the time action satisfies request
- time new input ready to time data is read
- time output device is idle to time new data is written

Static efficiency**RAM (variables) and ROM (constants, program)****Lab 1-6,9,10 9S12DP512**

2 kibibytes RAM, 32 kibibytes of EEPROM

Labs 8,11 9S12DP512, 9S12C32, 9S12C128, MSP430F2013

9S12C32, 2 KiB RAM, 32 KiB EEPROM

9S12C128, 4 KiB RAM, 128 KiB EEPROM

MSP430F2013, 128 bytes RAM, 2 KiB EEPROM

Specifications (what the system must do)**Bandwidth, latency****Accuracy (difference between measured and truth)****Resolution (smallest change that can be detected)****Repeatability (standard deviation of multiple observations)**

same operator, conditions, day, machine

Reproducibility (standard deviation of multiple observations)

different operator, conditions, day, machine

Constraints (what the system must not do)**Power, size, weight****Must not stop running for within 24 hours on +9V****Must not be bigger than 5 by 3 by 1 inch****Must not weight more than 1 lbs****software development costs,****Must not cost more than \$100,000****memory available,****Must not need more than 2 KiB RAM or 32 KiB ROM****time-table.****Must not take more than 1 month to produce****Software maintenance**

- Verification of proper operation
- Fixing bugs
- Adding new features
- Extending it to solve new applications

client

programmers who will use our software
develops software that will call our functions

coworkers

programmers who will debug and upgrade our software
develops, tests, and modifies our software.

attitude

- embarrassed to deliver poorly written software
- modules that are easy to change
- expect our code will be modified
- reward good behavior
- punish bad behavior
- test it now, fix it now
- plan for testing

Golden Rule of Software Development

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

The turtle wins the race! It is better to have a software system that runs slow than one that does run at all.

Observation: There are two ways of constructing a software design: one way is to make it so simple that there are obviously no deficiencies and the other way is make it so complicated that there are no obvious deficiencies. C.A.R. Hoare, "The Emperor's Old Clothes," CACM Feb. 1981.

Debugging

- performance debugging (timing)
- functional debugging (data)

Goal of debugging

maintain and improve software
remedy faults or to correct errors in a program
role of a debugger is to support this endeavor

The debugging process

testing,
stabilizing,
localizing, and
correcting errors.

Intrusiveness

degree of perturbation caused by the debugging itself
how much the debugging slows down execution

Nonintrusive

characteristic or quality of a debugger
allows system to operate as if debugger did not exist
e.g., logic analyzer, ICE

Minimally intrusive

negligible effect on the system being debugged
e.g., dumps (ScanPoint) and monitors, BDM

highly intrusive

e.g., print statements, breakpoints and single-stepping

“rough and ready” manual methods

desk-checking

hand execute the program and think about it a lot

dumps,

save important data into an array, look at it later

print statements

print important as it is running

2.3 Self-Documenting Code**Naming convention**

- Names should have meaning.
- Avoid ambiguities.
- Give hints about the type.
- Use the same name to refer to the same type of object.
- Use a prefix to identify public objects.
- Use upper and lower case to specify the scope of an object.
- Use capitalization to delimit words.

An object's properties (public/private, local/global, constant/variable) are always perfectly clear at the place where the object is defined. The importance of the naming policy is to extend that clarity also to the places where the object is used.

type	examples
constants	SAFE_TO_RUN START_OF_RAM PORTA
local variables	maxTemperature lastCharTyped errorCnt
private global variable	MaxTemperature LastCharTyped ErrorCnt
public global variable	DAC_MaxTemperature Key_LastCharTyped Network_ErrorCnt
private function	ClearTime wrapPointer InChar
public function	Timer_ClearTime RxFifo_Put Key_InChar

Examples of names.

C language Style Guidelines**Code File Structure, the *.c file****Opening comments in the code file.**

- Intended to be read by co-worker
- first line should contain the file name.
- the overall purpose of the software module,
- the names of the programmers,
- the creation (optional) and last update dates,
- the hardware/software configuration required to use the module,
- any copyright information.

Including .h files.

- will help us draw a call-graph
- avoid having one header file include other header files
- only those files that are absolutely necessary

#define statements.

they will be private
client does not need to use

struct union enum statements.

create the necessary data structures
they will be private.

Global variables and constants.

If **static** then it will be private
If no **static** then it will be public

The **scope** of a variable includes all the software in the system that can access it. In general, we wish to minimize the scope of our data.

```
char publicGlob;           // any function
static char privGlob;     // this file only
void function(void){
static char veryPrivGlob; // this function only
}
```

Maintain order in our system by restricting direct access to our data.

Prototypes of private functions

- maintains a top-down organization
- private functions by defining them as **static**.
- include the parameter names with the prototypes.

For example, which of the following prototypes is easier to understand?

```
static void plot(short, short);
static void plot(short time, short pressure);
```

Implementations of the functions

private functions should be defined as static
sequenced in a logical manner
highest level to lowest level, showing hierarchy
order in which the functions will be used
open
input
output
close

Header File Structure, the *.h file (show sci12.h, heap.h)

There are two types of header files.

- 1) has no corresponding code file (not part of a module)
 - list global constants and helper macros
 - I/O port addresses and calibration coefficients
 - Debugging macros could be grouped together
 - global in nature and do not belong to a module

- 2) has a corresponding code file (part of a module)
 - define the prototypes for public functions
 - contains the policies (behavior or what it does)
 - read by the client
 - often one sells header file with compiled object code

Opening comments in header file.

- intended to be read by the client
- first line should contain the file name.
- the overall purpose of the software module,
- the names of the programmers,
- the creation (optional) and last update dates,
- the hardware/software configuration required to use module
- any copyright information.

Including .h files

Nested includes in the header file should be avoided

#define statements.

Public constants and macros are next.

Public or private?

begin with everything private, and then shift if necessary
relates to how to use the module, then public.

relates to how it works or how it is implemented, private.

struct union enum statements.

public data structures

Global variables and constants

public global variables should be avoided

Prototypes of public functions

arrange in a meaningful order
comments should be directed to the client
clarify what the function does
explain how the function can be used.

Formatting

easier to understand,
easier to debug, and
easier to change.

Make the software easy to read on the screen.

no hardcopy printouts during the development phase
looks pretty on the computer screen.
no horizontal scrolling allowed
If we do make printouts then it will be easy to read.
functions should understandable without vertical scrolling

Indentation should be set at 2 spaces.

no tabs
Local variables
on same line as function definition, or
in first column on next line.

Be consistent about where we put spaces.

no space before a comma or a semicolon,
 at least one space or return after comma or semicolon.
 no space before or after open or close parentheses.
 Assignment and comparison operations
 should have a single space before and after the operation.
 we can line up the operators and values. For example

```
data      = 1;
pressure = 100;
voltage  = 5;
```

Be consistent about where we put braces {}.

braces cause both syntax and semantic errors
 opening brace visual clue that a new code block has started
 close brace gives visual clue that code is in different block

```
void main(void){ int i, j, k;
  j = 1;
  if(sub0(j)){
    for(i = 0; i < 6; i++){
      sub1(i);
    }
    k = sub2(i, j);
  }
  else{
    k = sub3();
  }
}
```

Bad

```
if(flag)
  n = 0;
```

Good

```
if(flag){
  n = 0;
}
```

Good

```
if(flag)
{
  n = 0;
}
```

Code Structure

Make the presentation easy to read

the majority of a function fits on a single computer screen.
 reduce the 2-D *area* of our functions
 encapsulating components
 and defining them as private functions,
 combining multiple statements on a single line.
 group multiple statements on a single line
 if the collection makes sense
 we can draw a circle around the statements
 and assign a simple collective explanation

The first example has a horrific style.

```
void testFilter(short start, short stop, short step){ short x,y; initFilter();
SCI_OutString("x(n) y(n)"); SCI_OutChar(CR); for(x=start;x<=stop; x=x+step){
y=filter(x); SCI_OutUDec(x); SCI_OutUDec(y); SCI_OutChar(CR);} }
```

The second example places each statement on a separate line, unnecessarily vertical.

```
void testFilter( short start, short stop, short step){
short x;
short y;
  initFilter();
  SCI_OutString("x(n) y(n)");
  SCI_OutChar(CR);
  for(x = start; x <= stop; x = x+step){
    y = filter(x);
    SCI_OutUDec(x);
    SCI_OutUDec(y);
    SCI_OutChar(CR);
  }
}
```

The last example collections are considered as a single object.

```
void testFilter(short start, short stop, short step){
short x, y;
  initFilter();
  SCI_OutString("x(n) y(n)"); SCI_OutChar(CR);
  for(x = start; x <= stop; x = x+step){
    y = filter(x);
    SCI_OutUDec(x); SCI_OutUDec(y); SCI_OutChar(CR);
  }
}
```

++ and -- should not appear in complex statements.

bad

```
*(--pt) = buffer[n++];
```

good

```
--pt;
*(pt) = buffer[n];
n++;
```

Comments

The beginning of every file (Show SCI12.c and heap.c)

The beginning of every function (Show SCI12.c)

- line delimiting the start of the function
- purpose
- input parameters
- output parameters, and
- special conditions that apply.
- explain the policies
- intended to be read by the client.

variable or constant definition

- clarify the usage
- specify the units
- include examples.

```
short V1; // voltage at node 1 in mV,
// range -5000 mV to +5000 mV
```

```
unsigned short Fs; // sampling rate in Hz
```

```
char FoundFlag; // keyword found yet?
// 0 if keyword not yet found, 1 if found
```

```
unsigned char RunMode; // system mode
// 0 means idle
// 1 means startup
// 2 means active run
// 3 means stopped
```

describe complex algorithms
intended to be read by our coworkers
assist in changing the code in the future

Examples of bad comments include:

```
time++; // add one to time
mode = 0; // set mode to zero
```

Good comments explain why the operation is performed, and what it means:

```
time++; // elapsed time in msec
mode = 0; // idle mode, no data is avail
```