Embedded System
Lecture Note 4. Embedded System Programming(I)

tags: `Embedded`

— Chenging Fri, May 1, 2021

課程綱要

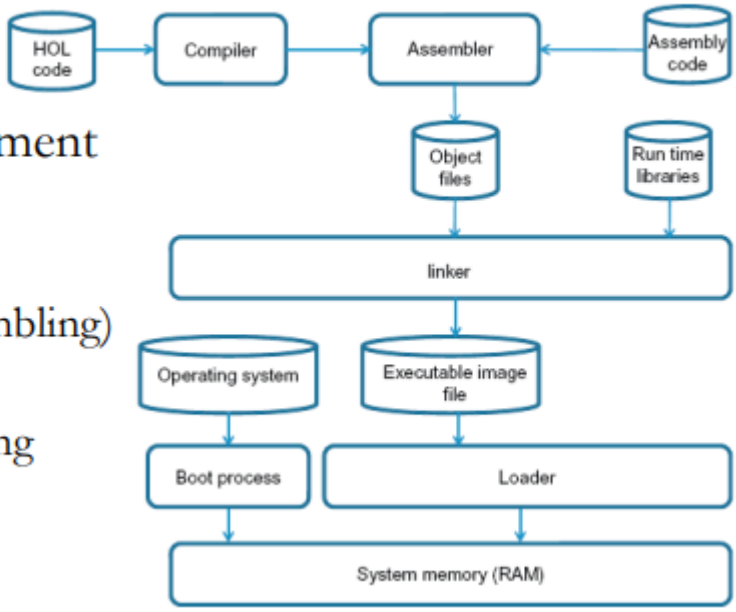

課程綱要：

編號	主　　題	內　容　重　點
1	嵌入式系統程式設計簡介	● 嵌入式系統程式設計與發展環境 ● 嵌入式系統程式開發與C程式語言 ● 程式設計開發工具簡介：Make
2	基本程式開發技術	● 程式開發流程與結構化程式設計 ● 程式效率與優化 (Optimization)
3	嵌入式系統程式設計技術	● 嵌入式系統程式基本控制流程 ● 嵌入式系統程式開發與狀態機實現 ● 程式強韌性與看門狗計時器(WDT)

I. 嵌入式系統程式設計簡介

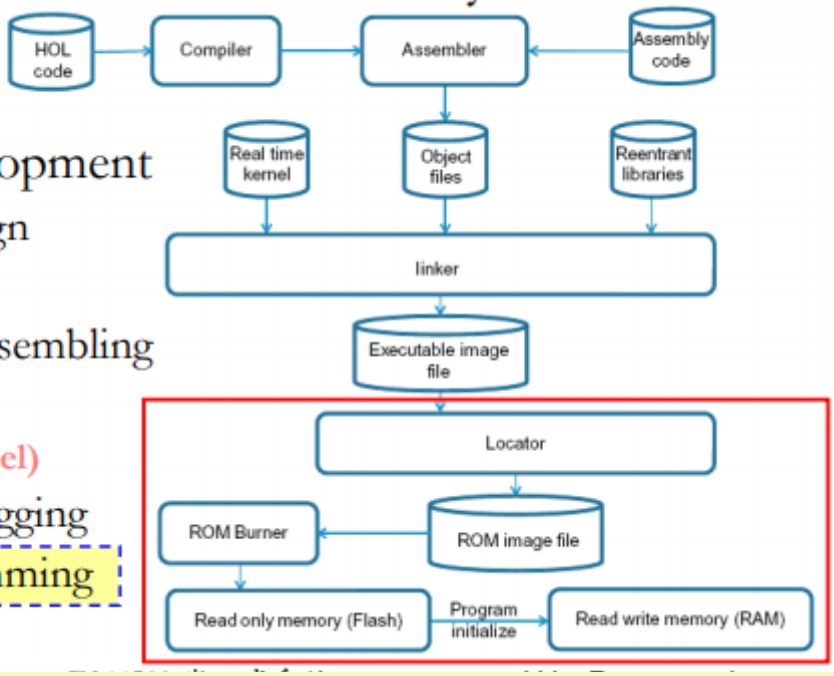
A. Embedded System Programming

Building Program for General Purpose Computer:

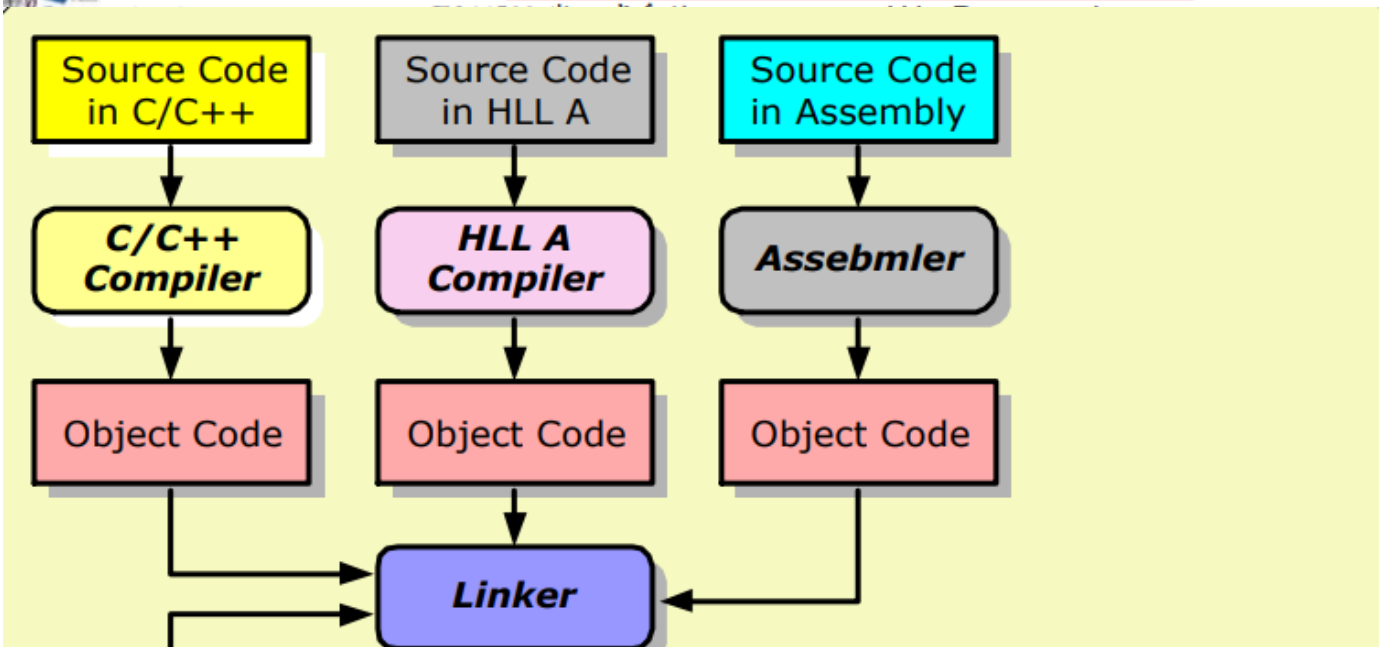


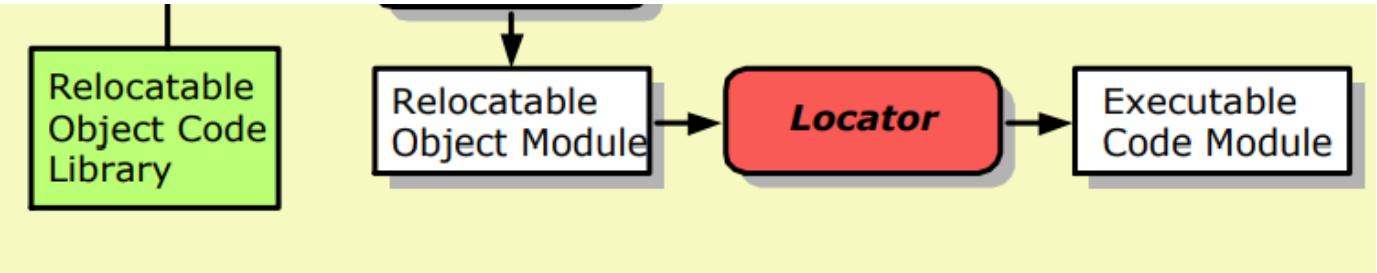
- Program development
  - Program Design
  - Coding
  - Compiling (Assembling)
  - Linking
  - Testing/Debugging

Building Program for Embedded System:



- Program development
  - Program Design
  - Coding
  - Compiling/Assembling
  - Linking
    (/w RTOS Kernel)
  - Testing/Debugging
  - ROM programming

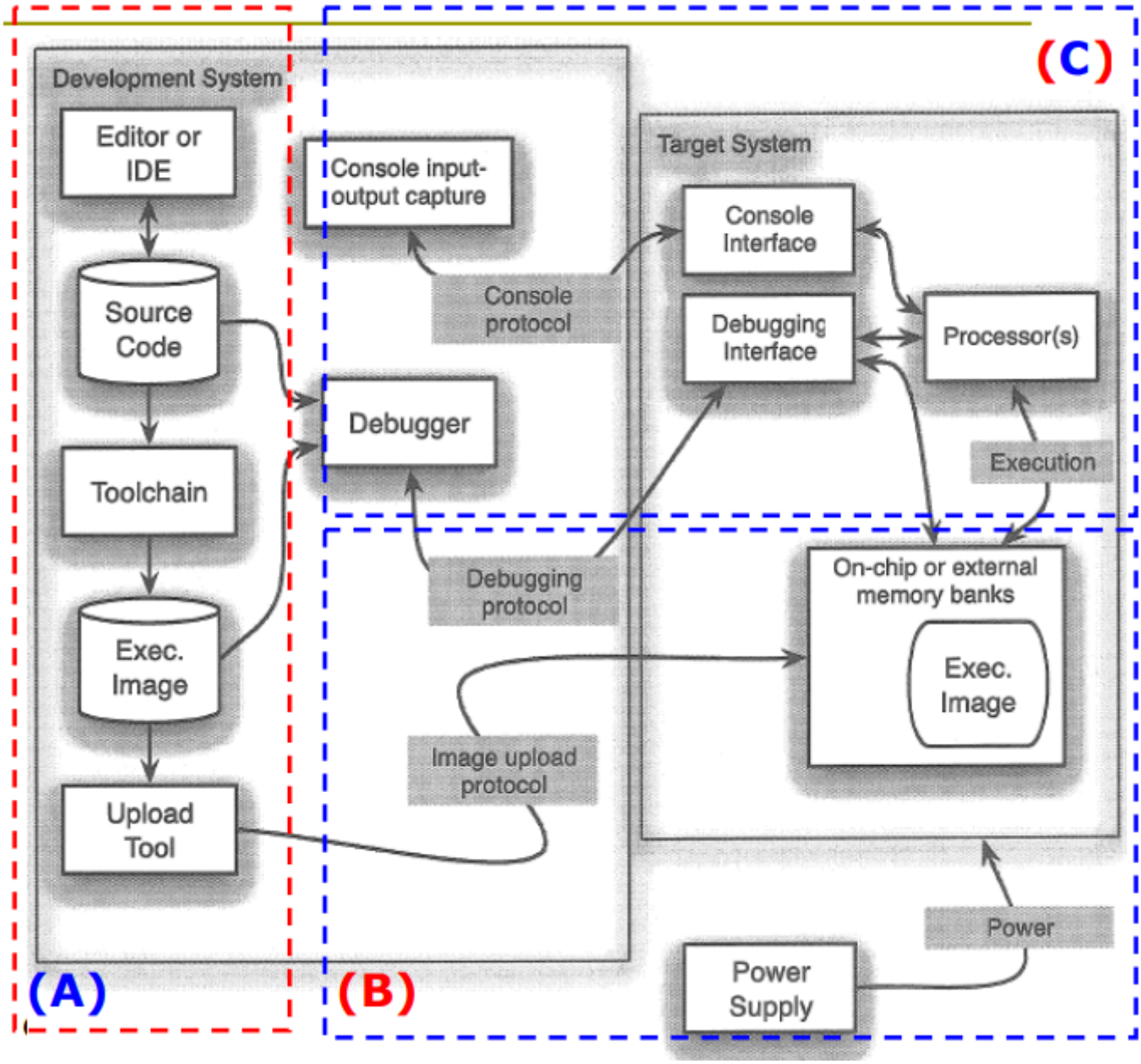




B. Embedded System & Software

The development of embedded system software

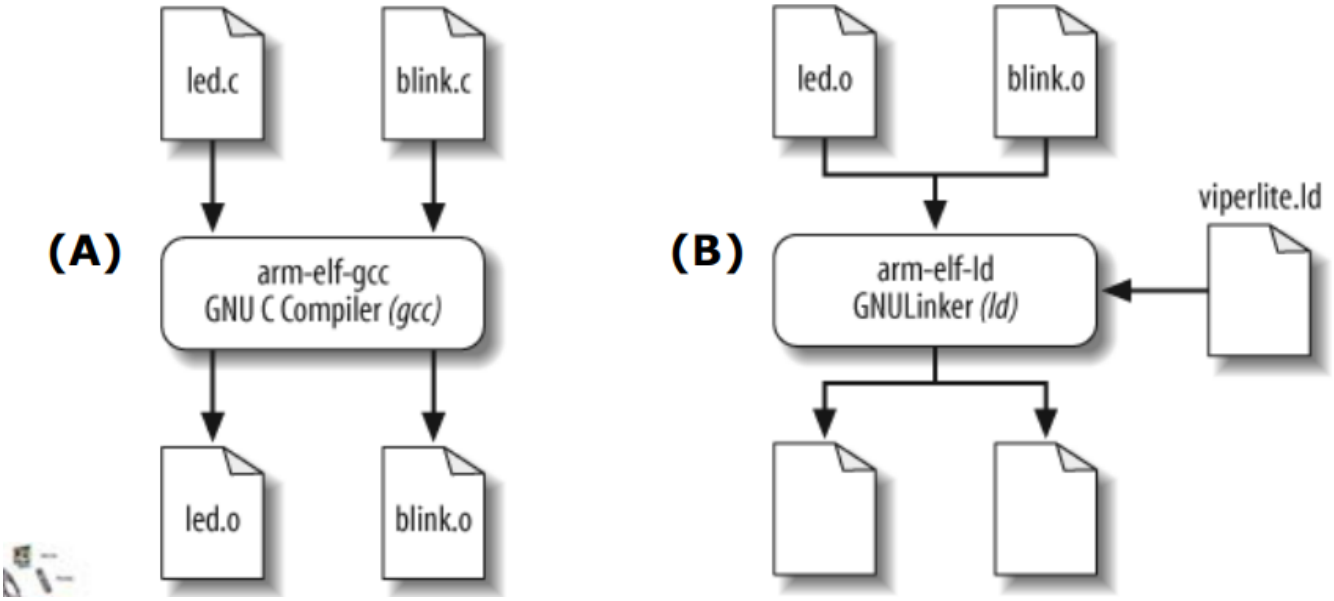
- (A) Building the software
  - Programming
  - Coding
  - Creating exec. image
- (B) Download the image
- (C) Testing (Debugging)



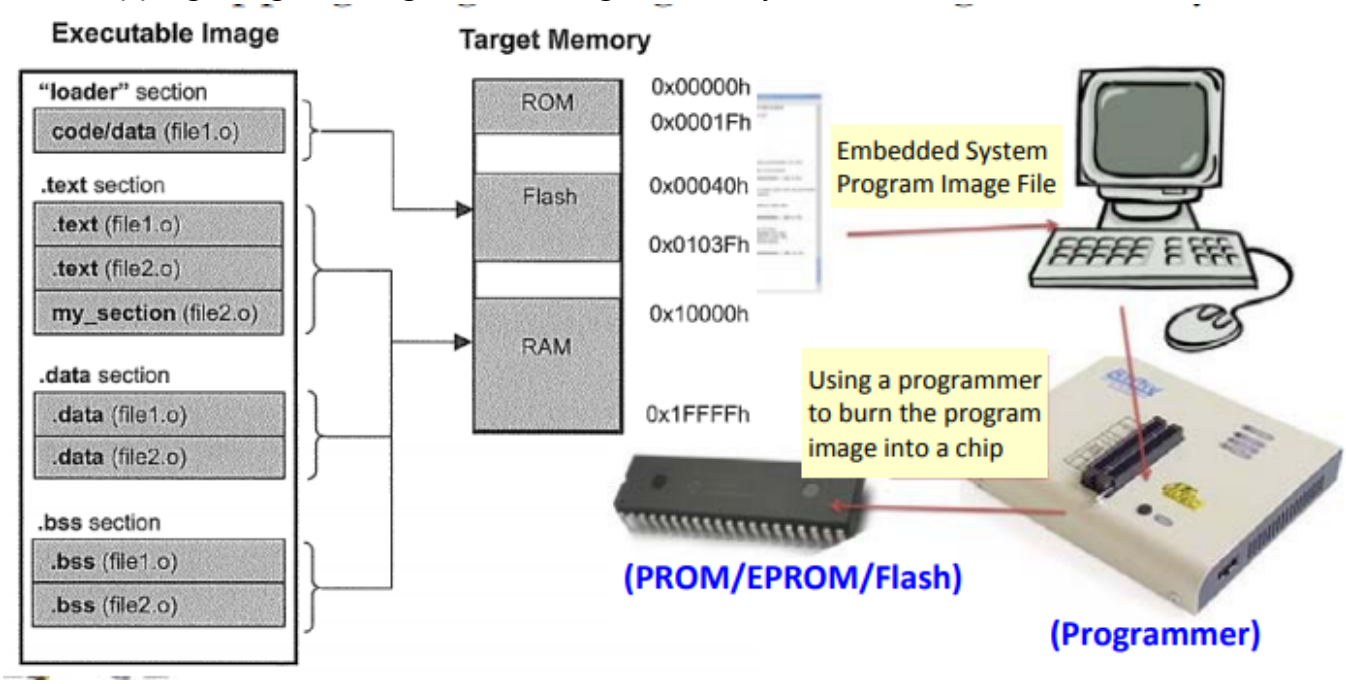
(A) Compiling using GNU GCC

(B) Linking and Locating using GNU Linker (ld)

C. Ex: “Hello, Embedded World!”



(C) Mapping Image & Program into Target Memory



- (1) Compiling using GNU GCC
  - gcc –c blink.c –o blink.o
  - gcc –c startup.s –o startup.o
- (2) Linking and Locating using GNU Linker (ld)
  - ld –T blink.ld blink.o startup.o
    -O blink.elf
- (3) Produce memory image
  - objcopy –o ihex blink.elf blink.bin
- (4) Program blink.bin into ROM/Flash memory
- (5) Reset the target machine to run the program

D. Embedded System Programming

1. The Programming languages

- High-level programming language is desirable
  - C is most widely used (might not be very satisfactory!)
  - Other high-level programming languages:
    - C++
    - Java ME/Java (Android)
    - Ada
    - ...
- Assembly language is powerful but not welcomed
  - Control hardware directly
  - Program efficiency
  - Hardware dependent
- Using high-level programming languages in embedded system programming
  - Advantages:
    - Provide good expressiveness
    - Enable high productivity
    - Ease Maintainability and portability
  - Disadvantages:
    - Requiring specific run-time environment
    - Lacking of desirable features
    - More impacting factors in program efficiency
- C is good and popular but not very satisfactory!
- Desirable features of programming languages:
  - Interrupt service routines (ISR)
  - Device control (I/O control)
  - Accessing physical memory by specifying address
  - Timing control and real-time constraints
  - Concurrent processes or multi-tasking
  - ...
- Common approach – Extending HLL features:
  - New syntax/keyword [e.g. C99/C11]
  - Run-time support function library [e.g. inp()/outp()]

- In-line assembly

2. C for Embedded System Programming:

- GNU C is freely available and widely used

- C language in embedded system programming
  - C language in embedded system programming
  - Useful standard features:
    - Union and declaration of bit data
    - Attribute descriptor volatile
    - Pointer (type *)<addr> for accessing memory
  - Features from platform-specific extensions:
    - In-line/embedded code in assembly language
    - Declaring desired location of object (at or _at)
    - Interrupt service routine (Keyword: interrupt)
  - Dependent of compiler implementation

- Union and bit data operations in C program:
  - Union:
    - Variable holding objects of different types and sizes
    - Similar syntax as the struct of C
    - All the members of a union share the same memory area
  - Bit data operations:
    - Bit-wise Boolean operations
    - Shift left/right
  - Applications:
    - Manipulating bit data
    - Input/Output control

- Syntax of **union** declaration:

```
union u_tag {
    union-member-1;
    union-member-2;
    ...
} uvar;
```

- Access of a union member:
  (1) *uvar.member*
  (2) *union-pointer -> member*

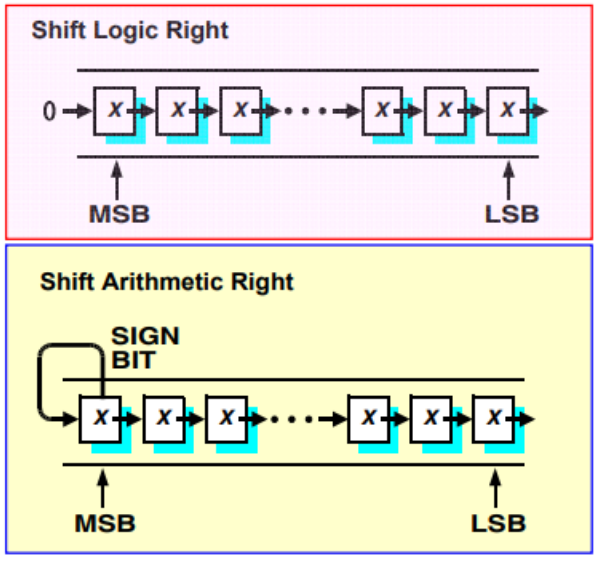
```
typedef unsigned char U8;
struct IO_Control {
  union {
      U8 CMD_REG;
      struct {
        U8 OP: 2;
        U8 DIR: 1;
        U8 IBIT: 1;
        U8 DontCare: 4;
      } BITS;
  } CMD;
} DEV_CMD;
DEV_CMD.CMD.CMD_REG = 0x8F;
DEV_CMD.CMD.BITS.OP = 0x2;
```

- Bit operations in C program
  - Common operations of bit manipulation:
    - Testing – Bit_Data & 1 (Byte_Data & 0x04)
    - Setting – Bit_Data | 1 (Byte_Data | 0x04)
    - Clearing – Bit_Data & 0 (Byte_Data & 0xFB)
    - Inverting – ~ Bit_Data (Byte_Data ^ 0x04)
  - Useful operations:

a	a & 0 ➡ 0	a & 1 ➡ a	a | 0 ➡ a	a | 1 ➡ 1	a ^ 0 ➡ a	a ^ 1 ➡ ~a
0	0	0	0	1	0	1
1	0	1	1	1	1	0

  - Bit-wise shift operations on data
    - **Shift left** <<
    - **Shift right** >>
  - Common operations:
    - Packing data bits
    - Unpacking data bits
    - Extracting data bits
    - Inserting data bits
  - Example:
    - Handling packed date/time

📖 **Note: Logic and Arithmetic shift**



- Some Examples of Bit operations

Example of Bit Operations – Testing bit value(s)

- **a & 0 ➡ 0** (bit unchanged)
- **a & 1 ➡ a** (bit set)

```
if((status_bits & 64) != 0) {      /* Check bit 6   */
 /*  Do the actions if bit 6 is set.  */
 }
```

$(b_7b_6b_5b_4b_3b_2b_1b_0)$ & (01000000) ⇒ $(0b_6000000)$

64 = 0x0040 / (1<< 6)

Example of Bit Operations – Setting bit(s)

- **a | 0 ➡ a** (bit unchanged)
- **a | 1 ➡ 1** (bit set)

```
status_data |= (1 << 5);        /* Sets bit 5       */
```

(1 << 5) ⇒ (00100000)

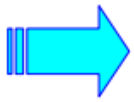
$(b_7b_6b_5b_4b_3b_2b_1b_0)$ ⇒ $(b_7b_61b_4b_3b_2b_1b_0)$

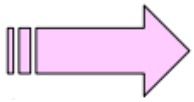
Example of Bit Operations – Clearing bit(s)

- **a & 0 ➡ 0** (bit cleared)
- **a & 1 ➡ a** (bit unchanged)

```
status_data &= ~(1 << 7);     /* Clears bit 7      */
```

(1 << 7) (10000000)
~(1 << 7) (01111111)

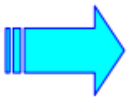


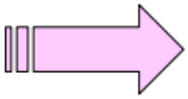
$(b_7b_6b_5b_4b_3b_2b_1b_0)$  $(\mathbf{0}b_6b_5b_4b_3b_2b_1b_0)$

Example of Bit Operations – Inverting bit(s)

- **a ^ 0 ➡ a**
- **a ^ 1 ➡ ~a** (Toggling)

```
status_data ^= (1 << 6);      /* Flips bit 6       */
```

(1 << 6)  (01000000)

$(b_7b_6b_5b_4b_3b_2b_1b_0)$  $(b_7\mathbf{0}b_5b_4b_3b_2b_1b_0)$

Example of Bit Operations – Extracting Bit Data

	Bits 15 - 11	Bits 10 - 5	Bits 4 - 0
time	Hours	Minutes	Seconds ÷ 2
time >> 5	?????	Hours	Minutes
(time >> 5) & 0x3F	00000	00000	Minutes

	15	0
minutes = (time >> 5) & 0x3F	Minutes	

Example of Bit Operations – Inserting Bit Data

	Bits 15 - 11	Bits 10 - 5	Bits 4 - 0
oldtime	Hours	Old Minutes	Seconds ÷ 2
newtime = oldtime & ~(0x3F << 5)	Hours	000000	Seconds ÷ 2
newtime |= (newmins & 0x3F) << 5	Hours	New Minutes	Seconds ÷ 2

Example of Bit Operations – (Un)Packing date

- The format of 16-bit packed date: **mm/dd/(year-2000)**

■ Data Declaration:

```
/* -- Unpacked Date Data -- */
 int   yy;
 int   mm;
 int   dd;
/* -- Packed Date Data ---- */
 unsigned int pdate;
/* ----------------------- */
```

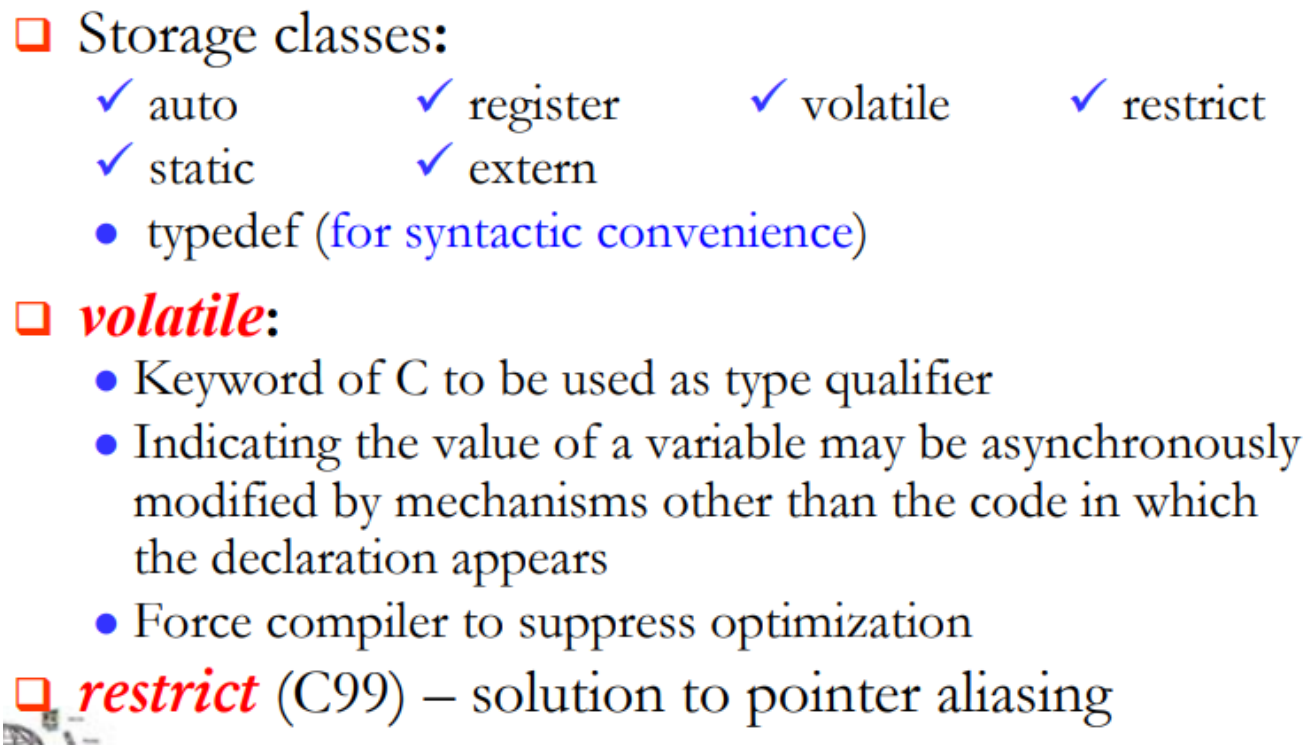
🛠 **Packing:**

```
pdate = mm <<(15-3);
pdate |= dd <<(11-4);
pdate |=(yy-2000)&0x7F;
```

🛠 **Unpacking:**

```
yy =(pdate & 0x7F)+2000;
mm =(pdate & 0xF000)>>12;
dd =(pdate & 0x0F80)>>7;
```

- Attribute descriptors in C program



- In-line assembly code in C program
  - Compiler dependent directive or macro
    - _asm, _asm{ … }
    - asm(…)
    - _asm/_endasm
- Compiler dependent directive or macro
  - _asm, _asm{ … }
  - asm(…)
- _asm/_endasm

- Some Examples of in-line assembly code in C

➢ Syntax of the _asm directive:

(1) _asm *asm-statement*

(2) _asm {
asm-statement-1
asm-statement-2
asm-statement-3
... ...
}

```
void TransBuff(  char *Buff,
                 unsigned count,
                 char  echar)
{
   _asm {
          mov  esi, Buff
          mov  ecx, count
          mov  al, echar
        LB1:
          xor  [esi], al
          inc  esi
          loop LB1
    }
}
```

```
#define LEDPort    0xFF5E
unsigned char setLedMask
(unsigned char newMask)
{
  unsigned char  oldMask;

  asm {
        mov     dx, LEDPort
        /* Save current state of  */
        /* the LEDs               */
        in      al, dx
        mov     oldMask, al
```

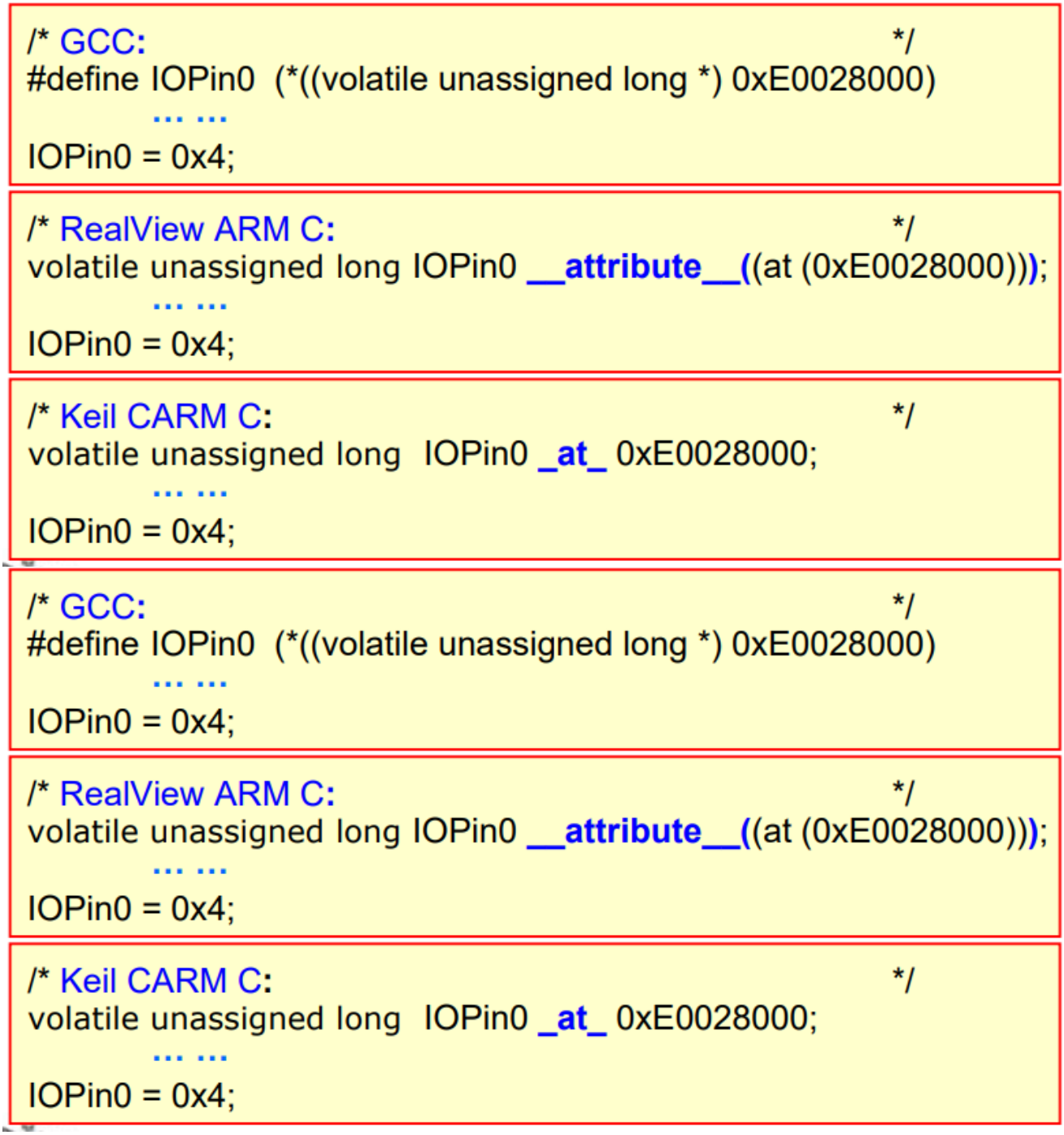
```
        /* Load the new mask */
        mov     al, newMask
        /* Modify state of LEDs */
        out     dx, al
    }
  return (oldMask);
}
```

```
void an_asm_function( )
{
  asm("   bp0:
        # housekeeping
          pushf
          pushl %eax
        # copy original flags
          pushf
          popl  %eax
        bp1:
        # flip all bits in that copy
          xorl  $0xffffffff, %eax
```

```
        bp2:
        # write bit-flipped flags
          pushl %eax
          popf
        bp3: ");
}
void main( )
{
     an_asm_function();
     exit(0);
}
```

- Common in embedded system programming

- Achieved via:
  - Pointer dereference capability of C
    - Standard feature of C
    - Possible issue: virtual address vs. physical address
    - Might need operating system support
  - Compiler implementation dependent feature

- 

- Writing Interrupt Service Routine (ISR) in C

- Common in embedded system programming

- Compiler dependent feature:
  - Non-standard keyword interrupt as type specifier used in part of function declaration
  - Compiler-specific keyword `__attribute__` (GCC)
  - Processor-specific prama – #pragma interrupt

- The Implementation of compiler:
  - Generate processor-specific entry/exit code of an ISR
  - Protect the ISR function from incorrect invocation

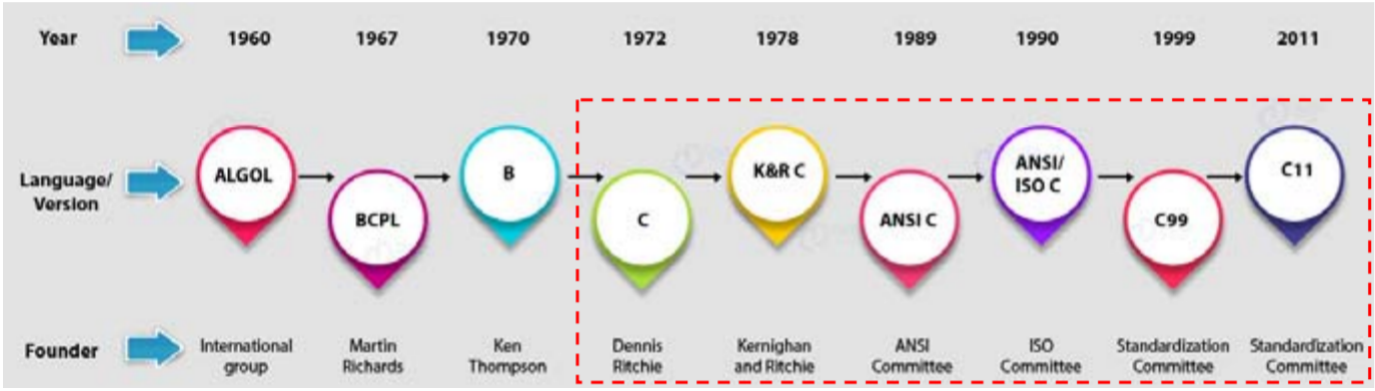
- Programmer needs to set up the vector

```
interrupt void  ISR_Vect1 (void) {
        /* code processing the interrupt */
}
```

```
void  _interrupt  ISR_Vect1 (void) {
        /* code processing the interrupt */
}
```

```
void  __attribute__ ((interrupt)) ISR_Vect1 (void) {
        /* code processing the interrupt */
}
```

3. C is evolving



- (1) New features of C99 (ISO/IEC 9899:1999)

- Single line // comments
- Variable length arrays (VLA) and flexible array members
- Mixed declarations and code
- New block scopes for selection and iteration statements
- Inline functions
- Restricted pointers
- Complex (and imaginary) support in <complex.h>
- boolean type in <stdbool.h> and other new data types
- ... more (Total 53? new features)

> Ref - http://www.open (http://www.open)-std.org/jtc1/sc22/wg14/www/newinc9x.htm (http://std.org/jtc1/sc22/wg14/www/newinc9x.htm)
> https://gcc.gnu.org/c99status.html (https://gcc.gnu.org/c99status.html)

- (2) New features of C11 (ISO/IEC 9899:2011) from C99
  - Alignment specification supported in <stdalign.h>
  - Multi-threading support (<threads.h> and <stdatomic.h >)
  - Improved Unicode support (char16_t and char32_t)
  - Type-generic expressions using _Generic keyword
  - Bounds checking interfaces and analyzability features
  - Anonymous structures and unions (useful in nesting)
  - Static assertions
  - ... more

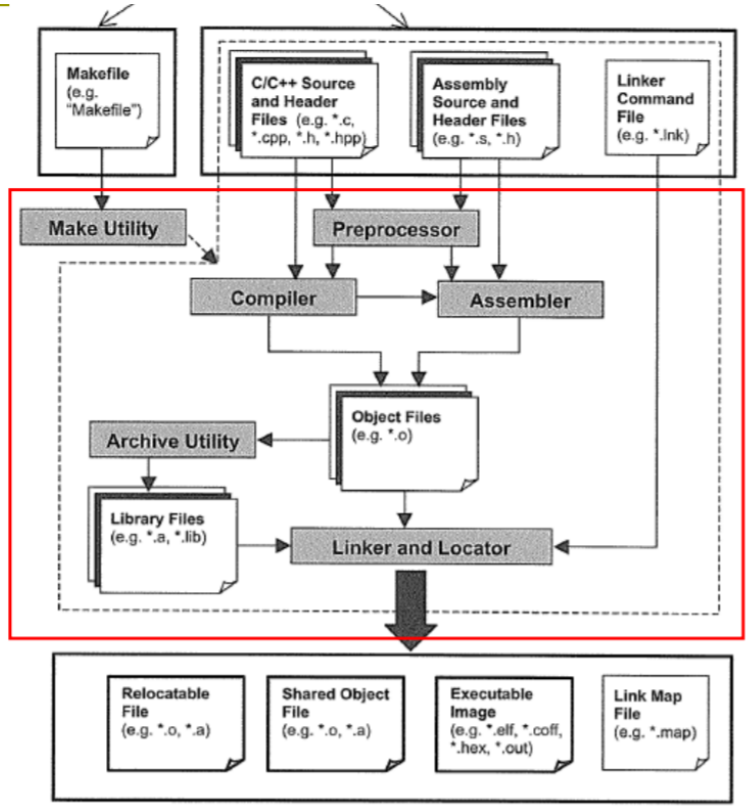
> Usage in GCC: gcc -std=c11

- (3) Extensions to C99 (ISO/IEC 9899:1999) supporting embedded processors
  - Data types for bitwise data units
  - Fixed-point number arithmetic
  - Named address spaces
  - Named register storage classes
  - Basic I/O hardware addressing
  - Vendor-specific compiler supports exist

E. Program Building Tools Program Building Tools

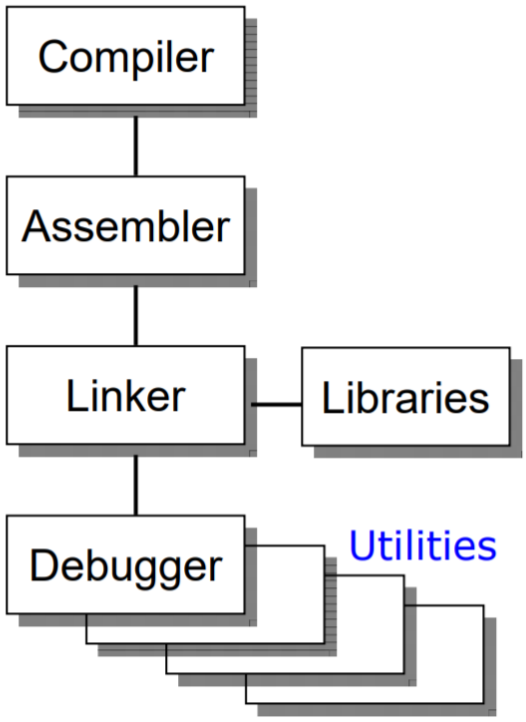
Basic Tools for Embedded System Programming:

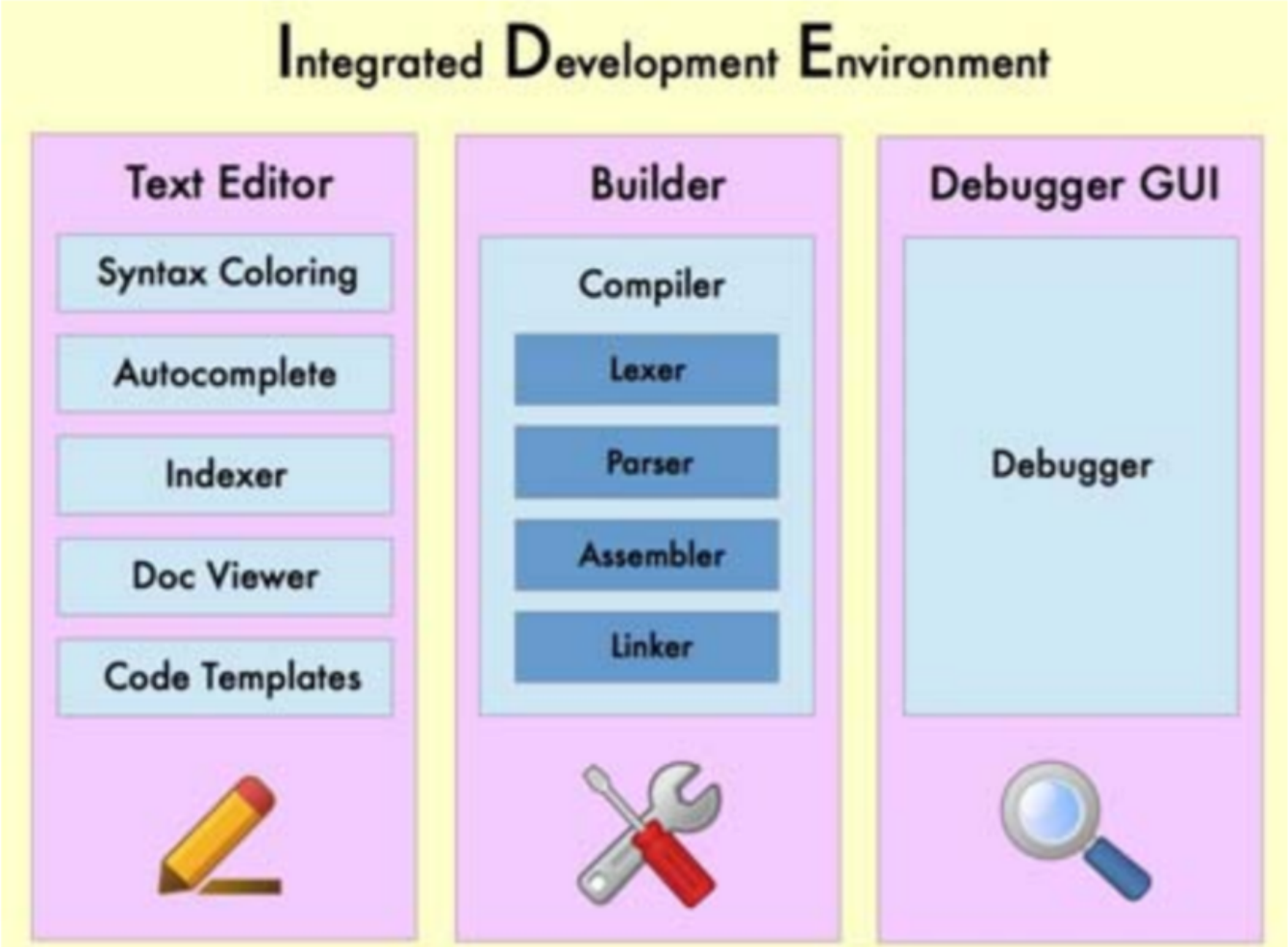
- **Make** utility
- Language processor
  - Preprocessor
  - Assembler
  - Compiler
- Linker
- Locator



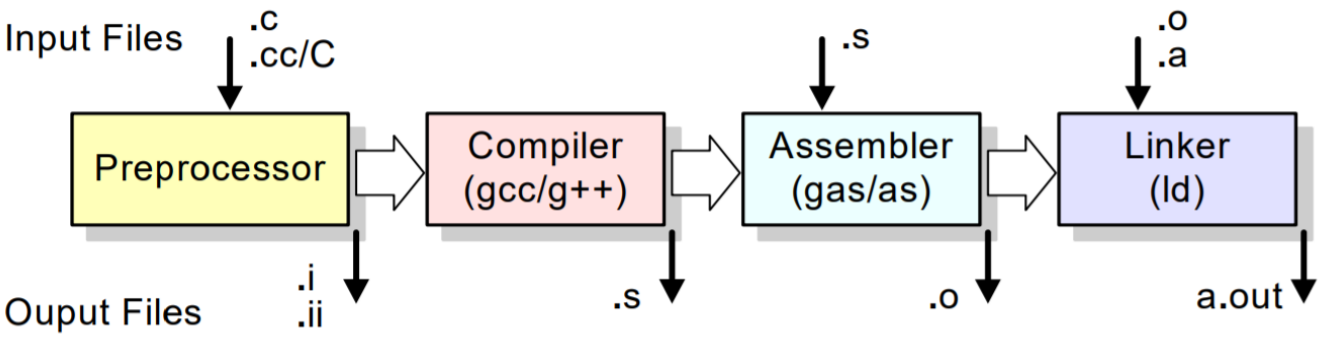
The Development Tools:

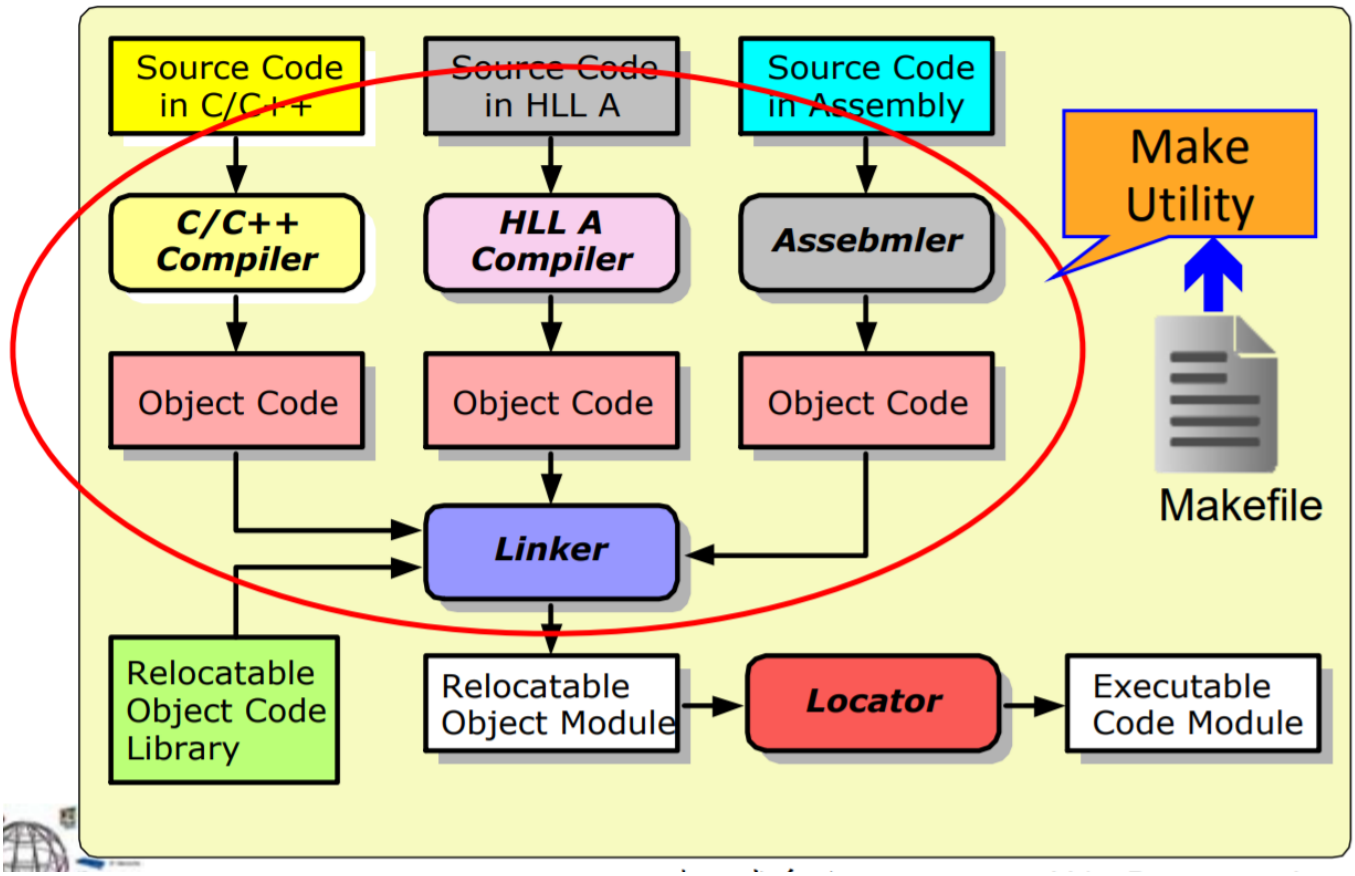
- Basic tools
  - Compiler/Assembler
  - Run-time Libraries
  - Linker
  - Debugger
  - Utilities
- **Toolchain**
  - A set of tools used in a serial (chained) manner
  - Examples –
    - GNU Toolchain
    - devkitARM





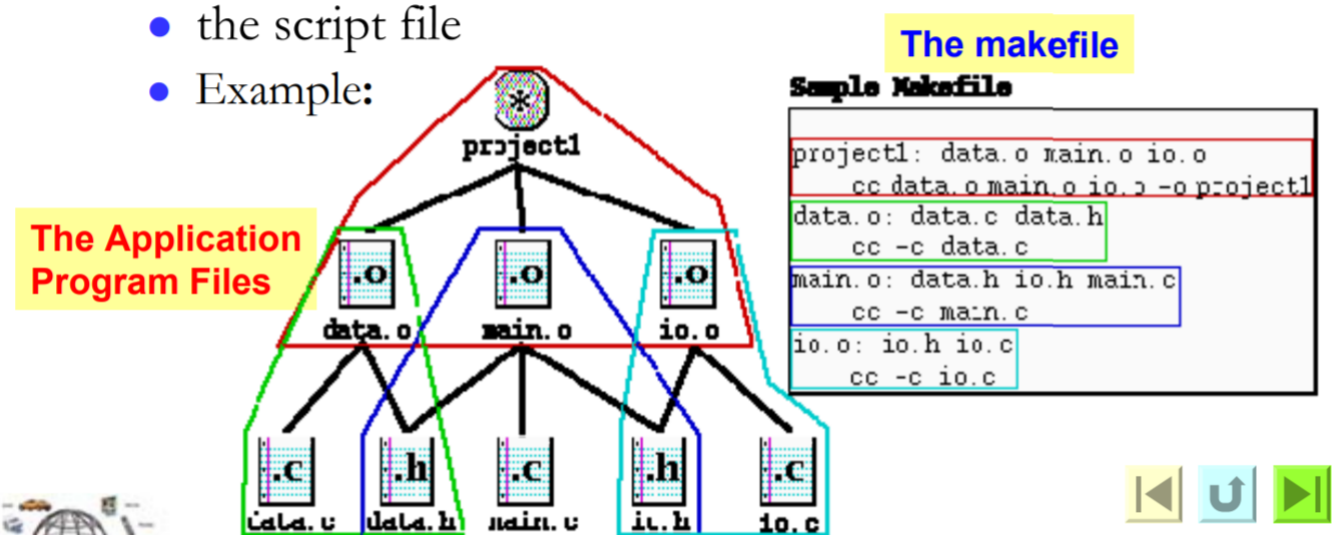
- Example: GCC (GNU Compile Collection)





1. Introduction to Make Utility

- The makefile:
  - a text file consisting of *make* directives
  - the script file
  - Example:

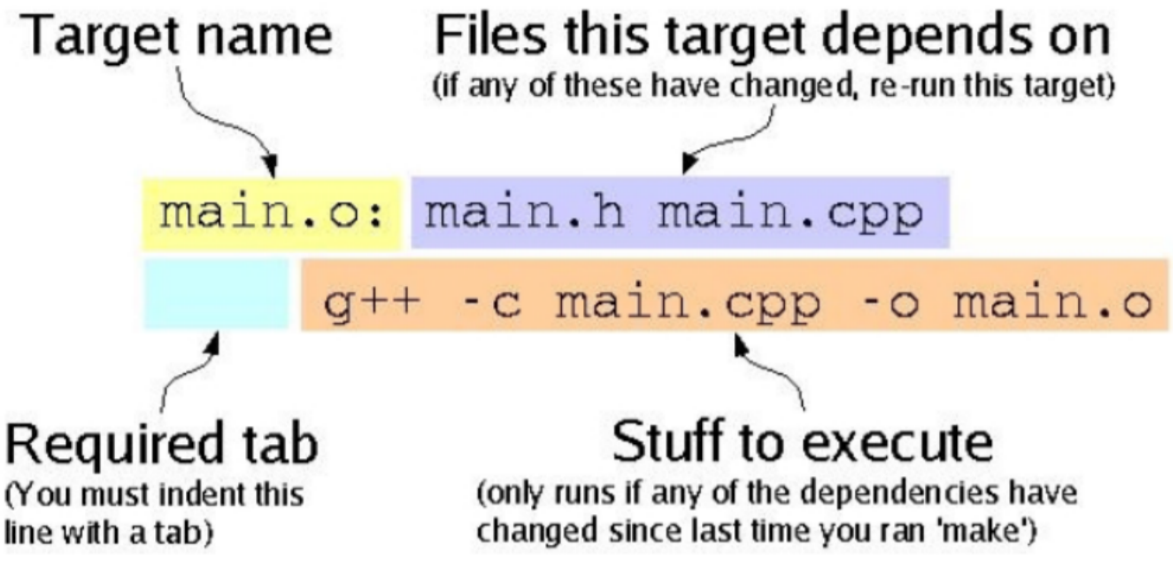


- Anatomy of makefile:
  - A simple text file of ***rules***
    - Target: name of file or action
    - Pre-requisite: file name (used as input to create the target)
    - Recipe: action to be carried out
  - Composition:
    - variable/macro definitions
    - dependency rules
    - directives
    - comments
    - shell commands

Rules in a *makefile*:
Target: *Pre-requisite ...*
<Tab>Recipe
<Tab>Recipe
...

A <**TAB**> character (0x09) must be put at the beginning of each recipe line

- Specifying dependency rules



Example of *Makefile*: (1/2)

changelog | MAKEFILE | SOURCES | bigclock.ini | Makefile

```
1  OUTPUT_NAME=lab
2
3  PROJECT_PATH=./
4  COMMON_PATH=./common
5  INCLUDE_PATH1=$(PROJECT_PATH)
6  INCLUDE_PATH2=$(COMMON_PATH)/driver
7  INCLUDE_PATH3=$(COMMON_PATH)/gnu/include
8  VPATH = $(PROJECT_PATH):$(COMMON_PATH)
9
10 #
11 # Include the make variables (CC, etc...)
12 #
13 PREFIX=/usr/local
14 CROSS_COMPILE = arm-elf-
15 CC=$(PREFIX)/bin/$(CROSS_COMPILE)gcc
16 LD=$(PREFIX)/bin/$(CROSS_COMPILE)ld
17
18 DBGFLAGS= -g
19 OPTFLAGS= -O0
20 AFLAGS_DEBUG := -Wa,-gstabs
21
22 CFLAGS= -nostartfiles $(DBGFLAGS) $(OPTFLAGS) -I$(INCLUDE_PATH1) -I$(INCLUDE_PATH2) -I$(INCLUDE_PATH3) -I.
23 AFLAGS=-mapcs-32 -msoft-float -mno-fpu  $(AFLAGS_DEBUG) $(CFLAGS)
24 LFLAGS= -Wl,-M,-Map=$(OUTPUT_NAME).map
25
26 PROJECT_OBJS =  main.o  lcd.o stdin_out.o
27 GNU_OBJS     =  gnu/src/sbrk.o  gnu/src/mmulib.o gnu/src/startup.o gnu/src/low-level-init.o
28 REUSE_OBJS   =  driver/irq/src/irq.o driver/common/src/mmu.o driver/common/src/creator_io.o
29 OBJS         = $(PROJECT_OBJS) $(GNU_OBJS) $(REUSE_OBJS)
30
31 all : $(OBJS)
32     $(CC)  -T$(OUTPUT_NAME).ld $(LFLAGS) -o "$(OUTPUT_NAME).axf" $(notdir $(OBJS)) $(LIBMODULES)
33     arm-elf-objcopy -O binary -S $(OUTPUT_NAME).axf $(OUTPUT_NAME).bin
34
35
36 %.o:    %.S
37     $(CC) $(AFLAGS) -c -o $(notdir $@) $<
38 %.o:    %.c
39     $(CC) $(CFLAGS) -c -o $(notdir $@) $<
40
41
```

Makefile | length : 1299 lines : 46 | Ln : 1 Col : 1 Sel : 0 | Dos\Windows | ANSI

Variable Settings

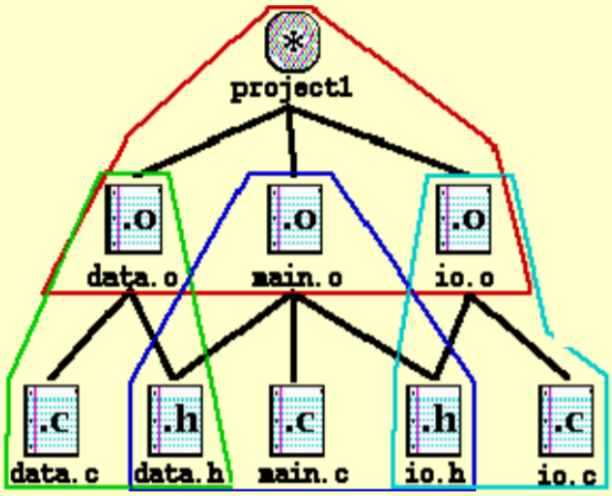
Targets and Rules/Recipes

2. Make

- The makefile (make script file)
  - a text file consisting of ***make*** directives
  - the script file
  - contents
    - Definitions of variables (names, macros)
      ex: **CC = gcc**
      DBG_FLAG = -d
    - Rules
      ex: **helloworld: helloworld.o**
      <TAB>cc -o helloworld helloworld.o
    - Comments - text begins with "#"
    - Inclusion of other **makefile(s)**

(1) The makefile: A script file of dependency rules

```
project1: data.o  io.o  main.o
    cc  -o project1  data.o  io.o  main.o
main.o: data.h  io.h  main.c
    cc  -c main.c
data.o: data.h  data.c
    cc  -c data.c
io.o: io.h  io.c
    cc  -c io.c
.PHONY: clean
clean:
    rm  main.o  data.o  io.o
```



(2) The makefile: Dependency rules with macros

```
OBJS = main.o data.o io.o
DBG_FLAG =          # assign  -g  when debugging
project1: $(OBJS)
    $(CC) $(DBG_FLAG)  -o $(OBJS)
main.o: data.h  io.h  main.c
    cc  -c main.c
data.o: data.h  data.c
    cc  -c data.c
io.o: io.h  io.c
    cc  -c io.c
.PHONY: clean        # This line can be omitted!
clean:
    rm  $(OBJS)
```

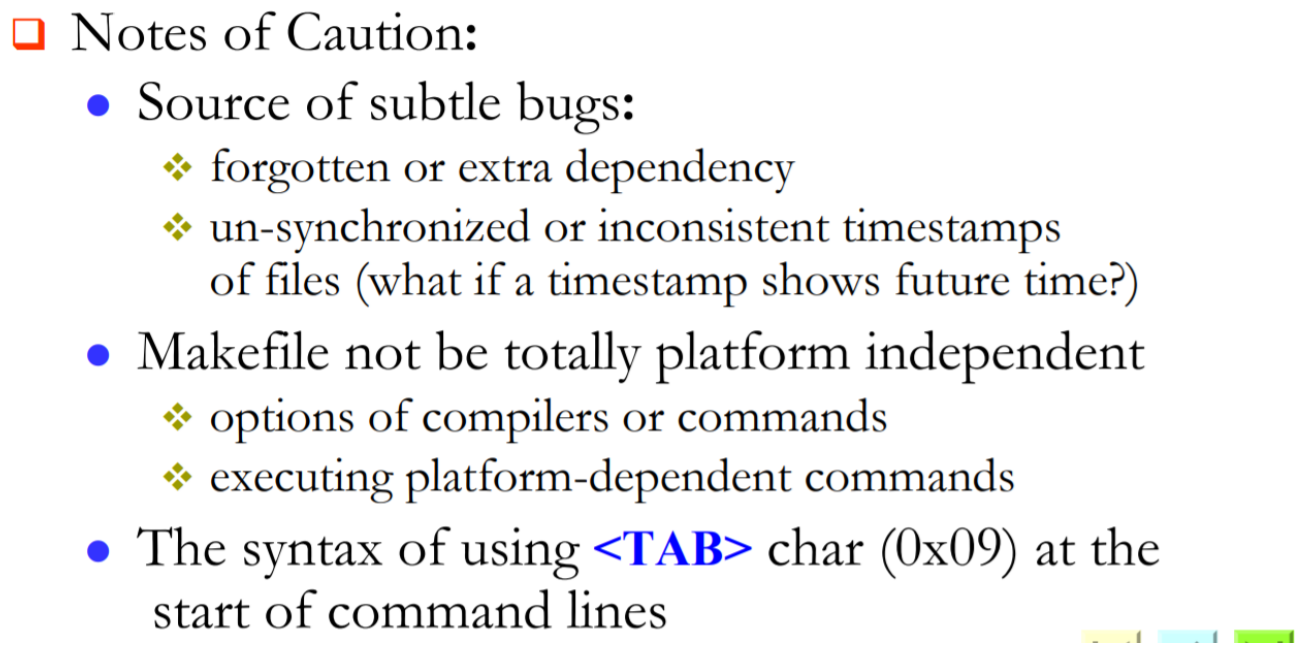
(3) Example of automatic variables in makefile:

```
HelloMake: One Two Three
    # Output is “HelloMake” because it is the 1st target
    @echo $@
    @echo $?
    @echo $^
    touch HelloMake
One:
    touch One
Two:
    touch Two
clean:
    rm  -f HelloMake One Two Three
```

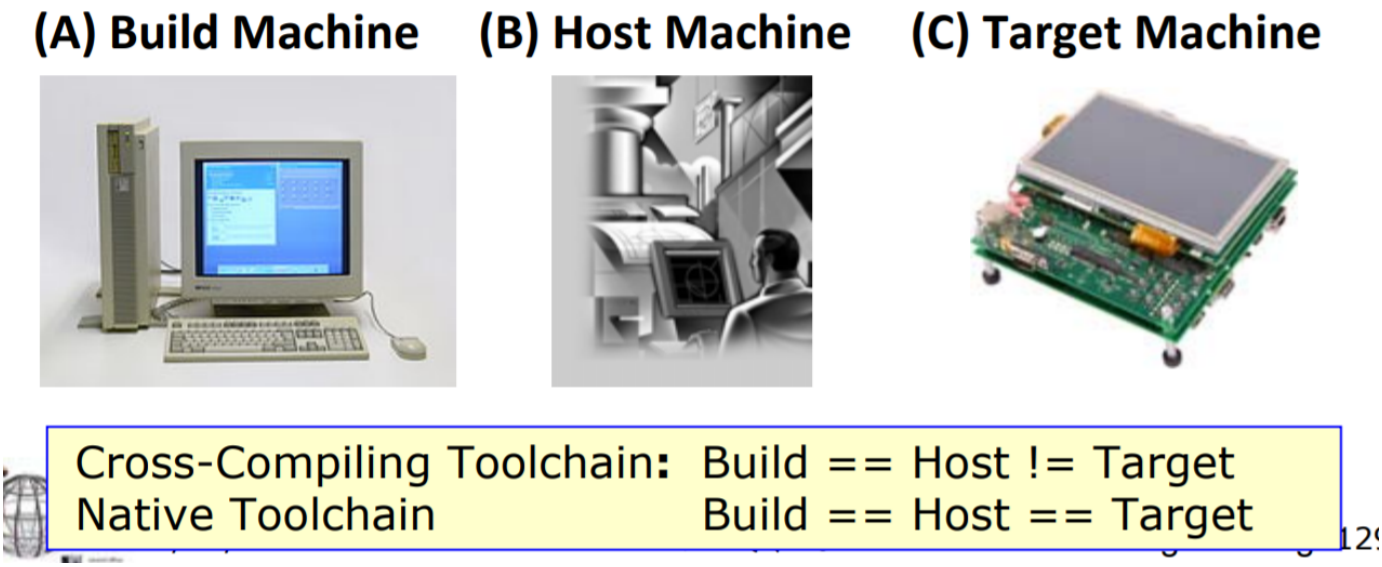
General rules and notes:

- Building the 1st target by default
- Phony target: target with no dependency rules
- Frequently used file name of input: Makefile
- May include other files: include file1 file2 ...
- Each command is executed by separate shell
  - The shell used by default is /bin/sh
  - The shell used can be changed inside the makefile

    - Selecting the shell: SHELL=/bin/bash
- Each line with commands must start with <TAB>
- Multiple commands can be in the same line
    - separated by ';'
    - executed in the same invoked shell
- make utility stops when an error occurs
- Non-Stop Execution of make
    - Errors are ignored
    - prefix command with "-", ex: -rm oldfiles
    - other options: -i switch, .IGNORE
- An empty line that begins with <TAB> is an empty command
- Variables in makefiles may be overridden by giving the values as command-line arguments
- Example: make DBG_FLAG=-d



F. Cross-Compiling Toolchain



Difference:

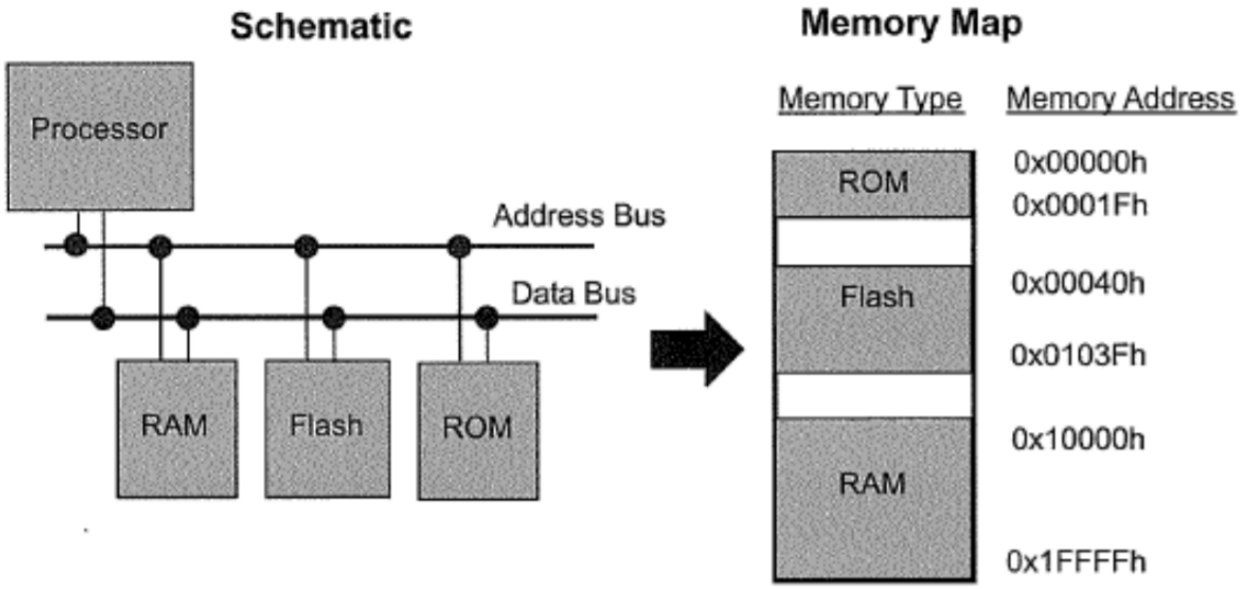
- CPU Architecture (ISA)
- ABI (Application Binary Interface)
  - Source code compatible vs. Binary code compatible
- Operating System
- Run-time Library 程式執行的支援

Core components in Linux cross-compiling toolchain:

- GCC
- C library
- Binutil
  - Binary Utility 二進位機器碼工具組，像是Assembler
- Linux kernel headers
  - 存放在特定的資料夾，像是IO的控制程式，Linux Loadable Kernel Module

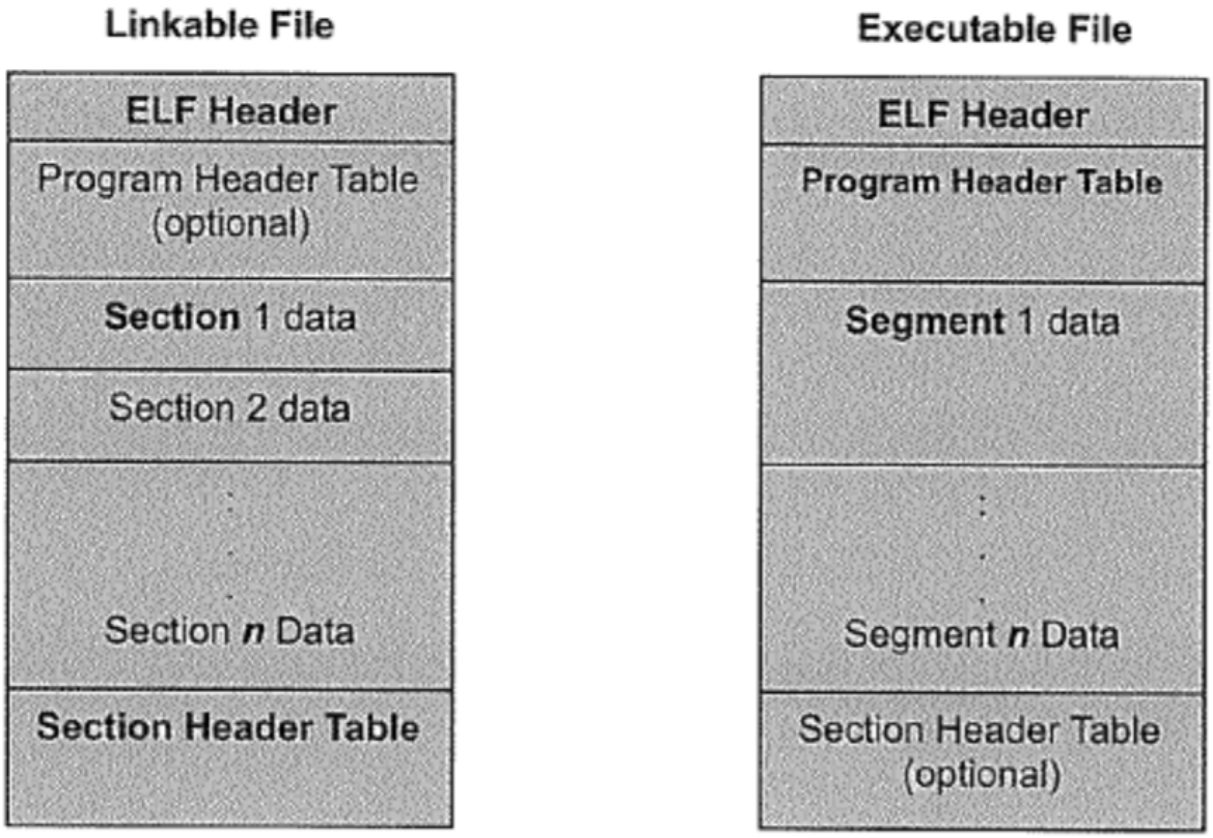
G. Memory Map

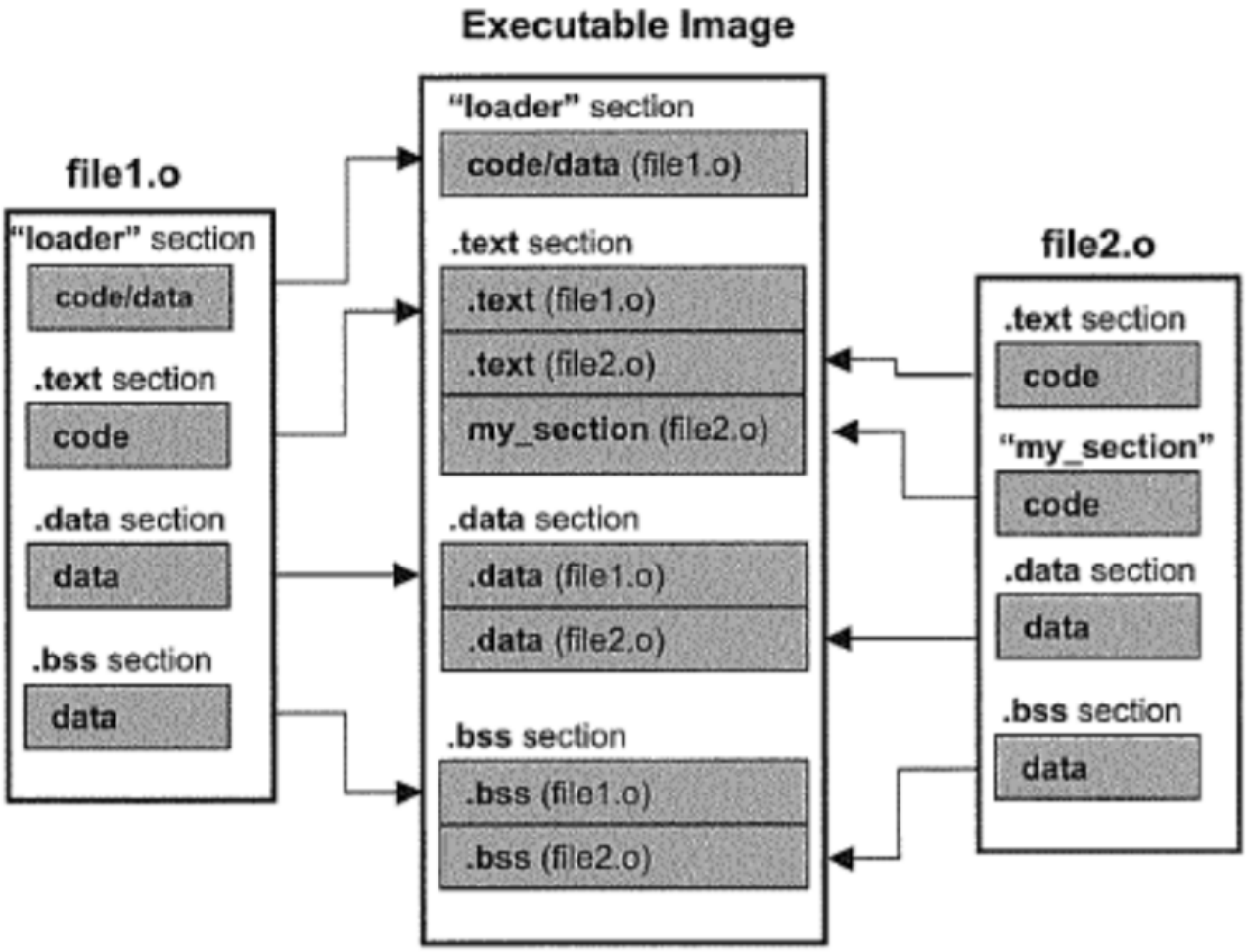
Memory Map of an Example Target (simplified)



ROM不能修改，Flash是EEPROM的概念，可以修改

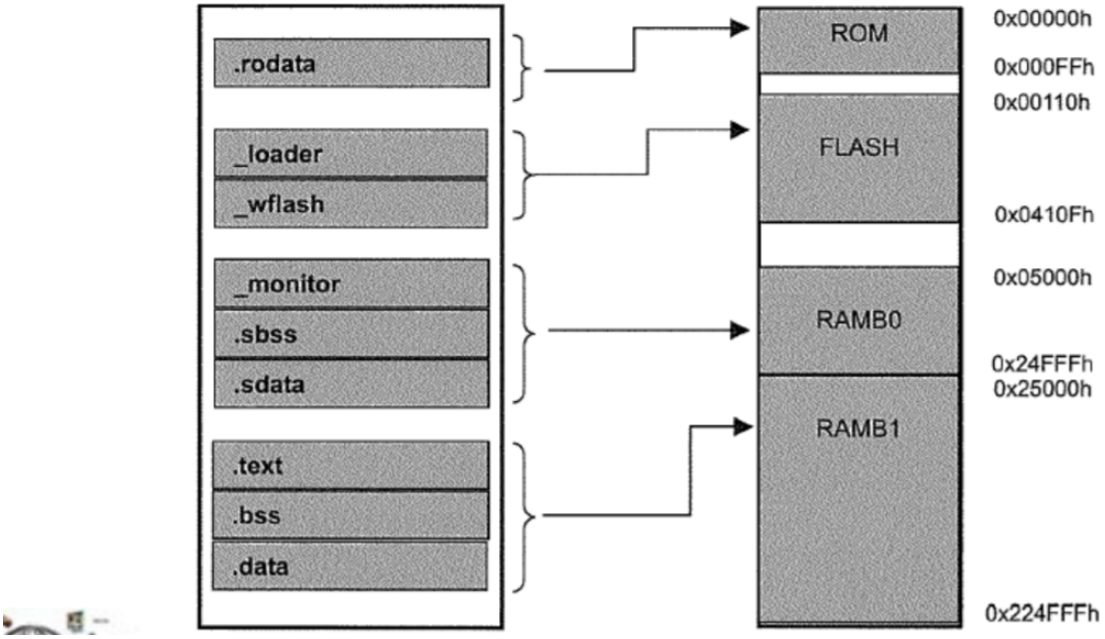
ELF:Executalbe Linkable File

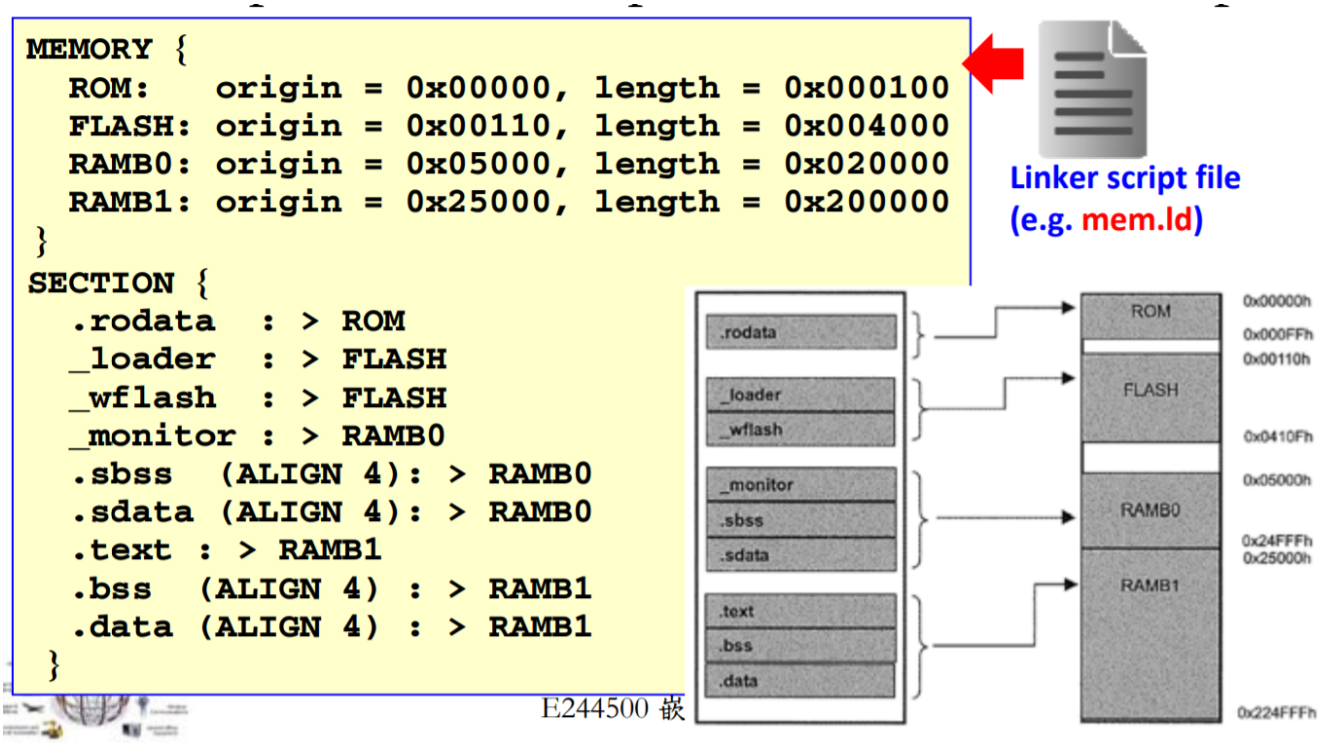




在嵌入式系統開發link不一定能自動做

Mapping Executable Image into Target Memory





Memory bus的寬度是4個bytes

II. 基本程式開發技術

A. Basic Computer Programming

Techniques in Program Development:

- Top-down design vs. bottom-up design
- Function decomposition and modularization
- Structured programming
- Information hiding and encapsulation
- Stepwise refinement
- Mixed language programming
- Object-oriented (OO) programming
- ...

B. Structured Programming Techniques

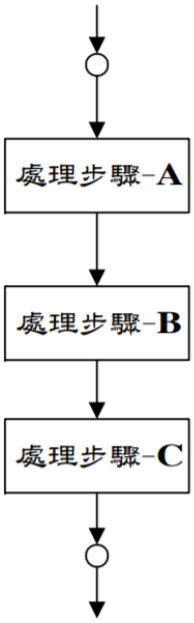
要點：

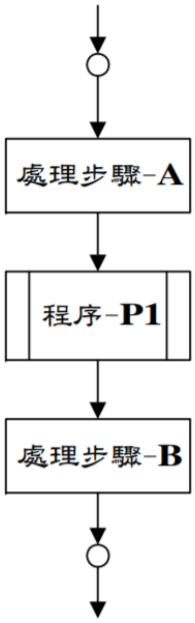
- 程式設計模組化
- 各模組僅有單一入口與單一出口
- 適度控制各模組之規模與複雜度
- 避免使用無條件之 GOTO 敘述

- 使用基本之程式執行控制結構
- 維持程式模組之一貫性

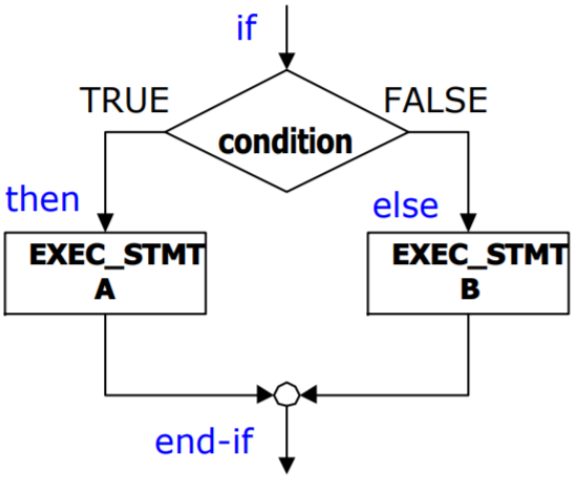
❑ **Sequences –**

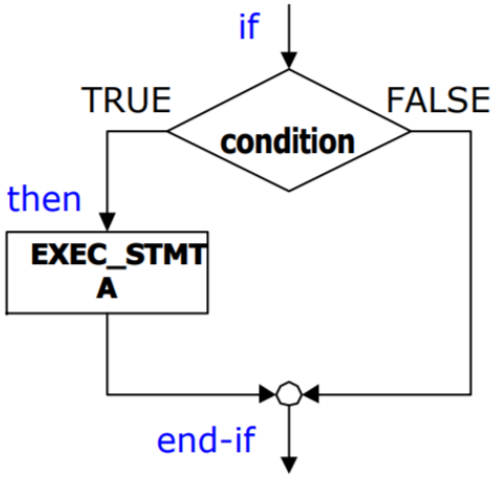
- Default flow control
- Implicitly enforced by system architecture
- Controlled by the program counter



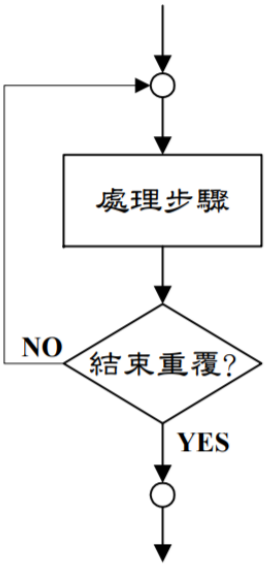


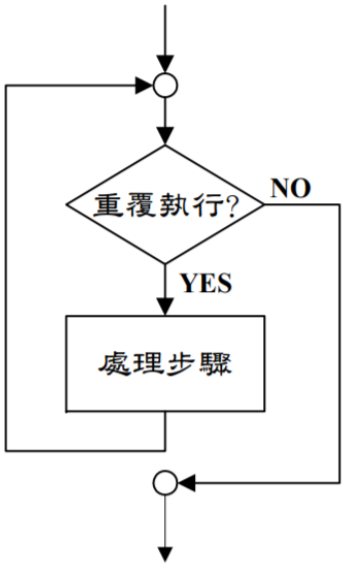
Selections – if/else, switch, case



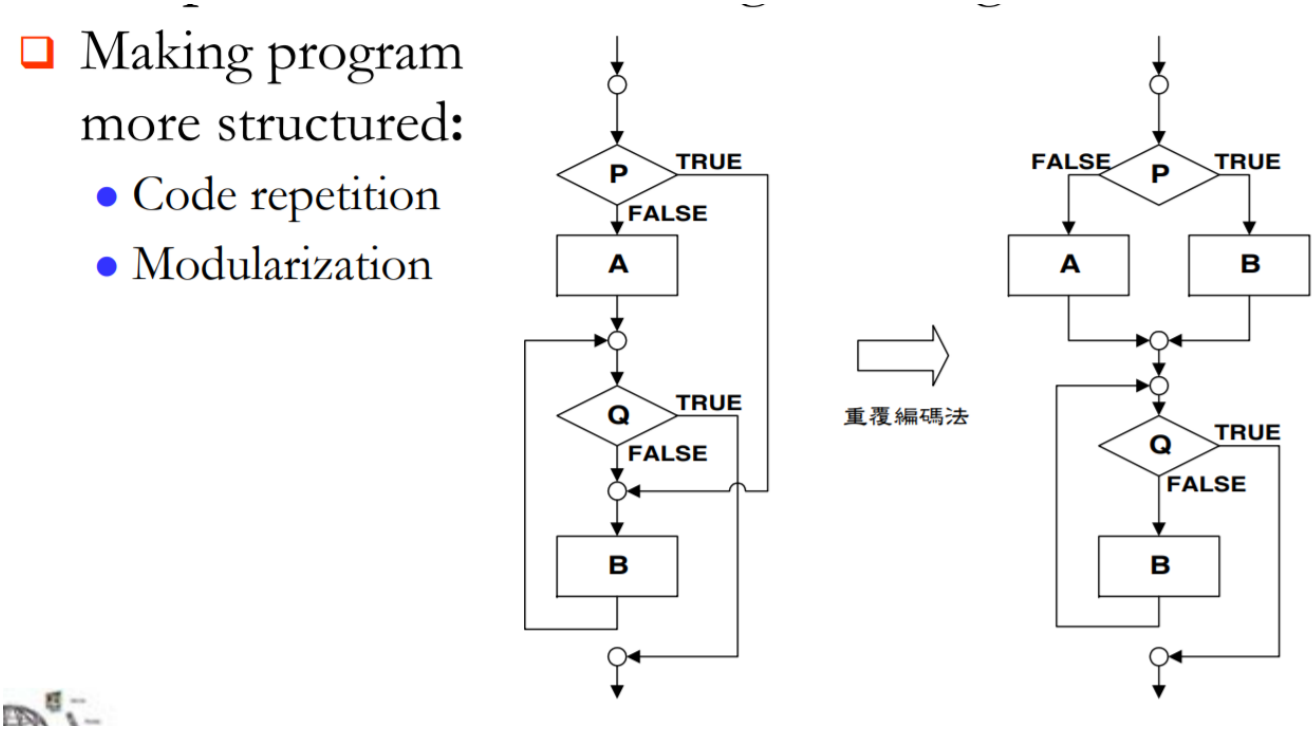


Repetitions – while, do/while, repeat/until, for





Example of Structured Programming:



C. Efficiency and Optimization

- Program efficiency
  - The metrics:
    - The execution time (worst case, average case)
    - The storage used (memory) – footprint
    - ...
  - Might be part of non-functional requirement
- Program Optimization
  - To improve program efficiency:
    - Program execution time
    - Program memory
    - Power/Energy consumption

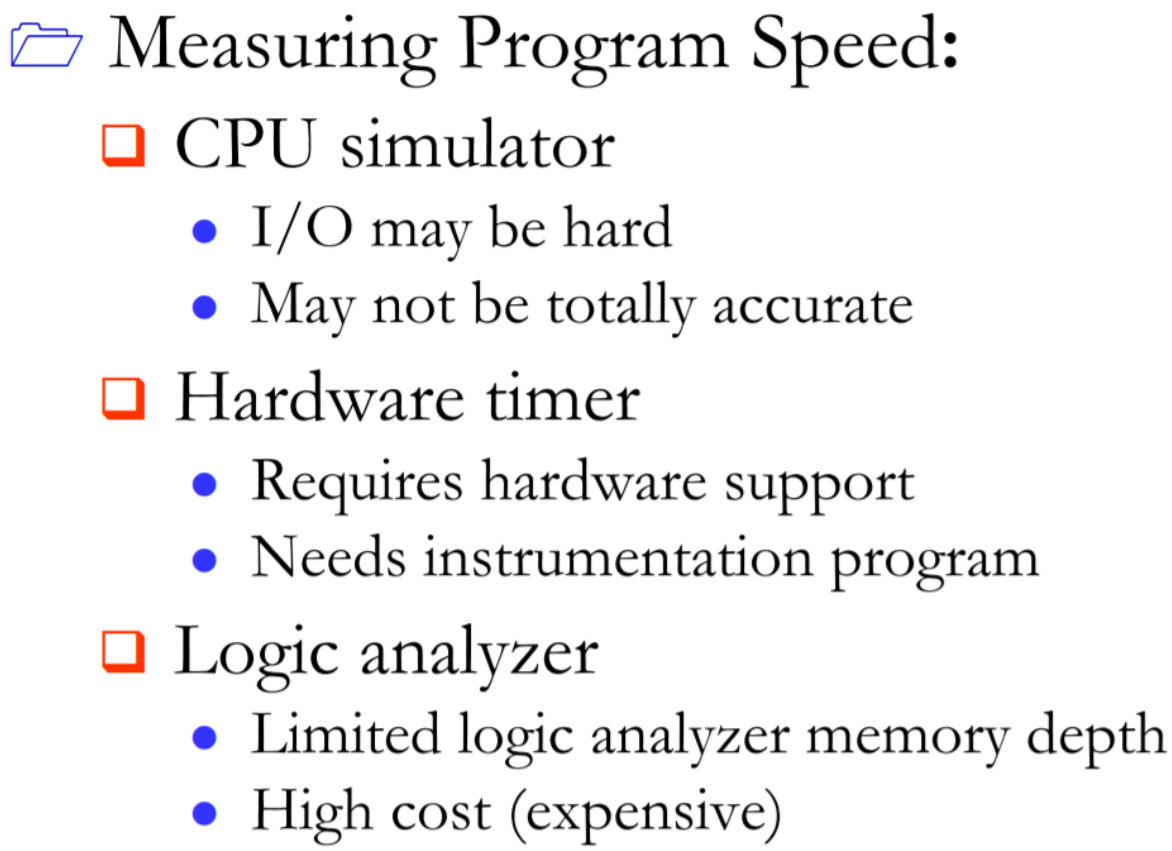
注意，演算法所說的時間複雜度主要是說資料量大的時候，對於嵌入式系統而言較不適用，所以程式效率就是直接看執行時間，執行效率被分類在非功能性需求

Program execution times depend on:

- CPU processing speed
- Architecture effects:
  - pipelining
  - superscalar
  - ...
- State of the (instruction and data) caches
- Input data values

- Software factors: language and translator
- Programming skill

測量程式執行速度



(1) 用Simulator不是花多少時間執行你的程式，而是要本身提供分析的功能
(2) Timer就像是體育場上用電子計時，非人工計時，instrmentation program 就是開頭與結尾加上時間ticks
(3) 程式執行經過的指令都會記錄起來

D. Program Performance

- Elements of program performance (Shaw):

  T = P + I

  T: Execution time,
  P: Program path,
  I : Instruction timing

- Path depends on data values. Choose the case you are interested in.
- Instruction timing depends on
  - CPU speed
  - Pipelining
  - Cache behavior

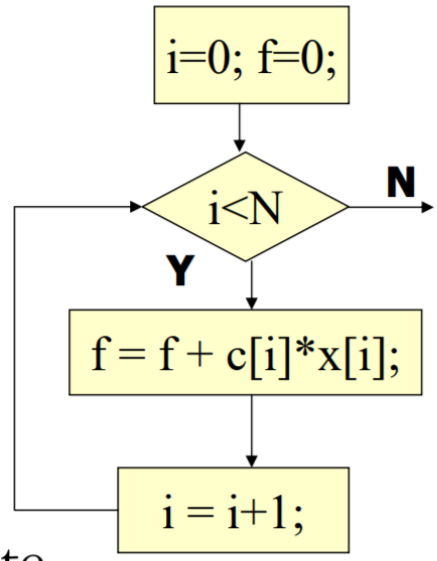
Performance Path

Performance Path

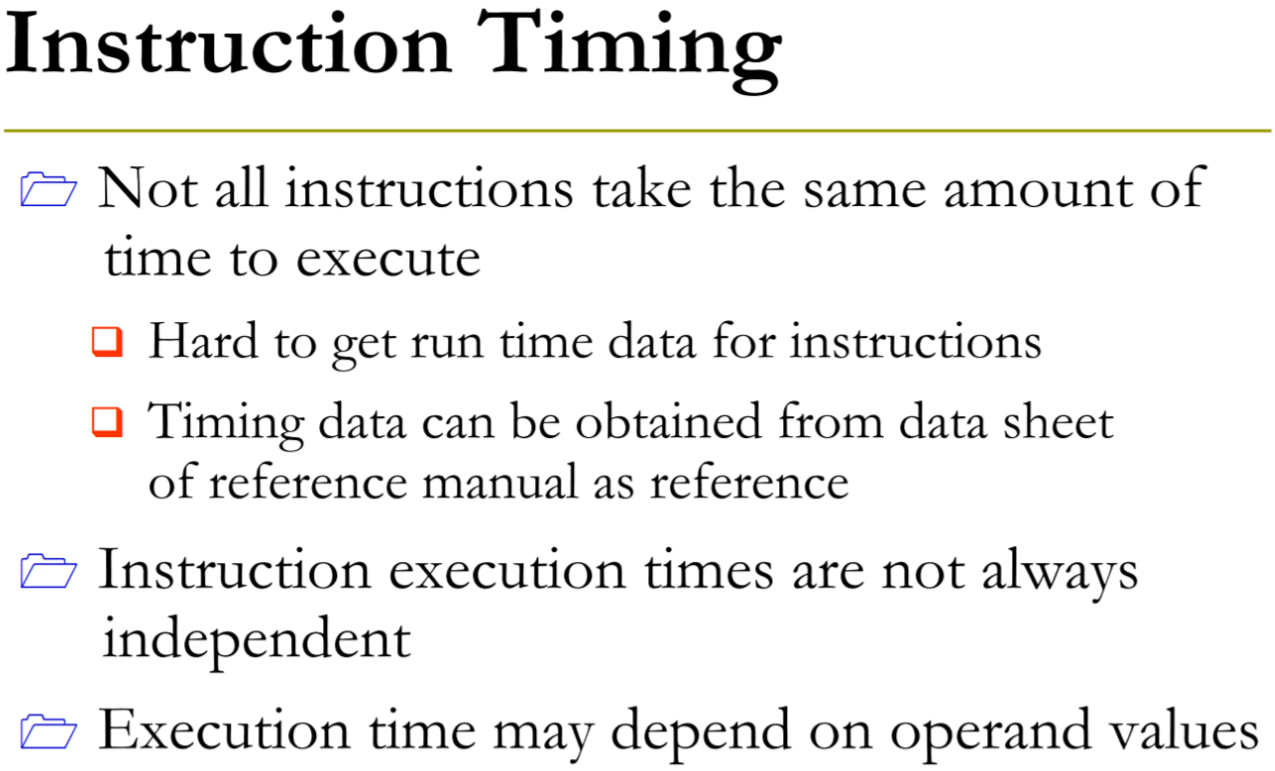
- Consider for loop:

  for (i=0, f=0; i<N; i++)
  f = f + c[i]*x[i];

- Loop initiation block is executed once.
- Loop test is executed **(N+1)** times.
- Loop body and variable update are executed **N** times.



Instruction Timing



指令的執行時間並非獨立，可以用統計的方式，多執行幾次

Best results come from analyzing optimized instructions, not high-level language code:

- non-obvious translations of HLL statements into instructions
- code may move
- cache effects are hard to predict
  程式執行的stack要多大?stack overflow不可忽視，會影響答案

E. Trace

Trace: A record of the execution path of a program

- Trace gives execution path for analyzing performance
- A useful trace:
- requires proper input values
- is large (gigabytes)

Hardware capture:

- logic analyzer
- hardware assist in CPU

Software: 軟體可以減少成本

- PC (Program Counter) sampling 紀錄通過那些點就好，並不用記錄所有指令，profiling則是計算點被經過的次數，越多次數的點就是熱點

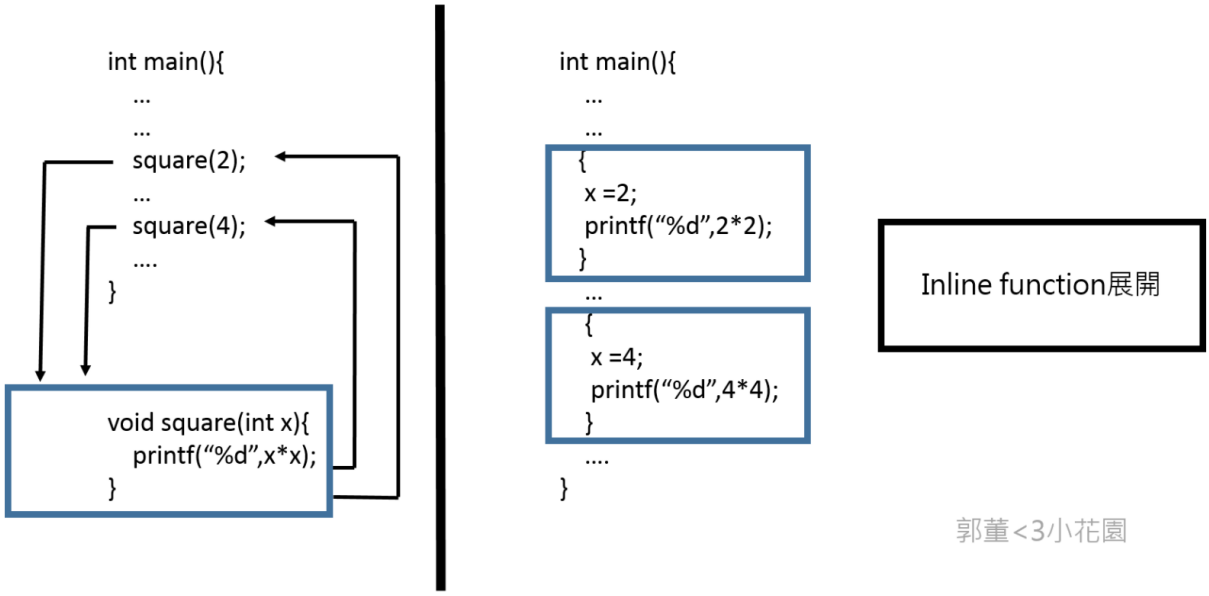
- Instrumentation instructions
- Simulation

為改善程式的speed, storage, power comsumption可以執行的方法有(1) 了解compiler (2) 優化你的code

F. Optimization by translators

Optimization on generated code

- Register allocation
- Common sub-expression elimination 消除共同計算式
- Jump optimization 把要jump過去的程式直接搬到附近，把目的地搬過來
- Loop optimization 減少loop的次數，通常是unrolling或是把一些變數拿出來
- Strength reduction 用一些簡單指令達到效果，*2 == <<2
- Constant folding/propagation 2*3 直接先做成 6
- Procedure inlining 呼叫函式時，需要準備參數，把參數存到stack裡面，再呼叫函式執行，執行後還要return回來，再清除stack，這些都是overhead



- Instruction selection 選怎樣的機器指令可以最有效率

Compiler需要做到什麼樣的程度?
Controlled by setting the levels of optimization

Example of Optimization by Compiler:

- GNU Compiler
  - general optimization level: 0 ~ 3
  - individual options for specific types of optimizatic
  - command-line switch: -Olevel (level: 0 ~ 3)
  - -O0 : eqv. “no -O setting” (default)
  - -O1 : eqv. “-O”
  - -Os : reduce the size of executable file
  - -funroll-loops
  - …

程式還在除錯階段時就不用優化

Optimization Levels

- Level of optimizations carried out by the compilers is selectable

-O0
- Minimum optimization
- The least optimized code, but with the best debug view

-O1
- Restricted optimization
- Optimized code and a reasonable debug view.

-O2 (default)
- High optimization
- Well optimized code, but with limited debug view

-O3
- More aggressive optimization, weighted toward `-Ospace` / `-Otime` choice
- Enables multifile compilation by default (more later)

ARM RealView

- Select optimization for code size or execution speed with `-Ospace` (the default) or `-Otime`
- Use the `-g` or `--debug` option to generate source level debug information

G. Optimization by Programer

Optimization by programmers

- Programming skills and algorithm
  - Structured programming
  - Modularization
- Assembly language/low-level programming
  - Examples:

```
XOR   r1, r2
XOR   r2, r1
XOR   r1, r2
```

```
CWD
XOR   AX, DX
SUB   AX, DX
```

  - r1 = (r1 ^ (r1 >> 31)) – (r1 >> 31);
- Mixed-language programming

格式：CWD

功能：將AX中16位帶符號數，進行帶符號擴展為32位，送DX和AX中。高16位送DX中，低16位送AX中。

Loops are good targets for optimization

- Generally consume much program execution time
- The **80/20** principle

Basic loop optimizations:

- Code motion
- Induction-variable elimination
- Strength reduction (x*2 ⇨ x << 1)
- Loop unrolling

```
for (i=0; i < N*M; i++)
    z[i] = a[i] + b[i];
```

```
X = N*M;
for (i=0; i < X; i++)
    z[i] = a[i] + b[i];
```

- Induction variable: loop index

- Consider the loop:

```
for (i=0; i<N; i++)
  for (j=0; j<M; j++)
      z[i][j] = b[i][j];
```

- Rather than re-compute **i*M+j** for array element in each iteration,
  - share the induction variable between arrays, and
  - do the increment at the end of loop body.
- Consider the following loop:

```
for (i=0; i<N; i++)
  for (j=0; j<M; j++)
```

```
        z[i][j] = b[i][j];
```

- Optimized versions:

(1)
```
K = N * M;
tmp_z = z;
tmp_b = b;
for (i = 0; i < K; i++) *tmp_z++ = *tmp_b++;
```

(2)
```
for (i = 0; i < N*M; i++) *(tmp_z+i) = *(tmp_b+i);
```

- Compare the following code segments: (1/2)

CODE A

```
for (i = 0; i < N; i++)
   for (j = 0; j < N; j++)
      if (i == j)  A[i][j] = 1; else A[i][j] = 0;
```

CODE B

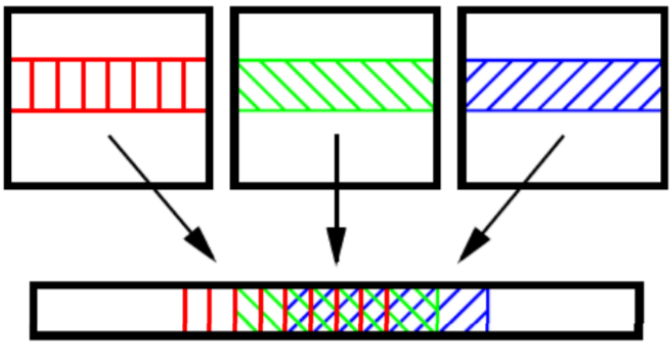
```
tmp_A = A;
for (i = 1; i <= N; i++)
   for (j = 1; j <= N; j++)
      *tmp_A++ = (i/j)*(j/i);
```

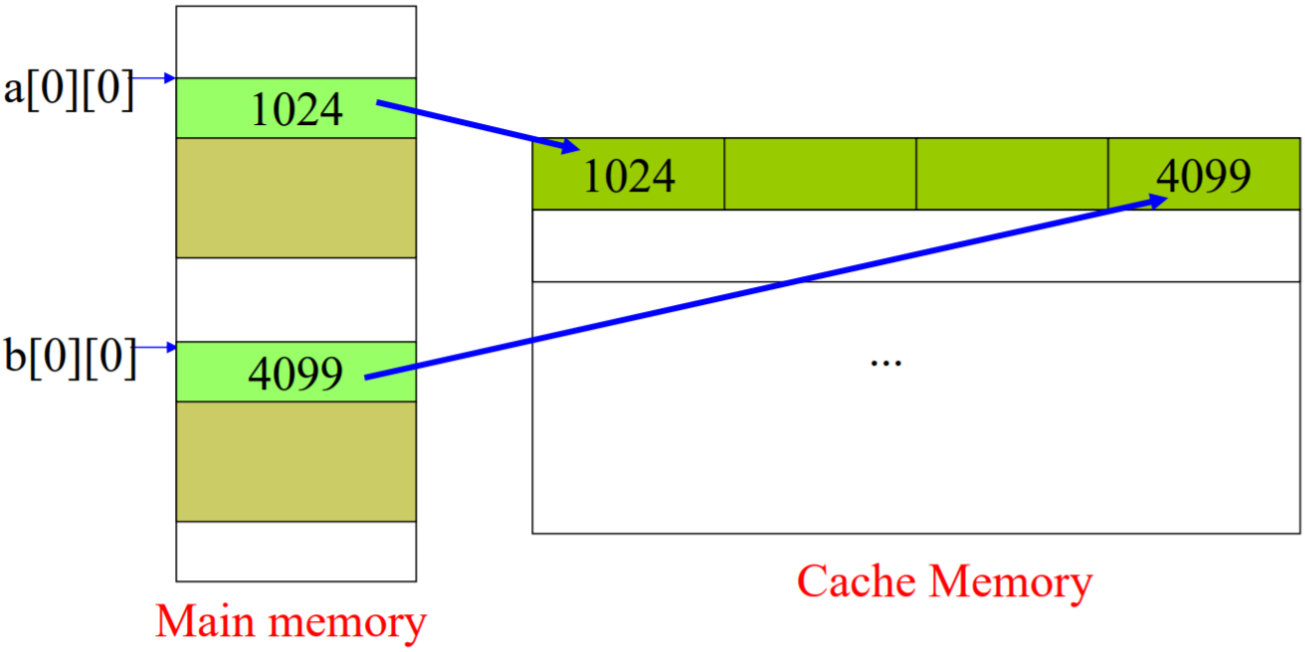
CODE C

```
temp_A = A;
K = N * N;
for (i = 0; i < K; i++)  *temp_A++ = 0;
K = N + 1;
for (i = 0, j = 0; i < N; i++, j += K)
 *(A+j) = 1;
```

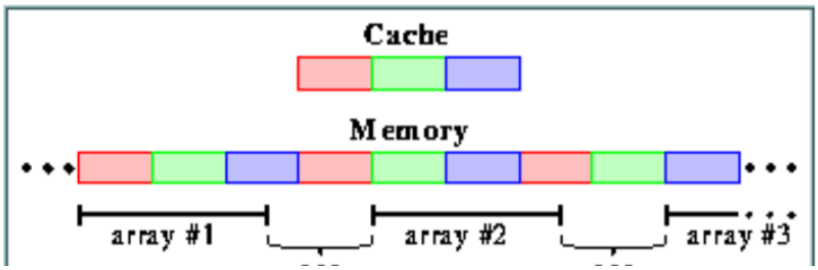
H. Cache Analysis

- Loop nest: set of loops, one inside another.
- Perfect loop nest: no conditionals in nest.
- Because loops use large quantities of data, cache conflicts are common.
- Cache conflict occurs when data mapping overlaps in cache.





- Array elements conflict as they are mapped into the same cache line, even if they are not mapped to same location.
- Solutions:
  - Move one array
  - Pad array
  - Cache partition



Example – On optimizing C/C++ Code

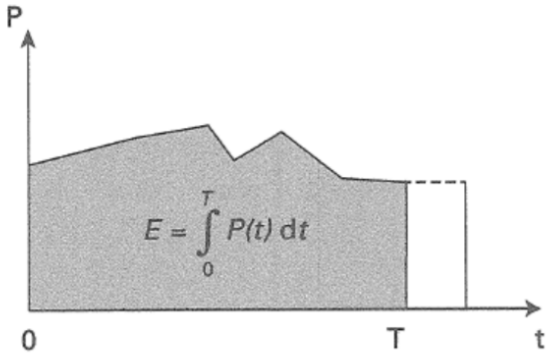
- Adjust structure sizes to power of two
- Place case labels in narrow range
- Place frequent case labels first
- Minimize local variables
- Declare local variables in the innermost scope
- Reduce the number of parameters
- …
  http://www.eventhelix.com/RealtimeMantra/Basics/
  (http://www.eventhelix.com/RealtimeMantra/Basics/)

Trade-off between execution speed & code size
Helpful tool: profiler

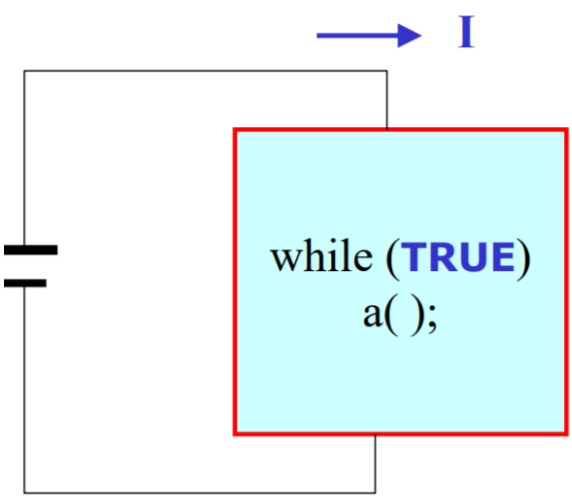
- Techniques to reduce execution time:
- Inline functions
- Table lookup
- Hand-coded assembly
- Register variables / global variables 放在stack去獲取的時間比放在memory長
- Pooling vs. Interrupt 程式效率不一定，但CPU效率Interrupt可能比較好，因為CPU有可能閒著
- Fixed-point arithmetic

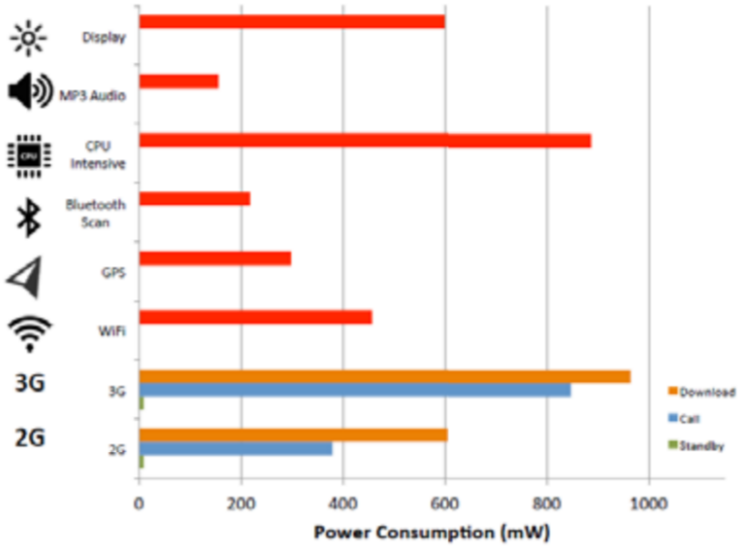
I. Energy/Power Optimization

- **Energy**: ability to do work.
  - E = P × Time
  - Most important in battery-powered systems



- **Power**: energy per unit time.
  - P = V × I = V² / R
  - Important even in wall-plug systems --- power becomes heat

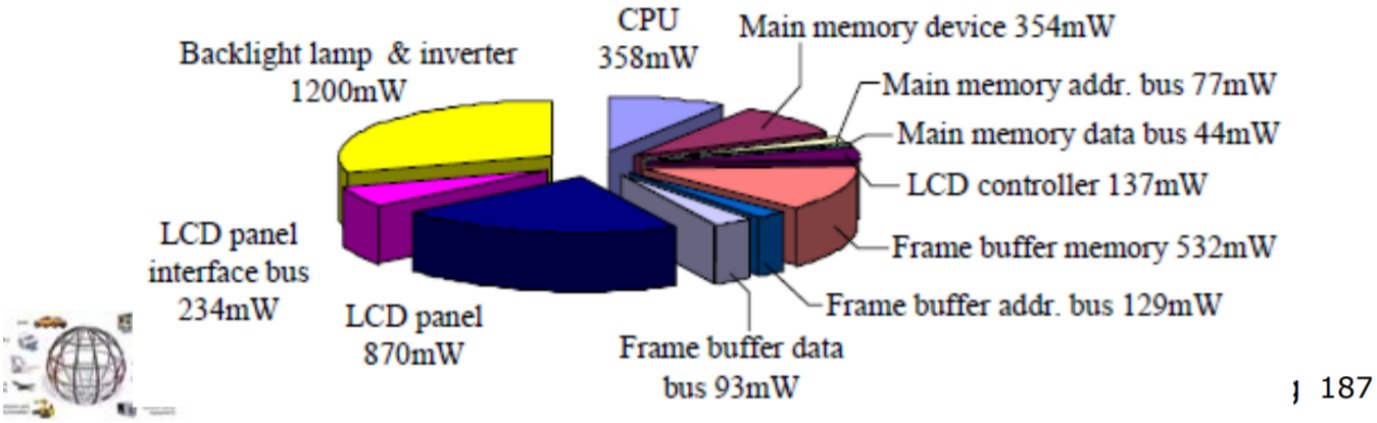




Power Consumption of a Mobile Device (2013)

- Relative energy per operation (Catthoor *et al*):
  - Addition: 1
  - Multiplication: 3.6
  - External I/O: 10
  - Memory transfer: 33
  - SRAM read: 4.4
  - SRAM write: 9

Ref: Power Consumption of System Components when an MPEG-4 Player is running



187

Energy consumption has a sweet spot as cache size changes:

- Cache too small: program thrashes, burning energy on external memory accesses
- Cache too large: cache itself burns too much power

- Use registers efficiently
- Identify and eliminate cache conflicts
- Moderate loop unrolling eliminates some loop overhead instructions
- Eliminate pipeline stalls
- Inlining procedures may help:
  - reduces linkage,
  - but may increase cache thrashing.

J. Optimizing for Program Size

Optimizing for Program Size

- Goals:
  - reduce hardware cost of memory
  - reduce power consumption of memory units
- **Program Size = Code Size + Data Size**
- Two opportunities:
  - data
  - instructions
- **Issue:** Reduce program size usually results in increasing execution time!

Minimization of Data Size

- Reuse constants, variables, data buffers in different parts of code.
  - Requires careful verification of correctness
  - Not suitable for shared data
- Generate data using instructions

Reducing Code Size

- Avoid function inlining
- Choose CPU with compact instructions
- Use specialized instructions where possible
- Select/Enable compiler switches to do the optimization on reducing code size
  - Be careful on "dead code elimination"
  - Never assume optimized code remains the same as the un-optimized code
- Reducing the code size by hand:
  - Avoid using standard library routines
    - Less general functions
    - Cygnus ***newlib***
  - Use "**native word size code**"
  - Use **goto** statement in case of complicated control structure (**but use carefully**!)
- Issues:
  - Conflicting with execution efficiency
  - Difficulty in debugging

Discussion:
Memory Address Alignment

1. Is it related to performance issue?
2. What is the (potential) impact?
3. How to solve if it imposes impact ?

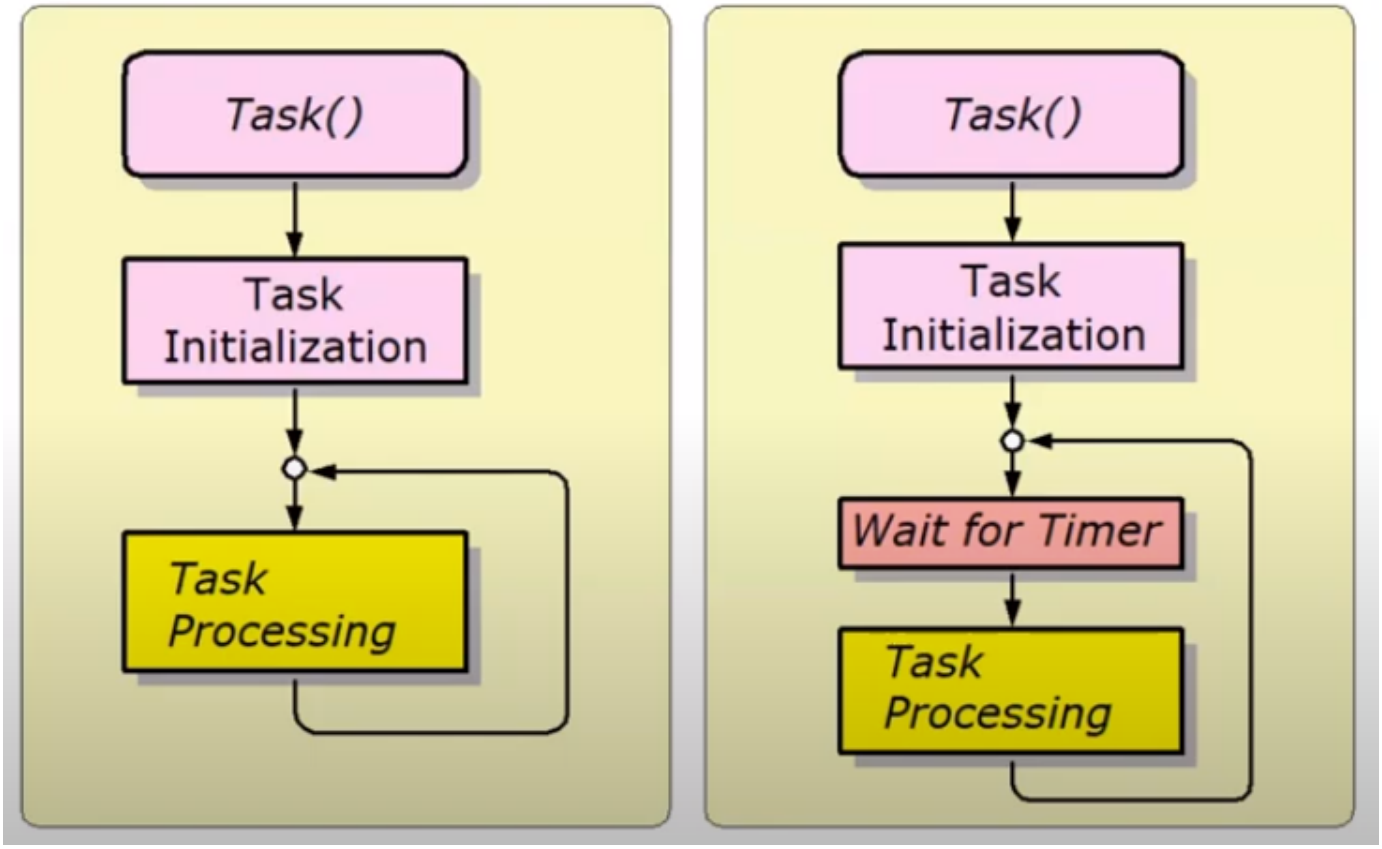
III. 嵌入式系統程式設計技術

A. 嵌入式系統程式基本控制流程

1. 嵌入式系統程式基本結構

(1) 嵌入式系統程式基本結構

a. Basic Structure



- 電源關掉後，重新開機後初始化完，就會繼續執行，跟電腦很像，因為嵌入式系統的程式是放在ROM、FLASH中
- 如果有作業系統的話，關機就不是只有關電源而已，需要在硬體上有特定的開關，讓它啟動關機程序，跟手機很像，按關機按鈕，會出現系統提示，根據提示才能關機
- 如果在wait for timer時，就有空閒時間，可以做到multi-tasking

- 例題1 : A simple task to blink LEDs

```
#include <headers.h>
void main(void) {
   char  LED_Pattern = 0x01;          // 0x5A = 00000001 B
   Task_Initialization();
   while(1) {                         // 簡易跑馬燈效果
     LED_Turn_On(LED_Pattern);        // Turn on LEDs.
     Delay(500);
     LED_Pattern =<< 1;               // new LED pattern.
    }
 }
```

- 例題2 : A simple task to blink LEDs on PXA270 EVM

```
#include <PXA270.h> (Simplified)
/***** Function prototypes ************************************/
static int  creator_system_init(void);
void Delay(unsigned int ms);
static int  process(void);
/*-------------------------------------------------------------------------*/
int main(void) {
   if (!creator_system_init()) // Creator 硬體初始設定
         return (0);
   process();                  // 執行主要任務
   return (0);
 }
```

```
static int process (void)
{     U8   state;
       if (!LCD_init())   return (0);
       stdio_init();
       mt_enable_cursor();    mt_clrscr();
       while (1){
          state = dipsw;   // 取得Creator EVM之DIP_SW狀態
          /* 將DIP_SW狀態反應在LED (D9 ~ D16)使之閃爍 */
          mt_main_led_set(state);       Delay(100); // 點亮LED
          mt_main_led_set(0xff);        Delay(100); // 熄滅LED
          mt_setcursor(0, 1);   mt_printf("DIP SW=%0X\n", state);
        }
       return (1);
}
```

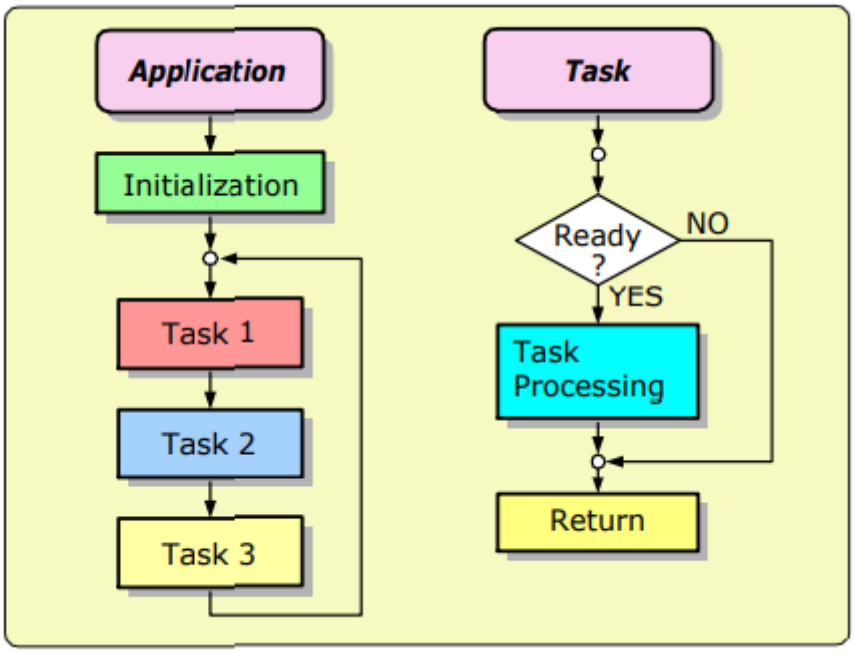
- dip switch是一種開關，可以用來作為輸入或簡單設定的資料

- mt_printf是類似printf的，用在特定硬體環境，廠商提供

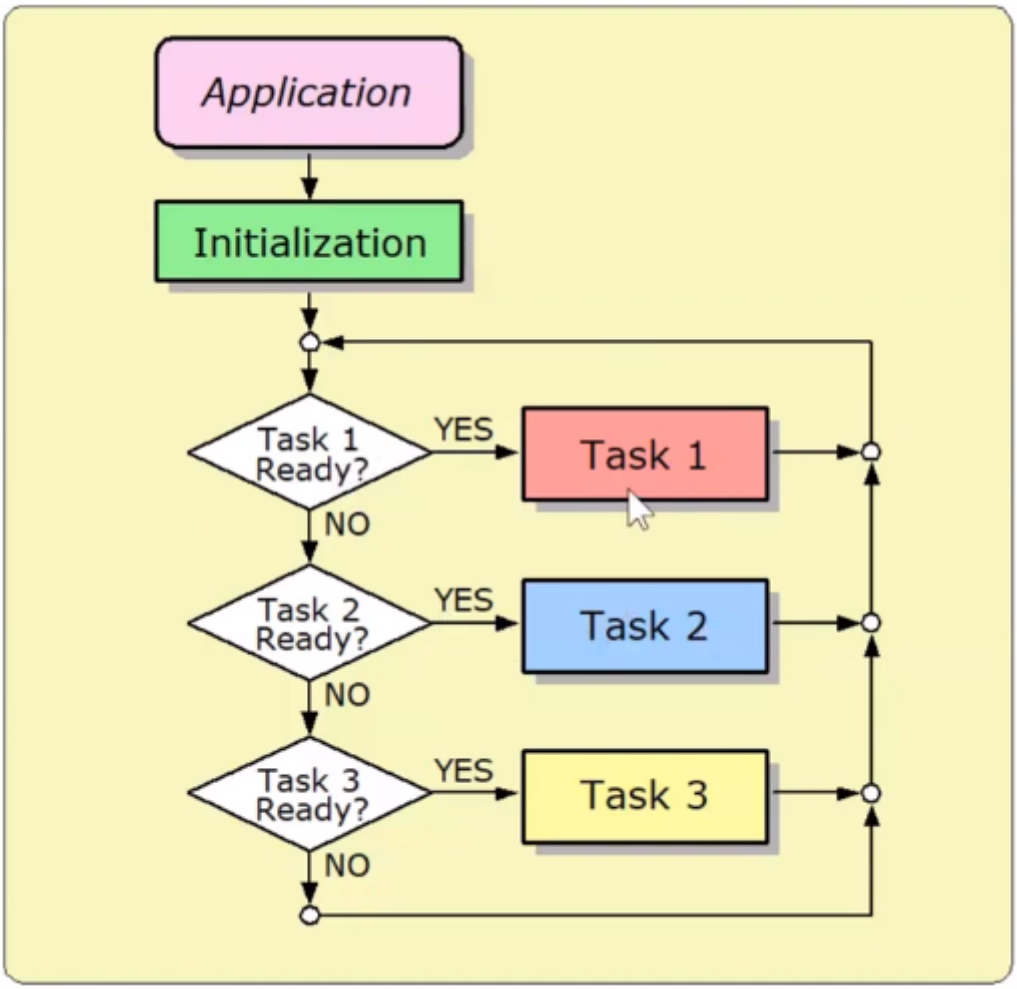
```
/*****************************************************************
 * 功能：簡易延遲millisecond(ms)                                 *
 * 輸入：ms ..... : 延遲多少ms                                   *
 * 輸出：無                                                      *
 *****************************************************************/
void  Delay (unsigned int  ms)
{
      /*  一定要加上volatile, 避免Compiler最佳化迴圈功能  */
      volatile int i, j;
      /*  使用軟體迴圈達到概略的時間延遲   */
      for (i = 0; i < ms; i++) {  for (j = 0; j < 10000; j++){  }
      }
}
```

- 注意用sleep這個函式，可能是在作業系統的環境下才能實現
- sleep即使在作業系統下實現，可能還需要硬碟輔助等，會有延遲
- sleep在桌電可能是以秒為單位，而且沒有resource constrain，誤差容許較大
- for loop什麼事都沒做，所以要使用volatile，避免compiler優化，省略掉
- 空跑for loop會讓嵌入式系統時間浪費，不過嵌入式系統本身就是特定應用，除非要做別的事情，這樣其實沒關係
- 也可以使用timer來發送interrupt訊號，設定timer也是需要花費時間，如果只要delay短暫時間，設定timer就太浪費時間
- No operation指令讓CPU慢一點，配合IO
  Basic control flow

b. Basic Control flow

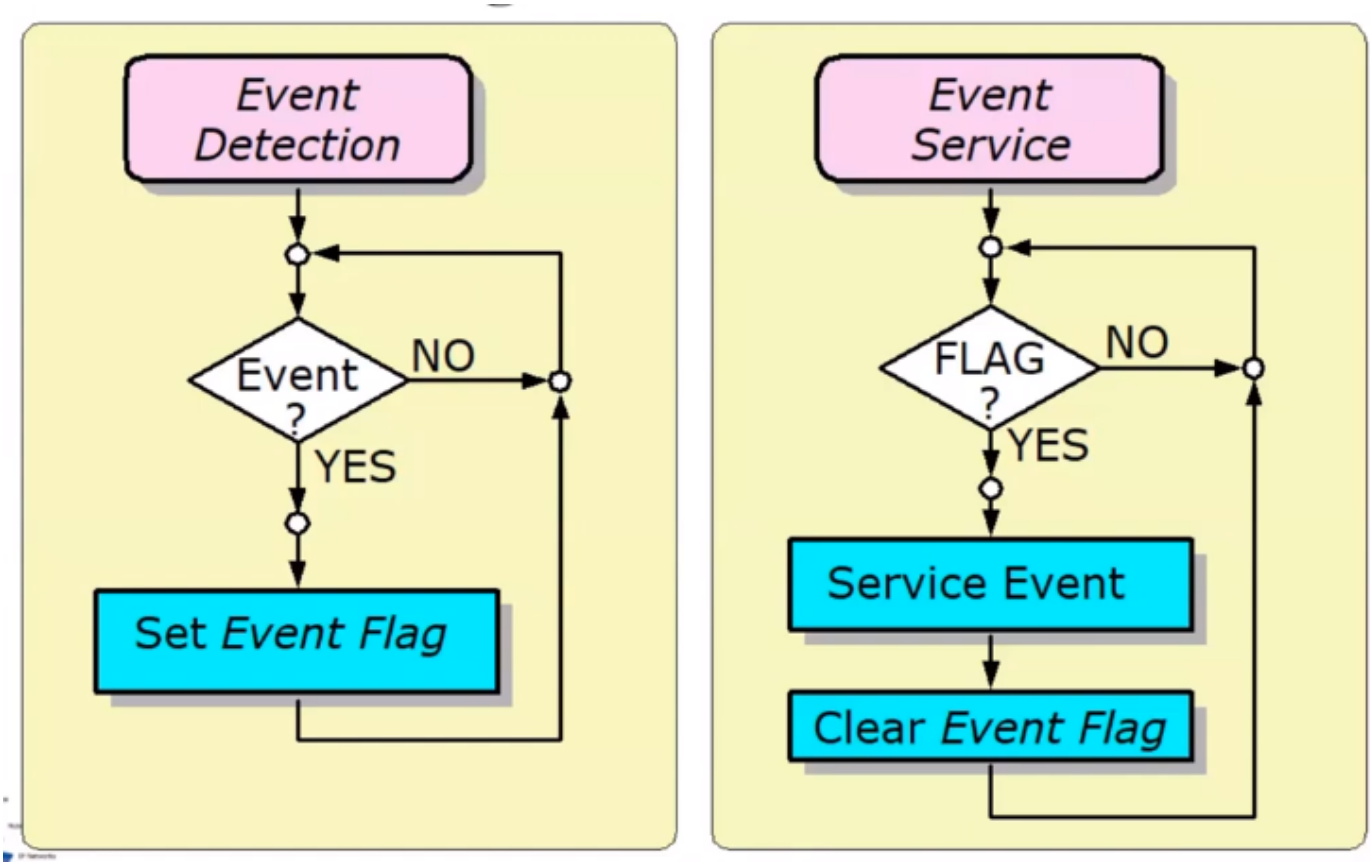


- 要就做完，要馬不做



- Priority-based
- Non-preemptive
  - Task runs until completion
  - High priority task may be blocked
  - Low priority task may starve

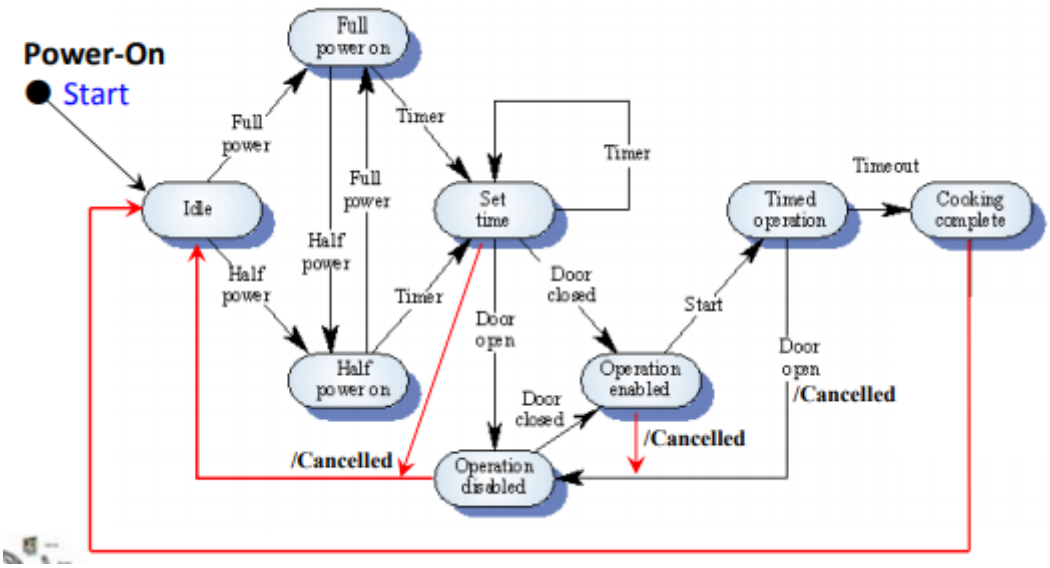
c. Inter-task (Inter-process) communication



- Shared data
- Operating system supported mechanisms
  - Shared memory and Mutex
  - Pipe (shared queue)
  - Message passing
  - ...

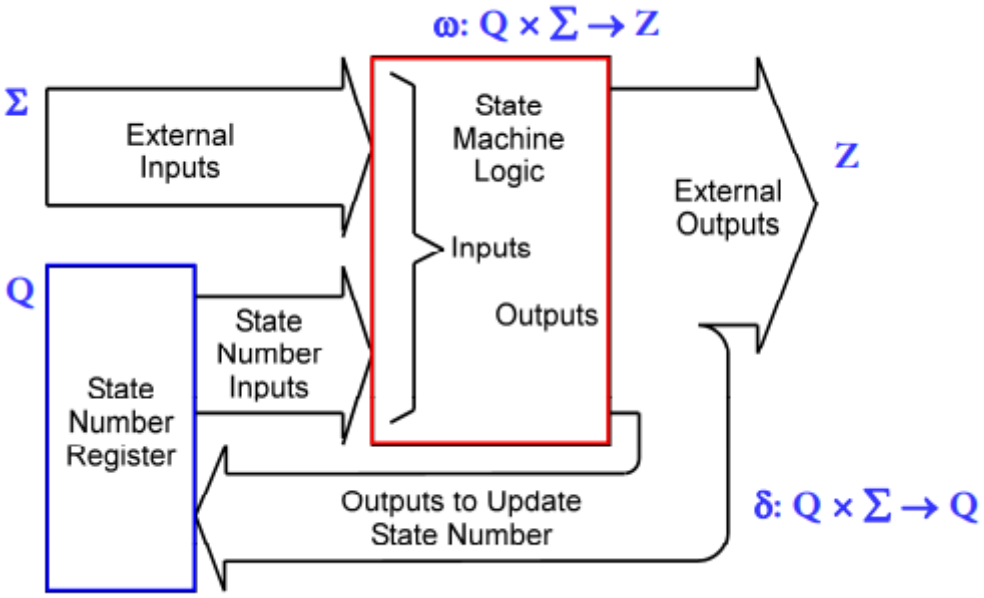
B. 嵌入式系統程式開發予狀態機實現

a. State Machine & Embedded System Design



StateChart (State Diagram):

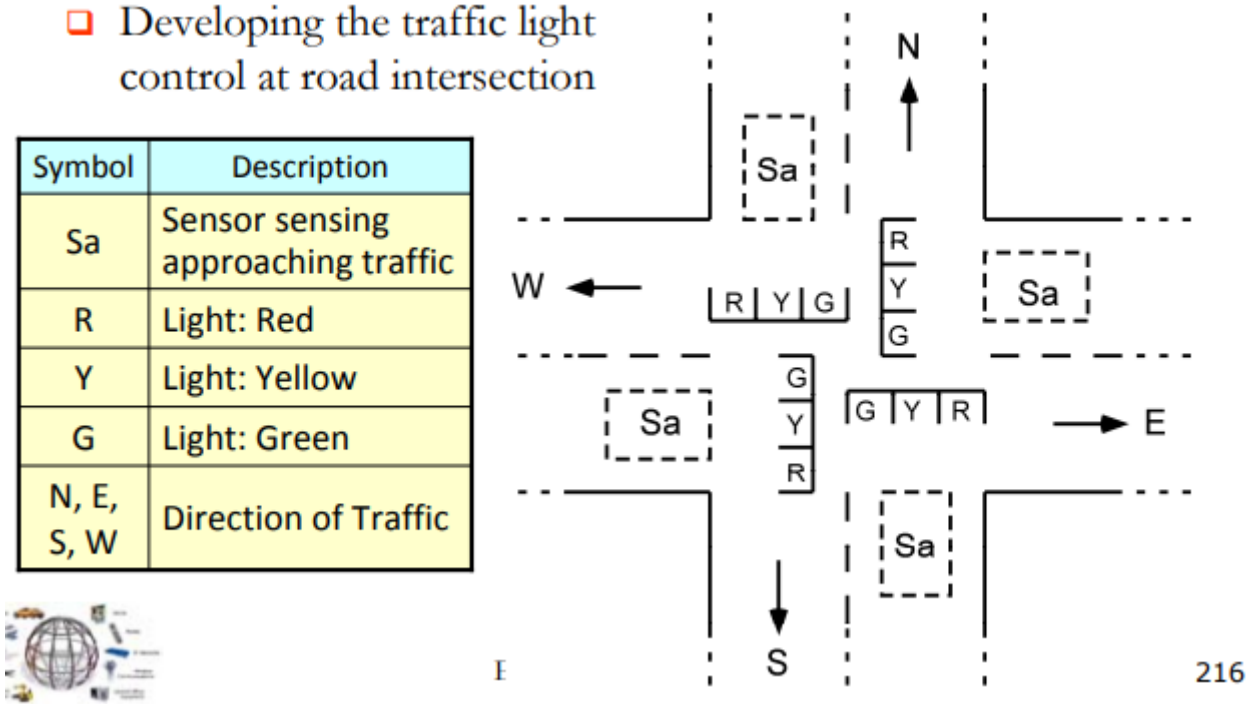
- Representation describing finite state machines
- Components:
  - ☑ Q – a finite set of states
  - ☑ Σ – a finite collection of input symbols (alphabet)
  - ☑ Z – a finite collection of output symbols
  - ☑ ω – output function representing $Q \times \Sigma \rightarrow Z$
  - ☑ δ – representation of transitions: $Q \times \Sigma \rightarrow Q$
  - ☑ q_0 – the start state ($q_0 \in Q$)
  - ☑ F – a finite set of accepting states ($F \subseteq Q$)



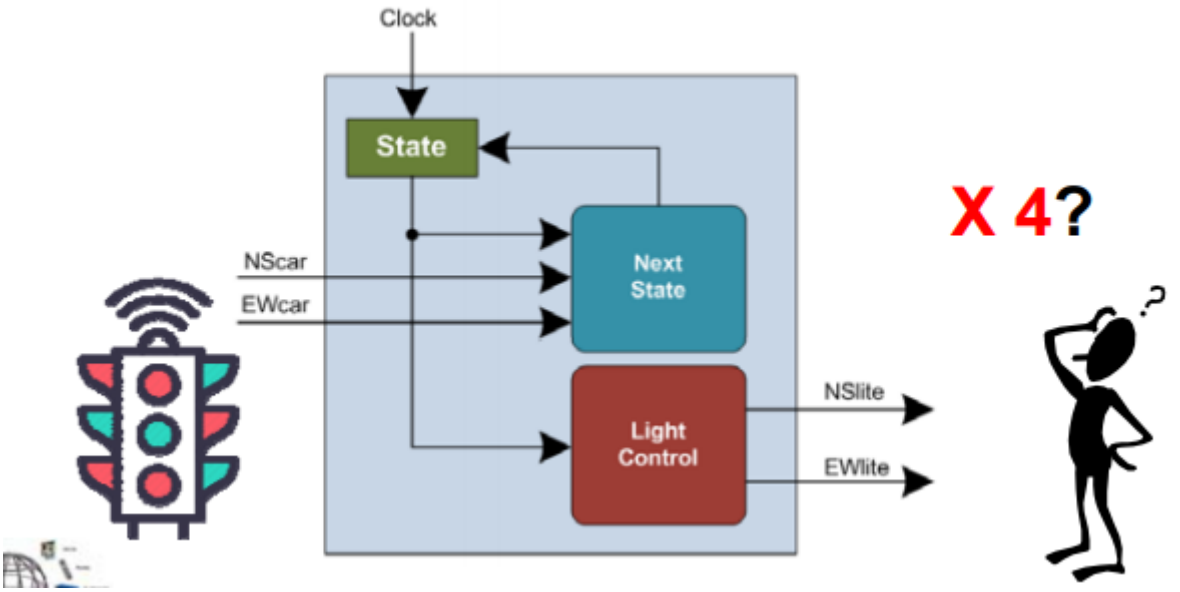
- Requirement/Specification Analysis
- State Machine Design:
  - Lists of inputs, outputs, states and transitions
  - State minimization
  - State assignment (state encoding)
  - Handling of unused states and "don't care" states
- Implementation:
  - Hardware
  - Software
- The state machine diagram must be unambiguious
  - Transition expressions leaving a state are mutually exclusive
  - Transition expressions leaving a state are all inclusive
- Except the starting state, a state must have at least one previous state (so there is transition entering the state)
- No dead-end state
- States can be grouped into a sub-state-machine

- A demonstrating example of state machine:
  - Developing the traffic light control at road intersection

Symbol	Description
Sa	Sensor sensing approaching traffic
R	Light: Red
Y	Light: Yellow
G	Light: Green
N, E, S, W	Direction of Traffic

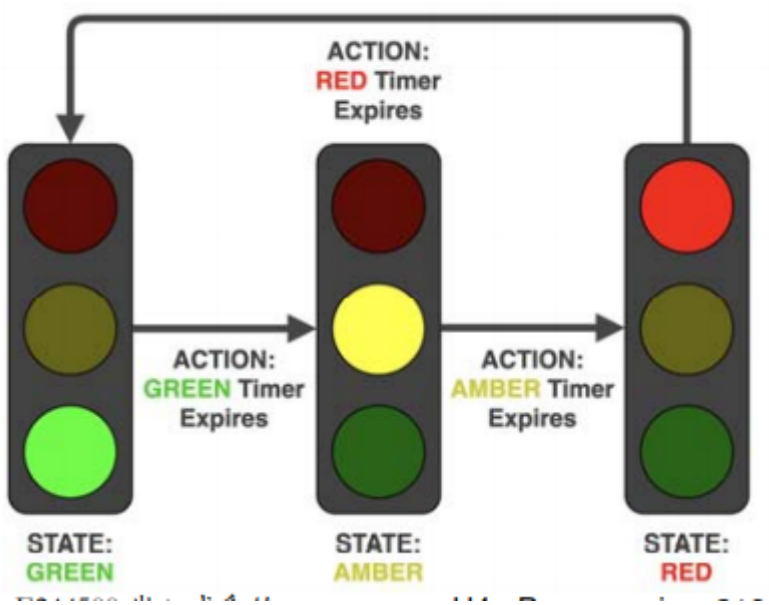


Developing the traffic light control at road intersection



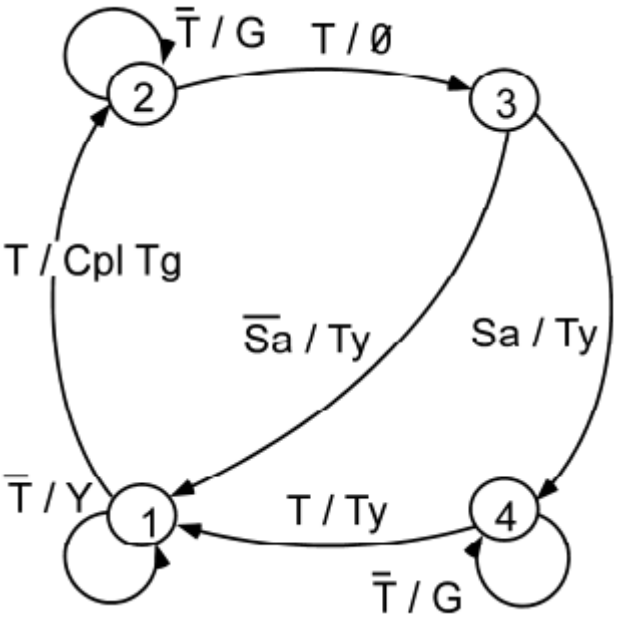
Developing the traffic light control at road intersection

- The controller on EW and NS directions are
  - same operations
  - mutually exclusive
- One state machine fits
- Input:
  - Timer time out signal
  - Sensor signal
- Output:
  - Red light On/Off
  - Yellow light On/Off
  - Green light On/Off

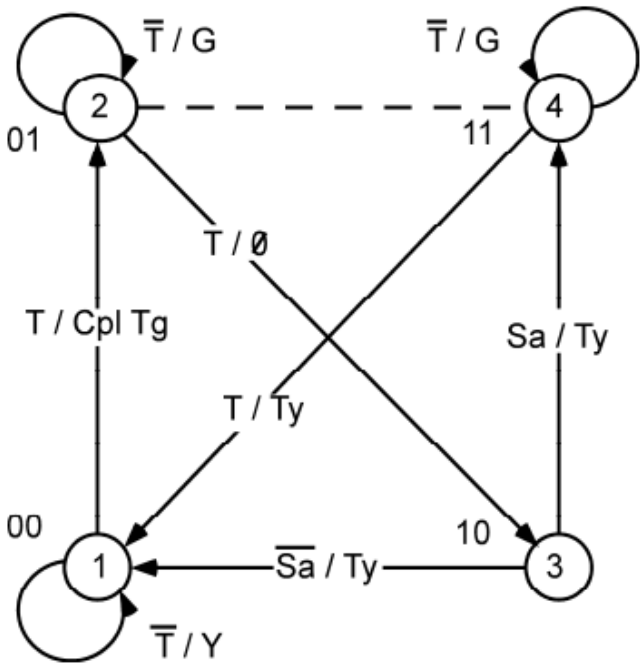


State diagram of the traffic light control

Type	Symbol	Description
Input	T	Timer alarm
	Sa	Approaching traffic
Output	Ty	Timer starts for Y
	Tg	Timer Starts for G
	Y	Yellow light on
	G	Green light on
	Cpl	Complementing Flag
Misc.	Φ	Don't care



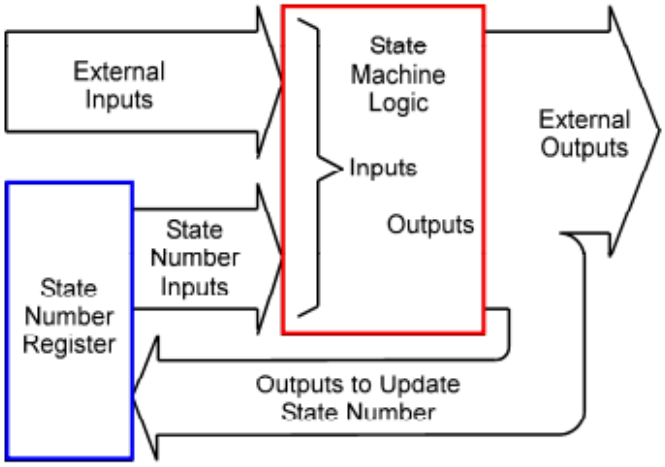
- State assignment diagram of the traffic light control

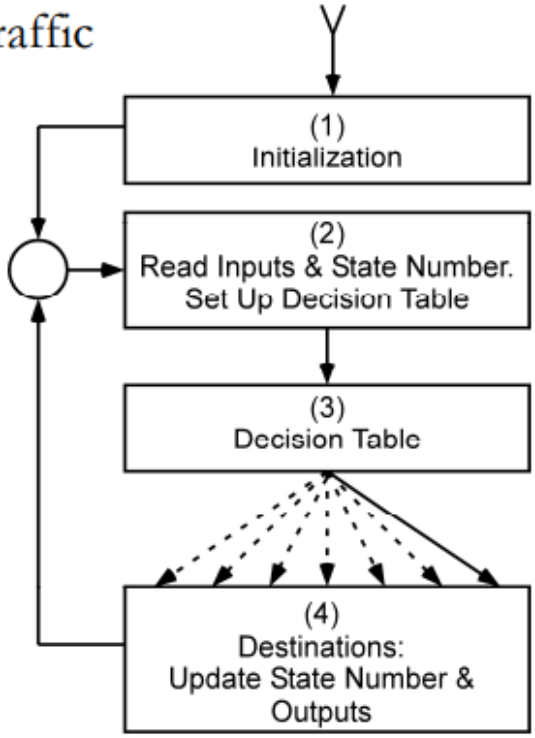


Truth Table

State			Inputs		Outputs							Next State		
No.	Q1	Q0	Sa	T	Y	G	Cpl	Tg	Ty	Rst	Inc	Q1	Q0	No.
1	0	0	0	0	1	0	0	0	0	0	0	0	0	1
	0	0	0	1	0	0	1	1	0	0	1	0	1	2
	0	0	1	0	1	0	0	0	0	0	0	0	0	1
	0	0	1	1	0	0	1	1	0	0	1	0	1	2
2	0	1	0	0	0	1	0	0	0	0	0	0	1	2
	0	1	0	1	0	0	0	0	1	0	1	1	0	3
	0	1	1	0	0	1	0	0	0	0	0	0	1	2
	0	1	1	1	0	0	0	0	1	0	1	1	0	3
3	1	0	0	0	0	0	0	0	1	1	0	0	0	1
	1	0	0	1	0	0	0	0	1	1	0	-	0	1
	1	0	1	0	0	0	0	0	1	0	1	1	1	4
	1	0	1	1	0	0	0	0	1	0	1	1	1	4
4	1	1	0	0	0	1	0	0	0	0	0	1	1	4
	1	1	0	1	0	0	0	0	1	1	0	0	0	1
	1	1	1	0	0	1	0	0	0	0	0	1	1	4
	1	1	1	1	0	0	0	0	1	1	0	0	0	1

- Software implementation of the traffic light control: **Top-level flow**





Software Implementation of State Machine:

- Basic approaches
  - Hardwired coding
  - State-coded/driven
  - Table driven (Transition table)
- Issues and design choices
  - Complexity in implementation
  - Difficulty/Cost in maintenance
  - Memory consumption

Hardwired coding

- *Ad-hoc* implementation
- States are maintained implicitly
- Achievable efficiency (via code optimization)
- Issues:
  - Complexity – case by case
  - Maintenance – generally difficult and costly

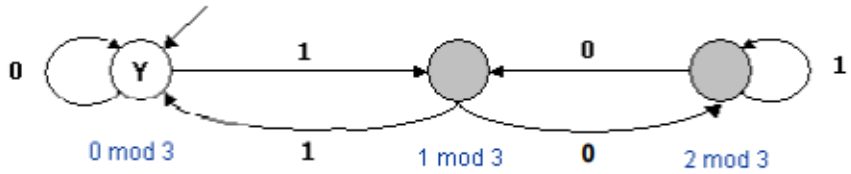
- 經驗法則

State-coded/driven

- Direct simulation of a state machine
- Basic components:
  - State variable
  - Nested switch/case statements
- Issues:
  - Complexity – moderate
  - Maintenance – in general manageable

- Table-driven
  - Generic implementation
  - Components:
    - Data structure – transition table
    - Generic code as the engine to drive state machine
    - The action routines
  - Low cost in maintenance
  - Issues:
    - The transition table can be space-consuming

- 可以用sparse matrix解
  - An example (1/4) –
    - Implementing the *DFA* that can determine whether the input binary number is a multitude of three. (Note: Input fed from **left to right**.)
    - The state diagram:



1	1	0	1

An example (2/4) –

- The skeleton of implementation:

```
Initialize();
get_input();
if (End_of_Input) output("NO INPUT!");
 else   { do { state_transition();
                    get_input();
             } while(!End_of_Input);
             if (state == accept) output("YES.");
                   else output("NO.");
 }
```

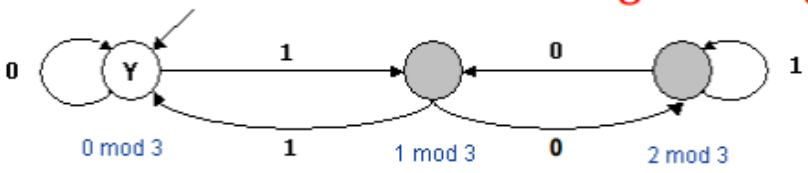
Implementation of state machine: (cont'd)

An example (3/4) –

- Implementation of state and state transition:

The pseudo code of *state_transition();*

```
switch (state)
  0:  if (Input == 1) state = 1; break;
  1:  if (Input == 1) state = 0; else state = 2; break;
  2:  if (Input == 0) state = 1; break;
  default: something is wrong!
```



Really necessary?

- default可能是硬體問題，在嵌入式系統裡面很重要

An example (4/4) –

- Implementation of state and state transition:

  The pseudo code of *state_transition();*

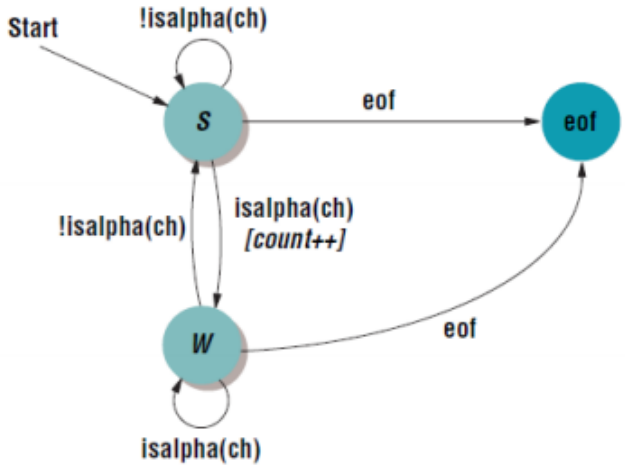
state = transition_table[*Input*, *state*];

Input \ state	0	1	2
0	0	2	1
1	1	0	2

Another example (1/4) –

- Implementing the *DFA* for the program that can count the words in a text file
- The state diagram:

State	Code	Description
Start	00	DFA starts
S	01	Skipping white spaces
W	10	In a word
eof	11	DFA terminates



Another example (2/4) –

- The skeleton of implementation:

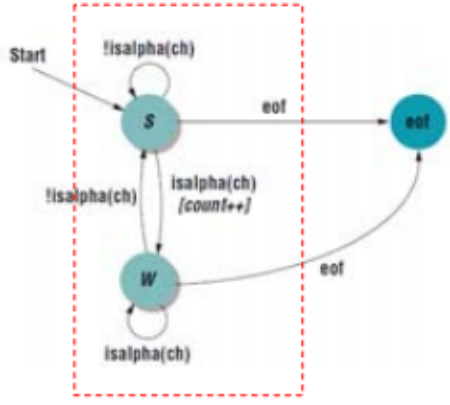
```
Initialize();
wc = 0;          //  Initializing wc (word count)
read_file(ch);
if ( End_of_File ) output("NO INPUT!");
  else  {  do {  state_transition();
                 read_file(ch);
              } while( !End_of_File );
             output("The word count is ", wc);
  }
```

Another example (3/4) –

- Implementation of state and state transition:

The pseudo code of *state_transition*();

```
switch (state)
  1:  if ( isalpha(ch) ) {
          wc++; state = 2;
      }
      break;
  2:  if ( !isalpha(ch) ) state = 1;
      break;
  default: something is wrong!
```



default: something is wrong!

Another example (4/4) –

- Implementation of state and state transition:

The pseudo code of *state_transition*();

state = transition_table[*Input_ch*, *state*];

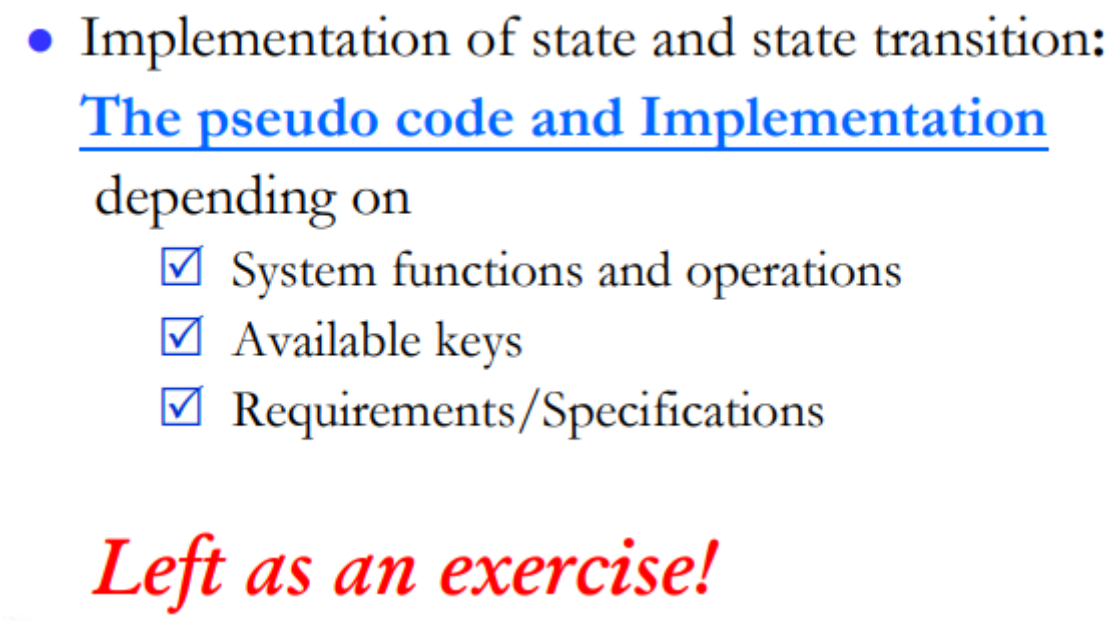
Input ch \ state	1	2
white space	1	1
alphabet	2	2

- 注意action要另外紀錄

- Yet another example (1/4) –
  - Implementing the user interface for controlling the system operations:
    - Mode control
    - Functions: F1~F5
    - Cursor movement
    - Data input
    - Display control
  - Typical U/I design

- Design Issues:
  - Multiple roles of key-switch
  - Hierarchical structure of states
  - Super-state/Sub-state and rules of state transition
  - … …
- The Essential Modules:
  - get_input_key();
  - update_display();
  - the state machine
- The skeleton:

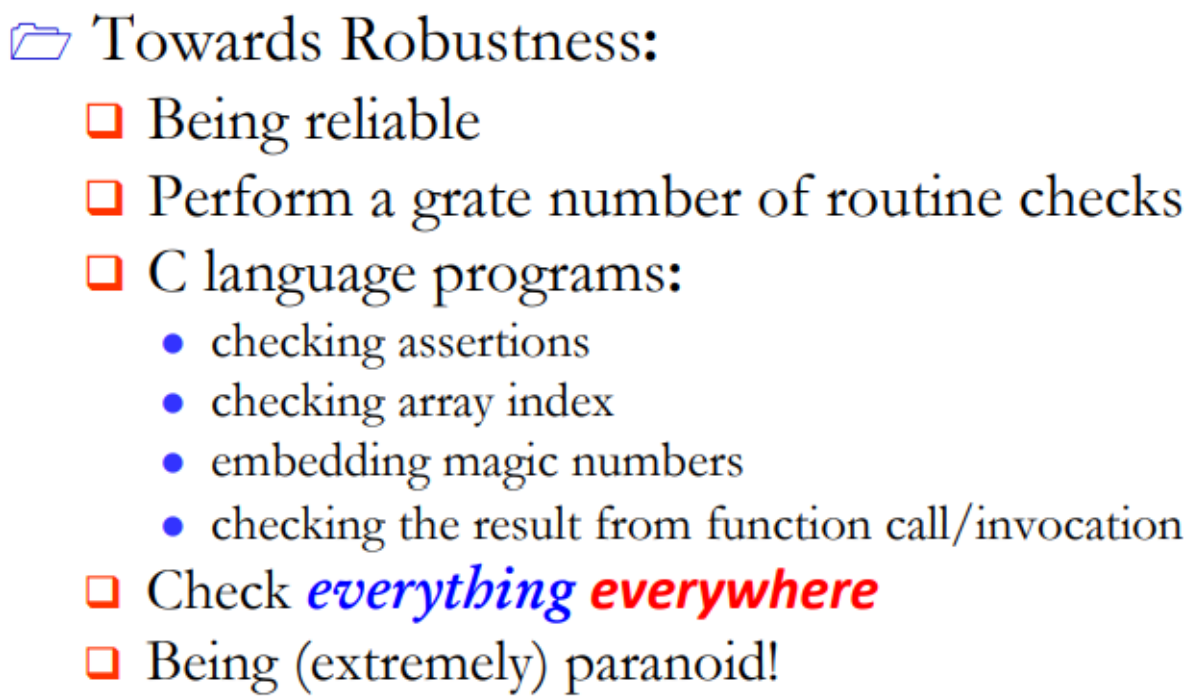
```
System_Initialization( );
while ( state != PowerOff ) {
    Update_Display( );
    if( Key_Pressed( ) ) {
        Get_Input( );
        Processing & state_transition( );
    }
}
System_ShutDown( );
```



C. 程式強韌性與看們狗計時器(WDT)

Robustness of System:

- A system that does not break down easily or is not wholly affected by a single application failure.
- A system that either recovers quickly from or holds up well under exceptional circumstances.
- A system that is not wholly affected by a bug in one aspect of it.
- ...



- magic numbers特別給定的值，是不可以被改變的

Robust Design:

- With exceptional conditions in a way that is both "safe" and "intelligent".
- Whenever failure is detected, the program must perform "cleanup" to "undo" the current operation.

Robust Programming Techniques:

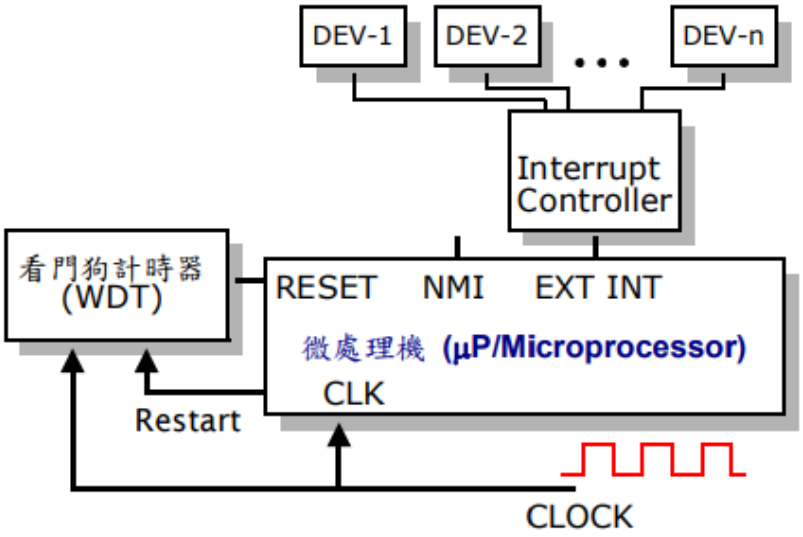
- Being Paranoia (Check Everything Everywhere)
- One-or-Nothing (Commit or Rollback)
- Structured exception handling (SEH)
- Clean stack exception handling (CSEH)

Watchdog Timer (WDT):

- Hardware to automatically detect software anomalies/malfunctioning
- Based on a count-down counter
- Required to be restarted by software
- Asserts reset or NMI signal upon counting to zero
- Useful in helping recover from transient failures

- NMI : Non Maskable Interrupt

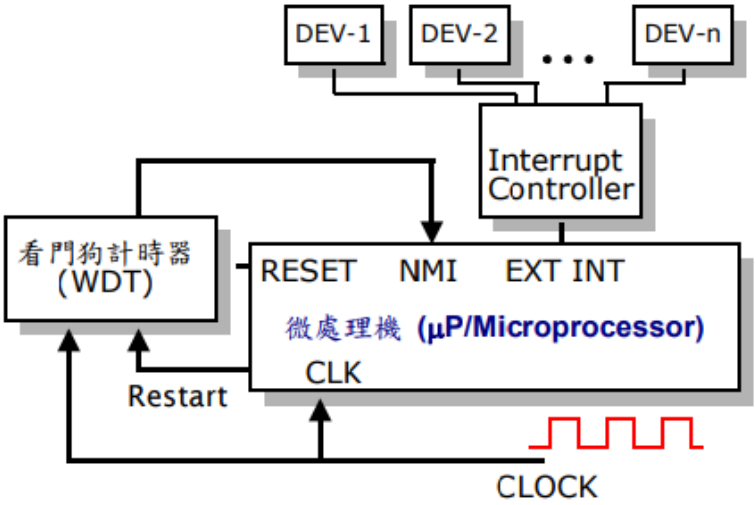
Setup of Watchdog Timer (1):



* The WDT connects to the reset signal input of the μP

- Interrupt controller可以遮蔽Interrupt，然而NMI是不可被遮蔽，微處理機對它一定會有反應

Setup of Watchdog Timer (2):



* The WDT connects to the NMI signal input of the μP

Programming the Watchdog Timer (WDT):

```
/* Define the configuration of WDT*/
...
main (void)
{
    HW_Init( );
    for ( ; ; ) {
        WDT_Kick ( );  /* Retriggering the WDT */
        read_sensors ( );
        output_control_signal ( );
        Dev_display_status ( );
    }
}
```

Programming the Watchdog Timer (WDT):

- Issues
  - The length of time duration
  - Where or which task does the kicking?
  - Can the WDT be disabled (turned off)?
  - ...