EMBEDDED PROGRAMING

Bosch Embedded Academy
Version 3

BOSCH

Objectives & Assumptions

Remind general programming principles
Remind key embedded system knowledges
How to program for embedded system effectively
Know and practice Object-oriented programming

Experienced with programming at basic level
Not focus on how to programming
C and C++

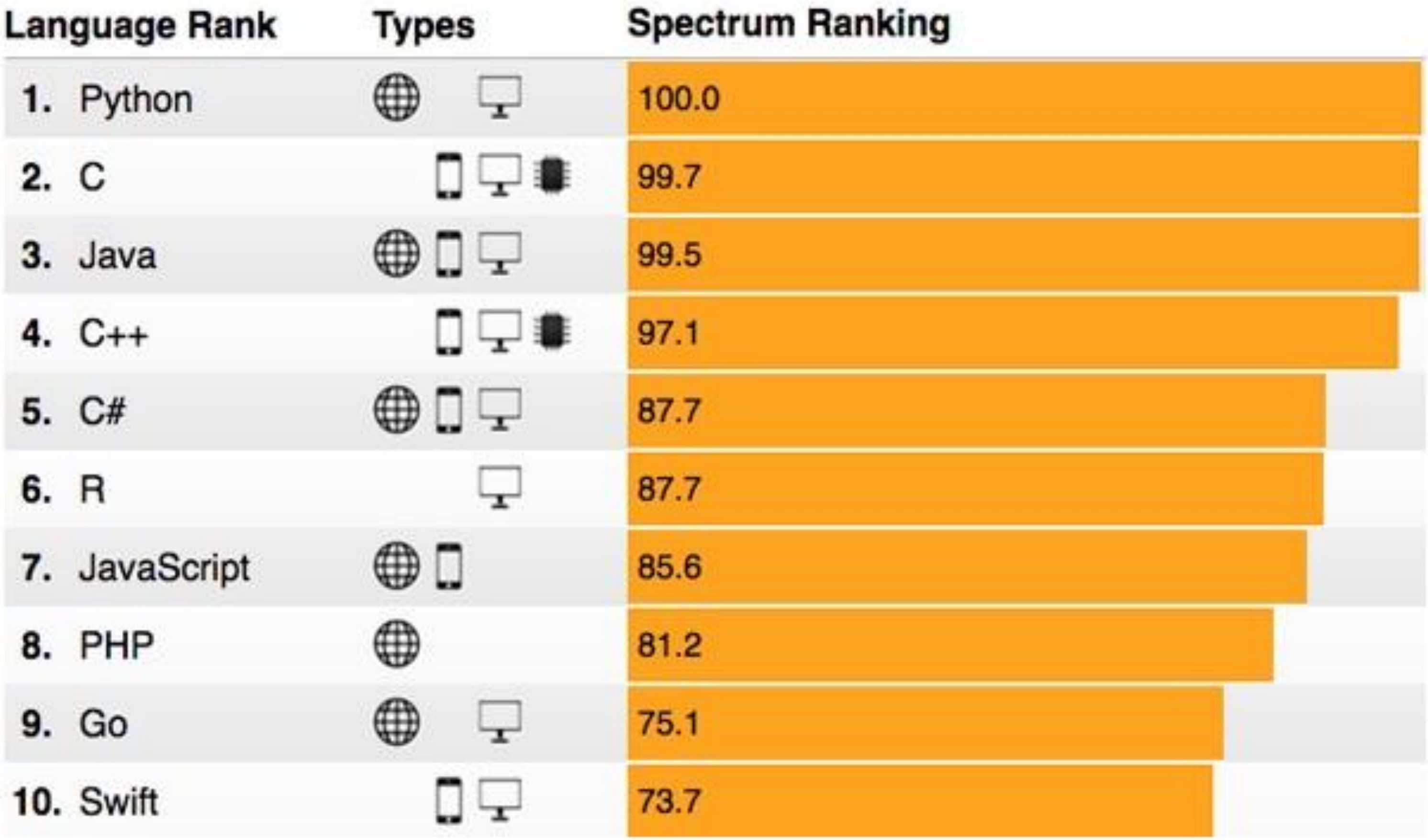
Agenda

1. Programing principles remind
    1. Programing remind/Overview
    2. Language Evaluation Criteria
    3. The Compiling Process
2. Embedded system programing
    1. What is embedded system?
    2. Which common elements inside Embedded SW?
    3. Constraints affect design choices?
    4. Why C is most common language for Embedded programing?
    5. What skills required for Embedded programmer?
    6. RTOS
3. Some importance topics that related to embedded programming
4. OOP principles basic

Agenda

1. Programing principles remind
    1. Programing remind/Overview
    2. Language Evaluation Criteria
    3. The Compiling Process

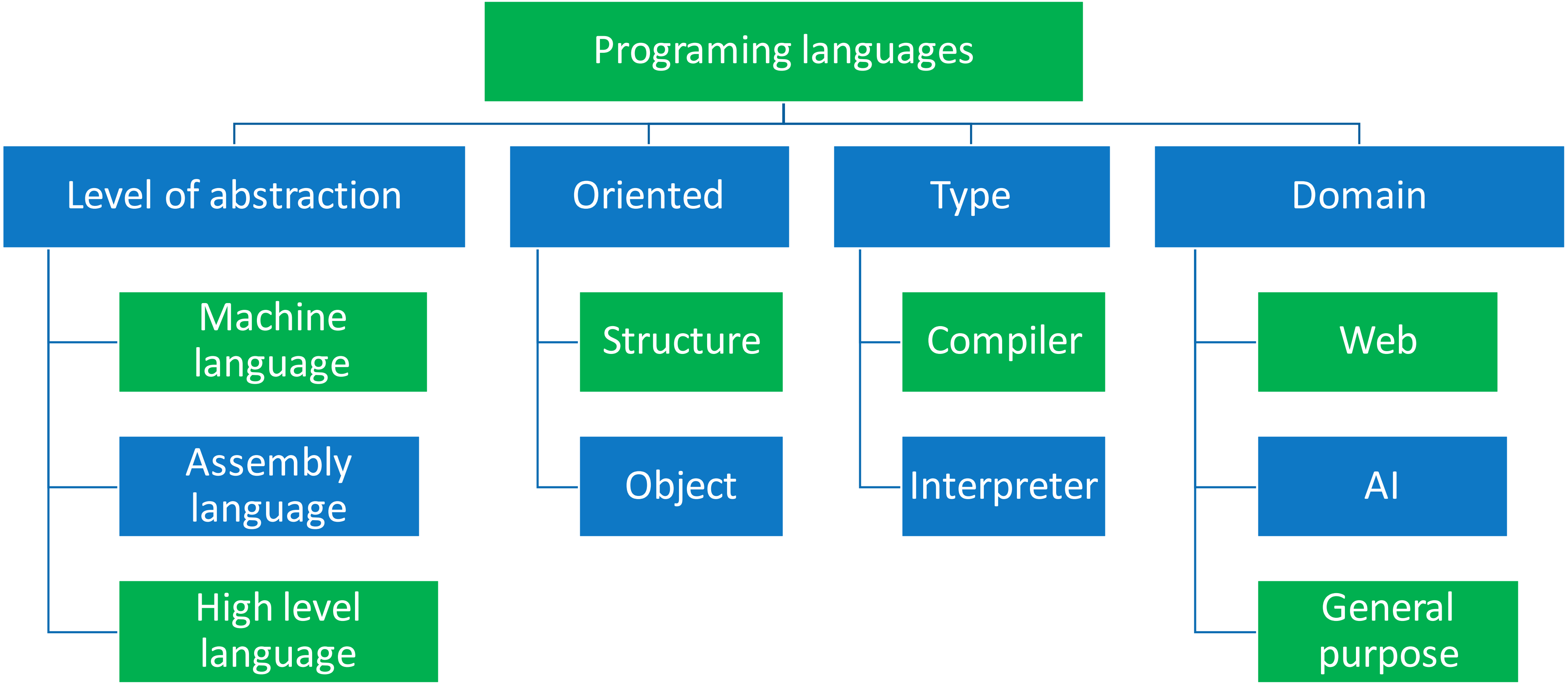
Programing principles
Programing languages



Language Rank	Types	Spectrum Ranking
1. Python	🌐 🖥	100.0
2. C	📱 🖥 ■	99.7
3. Java	🌐 📱 🖥	99.5
4. C++	📱 🖥 ■	97.1
5. C#	🌐 📱 🖥	87.7
6. R	🖥	87.7
7. JavaScript	🌐 📱	85.6
8. PHP	🌐	81.2
9. Go	🌐 🖥	75.1
10. Swift	📱 🖥	73.7

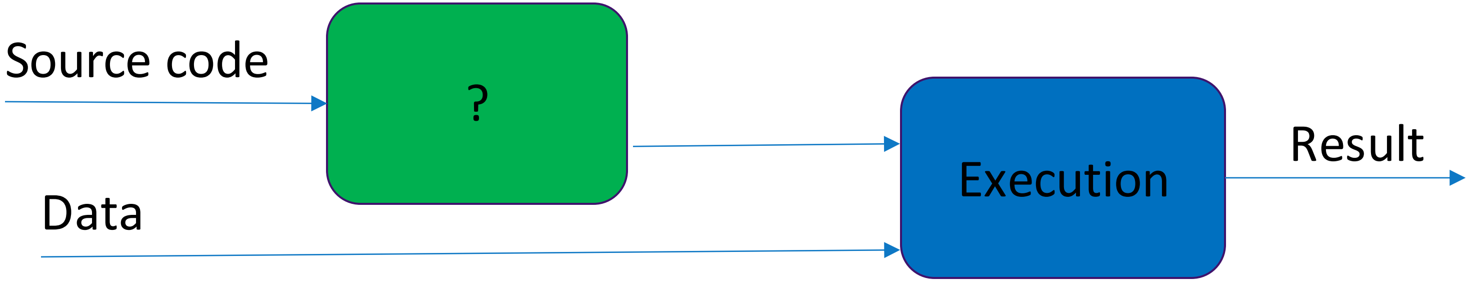
Programing principles

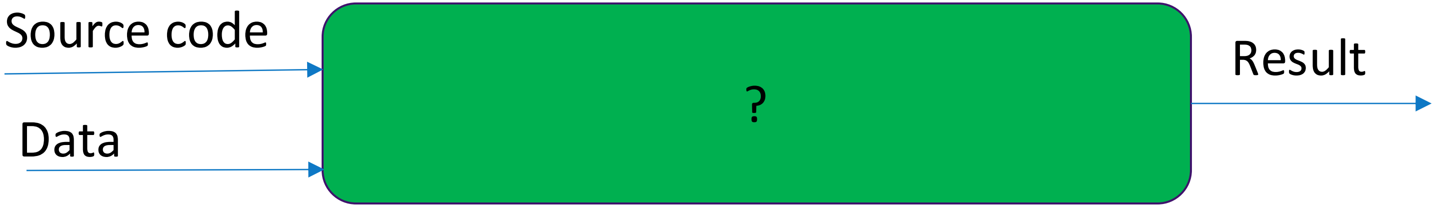
Programing language classification



Programing principles

Compiler vs Interpreter





Programing principles
Programing Domains

- Scientific Applications
  - Fortran, ALGOL60
- Business Applications
  - COBOL
- Artificial Intelligence
  - LIPS, Prolog
- System Programing
  - PL/S, BLISS, Extended ALGOL
- Web Software
  - XHTML, JavaScript, PHP

Programing principles

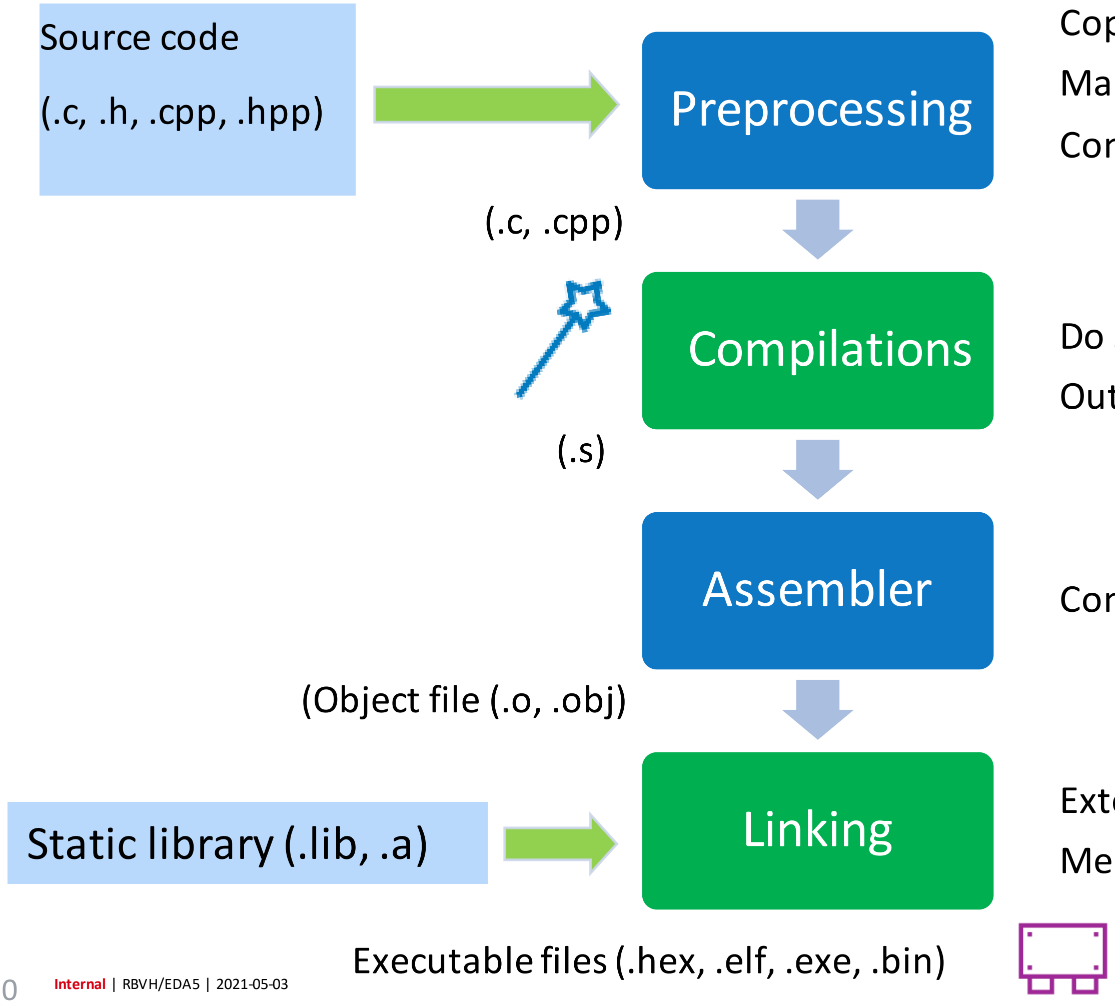
Language Evaluation Criteria and Characteristics

	REABILITY	WRITABILITY	REALIABILITY
Simplicity	*	*	*
Data types	*	*	*
Syntax	*	*	*
Abstraction		*	*
Type checking			*
Exception handling			*

- Cost of learning.
- Cost of writing.
- Cost of compiling, executing, optimization, maintaining, portability…

Programing principles

Compilations steps



Source code

(.c, .h, .cpp, .hpp)

Preprocessing

Copy all contains of header files (.h, .hpp) into target file (.c, .cpp)

Macros replace (#define)

Conditional compilation (#if, #ifdef, #ifndef, #else, #endif)

(.c, .cpp)

Compilations

Do some magic ☺

Output as Assembly format (Load r1, X)

(.s)

Assembler

Convert Assembly code to Machine code (0/1)

(Object file (.o, .obj)

Static library (.lib, .a)

Linking

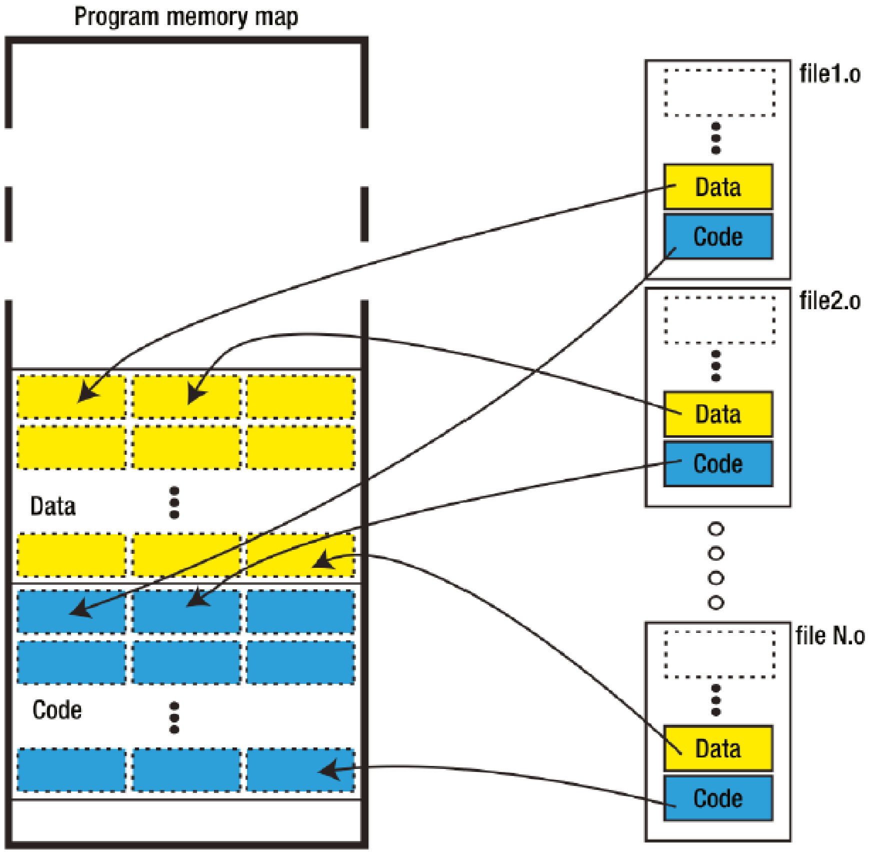
External symbol linking

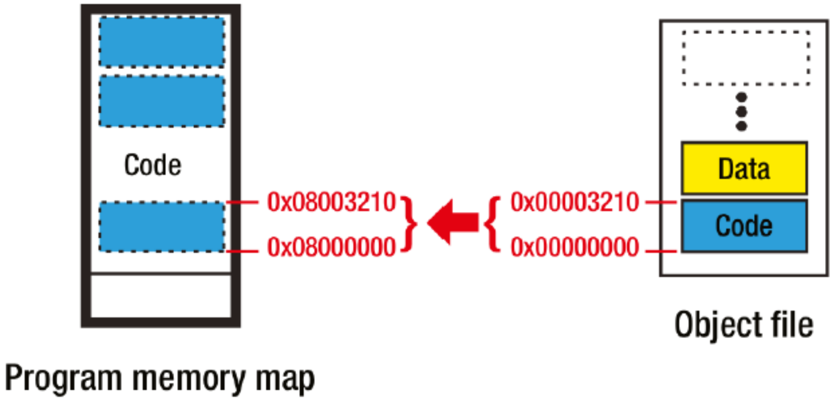
Memory relocation

Executable files (.hex, .elf, .exe, .bin)

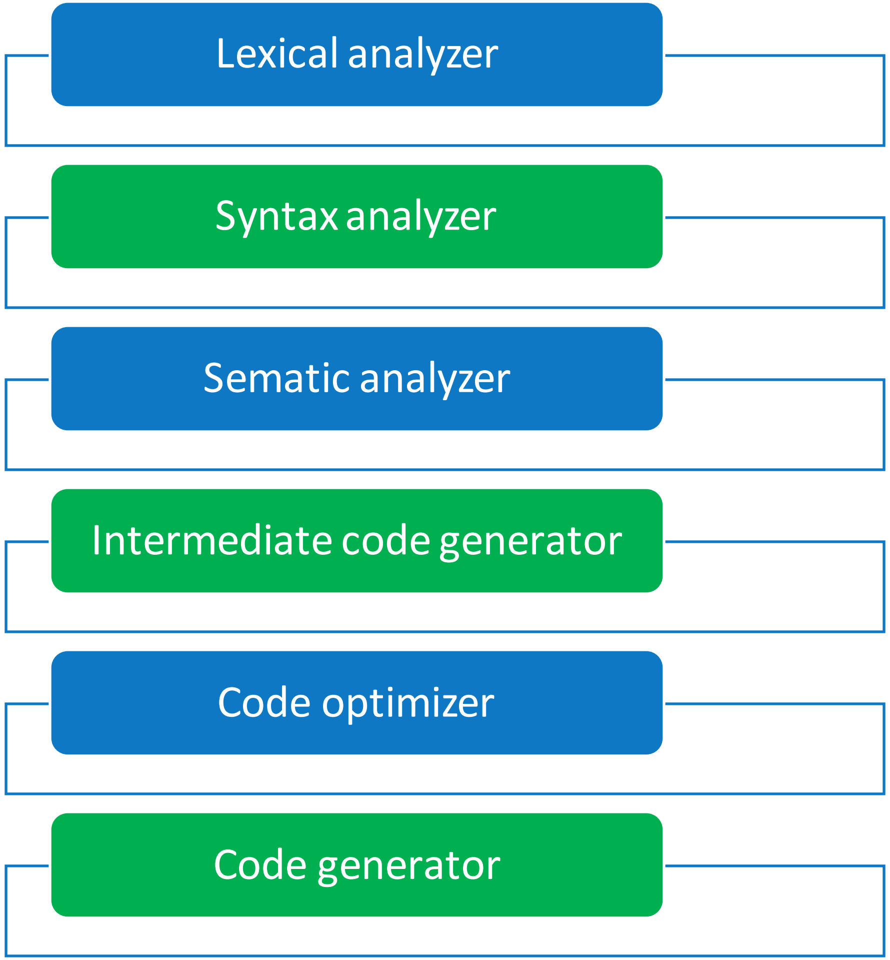
Programing principles

Linking: memory relocation





Programing principles
Compilation Phases



Lexical analyzer

Syntax analyzer

Sematic analyzer

Intermediate code generator

Code optimizer

Code generator

Programing principles

Q1

```
#define MAX(a,b) (a>b)?a:b
inline int ReturnMaxNumber (int a, int b) {if a > b return a; else return b;}
int ReturnMaxNumber (int a, int b) {if a > b return a; else return b;}
```

Which statement about Macro function and Inline function are Incorrect?

A. Macro function is replaced in Preprocesor phase, while Inline function is replaced in Compiler phase.

B. When running, both Macro function and Inline function will not do context saving and make function call jumping. So they both run faster than normal function.

C. Both may take more memory code size compare to normal function.

D. Both Macro function and Inline function will harder for debugging compare to normal function.

E. An Class member can be access inside both Inline function and Macro function.

Programing principles

Inline function vs Macro

```
inline return_type function_name (parameters)
{
    // inline function code
}
```

```
#define MACRO_NAME Macro_definition
```

Inline function	Macro
Parsed by the compiler	Expanded by the preprocessor
It can be defined inside or outside the class	It is always defined above of the program.
Type safe: debugging is easy for an inline function as error checking is done during compilation	Debugging becomes difficult for macros as error checking does not occur during compilation
No function call, no jumping, replace contains, run faster, more Code size	

Programing principles

Q2

Below statements about Static library and Dynamic library are Correct or Incorrect?

A. (.a, .lib) is static library file; (.so, .dll) is dynamic library file.

B. Static library is created in Compilation phase.

C. Both Static library and dynamic library can be created from multiple source files.

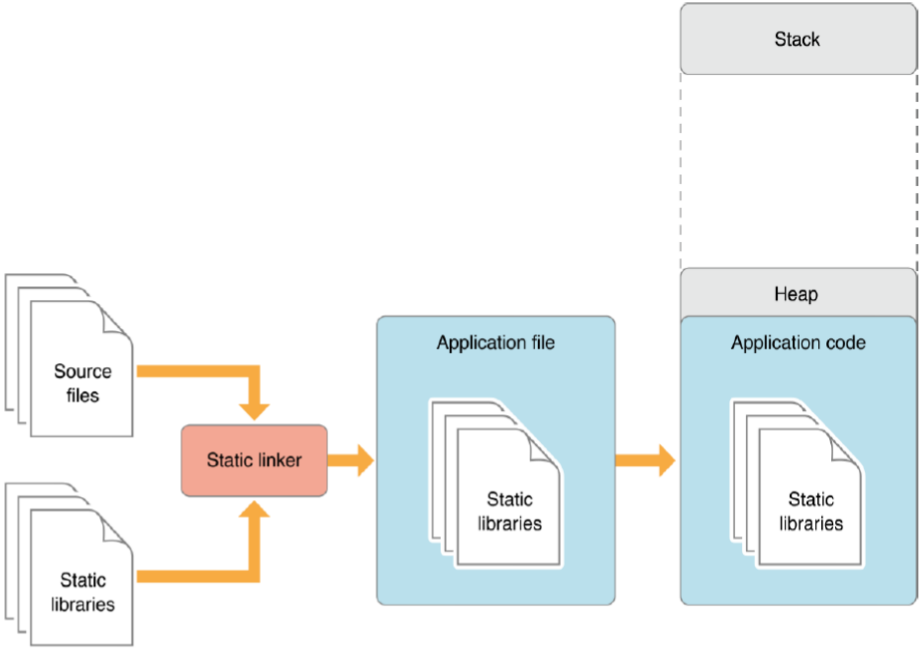
D. Static library is used in Assembler phase.

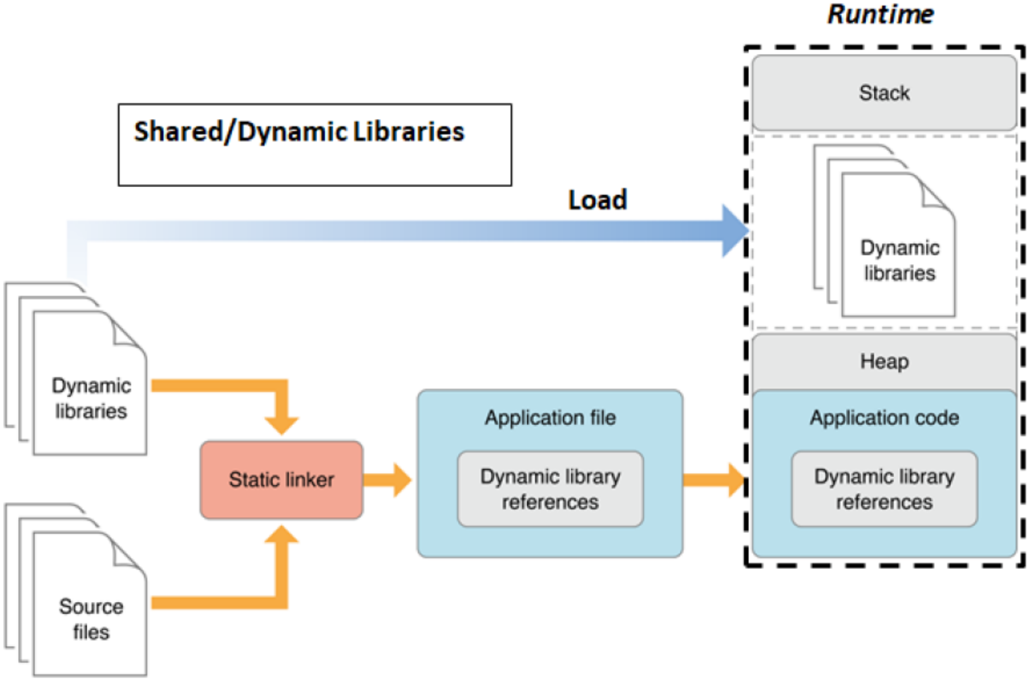
E. Using Static library can help to reduce SW compilation time.

F. Using Dynamic library can help to reduce SW running time.

Programing principles

Static library and dynamic library





Programing principles

Q3

```
int X = 0;      int Y = 0;
X += X++;
Y = ++Y + Y++;
printf("Value of X: %d\n",X);    printf("Value of Y: %d\n",Y);
```

What are output of above code?

A. X = 1; Y = 3

B. X = 2; Y = 4

C. X = 1; Y = 2

D. Wrong syntax of Y

Programing principles

Q4

```
int X[5] = {0,1,2,3,4};
int Y = 5[X];
printf("Value of Y: %d\n",Y);
```

What are output value of Y?

A. 5

B. 0

C. Wrong syntax. Cannot compiled

D. Unknown value

Agenda

1. Programing principles remind
   1. Programing remind/Overview
   2. Language Evaluation Criteria
   3. The Compiling Process
2. Embedded system programing
   1. What is embedded system?
   2. Which common elements inside Embedded SW?
   3. Constraints affect design choices?
   4. Why C is most common language for Embedded programing?
   5. What skills required for Embedded programmer?
   6. RTOS & RTOS

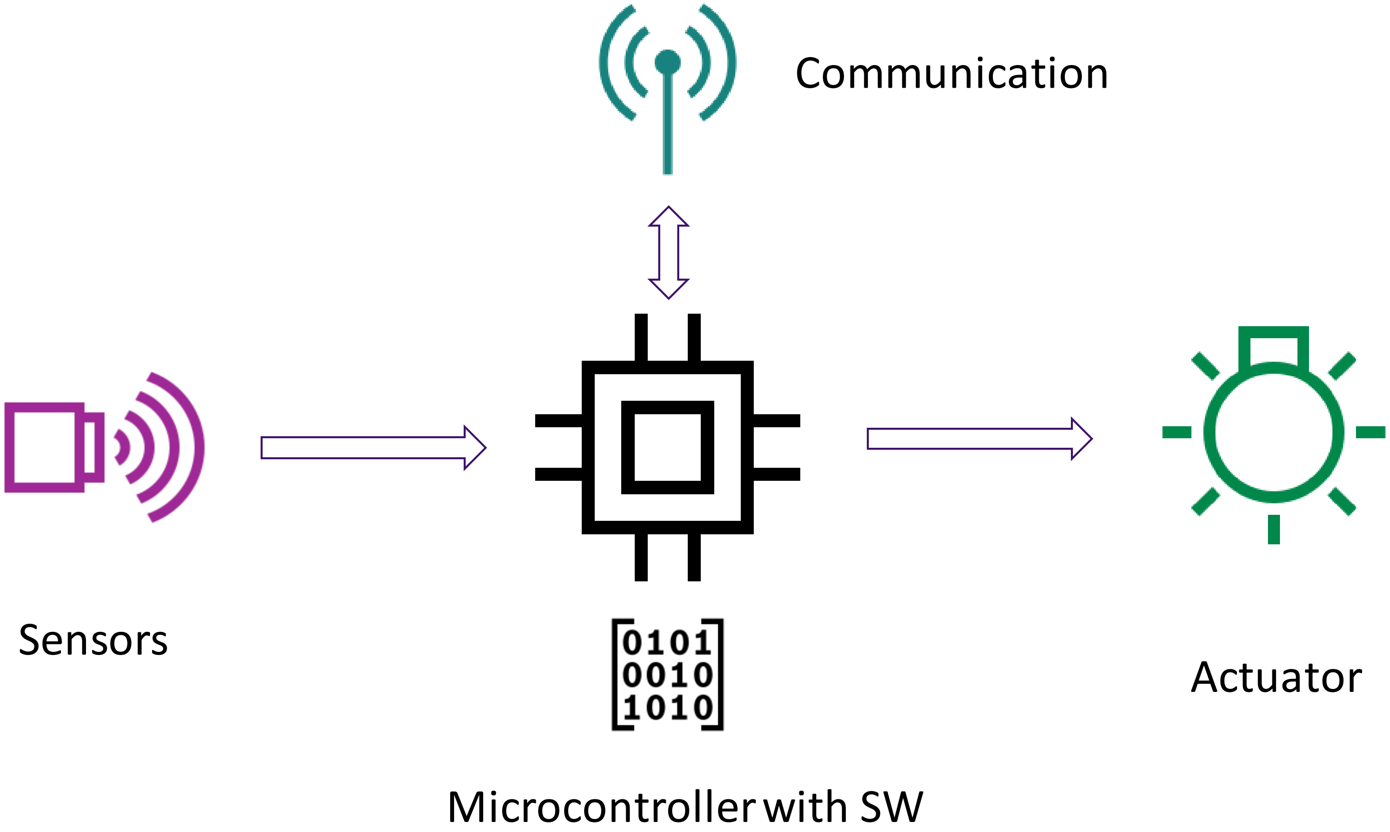
Embedded Programing

Embedded system programing: What is embedded system?

- Example of embedded system?
- Embedded system = Computer Hardware + Software + Additional parts
- Embedded system is combination of Computer HW and SW, and perhaps with additional part (mechanical, electronic) designed to perform a dedicated function.
- Frequently, Embedded system is component within lager systems.
- Automotive embedded systems are connected by Communication networks.

Embedded Programing

Basic elements of traditional Embedded system



Communication

Sensors

Actuator

Microcontroller with SW

Embedded Programing

Embedded system characteristics basic

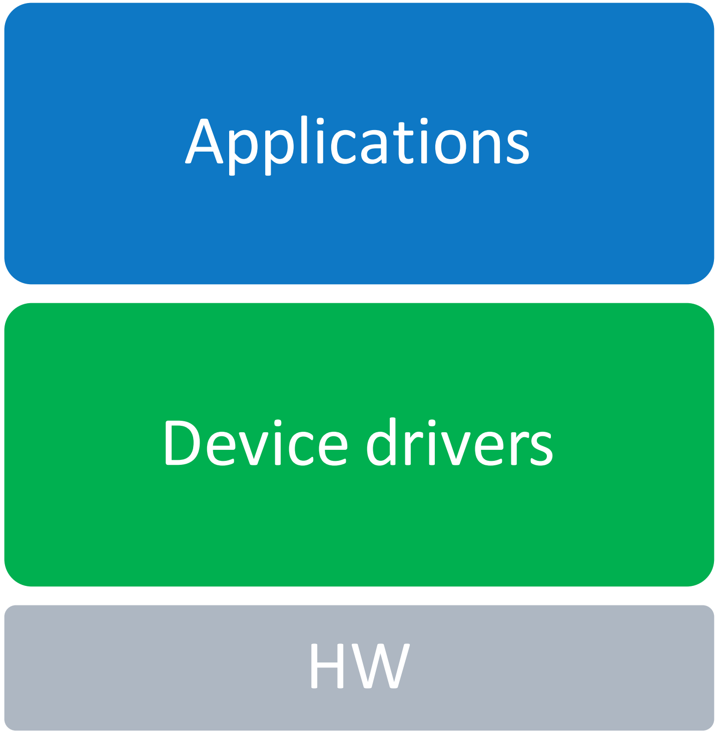
Small Size

Low cost per-unit

Low power consumption

Embedded Programing

Embedded system programing: Which common elements inside Embedded SW?



- ▶ Normally, less interact with HW.
- ▶ Device drivers developer must have detailed knowledge about using HW of the system

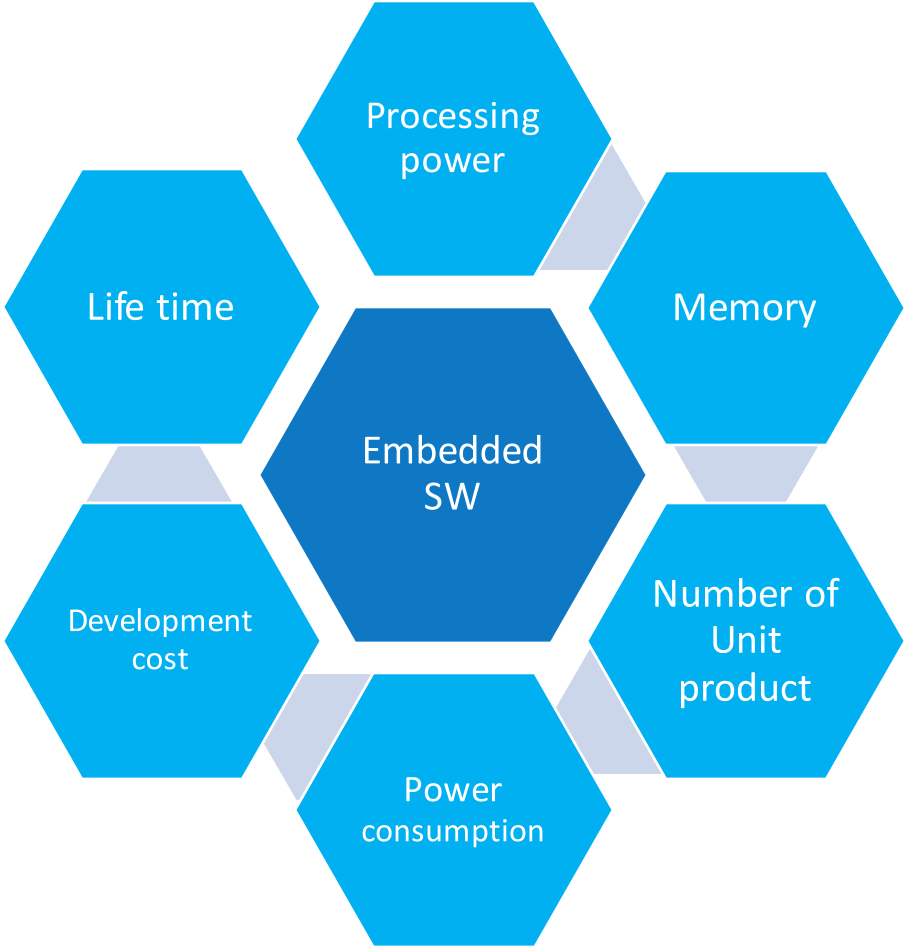
Embedded Programing

What skills required for Embedded programmer?

- HW knowledge: must familiar with microcontroller, circuits, boards, schematic, oscilloscope probe, VOM,…
- Peripheral interfaces knowledge: SPI, I2C, 1-Wire, UART,…
- Efficient code.
- Robust code.
- Minimal resources.
- Reusable code.
- Development tools/Debugging tools using.

Embedded Programing

Embedded system programing: Constraints affect design choices



- Which constrains are most importance for below example product:
  - Digital Watch
  - Video game player
  - Mars Rover

Embedded Programing

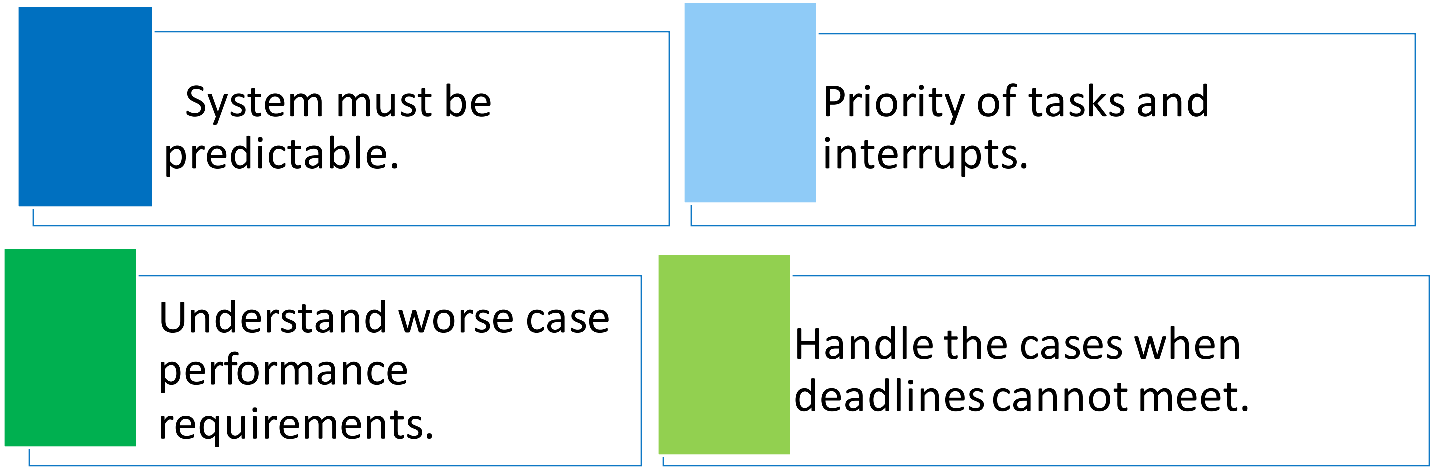
Why C is most common language for Embedded programing?

- “Low level” of high level language.
- Close with computer do, interact with HW more easily.
- Many people know and learn.
- Fairly simple to learn.
- Compilers available for most of processors.
- Processer independence.

Embedded Programing

Real Time System (RTS)

- A RTS has timing constraints. The function has deadline for completion.
- The function of a Real time system specified by ability to make calculations/decisions in timely manner.
- A RTS is not simply about the speed. It is about deadline. Guarantee that the deadlines of the system always meet.



System must be predictable.

Priority of tasks and interrupts.

Understand worse case performance requirements.

Handle the cases when deadlines cannot meet.

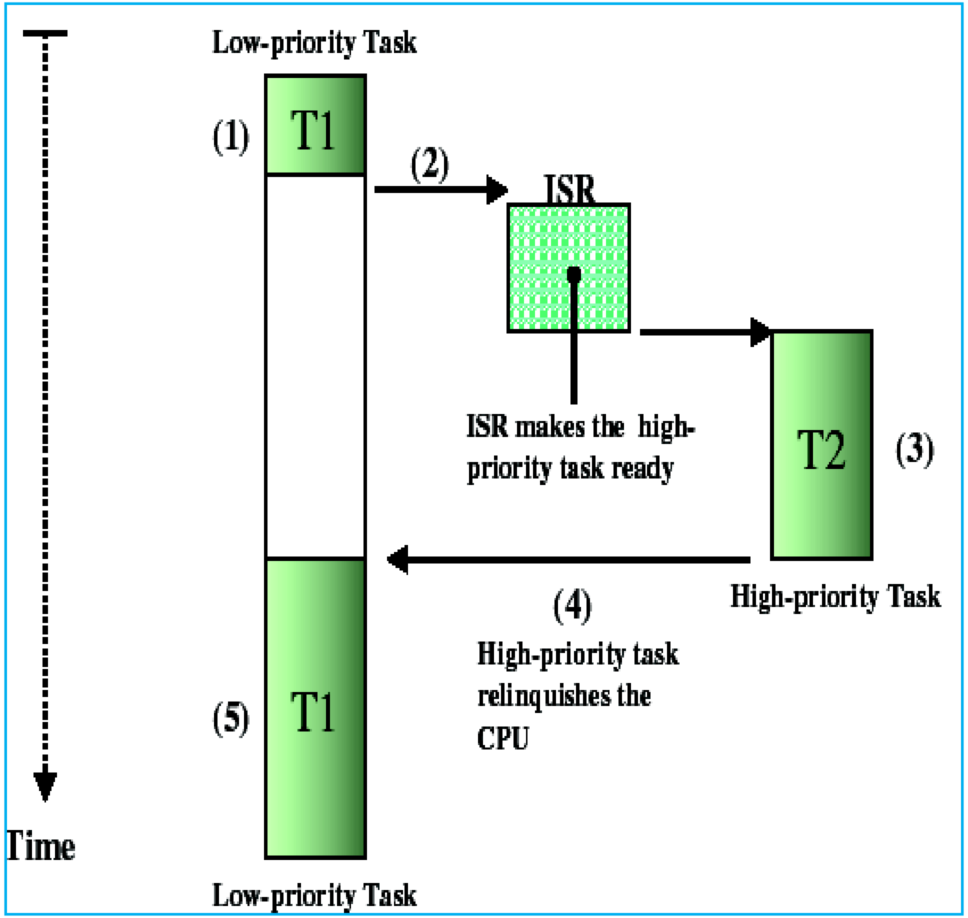
Embedded Programing

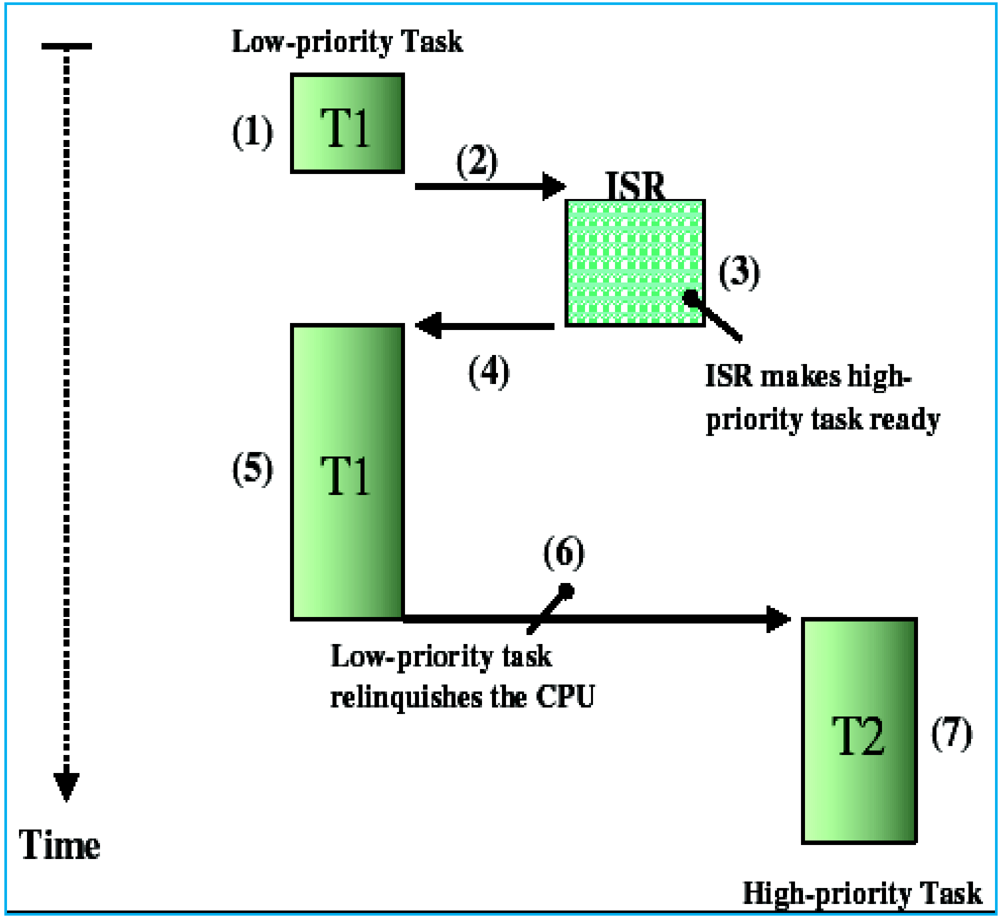
Real Time Operation System (RTOS)

- Real time task scheduling
- Resource management
- Task: a group of functions/applications. Common tasks:
  - Initialization task.
  - 1ms task
  - 5ms task
  - 10ms task
  - Background task/Algorithm task
- Scheduling: decide which task should be execute, which task should be suspensed
- Resource management: Mutex, Semaphore

Embedded Programing

Preemptive vs Non-Preemtive




Low-priority Task
(1)
T1
(2)
ISR
(3)
ISR makes high-priority task ready
(4)
(5)
T1
(6)
Low-priority task relinquishes the CPU
T2
(7)
Time
High-priority Task

Embedded Programing

Task and Timing

- What is Task Execution Time
- What is Task Deadline
- What is Task Response Time

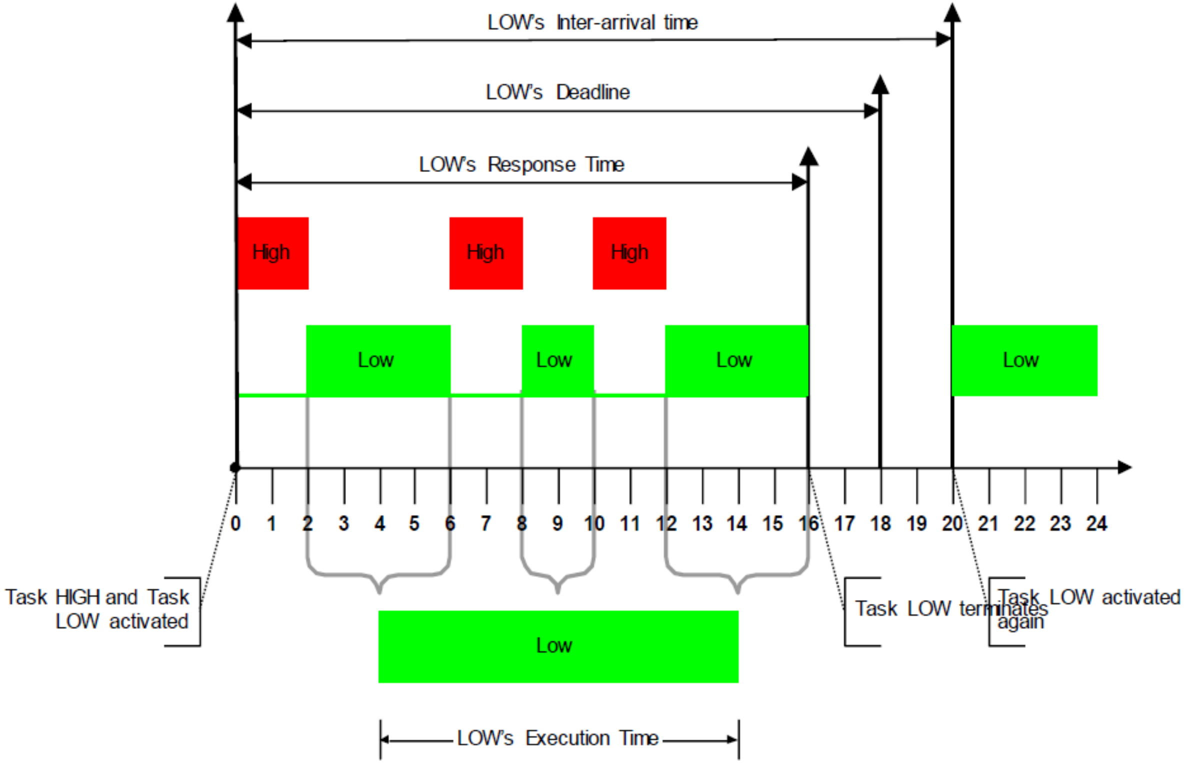
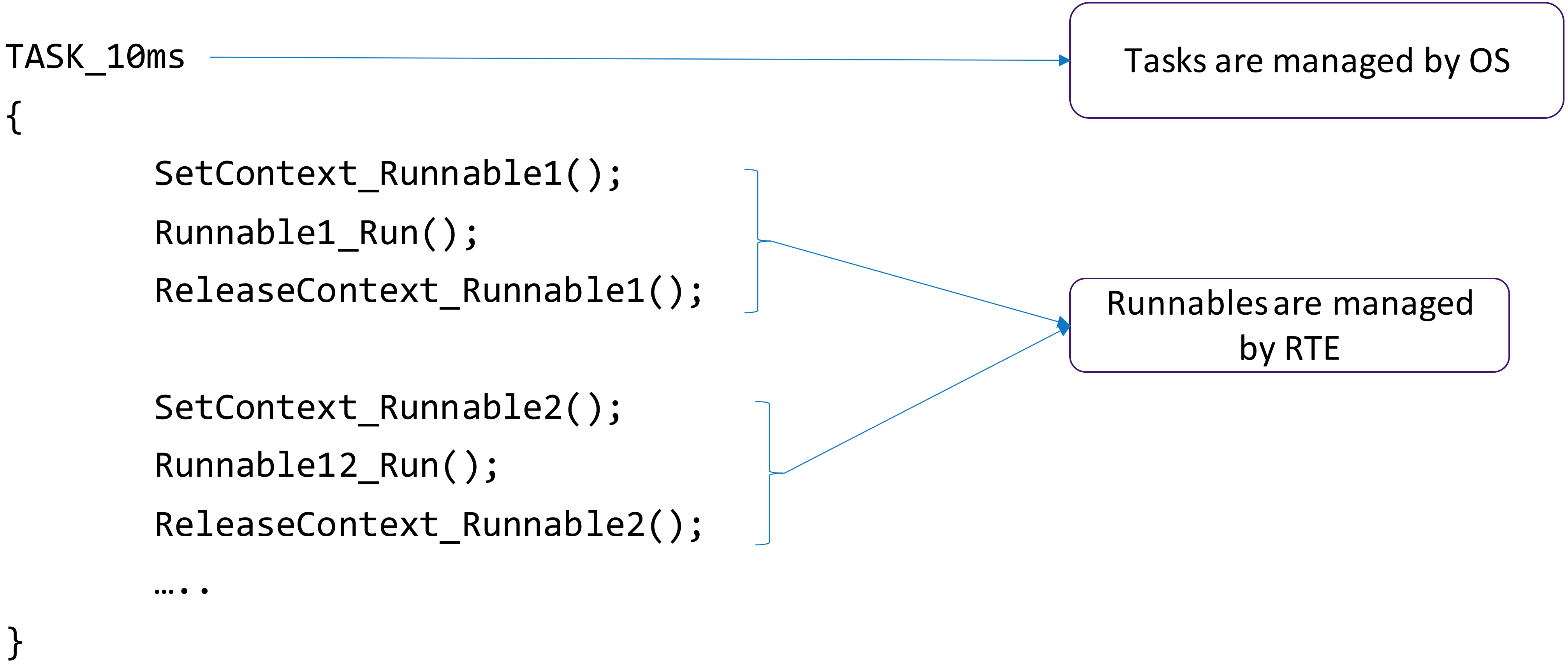


Figure 2.1: Definition of Timing Terminology

Embedded Programing

Task and runnable



```
TASK_10ms
{
        SetContext_Runnable1();
        Runnable1_Run();
        ReleaseContext_Runnable1();

        SetContext_Runnable2();
        Runnable12_Run();
        ReleaseContext_Runnable2();
        …. .
}
```

Tasks are managed by OS

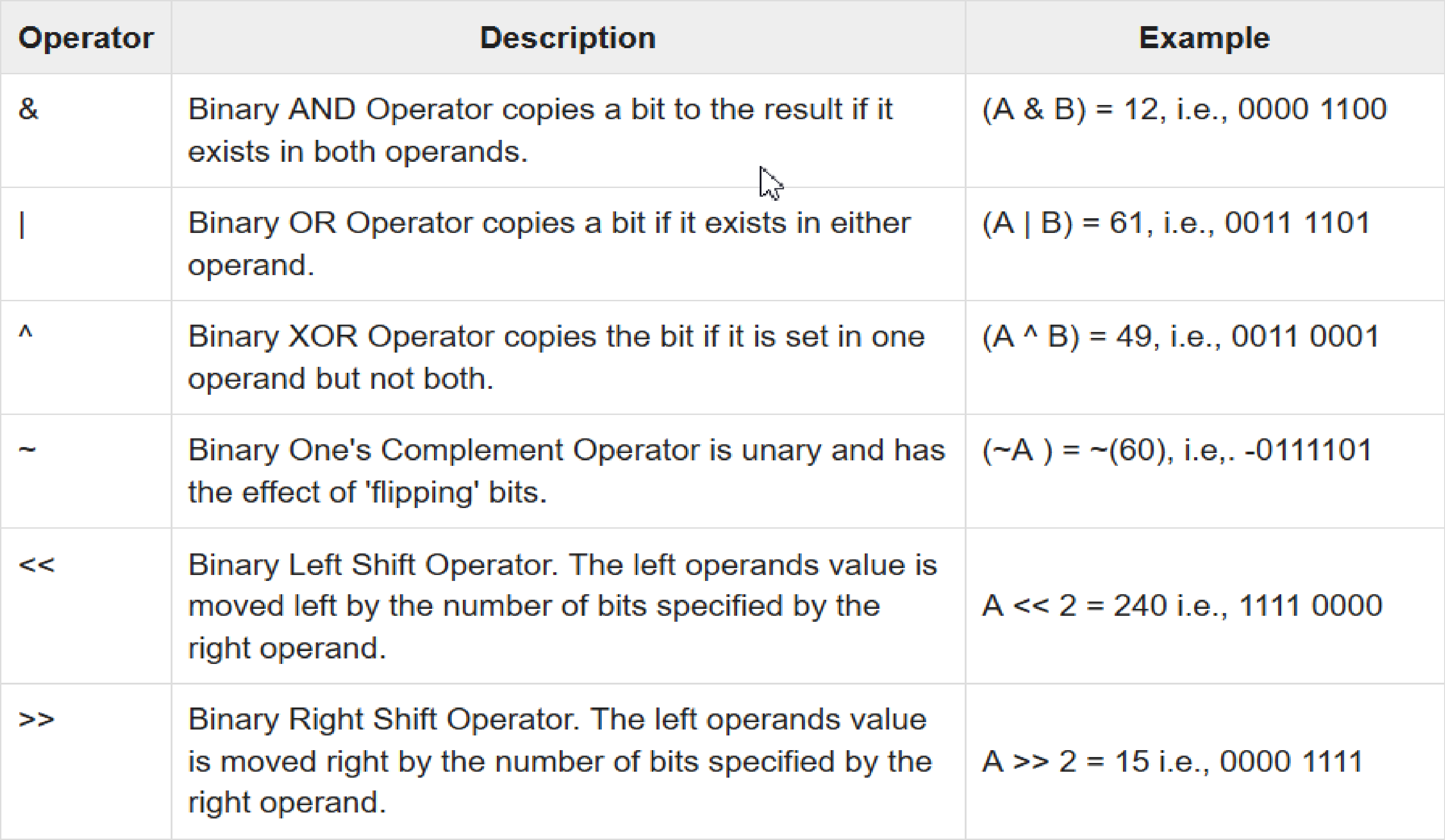
Runnables are managed by RTE

Agenda

1. Programing principles remind
   1. Programing remind/Overview
   2. Language Evaluation Criteria
   3. The Compiling Process
2. Embedded system programing
   1. What is embedded system?
   2. Which common elements inside Embedded SW?
   3. Constraints affect design choices?
   4. Why C is most common language for Embedded programing?
   5. What skills required for Embedded programmer?
   6. RTOS
3. Some advance programing topic/related topic to embedded programming.

Embedded Programming

Bitwise Operators (1)



Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) = 12, i.e., 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) = 61, i.e., 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) = 49, i.e., 0011 0001
~	Binary One's Complement Operator is unary and has the effect of 'flipping' bits.	(~A) = ~(60), i.e,. -0111101
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 = 240 i.e., 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 = 15 i.e., 0000 1111

Embedded Programming

Bitwise Operators (2)

```
unsigned char a = 0xFF;
char b = 0xFF;

printf("%d \r\n", a>>1);
printf("%d \r\n", b>>1);
```

What will be output?

A. 127 ;   127

B. 127 ;   -1

C. -1   ;   -1

D. 127;   0

Embedded Programing

Pointer remind

```
int a[] = {1, 5, 6, 7};

int* p = a;
```

```
 _ _ _ _ _ _ _
|             |
|_____________|
|      p      | 0x010
|_____________|
|     ...     |
|_____________|
|      a      | 0xF00
|_____________|
|             |
|_ _ _ _ _ _ _ |
```

▶ Determine type and value of:

1. a = ?
2. &a = ?
3. *a = ?
4. &a + 1 = ?
5. a++ = ?
6. p = ?
7. &p = ?
8. *p = ?
9. p + 1 = ?
10. *p + 1 = ?
11. &p + 1 = ?
12. *(p + 1) = ?
13. p++ = ?

Embedded Programming

const and pointer

```
int* a;                  // Pointer to int
const int* b;            // Pointer to const int
int* const c;            // Const pointer to int
const int* const d; // Const pointer to const int
```

Which statement is showing compiler error ?

1. a++;
2. b++;
3. c++;
4. d++;

5. *a = 1;
6. *b = 1;
7. *c = 1;
8. *d = 1;

Why we need to use them?

Embedded Programing

Pointer to function

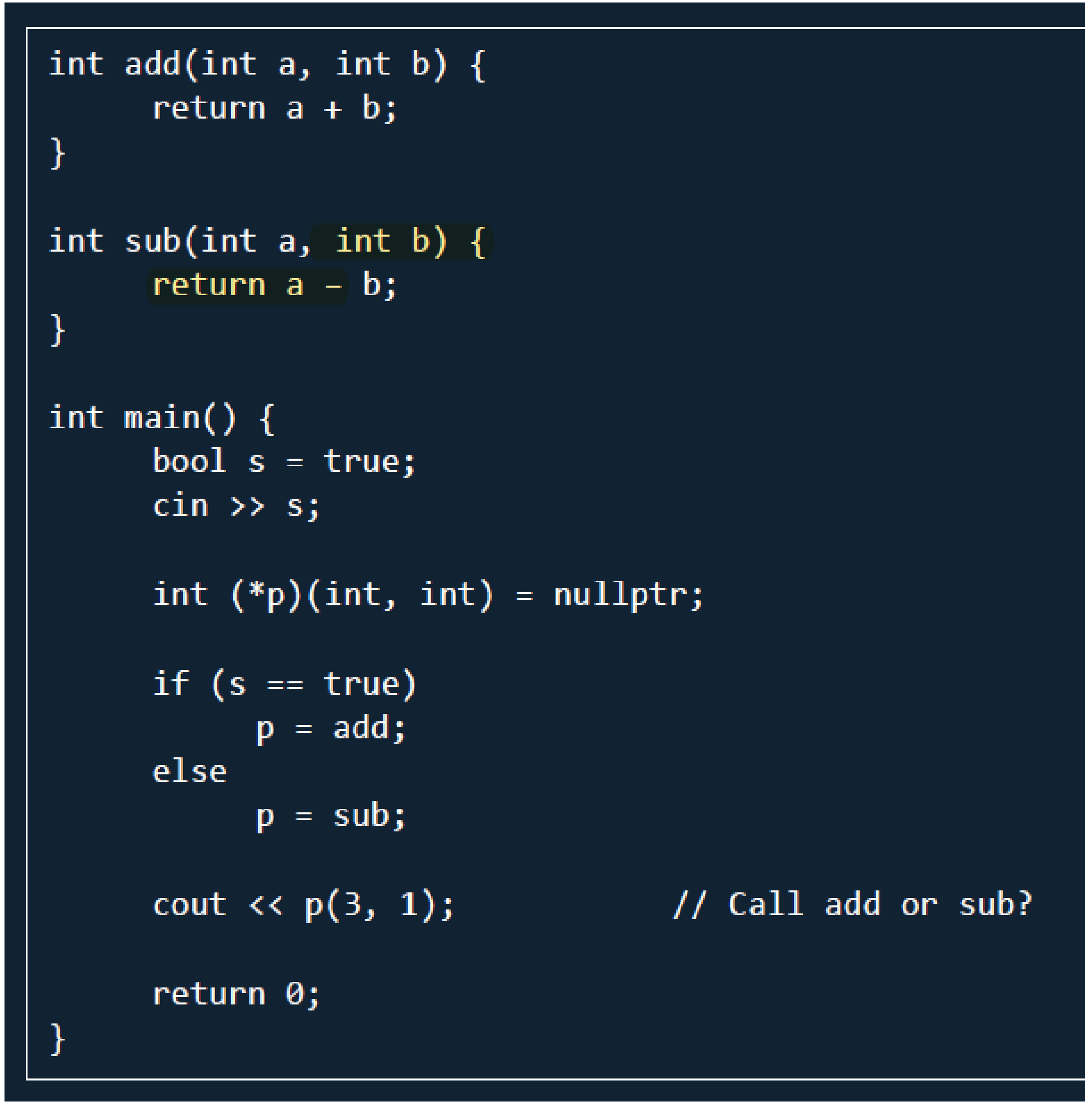
- Do not know the work of client side, make program more portable.
- Make the program easier to extend.

Syntax:

```
<returnType>* <pointerName> ([type1 param1, type2 param2, …])
```

Example:

```
void (*p)(int, int, int, int, int, int) = nullptr;
p = DrawTriangle;
```



```
int add(int a, int b) {
        return a + b;
}

int sub(int a, int b) {
        return a - b;
}

int main() {
        bool s = true;
        cin >> s;

        int (*p)(int, int) = nullptr;

        if (s == true)
                p = add;
        else
                p = sub;

        cout << p(3, 1);                // Call add or sub?

        return 0;
}
```

Embedded Programing

Switch..case vs multiple if..else

Code optimization hints

Hint 1: Inline function/Macro function

```
Sint16 g_mtl_Abs_si16(Sint16 x)(if (x) > 0 return (x); else return (-x);)
```

```
INLINE Sint16 g_mtl_Abs_si16(Sint16 x)(if (x) > 0 return (x); else return (-x);)
#define g_mtl_Abs_mac(x)          (((x) >= (0)) ? (x) : (-(x)))
```

- Use when: function is small but called many places.
- Optimize: Run faster but more code size.

Code optimization hints

Hint 2: Use switch instead of multiple if-else

```
If(x == 1){A();} else if (x == 2) {B();} else if (x == 3) {C();} else {D();}
```

```
Switch (x)
{    case 1: A(); break;
     case 2: B();break;
     case 3: C(); break;
     default: D(); break;}
```

- Use when: more than 3 specific integer comparision.
- Optimize: Run faster.

Code optimization hints

Hint 3: Use integer type for loop index/array member access

```
For(unsign byte i = 0; i >= 100; i++){ArrayBuffer[i] = 0;}
```

```
For(int i = 0; i >= 100; i++){ArrayBuffer[i] = 0;}
```

- Use when: always.
- Optimize: Run faster.

Code optimization hints

Hint 4: Use bit shift instead of division/multiplex

```
Unsign integer a, b;
a = a/2;  b = b*4;
```

```
Unsign integer a, b;
a = (unsign integer) (a>>1);  b = (unsign integer) (b<< 2);
```

- Use when: always
- Optimize: Run faster.

Code optimization hints

Hint 5: Use integer type instead of float/double number

```
Float a = 1.9;
If (a > 1.5f) { /* do something */}
```

```
Float a = 1.9;
Int b =(int)(a*10);
If (b > 15) { /* do something */}
```

- Use when: always
- Optimize: Run faster.

Code optimization hints

Hint 6: Avoid to use multiple/division operator

```
Int a = 2; Int b = 2;
a = a*2;
```

```
Int a = 2;
a = a + a;
```

- Use when: always
- Optimize: Run faster.

Code optimization hints

Hint 7: Use local variable instead of global variable

```
Extern int a; A();
Void A(void){ if(a > 1) {/* Do something */}}
```

```
Extern int a; A(a);
Void A(int b){ if(b > 1) {/* Do something */}}
```

- Use when:
- Optimize: Run faster. Take more RAM.

Code optimization hints

Hint 8: Use if branch for higher probability

```
If (a == NULL_PTR){
       /* Null pointer. Do nothing */
} else { A();}
```

```
If (a != NULL_PTR){
       A();
}
```

- Use when: always for most of Microcontroller architechture.
- Optimize: Run faster. Take more RAM.

Code optimization hints

Hint 9: Function is called only as often as needed

```
Com_ReceiveSignal(1, &l_SignalData_ui8);
If (l_SignalData_ui8 == 1) { A(); }
```

```
If(NewMsgReceived_b == TRUE){
    Com_ReceiveSignal(1, &l_SignalData_ui8);
    If (l_SignalData_ui8 == 1) { A(); }}
```

- Use when: always.
- Optimize: Run faster.

Code optimization hints

Hint 10: Reduce number of loop

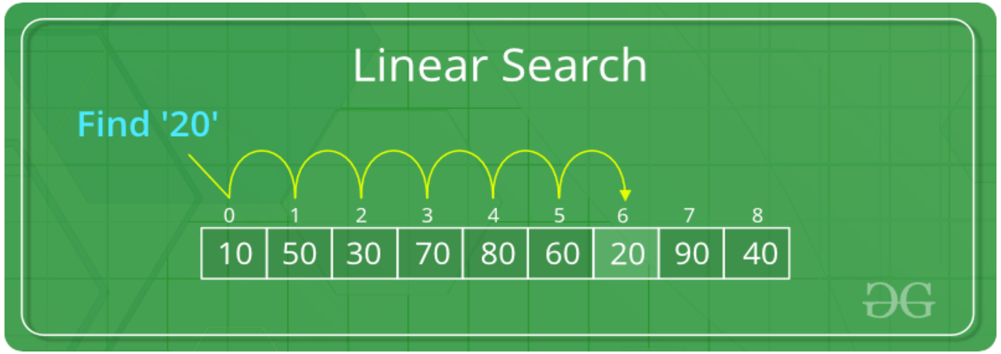
```
For (int i = 0; i < 1000; ++i){
    if(ArrayA[i] > 0) {Flag = TRUE;}
}
```

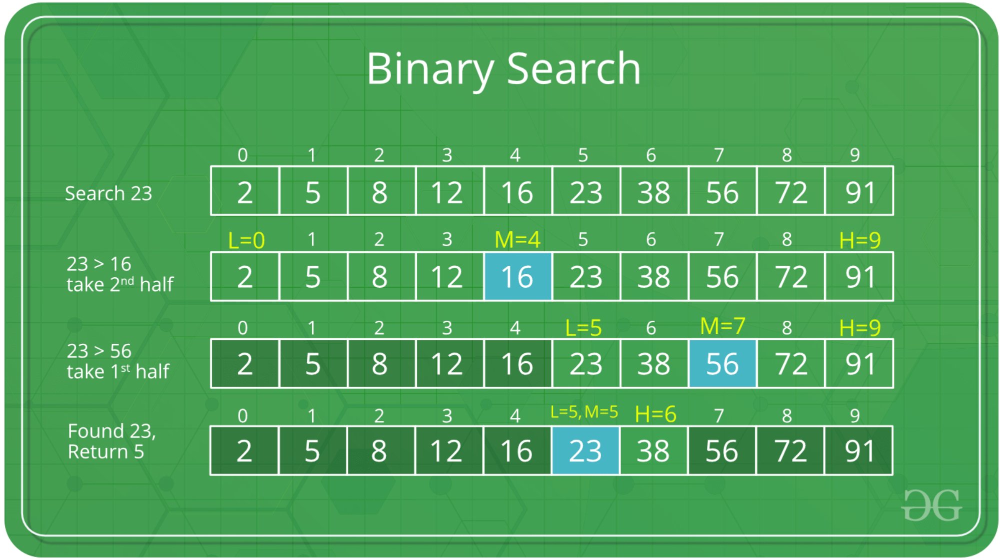
```
For (int i = 0; i < 1000; ++i){
    if(ArrayA[i] > 0) {Flag = TRUE; break;}
}
```

- Use when: always. Depend on Coding rule.
- Optimize: Run faster.

Code optimization hints

Hint 11: Use better/smarter algorithm! (1)





- Use when: always.
- Optimize: Run faster.

Agenda

1. Programing principles remind
    1. Programing remind/Overview
    2. Language Evaluation Criteria
    3. The Compiling Process
2. Embedded system programing
    1. What is embedded system?
    2. Which common elements inside Embedded SW?
    3. Constraints affect design choices?
    4. Why C is most common language for Embedded programing?
    5. What skills required for Embedded programmer?
    6. RTOS
3. Some advance programing topic
4. OOP principles basic

THANKS!