CHW 469 : Embedded Systems

Instructor:

Dr. Ahmed Shalaby http://bu.edu.eg/staff/ahmedshalaby14#

https://piazza.com/fci.bu.edu.eg/spring2017/chw469/home

Assignment no. 4

In Banha-bank branch, It is proposed to install an embedded system to monitor the client queue in front of the tellers. One of the requirement, the system should display the number of the clients in the queue. given that, both queue ends are equipped with a photocell. Each photocell generates a logic ‘1’ signal if nobody interrupts a light beam generated at the corresponding queue end. When the light beam is interrupted, the photocell output changes to logic ‘0’ and stays at that value until it is no longer interrupted. Clients are supposed to enter the queue only from the back end and leaves only from the front end.
Your team leader asks you to provide the block diagram for the system. In addition, he asks you to provide the flow chart , the required function/s declaration and the main function to implement the counting requirement .

Introduction to C Programming – How ?

- Variables and Data Types.
- Operators and Hardware Manipulation.
- Program Flow Control.
- Advanced Types, Constants and Expressions.
- Functions.
- Arrays and Pointer Basics.
- Structures and Unions.
- Scheduling Techniques.
- Arrays of Pointers.
- Declarations.
- Preprocessor.
- Real-Time Operating Systems.

C Programming / Arrays

- An array is a collection of like variables that share a single name.
- Usually, the array size is fixed.
- The individual elements of an array are referenced by a subscript.
- The first element occupies position zero. The last element is subscripted by [N-1] where N is the number of elements in the array.
- Initialize 100 elements of the array data to zero.

```
for(j=0; j<100; j++) data[j] = 0; // Initialization
```

- Strings are arrays of ASCII characters terminated with a NULL (0) character.
- If the array has three dimensions, then three subscripts are specified when referencing

```
humidity[2][3][7] = 51;
```

C Programming / Arrays

- like variables, arrays must be declared before they can be accessed.
- The number of elements in an array is determined by its declaration. Multi-dimensional arrays require multiple sets of brackets.
- The array dimensions determines how much memory to reserve.
- It is the programmer's responsibility to stay within the proper bounds. In particular, you must not let the subscript become negative for above N-1, where N is the size of the array.

```
short data[5];          /* define data, allocate space for 5 16-bit integers */
char string[20];        /* define string, allocate space for 20 8-bit characters */
int time,width[6];      /* define time, width, allocate space for 16-bit characters */
short xx[10][5];        /* define xx, allocate space for 50 16-bit integers */
short pts[5][5][5];     /* define pts, allocate space for 125 16-bit integers */
extern char buffer[]; /* declare buffer as an external character array */
```

C Programming / String Functions

- Most compliers implement string manipulation functions. The prototypes for these functions can be found in the string.h file.

```
// general-purpose memory handling routines
// search for the first unsigned 8-bit byte that matches the value in c
void *memchr(void *p, int c, size_t n);
// compare and return value will be zero if they match
int memcmp(void *p, void *q, size_t n);
// copy the data pointed to by src, placing it in the memory block pointed to by dst.
void *memcpy(void *dst, void *src, size_t n);
// copy the data even if the blocks overlap
void *memmove(void *dst, void *src, size_t);
// 8-bit bytes to the 8-bit value in c
void *memset(void *p, int c, size_t n);
```

C Programming / String Functions

```
// append a copy of the string pointed by q, placing it the end of the string to by p.
char *strcat(char *p, const char *q);
// search for the first unsigned 8-bit byte that matches the value in c.
char *strchr(const char *p, int c);
// compare and return value will be zero if they match
int strcmp(const char *p, const char *q);
//copy the string (including the null) pointed to by src, into the buffer pointed by dst
char *strcpy(char *dst, const char *src);
// compute the length of the maximal initial sub-string within the string pointed by p
size_t strcspn(const char *p, const char *q);
//  returns the length of the string pointed to by pointer p, without null
size_t strlen(const char *p);
// append the string pointed by q, placing it the end of the string pointed by p.
char *strncat(char *p, const char *q, size_t n);
/* search the string pointed to by p for the first instance of any of the characters in
the string pointed to by q */
char *strpbrk(const char *p, const char *q);
```

C Programming / Pointers

- The ability to work with memory addresses is an important feature of the C language. This feature allows programmers the freedom to perform operations similar to assembly language.
- In many situations, array elements can be reached more efficiently through pointers than by subscripting.
- A pointer is really just a variable that contains an address. Although, they can be used to reach objects in memory, their greatest advantage lies in their ability to enter into arithmetic (and other) operations, and to be changed.
- The compiler knows the format (8-bit 16-bit 32-bit, unsigned signed) of the data pointed to by the address.
- Not every address is a pointer. For instance, we can write &var when we want the address of the variable var. array name yields the address of the array. a structure name yields the address of the structure. character string yields the address of the character array specified by the string.

C Programming / Pointers

- Pointers are distinguished by an asterisk that prefixes their names.
- * means "object at" or "object pointed to by".

```
short *pt1;                 /* define pt1, declare as a pointer to a 16-bit integer */
char *pt2;                  /* define pt2, declare as a pointer to an 8-bit character */
unsigned short data,*pt3;   /* define data and pt3 unsigned 16-bit integer */
long *pt4;                  /* define pt4, declare as a pointer to a 32-bit integer */
extern short *pt5;          /* declare pt5 as a pointer to an integer */
```

C Programming / Pointers

- We can use the pointer to retrieve/store data from/to memory. Both operations are classified as pointer references.

```
long *pt;          // pointer to 32-bit data
long data;         // 32-bit
long buffer[4];    // array of 4 32-bit numbers
int main(void)
{
  pt = &buffer[1];
  *pt = 1234;
  data = *pt;
  return 1;
}
```

address	data	contents
0x20000000	0x00000000	pt
0x20000004	0x00000000	data
0x20000008	0x00000000	buffer[0]
0x2000000C	0x00000000	buffer[1]
0x20000010	0x00000000	buffer[2]
0x20000014	0x00000000	buffer[3]

pt = &buffer[1];

address	data	contents
0x20000000	0x2000000C	pt
0x20000004	0x00000000	data
0x20000008	0x00000000	buffer[0]
0x2000000C	0x00000000	buffer[1]
0x20000010	0x00000000	buffer[2]
0x20000014	0x00000000	buffer[3]

*pt = 1234;

0x20000000	0x2000000C	pt
0x20000004	0x00000000	data
0x20000008	0x00000000	buffer[0]
0x2000000C	0x00001234	buffer[1]
0x20000010	0x00000000	buffer[2]
0x20000014	0x00000000	buffer[3]

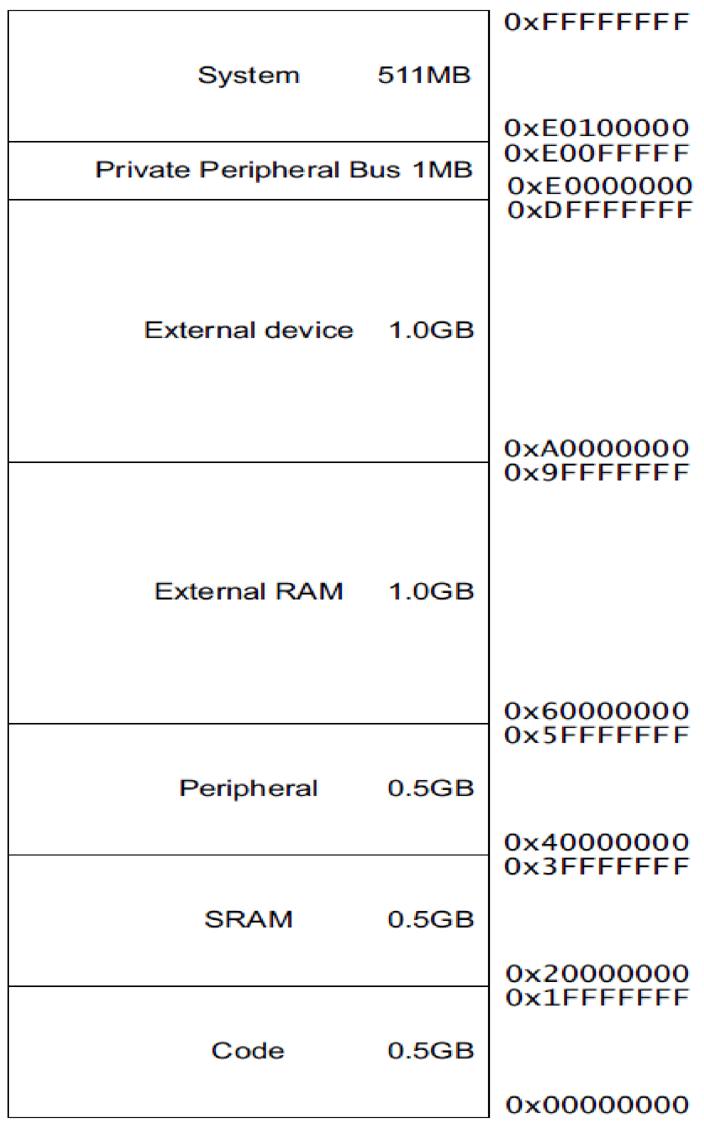
data = *pt;

0x20000000	0x2000000C	pt
0x20000004	0x00001234	data
0x20000008	0x00000000	buffer[0]
0x2000000C	0x00001234	buffer[1]
0x20000010	0x00000000	buffer[2]
0x20000014	0x00000000	buffer[3]

C Programming / Memory Addressing

- The size of a pointer depends on the architecture of the CPU and the implementation of the C compiler. Most embedded systems employ a segmented memory architecture (code, data, heap, stack) .
- Usually put global variables and local variables in RAM because these types of information can change during execution. The machine instructions and fixed constants must be stored in nonvolatile memory (ROM).
- Since an address points to an object of some particular type, adding one (for instance) to an address should direct it to the next object, not necessarily the next byte. If the address points to integers, then adding one to an integer address must actually increase the address by two.
- One of the most common mistakes: *ptr+1 is the same as (*ptr)+1 and not *(ptr+1).

Memory Maps For Cortex M0 and MCU



C Programming / Port Access

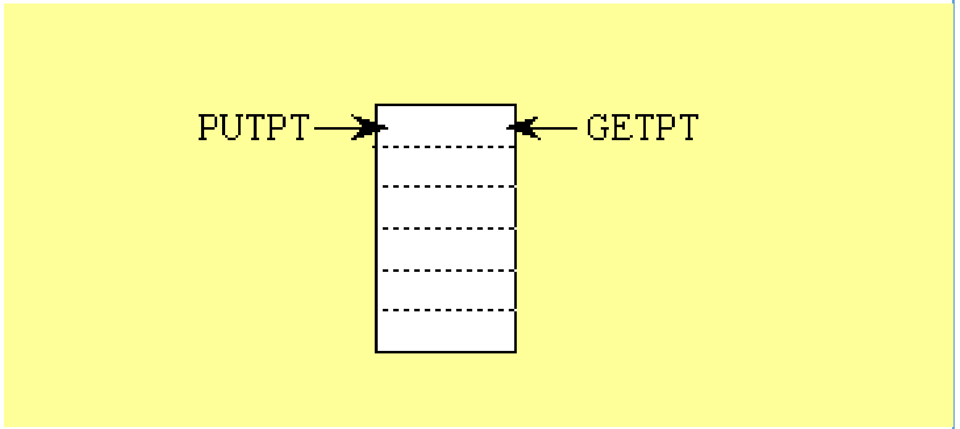
to access bit	*Constant*
7	0x0200
6	0x0100
5	0x0080
4	0x0040
3	0x0020
2	0x0010
1	0x0008
0	0x0004

```
// Port A address 0x40004000
// 0x40004080 is the address of Port A bit 5.
# define PA5   (*((volatile unsigned long *)0x40004080))
data = PA5;
becomes  data = (*((volatile unsigned long *)0x40004080));
// What If
#define PA5 0x40004080
becomes  data = 0x40004080; // does not read the contents of PA5 as desired.
// What If
#define PA5 (*0x40004080)
becomes data = (*0x40004080);
/* This will attempt to read the contents at 0x40004080, but doesn't know
whether to read 8, 16, or 32 bits. So the compiler gives a syntax error because
the type of data does not match the type of (*0x40004080).*/
```

C Programming / Pointer Comparison

- Pointers are always considered to be unsigned.
- To avoid portability problems, only addresses within a single array should be compared for relative value. To do otherwise would necessarily involve assumptions about how the compiler organizes memory.

```
/* Pointer implementation of the FIFO */
/* Number of 8 bit data in the Fifo */
#define FifoSize 10
/* Pointer of where to put next */
char *PUTPT;
/* Pointer of where to get next */
char *GETPT;
/* The statically allocated fifo data */
char Fifo[FifoSize];
void InitFifo(void)
            { PUTPT=GETPT=&Fifo[0];}
```



C Programming / Pointers – FIFO Put

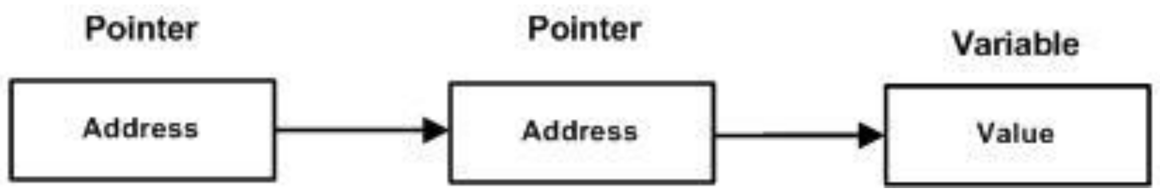
```
int PutFifo (char data)
   {        char *Ppt; /* Temporary put pointer */
            Ppt=PUTPT; /* Copy of put pointer */
            *(Ppt++)=data; /* Try to put data into fifo */
            if (Ppt == &Fifo[FifoSize])
                        Ppt = &Fifo[0]; /* Wrap */
            if (Ppt == GETPT )
                        { return(0);}   /* Failed, fifo was full */
            else{
                        PUTPT=Ppt;
                        return(-1);   /* Successful */
            }
   }
```

C Programming / Pointers – FIFO Get

```
int GetFifo (char *datapt)
 {
    if (PUTPT== GETPT){
        return(0);}   /* Empty if PUTPT=GETPT */
    else
              {
              *datapt=*(GETPT++);
              if (GETPT == &Fifo[FifoSize])
              GETPT = &Fifo[0];
              return(-1);
              }
}
```

C Programming / Pointers to

- A pointer to a pointer is a form of multiple indirection, or a chain of pointers. Normally, a pointer contains the address of a variable. When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value.



```
int  var;
int  *ptr;
int  **pptr;

var = 3000;
/* take the address of var */
ptr = &var;
/* take the address of ptr using address of operator & */
pptr = &ptr;
```

C Programming / Pointers to/from

- C programming allows passing a pointer to a function. To do so, simply declare the function parameter as a pointer type.
- C programing allows to return a pointer from a function.

```
void getSeconds (unsigned long *par); // declare a function

unsigned long sec; // define the par
getSeconds( &sec ); // call by reference



int * getRandom( );   // declare a function

int *p;  // pointer to integer
p = getRandom(); // call the function
```

C Programming / Structures

- A structure is a collection of variables that share a single name. In an array, each element has the same format.
- The types and the names is defined of each of the elements or members of the structure. The individual members of a structure are referenced by their subname.
- Structures provide a mechanism for extending the data types. Functions allow us to extend the C language to include new operations.

```
struct theport {
  unsigned char mask;    // defines which bits are active
  unsigned long volatile *addr;  // pointer to its address
  unsigned long volatile *ddr;}; // pointer to its direction reg


struct theport PortA,PortB,PortE;
```

C Programming / Structures

- typedef to actually create a new data type that behaves syntactically like char int.

```
typedef struct {
   unsigned char mask;    // defines which bits are active
   unsigned long volatile *addr;  // pointer to its address
   unsigned long volatile *ddr;} port_t; // pointer to its direction reg

port_t PortA,PortB,PortE;

PortE.mask = 0x3F;      // the TM4C123 has 6 bits on PORTE
PortE.addr = (unsigned long volatile *)(0x400243FC);
PortE.ddr = (unsigned long volatile *)(0x40024400);
(*PortE.ddr) = 0;                 // specify PortE as inputs
(*PortB.addr) = (*PortE.addr);  // copy from PortE to PortB
```

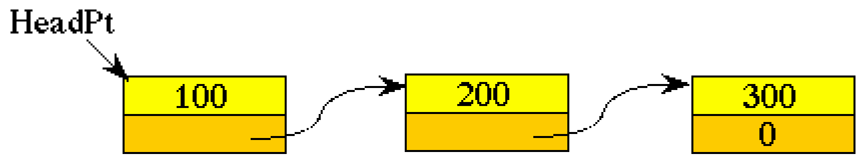
C Programming / Structures

- Like any other data type, we can pass structures as parameters to functions. Because most structures occupy a large number of bytes, it makes more sense to pass the structure by reference rather than by value.

```
int MakeInput(port_t *ppt)
        { (*ppt->ddr )= 0x00; // make input
        return 1;}

 MakeInput(&PortE);
```

- One of the applications of structures involves linking elements together with pointers. A linear linked list is a simple 1-D data structure where the nodes are chained together one after another.



C Programming / Preprocessor Directives

- The preprocessor is controlled by directives which are not part of the C language proper. Each directive begins with a #character and is written on a line by itself. Only the preprocessor sees these directive lines since it deletes them from the code stream after processing them.
- **Macro Processing** : We use macros for three reasons. 1) To save time no need to repeat many times. 2) To clarify the meaning of the software we can define a macro giving a symbolic name to a hard-to-understand sequence. 3) To make the software easy to change.
- **Conditional Compiling** : A good application of conditional compilation is inserting debugging instruments. Once the system is debugged, we can remove all the debugging code, simply by deleting the #define Debug.
- **Including Other Source Files** : The preprocessor also recognizes directives to include source code from other files.
- **Inline assembly** : __asm uses this syntax to embed assembly code into C programs

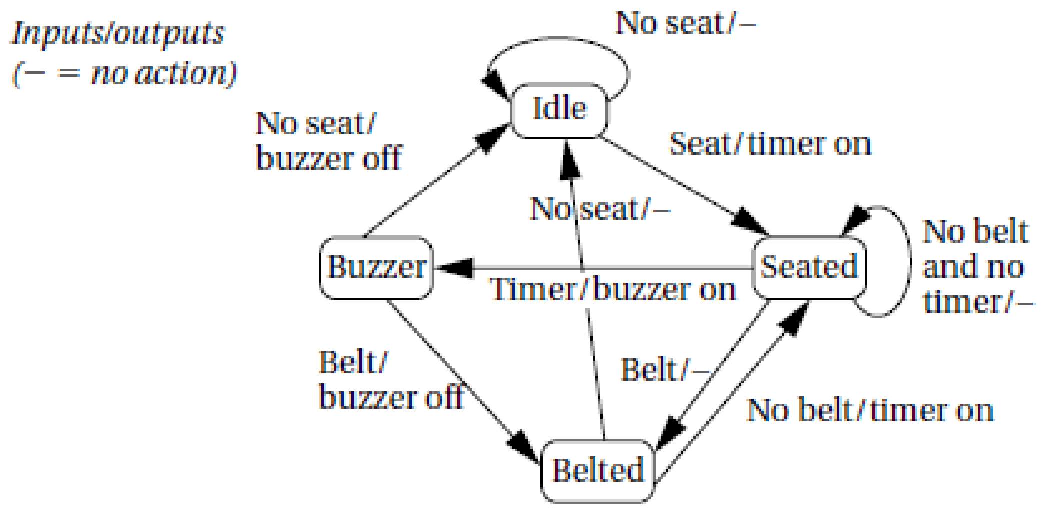
Good Enough Software, Soon Enough

- How do we make software *correct enough* without going bankrupt?
  - Need to be able to develop (and test) software efficiently
- Follow a good plan
  - Start with customer requirements
  - Design architectures to define the building blocks of the systems (tasks, modules, etc.)
  - Add missing requirements
    - Fault detection, management and logging
    - Real-time issues
    - Compliance to a firmware standards manual
    - Fail-safes

Program Design and Analysis

Components for Embedded Programs

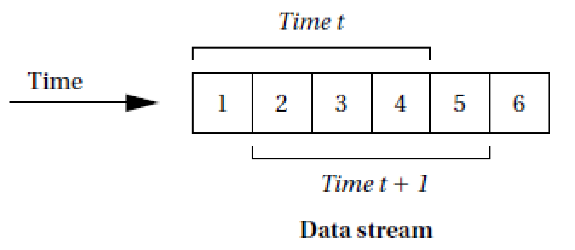
- **State Machine**
  - When inputs appear intermittently rather than as periodic samples. it is often style of describing the reactive system's behavior.

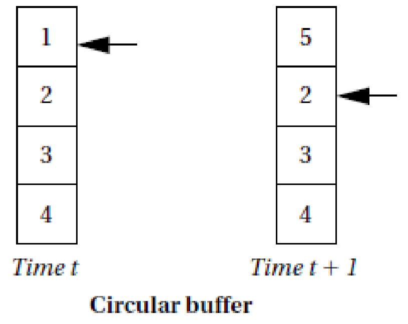


Program Design and Analysis (Cont….)

Components for Embedded Programs

- **Circular Buffers**
  - The circular buffer is a data structure that lets us handle streaming.
  - The data stream style makes sense for data that comes in regularly and must be processed on the fly.
  - A window slides with time as we throw out old values no longer needed and add new values.
- **Queues**
  - Queues are used whenever data may arrive and depart at somewhat unpredictable times or when variable amounts of data may arrive. A queue is often referred to as an elastic buffer.
  - Implementations: One way to build a queue is with a linked list allows arbitrary size growth But in dynamic memory allocating is expensive another to use an array to hold all the data.





Program Optimization

- **Expression Simplification**
  - Algebra simplification -- $a*b + a*c \rightarrow a*(b + c)$
  - Constant simplification -- $i < 8 + 1 \rightarrow i < 9$
- **Dead Code Elimination**
  - Code that will never be executed can be safely removed from the program.
  - Copy one variable to another can be replaced by references first one to the second one.
- **Procedure Inlining**
  - The body of the procedure is substituted in place of the procedure call.
  - Inlining is not always the best thing to do. Although it does eliminate the procedure linkage instructions But also increases code size, and memory may be precious.

Program Optimization (Cont…)

- **Loop Transformations**
  - Loop unrolling helps expose parallelism But procedure—it may interfere with the cache and expands the amount of code required.
  - Loop fusion combines two or more loops into a single loop.
  - Loop distribution is decomposing a single loop into multiple loops.
  - Loop tiling breaks up a loop into a set of nested loops, with each inner loop performing the operations on a subset of the data.
  - Array padding adds dummy data elements to a loop in order to change the layout of the array in the cache.

Program Optimization (Cont…)

- **Register Allocation**
  - A naive register allocation, assigning each variable to a separate register. However, we can do much better by reusing a register once the value stored in the register is no longer needed.
  - If a section of code requires more registers than are available, we must spill some of the values out to memory temporarily.
  - Spilling registers is problematic. It requires extra CPU time and uses up both instruction and data memory.
  - Solve register allocation problems can be by building a conflict graph and solving a graph coloring problem.
  - Graph coloring is NP-complete, but there are efficient heuristic algorithms that can give good results on typical register allocation problems.

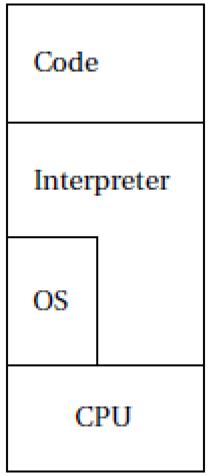
Program Optimization (Cont…)

- **Scheduling**
  - Choosing the order in which operations will be performed. Thereby changing the lifetimes of the variables (The time the variable live in Memory).
  - Scheduling is used to improve register allocation, maximize performance.
  - Solve scheduling problems by keep tracking of resource utilization over time.
  - Track of CPU resources during instruction scheduling using a reservation
  - table (rows in the table represent instruction execution time slots and columns represent resources that must be scheduled).
  - Software pipelining is a technique for reordering instructions across several loop iterations to reduce pipeline bubbles. Rather than pad the loop with no-ops, we can start instructions from the next iteration.

Time	Resource A	Resource B
t	X	
t + 1	X	X
t + 2	X	
t + 3		X

Program Optimization (Cont…)

- **Instruction Selection**
  - Selecting the instructions to use to implement each operation.
  - Each instructions has a cost (execution time, size).
- **Understanding and Using Your Compiler**
  - Studying the assembly language output of the compiler is a good way to learn about what the compiler does.
  - Writing the same algorithm in several ways to see how the compiler's output changes.
- **Interpreters**
  - Translate the program into instructions during execution.
  - Interpretation or JIT compilation adds overhead—both time and memory.
  - Interpretation and JIT compilation provide added security.
  - Interpreter sits between the program and the machine. It translates one statement of the program at a time.



Measure program performance

- **measure program performance**
  - Simulators to measure the execution time of the program. Not 100% accurate.
  - Timer connected to the microprocessor bus can be used to measure performance of executing sections of code.
  - Logic analyzer connected to the microprocessor bus to measure the start and stop times of a code segment.

- **Three different types of performance measures**
  - Average-case execution time: This is the typical execution time
  - Worst-case execution time: The longest time that the program can spend
  - Best-case execution time: The shortest time. This measure can be important in multirate real-time system.