Real Time & Embedded Systems

C Language Programming

Selected Topics

Agenda

- A brief history of C
- Logical and Bit operations
- Shifting and Inversion
- Arrays and Pointers
- C Structures (struct)
- Constant qualifier (const)
- Symbolic Names (typedef)

A brief history of C

A Bit of History

- Developed in the early to mid 70s
  - Dennis Ritchie as a systems programming language.
  - Adopted by Ken Thompson to write Unix on a the PDP-11.
- At the time:
  - Many programs written in assembly language.
  - Most systems programs (compilers, etc.) in assembly language.
  - Essentially ALL operating systems in assembly language.
- Proof of Concept
  - Even small computers could have an OS in a HLL.
  - Small: 64K bytes, 1µs clock, 2 MByte disk.
  - We ran *5 simultaneous users* on this base!

Why C?

C is a good choice for embedded systems programming because

- It is a relatively defeatured, simple to learn, understand, program and debug.
- C Compilers are available for almost all embedded devices in use today!!
- Many/most support libraries for embedded systems are written in C.
- Unlike assembly, C has advantage of processor-independence and is not specific to any particular microprocessor/ microcontroller or any system. It is very portable.
- C is a mid- to high-level language that is is fairly efficient (size, speed)
- It supports access to I/O and provides ease of management of large embedded projects.

Logical and Bitwise Operators

Logical Operators

- A logical operator is used to combine 2 or more conditions in an expression.
- Logical AND - &&
  - Operator **&&** returns true when both the conditions in consideration are true; else false
- Logical OR - ||
  - Operator **||** returns true when either or both the conditions in consideration are true; else false
- Logical NOT - !
  - Operator **!** returns true when either or both the conditions in consideration are true; else false
- Logical XOR
  - In the Boolean sense, this is just **!=** **(not equal)**

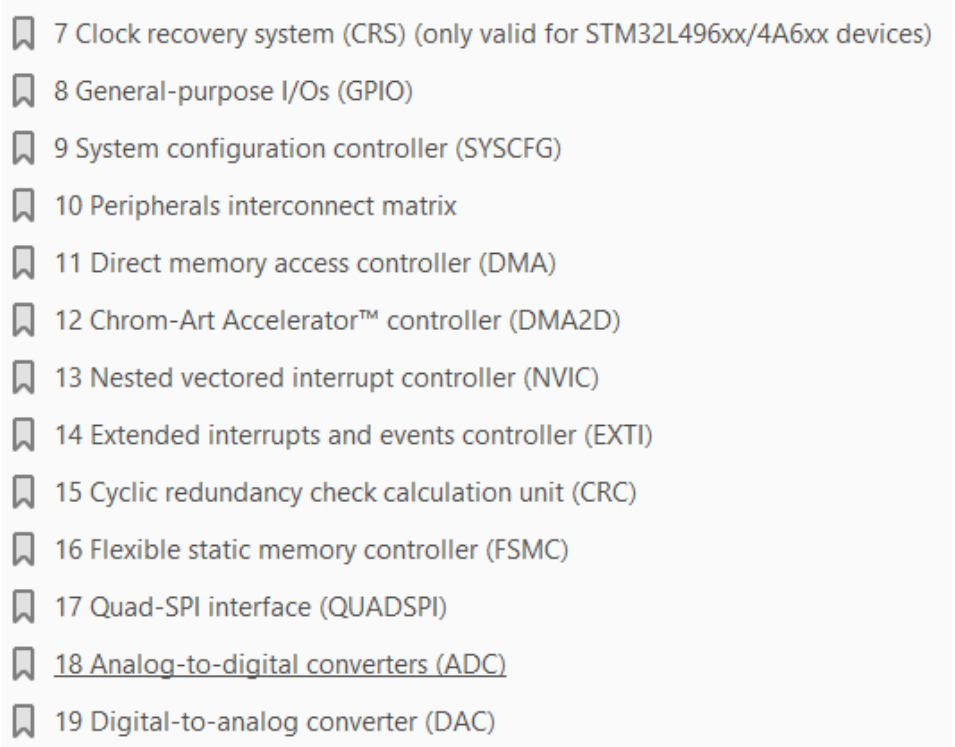
Logical example

```
int a = 10, b = 4, c = 10, d = 20;

// logical AND example
if (a > b && c == d)
    printf("a is greater than b AND c is equal to d\n");
    // doesn't print because c != d

// logical OR example
if (a > b || c == d)
    printf("a is greater than b OR c is equal to d\n");
    // NOTE: because a>b, the clause c==d is not evaluated

// logical NOT example
if (!a)
    printf("a is zero\n");   // doesn't print because a != 0
```

Software Engineering
Rochester Institute of Technology

Bitwise Operators

- A key feature of C essential to RT & ES programming is the set of bit manipulations
- Microcontrollers are filled with pages and pages of registers that control MCU peripheral hardware. These are all bit-based definitions.
- Some peripherals from STM32 Reference Manual…



23.5 OPAMP registers

23.5.1 OPAMP1 control/status register (OPAMP1_CSR)

Address offset: 0x00

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
OPA_RANGE	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
rw															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CAL OUT	USER TRIM	CAL SEL	CALON	Res.	VP_SEL	VM_SEL		Res.	Res.	PGA_GAIN		OPAMODE		OPA LPM	OPAEN
r	rw	rw	rw		rw	rw	rw			rw	rw	rw	w	rw	rw

Bit 31 **OPA_RANGE:** Operational amplifier power supply range for stability
All AOP must be in power down to allow AOP-RANGE bit write. It applies to all AOP embedded in the product.
0: Low range (VDDA < 2.4V)
1: High range (VDDA > 2.4V)

Bits 30:16 Reserved, must be kept at reset value.

Bit 15 **CALOUT:** Operational amplifier calibration output
During calibration mode offset is trimmed when this signal toggle.

Bit 14 **USERTRIM:** allows to switch from 'factory' AOP offset trimmed values to AOP offset 'user' trimmed values
This bit is active for both mode normal and low-power.
0: 'factory' trim code used
1: 'user' trim code used

Bit 13 **CALSEL:** Calibration selection
0: NMOS calibration (200mV applied on OPAMP inputs)
1: PMOS calibration (VDDA-200mV applied on OPAMP inputs)

Bit 12 **CALON:** Calibration mode enabled
0: Normal mode
1: Calibration mode (all switches opened by HW)

38.6.4 RTC initialization and status register (RTC_ISR)

This register is write protected (except for RTC_ISR[13:8] bits). The write access procedure is described in *RTC register write protection on page 1193*.

Address offset: 0x0C

Backup domain reset value: 0x0000 0007

System reset: not affected except INIT, INITF, and RSF bits which are cleared to '0'

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	ITSF	RECALPF
														rc_w0	r
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TAMP3F	TAMP2F	TAMP1F	TSOVF	TSF	WUTF	ALRBF	ALRAF	INIT	INITF	RSF	INITS	SHPF	WUTWF	ALRBWF	ALRAWF
rc_w0	rc_w0	rc_w0	rc_w0	rc_w0	rc_w0	rc_w0	rc_w0	rw	r	rc_w0	r	r	r	r	r

Bits 31:18 Reserved, must be kept at reset value

Bit 17 **ITSF**: Internal tTime-stamp flag

This flag is set by hardware when a time-stamp on the internal event occurs.

This flag is cleared by software by writing 0, and must be cleared together with TSF bit by writing 0 in both bits.

Bit 16 **RECALPF**: Recalibration pending Flag

The RECALPF status flag is automatically set to '1' when software writes to the RTC_CALR register, indicating that the RTC_CALR register is blocked. When the new calibration settings are taken into account, this bit returns to '0'. Refer to *Re-calibration on-the-fly*.

Bit 15 **TAMP3F**: RTC_TAMP3 detection flag

This flag is set by hardware when a tamper detection event is detected on the RTC_TAMP3 input.

It is cleared by software writing 0

Bit 14 **TAMP2F**: RTC_TAMP2 detection flag

This flag is set by hardware when a tamper detection event is detected on the RTC_TAMP2 input.

It is cleared by software writing 0

C Bitwise Operators

C has 6 operators for performing bitwise operations on integers

Operator	Meaning	
&	Bitwise AND	Result is 1 if both bits are 1
|	Bitwise OR	Result is 1 if either bit is 1
^	Bitwise XOR	Result is 1 if both bits are different
>>	Right shift	
<<	Left shift	
~	Ones complement	The logical invert, same as NOT

Bitwise Boolean examples

```
char j = 11;     // 0 0 0 0 1 0 1 1 = 11
char k = 14;     // 0 0 0 0 1 1 1 0 = 14
```

Bitwise Boolean Operators

```
char m = j & k; // 0 0 0 0 1 0 1 0 = 10
char n = j | k; // 0 0 0 0 1 1 1 1 = 15
char p = j ^ k; // 0 0 0 0 0 1 0 1 = 5
```

NOTE: This is a logical (not Boolean) operation

```
bool q = j && k;    // true == 1
bool q = 0 && k;    // false == 0
```

Shifting and Inversion

Shifting

Shifting

```
char j = 11;      // 0 0 0 0 1 0 1 1 = 11
char k = j<<1;    // 0 0 0 1 0 1 1 0 = 22 (j*2)
char m = j>>1;    // 0 0 0 0 0 1 0 1 = 5  (j/2)
```

Shifting

```
char s1, s2, s3, s4;
s1=-11;          // 1 1 1 1 0 1 0 1 -11
s2=s1>>1;        // 1 1 1 1 1 0 1 0  -6

s3=117;          // 0 1 1 1 0 1 0 1  117
s4=s3>>1;        // 0 0 1 0 0 0 0 0  58
                 // sign extension!

unsigned char u1, u2;
u1=255;          // 1 1 1 1 0 1 0 1  245
u2=u1>>1;        // 0 1 1 1 1 1 1 1  122
                 // no sign extension!
```

Inversion

Logical invert

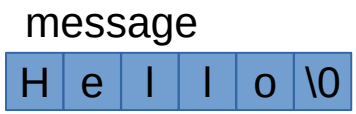
```
char j = 11;      // j = 0 0 0 0 1 0 1 1 =  11
char k = ~j;      // k = 1 1 1 1 0 1 0 0 = 244
                  // Note:         j + k = 255
```

Arrays and pointers

Software Engineering
Rochester Institute of Technology

Array Identifiers & Pointers

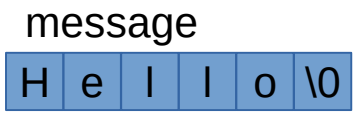
- `char message_array[] = “Hello” ;`



- Question: So what exactly is **message**?
- Answer: In C, an array name is a constant pointer that references the 0th element of the array's storage.
- **Constant** means it cannot be changed (just as we can't change the constant 3).

Consequences - Part 1

- `char message_array[] = “Hello” ;`
- `char *message = “Hello”;`



Question: What is ***message**?

- `*message == ‘H’;` `// an array pointer. It points to the`
  `// start of the array (to 0th element)`

Read `*message` as “what message points to”

What is another expression for message?

- `message == &message[0];` `message[0]==‘H’`

Pointer Variables and Arrays - 1

```
char *hi = “Hello” ;
```

Allocates space and initializes a constant string “Hello”, then
allocates space for pointer hi and initializes it to point to the 0th element.

```
char message[] = “Greetings!” ;
```

Allocates space for the array message and initializes its contents to the string “Greetings!”.

```
char *p_mesg = message ;
```

Allocates space for pointer p_mesg and initializes it to point to message.

```
char ch ;          // Declares ch as a char
p_mesg++ ;         // Advance p_mesg by one element (char in this case)
ch = *p_mesg ;     // Set ch to the character p_mesg points to (in this case 'r').
```

C Structures

C Structs

- A *struct* is a way of grouping named, heterogeneous data elements that represent a coherent concept.

C Structs

- A struct is a way of grouping named, heterogeneous data elements that represent a coherent concept.
- Example:

```
#define MAXNAME (20)

struct person {
    char name[MAXNAME+1] ;
    int age ;
    double income ;
} ;
```

C Structs

- Question: What is an object with no methods and only instance variables public?
- Answer: A struct! (well, sort of).
- A struct is a way of grouping named, heterogeneous data elements that represent a coherent concept.
- Example:

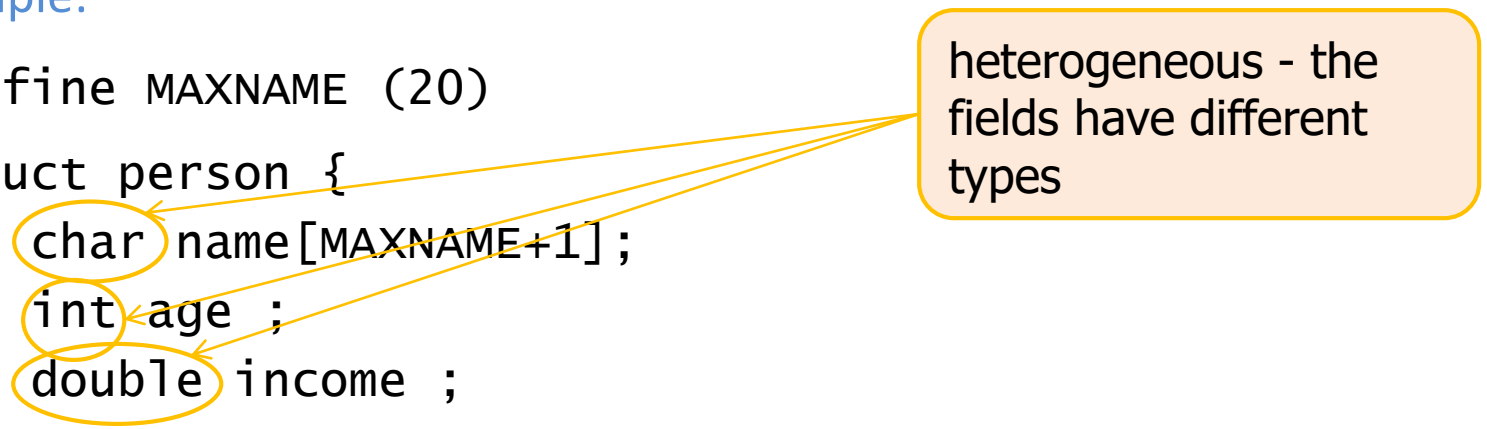
```
#define MAXNAME (20)
struct person {
  int age ;
  double income ;
} ;
```

coherent concept - the information recorded for a person.

C Structs

- Question: What is an object with no methods and only instance variables public?
- Answer: A struct! (well, sort of).
- A struct is a way of grouping named, heterogeneous data elements that represent a coherent concept.
- Example:

```
#define MAXNAME (20)

struct person {
    char name[MAXNAME+1];
    int age ;
    double income ;
} ;
```



Software Engineering
Rochester Institute of Technology

C Structs

- Question: What is an object with no methods and only instance variables public?
- Answer: A struct! (well, sort of).
- A struct is a way of grouping named, heterogeneous data elements that represent a coherent concept.
- Example:

```
#define MAXNAME (20)

struct person {
    char name[MAXNAME+1] ;
    int age ;
    double income ;
} ;
```

the field names in the struct

Using Structs

- Declaration:

```
struct person {
    char name[MAXNAME+1] ; // explicit size known
    char *title;          // a pointer has explicit size
    char profession[];  // ILLEGAL, size not known
    int age ;
    double income ;
} ;
```

- Definitions:

```
struct person mike, pete, chris ;
```

- Assignment / field references ('dot' notation):

```
mike = pete ;        // this does a shallow copy!!
   // If the structure contains pointers, the pointers will be
   // copied, but not what they point to.  Thus, after the copy,
   // there will be two pointers pointing to the same memory.
pete.age = chris.age + 3;
```

Using Structs

- Note: Space allocated for the whole struct at definition.
- Struct arguments are passed by value (i.e., copying)

WRONG

```
void give_raise( struct person p, double pct) {
    p.income *= (1 + pct/100) ;
    return ;
}

give_raise(mike, 10.0); // what is mike's income after raise
```

RIGHT

```
struct person give_raise(struct person p, double pct) {
    p.income *= (1 + pct/100) ;
    return p ;
}

mike = give_raise(mike, 10.0) ; // what is mike's income after raise?
```

Using Structs pointers

- Better if you can pass a pointer to the structure

```
void give_raise(struct person *p, double pct) {
    p->income *= (1 + pct/100) ;
    return ;
}

give_raise(&mike, 10.0) ;
```

Const qualifier

Const qualifier

- The const qualifier applied to a declared variable states the value cannot be modified.
- Using this feature can help prevent coding errors.
- Good for settings and configurations.

const char * - a pointer to a const char
the value being pointed to can't be changed but the pointer can.

char * const - is a constant pointer to a char
the value can be changed, but the pointer can‘t

Order can be confusing…

Const qualifier cont.

- To avoid confusion, always *append* the const qualifier.

```
int * mutable_pointer_to_mutable_int;

int const * mutable_pointer_to_constant_int;

int * const constant_pointer_to_mutable_int;

int const * const constant_ptr_to_constant_int;
```

Symbolic Names

typedef

Symbolic Type Names - typedef

- Suppose we have a pricing system that prices goods by weight.
  - Weight is in pounds, and is a double precision number.
  - Price is in dollars, and is a double precision number.
  - Goal: Clearly distinguish weight variables from price variables.

Symbolic Type Names - typedef

- Suppose we have a pricing system that prices goods by weight.
  - Weight is in pounds, and is a double precision number.
  - Price is in dollars, and is a double precision number.
  - Goal: Clearly distinguish weight variables from price variables.
- Typedef to the rescue:
  - typedef ***declaration*** ;

  Creates a new "type" with the variable slot in the ***declaration***.

Symbolic Type Names - typedef

- Suppose we have a pricing system that prices goods by weight.
  - Weight is in pounds, and is a double precision number.
  - Price is in dollars, and is a double precision number.
  - Goal: Clearly distinguish weight variables from price variables.
- Typedef to the rescue:
  - typedef ***declaration*** ;Creates a new "type" with the variable slot in the ***declaration***.
- Examples:

```
typedef double PRICE_t;  // alias for double to declare price variables
typedef double WEIGHT_t; // alias for double to declare weight variables
PRICE_t p ;              // double precision value that's a price
WEIGHT_t lbs ;           // double precision value that's a weight
```

typedef In Practice

- Symbolic names for array types

```
#define  MAXSTR  (100)

typedef  char  LONG_STRING_t[MAXSTR+1] ;

LONG_STRING_t line ;
LONG_STRING_t buffer ;
LONG_STRING_t *p_long_string;
```

typedef In Practice

- Symbolic names for array types

```
#define MAXSTR (100)

Typedef char LONG_STRING_t [MAXSTR+1] ;

LONG_STRING_t  line ;
LONG_STRING_t  long_string;
```

- Shorter name for struct types:

```
typedef struct  {
    LONG_STRING_t label ; // name for the point (fixed length)
    double  x ;           // x-coordinate
    double  y ;           // y-coordinate
}  POINT_t;

POINT_t origin ;
POINT_t focus ;
POINT_t *p_point = origin;
```