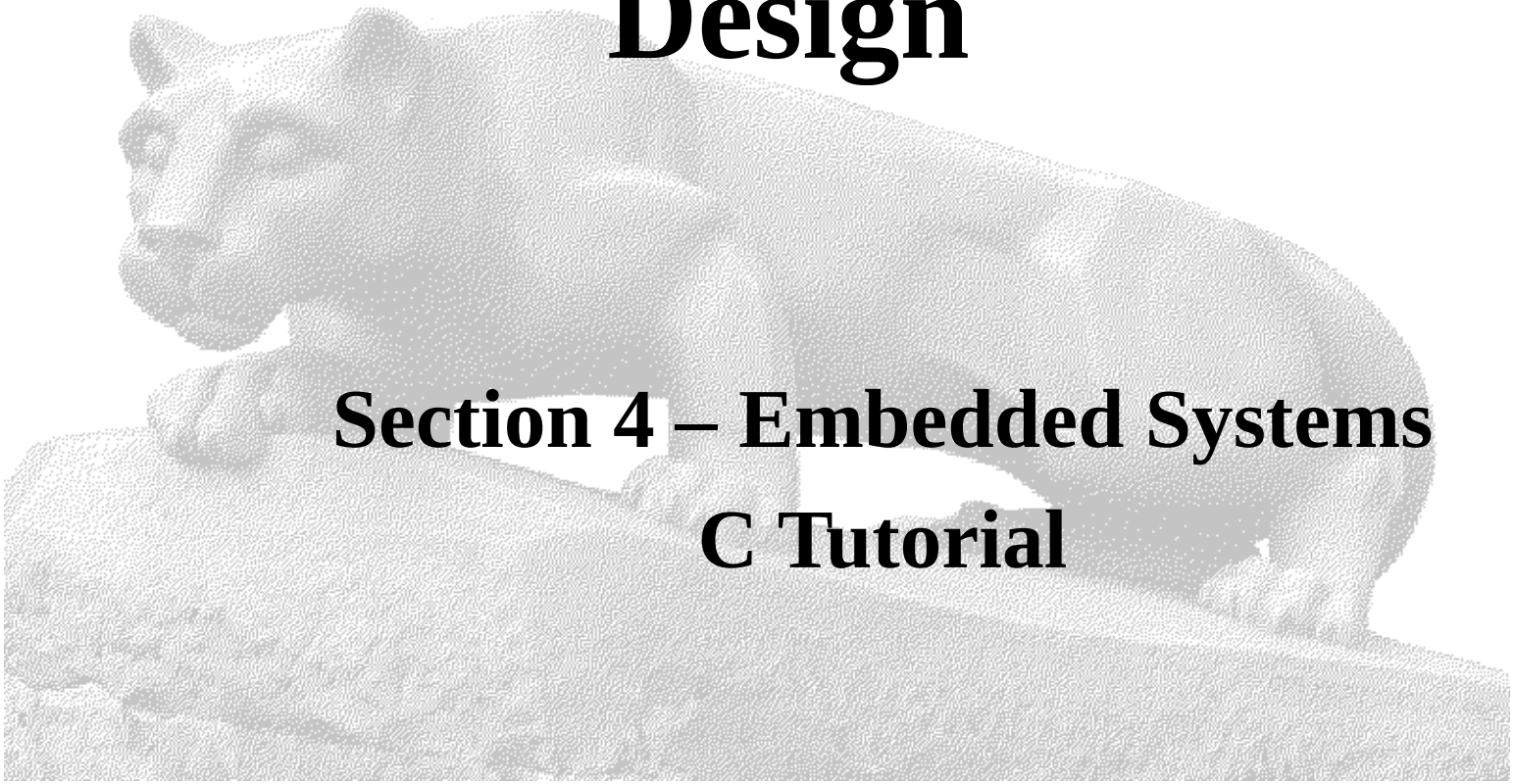
PENNSTATE

EE403W Senior Project Design



Section 4 – Embedded Systems

C Tutorial

QUIZ

```
// initialization section
unsigned char x = 2;

// execution section
if (x = 7)
    x = 0;
```

- After this code runs, what is the value stored in the memory location referenced by the variable ‘x’?

‘C’ Programming Language

Why program microcontrollers in ‘C’?

- More compact code (visually speaking)
- Less cryptic code – easier to understand
- Easier to maintain/update
- Easier to manage large projects w/ multiple programmers
- More portable (to an extent)
- ‘C’ is a more marketable skill (than BASIC, etc)

Why NOT program in ‘C’?

- $$$ for a compiler
- Assembly potentially more compact code (memory size, execution speed)
  - Assuming you have a competent assembly programmer on-hand
- May be quicker/easier for very small projects to code in ASM (< 1kbyte)

‘C’ Programming Language

C vs. Assembly Language

C	Assembly	Machine
i = 3	LDAA #$03	4000:86 03
	STAA $800	4002:7A 08 00
j = 5	LDAA #$05	4005:86 05
	STAA $801	4007:7A 08 01
k = i + j	LDAA $800	400A:B6 08 00
	ADDA $801	400D:BB 08 01
	STAA $803	4010:7A 08 03

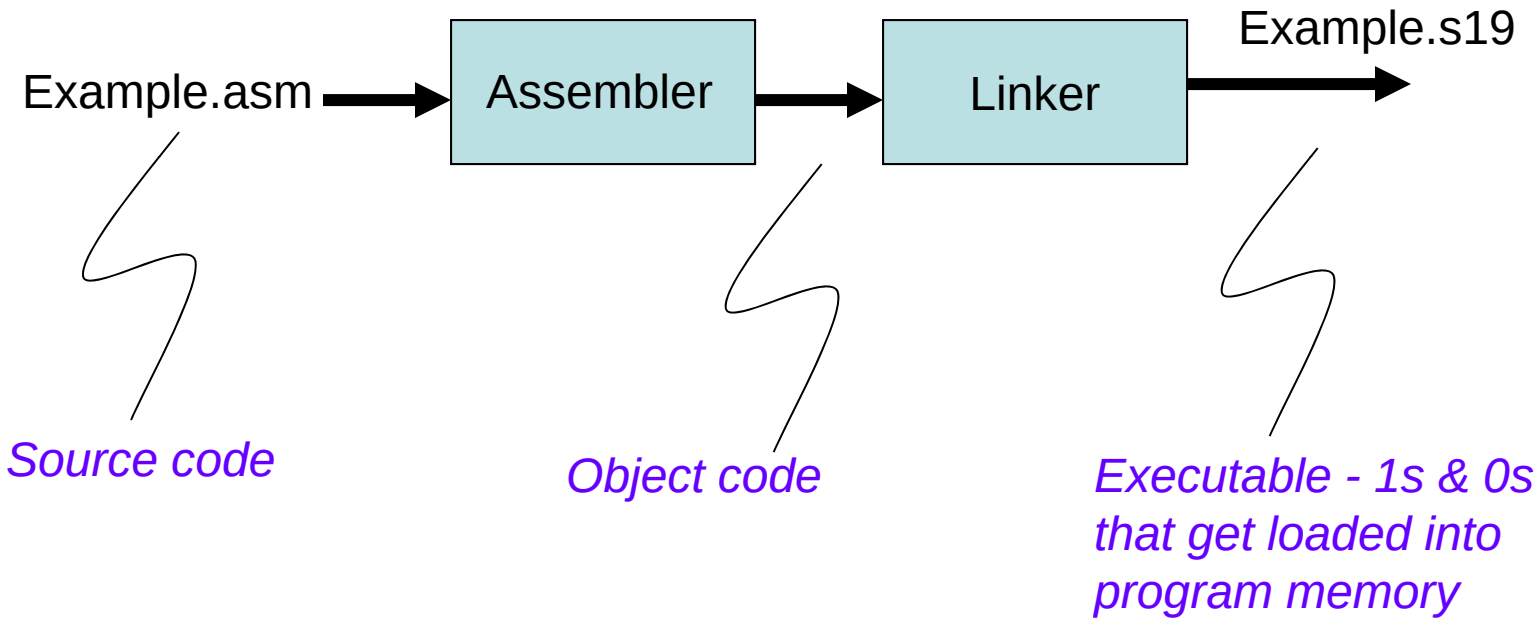
Compiler converts

Assembler converts

Downloaded to chip

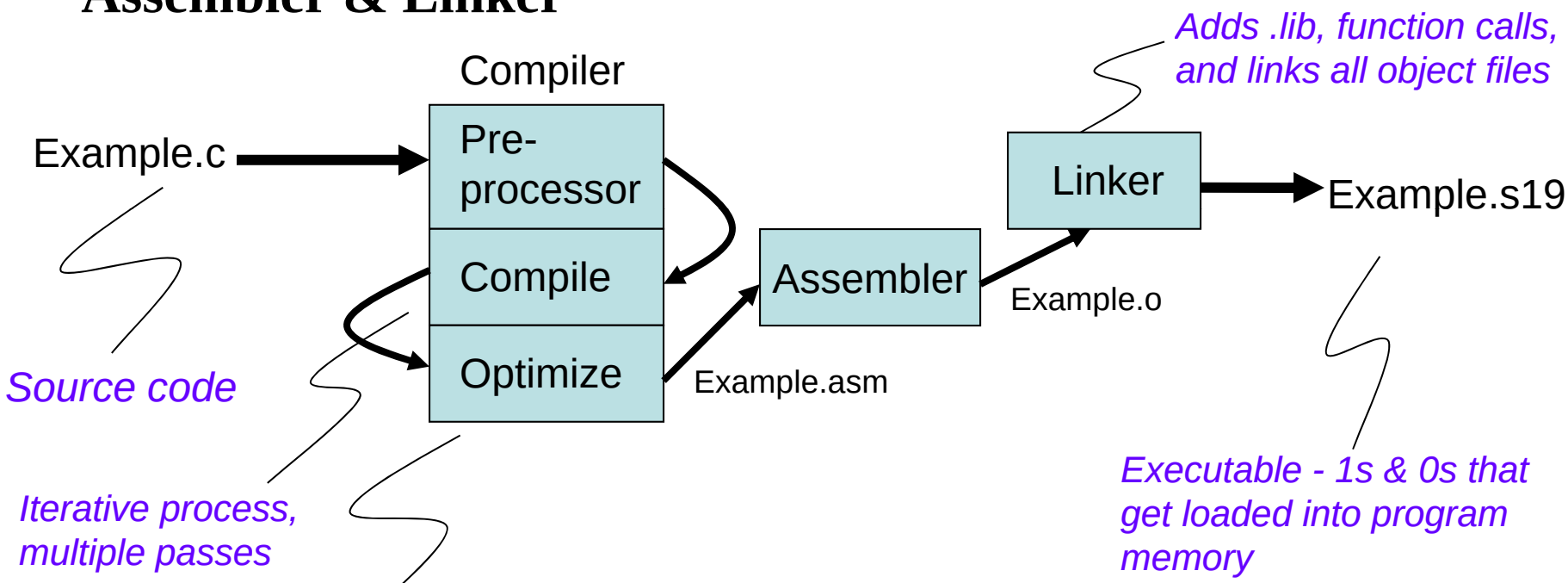
‘C’ Programming Language

Going from Assembly to Machine Code requires an Assembler



‘C’ Programming Language

Going from ‘C’ to Machine Code requires a Compiler, Assembler & Linker



‘C’ Programming Language

Basic Program Structure

#_____ are preprocessor directives

```
#include < stdio.h>

void main( )
{
   printf("\nHello World\n");
}
```

Every program needs 1 (and only 1) main function

printf( ) is a function defined in stdio.h

Function is in brackets

‘C’ Programming Language

Use Lots of Comments!!!

1. Traditional ‘C’ comments

```
/*    Everything between is a comment      */
```

2. C++ style comments

```
// Everything on this line is a comment
```

3. Preprocessor-enforced comments

```
#if (0)
        Everything between is a comment;
#endif
```

‘C’ Programming Language

Variable Names & Keywords

Variable Names - can be up to 31 characters long

- may use upper/lower case letters, digits 0-9, and ‘_’
- compiler & library vars use ‘_’ as first char in names

Reserved keywords – can’t be used for var names

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

‘C’ Programming Language

Data Types

char	8 bits	Integers types are signed by	
short	16 bits	default.	
long	32 bits		
long long	64 bits		
float	32 bits	$+/-10^{+/-38}$	~ 6.5 significant digits
double	64 bits	$+/-10^{+/-308}$	~ 15 significant digits
int	Usually depends on architecture		
	(32-bits for x86s 16 bits for HCS08)		

Signed #s use Two’s complement form

- **signed char** is 8 bits, range is -128 to +127
- **unsigned char** is 8 bits, range is 0 to +255

‘C’ Programming Language

Straight binary (unsigned)

MSB LSB

1	0	0	1	1	1	0	1

= 0x9D

$1 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times ^{21} + 1 \times 2^0 = 157_{10}$

Total range of possible values is $0_{10} \rightarrow 255_{10}$

To divide by two, shift one position to the left

MSB LSB

0	1	0	0	1	1	1	0

= 0x4E

LSB = 0 if even #
= 1 if odd #

$0 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 78_{10}$

‘C’ Programming Language

Useful # conversion chart

Table 1. Decimal, Binary, and Hexadecimal Equivalents

Base 10 Decimal	Base 2 Binary	Base 16 Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F
16	0001 0000	10
17	0001 0001	11
100	0110 0100	64
255	1111 1111	FF
1024	0100 0000 0000	400
65,535	1111 1111 1111 1111	FFFF

‘C’ Programming Language

ASCII text

American Standard Code for Information Interchange

- ASCII is often used in computer systems to represent characters
  - Hyperterm
  - Many LCD screens

Table 2. ASCII to Hexadecimal Conversion

Hex	ASCII	Hex	ASCII	Hex	ASCII	Hex	ASCII
$00	NUL	$20	SP space	$40	@	$60	` grave
$01	SOH	$21	!	$41	A	$61	a
$02	STX	$22	“	$42	B	$62	b
$03	ETX	$23	#	$43	C	$63	c
$04	EOT	$24	$	$44	D	$64	d
$05	ENQ	$25	%	$45	E	$65	e
$06	ACK	$26	&	$46	F	$66	f
$07	BEL beep	$27	‘ apost.	$47	G	$67	g
$08	BS back sp	$28	(	$48	H	$68	h
$09	HT tab	$29	)	$49	I	$69	i
$0A	LF linefeed	$2A	*	$4A	J	$6A	j
$0B	VT	$2B	+	$4B	K	$6B	k
$0C	FF	$2C	, comma	$4C	L	$6C	l
$0D	CR return	$2D	– dash	$4D	M	$6D	m
$0E	SO	$2E	. period	$4E	N	$6E	n
$0F	SI	$2F	/	$4F	O	$6F	o
$10	DLE	$30	0	$50	P	$70	p
$11	DC1	$31	1	$51	Q	$71	q
$12	DC2	$32	2	$52	R	$72	r
$13	DC3	$33	3	$53	S	$73	s
$14	DC4	$34	4	$54	T	$74	t
$15	NAK	$35	5	$55	U	$75	u
$16	SYN	$36	6	$56	V	$76	v
$17	ETB	$37	7	$57	W	$77	w
$18	CAN	$38	8	$58	X	$78	x
$19	EM	$39	9	$59	Y	$79	y
$1A	SUB	$3A	:	$5A	Z	$7A	z
$1B	ESCAPE	$3B	;	$5B	[	$7B	{
$1C	FS	$3C	<	$5C	\	$7C	|
$1D	GS	$3D	=	$5D	]	$7D	}
$1E	RS	$3E	>	$5E	^	$7E	~
$1F	US	$3F	?	$5F	_ under	$7F	DEL delete

‘C’ Programming Language

Math Operators

+	Addition	
-	Subtraction	
*	Multiplication	*Note: ‘*’ and ‘/’ have higher*
/	Division	*precedence than ‘+’ and ‘-’*
%	Modulus operator	

if Ans, Rem and the numbers 5 and 8 are integers, then

```
Ans = 5/8;      // result is 0
Rem = 5%8;      // result is 5
```

‘C’ Programming Language

Convert a number to ASCII

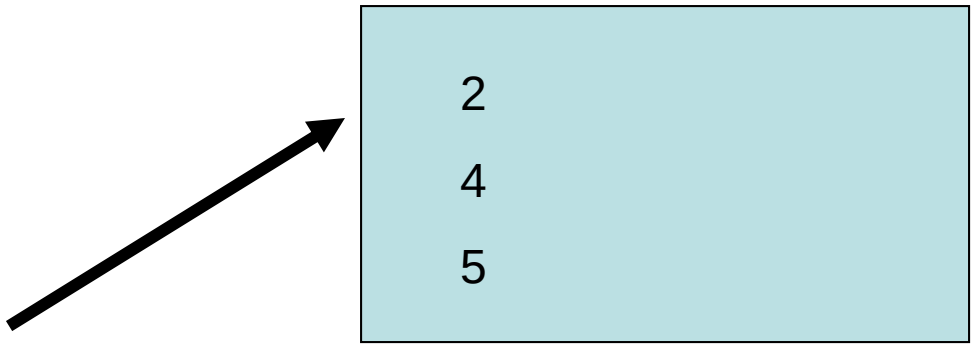
```
unsigned char  value;
            …

value = 0xF5;       // 245 base 10

temp1 = (value%10)+0x30;
temp2 = value/10;
temp3 = temp2/10+0x30;
temp2 = temp2%10+0x30;

printf(temp3”\n"); // MSB
printf(temp2”\n");
printf(temp1”\n"); // LSB
```

On hyperterm window you’d see …



‘C’ Programming Language

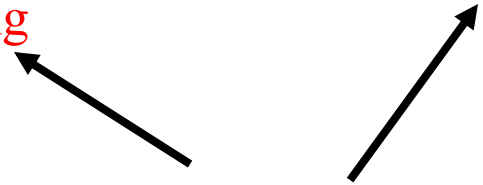
Increment & Decrement

i = i + 1;	is equivalent to …	i++;
k = k - 1;	“ “	k--;

C allows pre- and post-incrementing and pre- and post-decrementing

```
num = 1;
while (num++ < 3)
    {
        // do something
    };
```

```
num = 0;
while (++num < 3)
        {
            // do something
        }
```



Are these code snippets equivalent?

‘C’ Programming Language

Shift

x = 8;

x = x >> 2;

$0000\ 1000\ (8_{10}) \rightarrow 0000\ 0010\ (2_{10})$

Equivalent to dividing by 2^2

y = 8;

y = y << 3;

$0000\ 1000\ (8_{10}) \rightarrow 0100\ 0000\ (64_{10})$

Equivalent to multiplying by 2^3

May take less clocks than executing a multiply or divide instruction

‘C’ Programming Language

Logical Operators

<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	is equal to
!=	is not equal to
&&	AND
||	OR

NOTE:

if (0); // Is always false

if (1); // Is always true

Logical operators are binary operators
The statement

if (A >= B) …

Returns 0 if the statement is false and 1 if true

‘C’ Programming Language

Bitwise Operators

&	Bitwise AND
|	Bitwise OR
~	Bitwise NOT
^	Bitwise Exclusive OR

```
#define  MASK   (%1111 0000)
A = 0x88 & MASK;          // result is 0x80
B = 0x88 | MASK;          // result is 0xF8
C = 0x88 ^ MASK;          // result is 0x78
C = ~C;                   // result is 0x87
```

Note:

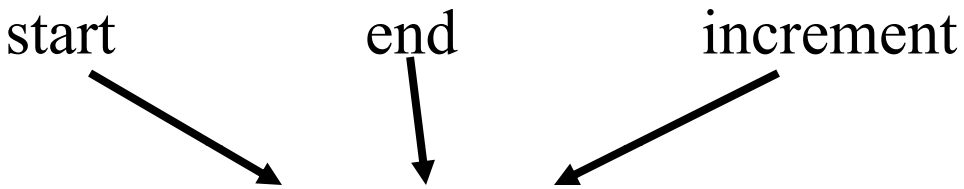
B &= MASK;

is equivalent to

B = B & MASK;

‘C’ Programming Language

Loops – for loop




```
for (i=0; i<10; i++)
        power[i] = volts[i] * current[i];
```

```
for(;;)
{
   //loop forever
}
```

‘C’ Programming Language

Loops – while loop

```
cntr = 0;
while (cntr < 10)          // loop will execute 10 times
{
        num[cntr] = 5;
        cntr++;
}

while (1)
{
        // this loop will execute forever
}
```

‘C’ Programming Language

Loops – do while loop

```
cntr = 0;
do
{
        num[cntr] = 5;
        cntr++;
} while (cntr < 10);          // still executes 10 times
```

‘C’ Programming Language

If statement

```
if (num <= 10  || eli==7)
{
        // do something
}
else if (num >= 20)
{
        // do something
}
else
{
        // default case
}
```

‘else if’ and ‘else’ never get tested if “if (num<=10) is TRUE

Can have only one ‘if’ and one ‘else’, but as many ‘else if’s as you want

‘C’ Programming Language

The switch statement

```
switch (buffer[3])
{
case 1:
        // execute function 1
        break;
case 2:
        //function eli
case 3:
case 5:
        // execute function 2
        break;
…
case n:
        // execute function n
        break;
default:
        // execute function _ERROR
        break;
}
```

If you were to look at the assembly or machine code, switch and if-else statements are functionally equivalent. But if there are many cases, a switch statement is usually easier to look at, add to, etc.

Switch statements lend themselves well to things like command parsers and state machines.

‘C’ Programming Language

Functions

type function_name(type, type, type, …)

Return argument – can be char, int, etc.

‘void’ means no return argument

If not ‘void’ function needs a ‘return (value)’ statement at the end

Values passed to a function – one way copy to function

‘void’ means no values passed

FUNCTION PROTOTYPE

- At top of ‘C’ file or included in header file

‘C’ Programming Language

```
// Includes
#include <Timer.h>

// Function PROTOTYPES
void config_IO (void);

// MAIN Routine
void main (void)
{
      // Configure Port I/O
      config_IO ();
      // Initialize Timer 3
      config_Timer();
      }
}

// Other Routines
void config_IO (void)
{
      //Set up micro I/O ports
}
```

Functions (cont.)

```
// Function PROTOTYPES
void config_Timer (void);
```

Timer.h

Main.c

‘C’ Programming Language

Note on Recursion / Reentrancy

n! = n * (n-1) * (n-2) * …

Function calculates a factorial by calling itself until n = 0.

```
long factorial (int n)
{
    if (n == 0)
        return (1);
    else
        return (n * factorial (n-1));
}
```

Need to be careful doing this, every function call puts multiple bytes on the stack. If not terminated correctly could overflow the stack very easily.

‘C’ Programming Language

Why use functions?

- Makes code more modular – easier to read
- If sections of code are repeated multiple times, putting that code in a function saves code space
- If section of code is not repeated more than once, function call adds extra code (and hence runtime)

- *What if you want the modularity but not the extra stuff, what do you do?*

‘C’ Programming Language

Macros

A way to “modularize” code without the penalty of a function call

In ‘file_name.h’ …

```
#define  square (x)      (x) * (x)
```

In ‘file_name.c’ …

```
Power = square (I) * R;
```

If you look at the compiled code, the macro definition gets inserted as in-line code, whereas functions get treated as jumps to a single block of code somewhere else in memory.

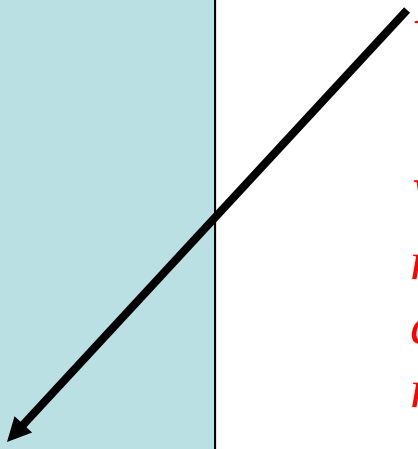
‘C’ Programming Language

Local Variables vs. Global Variables

```
// Function PROTOTYPES
void calc_number (void);
static unsigned char this_is_global;

// Main Routine
void main (void)
{
    unsigned char this_is_local;
    this_is_global = 10;
    this_is_local = this_is_global;
    calc_number ( );
}

// Other Routines
void calc_number (void)
{
    unsigned temp1, temp2;
    temp1 = this_is_global;
    temp2 = this_is_local;
}
```



This won’t compile - error.

Why? Global var’s get dedicated memory locations, local variables all share a section of ‘scratchpad’ memory – the compiler figures out exactly which variable gets which memory location at any one time.

‘C’ Programming Language

How to share Global Variables among multiple ‘C’ files

```
// Main.c
#include <main.h>
unsigned char this_is_global = 7;

// Main Routine
void main (void)
{
    unsigned char this_is_local;
    this_is_local = this_is_global;
    calc_number ( );
    run_algorithm ( );
}

// Other Routines
void calc_number (void)
{
    unsigned temp1;
    temp1 = this_is_global;
}
```

```
// main.h
extern unsigned char this_is_global;
extern void calc_number ( );
```

```
// Algorithm.c
#include <main.h>

// Routine
void run_algorithm (void)
{
    unsigned char this_is_local_too;
    this_is_local_too = this_is_global;
    calc_number ( );
}
```

Variables and functions can be external / global.

‘C’ Programming Language

Arrays

```
unsigned char cnum[5] = {5, 7, 2, 8, 17};      // 5 bytes of memory
unsigned int inum[5];                          // 10 bytes
float fnum[5];                                 // 20 bytes
```

Mem Location	Value
0100	5
0101	7
0102	2
0103	8
0104	17

Address of cnum[0]

‘C’ Programming Language

Multidimensional Arrays

```
unsigned char cnum[2][3];
cnum[0][0] = 3;
cnum[0][1] = 6;
cnum[0][2] = 8;
cnum[1][0] = 1;
cnum[1][1] = 0;
cnum[1][2] = 12;
```

Mem Location	Value
0100	3
0101	6
0102	8
0103	1
0104	0
0105	12

‘C’ Programming Language

Pointers

* - deference operator

& - address of

```
unsigned int *p, *q;
unsigned int i;
unsigned int **r

i = 5;
p = &i;
r = &p;
```

Memory Location	Value	
F004	F00C	p
F008		q
F00C	5	i
F010	F004	r
F014		

```
*p = 0;    // like saying “set the contents of
           // memory location pointed to by p
           // to 0” (i.e. i = 0)
**r = ?
```

‘C’ Programming Language

Pointers – another example

You are working on a 16 bit machine, and the memory location at absolute address 0x67A9 needs to be set to an initialization value of 0xAA55. How do you do it?

```
int *ptr;
ptr = (int *) 0x67A9; //Type Cast!
*ptr = 0xAA55;
PORTA_DATA_REG = 1;
```

```
#define PORTA_DATA_REG   *(unsigned char *)(0x0004)
```

‘C’ Programming Language

Advantage of Using Pointers

- Allows you to directly access machine memory
  - i.e. contents of specific registers
- Helps to modularize code
  - can pass a pointer in a function call

function call

```
BufCopy (num, &outputBuf, &inputBuf);
```

```
void BufCopy (char nbytes, char *DstBufPtr, char *SrcBufPtr)
    {
    while (nbytes-- > 0)
        {
            *DstBufPtr = *SrcBufPtr;
            DstBufPtr++; SrcBufPtr++;
        }
    }
```

function

‘C’ Programming Language

typedef, struct and union

```
struct FOURBYTES
{
    char byte4;
    char byte3;
    char byte2;
    char byte1;
};

typedef union FLOAT
{
    float f;
    struct FOURBYTES b;
};
```

```
union FLOAT Impedance [8],
unsigned char I_bytes [8][4];

Impedance [6].f = 23.556;

I_bytes [6][0] = Impedance [6].b.byte1;
I_bytes [6][1] = Impedance [6].b.byte2;
I_bytes [6][2] = Impedance [6].b.byte3;
I_bytes [6][3] = Impedance [6].b.byte4;
```

‘C’ Programming Language

typedef, struct and union

```
typedef struct
{
    unsigned char BIT_0 : 1;
    unsigned char BIT_1 : 1;
    unsigned char BIT_2 : 1;
    unsigned char BIT_3 : 1;
    unsigned char BIT_4 : 1;
    unsigned char BIT_5 : 1;
    unsigned char BIT_6 : 1;
    unsigned char BIT_7 : 1;
} BITFLAG;

#define  TRUE     (1)
#define  FALSE    (0)
```

```
// Definition
BITFLAG UserFlags;

// In code
UserFlags.BIT_0 = TRUE;
UserFlags.BIT_1 = FALSE;
```

‘C’ Programming Language

Preprocessor Directives

#include

#define

#pragma

Conditional compilation

#if / #ifdef / #ifndef

#elif

#else

#endif

```
#define GREEN
#define RED
            …
#if (GREEN)
//compile this code
            …
#elif (RED)
// compile this code
            …
#endif
```

‘C’ Programming Language

Other keywords – static

1. Local variables declared static maintain their value between function invocations
2. Functions declared static can only be called by functions in the same module

```
void process_buttons (void)
{
  static unsigned char button_old = 0;

  button_new = PORTA & %0000 0001;
  if (button_new & button_old)
  {
    // button press TRUE 2 times
    // in a row do something!
  }
  button_old = button_new;
}
```

‘C’ Programming Language

Other keywords – volatile

- When a variable is declared volatile, the compiler is forced to reload that variable every time it is used in the program. Reserved for variables that change frequently.
  - Hardware Registers
  - Variables used in interrupt service routines (ISR)
  - Variables shared by multiple tasks in multi-threaded apps

Ex.

```
volatile unsigned char UART_Buffer [48];
// UART_Buffer is used in the UART ISR to
// record the incoming data stream
```

‘C’ Programming Language

Other keywords – const

- Doesn’t mean ‘constant’, means ‘read-only’
- The program may not attempt to write a value to a const variable
- Generates tighter code, compiler can take advantage of some additional optimizations