Embedded Systems Tutorial 4
Intro to the C programming language

Marcus Pham & Stephen Whitely

Anatomy of a Function

In Assembly

- **we have subroutines**
- **arguments are place on the stack or data registers**
- **preservation of data was not guaranteed (eg D0, A0)**
- **results returned in D0**
- **even simple math can be tedious**
- **simple data structures can be difficult to use and are broken easily**

In C

- **we have functions**
- **arguments are passed as part of the call to the function, all in one line of code**
- **preservation of data can generally be considered safe***
- **results returned can be assigned to any variable (of the same datatype)**
- **many simple math operations can be achieved in a single line of code**
- **simple data structures are simple to use but can still be broken easily**

*** with exception of using global variables and pointers**

Anatomy of a Function

```
datatype function(datatype arg1, datatype arg2)
{     ...
      return result;
}
```

datatype – what sort of datatype the function will return or needs as arguments

function – name of the function

arguments – input values to be used by the function

{ ... } – the statements making up the function, ending with the return statement

eg

```
int sum(int x,int y)
{     int z;
      z = x + y;
      return z;
}
```

Syntax differences

Assembly

| uses 'vertical bar' or 'pipe' symbol for
| comments – no need to close comments
| but new lines need to be marked

```
.include "eyebot.i"
```

arguments placed in stack

```
        jsr function
```

restore stack pointer

```
                …
function:       …
            rts
```

result stored in D0

instructions end in whitespace/new line

C

/* comments are put between forward slashes and asterisks – need to be closed but can span multiple lines */ (or // comment)

```
#include "eyebot.h"

x = function([args]);
…
int function([args])
{       …
        return result;
}
```

result stored in x

instructions end in a semicolon ;

Datatypes

- In assembly we saw .b, .w, .l for byte, word and long as well as a special type .asciz for strings.
  - Integer operators could be interchanged if data permitted
- In C, the main datatypes are int, float, double and char.
  - int is for integers
  - float and double are for floating point (decimal fraction) numbers, double is double precision (double storage space)*
  - char is for characters, an array of characters makes a string
- Another datatype is void
  - Used to create functions which return nothing or have no arguments

*In a memory restricted environment, such as many embedded systems, float and double should be avoided if not necessary.

Brackets, Braces and Parenthesis, oh My!

- [ ] square brackets are used to enclose array references – "table[element number]", and when declaring arrays – eg. "int table[number of elements];"
- { } braces, or curly brackets are used to enclose statements inside functions and structures
- ( ) parenthesis are used to enclose and group arguments to a function – eg. "function(arg1, (arg2+arg3)xarg4);"

Arrays

- Arrays are an indexed group of elements of a certain type used to store data.
- Declare by

```
datatype name[number of elements];
```

e.g. -

```
int table[10];
```

- Reference by

```
name[element number]
```

e.g. -

```
x = table[0]; /* stores value in table element 0 to x */
table[9] = y; /* stores value of y to the element 9 */
```

Note – An array initialised "datatype array[n];" will have elements from "array[0]" to "array[n-1]"

Arrays

- Arrays can be filled a number of ways.
  - directly at declaration e.g.

```
int table[5] = {1,2,3,4,5};
```

  - inside a loop using control structures such as for loops (see upcoming example)
- Arrays are not automatically filled with zeros. There may be random data left in them from previous memory storage.
- A special array operation is available when declaring an array, which will fill the array with zeroes or NULL.

```
int table[size] = {0};
```

  - This special operation only works when declaring the array.

Multi-dimensional Arrays

- Multi-dimensional arrays are essentially arrays of arrays
- Declare by

```
datatype name[rows][columns];
```

e.g. -

```
int table[4][5];
```

- This is extendable out to how ever many dimensions is required e.g. –

```
int table[1][2][3]...[n];
```

- Large arrays will use a lot of memory. Try to make arrays no larger than required.

Multi-dimensional Arrays

- Multi-dimensional arrays can be filled at declaration just like a one dimensional array

e.g. -

```
int table[4][3] = {{1, 2, 3}, {4, 5, 6},
                   {7, 8, 9}, {10, 11, 12}};
```

Represents this table

1	2	3
4	5	6
7	8	9
10	11	12

Simple Loop Example

- Create a data structure with ten elements and store the numbers 1 to 10, in order, in that data structure.

Simple Loop Example

In Assembly

```
.section .data
table:  ds.l 10

.section .text
.globl main

main:   lea table, A0
        move.l #1, D0
loop:   move.l D0, (A0)
        adda.l #4, A0
        addi.l #1, D0
        cmpi.l #10, D0
        bne loop
        rts
```

```
| declare data structure 'table,' ten
| long elements (40 bytes in total)


| copy table address to add. register
| put the number 1 in data register
| copy number (D0) into table (at A0)
| increment A0 to next table element
| increment D0 by 1
| compare D0 to 10
| branch if not equal to 'loop'
| otherwise end (return to system)
```

Simple Loop Example

In Assembly

```
.section .data
table:  ds.l 10

.section .text
.globl main

main:   lea table, A0
        move.l #1, D0
loop:   move.l D0, (A0)
        adda.l #4, A0
        addi.l #1, D0
        cmpi.l #10, D0
        bne loop
        rts
```

In C

```
int main()
{  int table[10];
   int i;
   for (i=1; i<=10; i++)
   {  table[i-1] = i;
   }
   return 0;
}
```

Simple Loop Example

In C

```
int main()
{   int table[10];
    int i;
    for (i=1; i<=10; i++)
    {   table[i-1] = i;
    }
    return 0;
}
```

Control Structures

- In assembly we used branching and labels
- In C we use if, else, do, while
- These control structures allow much simpler ways to create loops and iterations

Control Structures

```
int x = 0;
while (x<=10)
/* check that x is less than or = 10, if not skip to next */
{  printf(x);  /* print the value of x */
x++;    /* increment x by 1 (same as x = x + 1) */
}       /* loop back to while statement */

int x = 0;
do {    /* do while always runs at least once */
printf(x);      /* print the value of x */
x++;    /* increment x by 1 (same as x = x + 1) */
} while (x<10)
/* check that x is greater than zero, if so loop back */
```

The difference is simply when the while statement is evaluated.

Control Structures

```
int i;
for (i = 0, i <= 10, i++)
/* set initial and final value for i, and increment each
iteration */
{    printf(i);                    /* print the value of i */
}
```

The for loop works a lot like the while loop but the 'counter' is contained within the for statement. The increment occurs after the code inside the loop is run and the comparison operation occurs before the loop is run.

Control Structures

```
if (x==1)
{       ...
} else {
        ...
}
```

```
switch (x)
{       case 1:         x = f1(x);
                break;
        case 2:         x = f2(x);
                break;
        ...
        case n:         x = fn(x);
                break;
        default:        x = 0;
}
```

```
if (x==1)
{       x = f1(x);
} else if (x==2)
{       x = f2(x);
} …
```

Note that switch/case only works with exact integers, not floating point numbers and not ranges of values.

Complex Statements

- In assembly, each command only had two operands
  - The operands had to be constant numbers, registers or variables
- In C, each statement or function can contain a number of operands and the operands themselves can be statements or functions
  - e.g.

```
d = sum(a, sum(b, c));
```

- This means we can accomplish in one line of C code something that would take many lines of assembly.

Complex Statements

In Assembly

```
.asciz text1 "I only want to print "
.asciz text2 " things but I need "
.asciz text3 " lines of code./n"

main:     move.l #5, D1
          move.l #23, D2
          jsr printmsg
          rts

printmsg: pea text1
          jsr LCDPrintf
          add.l #4, SP
          move.l D1, -(SP)
          jsr LCDPutInt
          add.l #4, SP
          pea text2
          jsr LCDPrintf
          add.l #4, SP
          move.l D2, -(SP)
          jsr LCDPutInt
          add.l #4, SP
          pea text3
          jsr LCDPrintf
          add.l #4, SP
          rts
```

23 lines of code (10 more if you backup registers)

In C

```
int main()
{   int a = 5;
    int b = 8;
    printmsg(a,b);
    return 0;
}

void printmsg(a,b)
{   LCDPrintf("I only want to print %d things but
    I need %d lines of code./n",a,b);
    return void;
}
```

8 lines of code

Pointers

- Best thing about them is that you don't have to use them yet :P.
- They have there uses but for inexperienced programmers they are most likely only going to cause you to break things.
- Please don't even think about using them in the early labs.
- We will come back to pointers later in the unit.

Pointers

```
int x, y;          /* integer */
x=7; y=5;
x → 7 (value)
&x → $F0A0 (address)
int *a;            /* pointer */
a = &y;            /* address of y */
a → $F0A4 (address)
*a → 5 (dereferenced value)
```

Fun Facts about C

- C programming language, which was developed in 1972, was preceded by B (in 1969) but B was not preceeded by A.
- The name of the C++ programming is a play on the increment function in C.
- The name C# is a reference to the musical notation. For absolutely no good reason.
- Fun facts about programming are rarely ever actually fun.
- If you do find these facts fun, you are probably a big nerd but I still like you.