

Notes on Parameter Passage in C Functions

All parameter passage in C is *by value*. This means that the value of a parameter is passed to a function – not the parameter itself. If the function changed the passed value it does not change the original variable in the calling program.

Example 1

The following function has three parameters and it returns an int.

```
int x, y = 7, z = 2;
float r = 3.2;
x = Parameterlist(y, z, r);
```

```
int ParameterList(int a, int b, float r)
{int x = 5; // Local variable
 r = (float)a*b; //changing a, b, and r does not
 a = x + (int)r; // change any values in the main
 return a + b; // program
}
```

When the function is called the values of y, z, and r in the main program are copied into the new variables a, b, and r in the function. Since the values are copied to new variables in the function, changing those new variables does not change the original variables in the main program. The function, in this case returns an int and that is the only thing that comes back from the function.

In order to get more than one thing back from a function we must pass the address of a variable – not the value of the variable. Another name for *address* is reference so that when we pass an address of a variable we say we are passing the parameter by reference.

In C you can create a variable which holds the address of another variable. Such a variable is called a *pointer*. To create a pointer you do something like this:

```
int *pi; //pointer to an int
float *pf; //pointer to a float
```

Creating a pointer variable does not make it point to a variable. To do that you must create the variable and assign its address to the pointer. The ampersand (&) operator provides the address of a variable. In C we could create an int and a float and assign our pointers to point to them like this:

```
int i = 5;
float f = 3.14;
pi = &i;
pf = &f;
```

To pass a variable to a function by reference we need to pass its address and catch that address with a pointer. The following example illustrates how this is done.

Example 2

```
//Prototype
void ParameterPointer(int *x);
```

```
//function call
int y = 7;
ParameterPointer(&y);
```

```
void ParameterPointer(int *x)
{int a = 2;
 *x = 2*a; //changing x changes y in the main
 // program.
}
```

In this example `*x` is a pointer to the main program variable `y`. Changing `*x` in the function changes the value of `y` in the main program.

In some cases, it is necessary to pass an array to a function. In C an array name is already a pointer so it's not necessary to create a pointer to pass an array by reference. All arrays are already passed by reference as a default. Here is an example that illustrates this concept.

Example 3

```
//Prototype
void ParameterArray(int *x);

//Function call
int a[] = {1, 2, 3};
ParameterArray(a);

void ParameterArray(int *x)
{
    x[0] = 2;
    x[1] = 9;
}
```

In this case function `ParameterArray` has a pointer argument. Since an array name (in this case it's just `a`) is already a pointer so we just include the name as the argument. Changes to the array values in the function cause change to the array in the main program since they both have the same address.

Example 4

```
//Prototype
void ParameterArray(int a[]);

//Function call
int a[] = {1, 2, 3};
ParameterArray(a);

void ParameterArray(int x[])
{
    x[0] = 2;
    x[1] = 9;
}
```

This is the same as example 3 but instead of using an explicit pointer we pass the array name which is itself a pointer. If you are confused by pointers this may make a little more sense.

Example 5

```
//Prototype
void ParameterArray(int x[][3]);

//Function call
int a[] = {{1, 2, 3},
           {4, 5, 6}};
ParameterArray(a);

void ParameterArray(int x[][3])
{
    x[0][0] = 2;
    x[0][1] = 9;
    x[1][0] = 4;
}
```

This is a 2D array example. Note that the prototype and the function definition have `x[][3]`. The first item must be blank. If it is not, say as in `x[2][3]`, the compiler interprets this as passing the single integer row 2 column 3.

In general, C passes parameters to functions by way of the stack but this may differ from compiler to compiler. Since the 8051 has only a small amount of read/write memory and stack operation are slow, the Keil C-compiler uses a mix of registers and the stack to pass parameters. For a Keil C program the first three parameters are passed in registers and all parameters beyond the first three are pushed onto the stack. All parameters are passed as 16-bit numbers even though they may be declared as 8-bit quantities. The first parameter goes into registers R6-R7, the second in R4-R5, and the third in R2-R3. (R3, R5, and R7 are the least significant bytes.) Functions which return a value place the value in R6- R7.

Parameter Passage Registers ARM

Since the stack is a memory operation passing parameters in registers is faster. The Keil C-compiler for the ARM passes the first four parameters in registers R0 to R3. The returned value come back in R0. If you pass more than four parameters those after the fourth are passed on the stack.