CFunction Overview

Introduction

MMBasic has User Defined functions which enable the MMBasic programmer to write functions in MMBasic which then may be used just like the inbuilt functions of MMBasic itself. For example, MMBasic doesn't have Trim/LTrim/RTrim functions which are present in some other BASIC's - the Trim functions remove leading and trailing spaces. So we could write a set of 3 MMBasic functions to implement Trim, RTrim, and LTrim, eg

```
' trim any characters in c$ from the start and end of s$
FUNCTION Trim$(s$, c$)
  Trim$ = RTrim$(LTrim$(s$, c$), c$)
END FUNCTION

' trim any characters in c$ from the end of s$
FUNCTION RTrim$(s$, c$)
  RTrim$ = s$
  DO WHILE INSTR(c$, RIGHT$(RTrim$, 1))
    RTrim$ = MID$(RTrim$, 1, LEN(RTrim$) - 1)
  LOOP
END FUNCTION

' trim any characters in c$ from the start of s$
FUNCTION LTrim$(s$, c$)
  LTrim$ = s$
  DO WHILE INSTR(c$, LEFT$(LTrim$, 1))
    LTrim$ = MID$(LTrim$, 2)
  LOOP
END FUNCTION
```

and then use them in our MMBasic programs just as though they were real built-in MMBasic functions, eg

```
print Trim$(A$)
B$=LTrim$(A$)
```

MMBasic User-Defined functions are really useful and powerful, but can only execute as fast as the MMBasic Interpreter can interpret the MMBasic lines of code. Empirically, a line of MMBasic code takes about 50~60uSecs to execute on a PIC32 MX170 at CPU=48MHz. This speed is usually quite adequate, but there are circumstances when much higher performance is required, and/or instances where the amount of MMBasic code lines required to implement a required capability would result in an unacceptable/unusable level of performance.

Until the addition of CFunctions to MMBasic, these performance limitations meant that there would always be certain application areas which would be off-limits to MMBasic, and one had to go look elsewhere for a solution. Now when the performance barrier is hit, we can use CFunctions for the time critical pieces, and stay with MMBasic for the vast majority of the rest of the application code.

CFunctions are the next level of User-Defined function. CFunctions comprise fully compiled code or assembler and thus run at the full native speed of the processor without the overhead of the MMBasic interpreter. CFunctions typically execute at between 50~100 times faster than an equivalent MMBasic User-Defined function. The downside is that CFunctions have to be programmed in "C" (or MIPS assembler for the really hardened programmer !), which is a different and some would say more difficult environment than the comfort of BASIC.

In use, calling CFunctions is just like calling a MMBasic User-Defined function.

This document describes how to get going with CFunctions.

The 'C' Language

To create CFunctions one needs to be able to programme in the C language - yes that one with curly braces all over the place ! We're not going to discuss writing MIPS assembly language - life's just too short !

C is an enormously powerful language with any number of ways to get one's self into trouble. BUT, for CFunction creation purposes, one only needs to use a very small subset of the full capabilities of C. I hope that as you work your way through the tutorials in this document, that you will see that writing CFunctions is not that complicated at all. Also, there is an extensive library of CFunctions at http://www.g8jcf.dyndns.org/MMBasicIILib/ which is available for examination.

There are any number of places on the Internet where one can learn/brush-up on C programming, but since we're going to be focusing on embedded applications, it would seem best to go with one of the leading embedded MCU manufacturers, ie Microchip.

Microchip - "Fundamentals of the C Programming Language"

http://microchip.wikidot.com/tls2101:start.

Abstract

This class provides an introduction to the C programming language (as specified by the ANSI C89 standard) in the context of embedded systems. We cover the C language from the ground up from a non-hardware specific point of view in order to focus on the various elements of the C language itself. While not required, previous experience with any programming language or experience with microcontrollers would be helpful. The material is accompanied by a series of hands-on exercises designed to reinforce the fundamentals, all of which are conducted within MPLAB® X using the 16 bit C compiler and Simulator. Skills learned in this class are applicable to any ANSI C compiler. Hardware and compiler specific details such as interrupts, memory models and optimization are NOT be discussed. These topics are covered in the compiler specific classes.

K & R

Also of course, it's always worthwhile reading the seminal work on 'C' by Brian Kernighan & Dennis Ritchie, normally referred to as just "K &R". You can download a .PDF version of the second edition from http://www.iups.org/media/meeting_minutes/C.pdf

Abstract

This second edition describes C as defined by the ANSI standard. This book is meant to help the reader learn how to program in C. The book assumes some familiarity with basic programming concepts like variables, assignment statements, loops, and functions. A novice programmer should be able to read along and pick up the language.

FEATURES:
All examples have been tested, which is in machine-readable form.
It discusses various aspects of C in more detail, although the emphasis is on examples of complete programs, rather than isolated fragments.
It deals with basic data types, operators and expressions.
Covers functions and program structure, external variables, scope rules, multiple source files, and also touches on the pre-processor.
It also describes an interface between C programs and the UNIX operating system, concentrating on input/output, the file system, and storage allocation.
It also provides a language reference manual. The official statement of the syntax and semantics of C is the ANSI standard.

Pointers !!

So what is a pointer ?

A computer consists of three main elements

1) The CPU which does the actual crunching/processing of data
2) Memory which holds/stores the instructions and data values operated on by the CPU
3) Input/Output devices to interact with the external world

Memory and Input/Output devices are "addressed" by the CPU, ie the CPU sends out an address on the Memory/I/O address bus and the Memory/I/O returns the data value stored in the unique addressed location, the same applying when writing values out to Memory/I/O.

In the PIC32 MCU, the memory addresses are all 32 bit wide/long, and the memory locations are 32 bits wide.

So, a pointer is simply a variable which contains a number which is used as a memory address rather than just as a number to count/add/subtract things.

MMBasic has pointers !

If one writes in MMBasic

```
Dim AddrOfMyVar as Integer
Dim MyVar as Integer
AddrOfMyVar = Peek(VarAddr MyVar)
```

AddrOfMyVar will contain the address of the variable MyVar, ie AddrOfMyVar points at MyVar, ie AddOfMyVar is a Pointer.

Now append

```
MyVar=10
PRINT "Variable MyVar contains ";MyVar

PRINT "Variable MyVar is located at &H";HEX$(AddrOfMyVar)
PRINT "Variable MyVar contains using Peek";PEEK(WORD AddrOfMyVar)
PRINT

PRINT "Using AddOfMyVar as a pointer to change the value of MyVar to 25"

POKE WORD AddrOfMyVar, 25
PRINT "Variable MyVar contains ";MyVar
PRINT "Variable MyVar contains using Peek";PEEK(WORD AddrOfMyVar)
```

When you run this short program you should see

```
> RUN
Variable MyVar contains  10
Variable MyVar is located at &HA0001378
Variable MyVar contains using Peek 10

Using AddOfMyVar as a pointer to change the value of MyVar to 25
Variable MyVar contains  25
Variable MyVar contains using Peek 25
>
```

The difference between MMBasic 'pointers' and C pointers is that MMBasic 'pointers' point at anything, so if you add 1 to an MMBasic 'pointer', the value stored in the 'pointer' variable is incremented by 1. In C, if you add one to a pointer, then the actual value stored in the pointer variable **<u>will be adjusted depending on the type of the pointer</u>**, eg if the pointer is declared as pointing at 64 bit integers, then the value stored in the pointer variable itself will be adjusted by 8 because 64 bit integers take 8 bytes of memory to store them.

C Pointers

Many if not most, programming languages do NOT support pointers, and indeed many computer scientists/language creators "abhor" pointers and will do anything they can to persuade programmers that pointers are "evil" and unnecessary, and indeed, one can make a right mess of an environment if pointers are misused.

But for low level programming such as in writing an Operating System, manipulating hardware, writing low-level software primitives, for the highest performance without using assembler, then pointers are essential, and unavoidable. C has a very rich set of pointer mechanisms - as befits the language used to write almost every serious Operating System in widespread use, eg UNIX/LINUX/Windows, and of course the MMBasic Interpreter itself.

Luckily, unlike earlier Microchip 8/16 bit processors, the PIC32 MCUs have memory protection mechanisms built into the chip, so that if a pointer is pointed at something/somewhere inappropriate, a "bus exception" will be generated which the MMBasic interpreter can catch, and/or, the MCU is reset, so nothing can get damaged.

In the PIC32 MCU, memory addresses are all 32 bit wide/long, and the memory locations themselves are 32 bits wide.

A pointer is simply a variable which contains an address rather than a just a number. For example when one increments a pointer to a 32 bit integer by 1, the compiler actually adds 4 to the value stored in the pointer variable because the compiler knows that addresses are 32 bits long and each memory location is 32 bits/4 bytes wide, so when the programmer says point to the next memory location, the compiler knows that means 4 bytes further on from the current value.

In C, variables which will contain addresses rather than just a number, ie they "point" at somewhere, are declared with * in front of the variable name, eg *a, *b, *varname.

In C, variables which point at somewhere also have a type which denotes the type of thing they will be pointing at, eg

```
int *varname;             //varname is a pointer to 32 bit variables
char  *varname;           //varname is a pointer to character or byte variables
long long *varname;       //varname is a pointer to 64 bit variables
double * varname;         //varname is a pointer to Double precision floating point
variables
float  * varname;         //varname is a pointer to Single precision floating point
variables
```

One can also have pointers to functions, pointers to pointers and much more, but for CFunction purposes, understanding and using pointers to data variables is generally sufficient.

The MMBasic to CFunction interface

When MMBasic calls a CFunction and passes arguments through to the CFunctions, what is actually passed to the CFunction is the memory address of each argument - in BASIC parlance, By Reference.

For example, say we have a CFunction which will add two 64 bit integers together and return the result back to MMBasic. In MMBasic we could write;

```
Dim A%=10
Dim B%=20
Dim C%=0
C%=Add(A%,B%)
```

The CFunction itself could look like;

```
long long Add(long long *a,  long long *b){
  long long c;             //variable c will hold the result of the add
  c=*a + *b;               //add a+b result in c
  return c;                //return the result of the add back to MMBasic
}                          //End of the Add function
```

Now let's examine the 5 lines of C code which make up function Add.

*Line 1: long long Add(long long *a, long long *b){*

Line 1: says that the function

1) Will return a 64 bit number, ie long long
2) is named Add
3) expects to be passed a pointer, ie *a, to a 64 bit variable, ie long long
4) expects to be passed a pointer, ie *b, to a 64 bit variable, ie long long

Line 2: long long c;

Line 2: says

1) we want a variable named c, which will contain 64 bit quantities

*Line 3: c=*a + *b;*

Line 3: says

1) get the value from the address pointed to by pointer variable a, ie ***a**
2) get the value from the address pointed to by pointer variable b, ie ***b**
3) add the two values from 1) and 2) together, ie **+**
4) store the result of the add into variable c, ie **c=**

Line 4: return c;

Line 4: says

1) Get the value stored in variable c, and pass that value back to the caller, ie MMBasic

Line 5: }

Line 5: says

1) End of Function, ie }, equivalent to "End Function" in MMBasic

Line 3: shows how the C code accesses the data values passed in from MMBasic, ie *a and *b. In BASIC parlance, the arguments to function Add are passed " **ByReference**" .

When the variables passed from MMBasic are going to be used more than once inside the CFunction, a useful technique is to copy the value passed into local variables and then perform all subsequent operations on those local variables, ie minimise all those asterisks. So we could rewrite the little test CFunction above as

```
1:  long long Add(long long *a,  long long *b){
2:    long long Mya=*a;       //declare variable Mya and load it up what *a points at
3:    long long Myb=*b;       //declare variable Myb and load it up what *b points at
4:    long long c;            //variable c will hold the result of the add
5:    c=Mya + Myb;            //add a+b result in c
6:    return c;               //return the result of the add back to MMBasic
7:  }                         //End of the Add function
```

So the recipe for writing a CFunction is as follows;

1) The function MUST return a 64 integer result, ie long long, even if that result is not used
2) The arguments to the CFunction MUST be declared as pointers, ie prefixed by * (asterisk)

    a. There is one exception to the * rule ! If the variable being passed is an array, eg A$, then the arguments to the function in the C code may be alternatively written as vartype varname[], where vartype is the type of the array, eg char for byte/char, long long for 64 bit integer, and varname is the symbolic name of the argument, eg MyPort, a, b, c etc. So for example char A[] declares that the CFunction expects to receive an array called A which contains byte/single character values

3) Whenever in the function, the values of the arguments needs to be accessed, prefix the variable name with *, ie get the value from the memory location given (pointed at) by the variable

4) MMBasic strings are equivalent to char arrays in C, where the first byte of the char array contains the length of the MMBasic string.

Further Reading

My explanation of pointers above is focused entirely on getting you started with CFunctions, for more general treatment of pointers in C, take a look at

http://boredzo.org/pointers/

http://www.tutorialspoint.com/cprogramming/c_pointers.htm

and of course read K & R !!

At the end of the day, the only way to get the hang of CFunctions is to work through the Tutorials document, and modify the tutorial code to get a feel/understanding of what is going on.

I can always be contacted at support@cfuncgen.dyndns.org and am happy to try and help you create/use CFunction.

Data Types

In this document the various data types referred to are

Single	MMBasic Floating Point Variable occupying 4 bytes in memory. Without considerable effort, Single variables are of little use in writing MMBasic CFunctions - we will not be using MMBasic Single variables in CFunctions.
String	MMBasic String occupying 1 byte per character, plus one byte for the string length. In 'C', MMBasic strings are Unsigned Character arrays starting at index 0. String[0] is the number of characters in the MMBasic string, ie the value returned by LEN(String$)
INT64	MMBasic 64 bit Signed Integer, often denoted by the suffix %, eg A%, occupying 8 bytes in memory. In 'C', INT64 refers to the 'long long' data type
INT32	In 'C', INT32 refers to 'int' or 'long' which is a 32 bit signed integer. INT32 values are automatically expanded to INT64 size when returned to MMBasic. INT32 (and UINT32) is the "natural size" for the PIC32 and leads to the smallest and fastest code. MMBasic does not directly support INT32 variables.