FreeBasic Bare Bones

From OSDev Wiki

While the forum has several flamewars about BASIC, it is a turing-complete language. FreeBasic adds functionality that makes it suitable for OS development. This tutorial provides a working kernel in Basic, together with the pitfalls associated with it.

Difficulty level

Medium

Contents

About FreeBasic and the Runtime

FreeBasic is a BASIC compiler with syntax compatible to QuickBasic, with several improvements that make it a viable compiler for Operating System Development. However, normal Basic programs rely heavily on part of the language that is called the Runtime. The Runtime consists of all operations that cannot be performed by a set of basic processor instructions. When engaging in operating systems development, you must be aware that you can not use anything that is part of the Runtime. That includes:

- String operations. You can define strings, but not use them in any regular way.
- Dynamic-sized arrays. Only arrays with a fixed size can be used, and even then some operations won't work on them.
- Any input or output statements. Normally the program communicates with the OS about these things, but since we are writing the OS, these things can't be used.

When you get messages about undefined symbols starting with fb_, it means you are using parts of the Runtime.

To compensate, FreeBasic provides pointers that can be used to perform functionality the runtime would provide. Inline assembly can be used as well.

Pointers for Beginners

Pointers can be difficult to understand. Many modern languages do not work with pointers, while if you know a language like C, C++ or Assembly, you will probably know what they do. Since you will need to use them, a quick introduction in how they work in FreeBasic.

The computer uses a set of registers, and memory. Memory is divided into bytes, each byte has its own location number. Big numbers, strings, types and arrays use up multiple bytes. These bytes are stored next to each other. A Long will hold 4 bytes, and will for example occupy locations 239394, 239395, 239396 and 239397. In the computer, these location numbers are called addresses (like the address when sending letters)

Pointers hold these location numbers. For bytes, they hold the exact address, for larger objects, it will hold the address of the lowest address (you can determine the other addresses as they will immediately follow the first)

In FreeBasic, a pointer is defined by adding the Ptr keyword.

```
Dim mypointer As Byte Ptr
```

This one holds the location of a byte of memory.

```
Dim mypointer As Long Ptr
```

This one points to the first of 4 bytes of memory, which together form the number.

You can not use pointers straight away. Consider sending a letter with no address on it will not do any good. So we must first put an address in there. That leads to the question: how do we get an address. Some things have fixed addresses, like the video card. You can also ask variables for their addresses.

```
Dim variable as Byte
Dim pointer as Byte Ptr
pointer = @variable
```

The @ returns the address of the variable that follows it.

```
Dim pointer as Byte Ptr
Dim address as Long
address = 12345
pointer = CPtr(Byte Ptr, address)
```

Here CPtr (Convert to Pointer) is used to create pointers. You give it a type (Byte Ptr) and the address. You can also change pointers:

```
Dim pointer1 as Byte Ptr
Dim pointer2 as Long Ptr
pointer2 = CPtr(Long Ptr, pointer1)
```

Be careful when you do so: a byte occupies one location in memory, a long occupies four. if we would use this pointer, we would use three locations in memory of which we know nothing about. Sometimes, they are necessary, like when we want to work with strings without having the Runtime.

You can access pointers with an index:

```
Dim value as Long
Dim pointer as Long Pointer
pointer = @value
' value = pointer[0]
```

when we want to know what is behind the pointer, we ask for the memory at its location with the [ ], and then we add a number which tells us how many locations further to look. When we use [0] it simply means that we do not want to do anything with the address stored in the pointer. The result is the value stored in the memory locations pointed to by the pointer. For byte pointers, this will look at the address stored and return the result. For other pointers, it will look at a series of locations and return what those values represent.

Strings are useful, but tedious. A string is stored as a series of bytes in consecutive addresses. You can build a pointer to a string as well:

```
Dim s as String
Dim pointer as String Ptr
pointer = @s
```

But since we do not have the Runtime, we have to use something that does not use it.

```
Const s = "Text"
Dim pointer as Byte Ptr
pointer = CPtr(Byte Ptr, @s)
' pointer[0] = Asc("T")
' pointer[1] = Asc("e")
' pointer[2] = Asc("x")
' pointer[3] = Asc("t")
' pointer[4] = 0
```

The first example uses the Runtime which will not compile. The second example shows the only working method of using strings: Define one using Const, then ask for a pointer to that. Since a string is a sequence of characters (bytes), we change the type of the pointer in the process. Next we can ask for individual characters as if it were an array. However, they now appear as numbers. The ASCII codes to be precise, and the same as when you use Chr$() and Asc().

Basic uses an borrowed trick to tell us the end of the string. After the last character, there will always be a 0. So if we read a string in order, we can tell when it has ended.

```
Sub PrintString(src As Byte Ptr, x As Long, y As Long)

    Dim dst as Byte Ptr
    Dim counter as Long

    dst = CPtr(Byte Ptr, &HB8000 + y * 160 + x * 2)
```

```
    counter = 0

    while src[counter] <> 0
        dst[2 * counter] = src[counter]
        dst[2 * counter + 1] = 15
        counter = counter + 1
    wend
End Sub
```

To conclude, this function prints a string (a converted Byte Ptr string). It creates a pointer that is aimed at the video card (it occupies among others a range starting from address B8000 hex), and we pick a location in there. Next we take a character from the string, check if its 0, and if it isn't, copy it to the video card and go to the next character. Due to the way the video card works, we add a color (15) as well.

Tuning FreeBasic for OS Development

A normal install of FreeBasic is in most cases configured to build for the system it was installed on. That means that it will try to compile programs for linux or windows OSes, rather than your own.

To fix this, you should build at the very least a crosscompiling binutils (2.17 or later recommended). While you're at it, you can also build GCC, which comes in handy once you start porting software written in C (that includes the runtime).

FreeBasic stores its auxiliary binaries in the bin directory, or a subdirectory thereof. To get started quickly, replace `ld`, `ar` and `as` with the versions built in the previous step. Note that this stops FreeBasic from working on its previous host. Under windows, the bin directory is subdivided into platforms, and you can add a new one rather than modifying the existing one. To use this, create the 'linux' directory, copy `i586-elf-ld`, `i586-elf-ar` and `i586-elf-as`, and rename them to `ld`, `ar` and `as`. When running freebasic you can use the `-t linux` command line switch to compile for your OS instead of windows, allowing you to continue using Freebasic normally for non-os development. The Linux version does not come with this target switch - here you'll have to replace the bundled binaries with your own versions.

The code

Here is the entire project for reference

kernel.bas

```
DECLARE SUB PrintString(src AS Byte Ptr, x AS LONG, y AS LONG)
DECLARE SUB main ()

SUB multiboot ()
   Asm

     'setting up the Multiboot header - see GRUB docs for detail

     .set ALIGN,    1<<0
```

```
      .set MEMINFO,  1<<1
      .set FLAGS,    ALIGN | MEMINFO
      .set MAGIC,    0x1BADB002
      .set CHECKSUM, -(MAGIC + FLAGS)

      .align 4
      .LONG MAGIC
      .LONG FLAGS
      .LONG CHECKSUM

      .set STACKSIZE, 0x4000
      .comm stack, STACKSIZE, 32

      .global loader

      loader:
        lea   esp, stack + STACKSIZE
        push  eax
        push  ebx

        CALL  MAIN

        cli
        hlt
    END Asm

END SUB

SUB main ()
    CONST s = "Hello World"

    PrintString CPtr(Byte Ptr, @s), 35, 12

END SUB

SUB PrintString(src AS Byte Ptr, x AS LONG, y AS LONG)

    DIM dst AS Byte Ptr
    DIM counter AS LONG

    dst = CPtr(Byte Ptr, &HB8000 + y * 160 + x * 2)

    counter = 0

    WHILE src[counter] <> 0
        dst[2 * counter] = src[counter]
        dst[2 * counter + 1] = 15
        counter = counter + 1
    WEND
END SUB
```

link.ld

```
OUTPUT_FORMAT("elf32-i386")
ENTRY (loader)

SECTIONS{
    . = 0x00100000;

    .text :{
        KERNEL_START = .;

        *(.text)
    }

    .rodata ALIGN (0x1000) : {
        *(.rodata)
        _CTORS = .;
        *(.ctors)
        _CTORS_END = .;
    }

    .data ALIGN (0x1000) : {
        *(.data)
    }

    .bss : {
        SBSS = .;
        *(COMMON)
        *(.bss)
        EBSS = .;

        KERNEL_END = .;
    }
}
```

Build instructions

```
fbc -c kernel.bas -o kernel.o
i586-elf-ld -T link.ld -o kernel.bin kernel.o
```

`kernel.bin` can then be loaded by GRUB

Getting the Runtime to work

Once you progress with your kernel, you can try to get runtime functionality to work.

The Runtime is built on top of the C library. FreeBasic provides sources to the Runtime (which are written in C). You'll need to provide the C library. Since many parts of the kernel rely heavily upon each other, you will probably add the Runtime in steps. Some functions compile cleanly with minimal effort. When you compile the CType, String and Stdlib (excluding malloc, calloc, and free) parts of the runtime can be compiled on top of those. Before you can use the string and array instructions, you will need to have memory management implemented. That will allow you to compile malloc, calloc, free from the C library, after which the string and some of the array functions can be compiled.

Categories: Level 2 Tutorials | Bare bones tutorials