Technical Note PT37
Using MPW for Non-Macintosh 68000 Systems

CONTENTS

This Technical Note discusses using MPW 3.0 for creating software intended to run on 68000-based systems that do not implement the Macintosh run-time architecture. These systems include NuBus(TM) cards, peripheral devices, and proprietary 68000 systems.

[Jun 01 1989]

Introduction

Occasionally there is a need to create routines or programs for non-Macintosh systems. Such situations can occur if you are writing a driver for a NuBus board, developing a peripheral that uses a 68xxx microprocessor, or perhaps targeting a proprietary 68xxx machine (Apple uses MPW for all of its ROM and NuBus development.)

For tasks such as this, MPW 3.0 can provide the solution. This Note discusses the problems and issues that arise when doing using MPW 3.0 for this type of development, and it gives some hints and solutions.

To aid you in your efforts, there are several tools available on AppleLink in the Developer Services bulletin board (Developer Techical Support:Macintosh:Tools:Card Dev Tools:) and on *Phil & Dave's Excellent CD* . These tools include utilities to generate checksum data and to prepare your program for downloading.

The following is a brief summary of problem areas you may encounter:

- A5-Relative Globals Segmenting and the Jump Table ToolBox and OS Routines Setting up Your Run-Time Environment

Back to top

A5-Relative Globals

The Problem

In traditional machine environments, the compiler allocates a certain range of memory in which to store global variables. This memory is established by the machine's memory architecture, and it can usually be referenced by using absolute addressing modes.

Because the Macintosh has a very dynamic run-time environment, programs cannot be written with specific memory

locations in mind. Programs are not given a fixed place in memory in which to store their data that will be the same between program invocations. To solve this problem, all Macintosh programs are designed to store global variables in a 32K area pointed to by the 68000 register A5. This could be a problem if your needs require you to reference or store your variables in specific memory locations.

The Solution

This problem can be solved if you are willing to use some macros. A set of macros to do this could look something like the following:

```
#include     <sysEqu.h>

#define pShort         *(short *)
#define pLong           *(long *)
#define lmSFSaveDisk        pShort SFSaveDisk
#define lmCurDirStore    pLong CurDirStore

main()
{
    short foo;
    long bar;

    foo = lmSFSaveDisk;
    lmCurDirStore = bar;

/* or */

    foo = pShort SFSaveDisk;
    pLong CurDirStore = bar;
}
```

Back to top

Segmenting and the Jump Table

The Problem

When the Macintosh was first developed, memory space was tight. For this reason, a run-time architecture was designed that allowed programs to be divided into segments that could be dynamically loaded and unloaded. Because of this, a program cannot rely on any specific memory locations into which it can be loaded, and hence it has to be freely relocatable. This means that any intra-segment calls (i.e., calls from one routine to another within the same code segment) have to use the PC-relative addressing modes of the 68000. Since these instructions use only signed 16-bit offsets, these branches are limited to a range of 32K bytes. This, in turn, leads us to the historical 32K limit on 'CODE' resource segments. While the restriction in the linker limiting 'CODE' resources to 32K has been lifted with MPW 3.0, it does not resolve the issues with long distance branching.

In order to be larger than 32K, a program should be divided into multiple 'CODE' resource segments. Calls from a procedure in one segment to a procedure in another segment are called inter-segment calls. These calls are performed through a jump table referenced with positive offsets from A5 (Refer to *Inside Macintosh*, Volume II-53, The Segment Loader, for more information on the jump table). The problems that arise from this mechanism are that ROMable code does not get loaded into memory by a Segment Loader, and supporting an A5 jump table may not be desirable.

The Solution

Programs compiled with Pascal or C currently **always** use the 16-bit PC-relative address mode when generating branch instructions. There is no way to change that. However, there are several ways you can get around it:

- Implement your own A5 world Use islands for long jumps Use assembly language

1. Implement an A5 world in your device that mimics the Macintosh's as closely as possible. This is probably the easiest solution. First, you will be able to program in a normal Macintosh style and not have to take into account

considerations that are presented in solutions #2 and #3. It will also allow you to compile and link your program without having to specify any special options.

After this has been done, and you are ready to download your program to its destination, you can run your program through a filter that: a) determines the final locations of all of the 'CODE' resource segments in the file, and b) creates a jump table with the addresses correctly resolved. In essence, this would be the same as a Macintosh program with all of its segments loaded in memory at the same time.

Let's take a look at an example. Assume that you have developed a program that is about 40K long, and you would like to have it loaded at location $1000. Because of its length, it is divided into two segments. You have one routine in 'CODE' = 1 that is referenced from 'CODE' = 2 and three routines in 'CODE' = 2 that are referenced from 'CODE' = 1. All of these routines will generate jump table entries. In addition, a jump table entry is generated for the main entry point of your program, as per the Segment Loader chapter of *Inside Macintosh* . This gives us a total of five jump table entries in our program. The file created with MPW would look something like the following:

'CODE' = 1

```
00000000:          main()
...          ...
000038B4:          importantRoutine1()
...          ...
```

'CODE' = 2

```
00000000:          importantRoutine2()
...          ...
00003D0F:          importantRoutine3()
...          ...
00004969:          importantRoutine4()
...          ...
```

'CODE' = 0 (the jump table)

```
00000000:
00000008:     $20 bytes of overhead
00000010:
00000018:
00000020:     00 00 3F 3C 00 01 A9 F0     ; dc.w $0000 / MOVE.W #1,-(A7) / _LoadSeg
00000028:     38 B4 3F 3C 00 01 A9 F0
00000030:     00 00 3F 3C 00 02 A9 F0
00000038:     3D 0F 3F 3C 00 02 A9 F0
```

When we create our downloadable image, the routines that we are interested in will end up at these locations:

```
main()               $0000 1000     ($1000 + $0000)
importantRoutine1()     $0000 48B4     ($1000 + $38B4)
importantRoutine2()     $0000 59F0     ($1000 + $49F0 + $0000)
importantRoutine3()     $0000 96FF     ($1000 + $49F0 + $3D0F)
```

Therefore, we should modify our Jump Table to look like this:

'CODE' = 0 (the jump table)

```
00000000:
00000008:      $20 bytes of overhead
00000010:
00000018:
00000020:      00 01 4E 59 00 00 10 00      ; dc.w $0001 / JMP $0000 1000
00000028:      00 01 4E 59 00 00 48 B4
00000030:      00 02 4E 59 00 00 59 F0
00000038:      00 02 4E 59 00 00 96 FF
```

2. For some reason, it may be impossible or undesirable to segment your code in Macintosh fashion. You may be importing source code from somewhere else, or you may not be able to utilize a jump table. In cases like this, where your program has to be compiled as one segment, you will hit problems if it is a large program. The Pascal and C compilers will still limit you to branches smaller than 32K. In the cases where you need to execute long distance jumps, the only thing you can do is create "islands" that allow you to make several short hops to your destination. For instance, if it turns out that you are writing a C program which needs to call a procedure that is 70K away, you will have to break up the branch into three smaller ones as follows:

```
main()

[ ... some random code ... ]

procedureNearTheBeginningOfMyProgram()
{
    ...
    Island1(); /*Calling importantButFarAwayRoutine() */
    ...
}

[ ... 20K of intervening code ... ]

Island1()
{ Island2(); }

[ ... 30K of intervening code ... ]

Island2()
{ importantButFarAwayRoutine(); }

[ ... 20K of intervening code ... ]

importantButFarAwayRoutine()
{
    ...
```

3. If programming little islands into your program is too gross for you to comtemplate, then program using the 68xxx assembler, eschewing the high-level compilers. This will allow you to use the absolute addressing mode directly, avoiding the fact that the compilers will not use them. It will also allow you to store into and access fixed memory locations more easily. The following shows some ways of doing this:

```
test    Main
    import test5
    import test6

    org $1000

    jsr test2              ; test of an intraprocedure call just
    jmp test2              ; a few bytes away.

    jsr test3              ; test of an intraprocedure call a
    jmp test3              ; significant number of bytes away.
test2
    jsr (test4).l          ; test of an intraprocedure call more
    jmp (test4).l          ; than 32K away.

; The following instructions won't work on a 68000, but will on a
; 68020 or better. They demonstrate a better alternative to the above
; method, in that they generate PC-relative branching. In order to
; use them, include "MACHINE MC68020" in your assembly source code.

;    bsr.l test4           ; test of an intraprocedure call more
;    bra.l test4           ; than 32K away.

    lea (test4).l,A0       ; alternate test of a > 32K jump
    jmp (A0)

    ds.b 17000          ; padding to force > 16K jump

test3
    jsr test5              ; test of an interprocedure call a
    jmp test5              ; significant number of bytes away

    jsr (test6).l          ; test of an interprocedure call more
    jmp (test6).l          ; than 32K away

    ds.b 17000          ; padding to force > 32K jump

test4
    rts
    endp


test5    proc
    entry test6

    rts

    ds.b 17000          ; padding to force > 32K jump

test6
    rts

    endp
```

[Back to top](#)

Toolbox and OS Routines

The Problem

Because your program will be operating in a non-Macintosh environment, you will not be able to make any ToolBox or operating system calls. This would not seem to be a problem until you consider that the library routines you are calling my be making such calls themselves. For example, `malloc()` and most `stdio` calls fall into this category.

The Solution

Don't use our libraries. Use your own. Most of the MPW library routines are "clean," but the low-level routines that they rely on use the Toolbox or OS. Identifying those low-level routines that call the Macintosh operating system, determining all the high-level routines that depend upon them, and then programming around them is too difficult a task to undertake. Even if it were done, you would still have to contend with routines that allocated global variables. The best thing to do is avoid our libraries altogether and just write your own.

Back to top

Setting Up Your Run-Time Environment

The Problem

The Pascal and C compilers do some hidden work to initialize the run-time environment before the part of your application that you have written is actually executed. It is possible that you may wish to take advantage of this setup or may need to duplicate it in order to get your program to execute.

The Solution

With Pascal, most of this initialization is automatically inserted into your main procedure. There is very little you can do about it except to put all of your Pascal routines into separately compiled UNITs and write your entry point in C or assembly.

In the case of C, this initialization is performed by a routine in the file CRuntime.o called `CMain()`. The following is a description of what happens to your source code from the time the C compiler gets it to the time the code you have written is executed:

- MPW C compiles all of the source files and creates object files for the linker. All functions are compiled in **exactly** the same way, including `main()`.
- These files are linked together. If you do not link with the file CRuntime.o, these routines will link together, but they will not have an entry point; the linker will not have any routine explicitly defined as the first one to be called, and it will default to setting up the first routine that it finds as the entry point.
- If you **do** link with the file CRuntime.o, then you will be linking with a routine called `CMain()`. This routine is marked as being an Entry routine, and it will be the routine that is executed when you launch the Macintosh program.

- `CMain()` performs the following steps:

1. Call _RTInit (runtime init) Call `setjmp()` Check the result of `setjmp()`. If <> 0, go to 6. Call `main()` Call `exit()` with result from `main()`. RTS

- This is what _RTInit does:

1. Call `_DataInit()`.
2. Save the return address back to whomever ran this program.
3. Check to see if launched by MPW. If not, then setup `argv` and `argc` to indicate the name of the program with no parameters.
4. If launched under MPW, initialize some things so that the run-time environment will integrate with MPW. **Calls the Memory Manager**, so make sure that this part of the code is never executed. This is not likely to happen, as _RTInit checks and validates several memory locations before it gets this far.

- This is what `_DataInit()` does:

1. Assume that A5 is valid, and that there is data appended to the end of `DataInit` that is used to initialize the globals. This will be done by the linker automatically.
2. Determine the size of the globals and zero it out.
3. Read the data at the end of the procedure and use it to initialize the globals. Normally, this process will attempt to

use `_BlockMove` on sufficiently large blocks of data, and a small loop for small blocks of data. A version of `DataInit()` that does not call `_BlockMove` is available from Macintosh Developer Technical Support. However, this limits you to 64K of contiguous pre-initialized storage.

- This is what `exit()` does:

1. Call any user installed exit procedures.
2. If called from MPW, set the value of `{Status}`
3. Determine if `setjmp()` was ever called. If so then call `longjmp()` with a value of 1.
4. If `setjmp()` was never called, then return directly to the process caller, as saved in step two of `_RTInit`.

While MPW was designed with creating Macintosh programs in mind, it can also be used to write software for non-Macintosh targets. After resolving such issues as creating an appropriate run-time environment, making sure that Toolbox calls are not made, and being aware of the 32K limit for branches and jumps, you should be able to use the high-level Pascal and C compilers. By using assembly language, you should even be able to avoid the problems that they pose.

Back to top

References

Inside Macintosh , Volume II-53, The Segment Loader

M.OS.SegmentLoader

NuBus is a trademark of Texas Instruments

Back to top

Downloadables

Acrobat version of this Note (K).

Download