NOTE: This Technical Note has been retired. Please see the Technical Notes page for current documentation.

Technical Note PT14
Glue Code: It Gets You Out of Sticky Situations

CONTENTS

This Technical Note describes glue code, explains how it quietly improves your complicated life, and shows you how to write your own when necessary.

[May 01 1993]

Introduction

Programmers live in a silicon world. Their environment can be made up of a single development platform (for example, MPW Pascal), or it can be mixed and matched (THINK C and MPW Assembly and Pascal, and so on). When an environment is a single package, the designers of that package can take care of the nitty gritty of code interfacing almost transparently. Examples of this transparency would be having a C compiler straighten out all of the parameters necessary to deal with a Toolbox coded in Pascal with Pascal calling conventions or having a compiler generate extra interface code to make sure the programmer never sees some problem. When the environment does this for you, it does it by providing glue code. If you are using more than one development environment, if you need to call register-based assembly routines, or if you would like to use otherwise good routines that are written in another language, you run into the need for glue code.

Glue code is any code that straightens out interfacing protocols between two different calling interfaces; it "glues" together otherwise non-communicating procedures. It also provides a method of prefacing calls to fix parameters, adjusting for environmental oddities, and cleaning up after returns to get the results "just so."

Back to top

Parameter Passing

There are three common ways that routines pass information back and forth.

The first, register-based, is for the routine to store the value in a data register ($D0$), or, if the data is more than a single value, to pass a pointer to a record (parameter block, control block, buffer, and so on) in address register $A0$. This method is commonly found on the Macintosh in the system routines in the ROM, like the Memory Manager. Results are returned in the same way: The result is placed in a register ($D0$) or fields of the parameter record are changed or filled in.

The second method is in Pascal and involves placing values, pointers to variables, and space for the result on the system stack (pointed to by address register $A7$). With this method, values up to 4 bytes are pushed onto the stack, or, if the value is a variable or is longer than 4 bytes, a pointer (itself a 4-byte entity) is pushed onto the stack.

The sequence for this method is as follows:

- (If a function call) push space for or pointer to the result.
- Push the first parameter or pointer, then the second parameter or pointer, then the third, and so on, until all parameters or pointers are pushed.
- Make the call, which places the return address on the stack.

Before returning, the routine called is responsible for placing the result into the result space on the stack or into the space pointed to by the result pointer and popping off all of the parameters.

The third method is in C, and it is much like the Pascal method; however, there are some differences between the two methods. In the C method, no result space is reserved on the stack--the result or the pointer to the result is always passed back in register $D0$; the parameters are pushed into the stack in the reverse order of the Pascal method; and the routine called does not remove parameters from the stack, leaving this clean up for the calling routine to do.

It could be said that C programmers have an advantage, in that their compiler can generate code to make the call with the Pascal method if they declare the function as `pascal`, while Pascal programmers have an advantage on the Macintosh as the majority of the Toolbox managers are designed with Pascal interfaces (with the rest of the calls being register based).

The MPW C compiler also provides the #pragma parameter. This instructs the compiler to pass the handle in AO and flags in DO automatically, and without any glue. Also note that #pragma parameter only works for routines declared as inlines. So you cannot define your own routines as being register-based. An example of using #pragma parameter to better "glue in" an inline call to HSetState would be to:

and then define your inline as:

Back to top

What's Going On Here?

"We are writing a HyperCard XFCN in MPW Pascal that calls an external function written in MPW Assembler. When our assembler code gets control, we find that in addition to pushing return space and parameters onto the stack, the Pascal code has also pushed `A6`. Is this normal?"

This section includes a template that is useful for most of the kinds of glue you could encounter. In addition to demonstrating the Pascal method of parameter passing already described, it also sets up for the use of address register `A6` as a local (and recursive) frame reference.

When Pascal calls a procedure or function in Pascal or assembly (or if assembly calls Pascal for that matter), it takes the following steps to set up the stack to: pass parameters, have a result returned, be able to return from the code, and allow for local variables.

If there is a returned value (that is, if the call is to a function), the calling routine makes space on the stack for either the returned value (if it is four bytes or less) or a pointer to the data to be returned (in this case the pointer size of 4 bytes is reserved). For example, for the call:

a calling routine would do the following:

1. Reserve 4 bytes for a returned value. Typically, the calling routine would subtract 4 bytes from the current address in the Stack Pointer, then zero the 4 bytes to which it then points (that is, `CLR.L –(SP)`).
2. Push the value (or a pointer to the value) for each parameter in the call onto the stack. In the case of `VAR` parameters, a calling routine should always pass a pointer to the value, regardless of byte count. In this example, Pascal would generate a `PEA MyValue(A5)` if `MyValue` is a global variable or a `PEA MyValue(A6)` if it is a local variable.
3. Make the call with a `BSR MyFunction`, remembering that this pushes the 4-byte return address onto the stack.

At this point, the stack is set up properly. Listing from low addresses to high addresses, there is the Stack Pointer, which points to the first byte of the return address; next are 4 bytes that contain the pointer to `MyValue`; and following that are 4 bytes of zero, which are ready to hold the return value. Above all this in the stack, there may well be another of this same kind of structure from the call to the procedure that calls `MyFunction`. Figure 4 in *Inside Macintosh* Volume I, page 92, "Using Assembly Language," illustrates this structure in a general way.

Once you call a subroutine, it may want to allocate space for its own local variables. Even if it does not need to allocate this space, it is convenient to set up a register to access the result, parameters, and return address. By convention, register `A6` is used for the purpose, and you can use the assembly-language instruction `LINK` (later followed by `UNLK` to remove the link) to set up this stack frame. Pascal (and most of the assembly language seen on the Macintosh) use

where `MyLocalSize` is a negative value representing the total number of bytes needed to hold the local variables. This instruction does the following:

1. Pushes the old value of `A6` onto the stack (thus saving the original calling routine's stack frame) and setting `A6` to point to this location. By setting `A6` to point to this location, you can retrieve the old `A6` later, use offsets from `A6` to access the result, parameters, and return address, and use negative offsets from `A6` to access all of your local variables.
2. Adds `MyLocalSize` (a negative value) to the stack pointer, thus allocating local variable space and setting up the stack for additional use (that is, more calls from this subroutine).

The following template is an assembly-language version of the Pascal procedure definition, which is taken from the MPW example program Sample.a:

```
* PROCEDURE MyProcedure (Param1: type, Param2: type, ParamN: type) Result1: type;
*
* VAR  Local1: type;
*      Local2: type;
*      LocalN: type;
*
* or for C
* pascal type MyProcedure(Param1 type, Param2 type, ParamN  type)
*

MyProcedure     PROCEDURE    EXPORT              ; any source file can use this routine

* define registers that need to be saved as EQUATES (typically A3-A7 and D4-D7)

StackFrame      RECORD   {A6Link},DECR   ; build a stack frame record
Result1         DS       {size}          ; function's result returned to caller
ParamBegin      EQU      *               ; start parameters after this point
Param1          DS.      {size}          ; the first parameter on the stack
Param2          DS.      {size}          ; rest of the parameters passed by caller
ParamN          DS.      {size}          ; the last parameter passed by caller
ParamSize       EQU      ParamBegin-*    ; size of all the passed parameters
RetAddr         DS.L     1               ; placeholder for return address
A6Link          DS.L     1               ; placeholder for A6 link
Local1          DS.      {size}          ; a local variable for this procedure only
Local2          DS.      {size}          ; other local variables for this procedure
LocalN          DS.      {size}          ; the last local variable
LocalSize       EQU      *               ; size of all the local variables
                ENDR
* The above record can be confusing the first time you see it as it generates both
* positive and negative offsets. The MPW 3.0 Assembly Reference provides a complete
* description of how this works on pages 78-80.

                WITH    StackFrame          ; cover our local stack frame
                LINK    A6,#LocalSize       ; allocate our local stack frame

*               save registers trashed by this routine

*               # # INSERT YOUR CODE HERE # #

Exit            restore registers trashed by this routine
                UNLK    A6                  ; destroy the link
                MOVEA.L (SP)+,A0            ; pull off the return address
                ADDA.L  #ParamSize,SP       ; strip all of the caller's parameters
                JMP     (A0)                ; return to the caller
```

The UNLK A6 (UnLink) instruction near the end of this example sets the stack pointer to the current A6 (automatically deleting the local variable allocation) and pulls the old A6 off the stack, putting it into A6, thus restoring the calling routine's local stack frame. All of this rigmarole allows for recursive subroutines as well as a stack frame list for intelligent debuggers to know about your procedure's calling history. The code following the UNLK instruction basically deallocates the parameters from the stack, leaving just your returned value.

As for the original question, the A6 you see on the stack when your XFCN is called is just the calling routine's stack frame (pointing back the calling chain of stack frames), which should be all set up for your LINK and local stack frame use. For more detailed information on this topic, refer to *Inside Macintosh* Volume l, page 85, "Using Assembly Language."

Back to top

A Real-World Example

"Okay, so what does the code look like that we insert to do the glue call?"

The following code is actual glue to return an extra parameter, which is available in a returned register, but not already part of the standard Pascal interface.

```
*       =================================================
*       PROCEDURE MungerGlue(h: Handle; offset: LONGINT;
*                       ptr1: Ptr; Len1: LONGINT;
*                       ptr2: Ptr; Len2: LONGINT;
*                       VAR ErrFromD0: LongInt) : LONGINT;
*       Glue code to call Munger and return the undocumented
*       error return code available in D0.
*       =================================================
                        PRINT       PUSH,OFF            ; don't print any of this stuff
                        INCLUDE     'Traps.a'
                        PRINT       POP

MungerGlue              PROCEDURE   EXPORT

StackFrame              RECORD      {A6Link},DECR       ; build a stack frame record
MungerGlueReturn        DS.L        1                   ; LongInt returned value
ParamBegin              EQU         *                   ; start parameters after this point
h                       DS.L        1                   ; Handle
offset                  DS.L        1                   ; LongInt
ptr1                    DS.L        1                   ; Pointer
len1                    DS.L        1                   ; LongInt
ptr2                    DS.L        1                   ; Pointer
len2                    DS.L        1                   ; LongInt
ErrFromD0               DS.L        1                   ;  New LongInt
ParamSize               EQU         ParamBegin-*        ; size of all the passed parameters
RetAddr                 DS.L        1                   ; place holder for return address
A6Link                  DS.L        1                   ; place holder for A6 link
* No need for local variables but any would be inserted here
LocalSize               EQU         *                   ; size of all the local variables
                        ENDR

                        WITH        StackFrame          ; Use this record for our local stack
                                ; frame.
                        Link        A6,#LocalSize       ; Allocate our local stack frame.
                        Clr.L       -(SP)               ; Reserve space for out returned value.
                        Move.L      h(A6),-(SP)         ; Place parameters on the stack
                        Move.L      offset(A6),-(SP) ;     by copying the values passed
                        Move.L      ptr1(A6),-(SP)   ;     to us in the same order that
                        Move.L      len1(A6),-(SP)   ;     they appear on the stack as
                        Move.L      ptr2(A6),-(SP    ;      passed to us.
                        Move.L      len2(A6),-(SP)   ;
                        _Munger
                        Move.L      (SP)+,MungerGlueReturn(A6)  ; Pass what is returned to
                                                            ; us back to our caller.
                        MoveA.L     ErrFromD0(A6),A0 ; Pass the value from register D0 into
                        Move.L      D0,(A0)             ; the variable pointed to by the new
                                ; parameter.
Exit                    Unlk        A6                  ; Clean up for return by destroying the
                        MoveA.L     (SP)+,A0
; link. Pull off the return address.
                        AddA.L      #ParamSize,SP       ; Strip all of the caller's parameters.
                        Jmp         (A0)                ; Return to the caller.
```

This glue includes the file Traps.a for the definition for the symbol (label) _Munger, and it is surrounded by PRINT pseudo-operators to tell the assembler not to include the entire file in the listing. The PROCEDURE statement defines MungerGlue as the entry point (or procedure name) of this routine, while the code between the RECORD and ENDR instructions define a record named StackFrame that defines the information passed to this routine on the stack. Finally, the body of the assembly-language code passes the parameters it receives to the _Munger call, passes back the additional value, and then cleans up the stack and returns.

Back to top

Can You Have Too Many Examples?

This final example is of Pascal calling a native C routine (not defined as a pascal function). Assuming the routine

was instead defined in C as

then you would need the following glue to make this call from Pascal:

```
* FUNCTION MyCBitTstGlue(bytePtr: Ptr;bitNum: LONGINT): BOOLEAN;

MyCBitTstGlue PROCEDURE     EXPORT              ; any source file can use this routine

StackFrame    RECORD  {A6Link},DECR            ; build a stack frame record
Result1       DS.W    1                        ; Boolean always takes a word on the stack
ParamBegin    EQU     *                        ; start parameters after this point
bytePtr       DS.L    1                        ; Pointer
bitNum        DS.l    1                        ; LongInt
ParamSize     EQU     ParamBegin-*             ; size of all the passed parameters
RetAddr       DS.L    1                        ; placeholder for return address
A6Link        DS.L    1                        ; placeholder for A6 link
LocalSize     EQU     *                        ; size of all the local variables
              ENDR

              IMPORT  CBitTst
              WITH    StackFrame               ; cover our local stack frame
              LINK    A6,#LocalSize            ; allocate our local stack frame

              Move.L  bitNum(A6),-(SP)         ; Nth parameter
              Move.L  bytePtr(A6),-(SP)        ; N-1th (actually 1st) parameter
              Jsr     CBitTst                  ; Call C version of BitTst
              Move.W  D0,Result1(A6)           ; return to Pascal the value returned by C
                      ; Exit:
              UNLK    A6                       ; destroy the link
              MOVEA.L (SP)+,A0                 ; pull off the return address
              ADDA.L  #ParamSize,SP            ; strip all of the caller's parameters
              JMP     (A0)                     ; return to the caller
```

It is interesting to note another problem that Pascal has with C. Since, in C, the calling routine removes the parameters after the call, C programmers can make a call to a routine with more parameters than necessary, which means that they can call a function with a variable number of parameters. Although glue can straighten this out, the programmer must know how many parameters are actually being passed to make the connection.

Back to top

Don't Abandon All Hope All Ye High-Level Programmers

Even though all the examples in this Note are in assembly-language code, it is possible to write some glue in C. In the case of creating Pascal interfaces for existing C object modules, the C programmer can create a small `pascal` function that calls C as C itself. In addition, the techniques in this Note work the same for any other language, only the interfacing details differ. If using another language, refer to the language manuals for the specific interfacing details.

The moral of the story is that by carefully applying a little glue code, you can greatly simplify your development in mixed-language environments as well as getting yourself up to speed quickly when you encounter new interfaces (such as QuickTime, AppleScript, and so on) that your development environment does not already include.

Back to top

References

Inside Macintosh, Volume I, Introduction to Memory Management

Inside Macintosh, Volume I, Using Assembly Language

MPW reference manuals, various chapters on parameter passing conventions

`DumpObj` output from any code in which you have an interest

THINK C and THINK Pascal are trademarks of Symantec Corporation.

Back to top

Downloadables

| | Acrobat version of this Note (K). | Download |

Technical Notes by Date | Number | Technology | Title
Developer Documentation | Technical Q&As | Development Kits | Sample Code