TECHNOTE:

Using QuickDraw GX Functionality from Pascal or Modula-2 — Without Writing Any C Code

By Lawrence D'Oliveiro
ldo@waikato.ac.nz
The University of Waikato, New Zealand

QuickDraw GX was designed entirely with the C programmer in mind. As a consequence, the user of a high-level language, such as Pascal or Modula-2, is left unable to take advantage of all the capabilities of GX. This Technote outlines how you can get around this hurdle.

This Technote is aimed at programmers with some 68K assembly-language experience, who have an aversion to the C programming language. You will need to be familiar with the Macintosh Programmer's Workshop (MPW) environment, if only because that hosts the only 68K assembler on the Macintosh worth using. I will discuss how to make use of *all* the functionality of GX from a language like Pascal or Modula-2, without writing any C code.

The discussion breaks the problem down into all the important cases, and gives an example of how to deal with each. Based on this, you can construct your own adaptation of Apple's GX interfaces, or you can make use of the adaptation (in Modula-2) I have already done, which is available at <ftp://ftphost.waikato.ac.nz/pub/ldo/MyModLib.sit>. The examples use Modula-2 rather than Pascal, mainly because Modula-2 allows the definition of routine types, which Pascal does not.

Note: This discussion applies only to using GX from 68K code. Differences in argument-passing conventions may be less of an issue in PowerPC code.

Overview of the Problem

QuickDraw GX is a large piece of code, consisting of a number of subsystems. At the core is the graphics engine, and closely connected with this are the font and layout routines. All this code follows C calling conventions exclusively.

The GX Printing Manager uses an odd mixture of C and Pascal calling conventions. All application-level printing calls (and most calls back to application routines, apart from message overrides) use Pascal conventions, whereas calls made from drivers and extensions (and calls made *to* them from GX) follow C conventions.

Ancillary services, used heavily by the GX printing architecture, are provided by the Message Manager and the Collection Manager. The Collection Manager uses Pascal conventions exclusively, so I don't need to discuss it further. The Message Manager uses C conventions exclusively.

Differences Between C and Pascal Calling Conventions

The salient differences between C and Pascal calling conventions for 68K code, as far as GX calls are concerned, are:

- arguments end up in the opposite order on the stack;
- Pascal expects the callee to pop the argument list, while C expects the caller to do this;
- function results of 4 bytes or less are returned in register D0 in C, on the stack in Pascal.

There are other differences, such as treatment of arguments and results larger than 4 bytes, but it so happens these cases do not arise in the problem at hand, so they can be ignored for now.

There are two halves to the problem: your code calling GX, and GX calling your code. The second is by far the more difficult one.

Calling GX From Pascal-Conformant Code

In 68K code, all the GX graphics, layout and font functions are invoked via the same A-trap, $A832. (Earlier versions of MacsBug named this trap as "Skia," which was the code name for GX graphics.) To call an A-trap routine that follows

C-language conventions from Pascal-conformant code, in general you have to generate "glue" code that does the following:

■ Make a copy of the argument list on the stack with the arguments in reverse order;

■ Call the A-trap routine;

■ If the routine returns a function result, then copy register D0 into the stack location where Pascal expects the result to be;

■ Pop both copies of the argument list off the stack.

One fact that helps to simplify things is that the arguments to, and results from, GX routines are exclusively 32-bit quantities (the only exceptions being GX Printing Manager-related calls, which follow Pascal conventions anyway).

You can further simplify this in the situation where the routine takes no more than one argument: in this case, there is no need to reorder the argument list—you can pass the compiler-generated list directly to the GX routine. In fact, the glue becomes so simple, it can be done as an inline.

Following are examples of all of the cases that can be handled with inlines. These can be adapted simply by changing the selector code as appropriate. First, the simplest possible case: a GX routine that takes no arguments and returns no result:

```
PROCEDURE GXEnterGraphics;

   CODE
        0705FH,        (* moveq.l #$5F, d0 *)
        0A832H;        (* _Skia *)
```

It so happens that all the calls with no arguments and no result have selector codes between 0 and 127, hence the above example can be adapted directly, including the use of the moveq instruction.

Here is one with a result but no arguments:

```
PROCEDURE GXNewInk() : gxInk;

     CODE
          0303CH, 0009DH, (* move.w #$9D, d0 *)
          0A832H,         (* _Skia *)
          02E80H;         (* move.l d0, (sp) *)
```

This one has an argument but no result:

```
PROCEDURE GXDrawShape
  (
     source : gxShape
  );

     CODE
          0303CH, 000DCH,  (* move.w #$DC, d0 *)
          0A832H,          (* _Skia *)
          0588FH;          (* addq.l #4, sp *)
```

And this one has both a single argument and a result:

```
     PROCEDURE GXNewShape
       (
          aType : gxShapeType
       ) : gxShape;

          CODE
               0303CH, 0009EH,  (* move.w #$9E, d0 *)
               0A832H,          (* _Skia *)
               0588FH,          (* addq.l #4, sp *)
               02E80H;          (* move.l d0, (sp) *)
```

When you have more than one argument, then it becomes easier to call an assembly-language glue routine, than to use an inline. The repetitiveness of the structure of the glue makes it natural—in fact, imperative—to use assembly-language macros to generate it.

Calling Routines with More than One Argument

Where the GX routine you're calling has more than one argument, the glue you need to interface to it would look something like this:

```
          lea       4(sp), a0  ; address of argument list
          move.l    (a0)+, -(sp) ; copy arguments in reverse order
          move.l    (a0)+, -(sp) ; repeat as many times as necessary
          ...
          move.w    #TrapCode, d0
          dc.w      $A832         ; call the GX routine
          add.l          #NrArgs*4, sp ; pop the copied argument list
          move.l    (sp)+, a0     ; get the return address
          add.l          #NrArgs*4, sp ; pop the original argument list
          move.l    d0, (sp)      ; return the result (if appropriate)
          jmp       (a0)          ; back to caller
```

My actual macros make a few embellishments to this. First off, not all the arguments are actually longwords: there are a couple of layout routines (GXGetOffsetGlyphs and GXGetGlyphOffset) with arguments that are more naturally represented as booleans. The glue will take care of converting all arguments to longwords, as GX expects.

Here is a basic macro to generate the GX glue:

```
_Skia     opword    $A832

     macro
     GXCall    &ModuleName, &RoutineName, &ArgList, &Result=, &TrapCode
.* General macro for generating call glue for both A-trap routines
.* and library routines.
.* ModuleName is the name of the Modula-2 definition module
.* containing the external declaration; RoutineName is the name of
.* the routine. ArgList is the list of arguments,
.* each argument being of the form <name>.<size> (name is ignored,
.* but is useful for checking purposes), Result specifies the
.* function result size (b, w, l for byte word or long, omit if
.* no result), and TrapCode is the A-trap selector.
     seg   '&ModuleName'
&ModuleName._&RoutineName proc export
     lcla  &ArgIndex, &ArgListLength, &NrArgs, &ResultSize
     lclc  &ThisArgSize, &ThisArg
&NrArgs seta &nbr(&ArgList)
    if &NrArgs <> 0 then
     lea   4(sp), a0
    endif
    if &Result <> '' then
    if &Result = 'b' then
&ResultSize seta 1
    elseif &Result = 'w' then
&ResultSize seta 2
    elseif &Result = 'l' then
&ResultSize seta 4
    else
     aerror     &Concat('Unrecognized result size: ', &Result)
    endif
    else
.* no result
&ResultSize seta 0
    endif
&ArgListLength seta 0
&ArgIndex seta &NrArgs
    while &ArgIndex > 0 do
&ThisArg setc &ArgList[&ArgIndex]
&ThisArgSize seta &pos('.', &ThisArg)
&ThisArg setc &ThisArg[&ThisArgSize + 1 : 99]
    if &ThisArg = 'b' then
&ThisArgSize seta 2
     move.b     (a0)+, d0
     tst.b(a0)+ ; bytes are pushed as words
     extb.l     d0
     move.l     d0, -(sp)
    elseif &ThisArg = 'w' then
&ThisArgSize seta 2
     move.w     (a0)+, d0
     ext.ld0
     move.l     d0, -(sp)
    elseif &ThisArg = 'l' then
&ThisArgSize seta 4
     move.l     (a0)+, -(sp)
    else
     aerror     &Concat('Unrecognized arg size: ', &ArgList[&ArgIndex])
    endif
&ArgListLength seta &ArgListLength + &ThisArgSize
&ArgIndex seta &ArgIndex - 1
    endwhile
     move.w     #&TrapCode, d0
     _Skia
    if &NrArgs <> 0 then
     add.l#&NrArgs*4, sp
    endif
     move.l     (sp)+, a0
    if &ArgListLength <> 0 then
     add.l#&ArgListLength, sp
    endif
    if &ResultSize = 1 then
     move.b     d0, (sp)
    elseif &ResultSize = 2 then
     move.w     d0, (sp)
    elseif &ResultSize = 4 then
     move.l     d0, (sp)
    endif
```

```
     jmp   (a0)
     endproc
     endm
```

And here is an example of how to use the above. First, the original C prototype of the GX routine:

```
     void GXGetOffsetGlyphs
       (
          gxShape layout,
          gxByteOffset trial,
          long leadingEdge,
          gxLayoutOffsetState *offsetState,
          unsigned short *firstGlyph,
          unsigned short *secondGlyph
       );
```

Here’s how I declared it in Modula-2:

```
     PROCEDURE GXGetOffsetGlyphs
       (
          layout : gxShape;
          trial : gxByteOffset;
          leadingEdge : BOOLEAN;
          VAR offsetState : gxLayoutOffsetState;
          VAR firstGlyph : ShortCard;
          VAR secondGlyph : ShortCard
       );
```

Here’s the macro call to generate the glue:

```
     GXCall     GXLayoutRoutines, GXGetOffsetGlyphs, (layout.l, trial.l,
leadingEdge.b, offsetState.l, firstGlyph.l, secondGlyph.l), , $15
```

And a quick dump of the generated object code:

```
     LEA        $0004(A7),A0
     MOVE.L     (A0)+,-(A7)
     MOVE.L     (A0)+,-(A7)
     MOVE.L     (A0)+,-(A7)
     MOVE.B     (A0)+,D0
     TST.B      (A0)+
     EXTB.L     D0
     MOVE.L     D0,-(A7)
     MOVE.L     (A0)+,-(A7)
     MOVE.L     (A0)+,-(A7)
     MOVE.W     #$0015,D0
     DC.W       $A832                 ; TB 0032
     ADDA.W     #$0018,A7
     MOVEA.L    (A7)+,A0
     ADDA.W     #$0016,A7
     JMP        (A0)
```

Note the use of the EXTB instruction for extending the byte boolean to a long. That’s only valid on 68020 and better processors—but then, GX won’t run on anything less!

Calling Pascal Code From GX

This part of the problem is the opposite of the previous one: here you have to wrap your routine in some "glue" that will convert between calling conventions, and pass that to GX for it to call.

This part is best broken into two main cases, with different applicable methods of attack. The first is, when calling a GX routine, how to pass the address of one of your routines for it to call back. The second is how to write a GX printing extension or driver in a language that uses Pascal calling conventions.

Implementing a GX Callback

One way to let GX call one of your routines is to write wrapper glue for it in assembly language. There are a couple of reasons why I feel this is undesirable:

■ It is repetitive, tedious and error-prone. Having done it once, of course, you could copy and paste (and adapt) the code to a new project, but that is *not* a particularly effective way to reuse code.
■ It adds extra source files, which needlessly complicate the building of your project.

Instead, my preferred approach is to have a library routine to generate the appropriate glue at run-time. I can write the routine once for each case, and reuse it every time that case arises. Some people may throw their hands up in horror at this, and cry "self-modifying code!" However, the technique is in fact quite simple and safe to use — certainly simpler and safer than most of the alternatives...

The case I will use as an example is how to define the gxSpoolProcedure callback that you pass to GX when flattening and unflattening shapes. One nice thing about defining your own interfaces is that you can make tweaks to improve their usability in some way. For instance, here is how I define the type of a gxSpoolProcedure in Modula-2:

```
gxSpoolProcedure =
     PROCEDURE
       (
          (*command :*) gxSpoolCommand,
          (*block :*) gxSpoolBlockPtr,
          (*arg :*) ADDRESS (* meaning is up to caller *)
       ) : LONGINT;
```

The difference from the “official” definition is the addition of an extra argument, which allows me to pass additional information to the routine. This information is passed to the glue-generating routine, and bound into the generated wrapper code, so GX doesn’t need to know it is being passed!

Here is the structure for holding the wrapper glue code;

```
     TYPE
          BoundgxSpoolProcedure = ARRAY [1 .. 13] OF ShortCard;
```

And here is the routine that takes the address of your spool callback routine and the value of the extra argument, and actually generates the wrapper glue:

```
     PROCEDURE BindgxSpoolProcedure
       (
          TheProc : gxSpoolProcedure;
          ToArg : ADDRESS;
          VAR Result : BoundgxSpoolProcedure
       );
       (* creates a closure of the specified spool procedure
          which can be passed where QuickDraw GX expects one. *)

     BEGIN
          Result[1] := 041EFH; (* lea n(sp), a0 *)
          Result[2] := 00004H; (* n for above *)
          Result[3] := 042A7H; (* clr.l -(sp) *) (* room for result *)
          Result[4] := 02F18H; (* move.l (a0)+, -(sp) *) (* copy command arg
*)
          Result[5] := 02F18H; (* move.l (a0)+, -(sp) *) (* copy block arg *)
          Result[6] := 02F3CH; (* move.l #n, -(sp) *) (* insert additional arg
*)
          Result[7] := HiWrd(ToArg); (* n for above *)
          Result[8] := LoWrd(ToArg); (* ditto *)
          Result[9] := 04EB9H; (* jsr l *)
          Result[10] := HiWrd(CAST(LongCard, TheProc)); (* l for above *)
          Result[11] := LoWrd(CAST(LongCard, TheProc)); (* ditto *)
          Result[12] := 0201FH; (* move.l (sp)+, d0 *) (* return result where
C expects it *)
          Result[13] := 04E75H; (* rts *)
          FlushCaches
     END BindgxSpoolProcedure;
```

As an example of how to use the above, here is my version of the HandleToShape routine from the Apple-provided GX library source code. Note in particular how the extra argument is used to pass the environment pointer to allow access to outer local variables from the inner routine:

```
     PROCEDURE HandleToShape
       (
          TheHandle : Handle;
          NrViewPorts : LONGINT;
          ViewPorts : gxViewPortPtr (* array *)
       ) : gxShape;
       (* unflattens a handle into a new shape. Any graphics errors are
          automatically posted. *)

          VAR
```

```
                SrcOffset : LONGCARD;
                BoundSpoolFromHandle : BoundgxSpoolProcedure;
                HandleSpool : gxSpoolBlock;

            PROCEDURE SpoolFromHandle
              (
                command : gxSpoolCommand;
                VAR block : gxSpoolBlock
              ) : LONGINT;
              (* spool routine to retrieve the flattened shape data from
TheHandle. *)

            BEGIN
                SAVEREGS;
                CASE VAL(INTEGER, command) OF
                | GXGraphicsTypes.gxReadSpool:
                    BlockMoveData
                      (
                        (*sourcePtr :=*) TheHandle^ + SrcOffset,
                        (*destPtr :=*) block.buffer,
                        (*byteCount :=*) block.count
                      );
                    SrcOffset := SrcOffset + VAL(LONGCARD, block.count)
                ELSE
                  (* ignore *)
                END (*CASE*);
                RETURN
                    0
            END SpoolFromHandle;

        BEGIN (*HandleToShape*)
            SrcOffset := 0;
            BindgxSpoolProcedure
              (
                (*TheProc :=*)
                    CAST(GXUseful.gxSpoolProcedure, ADR(SpoolFromHandle)),
                (*ToArg :=*) CurrentA6(),
                (*@Result :=*) BoundSpoolFromHandle
              );
            HandleSpool.spoolProcedure := ADR(BoundSpoolFromHandle);
            HandleSpool.buffer := NIL; (* let GX allocate the buffer *)
            HandleSpool.bufferSize := 0; (* ditto *)
            RETURN
                GXUnflattenShape
                  (
                    (*@block :=*) HandleSpool,
                    (*count :=*) NrViewPorts,
                    (*portList :=*) ViewPorts
                  )
        END HandleToShape;
```

Implementing a GX Extension or Driver

When trying to write a GX printing extension or driver in a language that conforms to Pascal calling conventions, you have the job of generating appropriate glue for each of your message overrides. As I made clear in the previous section, doing this by hand, in assembly language, is not my idea of fun.

Thus, I hit upon the idea of writing an MPW tool, called PrintingSegment, to automatically generate the glue code. The result takes the form of a .o object file that you link at the front of your “pext” or “pdvr” code segment. The same tool can also generate the “over” resources that tell GX printing where the routine entry points are and which messages they handle. Thus, the tool also solves the problem of keeping the “over” resources in sync.

I did make one simplification: I decided that giving the tool the information necessary to copy and convert the argument list was too much work. Instead, I pass a single pointer to the argument list as the argument to the Pascal-conformant routine.

For example, consider the interface for an override for the gxDespoolPage message. In Apple’s C interfaces, this is defined as

```
typedef OSErr (*GXDespoolPageProcPtr)
  (
     gxSpoolFile theSpoolFile,
     long numPages,
     gxFormat theFormat,
     gxShape *thePage,
     Boolean *formatChanged
  );
```

The following record structure exactly mirrors the layout of the C argument list:

```
TYPE
     BooleanPtr = POINTER TO BOOLEAN;
     gxShapePtr = POINTER TO gxShape;
     GXDespoolPageArgs =
          RECORD
               theSpoolFile : gxSpoolFile;
               PageNumber : LONGINT;
               theFormat : gxFormat;
               thePage : gxShapePtr;
               formatChanged : BooleanPtr
          END (*RECORD*);
```

(Note that all the C arguments are passed as longwords.) So the Pascal-conformant override for this message is declared as:

```
PROCEDURE HandleDespoolPage
  (
     VAR Args : GXDespoolPageArgs
  ) : OSErr;
```

and the glue that PrintingSegment generates to call this override looks like this:

```
CLR.W      -(A7)
PEA        $0006(A7)
JSR        HandleDespoolPage
MOVE.W     (A7)+,D0
RTS
```

Here is what the body of HandleDespoolPage might look like, in a printing extension that allows the user to reverse the order in which pages are printed:

```
PROCEDURE HandleDespoolPage
  (
     VAR Args : GXDespoolPageArgs
  ) : OSErr;
  (* override for gxDespoolPage message. *)

     VAR
          ThisPageNumber : LONGINT;
          Err : OSErr;

BEGIN
     SAVEREGS;
     IF ReverseOrderPrinting THEN
          ThisPageNumber := NrPages - Args.PageNumber + 1
     ELSE
          ThisPageNumber := Args.PageNumber
     END (*IF*);
     Err := ForwardGXDespoolPage
       (
          (*theSpoolFile :=*) Args.theSpoolFile,
          (*PageNumber :=*) ThisPageNumber,
          (*theFormat :=*) Args.theFormat,
          (*@thePage :=*) Args.thePage^,
          (*@formatChanged :=*) Args.formatChanged^
       );
     RETURN
          Err
END HandleDespoolPage;
```

Summary

It is certainly feasible to write all kinds of GX-based code — from straight applications to printer drivers and extensions — in a high-level language rather than C.

Acknowledgments

Thanks to Ingrid Kelly and Dave Polaschek for reviewing this Technote. Thanks to Niklaus Wirth for creating Pascal, then improving it with Modula-2. Thanks to the MPW engineers at Apple for creating what is still the only development environment on the Macintosh that will never hold you back from wherever you want to go. Thanks also to Patrick McGoohan and Sir Clough Williams-Ellis, for reminding us of the need sometimes to hold out against conformity.

Modification History

LDO 1996 August 4: This draft.