

# New Technical Notes

Macintosh



®

---

Developer Support

## Getting through CUSToms Platforms & Tools

Revised by:  
Written by: Rick Blair

March 1988  
July 1987

This technical note provides a way for developers to allow sophisticated users to add code to an off-the-shelf application. Using this scheme, the user can easily install the code module; the application has to know how to call it and, optionally, be able to respond to a set of predefined calls from the custom package.

---

### Note

The following code makes heavy use of features of the Macintosh Programmer's Workshop. It also assumes a basic familiarity with the standard Sample program included with MPW. The Pascal code (which is here only as an example implementation of the mechanism) is presented as only those sections which *differ* from Sample.p. The assembly language code also includes MPW-only features, such as record templates. Some of these are explained in TM.PT.Signals.

In addition, since the order in which parameters to various routines are passed is critical, special care will have to be taken in writing interfaces for use with C. It is probably best to declare them as Pascal in the C source.

### Concepts

Basically, we create a code resource of type CUST with an entry point at the beginning which takes several parameters on the stack; this code is reached via a dispatching routine which is written in assembly language.

The data passed on the stack to this dispatcher includes:

- a selector (to specify the operation desired)
- the address of a section of application globals (for communication back and forth between the application and the module when the stack parameters are insufficient)
- a handle which references the custom code resource on the stack.

Other parameters may be added (as long as they are pushed on the stack before the required ones) if desired. Since these extra parameters would **always** have to be included in any calls to a given package, it might be more convenient to use the application global space area which is accessed through the `appaddr` parameter.

## Template

Your application must contain the following global data and procedure declarations to support this model:

```
VAR
    custhandle: Handle;

    {the following globals constitute the data known to the custom code}
    appdispatch: ProcPtr; {address of dispatch routine custom code can call}
    {examples of further application globals for the custom package:}
    (*
    paramptr: Ptr; {general pointer used as param. to appdispatch code}
    paramword1: INTEGER;
    paramword2: INTEGER;
    CUSTerr: INTEGER;
    *)
    {any other globals the module should get at}

    {the two assembly language glue routines which are linked into the
    application}
    PROCEDURE CustomInit(resID: INTEGER; VAR custhandle: Handle);
    EXTERNAL; {the routine used to set up the custhandle resource handle}

    PROCEDURE CustomCall({application & package-specific paramters}
        selector: INTEGER; appaddr: UNIV Ptr; ourhandle:
        Handle);
    EXTERNAL; {this is the code dispatcher}

    {this is called by the custom package to perform a service which is more easily
    provided by the application; since we pass a pointer to it to the package, CustDispatch
    must be at the outermost nesting level in the main segment }

    PROCEDURE CustDispatch(selector: INTEGER);

BEGIN
    CASE selector OF
        {.
        .
        .}
    END; {CASE}
    END; {CustDispatch}

{your initialization code should contain the following:}

    {Custom package initialization stuff}
    appdispatch := @CustDispatch; {put pointer where the package can see it}
    CustomInit(69,custhandle);    {our CUST resource has ID = 69}

    {then whenever you want to invoke the package you use CustomCall}
```

You must also assemble CustomInit and CustomCall and link them with into your application. The custom package itself can be written in any language which can produce stand-alone code.

## The example

CustomCall is only referenced once in this example. When a variety of unrelated functions are provided, however, it is more convenient to provide a separate interfacing procedure to invoke each one and have them make their own CustomCall calls.

Note that this example is somewhat contrived; you probably wouldn't "externalize" the code for finding a word or sequence of characters like this. This is an idealized situation. More realistic uses would be: to add-on special routines to a database to perform custom calculations or the like; allow for localization when code is required (and hooks aren't already provided); let documents carry around code which may vary among software versions, etc. so that older documents would be able to work alongside the new ones, etc.

## What it does

We simply add a new menu to the sample program which allows **Find** by characters or word. We just pass the menu item to the package and let it do the finding; it then calls back to the application dispatch routine to highlight text or display the "not found" message.

The Pascal source for the example application appears first:

```
{ $R- }
{ $D+ }
PROGRAM P;

  USES
  { $LOAD :: PInterfaces:most.dump }
  Memtypes, Quickdraw, OSIntf, ToolIntf, PackIntf {, MacPrint }
  { $LOAD }
  , { $U ErrSignal.p } ErrSignal;

  CONST
  appleID = 128; { resource IDs/menu IDs for Apple, File and Edit menus }
  fileID = 129;
  editID = 130;
  findID = 131;

  appleM = 1; { index for each menu in myMenus (array of menu handles) }
  fileM = 2;
  editM = 3;
  findM = 4;

  menuCount = 4; { total number of menus }

  windowID = 128; { resource ID for application's window }

  undoCommand = 1; { menu item numbers identifying
                    commands in Edit menu }
  cutCommand = 3;
  copyCommand = 4;
  pasteCommand = 5;
  clearCommand = 6;

  findcharsCommand = 1; { menu items for Custom menu }
  findwordCommand = 2;
```

```
aboutMeCommand = 1; {menu item in apple menu for About sample item}

aboutMeDLOG = 128;
findDLOG = 129;
infoDLOG = 130;

{application dispatching code selectors}
highlightSel = 0;
notifySel = 1;

VAR
    •
    •
    •
errCode: INTEGER;
dlogString: Str255;
cushandle: Handle;

{here is the area known to the custom code}
appdispatch: ProcPtr;      {address of dispatch routine custom
                           code can call}

    {examples of further application globals for the custom package}
paramptr: Ptr; {general pointer used as param. to appdispatch code}
paramword1: INTEGER;
paramword2: INTEGER;
{any other globals the module should get at}

PROCEDURE CustomInit(resID: INTEGER; VAR cushandle: Handle);
EXTERNAL; {the routine used to set up the cushandle resource handle}

PROCEDURE CustomCall(text: Ptr; count: INTEGER; findstr: StringPtr;
    selector: INTEGER; appaddr: UNIV Ptr; ourhandle: Handle);
EXTERNAL; {this is the code dispatcher}

{this will do the "about" dialog and }
{the info dialog requested by the custom pack.}

PROCEDURE ShowADialog(meDlog: INTEGER);

CONST
    okButton = 1;
    authorItem = 2;
    languageItem = 3;
    infoItem = 2;

VAR
    itemHit,itemType: INTEGER;
    itemHdl: Handle;
    itemRect: Rect;
    theDialog: DialogPtr;

BEGIN
    theDialog := GetNewDialog(meDlog,NIL,WindowPtr(- 1));
```

```
        CASE meDlog OF
            aboutMeDLOG: BEGIN
                GetDitem(theDialog,authorItem,itemType,itemHdl,itemRect);
                SetIText(itemHdl,'Ming The Vaseless');
                GetDitem(theDialog,languageItem,itemType,itemHdl,itemRect);
                SetIText(itemHdl,'Pascal et al');
            END;

            infoDLOG: BEGIN {display the message requested by the custom
package)
                GetDitem(theDialog,infoItem,itemType,itemHdl,itemRect);
                SetIText(itemHdl,StringPtr(paramPtr)^);
            END;
        END; {CASE}

        REPEAT
            ModalDialog(NIL,itemHit)
        UNTIL (itemHit = okButton);

        CloseDialog(theDialog);
    END; {of ShowADialog}

    {this will put up the Find dialog to allow the user to type in the characters    to
search for}
    FUNCTION DoCustomDialog: BOOLEAN;

    CONST
        okButton = 1;
        cancelButton = 2;
        fixedItem = 3;
        editItem = 4;

    VAR
        itemHit,itemType: INTEGER;
        itemHdl: Handle;
        itemRect: Rect;
        theDialog: DialogPtr;

    BEGIN
        theDialog := GetNewDialog(findDLOG,NIL,WindowPtr(- 1));
        GetDitem(theDialog,editItem,itemType,itemHdl,itemRect);
        SetIText(itemHdl,dlogString);
        TESetSelect(0,MAXINT,DialogPeek(theDialog)^.textH);

        REPEAT
            ModalDialog(NIL,itemHit)
        UNTIL (itemHit IN [okButton,cancelButton]);
        GetIText(itemHdl,dlogString);
        DoCustomDialog := itemHit = okButton;

        CloseDialog(theDialog);
    END; {of DoCustomDialog}

    PROCEDURE DoCommand(mResult: LONGINT);
    .
    .
    .
    (* partial procedure fragment *)

    {here is one of the case sections for the DoCommand procedure}
```

```
findID:
  IF DoCustomDialog THEN
    BEGIN
      MoveHHi(Handle(textH)); {stop it from fragmenting heap}
      WITH textH^^ DO BEGIN
        HLock(hText);           { since we don't know what the }
                                { package might be up to }

        {now call the package to find characters or words}
        CustomCall(POINTER(ORD(hText^) + selEnd),
                    teLength - selEnd, @dlogString, theItem, @appdispatch,
                    custhandle);
        HUnlock(textH^^.hText);
      END; {WITH}
    END;

    END; {OF menu CASE} {to indicate completion of command,}
    HiliteMenu(0); {call Menu Manager to unhighlight }
    {menu title (highlighted by }
    {MenuSelect) }
    END; {OF DoCommand}

    { this is called by the custom package to set the new selection }
    { or display a message; it must be in CODE 1 at the outermost }
    { lexical level}
    PROCEDURE CustDispatch(selector: INTEGER);

BEGIN
  CASE selector OF
    highlightSel:           {hilight the characters selected }
                           {by the custom pack.}

    {paramptr=pointer to text to select, }
    { paramword1&paramword2=start,end chars}

    WITH textH^^ DO
      {we'll subtract the start of text from}
      {paramptr to get the base offset...}
      TESSetSelect(ORD(paramptr) - StripAddress (ORD(hText^)) +
paramword1, ORD(paramptr) - StripAddress (ORD(hText^)) +
paramword2,textH);

      notifySel: {put up message per request from custom pack.}
      {paramptr points to string to display}
      ShowADialog(infoDLOG);

    END; {CASE}
    END; {CustDispatch}

    BEGIN {main program}
    { Initialization }
    InitGraf(@thePort); {initialize QuickDraw}
    InitFonts; {initialize Font Manager}
    FlushEvents(everyEvent - diskMask,0); {call OS Event Mgr to discard
                                           non-disk-inserted events}
    InitWindows; {initialize Window Manager}
    InitMenus; {initialize Menu Manager}
    TEInit; {initialize TextEdit}
    InitDialogs(NIL); {initialize Dialog Manager}
    InitCursor; {call QuickDraw to make cursor (pointer) an arrow}
```

```
InitSignals;
errCode := CatchSignal;
IF errCode <> 0 THEN BEGIN
    Debugger;
    Exit(P);
END;

SetUpMenus; {set up menus and menu bar}
UnloadSeg(@SetUpMenus); {remove the once-only code}

{Custom package initialization stuff}
appdispatch := @CustDispatch;
CustomInit(69,custhandle);           {should test custhandle for NIL
                                     {and alert the user}

dlogString := '';
...
{etc. with the rest of initialization and the main event loop}
END.

; now for the assembly language code
; first, the dispatching and initializing code that must be linked
; into the application

; CustomCalling
; Custom packages initializing and dispatching
;
;   Rick Blair      May, 1987
;
;           PRINT  OFF
;           INCLUDE      'Traps.a'
;           INCLUDE      'ToolEqu.a'
;           INCLUDE      'QuickEqu.a'
;           INCLUDE      'SysEqu.a'
;           PRINT  ON
;
;           LOAD   'most.dmp'      ; from a dump of the files above

appdata      EQU          12

;Initialize a custom module
; Pascal call format:
;  CustomInit(resID:INTEGER;VAR custhandle:Handle);
;
; This will load the CUST module with the given resource ID, install a
; handle to it in custhandle, and set the module's appdata pointer to
; point to the address appaddr.
;
resID      EQU          8
custhandle EQU          4

CustomInit  PROC   EXPORT
    SUBQ.L #4,A7 ;make room for handle from GetResource
    MOVE.L #'CUST',-(A7)
    MOVE.W resID+8(A7),-(A7);resource ID
    _GetResource
    MOVE.L (A7)+,A0
    MOVE.L custhandle(A7),A1
    MOVE.L A0,(A1)      ;store handle in app's custhandle global
; (return with nil handle if GR failed)
    MOVE.L (A7),A0      ;get return address
```



```
ADD.L #10,A7 ;strip everything
JMP    (A0)   ;adieu
```

```
;Call a custom module
;Pascal format:
; CustomCall( {parameters as desired} selector: INTEGER; appaddr: Ptr;
;             module: Handle);
;
;This will call the code whose handle is passed on the stack. If the
;application was written in assembly language you would just
;dereference the handle and call it directly (you wouldn't need this at
;all).
```

```
;
CustomCall    PROC    EXPORT
               IMPORT Signal
               MOVE.L 4(A7),A0      ;get handle
               MOVE.L (A0),D0
               BNE.S @0             ;if hasna' been purged, ga' ahead
               MOVE.L A0,-(A7)      ;push handle
               _LoadResource
               MOVE.W ResErr,-(A7)
               JSR    Signal ;Signal is a NOP if a zero is passed to it
               MOVE.L 4(A7),A0      ;handle again
; we don't lock the handle here (we can't save it so we can unlock it
; later), so it's up to the package to lock/unlock itself
@0            MOVE.L (A0),A0        ;dereference
               JMP    (A0)         ;call CUST code

               END
```

```
; here is the module for the custom package itself
```

```
; CustomPack
; Example custom code package
;
; Rick Blair      May, 1987
;
; This demonstrates the recommend structure of a code module which a
; sophisticated user could add to an existing application which supported
; this mechanism. Aside from allowing for multiple routines within the
; module (via a selector), provision is made for calling a routine
; dispatcher within the application itself.
```

```
;Finding text
;We support a call to find a string anywhere within a block of text
; (selector=0), and one to find the string only as a separate "word"
; with spaces around it (selector=1).
;PROCEDURE CustomCall(text:Ptr; count:INTEGER; findstr:^STRING;
;                     selector:INTEGER; appaddr: UNIV Ptr; ourhandle:Handle);
;
;Rather than return a result indicating whether they succeeded or not,
;these routines take whatever action is appropriate (the application
;may not even know what these routines actually do).
;Once a call succeeds or fails, it then takes action by making a call to
;one of the services provided by the application. In this case the two
;functions provided are just what we need; the ability to select text and
;the ability to put up a message saying "Text not found".
```

```
STRING ASIS
```

```
; PRINT OFF
; INCLUDE    'Traps.a'
; INCLUDE    'ToolEqu.a'
```

```
;          INCLUDE      'QuickEqu.a'
;          INCLUDE 'SysEqu.a'
;          PRINT  ON

          LOAD   'most.dmp'    ; from a dump of the files above

CustPack          PROC   EXPORT

          BRA.S  Entry  ;skip header

          DC.W   0           ;flags
          DC.B   'CUST' ;custom add-on code module
          DC.W   69          ;resource ID (picked by Mr. Peabody &
Sherman)          ;
          DC.W   $10         ;version 1.0

StackFrame  RECORD {A6Link},DECR
paramsize   EQU    *-8
;          call-specific parameters... (optional)
text        DS.L   1        ;pointer to text block
count       DS.W   1        ;word count of characters in text
findstr     DS.L   1        ;pointer to p-string to find
;          selector(word, optional - you might only have 1 call)
selector    DS.W   1
fcharsCmd   EQU    1        ; selector for "find characters"
fwordCmd    EQU    2        ; selector for "find word"
;          pointer to app. globals      (long)
appaddr     DS.L   1
;          handle to this resource      (long)
ourhandle   DS.L   1
;          TOS:return address (long)
return      DS.L   1
;the stack link is built off the origin of the saved old A6 on the stack
A6Link      DS.L   1
LocalSize   EQU    *
            ENDR

;offsets into our application globals area
AppGlobals  RECORD {appdispatch},DECR
appdispatch DS.L   1
paramptr    DS.L   1
paramword1  DS.W   1
paramword2  DS.W   1
;CUSTerr    DS.W   1        ;if we had possible errors
            ENDR

Entry

          WITH   StackFrame,AppGlobals
          LINK   A6,#LocalSize
;          MOVEM.L      ...      ;we'd save any non-trashable regs here
;first lock us down...
          MOVE.L  ourhandle(A6),A0
          _HLock

          MOVE.W  selector(A6),D0
          CMP.W   #fcharsCmd,D0
          BEQ.S   charfind      ;go find characters
          CMP.W   #fwordCmd,D0
          BEQ.S   wordfind      ;go find a word
;well, M. App didn't call us with a selector we know, so...
```

```
;unlock ourselves, clean up, return
; (if we wanted to return an error code we could stuff it into the app.
; global area)
duhn      MOVE.L ourhandle(A6),A0
          _HUnLock
;          MOVEM.L      ...      ;restore any registers here
          UNLK  A6
          MOVE.L (A7)+,A0      ;return address
          ADD.L #paramsize,A7;strip parameters
          JMP   (A0)

;selector codes for calls to application
hilight   EQU    0      ;highlight characters, please
notify    EQU    1      ;beep a little

;find the string "findstr" anywhere in the block "text"
charfind
          JSR     findchars      ;see if findstr is anywhere in text
          BEQ.S   nofind ;if not then skip
          JSR     calcsels       ;compute selstart and selend
didfind   MOVE.L appaddr(A6),A0  ;get pointer to appl. globals area
          MOVE.L text(A6),paramptr(A0) ;setup text pointer and...
          MOVE.W D0,paramword1(A0) ;start character position,
          MOVE.W D1,paramword2(A0) ;end character position
          MOVE.W #hilight,-(A7)  ;pass proper selector
goapp     MOVE.L appdispatch(A0),A0 ;get dispatch address
          JSR     (A0)           ;call the application to select the range
          BRA.S   duhn           ;return to application (déjà vu)

nofind    MOVE.L appaddr(A6),A0  ;get pointer to appl. globals area
          LEA     oopstring,A1   ;get pointer to "Not found" message
          MOVE.L A1,paramptr(A0) ;put string pointer in "paramptr"
          MOVE.W #notify,-(A7)   ;tell app. to display message
          BRA.S   goapp

;figure selstart and selend
calcsels  NEG.W   D0      ; negate # characters unskipped in text
          SUBQ.W #1,D0    ;include 1st character
          ADD.W   count(A6),D0 ;compute 1st character position for
          ; select
          MOVE.L findstr(A6),A1
          MOVE.B (A1),D1   ;get length of string
          EXT.W   D1
          ADD.W   D0,D1    ;compute last char. pos. for select
          RTS

;find the characters, but only if surrounded by space (including end or
; beg.)
;we could extend the test to check for other delimiters (";",",",etc.)
wordfind
          JSR     findchars
wloop     BEQ.S   nofind
          MOVE.W D0,D2      ;save count of text remaining
          JSR     calcsels  ;figure start and end offsets
          MOVE.L text(A6),A1 ;point to text
          TST.W   D0        ;start=beginning of text?
          BEQ.S   @0        ;yep, so it passes
          CMP.B   #' ',-1(A1,D0) ;preceded by a space?
          BNE.S   @1        ;nope, keep looking
```

```
@0      CMP.W  count(A6),D1      ;D1=length of text?
      BEQ.S  didfind            ;yep, so it passes
      CMP.B  #' ',(A1,D1)       ;followed by a space?
      BEQ.S  didfind            ;yes, so we've found it

;this wasn't paydirt, so keep panning
@1      MOVE.W D2,D0            ;restore chars remaining count
      BMI.S  nofind            ;forget it if we ran out of text
      JSR    bigloop           ;keep looking
      BRA.S  wloop

;this code will find the string if it lies anywhere in the text
findchars  MOVE.L text(A6),A0      ;point A0 to chars to search
      MOVE.W count(A6),D0          ;size of text block
bigloop    MOVE.L findstr(A6),A1;point A1 to chars to find
      MOVE.W (A1)+,D1             ;get length byte and 1st char. (skip 'em)
      CMP.W  #255,D1
      BGT.S  @1                  ;enter loop if length<>0
      ADDQ.L #4,A7                ;strip findchar's return address
      BRA    duhn                ;return having done nothing

;search for first character
@0      CMP.B  (A0)+,D1           ;this one match 1st character?
@1      DBEQ   D0,@0             ;branch until found or done 'em all
      BNE.S  cnofind            ;skip out if no match on 1st character

      MOVE.B -2(A1),D1           ;length of findstr
      EXT.W  D1
      SUBQ.W #1,D1               ;length sans 1st character
      BEQ.S  cfound             ;if Length(findstr)=1, we're done
      CMP.W  D1,D0
      BLT.S  cnofind            ;fail if findstr is longer than text left
      MOVE.L A0,D2               ;save this character position
      CMP.W  D1,D1               ;force Equality
      BRA.S  @3                 ;enter loop

@2      CMP.B  (A0)+,(A1)+       ;match so far?
@3      DBNE   D1,@2             ;check until mismatch or end of findstr

      MOVEA.L      D2,A0         ;restore position (cc's unaffected)
      BNE.S  bigloop           ;if no match then keep looking

cfound    MOVEQ   #1,D1 ;return TRUE
      RTS

cnofind    SUB.W   D1,D1 ;return FALSE
      RTS

      STRING PASCAL
oopstring  DC.B    'Pattern not found.'

      END
```

#additions to the resource file

```
resource 'DLOG' (129, "Find dialog") {
    {72, 64, 164, 428},
    dBoxProc,
    visible,
    noGoAway,
    0x0,
```

```
        129,  
        "Find"  
};  
  
resource 'DLOG' (130, "Info") {  
    {66, 102, 224, 400},  
    dboxproc, visible, nogoaway, 0x0, 130, ""  
};  
  
resource 'DITL' (130) {  
    {  
/* 1 */ {130, 205, 150, 284},  
        button {  
            enabled,  
            "OK already"  
        };  
/* 2 */ {8, 32, 120, 296},  
        statictext {  
            disabled,  
            ""  
        }  
    }  
};  
  
resource 'DITL' (129) {  
    {  
/* array DITLarray: 4 elements */  
/* [1] */  
    {64, 48, 84, 121},  
    Button {  
        enabled,  
        "OK"  
    };  
/* [2] */  
    {64, 231, 84, 304},  
    Button {  
        enabled,  
        "Cancel"  
    };  
/* [3] */  
    {8, 8, 24, 352},  
    StaticText {  
        disabled,  
        "Find what?"  
    };  
/* [4] */  
    {32, 8, 48, 352},  
    EditText {  
        disabled,  
        ""  
    }  
    }  
};  
  
resource 'MENU' (131, "Custom", preload) {  
    131, textMenuProc, 0x3, enabled, "Custom",  
    {  
        "Find Chars...",  
        noicon, "F", nomark, plain;  
        "Find Word...",  
        noicon, "W", nomark, plain  
    }  
};
```

```
type 'CTST' as 'STR ' ;

resource 'CTST' (0) {
    "Custom Application - Version 1.0"
};

include "CustomPack.code";

# This makefile puts the program together incl. the CUST pack.

CustomTest      ff  CustomCalling.a.o CustomTest.p.o ErrSignal.a.o
# the predefined rule for assembly will build CustomCalling.a.o,
# CustomPack.code
    Link CustomTest.p.o CustomCalling.a.o ErrSignal.a.o 0
        "{Libraries}"Interface.o 0
        "{Libraries}"Runtime.o 0
        "{PLibraries}"Paslib.o 0
        -o CustomTest
CustomPack.code  f      CustomPack.a.o
    Link CustomPack.a.o -rt CUST=69 -o CustomPack.code
# Put the resource file together (including the custom code resource)
CustomTest      ff      CustomTest.r CustomPack.code
    Rez CustomTest.r -a -o CustomTest
```

---

**Further Reference:**

- M.PT.Signals