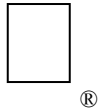


Macintosh Technical Notes



Developer Technical Support

Signals

Platforms & Tools

Revised by:
Written by: Rick Blair

March 1988
August 1986

Signals are a form of intra-program interrupt which can greatly aid clean, inexpensive error trapping in stack frame intensive languages. A program may invoke the `Signal` procedure and immediately return to the last invocation of `CatchSignal`, including the complete stack frame state at that point.

Signals allow a program to leave off execution at one point and return control to a convenient error trap location, regardless of how many levels of procedure nesting are in between.

The example is provided with a Pascal interface, but it is easily adapted to other languages. The only qualification is that the language **must** bracket its procedures (or functions) with `LINK` and `UNLK` instructions. This will allow the signal code to clean up at procedure exit time by removing `CatchSignal` entries from its internal queue.

Note: only procedures and/or functions that call `CatchSignal` need to be bracketed with `LINK` and `UNLK` instructions.

Important: `InitSignals` must be called from the main program so that A6 can be set up properly.

Note that there is no limit to the number of local `CatchSignals` which may occur within a single routine. Only the last one executed will apply, of course, unless you call `FreeSignal`. `FreeSignal` will “pop” off the last `CatchSignal`. If you attempt to `Signal` with no `CatchSignals` pending, `Signal` will halt the program with a debugger trap.

`InitSignals` creates a small relocatable block in the application heap to hold the signal queue. If `CatchSignal` is unable to expand this block (which it does 5 elements at a time), then it will signal back to the last successful `CatchSignal` with code = 200. A `Signal(0)` acts as a NOP, so you may pass `OSErrs`, for instance, after making File System type calls, and, if the `OSErr` is equal to `NoErr`, nothing will happen.

CatchSignal may not be used in an expression if the stack is used to evaluate that expression. For example, you can't write:

```
c:= 3*CatchSignal;
```

“Gotcha” summary

1. Routines which call `CatchSignal` must have stack frames.
2. `InitSignals` must be called from the outermost (main) level.
3. Don't put the `CatchSignal` function in an expression. Assign the result to an `INTEGER` variable; i.e. `i:=CatchSignal`.
4. It's safest to call a procedure to do the processing after `CatchSignal` returns. See the Pascal example `TestSignals` below. This will prevent the use of a variable which may be held in a register.

Below are three separate source files. First is the Pascal interface to the signaling unit, then the assembly language which implements it in MPW Assembler format. Finally, there is an example program which demonstrates the use of the routines in the unit.

```
{File ErrSignal.p}
UNIT ErrSignal;

INTERFACE

{Call this right after your other initializations (InitGraf, etc.)--in other words as early
as you can in the application}
PROCEDURE InitSignals;

{Until the procedure which encloses this call returns, it will catch subsequent Signal
calls, returning the code passed to Signal. When CatchSignal is encountered initially, it
returns a code of zero. These calls may "nest"; i.e. you may have multiple CatchSignals in
one procedure.
Each nested CatchSignal call uses 12 bytes of heap space }
FUNCTION CatchSignal:INTEGER;

{This undoes the effect of the last CatchSignal. A Signal will then invoke the CatchSignal
prior to the last one.}
PROCEDURE FreeSignal;

{Returns control to the point of the last CatchSignal. The program will then behave as
though that CatchSignal had returned with the code parameter supplied to Signal.}
PROCEDURE Signal(code:INTEGER);

END.
{End of ErrSignal.p}
```

Here's the assembly source for the routines themselves:

```
; ErrSignal code w. InitSignal, CatchSignal, FreeSignal, Signal
; defined
;
;               Version 1.0 by Rick Blair

        PRINT OFF
        INCLUDE      'Traps.a'
        INCLUDE      'ToolEqu.a'
        INCLUDE      'QuickEqu.a'
        INCLUDE      'SysEqu.a'
        PRINT ON

CatchSigErr EQU 200           ;"insufficient heap" message
SigChunks   EQU 5            ;number of elements to expand by
FrameRet     EQU 4           ;return addr. for frame (off A6)
SigBigA6     EQU $FFFFFFF     ;maximum positive A6 value
```

```
; A template in MPW Assembler describes the layout of a collection of data
; without actually allocating any memory space. A template definition starts
; with a RECORD directive and ends with an ENDR directive.
```

```
; To illustrate how the template type feature works, the following template
; is declared and used. By using this, the assembler source approximates very
; closely Pascal source for referencing the corresponding information.
```

```
;template for our table elements
```

```
SigElement   RECORD 0           ;the zero is the template origin
SigSP DS.L   1                 ;the SP at the CatchSignal-(DS.L just like EQU)
SigRetAddr   DS.L   1           ;the address where the CatchSignal returned
SigFRet      DS.L   1           ;return addr. for encl. procedure
SigElSize    EQU    *           ;just like EQU 12
                ENDR
```

```
; The global data used by these routines follows. It is in the form of a
; RECORD, but, unlike above, no origin is specified, which means that memory
; space *will* be allocated.
; This data is referenced through a WITH statement at the beginning of the
; procs that need to get at this data. Since the Assembler knows when it is
; referencing data in a data module (since they must be declared before they
; are accessed), and since such data can only be accessed based on A5, there
; is no need to explicitly specify A5 in any code which references the data
; (unless indexing is used). Thus, in this program we have omitted all A5
; references when referencing the data.
```

```
SigGlobals RECORD                ;no origin means this is a data record
                                ;not a template(as above)
SigEnd DS.L   1                 ;current end of table
SigNow DS.L   1                 ;the MRU element
SigHandle    DC.L   0           ;handle to the table
                ENDR
InitSignals PROC      EXPORT ;PROCEDURE InitSignals;
                    IMPORT CatchSignal
                WITH   SigElement,SigGlobals
```

```
;the above statement makes the template SigElement and the global data
;record SigGlobals available to this procedure
```

```
    MOVE.L #SigChunks*SigElSize,D0
    _NewHandle    ;try to get a table
    BNE.S  forgetit                ;we couldn't get that!?

    MOVE.L A0,SigHandle            ;save it
    MOVE.L #-SigElSize,SigNow      ;point "now" before start
    MOVE.L #SigChunks*SigElSize,SigEnd ;save the end
    MOVE.L #SigBigA6,A6            ;make A6 valid for Signal
forgetit    RTS
                ENDP
```

```
CatchSignal PROC      EXPORT ;FUNCTION CatchSignal:INTEGER;    IMPORT
    SiggysSetup,Signal,SigDeath
    WITH   SigElement,SigGlobals
    MOVE.L (SP)+,A1                ;grab return address
    MOVE.L SigHandle,D0            ;handle to table
    BEQ    SigDeath                ;if NIL then croak
    MOVE.L D0,A0                   ;put handle in A-register
    MOVE.L SigNow,D0
    ADD.L  #SigElSize,D0
    MOVE.L D0,SigNow                ;save new position
```

```
        CMP.L  SigEnd,D0                ;have we reached the end?
        BNE.S  catchit                 ;no, proceed
        ADD.L  #SigChunks*SigElSize,D0 ;we'll try to expand
        MOVE.L D0,SigEnd               save new (potential) end
        _SetHandleSize
        BEQ.S  @0                      ;jump around if it worked!

;signals, we use 'em ourselves
        MOVE.L SigNow,SigEnd           ;restore old ending offset
        MOVE.L #SigElSize,D0
        SUB.L  D0,SigNow               ;ditto for current position
        MOVE.W #catchSigErr,(SP)       ;we'll signal a "couldn't
        ;                               catch" error
        JSR    Signal ;never returns of course

@0      MOVE.L SigNow,D0

catchit  MOVE.L (A0),A0                ;deref.
        ADD.L  D0,A0                  ;point to new entry
        MOVE.L SP,SigSP(A0)          ;save SP in entry
        MOVE.L A1,SigRetAddr(A0)     ;save return address there
        CMP.L  #SigBigA6,A6          ;are we at the outer level?
        BEQ.S  @0                    ;yes, no frame or cleanup needed
        MOVE.L FrameRet(A6),SigFRet(A0);save old frame return
        ;                               address
        LEA    Siggypop,A0
        MOVE.L A0,FrameRet(A6)        ;set cleanup code address
@0      CLR.W  (SP)                   ;no error code (before its time)
        JMP    (A1)                  ;done setting the trap

Siggypop JSR    SiggypopSetup          ;get pointer to element
        MOVE.L SigFRet(A0),A0         ;get proc's real return address
        SUB.L  #SigElSize,D0
        MOVE.L D0,SigNow              ;"pop" the entry
        JMP    (A0) ;gone
        ENDP

FreeSignal PROC EXPORT ;PROCEDURE FreeSignal;
        IMPORT SiggypopSetup
        WITH   SigElement,SigGlobals
        JSR    SiggypopSetup          ;get pointer to current entry
        MOVE.L SigFRet(A0),FrameRet(A6);"pop" cleanup code
        SUB.L  #SigElSize,D0
        MOVE.L D0,SigNow              ;"pop" the entry
        RTS
        ENDP

Signal PROC EXPORT ;PROCEDURE Signal(code:INTEGER);
        EXPORT SiggypopSetup,SigDeath
        WITH   SigElement,SigGlobals
        MOVE.W 4(SP),D1               ;get code
        BNE.S  @0                    ;process the signal if code is non-zero
        MOVE.L (SP),A0                ;save return address
        ADDQ.L #6,SP                  ;adjust stack pointer
        JMP    (A0)                  ;return to caller(code was 0)

@0      JSR    SiggypopSetup          ;get pointer to entry
        BRA.S  SigLoop1

SigLoop  UNLK  A6                    ;unlink stack by one frame
SigLoop1 CMP.L SigSP(A0),A6           ;is A6 beyond the saved stack?
        BLO.S  SigLoop              ;yes, keep unlinking
```

```
        MOVE.L SigSP(A0),SP      ;bring back our SP
        MOVE.L SigRetAddr(A0),A0 ;get return address
        MOVE.W D1,(SP)          ;return code to CatchSignal
        JMP      (A0)            ;Houston, boost the Signal!
                                ;(or Hooston if you're from the Negative Zone)

SiggySetup    MOVE.L SigHandle,A0
              MOVE.L (A0),A0      ;deref.
              MOVE.L A0,D0        ;to set CCR
              BEQ.S SigDeath      ;nil handle means trouble
              MOVE.L SigNow,D0    ;grab table offset to entry
              BMI.S SigDeath      ;if no entries then give up
              ADD.L D0,A0         ;point to current element
              RTS

SigDeath      _Debugger          ;a signal sans catch is bad news

              ENDP
              END
```

Now for the example Pascal program:

```
PROGRAM TestSignals;
USES ErrSignal;

VAR i:INTEGER;

PROCEDURE DoCatch(s:STR255; code:INTEGER);
BEGIN
  IF code<>0 THEN BEGIN
    Writeln(s,code);
    Exit(TestSignals);
  END;
END; {DoCatch}

PROCEDURE Easy;
PROCEDURE Never;
  PROCEDURE DoCatch(s:STR255; code:INTEGER);
  BEGIN
    IF code<>0 THEN BEGIN
      Writeln(s,code);
      Exit(Never);
    END;
  END; {DoCatch}

  BEGIN {Never}
  i:=CatchSignal;
  DoCatch('Signal caught from Never, code = ', i );

  i:=CatchSignal;
  IF i<>0 THEN DoCatch('Should never get here!',i);

  FreeSignal; {"free" the last CatchSignal}
  Signal(7); {Signal a 7 to the last CatchSignal}
  END;{Never}
BEGIN {Easy}
Never;
Signal(69); {this won't be caught in Never}
END;{Easy} {all local CatchSignals are freed when a procedure exits.}

BEGIN {PROGRAM}
InitSignals; {You must call this early on!}
```

```
{catch Signals not otherwise caught by the program}
i:=CatchSignal;
IF i<>0 THEN
DoCatch('Signal caught from main, code = ',i);

Easy;
END.
```

The example program produces the following two lines of output:

```
Signal caught from Never, code = 7
Signal caught from main, code = 69
```

Further Reference:

- Using Assembly Language (Mixing Pascal & Assembly)