Technical Notes

Run-Time Error Handling in FORTH Using SETJMP and LNGJMP for Execution Control or GOTO in FORTH

Robert J. Paul, Jay S. Friedland, Jeremy E. Sagan
Turning Point Software, Inc.
11A Main Street
Watertown, MA 02172

Abstract

This paper discusses and provides two FORTH implementations of the SETJMP/LNGJMP concept for error handling in applications which have words nested many layers deep. SETJMP and LNGJMP may be familiar to programmers who have worked with UNIX/C environments. A SETJMP is paired with a LNGJMP, which when executed later, will return control to the code following the SETJMP, passing an appropriate value on the stack. In essence, this creates the equivalent of the GOTO statement in FORTH. In most FORTH applications a word at each level returns an error or success code which is processed before continuing. The use of SETJMP/LNGJMP eliminates the nesting of IF ... ELSE ... THEN or BEGIN ... UNTIL statements by transferring control to a level where all error conditions may be handled. SETJMP and LNGJMP may be nested to handle local as well as global situations. These constructs reduce code overhead and produce code which is easier to follow. SETJMP and LNGJMP are ideal programming constructs for error handling since they allow control flow to be optimized.

Introduction

There are certain programming tasks, notably error handling, that are most conveniently processed by a branch to an error handler, which resumes execution in some other part of the program. In non-structured languages, this is usually accomplished via a GOTO instruction. In a block-structured language like FORTH, a programmer can find himself several subroutine levels deep with an error that should be returned to the highest level of control. A typical solution would be for each word to return an error/success code, which must be checked before proceeding. However, the resulting IF ... ELSE ... THEN constructs often make the code more difficult to follow and can consume a lot of memory. The ideal solution would be to exit several subroutine levels, back to the main control level, where all errors could be handled.

History

The idea for SETJMP/LNGJMP comes from the language C as implemented under Unix. It is a somewhat late addition to that language, appearing after Kernighan and Ritchie wrote *The C Programming Language*. It is documented in the *Unix Programmer's Manual*, Berkeley 4.2 release, August 1983.

Implementation

The idea is quite simple. A word SETJMP simply marks a place in the FORTH program. Subsequent LNGJMPs will transfer control to (i.e. goto) the place marked by the SETJMP.

SETJMP (-- rc) saves the current execution environment, the return stack and parameter stack pointers, and returns the value 0 on the parameter stack.
LNGJMP (rc --) restores the (most recent) environment, the return stack and parameter stack pointers saved by SETJMP, and leaves the return code on the parameter stack.

The effect of a 0 LNGJMP is to continue execution as if the last SETJMP had just executed. Normally, a non-zero value (e.g. error code) is used to indicate why the jump was made.

Cautions

A facility this powerful must also carry with it some dangers. In order to work correctly, a SETJMP must be executed before any LNGJMPs, otherwise the results are undefined. Less obvious is the requirement that the return stack must be intact below the level where SETJMP was called (i.e. the SETJMP must be nested no deeper than the LNGJMP call). If the stacked version is used, exactly one UNSETJMP must be executed for each SETJMP; any number of LNGJMPs may be used.

Code Definitions

Two versions of the routines are provided. Figure 1 gives the implementation (on an IBM PC) of a version that supports only one active SETJMP. Each subsequent SETJMP forgets the previous location and installs the current location. This version is simpler than the second, and will suffice for most purposes. In this version, the word CLRJMP may be executed to initialize the SETJMP location so that control returns to the interpreter.

The second version allows for nested SETJMPs, up to 12 levels; an implementation is shown in Figure 2. This version requires that exactly one UNSETJMP be executed for each SETJMP executed. This version is more flexible, but also requires more careful coding.

Figure 3 provides test code for the stack version. A large variety of execution paths can be taken, depending on what sequence of keys is pressed. The sequence <Esc><Esc><?><Space><Esc><?><Esc> will get you out without executing any LNGJMPs. If a different key is pressed at any point, the calling routine will restart.

Machine independence is very simple. Three words must be coded: RP@, !SP, !RP. RP@ (-- rp) returns the return stack pointer on the parameter stack. !SP (sp --) changes the parameter stack pointer to the value sp, and !RP (rp --) changes the return stack pointer to the value rp.

Applications

Time is Moneypersonal, a personal/small business accounting package published by Turning Point Software, includes a master diskette program which was written in FORTH. It uses the one-level version of SETJMP and LNGJMP. At the highest control level is a menu handler which contains the sequence BEGIN SETJMP DROP ... AGAIN to ensure that control always stays at the menu handler. Each major function is implemented as an overlay, each of which begins with SETJMP -DUP IF /ERROR/ ELSE ... THEN, where /ERROR/ is a handler which prints a message based on the error number. Number 27 is used for <Esc> key processing, 26 for out of memory errors, and all others for disk I/O errors.

Figure 1. One Subroutine Level Version

```
( SETJMP/LNGJMP CODE - ONE LEVEL VERSION )
( 8088 Version for IBM PC )
CODE RP@ BP PUSH, NXT C;
CODE !RP BP POP,  NXT C;
CODE !SP AX POP, AX SP MOV, NXT C;

VARIABLE RPSAVE      VARIABLE SPSAVE      VARIABLE IPSAVE
: SETJMP I IPSAVE ! RP@ RPSAVE ! SP@ SPSAVE ! Ø ;
: LNGJMP >R SPSAVE @ !SP R> RPSAVE @ !RP R> DROP
     IPSAVE @ >R ;
: CLRJMP SETJMP -DUP IF ." LngJmp code " . QUIT THEN ;
EXIT

( 6052 Version for Apple )
CODE !RP XSAVE STX, BOT LDA, TAX, TXS, XSAVE LDX, POP JMP, C;
CODE !SP BOT LDA,  TAX, NEXT JMP, C;
CODE RP@ XSAVE STX, TSX, TXA, XSAVE LDX, PUSHØA JMP, C;
```

Figure 2. Stacked Version

```
( SETJMP/LNGJMP CODE - STACKED VERSION )
CODE RP@ BP PUSH, NXT C;
CODE !RP BP POP, NXT C;
CODE !SP AX POP, AX SP MOV, NXT C;

VARIABLE SLEVEL -2 SLEVEL !          ( CURRENT STACK LEVEL )
CREATE RPSAVE 24 ALLOT               ( 12 SUBROUTINE LEVELS )
CREATE SPSAVE 24 ALLOT           CREATE IPSAVE 24 ALLOT

: SETJMP 2 SLEVEL +!    I IPSAVE SLEVEL @ + !
     RP@ RPSAVE SLEVEL @ + !    SP@ SPSAVE SLEVEL @ + !  Ø ;
: LNGJMP >R SPSAVE SLEVEL @ + @ !SP R> RPSAVE SLEVEL @ + @ !RP
     R> DROP    IPSAVE SLEVEL @ + @ >R ;
: UNSETJMP -2 SLEVEL +! ;
```

Figure 3. Test Code

```
(STACKED SETJMP/LNGJMP TEST )
: W4 CR ." Called W4 from W" DUP . ." - hit <Esc> "
     KEY 27 = Ø= IF 4 LNGJMP THEN ;

: W3 3 SETJMP -DUP IF ." @3 rc=" .
     THEN W4 CR ." Called W3 from W" OVER . ." - hit <?> "
     KEY 63 = UNSETJMP Ø= IF 3 LNGJMP THEN DROP ;

: W2 2 SETJMP -DUP IF ." @2 rc=" . THEN W4 W3 CR
     ." Called W2 from W" OVER .
     ." - hit <Space> " KEY 32 = UNSETJMP 0=
     IF 2 LNGJMP THEN DROP ;

: W1 SETJMP -DUP IF ." @1 rc=" . THEN W2 W3 W4 ;
```