```
void exec_op()
{
#ifdef DEBUG
        printf("exec opcode\n");
#endif
        wa = pop_ps() ;
}

/*
 * Procedure to generate a trap so that the inner interpreter will trip
 * back into the outer interpreter at the end of this execution cycle.
 */

void trap_op()
{
#ifdef DEBUG
        printf("trap opcode\n");
#endif
        set_trap() ;
}

/*
 * Procedure to set the exit condition so that the outer interpreter will
 * terminate at the end of the inner interpreter execution cycle.
 */

void exit_op()
{
#ifdef DEBUG
        printf("exit opcode\n");
#endif
        set_exit() ;
}

/*
 * Procedure to generate a trap, reset the parameter stack pointer and
 * set the STATE user variable to false, so that the inner interpreter
 * will trip back into the outer interpreter at the end of this execution
 * cycle.
 */

void abort_op()
{
#ifdef DEBUG
        printf("abort opcode\n");
#endif
        forth_abort() ;
}
```

```
/*
 * Module : Stackops.h
 * Author : Steven James
 * Date   : 20th January 1990
 *
 * This module implements the seven main stack operations required by
 * the forth engine, namely DROP DUP SWAP OVER ROT EMPTY for the parameter
 * stack, and >R R> R J for the return stack.
 *
 * Functions : -
 *      drop_op() dup_op() swap_op() over_op() rot_op()
 *      pushr_op() popr_op() r_op() j_op()
 *      empty_op()
 */

#define DROP_OP         100
#define DUP_OP          101
#define SWAP_OP         102
#define OVER_OP         103
#define ROT_OP          104
#define PUSHR_OP        105
#define POPR_OP         106
#define R_OP            107
#define J_OP            108
#define EMPTY_OP        109

/*
 * Procedure to disguard the top item on the parameter stack.
 */

void drop_op()
{
#ifdef DEBUG
        printf("drop opcode\n") ;
#endif
        pop_ps() ;
}

/*
 * Procedure to duplicate the top item on the parameter stack.
 */

void dup_op()
{
        unsigned short top ;
#ifdef DEBUG
        printf("dup opcode\n") ;
#endif
        top = pop_ps() ;
        push_ps( top ) ;
        push_ps( top ) ;
}

/*
 * Procedure to swap the top two items on the parameter satck.
 */

void swap_op()
{
        unsigned short top , bottom ;
#ifdef DEBUG
        printf("swap opcode\n") ;
#endif
        top = pop_ps() ;
        bottom = pop_ps() ;
        push_ps( top ) ;
        push_ps( bottom ) ;
}

/*
 * Procedure to copy the second item on the parameter stack to the top.
 */

void over_op()
{
        unsigned short top , bottom ;
#ifdef DEBUG
        printf("over opcode\n") ;
#endif
```

```
        top = pop_ps() ;
        bottom = pop_ps() ;
        push_ps( bottom ) ;
        push_ps( top ) ;
        push_ps( bottom ) ;
}

/*
 * Procedure to move the third item on the parameter satck to the top.
 */

void rot_op()
{
        unsigned short top , middle , bottom ;
#ifdef DEBUG
        printf("rot opcode\n") ;
#endif
        top = pop_ps() ;
        middle = pop_ps() ;
        bottom = pop_ps() ;
        push_ps( middle ) ;
        push_ps( top ) ;
        push_ps( bottom ) ;
}

/*
 * Procedure to push the top item fron the parameter to the return stack.
 */

void pushr_op()
{
#ifdef DEBUG
        printf(">R opcode\n") ;
#endif
        push_rs( pop_ps() ) ;
}

/*
 * Procedure to push the top item from the return to the parameter stack.
 */

void popr_op()
{
#ifdef DEBUG
        printf("R> opcode\n") ;
#endif
        push_ps( pop_rs() ) ;
}

/*
 * Procedure to push the top element on the return stack onto the top
 * of the parameter stack as a 16 bit quantity, without disturbing the
 * top of the return stack.
 */

void r_op()
{
        unsigned short r ;
#ifdef DEBUG
        printf("R opcode\n") ;
#endif
        r = pop_rs() ;
        push_rs( r ) ;
        push_ps( r ) ;
}

/*
 * Procedure to push the third element on the return stack onto the top
 * of the parameter stack as a 16 bit quantity, without disturbing the
 * third element on the return stack.
 */

void j_op()
{
        unsigned short third ;
#ifdef DEBUG
        printf("J opcode\n") ;
#endif
        push_ps( pop_rs() ) ;
```

```
        push_ps( pop_rs() ) ;
        third = pop_rs() ;
        push_rs( third ) ;
        push_rs( pop_ps() ) ;
        push_rs( pop_ps() ) ;
        push_ps( third ) ;
}

/*
 * Procedure to check if the parameter stack is empty, and return
 * a 16 bit value to reflect the result.
 */

void empty_op()
{
#ifdef DEBUG
        printf("empty opcode\n") ;
#endif
        if( empty_ps() )
                push_ps( FTRUE ) ;
        else
                push_ps( FFALSE ) ;
}
```

```
/*
 * Module : Memoryops.h
 * Author : Steven James
 * Date   : 23rd January 1990
 *
 * This module implements the six main memory interfacing operations
 * required by the forth engine, namely ! (fetch) @ (store) ,  (comma)
 * for both 8 and 16 quantities
 *
 * Functions :-
 *      store_op() fetch_op() cstore_op() cfetch_op()
 */

#define STORE_OP        200
#define FETCH_OP        201
#define CSTORE_OP       202
#define CFETCH_OP       203

/*
 * Procedure to store a 16 bit quantity from the second position on the
 * parameter stack to the address on the top of the parameter stack.
 */

void store_op()
{
        unsigned short addr,value ;
#ifdef DEBUG
        printf("store opcode\n");
#endif
        addr = pop_ps() ;
        value = pop_ps() ;
        put_word( addr , value );
}

/*
 * Procedure to fetch the contents of the address on the top of the
 * parameter stack, and push it  as a 16 bit quantity onto the parameter
 * stack.
 */

void fetch_op()
{
        unsigned short addr ;
#ifdef DEBUG
        printf("fetch opcode\n");
#endif
        addr = pop_ps() ;
        push_ps( get_word( addr ) ) ;
}

/*
 * Procedure to store an 8 bit quantity from the second position on the
 * parameter stack to the address on the top of the parameter stack.
 */

void cstore_op()
{
        unsigned short addr,value ;
#ifdef DEBUG
        printf("cstore opcode\n");
#endif
        addr = pop_ps() ;
        value = pop_ps() ;
        put_byte( addr,value ) ;
}

/*
 * Procedure to fetch the contents of the address on the top of the
 * parameter stack, and push it as an 8 bit quantity onto the parameter
 * stack.
 */

void cfetch_op()
{
        unsigned short addr ;
#ifdef DEBUG
        printf("cfetch opcode\n") ;
#endif
        addr = pop_ps() ;
```

```
            push_ps( get_byte( addr ) ) ;
        }
```

```
/*
 * Module : Mathsops.h
 * Author : Steven James
 * Date   : 24th January 1990
 *
 * This module implements the seven main arithmetic operations required by
 * the forth engine, namely + - * / U* U/ MINUS.
 *
 * Functions :-
 *      add_op() daddop_() umul_op() udiv_op() minus_op() dminus_op(
 */

#define ADD_OP          300
#define DADD_OP         301
#define UMUL_OP         302
#define UDIV_OP         303
#define MINUS_OP        304
#define DMINUS_OP       305

/*
 * Procedure to add the top two 16 bit quantities on the parameter stack
 * together producing a 16 bit result.
 */

void add_op()
{
        unsigned short top , bottom ;
#ifdef DEBUG
        printf("add opcode\n");
#endif
        top = pop_ps() ;
        bottom = pop_ps() ;
        push_ps( bottom + top ) ;
}

/*
 * Procedure to add the top two 32 bit quantities on the parameter stack
 * together producing a 32 bit result with the most significant word
 * first, and the least significat word second.
 */

void dadd_op()
{
        unsigned int lsword , msword , top , bottom ;
#ifdef DEBUG
        printf("dadd opcode\n" );
#endif
        msword = ( int ) pop_ps() ;
        lsword = ( int ) pop_ps() ;
        top = ( msword << 16  ) + lsword ;
        msword = ( int ) pop_ps() ;
        lsword = ( int ) pop_ps() ;
        bottom = ( msword << 16 ) + lsword ;
        top += bottom ;
        push_ps( ( unsigned short ) ( top & 0xffff ) ) ;
        push_ps( ( unsigned short ) ( top >> 16 ) ) ;
}

/*
 * Procedure to multiply the top two 32 bit quantities on the parameter
 * stack together producing a 32 bit result with the most significant
 * word first, and the least significat word second.
 */

void umul_op()
{
        unsigned short top , bottom ;
        unsigned long result ;
#ifdef DEBUG
        printf("umul opcode\n");
#endif
        top = pop_ps() ;
        bottom = pop_ps() ;
        result = top * bottom ;
        push_ps( ( unsigned short ) result & 0xffff ) ;
        push_ps( ( unsigned short ) ( result >> 16 ) & 0xffff ) ;
}

/*
```

```
 * Procedure to divide the top 16 bit quantitie on the parameter stack by
 * the bottom 32 bit quantity producing a 32 bit result with the most
 * significant word first, and the least significat word second.
 */

void udiv_op()
{
        unsigned int top , msword , lsword , bottom ;
#ifdef DEBUG
        printf("udiv opcode\n") ;
#endif
        top = ( int ) pop_ps() ;
        if( top == 0 )
        {
                printf("Arithmetic error - attempt to divide by zero.\n" ) ;
                forth_abort() ;
        }
        else
        {
                msword = ( int ) pop_ps() ;
                lsword = ( int ) pop_ps() ;
                bottom = ( msword << 16  ) + lsword ;
                push_ps( ( unsigned short ) ( bottom % top ) ) ;
                push_ps( ( unsigned short ) ( bottom / top ) ) ;
        }
}

/*
 * Procedure to negate the top 16 bit quantity on the parameter stack
 * producing a 16 bit result.
 */

void minus_op()
{
        unsigned short top ;
#ifdef DEBUG
        printf("minus opcode\n") ;
#endif
        top = abs( ( int ) pop_ps() ) ;
        push_ps( ( unsigned short ) -top ) ;
}

/*
 * Procedure to negate the top 32 bit quantity on the parameter stack
 * producing a 32 bit result with the most significant word first, and
 * the least significat word second.
 */

void dminus_op()
{
        unsigned short top  , bottom ;
#ifdef DEBUG
        printf("dminus opcode\n" ) ;
#endif
        top = abs( ( int ) pop_ps() ) ;
        bottom = abs( ( int ) pop_ps() ) ;
        push_ps( -bottom ) ;
        if( bottom != 0 && top == 0 )
                push_ps( -1 ) ;
        else
                if( top == 0xffff )
                        push_ps( 0 ) ;
                else
                        push_ps( -top ) ;

}
```

```
/*
 * Module : Logicops.h
 * Author : Steven James
 * Date   : 25th January 1990
 *
 * This module implements the seven main logic operations required by the
 * forth engine, namely AND OR XOR NOT = > < .
 *
 * Functions :-
 *      and_op() or_op() xor_op() not_op() equal_op()
 *      greater_op() less_op()
 */

#define AND_OP          400
#define OR_OP           401
#define XOR_OP          402
#define NOT_OP          403
#define EQUAL_OP        404
#define GREATER_OP      405
#define LESS_OP         406

/*
 * Procedure to logical AND the top two 16 bit values on the paramter stack
 * together to produce a 16 bit result.
 */

void and_op()
{
#ifdef DEBUG
        printf("and opcode\n");
#endif
        push_ps( pop_ps() & pop_ps() ) ;
}

/*
 * Procedure to logical OR the top two 16 bit values on the parameter stack
 * together to produce a 16 bit result.
 */

void or_op()
{
#ifdef DEBUG
        printf("or opcode\n");
#endif
        push_ps( pop_ps() | pop_ps() ) ;
}

/*
 * Procedure to logical XOR the top two 16 bit values on the parameter stack
 * together to produce a 16 bit result.
 */

void xor_op()
{
#ifdef DEBUG
        printf("xor opcode\n") ;
#endif
        push_ps( pop_ps() ^ pop_ps() ) ;
}

/*
 * Procedure to logical NOT the top 16 bit value on the parameter stack
 * to produce a 16 bit result.
 */

void not_op()
{
#ifdef DEBUG
        printf("not opcode\n") ;
#endif
        push_ps( ~ pop_ps() ) ;
}

/*
 * Procedure to test for equality between the top two 16 bit values on the
 * parameter stack, and return a truth value to reflect the result.
 */

void equal_op()
```

```
{
#ifdef DEBUG
        printf("equal opcode\n") ;
#endif
        if( pop_ps() == pop_ps() )
                push_ps( FTRUE ) ;
        else
                push_ps( FFALSE ) ;
}

/*
 * Procedure to test two 16 bit quantities on the parameter stack, and
 * return a 16 bit result of true ( -1 ) if the top stack element is
 * greater than the second, otherwise return false ( 0 ).
 */

void greater_op()
{
        short top , bottom ;
#ifdef DEBUG
        printf("> opcode\n") ;
#endif
        top = pop_ps() ;
        bottom = pop_ps() ;
        if( bottom > top )
                push_ps( FTRUE ) ;
        else
                push_ps( FFALSE ) ;
}

/*
 * Procedure to test two 16 bit quantities on the parameter stack, and
 * return a 16 bit result of true ( -1 ) if the top stack element is
 * less than the second, otherwise return false ( 0 ).
 */

void less_op()
{
        short top , bottom ;
#ifdef DEBUG
        printf("< opcode\n" ) ;
#endif
        top = pop_ps() ;
        bottom = pop_ps() ;
        if( bottom < top )
                push_ps( FTRUE ) ;
        else
                push_ps( FFALSE ) ;
}
```

```
/*
 * Module : dataops.h
 * Author : Steven James
 * Date   : 26th January 1990
 *
 * This module implements the two data constructs required by the forth
 * engine, namely LIT and CONST.
 *
 * Functions :-
 *      lit_op() const_op()
 */

#define LIT_OP          500
#define CONST_OP        501

/*
 * Procedure to push the literal pointed to by the instruction register
 * onto the parameter stack, and increment the instruction register.
 */

void lit_op()
{
#ifdef DEBUG
        printf("lit opcode\n") ;
#endif
        push_ps( get_word( i ) ) ;
        i += 2  ;
}

/*
 * Procedure to push the constant pointed to be the word address register
 * onto the parameter stack.
 */

void const_op()
{
#ifdef DEBUG
        printf("const opcode\n") ;
#endif
        push_ps( get_word( wa ) ) ;
}
```

```
/*
 * Module : Branchops
 * Author : Steven James
 * Date   : 26th January 1990
 *
 * This module implements the branching operations required by the forth
 * engine, namely BRANCH JUMP DO LOOP +LOOP.
 *
 * Functions :-
 *      branch_op() jump_op() do_op() loop_op() addloop_op()
 */

#define BRANCH_OP       600
#define JUMP_OP         601
#define DO_OP           602
#define LOOP_OP         603
#define ADDLOOP_OP      604

/*
 * Procedure to perform a relative jump by the value pointed to by the
 * instruction register if the top value on the parameter stack is zero.
 */

void branch_op()
{
#ifdef DEBUG
        printf("branch opcode\n") ;
#endif
        if( pop_ps() == 0 )
                i += get_word( i ) ;
        else
                i += 2 ;
}

/*
 * Procedure to perform a relative jump by the value pointed to by the
 * instruction register.
 */

void jump_op()
{
#ifdef DEBUG
        printf("else opcode\n") ;
#endif
        i += get_word( i ) ;
}

/*
 * Procedure to push the top two 16 bit quantities form the parameter
 * stack to the return stack.
 */

void do_op()
{
unsigned short top ;
#ifdef DEBUG
        printf("do opcode\n") ;
#endif
        top = pop_ps() ;
        push_rs( pop_ps() ) ;
        push_rs( top ) ;
}

/*
 * Procedure to increment the top 16 bit quantity on the return stack, and
 * compare it with the second 16 bit quantity, If the second is greater
 * than the firsth then a relative backward jump is made, otherwise the
 * return stack is decremented and the instruction register is incremented
 * to contine normal execution.
 */

void loop_op()
{
short top , bottom ;
#ifdef DEBUG
        printf("loop opcode\n") ;
#endif
        top = pop_rs() + 1 ;
        bottom = pop_rs() ;
```

```
        if( top < bottom )
        {
                push_rs( bottom ) ;
                push_rs( top ) ;
                i += get_word( i ) ;
        }
        else
                i += 2 ;
}

/*
 * Procedure to increment the top 16 bit quantity on the return stack by
 * the top 16 bit quantity on the parameter stack, and compare it with the
 * second 16 bit quantity on the return stack. If the increment is negative
 * and the first 16 bit quantity is greater than the second, or the
 * imcrement is positive and the first 16 bit quantity is smaller than the
 * second, then a relative backward jump is made, otherwise the return stack
 * is is decremented and the instruction register is incremented to contine
 * normal execution.
 */

void addloop_op()
{
short increment , top , bottom ;
#ifdef DEBUG
        printf("addloop opcode\n") ;
#endif
        top = pop_rs() ;
        increment = pop_ps() ;
        top += increment ;
        bottom = pop_rs() ;
        if( ( increment > 0  ) && ( top < bottom  ) )
        {
                push_rs( bottom ) ;
                push_rs( top ) ;
                i += get_word( i ) ;
        }
        else
        {
                if( ( increment < 0  ) && ( top > bottom ) )
                {
                        push_rs( bottom ) ;
                        push_rs( top ) ;
                        i += get_word( i ) ;
                }
                else
                        i += 2 ;
        }
}
```

```
/*
 * Module : Ioops.h
 * Author : Steven James
 * Date   : 25th January 1990
 *
 * This module implements the five main input/output operations required by
 * the forth enginee, namley KEY (EMIT) INLINE FILE STDIN? (.")
 *
 * Functions :-
 *      terminal() configure_strm() open_strm() close_strm()
 *      key_op() emit_op() inline_op() file_op() stdin_op() string_op()
 *      congfigure_term() unconfigure_term()
 */

#define EMIT_OP         700
#define KEY_OP          701
#define INLINE_OP       702
#define FILE_OP         703
#define STDIN_OP        704
#define STRING_OP       705

struct file_elmt
{
        FILE *stream ;
        struct file_elmt *link ;
} ;

struct file_elmt *file_ptr ;

/*
 * Procedure to test if the current input stream is the terminal keyboard,
 * and return TRUE if so, or FALSE if otherwise.
 */

int terminal()
{
        if( file_ptr->stream == stdin )
                return( TRUE ) ;
        else
                return( FALSE ) ;
}

/*
 * Procedure to configure the input stream to that of the terminal keyboard.
 */

void configure_strm()
{
        struct file_elmt *new_file_ptr ;
        new_file_ptr = (struct file_elmt *)
                malloc( sizeof( struct file_elmt ) ) ;
        new_file_ptr->stream = stdin ;
        new_file_ptr->link = NULL ;
        file_ptr = new_file_ptr ;
}

/*
 * Procedure to open a file as an anternative input stream. The previous
 * stream pointer is placed upon the file stack, for later retrieval when
 * the new input stream is exhausted.
 */

open_strm( strm_name )
char *strm_name ;
{
        FILE *stream ;
        struct file_elmt *new_file_ptr ;
        stream = fopen( dp_string() , "r" ) ;
        if( stream == NULL )
        {
                printf("Unable to open file '%s'\n",strm_name ) ;
        }
        else
        {
                new_file_ptr = (struct file_elmt *)
                        malloc( sizeof( struct file_elmt ) ) ;
                new_file_ptr->stream = stream ;
                new_file_ptr->link = file_ptr ;
                file_ptr = new_file_ptr ;
```