Appendix C

Source Listings

```
/*
 * ------------------------------------------------------------------------
 *
 *                              T . I . L . E
 *
 *                Threaded  Interpretive  Language  Enviroment
 *
 * - . - . - . -.- .- . - . -  - . - . - . - . - . - . - . - . - . - . - .
 *
 *            1989/90 Senior Honours Major Project By Steven James
 *
 *                 For The Computational Science Department
 *
 *                                    Of
 *
 *                           St. Andrews University
 *
 *
 * ------------------------------------------------------------------------
 */

#include <stdio.h>
#include <strings.h>
#include <ctype.h>
#include <curses.h>
#include "forward.h"
#include "forthconst.h"
#include "user.h"
#include "trapops.h"
#include "memory.h"
#include "fstacks.h"
#include "fstring.h"
#include "innerops.h"
#include "stackops.h"
#include "memoryops.h"
#include "mathsops.h"
#include "logicops.h"
#include "dataops.h"
#include "branchops.h"
#include "ioops.h"
#include "dictops.h"
#include "defineops.h"
#include "inner.h"
#include "dictionary.h"

#define VERSION         1.15

/*
 * Procedure to display the initial startup message.
 */

void legand()
{
        printf( "TILE - Threaded Interpretive Language Environment.\n" ) ;
        printf( "Version %1.2f (c) University of St. Andrews.\n",VERSION ) ;
}

/*
 * Procedure to report a context error for the last dictionary search.
 */

void context_error()
{
        drop_op() ;
        printf("'%s' is not in the context vocabulary.\n" , dp_string() ) ;
        forth_abort() ;
}

/*
 * Procedure to compile the 16 bit value on top of the parameter stack
 * into the free dictionary as a literal.
 */

void compile_lit()
{
        push_ps( lit_cfa ) ;
        put_word( get_word( DP ) , pop_ps() ) ;
        put_word( get_word( DP ) + 2 , pop_ps() ) ;
        put_word( DP , get_word( DP ) + 4 ) ;
```

```
}

/*
 * Procedure to take a 16 bit code field address from the parameter stack
 * and return true if the cfa pointer to an immediate word.
 */

int is_immediate( nfa )
unsigned short nfa ;
{
        if( get_byte( nfa ) & IMMEDIATE_BIT )
                return( TRUE ) ;
        else
                return( FALSE ) ;
}

/*
 * Procedure to take a 16 bit code filed address from the parameter stack
 * and return true is it is executable.
 */

int executable( nfa )
unsigned short nfa ;
{
        if( get_word( STATE ) == FFALSE )
        {
                return( TRUE ) ;
        }
        else
                return( is_immediate( nfa ) ) ;
}

/*
 * Procedure to search the dictionary chain, starting from the vocabulary
 * specified by the CONTEXT, for the entry that matches the counted string
 * pointed to by DP. The pointer to the counted string is always returned
 * together with a 16 bit truth value of false ( 0 ) if the search fails,
 * or the name filed address of the matching dictionary word, and a 16 bit
 * truth value of true ( -1 ) if the search is successful.
 */

void chain_find()
{
        push_ps( get_word( DP ) ) ;
        push_ps( get_word( get_word( CONTEXT ) ) ) ;
        find_op() ;
        if( pop_ps() == FFALSE )
        {
                drop_op() ;
                push_ps( get_word( DP ) ) ;
                push_ps( get_word( get_word( get_word( CONTEXT ) - 2 ) ) ) ;
                find_op() ;
        }
        else
                push_ps( FTRUE ) ;

}

/*
 * Procedure to compile the 16 bit quantity on the top of the parameter
 * stack into the free dictionary.
 */

void compile_word()
{
        put_word( get_word( DP ) , pop_ps() ) ;
        put_word( DP , get_word( DP ) + 2 ) ;
}

main()
{
        unsigned short converted ;
        reset_ps() ;
        reset_rs() ;
        reset_trap() ;
        reset_exit() ;
        claim_memory() ;
        configure_variables() ;
        configure_term() ;
```

```
        configure_strm() ;
        build_dictionary() ;
        legand() ;
        do
        {
                if( terminal() )
                        printf( "ok\n" ) ;
                inline_op() ;
                push_ps( SPACE ) ;
                word_op() ;
                while( get_byte( get_word( DP ) ) != NULL )
                {
                        chain_find() ;
                        if( pop_ps() == FTRUE )
                        {
                                nfa = pop_ps() ;
                                drop_op() ;
                                push_ps( nfa + get_word( WIDTH ) + 3 ) ;
                                if( executable( nfa ) )
                                        inner( get_word( INNER ) ) ;
                                else
                                        compile_word() ;
                        }
                        else
                        {
                                drop_op() ;
                                number_op() ;
                                converted = pop_ps() ;
                                if( converted == FFALSE && !trapped() )
                                        context_error() ;
                                else
                                if( get_word( STATE ) == FTRUE )
                                {
                                        if( converted == 2 )
                                        {
                                                swap_op() ;
                                                compile_lit() ;
                                        }
                                        compile_lit() ;
                                }
                        }
                        reset_trap() ;
                        push_ps( SPACE ) ;
                        word_op() ;
                }
        }
        while( !finished() ) ;
        free_memory() ;
        unconfigure_term() ;
}
```

```
/*
 * Module : Forward.h
 * Author : Steven James
 * Date   : 13th February 1990
 *
 * This module defines all forward references used within the tile
 * source code.
 */

unsigned short dp , nfa , last ;
unsigned short lit_cfa , colon_cfa  , semi_cfa , builds_cfa ;

void reset_ps() ;
void reset_rs() ;

unsigned short get_word() ;
unsigned char get_byte() ;

void put_byte() ;

void word_op() ;
```

```
/*
 * Module : forthconst.h
 * Author : Steven James
 * Date   : 10th February 1990
 *
 * This module defines the main constants required by the forth engine.
 */

#define IMMEDIATE_BIT   128
#define HIDDEN_BIT      64
#define LENGTH_MASK     63
#define FTRUE           0xffff
#define FFALSE          0
#define MAXLINE         255
#define SPACE           32
#define TAB             9
```

```
/*
 * Module : User.h
 * Author : Steven James
 * Date   : 29th January 1990
 *
 * This module implements the initialisation, and configuration of the
 * seven user variables required by the forth engine.
 *
 * Functions :-
 *      configure_variables()
 */

void put_word() ;

#define DP              0
#define STATE           2
#define TIB             4
#define IN              6
#define BASE            8
#define CURRENT         10
#define CONTEXT         12
#define WIDTH           14
#define OUT             16
#define INNER           18

#define INITIAL_DP      120
#define INITIAL_STATE   0
#define INITIAL_TIB     40
#define INITIAL_IN      0
#define INITIAL_BASE    10
#define INITIAL_CURRENT 0
#define INITIAL_CONTEXT 0
#define INITIAL_WIDTH   20
#define INITIAL_OUT     0
#define INITIAL_INNER   0


/*
 * Procedure to initialises the user variables outlined above to their
 * initial cold start values.
 */

void configure_variables()
{
        put_word( DP , INITIAL_DP )   ;
        put_word( STATE, INITIAL_STATE ) ;
        put_word( TIB , INITIAL_TIB ) ;
        put_word( IN , INITIAL_IN ) ;
        put_word( BASE , INITIAL_BASE ) ;
        put_word( CURRENT , INITIAL_CURRENT ) ;
        put_word( CONTEXT , INITIAL_CONTEXT ) ;
        put_word( WIDTH , INITIAL_WIDTH ) ;
        put_word( OUT , INITIAL_OUT ) ;
        put_word( INNER , INITIAL_INNER ) ;
}
```

```
/*
 * Module : Trapops.h
 * Author : Steven James
 * Date   : 13th February 1990
 *
 * This module implements the seven main trap operations required by the
 * forth engine, namely trap and exit ( set, reset and test ) and abort.
 * and abort.
 *
 * Functions :-
 *      set_trap() reset_trap() int trapped() forth_abort()
 *      set_exit() reset_exit() int finished()
 */

int trap_evnt , exit_evnt ;

/*
 * Procedure to set the trap flag so that the innner interpreter will trip
 * back into the outer interpreter at the end of this execution cycle.
 */

void set_trap()
{
        trap_evnt = TRUE ;
}

/*
 * Procedure to reset the trap flag to avert the associated actions.
 */

void reset_trap()
{
        trap_evnt = FALSE ;
}

/*
 * Procedure to test if the trap flag is set and return true, otherwise
 * return false.
 */

int trapped()
{
        return( trap_evnt == TRUE ) ;
}

/*
 * Procedure to terminate the text input buffer interpretation.
 */

void forth_abort()
{
        put_word( get_word( TIB ) + get_word( IN ) , NULL ) ;
        set_trap() ;
        reset_ps() ;
        reset_rs() ;
        put_word( STATE , FFALSE ) ;
}

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

void set_exit()
{
        exit_evnt = TRUE ;
}

/*
 * Procedure to reset the trap flag to avert the associated actions.
 */

void reset_exit()
{
        exit_evnt = FALSE ;
}

/*
 * Procedure to test if the exit flag is set and return true, otherwise
```

```
 * return false.
 */

int finished()
{
        return( exit_evnt == TRUE ) ;
}
```

```
/*
 * Module : Memory.h
 * Author : Steven James
 * Date   : 19th January 1990
 *
 * This module implements the memory addresing mechanisms of the forth
 * engine, and simulates the 64K of contigious 8 bit bytes required by the
 * interpreter.
 *
 * Functions :-
 *      put_byte(ushort,char)   char get_byte(ushort)
 *      put_word(ushort,ushort) ushort get_word(ushort)
 *      claim_memory() free_memory()
 */

#define BLOCKSIZE 0xffff

char *base ;

/*
 * Procedure to initialise the address space to map onto the 64K of
 * contigious 8 bit bytes required by the interpreter.
 */

void claim_memory()
{
        base = ( char * ) malloc( BLOCKSIZE ) ;
}

/*
 * Procedure to free the 64K of contigious 8 bit bytes used by the i
 * interpreter and return them to the free heap.
 */

void free_memory()
{
        free( base ) ;
}

/*
 * Procedure to store an 8 bit quantity in the 68K address space.
 */

void put_byte( addr, value )
unsigned short addr;
unsigned char value ;
{
        char *place ;
        place = base + addr ;
        *place = value ;
}

/*
 * Procedure to store a 16 bit quantity in the 64K address space.
 * A 16 bit value is stored as two consecutive 8 bit bytes with the
 * most significant byte first.
 */

void put_word( addr , value )
unsigned short addr, value ;
{
        put_byte( addr , value >> 8 );
        put_byte( addr + 1 , value & 0xff );
}

/*
 * Procedure to fetech an 8 bit quantity from the 64K address space.
 */

unsigned char get_byte( addr )
unsigned short addr ;
{
        char *place ;
        place = base + addr ;
        return( *place ) ;
}

/*
 * Procedure to fetch a 16 bit quantity from the 64K address space.
```

```
 * A 16 bit value is stored as two consecutive 8 bit bytes with the
 * most significant byte first.
 */

unsigned short get_word( addr )
unsigned short addr ;
{
        unsigned short value ;
        value = get_byte( addr ) << 8 ;
        value = value + get_byte ( addr + 1 ) ;
        return( value ) ;
}
```

```
/*
 * Module : Stacks.h
 * Author : Steven James
 * Date   : 18th Jaunary 1990
 *
 * This module implements the two main stacks used by the forth engine,
 * namely the parameter and return stacks, each of which has associated
 * push, pop and empty functions.
 *
 * Functions :-
 *      push_ps(ushort) push_rs(ushort) ushort pop_ps() ushort pop_rs()
 *      bool empty_ps() bool empty_rs()
 *      reset_ps() reset_rs()
 */

struct stk_elmt
{
        unsigned short  value ;
        struct  stk_elmt *link ;
} ;

struct stk_elmt *ps ;
struct stk_elmt *rs ;

/*
 * Procedure to test if the parameter stack is empty and return a boolean
 * integer representing the result of the test ( 0 = false ,  -1 = true )
 */

int empty_ps()
{
        return(ps == NULL) ;
}

/*
 * Procedure to test if the parameter stack is empty and return a boolean
 * integer representing the result of the test ( 0 = false , -1 = true )
 */

int empty_rs()
{
        return(rs == NULL) ;
}

/*
 * procedure to push a 16 bit value onto the parameter stack.
 */

void push_ps( n )
        unsigned short n ;
{
        if( ! trapped() )
        {
                struct stk_elmt *new_ps ;
                new_ps = (struct stk_elmt *)
                        malloc( sizeof( struct stk_elmt ) ) ;
                new_ps->value = n ;
                new_ps->link = ps ;
                ps = new_ps ;
        }
}

/*
 * Procedure to push a 16 bit value onto the return stack.
 */

void push_rs( n )
        unsigned short n ;
{
        if( ! trapped() )
        {
                struct stk_elmt *new_rs ;
                new_rs = (struct stk_elmt *)
                        malloc( sizeof( struct stk_elmt ) ) ;
                new_rs->value = n ;
                new_rs->link = rs ;
                rs = new_rs ;
        }
}
```

```
/*
 * Procedure to pop a 16 bit value from the parameter stack. The routine
 * does not trap parameter stack underflow, and will return a zero under
 * such circumstances.
 */

unsigned short pop_ps()
{
        struct stk_elmt *new_ps ;
        unsigned short n = 0 ;
        if( empty_ps() )
        {
                if( ! trapped() )
                {
                        printf( "Parameter stack underflow.\n" ) ;
                        forth_abort() ;
                }
        }
        else
        {
                new_ps = ps->link ;
                n = ps->value ;
                free( ps ) ;
                ps = new_ps ;
        }
        return( n ) ;
}

/*
 * Procedure to pop a 16 bit value from the return stack. The routine does
 * not trap return stack underflow, and will return a zero under such
 * circumstances.
 */

unsigned short pop_rs()
{
        struct stk_elmt *new_rs ;
        unsigned short n  = 0 ;
        if( empty_rs() )
        {
                if( ! trapped() )
                {
                        printf( "Return stack underflow.\n" );
                        forth_abort() ;
                }
        }
        else
        {
                new_rs = rs->link ;
                n = rs->value ;
                free( rs ) ;
                rs = new_rs ;
        }
        return( n ) ;
}

/*
 * Procedure to empty the parameter stack.
 */

void reset_ps()
{
        while( !empty_ps() )
                pop_ps() ;
}


/*
 * Procedure to empty the parameter stack.
 */

void reset_rs()
{
        while( !empty_rs() )
                pop_rs() ;
}
```

```
/*
 * Module : fstacks.h
 * Author : Steven James
 * Date   : 18th Jaunary 1990
 *
 * This module implements the two main stacks used by the forth engine,
 * namely the parameter and return stacks, each of which has associated
 * push, pop and empty functions.
 *
 * Functions :-
 *      push_ps(ushort) push_rs(ushort) ushort pop_ps() ushort pop_rs()
 *      bool empty_ps() bool empty_rs()
 *      reset_ps() reset_rs()
 */

#define RSTKSIZE        10000
#define PSTKSIZE        1000

unsigned short rstk[ RSTKSIZE ] ;
unsigned short pstk[ PSTKSIZE ] ;

int rs , ps ;

/*
 * Procedure to test if the parameter stack is empty and return a boolean
 * integer representing the result of the test ( 0 = false , -1 = true )
 */

int empty_ps()
{
        return( ps == 0 ) ;
}

/*
 * Procedure to test if the parameter stack is empty and return a boolean
 * integer representing the result of the test ( 0 = false , -1 = true )
 */

int empty_rs()
{
        return( rs == 0 ) ;
}

/*
 * procedure to push a 16 bit value onto the parameter stack.
 */

void push_ps( n )
        unsigned short n ;
{
        if( ! trapped() )
                if( ps == PSTKSIZE )
                {
                        printf( "Parameter stack overflow.\n" ) ;
                        forth_abort() ;
                }
                else
                        pstk[ ps++ ] = n ;
}

/*
 * Procedure to push a 16 bit value onto the return stack.
 */

void push_rs( n )
        unsigned short n ;
{
        if( ! trapped() )
                if( rs == RSTKSIZE )
                {
                        printf( "Return stack overflow.\n" ) ;
                        forth_abort() ;
                }
                else
                        rstk[ rs++ ] = n ;
}

/*
 * Procedure to pop a 16 bit value from the parameter stack. The routine
```

```
 * does not trap parameter stack underflow, and will return a zero under
 * such circumstances.
 */

unsigned short pop_ps()
{
        unsigned short n = 0 ;
        if( empty_ps() )
        {
                if( ! trapped() )
                {
                        printf( "Parameter stack underflow.\n" ) ;
                        forth_abort() ;
                }
        }
        else
                n = pstk[ --ps ] ;
        return( n ) ;
}

/*
 * Procedure to pop a 16 bit value from the return stack. The routine does
 * not trap return stack underflow, and will return a zero under such
 * circumstances.
 */

unsigned short pop_rs()
{
        unsigned short n  = 0 ;
        if( empty_rs() )
        {
                if( ! trapped() )
                {
                        printf( "Return stack underflow.\n" );
                        forth_abort() ;
                }
        }
        else
                n = rstk[--rs ] ;
        return( n ) ;
}

/*
 * Procedure to empty the parameter stack.
 */

void reset_ps()
{
        ps = 0 ;
}

/*
 * Procedure to empty the parameter stack.
 */

void reset_rs()
{
        rs = 0 ;
}
```

```
/*
 * Module : fstring.h
 * Author : Steven January 1990
 * Date   : 14th February 1990
 *
 * This module implements the string handeling routines that enable 'c'
 * to manipulate the 'tile' counted string construct.
 *
 * Functions :-
 *      char *dp_string()
 */

/*
 * Procedure to construct a 'c' string from the 'tile' counted string
 * stored at he top of the free dictionary.
 */

char *dp_string()
{
        char string[MAXLINE] ;
        unsigned short buffer , length ;
        int i ;
        buffer = get_word( DP ) ;
        length = get_byte( buffer++ ) & LENGTH_MASK ;
        for( i = 0 ; i < length ; i++ )
                string[i] = ( char ) get_byte( buffer + i ) ;
        string[i] = ( char ) NULL ;
        return( string ) ;
}
```

```
/*
 * Module : Innerops.h
 * Author : Steven James
 * Date   : 21st January 1990
 *
 * This module implements the eight main inner interpreter operations
 * required by the forth engine, namely RUN NEXT COLON SEMI EXEC TRAP
 * EXIT ABORT .
 *
 * Functions :-
 *      run() next() colon_op() semi_op() exec_op() trap_op() exit_op()
 *      abort_op() ;
 */

#define COLON_OP        1
#define SEMI_OP         2
#define EXEC_OP         3
#define TRAP_OP         4
#define EXIT_OP         5
#define ABORT_OP        6

unsigned short i,wa,ca ;

/*
 * Procedure to fetch the word pointed to by the word address register,
 * and place it in the code address register. Then increment the
 * word address register.
 */

void run()
{
        ca = get_word( wa ) ;
        wa += 2 ;
}

/*
 * Procedure to fetch the word pointed to by the instruction register,
 * and place it in the word address register. Then increment the
 * instruction register.
 */

void next()
{
        wa = get_word( i ) ;
        i += 2 ;
        run() ;
}

/*
 * Procedure to push the instruction register onto the return stack,
 * and transfer the word addresss register to the instruction register.
 */

void colon_op()
{
#ifdef DEBUG
        printf("colon opcode\n");
#endif
        push_rs( i ) ;
        i = wa ;
}

/*
 * Procedure to pop the top item on the return stack into the instrcution
 * register.
 */

void semi_op()
{
#ifdef DEBUG
        printf("semi opcode\n");
#endif
        i = pop_rs() ;
}

/*
 * Procedure to pop the top item on the parameter stack into the word
 * address register.
 */
```

```
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 ;
```

```
        }
}

/*
 * Procedure to close the current input stream, and return to the previous
 * input stream.
 */

close_strm()
{
        struct file_elmt *new_file_ptr ;
        fclose( file_ptr->stream ) ;
        new_file_ptr = file_ptr->link ;
        free( file_ptr ) ;
        file_ptr = new_file_ptr ;
}

/*
 * Procedure to display the ASCII equivalent of the top 16 bit value on the
 * parameter stack.
 */

void emit_op()
{
#ifdef DEBUG
        printf("emit opcode\n") ;
#endif
        putchar( pop_ps() );
}

/*
 * Procedure to scan the keyboard and push the ASCII equivalent of the key
 * pressed onto the parameter stack as a 16 bit value.
 */

void key_op()
{
#ifdef DEBUG
        printf("key opcode\n") ;
#endif
        noecho() ;
        raw() ;
        push_ps( getchar() ) ;
        noraw() ;
        echo() ;
}

/*
 * Procedure to accept a line of text from the current input stream, and
 * transfer it to the text input buffer. If an EOF is encountered then the
 * input stream is reset to accept input from the keyboard. The text
 * interpreter offset is set to point to the begining of the text input
 * buffer reguardless of any stream directed action.
 */

void inline_op()
{
        char string[MAXLINE] ;
        unsigned short buffer = get_word( TIB ) ;
        int i = 0 ;
#ifdef DEBUG
        printf("inline opcode\n") ;
#endif
        if( fgets( string , MAXLINE , file_ptr->stream ) == NULL )
        {
                if( ! terminal() )
                {
                        printf( "[End of file]\n" ) ;
                        close_strm() ;
                }
                else
                        set_exit() ;
        }
        else
                for( i = 0 ; strcmp( &string[i] , "\n" ) ; i++ )
                {
                        if( string[i] == TAB )
                                string[i] = ( char ) SPACE ;
                        put_byte( buffer + i , string[i] ) ;
```

```
                }
        put_word( buffer + i , NULL ) ;
        put_word( IN , 0 ) ;
}

/*
 * Procedure to take the string pointed to be the IN offset within the
 * text input buffer, and attempt to open it as an input stream. If this
 * generates an error then the input stream is reset to accept input from
 * the keyboard, and an error messages is displayed.
 */

void file_op()
{
#ifdef DEBUG
        printf("file opcode\n");
#endif
        push_ps( SPACE ) ;
        word_op() ;
        open_strm( dp_string() ) ;
        forth_abort() ;
}

/*
 * Procedure to check if the input is indirect via the standard input
 * stream, and return a 16 bit value to reflect the result.
 */

void stdin_op()
{
#ifdef DEBUG
        printf("stdin opcode\n");
#endif
        if( terminal() )
                push_ps( FTRUE ) ;
        else
                push_ps( FFALSE ) ;
}

/*
 * Procedure to display the counted string pointed to by the contents of
 * the instruction register.
 */

void string_op()
{
        unsigned short count ;
#ifdef DEBUG
        printf("string opcode\n");
#endif
        count = get_byte( i++ ) ;
        do
        {
                putchar( get_byte( i++ ) ) ;
wrong! -->      get_word( OUT , get_word( OUT + 1 ) ) ;   put_word(OUT, get_word(OUT)+1);
                count-- ;
        }
        while( count != 0 ) ;
}


/*
 * Procedure to configure the terminal prier to processing by
 * the forth input/output operations.
 */

void configure_term()
{
        initscr() ;

}

/*
 * Procedure to re-configure the terminal the state prier to
 * envoking the configure_term procedure.
 */

void unconfigure_term()
{
```

```
        endwin() ;
}
```

```
/*
 * Module : dictops.h
 * Author : Steven James
 * Date   : 10th February 1990
 *
 * This module implements the five dictionary constructs required by the
 * forth engine, namely WORD FIND NUMBER.
 *
 * Functions :-
 *      word_op() find_op() number_op()
 */

#define WORD_OP         800
#define FIND_OP         801
#define NUMBER_OP       802

/*
 * Procedure to copy a sequence of bytes, delimited by the value specified
 * on top of the parameter stack, from the current position in the text
 * input buffer to the top of the free dictionay as a counted string.
 * The IN offset within the text input buffer is advanced to point the
 * position directly after the delimiter specified.
 */

void word_op()
{
        unsigned short dp , tib , in , separator , count ;
#ifdef DEBUG
        printf("word opcode\n");
#endif
        dp = get_word( DP ) ;
        tib = get_word( TIB ) ;
        in  = get_word( IN ) ;
        count = 0 ;
        separator = pop_ps() ;
        if( isspace( separator ) )
                while( isspace( get_byte( tib + in ) ) )
                        in++ ;
        while( get_byte( tib + in + count ) != separator &&
               get_byte( tib + in + count ) != NULL )
                count++ ;
        put_byte( dp , count ) ;
        dp++ ;
        while( count != 0 )
        {
                put_byte( dp , get_byte( tib + in ) ) ;
                dp++ ;
                in++ ;
                count-- ;
        }
        put_word( IN , in + 1 ) ;
}

/*
 * Procedure to search the dictionary, starting from the vocabulary
 * specified by a 16 bit quantity on the top of the parameter stack, for
 * the entry that matches the counted string pointed to by the second 16 bit
 * quantity on the parameter stack. The pointer to the counted string is
 * always returned together with a 16 bit truth value of false ( 0 ) if the
 * search fails, or the name filed address of the matching dictionary word,
 * and a 16 bit truth value of true ( -1 ) if the search is successful.
 */

void find_op()
{
        unsigned short  width , count ,  cstring  , lfa , nfa , offset ;
#ifdef DEBUG
        printf("find opcode\n") ;
#endif
        nfa = pop_ps() ;
        cstring = pop_ps() ;
        width = get_word( WIDTH ) ;
        do
        {
                lfa = nfa ;
                offset = cstring ;
                count = get_byte( cstring ) ;
                while( ( get_byte( nfa )  & LENGTH_MASK  ) == count &&
                       !( get_byte( nfa ) & HIDDEN_BIT ) )
```

```
                {
                        if( count > width )
                                count = width ;
                        do
                        {
                                nfa++ ;
                                offset++ ;
                                count-- ;
                        }
                        while( get_byte( nfa ) == get_byte( offset )
                               && count != 0 ) ;
                }
                if( count == 0 && get_byte( nfa ) == get_byte( offset ) )
                        nfa = lfa ;
                else
                        nfa = get_word( lfa + width + 1 ) ;
        }
        while( nfa != 0 && nfa != lfa ) ;
        push_ps( cstring ) ;
        if( nfa == 0 )
                push_ps( FFALSE ) ;
        else
        {
                push_ps( nfa ) ;
                push_ps( FTRUE ) ;
        }
}

/*
 * Procedure to convert a lowercase character into an uppercase character.
 */

char upper( n )
        char n ;
{
        if( islower( n ) )
                return( toupper( n ) ) ;
        else
                return( n ) ;
}

/*
 * Procedure convert the counted string on the top of the free dictionary
 * to a decimal value using the user variable BASE as the current number
 * base. The 16 bit decimal aproximation together with a 16 bit value of
 * 1 is returned if the counted string was a valid 16 bit number, or the
 * 32 bit decimal aproximation together with a 16 bit value of 2 if the
 * counted string was a valid 32 bit number. Otherwise 16 bit value 0 and
 */

void number_op()
{
unsigned short dp , count , base  , number ;
int result , negative , large , error ;
#ifdef DEBUG
        printf("number opcode\n") ;
#endif
        dp = get_word( DP ) ;
        base = get_word( BASE ) ;
        large = FALSE ;
        error = FALSE ;
        negative = FALSE ;
        count = get_byte( dp++ ) ;
        result = 0 ;
        if( get_byte( dp ) == 45 )
        {
                negative = TRUE ;
                dp++ ;
                count-- ;
        }
        do
        {
                number = upper( get_byte( dp ) ) ;
                if( number == 46 )
                {
                        if( large )
                                error = TRUE ;
                        else
```

```
                                large = TRUE ;
                        count-- ;
                        dp++ ;
                }
                else
                {
                        number -= 48 ;
                        if( number > 0 )
                                if( number > 9 )
                                        if( number > 16 )
                                                number = number - 7 ;
                                        else
                                                error = TRUE ;
                        if( ( number < base )  && !error )
                        {
                                result = result * base + number ;
                                count-- ;
                                dp++ ;
                        }
                        else
                                error = TRUE ;
                }
        }
        while( ( count != 0 )  && ! error ) ;
        if( negative )
                result = -result ;
        push_ps( ( unsigned short ) result & 0xffff ) ;
        if( large )
        {
                if( result > 0xfff )
                        push_ps( ( unsigned short ) ( result >> 16 ) ) ;
                else
                {
                        if( negative )
                                push_ps( -1 ) ;
                        else
                                push_ps( 0 ) ;
                }
        }
        if( ! error )
                if( large )
                        push_ps( 2 ) ;
                else
                        push_ps( 1 ) ;
        else
                push_ps( FFALSE ) ;
}
```

```
/*
 * Module : defineops.h
 * Author : Steven James
 * Date   : 19th February 1990
 *
 * This module implements the five defining constructs required by the
 * forth engine, namley <BUILDS DOES> CREATE : ; .
 *
 * Functions :-
 *      builds_op() does_op() create_op() define_op() end_op()
 */

#define BUILDS_OP       900
#define DOES_OP         901
#define CREATE_OP       902
#define DEFINE_OP       903
#define END_OP          904

/*
 * Procedure to push the the current content of the word address register
 * onto the top of th parameter stack as a 16 bit quantity.
 */

void builds_op()
{
#ifdef DEBUG
        printf("builds opcode\n") ;
#endif
        push_ps( wa ) ;
}

/*
 * Procedure to push the contents of the instruction register onto the top
 * of the return stack, set in instruction register to the 16 bit quantity
 * pointed to by the contents of the word address register and push the
 * nincremented contents of the word address register onto the parameter
 * stack as a 16 bit quantity.
 */

void does_op()
{
#ifdef DEBUG
        printf("does opcode\n") ;
#endif
        push_rs( i ) ;
        i = get_word( wa ) ;
        push_ps( wa + 2 ) ;
}

/*
 * Procedure to construct a header ( ie. the name and link fields ) for the
 * the next space delimited string in the text input buffer, and report any
 * attempts to redefine an existing word.
 */

void mkheader()
{
        push_ps( get_word( get_word( CURRENT ) ) ) ;
        push_ps( SPACE ) ;
        word_op() ;
        push_ps( get_word( DP ) ) ;
        push_ps( get_word( get_word( CONTEXT ) ) ) ;
        find_op() ;
        if( pop_ps() == FTRUE )
        {
                printf("'%s' has been redefined.\n" , dp_string() ) ;
                drop_op() ;
        }
        drop_op() ;
        put_word( get_word( CURRENT ) , get_word( DP ) ) ;
        put_word( DP , get_word( DP ) + get_word( WIDTH ) + 1 ) ;
        put_word( get_word( DP ) , pop_ps() ) ;
        put_word( DP , get_word( DP ) + 2 ) ;
}

/*
 * Procedure to construct a word header from the space delimited string in
 * the text input buffer, with the code field address pointing to the
 * body of the word.
```

```
 */

void create_op()
{
#ifdef DEBUG
        printf("create opcode\n") ;
#endif
        mkheader() ;
        put_word( get_word( DP ) , get_word( DP ) + 2 ) ;
        put_word( DP , get_word( DP ) + 2 ) ;
}

/*
 * Procedure to construct a colon definition from the space delimited string
 * in the text input buffer.
 */

void define_op()
{
#ifdef DEBUG
        printf("define opcode\n") ;
#endif
        put_word( CONTEXT , get_word( CURRENT ) ) ;
        mkheader() ;
        put_word( get_word( DP ) , colon_cfa ) ;
        put_word( DP , get_word( DP ) + 2 ) ;
        put_word( STATE , FTRUE ) ;
}

/*
 * Procedure to compile a semi colon and terminate compilation of a colon
 * definition by resetting the STATE user variable.
 */

void end_op()
{
#ifdef DEBUG
        printf("end opcode\n") ;
#endif
        push_ps( semi_cfa ) ;
        put_word( get_word( DP ) , pop_ps() ) ;
        put_word( DP , get_word( DP ) + 2 ) ;
        put_word( STATE , FFALSE ) ;
}
```

```
/*
 * Module : inner.h
 * Author : Steven James
 * Date   : 31th January 1990
 *
 * This module implements the operation of the forth inner interpreter.
 *
 * Functions :-
 *      inner() ;
 */

/*
 * Procedure to implements the operation of the forth inner interpreter.
 * On entry the address specified is placed in the instruction register
 * and a next is executed. The inner intrpreter will continue to operate
 * until a trap or invalid instruction is encounted, at which point the
 * interpreter will hault and return control back to the calling
 * procedure or program.
 */

void inner( addr )
unsigned short addr;
{
        i = addr ;
        next() ;
#ifdef DEBUG
        printf("registers : i %d wa %d ca %d\n",i,wa,ca) ;
#endif
        do
        {
                switch ( get_word( ca ) )
                {
                        case COLON_OP   : colon_op() ;
                                          next() ;
                                          break ;
                        case SEMI_OP    : semi_op() ;
                                          next() ;
                                          break ;
                        case EXEC_OP    : exec_op() ;
                                          run() ;
                                          break ;
                        case TRAP_OP    : trap_op() ;
                                          break ;
                        case EXIT_OP    : exit_op() ;
                                          next() ;
                                          break ;
                        case ABORT_OP   : abort_op() ;
                                          next() ;
                                          break ;
                        case DROP_OP    : drop_op() ;
                                          next() ;
                                          break ;
                        case DUP_OP     : dup_op() ;
                                          next() ;
                                          break ;
                        case SWAP_OP    : swap_op() ;
                                          next() ;
                                          break ;
                        case OVER_OP    : over_op() ;
                                          next() ;
                                          break ;
                        case ROT_OP     : rot_op() ;
                                          next() ;
                                          break ;
                        case PUSHR_OP   : pushr_op() ;
                                          next() ;
                                          break ;
                        case POPR_OP    : popr_op() ;
                                          next() ;
                                          break ;
                        case R_OP       : r_op() ;
                                          next() ;
                                          break ;
                        case J_OP       : j_op() ;
                                          next() ;
                                          break ;
                        case EMPTY_OP   : empty_op() ;
                                          next() ;
                                          break ;
```

```
case STORE_OP    : store_op() ;
                   next() ;
                   break ;
case FETCH_OP    : fetch_op() ;
                   next() ;
                   break ;
case CSTORE_OP   : cstore_op() ;
                   next() ;
                   break ;
case CFETCH_OP   : cfetch_op() ;
                   next() ;
                   break ;
case ADD_OP      : add_op() ;
                   next() ;
                   break ;
case DADD_OP     : dadd_op() ;
                   next() ;
                   break ;
case UMUL_OP     : umul_op() ;
                   next() ;
                   break ;
case UDIV_OP     : udiv_op() ;
                   next() ;
                   break ;
case MINUS_OP    : minus_op() ;
                   next() ;
                   break ;
case DMINUS_OP   : dminus_op() ;
                   next() ;
                   break ;
case AND_OP      : and_op() ;
                   next() ;
                   break ;
case OR_OP       : or_op() ;
                   next() ;
                   break ;
case XOR_OP      : xor_op() ;
                   next() ;
                   break ;
case NOT_OP      : not_op() ;
                   next() ;
                   break ;
case EQUAL_OP    : equal_op() ;
                   next() ;
                   break ;
case GREATER_OP : greater_op() ;
                   next() ;
                   break ;
case LESS_OP     : less_op() ;
                   next() ;
                   break ;
case LIT_OP      : lit_op() ;
                   next() ;
                   break ;
case CONST_OP    : const_op() ;
                   next() ;
                   break ;
case BRANCH_OP   : branch_op() ;
                   next() ;
                   break ;
case JUMP_OP     : jump_op() ;
                   next() ;
                   break ;
case DO_OP       : do_op() ;
                   next() ;
                   break ;
case LOOP_OP     : loop_op() ;
                   next() ;
                   break ;
case ADDLOOP_OP : addloop_op() ;
                   next() ;
                   break ;
case EMIT_OP     : emit_op() ;
                   next() ;
                   break ;
case KEY_OP      : key_op() ;
                   next() ;
                   break ;
case INLINE_OP   : inline_op() ;
```

```
                                        next() ;
                                        break ;
                        case FILE_OP    : file_op() ;
                                        next() ;
                                        break ;
                        case STDIN_OP   : stdin_op() ;
                                        next() ;
                                        break ;
                        case STRING_OP  : string_op() ;
                                        next() ;
                                        break ;
                        case WORD_OP    : word_op() ;
                                        next() ;
                                        break ;
                        case FIND_OP    : find_op() ;
                                        next() ;
                                        break ;
                        case NUMBER_OP  : number_op() ;
                                        next() ;
                                        break ;
                        case BUILDS_OP  : builds_op() ;
                                        next() ;
                                        break ;
                        case DOES_OP    : does_op() ;
                                        next() ;
                                        break ;
                        case CREATE_OP  : create_op() ;
                                        next() ;
                                        break ;
                        case DEFINE_OP  : define_op() ;
                                        next() ;
                                        break ;
                        case END_OP     : end_op() ;
                                        next() ;
                                        break ;
                        default         : printf("bad opcode\n") ;
                                        forth_abort() ;
                                        break ;
                }
#ifdef DEBUG
        printf("registers : i %d wa %d ca %d\n",i,wa,ca) ;
#endif
        }
        while ( ! trapped() ) ;
        reset_trap() ;
}
```

```
/*
 * Module : dictionary.h
 * Author : Steven James
 * Date   : 10th February 1990
 *
 * This module implements the dictionary structure required by the forth
 * engine.
 *
 * Functions :-
 *      build_dictionary()
 */

/*
 * Procedure to store a 16 bit quantity at the current dictionary address,
 * and increment the dictionary pointer.
 */

void dpword( value )
unsigned short value ;
{
        put_word( dp , value ) ;
        dp += 2 ;
}

/*
 * Procedure to store an 8 bit quabtity at the current dictionary address,
 * and increment the dictionary pointer.
 */

void dpbyte( value )
unsigned char value ;
{
        put_byte( dp , value ) ;
        dp++ ;
}

/*
 * Procedure to construct a dictionay header for the name specified,
 * and maintain the overall dictionary link structure.
 */

void header( string )
char string[INITIAL_WIDTH] ;
{
        int i = 0 ;
        dpbyte( ( unsigned char ) strlen( string ) ) ;
        do
        {
                if( i < strlen( string ) )
                        dpbyte( string[i] ) ;
                else
                        dpbyte( NULL ) ;
                i++;
        }
        while( i < INITIAL_WIDTH ) ;
        dpword( last ) ;
        last = dp - ( INITIAL_WIDTH  + 3 ) ;

}

/*
 * Procedure to set the IMMEDIATE_BIT on the name filed address of the
 * most reciently defined header.
 */

void immediate()
{
        put_byte( last , get_byte( last ) ^ IMMEDIATE_BIT ) ;
}

/*
 * Procedure to construct the initial forth dictionary.
 */

void build_dictionary()
{
        unsigned exec_cfa , trap_cfa  , word_cfa ;
        dp = INITIAL_DP ;
        last = 0 ;
```

```
colon_cfa = dp ;
dpword( COLON_OP ) ;

semi_cfa = dp ;
dpword( dp + 2 ) ;
dpword( SEMI_OP ) ;

trap_cfa = dp ;
dpword( dp + 2 ) ;
dpword( TRAP_OP ) ;

header( "execute" ) ;
exec_cfa = dp ;
dpword( dp + 2 ) ;
dpword( EXEC_OP ) ;

header( "abort" ) ;
dpword( dp + 2 ) ;
dpword( ABORT_OP ) ;

header( "bye" ) ;
dpword( dp + 2 ) ;
dpword( EXIT_OP ) ;

put_word( INNER , dp ) ;
dpword( exec_cfa ) ;
dpword( trap_cfa ) ;

header( "drop" ) ;
dpword( dp + 2 ) ;
dpword( DROP_OP ) ;

header( "dup" ) ;
dpword( dp + 2 ) ;
dpword( DUP_OP ) ;

header( "swap" ) ;
dpword( dp + 2 ) ;
dpword( SWAP_OP ) ;

header( "over" ) ;
dpword( dp + 2 ) ;
dpword( OVER_OP ) ;

header( "rot" ) ;
dpword( dp + 2 ) ;
dpword( ROT_OP ) ;

header( ">R" ) ;
dpword( dp + 2 ) ;
dpword( PUSHR_OP ) ;

header( "R>" ) ;
dpword( dp + 2 ) ;
dpword( POPR_OP ) ;

header( "R" ) ;
dpword( dp + 2 ) ;
dpword( R_OP ) ;

header( "I" ) ;
dpword( dp + 2 ) ;
dpword( R_OP ) ;

header( "J" ) ;
dpword( dp + 2 ) ;
dpword( J_OP ) ;

header( "empty?" ) ;
dpword( dp + 2 ) ;
dpword( EMPTY_OP ) ;

header( "!" ) ;
dpword( dp + 2 ) ;
dpword( STORE_OP ) ;

header( "@" ) ;
dpword( dp + 2 ) ;
```

```
dpword( FETCH_OP ) ;

header( "C!" ) ;
dpword( dp + 2 ) ;
dpword( CSTORE_OP ) ;

header( "C@" ) ;
dpword( dp + 2 ) ;
dpword( CFETCH_OP ) ;

header( "+" ) ;
dpword( dp + 2 ) ;
dpword( ADD_OP ) ;

header( "d+" ) ;
dpword( dp + 2 ) ;
dpword( DADD_OP ) ;

header( "u*" ) ;
dpword( dp + 2 ) ;
dpword( UMUL_OP ) ;

header( "u/" ) ;
dpword( dp + 2 ) ;
dpword( UDIV_OP ) ;

header( "minus" ) ;
dpword( dp + 2 ) ;
dpword( MINUS_OP ) ;

header( "dminus" ) ;
dpword( dp + 2 ) ;
dpword( DMINUS_OP ) ;

header( "and" ) ;
dpword( dp + 2 ) ;
dpword( AND_OP ) ;

header( "or" ) ;
dpword( dp + 2 ) ;
dpword( OR_OP ) ;

header( "xor" ) ;
dpword( dp + 2 ) ;
dpword( XOR_OP ) ;

header( "not" ) ;
dpword( dp + 2 ) ;
dpword( NOT_OP ) ;

header( "=" ) ;
dpword( dp + 2 ) ;
dpword( EQUAL_OP ) ;

header( ">" ) ;
dpword( dp + 2 ) ;
dpword( GREATER_OP ) ;

header( "<" ) ;
dpword( dp + 2 ) ;
dpword( LESS_OP ) ;

header( "lit" ) ;
lit_cfa = dp ;
dpword( dp + 2 ) ;
dpword( LIT_OP ) ;

header( "0branch" ) ;
dpword( dp + 2 ) ;
dpword( BRANCH_OP ) ;

header( "branch" ) ;
dpword( dp + 2) ;
dpword( JUMP_OP ) ;

header( "(do)" ) ;
dpword( dp + 2 ) ;
dpword( DO_OP ) ;
```

```
    header( "(loop)" ) ;
    dpword( dp + 2 ) ;
    dpword( LOOP_OP ) ;

    header( "(+loop)" ) ;
    dpword( dp + 2 ) ;
    dpword( ADDLOOP_OP ) ;

    header( "(emit)" ) ;
    dpword( dp + 2 ) ;
    dpword( EMIT_OP ) ;

    header( "key" ) ;
    dpword( dp + 2 ) ;
    dpword( KEY_OP ) ;

    header( "inline" ) ;
    dpword( dp + 2 ) ;
    dpword( INLINE_OP ) ;

    header( "load" ) ;
    dpword( dp + 2 ) ;
    dpword( FILE_OP ) ;

    header( "stdin?" ) ;
    dpword( dp + 2 ) ;
    dpword( STDIN_OP ) ;

    header( "(.\")" ) ;
    dpword( dp + 2 ) ;
    dpword( STRING_OP ) ;

    header( "word" ) ;
    word_cfa = dp ;
    dpword( dp + 2 ) ;
    dpword( WORD_OP ) ;

    header( "(find)" ) ;
    dpword( dp + 2 ) ;
    dpword( FIND_OP ) ;

    header( "number" ) ;
    dpword( dp + 2 ) ;
    dpword( NUMBER_OP ) ;

    builds_cfa = dp ;
    dpword( BUILDS_OP ) ;

    header( "create" ) ;
    dpword( dp + 2 ) ;
    dpword( CREATE_OP ) ;

    header( "(:)" ) ;
    dpword( dp + 2 ) ;
    dpword( DEFINE_OP ) ;

    header( "(;)" ) ;
    dpword( dp + 2 ) ;
    dpword( END_OP ) ;
    immediate() ;

    header( "\\" ) ;
    dpword( colon_cfa ) ;
    dpword( lit_cfa ) ;
    dpword( 0 ) ;
    dpword( word_cfa ) ;
    dpword( semi_cfa ) ;
    immediate() ;

    dpword( last ) ;
    put_word( CURRENT , dp ) ;
    put_word( CONTEXT , dp ) ;
    dpword( last ) ;

    put_word( DP , dp ) ;
}
```

```
(:) 0=
        0 = (;)

(:) 0<
        0 < (;)

(:) 0>
        0 > (;)

(:) <>
        = not (;)

(:) -
        minus + (;)

(:) d-
        dminus d+ (;)

(:) 1+
        1 + (;)

(:) 1-
        1 - (;)

(:) 2+
        2 + (;)

(:) 2-
        2 - (;)

(:) +!
        dup @ rot + swap ! (;)

(:) +C!
        dup C@ rot + swap C! (;)

(:) here
        0 @ (;)

(:) allot
        here + 0 ! (;)

(:) last
        12 @ @ (;)

(:) pfa
        14 @ 5 + + (;)

(:) cfa
        2- (;)

(:) lfa
        cfa 2- (;)

(:) nfa
        lfa 14 @ 1+ - (;)

(:) ,
        here ! 2 0 +! (;)

(:) C,
        here C! 1 0 +! (;)

(:) immediate
        last dup C@ 128 or swap C! (;)

(:) [
        0 2 ! (;) immediate

(:) ]
        -1 2 ! (;)

(:) -find
        32 word here last (find) dup not 0branch [ 22 , ]
        drop drop here 12 @ 2- @ @ (find) (;)

(:) '
        -find 0branch [ 14 , ] swap drop pfa cfa branch [ 8 , ] drop 0 (;)
```

```
(:) 2dup
        over over (;)

(:) -dup
        dup 0branch [ 4 , ] dup (;)

(:) 2drop
        drop drop (;)

(:) 2swap
        >R rot rot R> rot rot (;)

(:) 2over
        >R >R 2dup R> R> 2swap (;)

(:) literal
        [ ' lit , ] [ ' lit , ] , , (;) immediate

(:) ca!
        last pfa cfa ! (;)

(:) scode
        R> ca! (;)

(:) ;code
        [ ' scode ] literal , 0 2 ! (;) immediate

(:) constant
        create , ;code 501 ,

(:) <builds
        0 constant (;)

(:) does>
        R> last pfa cfa 2+ ! ;code 901 ,

(:) variable
        <builds 0 , does> (;)

 0 constant DP
 2 constant STATE
 4 constant TIB
 6 constant IN
 8 constant BASE
10 constant CURRENT
12 constant CONTEXT
14 constant WIDTH
16 constant OUT
18 constant INNER
20 constant HLD
22 constant FENCE

32 constant BL
64 constant C/L*
 0 constant FALSE
-1 constant TRUE

(:) hex
        16 BASE ! (;)

(:) binary
        2 BASE ! (;)

(:) decimal
        10 BASE ! (;)

(:) s->d
        dup 0< (;)

(:) ++
        1 swap +! (;)

(:) --
        -1 swap +! (;)

(:) +-
        0< 0branch [ 4 , ]  minus (;)

(:) d+-
```

```
        0< 0branch [ 4 , ] dminus (;)

(:) abs
        dup +- (;)

(:) dabs
        dup d+- (;)

(:) min
        2dup > 0branch [ 4 , ] swap drop (;)

(:) max
        2dup < 0branch [ 4 , ] swap drop (;)

(:) m*
        2dup xor >R abs swap abs u* R> d+- (;)

(:) m/
        over >R >R dabs R abs u/ R> R xor +- swap R> +- swap (;)

(:) *
        m* drop (;)

(:) /mod
        >R s->d R> m/ (;)

(:) /
        /mod swap drop (;)

(:) mod
        /mod drop (;)

(:) */mod
        >R m* R> m/ (;)

(:) */
        */mod swap drop (;)

(:) m/mod
        >R 0 R u/ R> swap >R u/ R> (;)

(:) fill
        >R dup 0> [ ' 0branch , 18 ,  ] 1- 2dup + R swap C!
        [ ' branch , -22 , ] R> drop drop drop (;)

(:) move
        dup 0> [ ' 0branch , 30 , ] 1- rot rot over @ over !
        2+ swap 2+ swap rot [ ' branch , -34 , ] 2drop drop (;)

(:) cmove
        dup 0> [ ' 0branch , 30 , ] 1- rot rot over C@ over C!
        1+ swap 1+ swap rot [ ' branch , -34 , ] 2drop drop (;)

(:) emit
        (emit) 1 OUT +! (;)

(:) cr
        13 emit 10 emit 0 OUT ! (;)

(:) space
        BL emit (;)

(:) spaces
        0 max -dup  0branch [ 14 , ] 0
        (do) space (loop) [ -4 , ] (;)

(:) count
        dup 1+ swap C@ (;)

(:) type
        -dup 0branch [ 18 , ] over + swap
        (do) I C@ emit (loop) [ -8 , ]
        branch [ 4 , ] drop (;)

(:) hold
        HLD -- HLD @ C! (;)

(:) pad
        here 68 + (;)
```

```
(:) <#
        pad HLD ! (;)

(:) #>
        drop drop HLD @ pad over - (;)

(:) sign
        rot 0< 0branch [ 8 , ] 45 hold (;)

(:) #
        BASE @ m/mod rot 9 over < 0branch [ 8 , ] 39 + 48 + hold (;)

(:) #s
        # 2dup or 0= 0branch [ -10 , ] (;)

(:) d.r
        >R swap over dabs <# #s sign #> R> over - spaces type (;)

(:) .r
        >R s->d R> d.r (;)

(:) d.
        0 d.r space (;)

(:) u.
        0 d. (;)

(:) .
        s->d d. (;)

(:) ."
        STATE @ 0branch [ 26 , ]
        [ ' (.") ] literal , here 34 word C@ 1+ allot branch [ 14 , ]
        34 word here count type (;) immediate

(:) ?comp
        STATE @ not 0branch [ 54 , ]
        ." Error - Word may only be used when compiling." cr abort (;)

(:) ?exec
        STATE @ 0branch [ 54 , ]
        ." Error - Word may only be used when executing." cr abort (;)

(:) ?pairs
        - 0 <> 0branch [ 35 , ]
        ." Error - Pairs don't match." cr abort (;)

(:) vocabulary
        <builds CONTEXT @ , last , does> 2+ CONTEXT ! (;)

(:) definitions
        CONTEXT @ CURRENT ! (;)

(:) seal
        here dup 1 C, 0 C, WIDTH @ 1+ + DP ! 0 , CONTEXT @ ! (;)

(:) id.
        count 63 and WIDTH @ min type (;)

(:) words
        ." Context vocabulary : " CONTEXT @ WIDTH @ 9 + - id. cr
        ." Current vocabulary : " CURRENT @ WIDTH @ 9 + - id. cr
        C/L 1+ OUT ! last
        OUT @ C/L > 0branch [ 4 , ] cr dup id. space space pfa lfa @ dup
        0= 0branch [ -34 , ]  drop cr (;)

(:) if
        ?comp [ ' 0branch ] literal , here 0 , 2 (;) immediate

(:) (endif)
        ?comp 2 ?pairs here over - swap ! (;)

(:) then
        (endif) (;) immediate

(:) else
        2 ?pairs [ ' branch ] literal , here 0 , swap 2 (endif) 2 (;)
        immediate
```