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.
 */
```