```
void add_op()
        {
                unsigned short top , bottom ;
                top = pop_ps() ;
                bottom = pop_ps() ;
                push_ps(bottom + top) ;
        }
```

The procedure **dadd_op** performs an unsigned addition upon the top two double length values on the parameter stack, and replaces both with the double length result. Double length values are represented upon the parameter stack as two single length value with the least significant 16 bits below the most significant.

```
void dadd_op()
unsigned int lsword , msword , top , bottom ;
{
        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 ) ;
}
```

4.8.4 Logic

The logic described in section 4.5.2 is based around the unsigned short bitwise logic operations provided as part of the basic 'C' environment. The procedure **and_op** is a typical example of the structure of the logic operations provide at the abstract machine level.

```
void and_op()
{
        push_ps( pop_ps() & pop_ps() ) ;
}
```

The procedure **equal_op** is slightly different, in that it operations upon unsigned integer representations, and returns Boolean truth values.

```
        void equal_op()
{
        if( pop_ps(() == pop_ps() )
                push_ps( FTRUE ) ;
        else
                push_ps( FFALSE ) ;
}
```

4.8.5 Stack Manipulation

The stack manipulation described in section 4.5.3 is based around the stack operators described in section 4.8.2. The procedure **rot_op** is a typical example of the structure of a stack operation.

```
void rot_op()
{
        unsigned short top , middle , bottom ;
        top = pop_ps() ;
        middle = pop_ps() ;
        bottom = pop_ps() ;
        push_ps( middle ) ;
        push_ps( top ) ;
        push_ps( bottom ) ;
}
```

Given that the stack is implemented as a vector, it would perhaps seem more appropriate to directly access the individual elements via the vector de-reference mechanism, rather than adhering to a combination of potentially slow push and pop operations. This approach would however, require that all stack manipulating operation re-coded if the implementation of the stack where changed.

4.8.6 I/O

The I/O described in section 4.5.4 is based around the concept of a unix stream. The procedure **inline_op** excepts a line of text (terminated by a carriage return or end of file marker) from the current input stream, and transfers it into the **TILE** text input buffer.

```
FILE *stream = stdin ;

void inline_op()
{
        char string[MAXLINE] ;
        unsigned short buffer = get_word( TIB ) ;
        int i = 0 ;
        if(fgets(string,MAXLINE,stream) == NULL)
        {
                if(stream != stdin)
                {
                        printf("[End of file]\n") ;
                        fclose(stream) ;
                        stream = stdin ;
                }
                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 ) ;
}
```

Upon termination of a file stream, input is reset back to the keyboard stream. However if the keyboard stream is terminated via a **control 'd'** the abstract machine and interpreter will exit and return to the operating system. This restriction is entirely due to the unix stream mechanism, and is not a deliberate **TILE** design consideration.

4.8.7 Memory Manipulation

The memory manipulation described in section 4.5.5 is based around the memory operators described in section 4.8.1. The procedure **store_op** is a typical example of the structure of a memory manipulation operation.

```
void  store_op()
{
        unsigned short addr , value ;
        addr = pop_ps() ;
        value = pop_ps() ;
        put_word(addr , value) ;
}
```

4.8.8 Dictionary Manipulation

The dictionary manipulation described in section 4.5.6 is based around the memory operators described in section 4.8.1. The procedure **create_op** constructs 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. If the word already exists within the vocabulary, then a warning message is displayed before the word is re-defined.

```
Void  create_op
{
        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  re-defined.\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) ;
        put_word(get_word(DP) , get_word(DP) + 2 ) ;
        put_word(DP, get_word(DP) + 2) ;
}
```

A more complete, and in-depth discussion of the concepts underlying the implementation of the dictionary can be found in section 2, together with detailed descriptions of both the word and vocabulary structures.

4.8.9 Numeric Manipulation

The numeric manipulation described in section 4.5.7 is based around the memory operators described in section 4.8.1. Both the **const_op** and **lit_op** procedures are virtually identical, save that the **lit_op** procedure de-references, and increments the instruction register and not the word address register, as is the case for the **const_op** procedure.

```
void  const_op()
{
        push_ps(get_word(wa))  ;
}
```

4.8.10 String Manipulation

The string manipulation described in section 4.5.8 is based around the memory operators described in section 4.8.1. Although the **number_op** and **word_op** procedure are essentially performing quit different functions, both manipulate **TILE** strings and user variables in a similar fashion.

```
void  word_op()
{
        unsigned short dp , tib , in , seperator , count ;
        dp = get_word(DP) ;
        tib = get_word(TIB) ;
        in = get_word(IN) ;
        count = 0 ;
        seperator = pop_ps() ;
        if(isspace(seperator))
                while(isspace(get_byte(tib + in))
                        in++ ;
        while(get_byte(tib + in + count) != seperator &&
                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) ;
}
```

4.8.11 Flow Control.

The two flow control mechanisms described in section 4.5.8, are essential based around the same technique. Both the **branch_op** and **jump_op** procedures manipulate the flow of execution by modifying the instruction register, conditionally in the case of the **branch_op** procedure, and unconditional in the case of the **jump_op** procedure.

```
void branch_op()
{
        if(pop_ps() == 0)
                i += get_word(i) ;
else
        i += 2 ;
}
```

4.8.12 Iteration.

The iteration mechanism described in section 4.5.10, is essentially based around the flow control technique described in section 4.8.11 with some additional iteration and loop indexing operations. The **do_op** procedure initiates the loop structure by pushing a loop index upper bound, and initial value onto the return stack. The **loop_op** and **addloop_op** procedures are essential the same, save that the **addloop_op** procedure requires an increment value on the parameter stack.

```
void addloop_op
{
        short increment , top , bottom ;
        top = pop_ps() ;
        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 += getword(i) ;
                }
                else
                        i += 2 ;
}
```

4.8.12 Execution Mechanism.

The execution mechanism described in section 4.6 is based around around a **case** statement **switched** by the contents of the code address register. Each abstract machine instruction has corresponding **case** statement, based upon the op-code for that instruction.

```
do
{
        switch(get_word(ca))
        {
                case  DUP_OP      :     dup_op() ;
                                        next() ;
                                        break ;
                case  SWAP_OP     :     swap_op() ;
                                        next() ;
                                        break ;
                .
                .
                .
                .
        }
}
while( trapped());
```

Each procedure within the **case** statement is followed by a **next** operation to traverse the graph representing the next function application until a leaf node is encountered, at which point case structure is again evaluated. This cycle continues until a trap instruction is executed, at which point the **case** structure will terminate and control will be returned to the interpreter.

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

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

The **next** procedure is the bottle neck of the entire execution mechanism. It is executed after each instruction and will therefore directly affect the overall performance of the abstract machine. For this reason every effort should be made to ensure the the implementation of **next** is a efficient as possible.

4.8.13 Termination.

The termination described in section 4.7 is based around the execution mechanism described in section 4.8.12. Essentially termination is dependent upon the state of two event variables **trap_evnt** and **exit_evnt**. Each of which has associated set, reset and test operations. The abstract machine tests that **trap_evnt** is true at the end of each execution cycle. If not it terminates and trips back to the interpreter. The interpreter tests that **exit_evnt** is true at the end of each interpretive cycle. If not it will terminate and return to the operating system.

```
int trap_event , exit_evnt

void set_trap
{
    trap_event = TRUE ;
}
```

4.9 Summary

The **TILE** abstract machine is based around a very simple computation model, requiring a minimal amount of hardware to achieve effective execution. Because of the relative simplicity, the abstract machine has proven to be both easy to implement and efficient in-terms of the execution, and memory requirements.

5

The TILE Project In Retrospect

5.1 Introduction

The **TILE** project as described in the preliminary specification has been successful, in that all the major design objectives described in this document have be implemented and can be seen to work correctly for all of the applications that have to date be implemented. Their are however a number of additional features not mentioned in the specification that in retrospect I feel should have been.

5.1.1 Extended File Support

The file handling facilities provided as part of the basic environment are limited in that there is no mechanism for reading from or writing to multiple files, the former is implemented at a rudimentary level with single file **load** operation, the later is not supported at any level. Incorporating such mechanisms into the existing the abstract machine would mean the addition of a third stack containing file handling information, and complementary file handling operations.

5.1.2 Graphics Support

Given the graphics capabilities of the hardware on which **TILE** is implemented it would seen natural to incorporate into the language some mechanism for creating, maintaining and manipulating raster images within the window environment.

5.1.3 Virtual Dictionary Support

The dictionary constitutes the primary **TILE** heap management system, providing a fundamental mechanism for the creation and maintenance of data objects. The life of such data objects is limited by the length of time **TILE** itself is executing, as termination will result in all data objects within the dictionary structure being lost. Extending the life of data objects requires a mechanism for storing and restoring sections of the dictionary to and from the backing store.

5.1.4 Floating Point Support

Conventionally **TIL's** have restricted all arithmetic and related operations to manipulate only single and double length integer quantities. This restriction is partly

due to the relative difficulty of implementing floating point operations in machine code and partly the result of a desire to prevent the type of operator overloading that would need to be incorporated should such operations be provided. The coding considerations are not relevant to **TILE** as the implementation is based around a high level language that supports floating point operations. The overloading on the other hand is a more complex problem that would need be to be reconciled should floating point be adopted.

5.2 Summary

Throughout this document I have attempted to described the problems and design decisions as they appeared to me and my subsequent solution of each. I feel however that in so doing I may over simplified the considerable deign effort necessary to accomplish these solutions, leaving the reader with the impression that the problems where trivial and the solutions straight-forward. I should also like to emphasise to the reader with some knowledge of other **TIL's** that the design specified is entirely of my own construction and although in many respects there are similarities between **TILE** and **FORTH** is was never my intention to copy and implement the later. Any features which are similar are include because I considered then to be beneficial in which case the design and implementation is of my own devising. An example is the vocabulary construct, **FORTH** has standard vocabulary structure that I considered to complex and cumbersome, as a result I completely re-designed the vocabulary structure of **TILE** to provide a simpler and more usable alternative. Whatever impression left with the reader, I for my part consider the design and implementation of **TILE** to been one of the most challenging and rewarding projects I have undertaken to-date.

References

ANON. 1980. The FORTH-79 Standard
FORTH Interest Group, San Carlos, California.

ANON. 1983. The FORTH-83 Standard
FORTH Interest Group, San Carlos, California.

BRODIE, L. 1981. Starting FORTH.
Prentice-Hall, Englewood Cliffs, New Jersey.

BRODIE, L. 1984.Thinking FORTH.
Prentice-Hall, Englewood Cliffs, New Jersey.

DERICK, M. 1982. FORTH Encyclopedia.
Mountain View Press, Mountain View, California.

KELLY, Mahlon G. 1986. FORTH: A Text and Reference.
Prentice-Hall, Englewood Cliffs, New Jersey.

LOELIGER, R. Threaded Interpretive Languages.
Byte Books, Peterborough, New Hampshire.