Programming Forth

Stephen Pelc

Programming Forth

Copyright

For further information

MicroProcessor Engineering Limited
133 Hill Lane, Southampton
SO15 5AF, UK
Tel: +44 (0)23 8063 1441
Fax: +44 (0)23 8033 9691

e-mail: mpe@mpeltd.demon.co.uk
tech-support@mpeltd.demon.co.uk

web: www.mpeltd.demon.co.uk

Acknowledgements

I would like to thank the following people in particular for their involvement in the content of this book:

Peter Knaggs, Chris E. Bailey, Bill Stoddart, Bob Marchbanks, Hans Bezemer

These people have influenced “Programming Forth” and contributed to it, and their input is valued. All faults are my own. Much criticism and proof-reading was provided by the readers of the *comp.lang.forth* newsgroup, especially:

William Cook, Anton Ertl

Contents

List of Figures

List of Tables

1 Introduction

About this book

Programming Forth introduces you to modern Forth systems. In 1994 the ANS Forth standard was released and unleashed a wave of creativity among Forth compiler writers. Because the ANS standard, unlike the previous informal Forth-83 standard, avoids specifying implementation details, implementers took full advantage. The result has been what I choose to call modern Forths, which are available from a range of sources both commercial and open-source.

This book concentrates on introducing people who already know some programming to ANS Forth systems. It is not a treatise on ANS Forth itself – if you need the gory details, the last public (freely distributable) draft of the ANS standard is included on the CD supplied with purchased copies of the book. Copies in PDF format are available by download from http://www.mpeltd.demon.co.uk/arena at no cost. If you are a novice programmer (or indeed at all interested in the craft of programming) read this book alongside “Starting Forth” and “Thinking Forth” by Leo Brodie. How to get them is in the chapter on other books and resources.

Apart from the introduction of ANS Forth itself, Programming Forth includes examples of varying sizes, exercises, some advanced topics, how to take best advantage of Forth and project management.

The material is derived from course material from MicroProcessor Engineering and teaching work at Teesside University by Bill Stoddart and Peter Knaggs, plus new material. Both the printed and the PDF versions are updated from time to time to incorporate changes requested by readers. If you want to comment on the book please send feedback to programforth@mpeltd.demon.co.uk - I appreciate all your comments and contributions.

What Forth is

Forth is a member of the class of extensible interactive languages, which includes classical implementations of Smalltalk. **Extensible** means that there is no distinction between the keywords (core words) and the routines that you write. Once a new definition has been compiled, even from the keyboard, it is immediately available to you. **Interactive** means that you can talk to it from your keyboard.

Forth is a different sort of computer language. Forth code is easy to debug because Forth is interactive, fast because Forth is compiled and powerful because it is extensible. Forth is a language with a definite style.

Forth was developed by Charles (Chuck) Moore in the early 1960s. Moore's work with computers at MIT and the Stanford Linear Accelerator Centre left him dissatisfied. The turn-round time for editing, compiling and running a program using the then current generation of ALGOL and FORTRAN compilers was too slow. His solution to this was to write a simple text interpreter in ALGOL which read in words (any printable characters except space) and performed actions associated with them. Words were either primaries (i.e. "understood" by the interpreter) or secondaries (i.e. defined in terms of other words).

After his initial success with an ALGOL based interpreter at MIT and Stanford, Moore moved on to work with Burroughs equipment. This hardware was strongly oriented around a stack. This influenced the further development of Forth.

Implementations were written in BALGOL, COBOL and SBOL (the Burroughs Systems Programming Language). These provided manipulation words for the stack: **DROP**, **DUP**, **SWAP**, etc. which are still found in modern Forth systems. The first true Forth system which resembled what we now perceive as Forth was then created by Moore on an IBM 1130. The word size of this machine limited the users to having names of not more than five characters. Had it not been for this IBM limitation the name `Forth' would have been `Fourth' - standing for Fourth Generation Language.

The first Forth application was a Radio Telescope Data Acquisition Program written for a Honeywell H316 at the National Radio Astronomy Observatory. This implementation of Forth used a dictionary to store words defined by the user.

Key concepts

This text is derived from a posting on comp.lang.forth by Dwight Elvey.

“Like many programming languages, Forth has an execution model or virtual machine. Unlike many languages, Forth exposes this to the programmer.

The Forth virtual machine is discussed in more detail in the next chapter.

*All source code elements are either numbers or **words**. Words are what are called functions, procedures or subroutines in other languages. Words can be either user defined or part of the Forth core and are entered as a **dictionary**. Words are composed of other Forth elements and/or true machine code.*

All elements are executed in sequential order, left to right, top to bottom. Sequential order is only broken by flow structure words.

*Normal input and output of words is by the **data stack**, **return stack**, variables (values), or arrays.*

Nesting is maintained by a return stack. Execution of words implies nesting, except at the lowest levels of true machine code.

The programmer is responsible to maintain stacks.

Forth can be either compiled or interpreted. These can be selected at will by the programmer but is usually done in an orderly fashion.

All the rest comes from the above.”

One of the key portions of Forth is its interactivity – you sit in front of the keyboard and play (explore) with your application. Daniel Ciesinger wrote:

“Further, I use Forth to test C libraries. Experience is that this finds errors which are not found by C test procedures. I even prefer it to Rational Test Realtime which is too indirect for my taste. Takes eternities to write test cases. In Forth, you just test interactively and cut and paste your test cases into your favourite editor afterwards, so you can run the same test again.”

Interactivity, Development, Testing and Debugging

Elizabeth Rather wrote this posting about working with Chuck Moore (the originator of Forth) on comp.lang.forth:

“Any given problem has a certain intrinsic level of complexity. In the solution of the problem, this complexity will be conserved: if you dive in with too little advance thought, your solution may become very complex by the time it's done. On the other

hand, if you invest more in thought, design, and preparation, you may be able to achieve a very simple solution (the complexity hasn't gone away, it's become embodied in the sophistication of the design).

In connection with our recent discussion, that investment certainly can take the form of prototyping. When I was working with Chuck, he'd typically go through the following process:

1. Read the spec. Assert confidently that it really isn't as complicated as that, and write a very clever program that solves the essence of the problem.

2. Enter a phase in which the customer repeatedly points out aspects of the spec that have been ignored or omitted; in fixing these, the code gets more and more complicated.

*3. At some point, a full understanding of the problem emerges. All the previous code is thrown out and a new program emerges which represents a simple, elegant solution to the **whole** problem.*

Elizabeth's notes illustrate that coding is not the major part of delivering software. I am very rarely given a formal specification for a new piece of software, despite the effort, time and cost-savings that a good specification provide. Writing a piece of software is mostly an iterative process involving exploration, design, coding and debugging. These realities of the software engineer's life lead to techniques whose buzzwords include "rapid prototyping" and "extreme programming". In such an environment, debugging is far more expensive than coding in both time and cost.

Interactive testing and debugging is a key feature of Forth. Rapid debugging can be achieved by the application of formal scientific method. The slide below is taken from one of our course presentations.

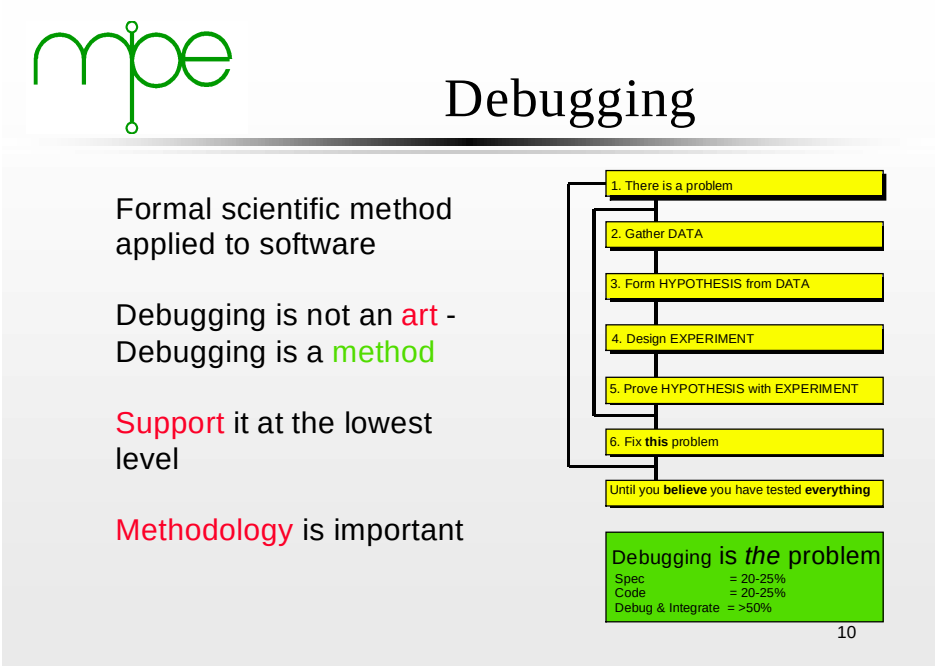

Figure 1: Formal debugging

The diagram above shows that debugging consists of two nested loops. How fast you can go around the inner loop determines how fast you can debug a system. Interactive debugging is the fastest route I have found. The stages of debugging are:

1) Make the problem repeatable. This usually involves finding out which inputs cause the problem.

2) Gather data about the problem. This observation is crucial, so take this stage slowly and carefully. I have seen people immediately dismiss exception displays and crash-dumps which contain vital clues.

3) From the data, form a hypothesis as to what caused the problem.

4) Design an experiment which tests the hypothesis. The important part in designing the experiment is to ensure that it gives a yes/no answer.

5) Run the experiment. If the hypothesis is incorrect, go back to stage 2.

6) Fix the problem.

7) If you have more bugs to fix, go back to stage 1 for the next problem.

Forth is a wonderful tool for debugging systems and their software. Effective debugging in any programming language requires us to understand the techniques required for efficient debugging and how to acquire the required information.

Writing programs in Forth

Forth often takes a little longer to learn than other languages. Just like spoken languages, there are many words to learn in Forth before you can use it well. In Forth parlance, what are called functions, procedures or subroutines in other languages are called **words**. Forth is a language in which very little is hidden from you. Nearly every word that we used along the way to some other function has a name, and is documented in a **glossary**.

As a result of this openness there are many words in the dictionary (type **WORDS** to see them). Functions in Forth are called words in Forth jargon. These words are stored as a **dictionary**, and the group of words forming your area of interest - the context in which you work - is known as a **vocabulary**. For example, words used to define the assembler are often kept in a vocabulary called **ASSEMBLER**. As in all computer languages, there is a jargon to Forth. In this instance the jargon is a technical language, and serves as a set of communication tools so that we can explain our ideas to each other without being bogged down in the minutiae. Persevere, Forth is not only well worth the effort, but is a tool of spectacular productivity in the right hands.

The Forth run-time package is actually a compact combination of interpreter, compiler, and tools. A command or sequence of commands (words) may be executed directly from the keyboard, or loaded from mass storage as if from the keyboard. In hosted versions of Forth (and some embedded systems), you can also take input from a normal operating system text file, created by a normal (non-Forth editor). Programs in Forth are compiled from combinations of existing words (already in the dictionary), new words as defined by the user, and control structures such as **IF ... ELSE ... THEN** or **DO ... LOOP**. Often, new words are developed interactively at the terminal before the final (and tested) version is then entered using the editor and saved on disc, where it can be invoked from the keyboard or used by another program. If you are teaching yourself Forth, get all your books ready in front of the terminal, and try things out as you go along.

The beauty and power of Forth lies in interactivity, extensibility and flexibility. New words can be added either at high or low (assembler) level. Forth is one of the very few languages which can define a data structure and how it is used inside a single definition. This ability to create new words known as **defining words**, which can add

new classes of operators to the language, is one of the keys to the extraordinary power of Forth in the hands of an experienced programmer. A bad Forth programmer is just as much a disaster as in any other language.

If your experience of programming has been in traditionally organised languages such as C or BASIC, you will find reading and writing programs in Forth somewhat bizarre at first. Patience brings rich rewards. Forth becomes much easier to understand once you have mastered a few ideas and played with the language. Among the most important aids in using Forth is the choice of word names. Think about the name of a word in advance. Poets make good Forth programmers. Verbs, nouns, and adjectives all have their place in good Forth programming style. Good choice of word names leads to very readable code, as does the use of white space in source code. You can use any character within a word name - the use of printable ones is sensible. Word names can be up to 31 characters long (more in some implementations), and all the characters are significant.

Forth programs keep most of their working variables on the stack, rather than in named variables, so reading some sections of code can be a little mind-boggling - even for the experienced. The secret is to keep definitions short and simple. Lazy programmers often make good programmers because they make life easy for themselves - and part of making life easy is making sure that you can work out what the code is doing a year from now.

The language lends itself well to bottom-up coding. Like the choice of word names, this can be a double-edged sword. There is no substitute for good overall program design, which can only be done properly from the top down. Bottom-up design and coding is excellent, however, for exploring the nuts and bolts of techniques, algorithms, and low-level interfaces. The ability to interactively create, test, and produce working code early in the development cycle is invaluable. Early working code also helps to keep your boss off your back, and it enables customers to make sensible reactions and discover specification errors before it is too late. Carefully used this feature can save you a great deal of time.

Bottom-up coding has an additional advantage in that it is easy to test with the Forth interpreter. Since you cannot use a word until all its components have been defined, you simply test an application in the order of the source code. Each component that you test is then based on previously tested code. This is a **much** more reliable strategy than testing by running the completed application.

You may well find it profitable to study the source code of the programs supplied in the files with your Forth as a guide to style. The style is the one we use, and has evolved over a number of years, rather than through any theoretical arguments. We find it usable by both the authors and those who have to read other people's code. Read the glossary documentation, and spend a while trying out the functions, and observing their action on the stack.

Editors

Forth source code is usually held in regular text files. You can use your programming editor of choice. Forth syntax colouring files are available for many of them. The MPE Forth layout standard discusses MPE practice for laying out source code.

The earlier ‘screen’ or ‘block’ layout of Forth source code is now obsolete except for special use, mostly on embedded systems or for legacy reasons. Block editors are discussed in the chapter on legacy issues.

Forth Implementation

As with all computer languages, Forth has evolved over time. This book assumes the use of an ANS Forth which conforms to the ANS Forth standard published in 1994. At the time of writing in 2005, the ANS Forth standard is the most widely used.

Modern Forth implementations usually generate optimised native code, and the good ones produce code of the same quality as the good C compilers. For those CPUs in the embedded world where code density is more important than performance, threaded code (interpreted code) implementations are still used, often with limited peephole optimisation.

The different implementation strategies are discussed in the Forth internals chapter.

MPEisms

Because this book was written at MPE using MPE's VFX Forth for Windows for testing the code, it inevitably suffers from the use of a few idioms which are specific to MPE's implementations. I have tried to avoid these and where appropriate they are marked. For example,

MPEism: *MPE implementations of Forth do not care about the case of characters in Forth word names.* ***CAT*** *is the same as* ***cat*** *is the same as* ***Cat****. Embedded comments may be as long as you wish without a space or speed penalty in the compiled code.*

2 Architecture of Forth

Most programming languages, including C, have an underlying architecture or model of the computer. This is often called the language's **virtual machine** (VM), regardless of how the final binary code is produced.

This chapter includes some details which you do not need to appreciate fully to use Forth, but are described here because they are a consequence of the architecture of Forth. You can always come back to these later.

Forth Virtual Machine

Classical or canonical Forth views the world as a CPU connected to main memory and two stacks. The stacks are not addressable, and are quite separate from main memory. C views the world as a CPU connected to memory, which includes a list of frames (usually a stack of frames) which **must** be in addressable memory.

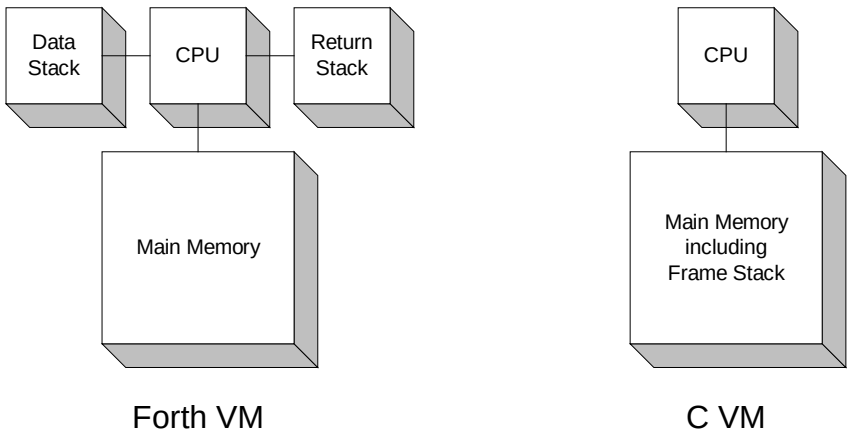

Figure 2: Forth and C Virtual Machines

MPEism: *By adding the necessary registers for the frame stack to the canonical Forth machine, we can support local variables in Forth and C-isms needed to interface efficiently to operating systems with C and Pascal calling conventions. This model is called the SENDIT virtual machine after the project which developed it.*

The SENDIT VM looks remarkably similar to other stack machine CPUs derived from a Forth architecture and designed to execute C efficiently.

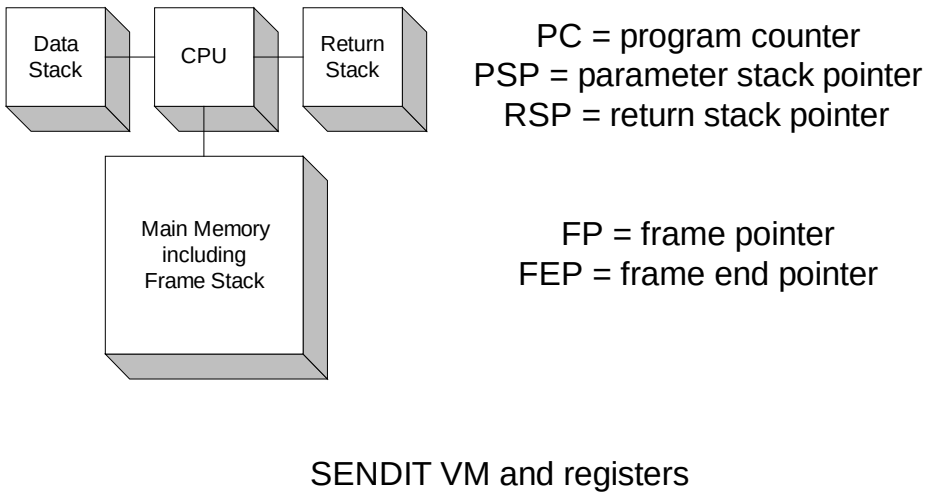

Figure 3: Sendit VM and registers

Several implementations of dual stack architectures have been produced in silicon for use in hard real-time embedded systems, for which the best definition comes from Bernd Paysan - “Late answers are wrong answers”. On some occasions early answers are also wrong answers.

Because dual-stack processors have very few registers, interrupt response is excellent. A 10 MHz RTX2000 has an interrupt response of four cycles, 400ns, and we have seen a video application with a 1 MHz (not a typo) interrupt rate. Because of this, stack CPUs with apparently low clock rates are used in specialised application areas.

Stacks and postfix notation

Forth contains two stacks, one for storing return addresses (what was I doing last/where do I go back to?), and one for storing data. The first stack is called the return stack, and the second is called the data or parameter stack. Where we just refer to “the stack” we nearly always mean the data stack, as this is the one the programmer uses most.

The return stack holds the return addresses of all the words that have been called, but have not yet been left. The return stack is also used for storing temporary data that would only get in the way if kept on the data stack. This sort of data includes loop limits and indices, and data taken off the data stack to reduce the amount of stack manipulation that would otherwise occur. There is a set of words used for transferring data between the stacks.

The data stack is an efficient method of passing data between the words that make up a Forth program. Any word that needs data takes it from the top of the stack, and puts any results back on top of the stack. The word **.S** can be used to display the contents of the stack without destroying them. It is a useful debugging tool. Nearly all modern processors provide for the use of stacks, so stack operations are very fast.

By keeping parameters and temporary data on the data stack, the return stack is not affected by the number of items passed into or returned from a function (word). This means that the C equivalent of **varargs** is efficiently handled, and also that a function can efficiently return more than one item without resorting to pointers and temporary data structures. Consequently, words that return more than one item are common in Forth.

Because stacks are used for data handling, the use of postfix, or Reverse Polish Notation (RPN), is very suitable. In this form of writing arithmetic expressions, operands (the data used) come before the operators (how you use the data), for example:

The normal algebraic notation :-

```
at**2 + bt + c
```

is better expressed for computer evaluation as :-

```
(at + b)t + c
```

which is then expressed in Reverse Polish Notation (RPN) as :-

```
a t * b + t * c +
```

Notice that the use of brackets becomes unnecessary. This is because of the use of the stack to hold intermediate results. Although the use of a stack is intimidating at first, after a while it becomes natural, and eventually it is only noticeable on rare occasions. Remembering that the word **.** is used to print what is on the top of the stack you can try a few bits of arithmetic.

```
1 2 + .
4 5 * .
9 3 / .
1 2 + 3 * .
1 2 3 + * .
1 2 3 * + .
```

Nearly all Forth words remove their data from the top of the stack, and leave the result behind. Words like **+** and ***** remove two items, and leave one behind. There is a Forth word **.S** which prints out the contents of the stack without destroying the contents. Use it whenever you want to see what is on the stack. So far, we have executed words by typing their names at the keyboard.

Data types

Forths are often characterised by the size of an item on the data stack, which is usually 32 bits or 16 bits. Several 64 and 8 bit Forths also exist, as well as 4, 20 and 24 bit systems. Forth is an untyped language, so there is nothing to stop you adding a character to a number. No casts are necessary. The amount of memory needed to store a stack item is called a **cell**.

Words that operate on stack items will just assume that the data is appropriate. Most Forth compilers do not check the data types at all.

An integer occupies a single cell on the stack, as does an address. If you need larger integers, use a double integer which occupies two cells on the data stack, most significant half topmost.

A character is smaller than (or the same size as) an integer, and is zero extended to fill a cell.

Two different floating point implementations are described by the ANS standard. In one, known as a separated float stack, floating point numbers are held on a separate stack reserved for floating point numbers. Floating point operations that require integer parameters take them from the data stack. The second form uses a combined stack, in which floating point numbers share the data stack.

Words and Numbers

All text entered to Forth is treated either as a number (literal, e.g. 1234) or a word (function). A unit of text is separated by spaces or line breaks. These units are either interpreted or compiled. That's all. Although this is a very simple process, it is a key to the use of Forth.

Interpreting and compiling

Forth contains both an interpreter and a compiler. Interpreting means taking the text fed in, converting it into a form the machine can execute, executing that form, and then discarding the executable form.

Compiling means taking the input text, completely converting it into a machine executable form, keeping the executable form, and discarding the text. This can produce a program that runs very fast, but you cannot change anything without first editing the source text, then compiling it (using a separate program called a compiler), and then loading the executable code when you want to run it.

Any text fed to Forth, either from the keyboard or from mass storage, is executed or compiled. Remember, all commands to Forth are pre-defined 'words' in its 'vocabularies', consequently Forth can look up the address of a given word for later execution. Some words in Forth change the way the compiler section deals with text.

For instance we could define a word that squares the value given to it.

```
: squared        ( n1 -- n1^2 )
  dup *  ;
```

The address of the word **:** is found and **:** is executed; the action of **:** is to tell the compiler section to start defining a new word whose name comes next, **squared**, and then compile into the new word the actions of the words that follow. This would carry on for ever unless we had a way of stopping it, and this is provided by 'immediate' words such as **;** which are always executed, regardless of what the compiler would otherwise be doing.

The action of **;** is to stop the compiler compiling word addresses, and return it to the mode of executing the addresses instead. There are other words (defining words) which are used to create words such as **:** - these are one of the keys to advanced use of Forth. At all times remember, however, that the basis of Forth is always very simple. Forth is a language built from a number of very simple ideas, rather than one founded on a few complex systems.

Defining Words and Immediate words

Some words such as **:** in the previous section, are called defining words, because they are used to define new words (**squared** in the previous section); these words are one of the keys to the power of Forth. The word **:** creates a new word in the dictionary and switches Forth from being an interpreter to being a compiler. Any word names met from now on will not be executed, but will be found and compiled into the dictionary. This process repeats until stopped, but it can only be stopped by the execution of

another word, and all words are being compiled, not executed. This problem is dealt with by immediate words.

The solution to the problem of the previous section is to have a class of words which are always executed, regardless of whether Forth is supposed to be compiling. Such words are called **immediate** words. The word **`;`** used to terminate a high level definition, is an example of such a word. When it executes it switches Forth from being a compiler back to being an interpreter, and also compiles the action of the word **`EXIT`** which performs the function of returning from a high level word.

Factoring

Factoring is the term Forth programmers use for splitting complex functions into several smaller ones. It is a key activity in writing good Forth. It is also a key reason why well-written Forth programs tend to be smaller than those written in many other languages. Forth promotes code reuse at a fine-grained level.

Because of the two-stack virtual machine, the performance and code size overheads of factoring are very low. They are outweighed by the benefits of easy code reuse, which improves code size, reliability and maintainability. Code size is reduced because code is called rather than repeated (call by text editor). Reliability is improved because small code is easier to test if it only occurs once. Maintainability is improved if code is simple and only occurs once. If code only occurs once and has no side effects such as modifying variables, it can be completely rewritten with confidence providing that the stack effect remains the same.

From the point of view of C programmers learning Forth, this emphasis on factoring appears strange. When I go back to programming in C, I find my exposure to Forth and factoring benefits my C code. It has been said that there are three types of procedure call:

1) call by value

2) call by reference

3) call by text editor

Call by text editor is not good practice in any language, yet I am constantly amazed by the number of two and three line code units in C programs that are repeated again and again. Repeated code units still need testing and maintenance.

An additional benefit of factoring is that each unit is simpler, usually with fewer items on the stack, and hence easier and faster to program. Programmers new to Forth will find factoring eases the learning curve. As it has big benefits factoring is a habit to be encouraged.

3 How Forth is documented

Comments

Forth has two primary comment words. You can put text between round brackets – parentheses. There must be a space after the opening parenthesis:

```
( <comment> )
```

anywhere in your source code. A comment to the end of the line is started by a backslash used as a word:

```
<code>   \ <comment>
```

If you want to display messages while a file is being compiled by

```
INCLUDE <filename>
```

You can use the word **.(** which behaves like the **(** comment but displays the text.

```
.( This is displayed during compilation )
```

Stack comments

Words in this book are documented in a style popular with many Forth programmers. It shows what is on the stack before the word executes (the input), and what is on the stack after the word has executed (the output). The top of the stack is right of the group, and the execution point is marked by two dashes.

The multiply operator ***** takes two parameters on input, and leaves one on output. It is thus shown:

```
( n1 n2 -- n3 )
```

or

```
\ n1 n2 -- n3
```

The round brackets are Forth's way of marking a comment, and n2 is the top of the stack before execution. In the manual Forth words are written in capital letters to distinguish them from the lower case letters of the rest of the text. It does not matter which you use in your programs. Personally, I prefer the look of programs written in lower case. All children are taught to read using lower case letters. All keyboards are marked in upper case, even those for use by children!

Notation

Data items are described using the following notation

```
OPERAND          DESCRIPTION
n1,n2..          signed numbers (integers)
d1,d2..          double precision signed numbers
u1,u2..          unsigned numbers (integers)
ud1,ud2..        double precision unsigned numbers
addr1..          address
b1,b2..          bytes
c1,c2..          ASCII characters
char1,char22..   ASCII characters
```

```
t/f,t,f          boolean flag, true, false
                 0=false, nz=true,
flag             0=false, -1 =true
```

Note that the ANS standard uses `f` for a flag, whereas many programmers still use `t` and `f` to indicate true and false.

4 First words in Forth

The only sure way to learn Forth is to use it. Forth programmers spend more time at the terminal, because the interactive nature of the language means that words can be tested as soon as they are entered. A result of this feature is that succeeding words, which use previous ones, use tested code. Adherence to the procedure of `top down design', followed by `bottom up' coding and interactive testing, leads to very rapid debugging, and successful program generation. Audits of large software projects reveal that over half the time may be spent on debugging. As this is the largest single activity, it is the one to reduce. If you are at all interested in software management, do read Fred Brook's book, 'The Mythical Man-Month'.

To write new words you can either just type them in at the keyboard, or you can use the editor to put source code in a file.

If you decide just to enter the examples directly, you do not need to enter the comments or use the same layout. In fact Forth is completely free-form. This means that the position of the words is unimportant, only their order.

MPEism: *MPE Forths do not care whether a word is in UPPER CASE or in lower case or in MiXeD case.*

Forth word names can contain any characters except spaces or nulls (ASCII character 0). It is sensible to use printable characters, but Forth does not actually check the characters.

New Forth words are defined by the word **:** whose first action is to pick up the name that follows and use it to make a new entry in the dictionary. Then, everything that follows up to the next **;** defines the action of the word.

By comparison with extended BASICs **:** is equivalent to `DEFine PROCedure` or `DEFine FuNction` and **;** is equivalent to `END PROCedure` or `END DEFine`. For example:

```
: NEW-NAME       \ --
  CR ." This is a new word" CR  ;
NEW-NAME
```

The example word above is called **NEW-NAME** - when you type **NEW-NAME** it will print a new line, and the print the text `This is a new word', and then print another new line. The word **."** prints out all the characters except the space after **."** up to but not including the next double quotation mark (**"**). The word **CR** is a predefined word that generates a new line - **CR** stands for Carriage-Return.

A new Forth word can contain any words that exist in the dictionary. The new Forth word can be executed by typing its name, or included in the definition of another word.

```
: TIMES   \ n1 n2 --
  * .  ;
2 4 TIMES
```

This example will multiply two numbers together and print the result. The word ***** is Forth's multiply word, and **.** is the word to print a number. **TIMES** can be used as part of a word that presents information more prettily to the user. First we print a new line using **CR** then we duplicate the two numbers using **2DUP** and print them out together with the result. Why is the word **SWAP** used? Try it without **SWAP**.

```
: MULTIPLY      ( n1 n2 -- )
  CR 2DUP SWAP .  ." multiplied by " .
  ." equals " TIMES CR  ;
4 3 MULTIPLY
```

We have shown the definition entered on two lines. When you type it in there will be no 'ok' prompt after the first line. This is because you have not finished the definition. The Forth system has converted the list of word names into a dictionary entry called **MULTIPLY** and its associated code. This process is called compilation, and during compilation source text entered from the keyboard is discarded.

If you want to keep source code available for re-use, use your standard text editor, save the file, conventionally with a .FTH extension, and use the word **INCLUDE** to load it, e.g.

```
INCLUDE MULTIPLY.FTH
```

which will compile it just as if you had entered the text at the keyboard. **INCLUDE**s can be nested inside other files. Many Forth programmers compile big applications by compiling a control file which just includes other files. On many systems you can find the source code of a word by typing:

LOCATE <name>

You can also often see what was compiled using one of the following:

```
DIS <name>
SEE <name>
VIEW <name>
```

Suppose we had a section of a program that had to greet people. First, we could define a word to say 'hello'. We use a dot at the beginning of the name because it is a Forth convention that words which print start with a dot. We use **:** to start a definition (followed by its name) and **;** to end it.

```
: .HELLO  \ -- ; has no effect on the stack
  ." Hello "  ;
```

If you now type **.HELLO** <ENTER>, Forth will respond, followed by `ok' to show that there were no errors in the last entry. We now need some words to print out the names of the people we want to greet.

```
( -- )
: .FRED
  ." Fred "  ;
: .MARY
  ." Mary "  ;
: .NEIL
  ." Neil "  ;
: .LINDA
  ." Linda "  ;
```

We will also need a word to link these together.

```
: .AND          \ --
  ." and "  ;
```

We can now define a word to greet all these people. Forth words can occupy as many lines as are needed. You can use line breaks and additional spaces to emphasize the phrasing of the word.

```
: .GREET \ --
  .HELLO .MARY .AND .FRED
  .AND .LINDA .AND .NEIL  CR  ;
```

When you type **`.GREET <ENTER>`**, Forth will respond -

```
Hello Mary and Fred and Linda and Neil ok
```

At some stage you will want to see what words are in the dictionary, to do this enter:

```
WORDS
```

You will see a long list of words roll past. You can stop the listing by pressing the space bar. Press it again and the listing will continue. Press any other key, and the listing will finish. All the names you see are the names of predefined words in Forth, plus any that you have created. All these words are available for you to use, and the predefined ones are documented later in this manual. The words that you write will use these words as their basis.

The secret of writing programs in Forth is to keep everything simple. Remember the KISS method (keep it simple, stupid). Simple things work, and complicated things can be built out of simple things. A programmer's job is to decide what those simple things should be, and then to design and code them. If the names of the words reflect what they are to do, then the code will be readable and easy to follow. The next sections give an introduction to the components of Forth, and a description of the program control structures available.

5 Components of Forth

Only the words needed for this book are documented here. Most Forths have many more. An HTML version of the last publicly available ANS Forth document, which is very close to the final standard, is provided on the CD supplied with this book. Start with DPANS.HTM.

Data stack operations

Learning any new language involves some grunt work. In Forth, much of this grunt work involves learning how to manipulate items on the data stack. To quote Elizabeth Rather:

"Development of good stack management skills is a core component of Forth practice, and a key to enjoying its benefits. If you regard the stack as a nuisance or an impediment, you're missing the whole point."

By convention the top item of the data stack is called TOS and second item is called NOS. You may also see the terms 3OS and 4OS in some code.

DUP `\ x -- x x`

DUPlicate the top stack item.

?DUP `\ x -- 0 | x x`

DUPplicate the top stack item only if it is non-zero. Nearly always used before a conditional branch.

DROP `\ x --`

Discard the top data stack item and promote NOS to TOS.

SWAP `\ x1 x2 -- x2 x1`

Exchange the top two data stack items.

OVER `\ x1 x2 -- x1 x2 x1`

Make a copy of the second item on the stack.

NIP `\ x1 x2 -- x2`

Dispose of the second item on the data stack.

TUCK `\ x1 x2 -- x2 x1 x2`

Insert a copy of the top data stack item underneath the current second item. Equivalent to **SWAP OVER**.

ROT `\ n1 n2 n3 -- n2 n3 n1`

ROTate the positions of the top three stack items such that the current top of stack becomes the second item.

-ROT `\ x1 x2 x3 -- x3 x1 x2`

The reciprocal of **ROT**. Non ANS, but widely available.

PICK `\ xu .. x0 u -- xu .. x0 xu`

Get a copy of the Nth data stack item and place on top of stack. **`0 PICK`** is equivalent to **DUP**, **`1 PICK`** to **OVER** and so on.

2DUP `\ x1 x2 -- x1 x2 x1 x2`

DUPlicate the top cell-pair on the data stack.)

2DROP `\ x1 x2 --`

Discard the top two data stack items.

2SWAP `\ x1 x2 x3 x4 -- x3 x4 x1 x2`

Exchange the top two cell-pairs on the data stack.

2OVER `\ x1 x2 x3 x4 -- x1 x2 x3 x4 x1 x2`

Copy cell pair x1 x2 to the top of the stack.

Return Stack operations

>R `\ x -- ; R: -- x`

Push the current top item of the data stack onto the top of the return stack.

R@ `\ -- x ; R: x -- x`

Copy the top item of the return stack and place on the data stack.

R> `\ --x ; R: x --`

Pop the top item off the return stack and place on the data stack.

2>R `\ x1 x2 -- ; R: -- x1 x2`

Push the current top cell pair from the data stack onto the return stack.

2R@ `\ -- x1 x2 ; R: x1 x2 -- x1 x2`

Copy the top cell pair of the return stack and place on the data stack.

2R> `\ -- x1 x2 ; R: x1 x2 --`

Pop the top cell pair from the return stack and place on the data stack.

Maths operations

+ `\ n1|u1 n2|u2 -- n3|u3`

Add two single precision integer numbers: n3=n1+n2.

- `\ n1|u1 n2|u2 -- n3|u3`

Subtract two single precision integer numbers: n3=n1-n2.

***** `\ n1 n2 -- n3`

Standard signed multiply: n3 = n1 * n2.

/ `\ n1 n2 -- n3`

Standard signed division operator: n3 = n1/n2.

MOD `\ n1 n2 -- n3`

Standard signed division operator returning the remainder: n3 = n1 mod n2.

UM* `\ u1 u2 -- ud3`

An unsigned multiply that produces an unsigned double result.

***/** `\ n1 n2 n3 -- n4`

Multiply n1 by n2 to give a double precision result, and then divide it by n3 returning the quotient. The point of this operation is to avoid loss of precision. The ANS standard permits systems to restrict n3 to positive numbers. This word is ideal for scaling operations, e.g. after reading an analogue to digital converter or converting radians to degrees.

```
: PI      \ n -- n*pi
  355 113 */
;
```

UM/MOD `\ ud u -- urem uquot`

Perform unsigned division of double number ud by single number u and return the remainder and quotient. In most CPU architectures that have divide operations, the remainder is produced at the same time as the quotient. Note that the dividend is a double number. Often used in conjunction with **UM*** for scaling operations that take advantage of an intermediate double result to preserve precision.

NEGATE `\ n1 -- n2`

Negate a single precision integer number.

ABS `\ n -- u`

If n is negative, return its positive equivalent (absolute value).

MIN `\ n1 n2 -- n1|n2`

Return the lesser of n1 and n2.

MAX `\ n1 n2 -- n1|n2`

Return the greater of n1 and n2.

Comparisons

In Forth there is no basic boolean type. The stack value 0 (all bits clear) is taken to represent FALSE, while any other value is taken to represent TRUE. The stack notation flag indicates a "well formed flag" which means that a TRUE value is represented by all bits set, which corresponds to the number **-1** for most CPUs.

< `\ n1 n2 -- flag`

Returns true if n1<n2

> `\ n1 n2 -- flag`

Returns true if n1>n2

U< `\ u1 u2 -- flag`

Unsigned, returns true if u1<u2

U> `\ u1 u2 -- flag`

Unsigned, returns true if u1>u2

= `\ n1 n2 -- flag`

Returns true if n1=n2

<> `\ n1 n2 -- flag`

Returns true if n1 not equal to n2

0= `\ x1 -- flag`

Returns true if x1=0.

0<> `\ x1 -- flag`

Returns true if x1 is not equal to 0.

WITHIN `\ n1|u1 n2|u2 n3|u3 -- flag`

Return TRUE if n2|u2 <= n1|u1 < n3|u3. Note the conditions. This word uses unsigned arithmetic, so that signed compares are treated as existing on a number circle.

Widely available

<= `\ n1 n2 -- flag`

Returns true if n1<=n2

>= `\ n1 n2 -- flag`

Returns true if n1>=n2

Bitwise logic

AND `\ n1 n2 -- n3`

Returns n3 = n1 AND n2

OR `\ n1 n2 -- n3`

Returns n3 = n1 OR n2

XOR `\ n1 n2 -- n3`

Returns n3 = n1 XOR n2

INVERT `\ n1 -- n2`

Returns bitwise inverse of n1.

LSHIFT `\ x1 u -- x2`

Logically shift x1 by u bits left.

RSHIFT `\ x1 u -- x2`

Logically shift x1 by u bits right.

Memory operations

In Forth terminology items in memory that are the same size as items on the data stack are called cells. The width of the stack is usually determined by the underlying CPU architecture, being 16 bits on 16 bit CPUs, 32 bits on 32 bit CPUs and so on. For byte-addressed CPUs (the majority), a character is a byte and bytes in an array are stored at consecutive addresses.

@ `\ addr -- n`

Fetch and return the cell at memory address addr.

! `\ n addr --`

Store the cell quantity n at memory address addr.

+! `\ n addr --`

Add n to the cell at memory address addr.

c@ `\ addr -- char`

Fetch and zero extend the character at memory address addr.

c! `\ char addr --`

Store the character char at memory address addr.

CELLS `\ n1 -- n2`

Returns n2, the memory size required to hold n1 cells. **CELLS** improves portability and readability.

CHARS `\ n1 -- n2`

Returns n2, the memory size required to hold n1 characters. **CHARS** improves portability and readability.

Widely available

w@ `\ addr -- val`

Fetch and zero extend the 16 bit item at memory address addr. Usually only found on 32 and 64 bit systems.

w! `\ val addr --`

Store the 16 bit item val at memory address addr. Usually only found on 32 and 64 bit systems.

Constants and variables

Many times in a program we need to use a value to represent something, a type of flower, or a bus route number. On other occasions the value corresponds to an actual value rather than an association, the price of roses today, the ASCII code for a special key. Some of this data never changes; it is constant. Other data changes from day to day or minute to minute. The two types of data are created by **CONSTANT**, **VARIABLE** and **VALUE**. Naming data makes programs easier to write and read, as people remember names more easily than numbers.

CONSTANTs

Constants are used when the data will not change, or will only be changed when the programmer edits the program (for instance, to change a control key). Constants return their value to the stack.

```
DECIMAL          \ all numbers are in decimal
( -- n )
13 CONSTANT ENTER-KEY
1 CONSTANT DAFFODIL
2 CONSTANT TULIP
3 CONSTANT ROSE
4 CONSTANT SNOWDROP
```

VARIABLEs

In Forth a variable is named and set up by the word **VARIABLE**.

```
( -- addr )
VARIABLE FLOWER
```

At this stage a variable called **FLOWER** has been declared with an initial value of 0 on most systems. The ANS standard does not mandate a specific initial value. When you use **FLOWER** the address of the data is given. Forth uses **@** ('fetch') to fetch the data from the address, and **!** ('store') to store data into an address. If a specific initial value is needed we can change it.

```
DAFFODIL FLOWER !
```

Later on in a program we can use the value of **FLOWER** to change the way the program acts. Part of the program might be about to draw the flower, and we need to set the colour properly.

```
: DAFF    \ --
  FLOWER @ DAFFODIL =
  IF  YELLOW INK  THEN   ;
```

To make a program wait until the ENTER key is pressed, you can try the code below.

```
: WAIT-ENTER     \ -- ; wait for <CR>
  BEGIN  KEY ENTER-KEY =  UNTIL  ;
```

In most words data is passed from word to word using the stack. If you find the stack usage getting too complex, try splitting the word into other words which only use one or two items on the stack. On other occasions you will find that all the words need to refer to the same value which controls what happens on this run of the program (say, the type of flower). In these cases the use of a variable is appropriate. As your Forth skills improve, you will find that you use fewer variables.

VALUEs

The word **VALUE** creates a variable that returns its contents (value) when referenced, and is widely used for variables that are mostly read and rarely changed. A **VALUE** is defined with an initial value.

```
55 VALUE FOO
```

In embedded systems, you can rely on a **VALUE** being initialised at power up, whereas the ANS specification does not require **VARIABLEs** to be initialised. You get the contents of a VALUE by referring to it, but you set it by prefixing it with the word **TO**.

```
: test          \ n --
  cr ." The value of FOO was " foo .
  to foo
  ." and is now " foo .
;
```

Control structures

A control structure is one that allows you to control the way a program behaves depending on the value of some piece of data you have previously calculated. This gives a program the power of choosing to do this if one thing happens, or that if another thing happens. Control structures in Forth allow execution and looping

determined by values on the stack. Although the user is not necessarily aware of it, control structures are usually implemented by means of words that execute at compile time (immediate words), and compile other words that actually execute at run time. Techniques like these allow error checking to be implemented as well. Control structures must be used inside a colon definition; they cannot be directly executed from the keyboard. Any one structure must be written entirely within one definition; you cannot put the **IF** in one word and the **THEN** in another. Control structures can be nested inside one another, but they must not overlap.

Although it is possible to write loops and control structures that return a variable number of items on the stack, this practice leads to errors and is not recommended.

IF ... THEN

```
flag IF <true words> THEN
```

The flag on the top of the stack controls execution. If the flag is non-zero (true), the words between **IF** and **THEN** are executed, otherwise they are not.

Like most Forth words **IF** consumes the flag used as input to it. If the value of the flag must be used again it can be duplicated by **DUP**. In the case of **IF** .... **THEN**, where the value may only be needed between **IF** and **THEN**, the use of **?DUP**, which only duplicates a number if it is non-zero, may be more appropriate.

```
: TEST          \ flag --
  IF  ." top of stack is non-zero "  THEN  ;
1 TEST top of stack is non-zero ok
0 TEST ok
```

IF...ELSE...THEN

```
flag IF <true words> ELSE <false words> THEN
```

This structure behaves just like **IF** ... **THEN** above except that an alternate set of words will execute when the flag is false (zero).

```
: TEST          \ flag
  IF  ." top of stack is non-zero "
  ELSE  ." top of stack is zero "
  THEN
;
1 TEST top of stack is non-zero ok
0 TEST top of stack is zero ok
```

EXIT

EXIT causes a return from the current definition. Before using **EXIT** you must ensure that the data stack is correct and that anything you put on the return stack with **>R** has been removed. **EXIT** is often used when an error check fails.

```
: FOO          \ a b c – result
  TEST1
  IF  DROP 2DROP  <errcode1> EXIT  THEN
  TEST2
  IF  DROP 2DROP  <errcode2> EXIT  THEN
  ...
  0                       \ for success
;
```

DO ... LOOP and DO ... n +LOOP

```
limit index DO <loop words> LOOP

limit index DO <loop words> <increment> +LOOP
```

This structure is very roughly the same as BASIC's FOR X=1 TO 10....NEXT.

To use this structure, place the limit value and the starting value of the loop index on the stack. **DO** will consume this data and transfer it to the return stack for use during execution of the loop. **LOOP** will add one to the index and compare it to the limit. If the index is still less than the limit the loop will be executed again. As a result of this the limit value is never used. **+LOOP** behaves in the same way except that the increment on the stack is added to the index.

You can get out of the loop early with the words **LEAVE** or **?LEAVE**.

LEAVE (--) cleans up the return stack, and execution then resumes after the **LOOP** or **+LOOP**. Note that words between **LEAVE** and **LOOP** or **+LOOP** are not executed.

?LEAVE behaves like **LEAVE** except that the loop is left only if the top of the stack is non-zero. This is a useful word when checking for errors. This word is non-ANS but is widely available. It is equivalent to **IF LEAVE THEN**.

DO ... LOOP structures may be nested to any level up to the capacity of the return stack. The index of the current loop is inspected using **I (-- n)**. The index of the next outer loop is inspected using **J (-- n)**.

If you need to terminate the loop early and get out of the current word by using **EXIT** (see above) the code will not pass through **LOOP** or **+LOOP**. In this case you must remove the loop control information using **UNLOOP (--)**. The following word polls the keyboard every 100 milliseconds for one second. It returns a key code immediately if a key is pressed, or returns a zero after one second if no key has been pressed.

```
: TESTKEY      \ -- char|0
  10 0 DO            \ ten times
    KEY? IF          \ true if char
      KEY            \ get character
      UNLOOP EXIT    \ clean up and get out
    THEN
    100 MS           \ wait
  LOOP
  0                  \ no key
;
```

WATCH OUT -

If you use the return stack for temporary storage after **DO**, you must remove it before **LOOP** or **+LOOP**.

If there is data on the return stack **I** and **J** will return incorrect values.

The loop will always be executed at least once. If you do not want this to happen you must add extra code around the loop or use **?DO** , which only executes if the limit is not the same as the index.

Example:

```
: TEST          \ -- ; display numbers 1..9
  10 1 DO  I .  LOOP  ;
TEST<cr> 1 2 3 4 5 6 7 8 9 ok
: TEST2
  10 1 DO  I .   3 +LOOP  ;
TEST2<cr> 1 4 7 ok
```

WATCH OUT -

```
: TEST3         \ -- ; display 9..0 in descending order
  0 9 DO  I .  -1 +LOOP  ;
TEST3<cr>9 8 7 6 5 4 3 2 1
```

When the loop increment is negative, **DO…+LOOP** will include an iteration with the limit. This occurs because the loop test is dependent on the CPU overflow flag.

?DO ... LOOP and ?DO ... n +LOOP

```
limit index ?DO <loop words> LOOP
limit index ?DO <loop words> increment +LOOP
```

These structures behave in the same way as **DO** ... **LOOP** and **DO** ... **n +LOOP** except the loop is not executed at all if the index and the limit are equal on entry. For example if the word **SPACES** which prints n spaces is defined as:

```
: SPACES  \ n --
  0 DO  SPACE  LOOP  ;
```

the phrase **0 SPACES** causes 65536 spaces to be displayed on a 16 bit system (2^32 spaces on a 32 bit system), whereas the definition below displays no spaces.

```
: SPACES  \ n --
  0 ?DO  SPACE  LOOP  ;
```

BEGIN ... AGAIN

```
BEGIN <words> AGAIN
```

This structure forms a loop that only finishes if an error condition occurs, or a word such as **THROW**, **ABORT** or **QUIT** (see later) is executed. The first example will read and echo characters from the keyboard forever, the second will exit when the ENTER key is pressed.

Example:

```
: TEST          \ -- ; runs forever
  BEGIN  KEY EMIT  AGAIN  ;

DECIMAL
: TEST2         \ -- ; runs until CR pressed
  BEGIN  KEY DUP 13 =
         IF  ABORT  THEN       \ could use EXIT
         EMIT
  AGAIN  ;
```

BEGIN ... UNTIL

```
BEGIN <words> flag UNTIL
```

This structure forms a loop which is always executed at least once, and exits when the word **UNTIL** is executed and the flag (on the data stack) is true (non-zero). If you

need to use the terminating condition after the loop has finished use **?DUP** to duplicate the top item of the stack if it is non-zero.

BEGIN ... UNTIL loops may be nested to any level.

Example:

```
: TEST
  BEGIN  KEY DUP EMIT 13 =  UNTIL  ;
```

BEGIN ... WHILE ... REPEAT

```
BEGIN <test words> flag WHILE <more words> REPEAT
```

This is the most powerful and perhaps the most elegant (though certainly not some purists' choice) of the Forth control structures. The loop starts at **BEGIN** and all the words are executed as far as **WHILE**. If the flag on the data stack is non-zero the words between **WHILE** and **REPEAT** are executed, and the cycle repeats again with the words after **BEGIN**.

This structure allows for extremely flexible loops, and perhaps because it is somewhat different from the structures of BASIC or Pascal, this structure is often somewhat neglected. It does however, repay examination. Like all the structures it can be nested to any depth, limited only by stack depth considerations. In the example below, the console is polled until a key is pressed, and a counter is incremented while waiting.

```
VARIABLE COUNTER
: TEST           \ --
  0 COUNTER !
  BEGIN  KEY? 0=
    WHILE  1 COUNTER +!
  REPEAT  ;
```

CASE ... OF ... ENDOF ... ENDCASE

```
CASE key
  value1 OF <words> ENDOF
  value2 OF <words> ENDOF
  ...
  <default words> ( otherwise clause )
ENDCASE
```

This is the most common **CASE** statement in Forth. It is the result of a competition for the best **CASE** statement. The competition was run by Forth Dimensions, the journal of the Forth Interest Group (FIG) in the USA. A large number of **CASE** statements were proposed, but this one has stood the test of time as it is secure, easy to use and understand, and easy to read. It was invented by Dr. Charles E. Eaker, and first published in Forth Dimensions, Vol. II number 3, page 37.

CASE statements exist to replace a large chain of nested **IF**s, **ELSE**s, and **THEN**s. Such chains are unwieldy to write, prone to error, and lead to severe brain-strain.

The function of a **CASE** statement is to perform one action dependent on the value of the key passed to it. If none of the conditions is met, a default action (the otherwise clause) should be available. Note that a value to select against must be available before each OF against which the entered parameter may be tested. The select value is top of stack, the parameter is next on stack (by requirement); **OF** then compares the two values, and if they are equal, the words between **OF** and **ENDOF** are executed, and the program continues immediately after **ENDCASE**. If the test fails, the code between **OF**

and **ENDOF** is skipped, so that the select value before the next **OF** may be tested. If all the tests fail the parameter is still on the data stack for the default action, and is then consumed by **ENDCASE**. Additional control structures can be used inside **OF** ... **ENDOF** clauses.

Example:

```
: STYLE? ( n -- )
  CASE
    1  OF  ." Mummy, I like you"  ENDOF
    2  OF  ." Pleased to meet you"  ENDOF
    3  OF  ." Hi!"  ENDOF
    4  OF  ." Hello"  ENDOF
    5  OF  ." Where's the coffee"  ENDOF
    6  OF  ." Yes?"  ENDOF
        ." And who are you?"
  ENDCASE  ;
```

The phrase:

```
n STYLE?
```

will select an opening phrase according to the value of n. If n is in the range 1..6 a predefined string is output, for any other number the default phrase 'And who are you?' is output. Case statements are often used to select actions based on ASCII characters. In an editor sections of code like the one below are often found.

```
CASE
  KEY
  [CHAR] I  OF  INDEX  ENDOF
  [CHAR] M  OF  MOVE-BLOCK  ENDOF
  [CHAR] D  OF  DIRECTORY  ENDOF
  ...
ENDCASE
```

MPEisms: CASE extensions

In order to extend the usefulness of the basic **CASE** structure we have added three extensions to the standard Eaker **CASE**.

The first **?OF** allows the use of a logical test rather than equality. If you need to test whether or not a character is in the right range the following replaces a large number of **OF** ... **ENDOF** sets. The word **WITHIN?** returns a true value if the value is between (or equal to) the lower and upper limits. **?OF** consumes the flag given to it.

```
DUP  32 127 WITHIN? ?OF ...  ENDOF
```

The second extension **END-CASE** behaves just like **ENDCASE**, except that it does not **DROP** anything from the stack, so allowing a default clause to consume the select value without having to **DUP**licate it.

The third extension **NEXT-CASE** compiles a branch back to the **CASE**, so producing a loop that exits via one the **OF ... ENDOF** or **?OF ... ENDOF** phrases. Such a loop performs a different exit action for each condition. The intention of this structure is to allow a formal method of constructing loops with more than one exit and exit action. Such loops are often necessary when dealing with text entry. **NEXT-CASE** consumes no data.

Example:

```
CASE  KEY
  13  OF  <cr action>  ENDOF
  10  OF  <lf action>  ENDOF
  DUP 32 127 WITHIN?
      ?OF  <normal action>  ENDOF
      CR ." Character code " .  ." is invalid" CR
NEXT-CASE
```

Restarts and errors

Exception handling is covered in more detail in a separate chapter. The **`BEGIN ... AGAIN`** example above uses **`ABORT`**, which usually performs a warm restart of the system, but is implementation dependent. Conventionally, **`ABORT`** resets the stacks and runs the Forth interpreter loop in **`QUIT`**.

Text and strings

There are not many words in a standard Forth that deal with strings, but they do allow the user to build words that will perform any required function. Two types of strings are defined, which I shall call **counted** and **character**. Forth systems interfacing to operating systems and C libraries also provide words to use **zero-terminated** strings. Applications that operate on large blocks of text and/or are internationalised use **wide** or **Unicode** strings as well.

Historically, Forth systems mainly used counted strings, which consist of a count byte followed by that many characters. They are referenced by the address of the count byte **`( -- caddr )`**. This is the same structure as used by many Pascal implementations. Counted strings are fast to manipulate because their length is always available. However counted strings are difficult to use when parsing and processing larger blocks of text.

Because of this limitation it became common practice to refer to them by address/length pairs on the stack **`( -- caddr len )`**, where caddr refers to a character address. The ANS committee decided to standardise on these character strings. Because of widespread past practice, it was impossible to eliminate counted strings, nor is it desirable to do so, as counted strings are a very convenient storage format.

Although there was some pressure to standardise zero terminated strings, their inefficiencies (especially for embedded systems) and lack of compatibility with memory operations lead to the rejection of this pressure.

It should be noted that many Forth programmers, just like programmers in other languages, have become accustomed to the assumptions that a character is a unit of 8 bits (an ‘octet’), that a byte is an 8 bit unit, and that memory is addressed in 8 bit units. This is especially true of programmers whose first language is English, and such programmers are also often only used to the ASCII 7 bit character set. Nowadays, even the 16 bit Unicode set (UTF-16) is inadequate for all languages. The CD accompanying this book includes the draft ANS Forth proposals for handling wide characters and internationalisation.

Counted strings

A counted string is stored as a count byte followed by that many characters. To put a string into a word use C" which compiles a string into the dictionary, and returns its address when the word executes.

```
: HELLO$ \ -- caddr
  C" Hello there "  ;
```

The string will start with the 'H' and end with the space before the quotation mark. The address returned by **HELLO$** points to the count byte. To convert this address to the address of the first character and the number of characters, the word **COUNT** is used.

When a string is to be printed, it is usually done by the word **TYPE**, which needs the address of the text to be printed and the number of characters to be printed. To print the string above we would use:

```
HELLO$ COUNT TYPE
```

To pick individual characters out of a string you simply add the number of the required character to the start address and fetch it. For the first 'l':

```
HELLO$ 3 + C@ EMIT
```

To print the sequence 'lo t':

```
HELLO$ 4 + 4 TYPE
```

Character Strings

A character string in memory is described by its start address and length.

```
: HELLO$ \ -- caddr
  C" Hello there "  ;
```

Text and String input

The principal word to fetch strings from the keyboard is **ACCEPT** which requires the address of a buffer in which to put the string, the maximum number of characters to read, and returns the number of characters read. To create an 80 byte buffer, read a string into it, and inspect the contents of the buffer, we create a buffer called **BUFFER$** and then reserve another 78 bytes (the variable reserves two), read 80 bytes into it, and then display the contents of **BUFFER$**.

```
CREATE BUFFER$  80 ALLOT
CR  BUFFER$ 80 ACCEPT
( now enter a string at the keyboard )
BUFFER$ SWAP DUMP
```

You will see that **BUFFER$** contains the text you entered without a count byte. Inside Forth there is an area called the terminal input buffer (its address is returned by the word **TIB**). The word **QUERY** reads a line of text into **TIB**. The word **WORD** then extracts a sub-string bounded by a specified character, and copies the string to the end of the dictionary as a counted string (count byte + characters), returning the address at which it left the string. Try entering the word below:

```
: T      \ just another test word
  BL WORD 40 DUMP  ;
T xxxx yyyy zzzz
```

ANS Forth specifies a buffer called **PAD (-- addr)** which is available for application use. Many Forth systems (even some nominally ANS compliant) use **PAD** internally. If you use **PAD** and see inconsistent results recheck your code with a buffer other than **PAD**.

Print Formatting

Forth has a very powerful number string formatting system. It is quite different from that supplied with C or BASIC. To print a number as pounds and pence (or dollars and cents, or francs and centimes), try the following:

```
HEX
: .POUNDS \ u --
  [CHAR] £ EMIT
  S>D  <# # # ASCII . HOLD #S #>  TYPE
;
DECIMAL
5050 .POUNDS
```

The phrase **[CHAR] £ EMIT** prints the pound sign. Number conversion is started by **<#** and finished by **#>**. The word **<#** needs a double number (**S>D** converts single numbers to double). Numeric conversion then proceeds LEAST SIGNIFICANT DIGIT first. For each **#** a digit is converted. The word **HOLD** takes a character and inserts it into the character string being generated. So the phrase

```
 # # ASCII . HOLD
```

produces the two least significant digits and a decimal point (the pence portion). The word **#S** converts the rest of the number, producing at least one digit. Numeric conversion is finished by **#>** which leaves the address of the generated string, and the number of bytes in the string. **TYPE** then prints the string.

You can easily produce your own number conversion formats. Suppose we wanted to print numbers as pounds and pence, with six figures before the decimal point, two after it, and leading zeros suppressed except in the character before the decimal point. The format we want is 'xxxxxy.yy' where x is a digit or a blank and y is a digit.

We need to generate a word **#B** which produces a digit if possible or a blank, if number conversion has finished.

In between **<#** and **#>** the number being converted is in double number form. When a digit is converted by **#** it is divided by the current base, the remainder is converted to a character to be output, and the quotient is returned. Thus the function of **#B** is to allow numeric conversion if the number is non-zero, otherwise to insert a blank into the output.

```
: D0=            ( d -- t/f )
  OR 0=  ;

: #B             ( d1 -- d2 )
  2DUP D0=
  IF  BL HOLD
  ELSE  #
  THEN  ;
```

The word **BL** returns the character code for a space. Now we can generate **.P** which will print a double number as eight characters with a decimal point.

```
: .P             \ d1 --
  <# # # ASCII . HOLD # #B #B #B #B #B #>
  TYPE  ;
500.00 .P
```

When you enter a number that includes a dot, it is treated as double number. The position of the comma has no significance, except that the number of digits after it is held in the variable **DPL** in many systems. Try:

```
5,0000 .P  DPL @ .
500.00 .P  DPL @ .
```

Both numbers are converted to the same value, but the system variable **DPL** tells you how many digits were entered after the dot. If no dot had been entered, the number would have been treated as a single number, and the value of **DPL** would have been -1.

If you want to explore print formatting more fully, and gain yourself fame for the more exotic style of programming, try formatting time and date from a 32 bit number of seconds. Changing the number conversion base in the middle of print formatting words can be very useful.

MPEism: *Because the '.' and ',' characters are both used as number separators for currency in Europe, MPE Forths default to using ',' in a number to indicate a double number. This also avoids confusion with floating point numbers. The default can be changed to the ANS standard.*

Vocabularies

In constructing an application written in Forth a large number of new words are generated. With the already large number provided in the nucleus, problems can arise in simply remembering what the words mean. Other problems can arise because of name reuse. In one part of an application, **ATTACH** may refer to attaching an interrupt to a piece of code. In another section, it may refer to an operation on a file, and again may refer elsewhere to the application performed by the robot under control.

A method is needed to control the environment in which we are working. In the examples above the word **ATTACH** has different meanings in different contexts. The context in which a word exists is its vocabulary. Technically, vocabularies are usually defined in terms of **wordlists**, but since the vast majority of Forths support vocabularies for user convenience, I have chosen to discuss vocabularies first.

Vocabularies are created by the phrase:

```
VOCABULARY <vocabulary-name>
```

For example the vocabulary of words dealing with the robot might well be called **ROBOTICS**, and would be defined by:

```
VOCABULARY ROBOTICS
```

The Forth system is told which vocabulary words are defined into by the word **DEFINITIONS**, which sets the vocabulary that words are currently to be defined in. After the phrase:

```
ROBOTICS DEFINITIONS
```

all new words will be part of the **ROBOTICS** vocabulary, and you could list all the words in that vocabulary by typing:

```
ROBOTICS WORDS
```

Having defined which vocabulary new words are built into, we must now define which contexts are relevant when searching for word names. For example, moving a robot arm might need access to floating point words in vocabulary **F-PACK**, the graphics

words in **GRAPH** for console displays, and the multi-tasking words in **TASKING**. To cope with all this we also need a way to start at the beginning again.

The word that means 'search the minimum' is **ONLY** which sets Forth to search the minimum, often only a tiny vocabulary called **ROOT**. Most of the common words are in **FORTH** which is the main vocabulary. Thus the phrase:

```
ONLY FORTH
```

resets the system to only use **FORTH** (and the little **ROOT**). We can add another vocabulary to be searched with the word **ALSO** which adds the new vocabulary so that its searched first. After executing:

```
ALSO F-PACK
```

the **F-PACK** vocabulary will be searched first, then Forth, and finally **ROOT**. To provide the complex order described earlier, the following phrase can be used:

```
ONLY FORTH
ALSO F-PACK  ALSO GRAPH  ALSO TASKING
ALSO ROBOTICS DEFINITIONS
```

It is usual for the vocabulary into which words are defined to be the first in the search order, and so the last one specified. When the **ROBOTICS** vocabulary is being defined into, it is most likely that other words in the same **ROBOTICS** context will be required, and if duplicate names exist, it is the ones in the **ROBOTICS** context that are most likely to be needed.

In many Forths you can get a list of all the vocabularies that have been defined by typing:

```
VOCS
```

and you can see the search order used by typing:

```
ORDER
```

The following Forth words are involved in vocabulary control, and they will all be documented in the glossary for your Forth:

```
CONTEXT CURRENT         \ pointers
FORTH ROOT              \ vocabularies
VOCS ORDER WORDS        \ display words
ONLY ALSO DEFINITIONS PREVIOUS
```

Wordlists

When a word is searched for by name, the name is part of a **wordlist**. Conceptually the list is single linked list of names, with the last word added being looked at first. This means that when a new definition has the same name as a previous one, the new one is found rather than the old one.

How wordlists are implemented is often rather different. Typically, the name is hashed to produce an index into several sub-lists. The number of sub-lists (usually called threads) varies quite widely. Other systems generate dynamic hash-tables or use databases. Whatever internal mechanisms are used, an ANS Forth system will follow the model above as the user sees it.

A new wordlist is created by the word **WORDLIST (-- wid)** which returns a **wid** or wordlist-identifier. You cannot assume anything about the meaning of a wid

nor use it except where a wid is required, otherwise you are making assumptions about the internal structures in the Forth system.

WORDLIST is provided to allow you to create and manipulate the search order for words: it is there for toolmakers. It is often used to provide a sealed environment for user commands, to hide groups of words in large projects, or to hide methods in object oriented extensions. When used like this, the most important word used is

SEARCH-WORDLIST `c-addr u wid -- 0 | xt 1 | xt –1`

> Find the definition identified by the string c-addr u in the word list identified by wid. If the definition is not found, return zero. If the definition is found, return its execution token xt and one (1) if the definition is immediate, minus-one (-1) otherwise.

The word **FIND** is used by the text interpreter to look for a word in all the words in the **CONTEXT** search order, which is usually a list of wordlists and vocabularies.

FIND `c-addr -- c-addr 0|xt 1|xt -1`

> Perform the SEARCH-WORDLIST operation on all wordlists within the current search order. This definition takes a counted string rather than a c-addr u pair. The counted string is returned as well as the 0 on failure.

An application of **SEARCH-WORDLIST** is provided in the chapter “Using the Forth interpreter”. Further use of wordlists will require the use of the following words defined in the ANS standard: **SET-CURRENT SET-CONTEXT GET-CURRENT GET-CONTEXT**.

6 Example: Date validation

Date Validation: the requirement

It is required to validate a date which is entered in the form:

```
dd mm yyyy
```

Validation should ensure the following:

- Year is valid in the range 1752 to 2050 inclusive.
- Month is valid in the range 1 to 12 inclusive.
- Day is valid in the range 1 to 28,29,30 or 31 depending upon the entered month and year

A leap year is defined as a year which is exactly divisible by 4 if it is not divisible by 100, but exactly divisible by 400 if it is. The expression defining a leap year is therefore:

```
((year mod 4)=0 and (year mod 100)<>0)
or (year mod 400)=0
```

A Forth word **VALID?** is required to meet this need. The word **VALID?** should expect day, month and year and return true (-1) if the date is valid and false (O) if it is not. It may be assumed that the inputs to the word will be three integers in the range -32768 to 32767 inclusive.

The stack description for **VALID?** is thus:

```
VALID?  \ day month year -- t|f
```

Designing the solution

In order to solve this problem, it is necessary to break it down into several sub-problems. This is the most natural way of problem solving and technique carries forward into computer programming under the name `Top-Down Design'. Forth is particularly suitable for this since problems can be broken down into very small reusable parts: Words.

In this case we note that the problem comprises three main parts. These are:

1) Validation of the year: a word called **YEAR?**
2) Validation of the month: a word called **MONTH?**
3) Validation of the day: a word called **DAY?**

We will consider each of these in turn:

Both the validation of the year and the validation of the month can be considered in isolation and we can code these easily. The validation of the day, however, is more complex and this will need to be considered more carefully given the day number and the year.

Coding the solution

For the year, we are required to check whether the number supplied is in the range 1752-2050. Our first attempt might be:

```
: year?          \ year -- t|f
  dup 1752 >=swap 2050 <= and
;
```

However, we can use the word **WITHIN?** to perform the range test. **WITHIN?** has the following stack effect:

```
WITHIN?          \ n1 n2 n3 -- t|f
```

It returns true if the following condition holds:

```
n2 <= n1 <= n3
```

If your Forth system does not have **WITHIN?** You can define it as follows using the ANS word **WITHIN**.

```
: WITHIN?        \ x n1 n2 -- flag
  1+ WITHIN
;
```

Our definitions of **YEAR?** and **MONTH?** are therefore:

```
: year?          \ year -- t/f
  1752 2050 within?  ;

: month?         \ month -- t/f
  1 12 within?  ;
```

Each word is suffixed with a question mark. This implies that the word will return a value of true and false. It is a naming convention.

Our definition of **DAY?** is not so simple, since a valid day is both dependent upon the month and year in which it occurs (`thirty days have September' etc.) The maximum number of valid days differs depending on the month and, in the case of February, on the year. Our definition of **DAY?** is therefore:

```
: day?           \ day year month -- t|f
  >days  1 swap within? ;
```

This calls a word we have not yet defined. In the top-down design path, this word must logically come next, but because we are writing Forth, we must define this new word first. The word **>DAYS** (pronounced "to days") will convert the given month and year into the maximum number of valid days for that combination. This value is then used as the upper bound for the **WITHIN?** test.

The definition of **>DAYS** will be in two parts: that which deals with February and leap years and that which deals with all other months. This is shown below:

```
: >days          \ year month -- maxdays
  dup 2 =
  if   drop >leapdays
  else nip >otherdays
  then
;
```

In the case of February we return a value depending on the year supplied, otherwise we can discard the year and just return a value based upon the month. The definition of **>OTHERDAYS** is:

```
: >otherdays     \ month -- maxdays
  dup  4 =
  over  6 = or
  over  9 = or
  swap 11 = or
  if  30  else  31  then
;
```

This word involves a lot of stack manipulation to keep the one item (the month) on the top of the stack for testing. More elegant solutions exist, but we will come to them later.

The word **>LEAPDAYS** must take the year, determine if it is a leap year and return either 28 or 29 accordingly. Our definition is therefore:

```
: >leapdays      \ year -- maxdays
  leap?
  if 29 else 28 then
;
```

Given the definition of a leap year (see earlier), our definition of **LEAP?** would be:

```
: leap?          \ year -- t/f
  dup 4 mod 0=
  over 100 mod 0<>
  and
  swap 400 mod 0=
  or ;
```

We now have all the sub-words needed to validate a given date. They may now be put together to make the required word.

Putting it all together

Our top level definition **VALID?** for validating a date will use the words we have just defined: **YEAR?**, **MONTH?** and **DAY?**. It has the form:

```
: valid? \ day month year -- t|f
  dup year?
  if  over month?
    if   swap day?
    else 2drop drop 0          \ clean up and leave false
    then
  else  2drop drop 0           \ clean up and leave false
  then
;
```

Note that if a test fails we must clean up the stack and leave just the value of false.

Lessons from this example

This example of the use of the stack in Forth to manipulate numbers and test results, etc. shows us several things about design of Forth applications:

- The problem should be split up into several smaller problems.
- Each of these should be tackled in the simplest way possible.

- The code should be tested as it is written.
- Smaller definitions are better.

Programmers used to C often complain that using many small functions must result in slow code. The Forth virtual machine is optimised for handling anonymous inputs and outputs and the code does not have to build and destroy stack frames. Consequently the only overheads of small functions are the CALL and RETURN machine instructions themselves. Some Forths, such as MPE's VFX Forth, can automatically expand small definitions to avoid even this overhead.

7 Simple character I/O

Although most of these words have been used before, they are defined here because they will be used in the next two examples.

Output

EMIT char --

This word will print the specified character at the current output position, and update the cursor position. **EMIT** may also correctly work with control codes. Conceptually, **EMIT** is the fundamental word used by more complex screen output words.

A few examples:

```
Decimal
65 emit          \ will print the letter "A"
48 emit          \ will print the digit "0"
7 emit           \ will usually cause a beep to sound
```

TYPE addr length --

The word **TYPE** will print length number of characters starting from the given address. To convert the address of a counted string to the parameters needed by **TYPE**, the word **COUNT** is necessary.

CR --

Perform a carriage return, line feed operation. In other words, go to the start of the next line.

PAGE --

This word will clear the screen and move the cursor to the top left-hand corner of the screen. Often called **CLS**.

Input

KEY -- char

KEY waits until a key is pressed at the keyboard; when one is pressed it returns the ASCII value of that key on the stack. From the keyboard, try typing **KEY**. You will see the cursor advance one space and wait for you to enter a key. If you now enter a key, and then type "dot" **.** you will see the ASCII code of the character. Try this small program:

```
: enter-key      \ --
  cr ." Enter character : " key
  cr ." You entered : " emit
;
```

KEY? -- t/f

KEY? returns a true flag if a character is available at the keyboard; this character can then be read by **KEY**.

ACCEPT addr +n -- len

Typical places to store text string input are **TIB**, and **PAD**. **ACCEPT** is used to accept more than one character at a time from the keyboard. **ACCEPT** takes an address, and a positive integer as it's arguments from the stack. The keyboard input is accepted until **either** a <CR> is entered, **or** `+n' characters are read. When this occurs the string is stored at the address given. The length of the string that was actually input is returned. Try this small word:

```
: greetings     \ --
  cr ." Enter your name : " pad 40 accept
  cr ." Hi there " pad swap type
;
```

String Output

Strings are handled in Forth in two ways.

The first form uses strings defined as two cells on the stack **(caddr len --)** which can be passed directly to **TYPE**. The address of characters is often shown as **caddr**.

```
: myTown       \ -- caddr len
  S”Southampton”  ;
```

Executing **myTown** will return the address of the first character of the string and its length.

```
| S | o | u | t | h | a | m | p | t | o | n |
  ^
```

The string can be displayed using:

```
MyTown type
```

The second form is called a counted string i.e. strings are stored as a count byte (the length of the string) followed by the actual characters that make up the string.

```
: town$          \ -- $addr
  C" Southampton" ;
```

When executed this word will return the address of the counted string

```
| 11 | S | o | u | t | h | a | m | p | t | o | n |
  ^
```

The address returned is that of the count byte. The address returned will now have to be converted to parameters that are usable by the word **TYPE** above.

COUNT \ addr -- addr+1 length

To print the string Southampton define:

```
: print-town$    \ --
  town$ count type ;
```

Since **TYPE** works with any address and length given to it, it is possible to alter the starting point from which to begin printing and also the number of characters printed. e.g.

```
town$            \ get counted string address
1+               \ skip the count byte
4 +              \ skip 4 chars. "h"
```

```
3               \ going to print 3 characters
type            \ go ahead and print
```

String input and the input stream

WORD `char “<text>” -- addr`

> **WORD** will allow you to show that Forth is capable of infix notation. **WORD** accepts input from the input stream without modifying it. It does this by examining each character in the input stream, if the character is equal to the delimiting character `char' supplied, **WORD** returns an address containing a counted string which represents the input up to the delimited character. **Note:** Leading delimiter characters are ignored.
>
> By convention, the form `“<text>”` on the left-hand side of a stack comment does not indicate a stack item. Instead, it indicates that text is read from the input stream.

WORD operates on the current line. The address and length of the line can be found using **SOURCE**.

SOURCE `-- c-addr u`

> Returns the address and length of the current line in the terminal input buffer.

The variable **>IN** is the offset in the current line of pointer "into the input stream". This tells the interpreter how far it has got into the input stream. It is a user variable.

```
: i'm           \ “<text>” --
  bl word
  cr ." Hi there " count type ;
```

Use this word in the form:

```
i'm Fred
```

Notice that "Fred" had already been typed when **I'M** was executed; so **ACCEPT** could not be used.

Programming Note: **BL** is a constant returning the ASCII code for a space character.

Number output

This word below will print the numbers 1 to 100 when executed:

```
: 1to100 \ --
  101 1
  do  i . space loop ;
```

The format of the output will be similar to:

```
1   2   3   4   5   6   7   8   9   10   11   12
13   14   15   16  17   18   19   20   21   22   23
24   25   26    27   28  29   30   31   32   33   34
35   36   37   38   39  40   41   42   43   44   45
46   47   48   49   50   51  52   53   54   55   56
57   58   59   60   61   62   63  64   56   66   67
68   69   70   71   72   73   74  75   76   77   78   79
80   81   82   83   84   85   86  87   88   90
91   92   93   94   95   96   97   99   100
```

This sort of output is messy and unsuitable for the output of tables etc. However there is a word **.R** which will output numbers in fields of specified length.

.R `\ n field-width --`

A slight re-definition of the previous word **1to100** using the word **.R** would consist of :

```
: 1to100 \ --
  101 1
  do  i 4 .r  loop ;
```

This word will give an output similar to:

```
01   02   03   04   05   06   07   08
09   10   11   12   13   14   15   16
17   18   19   20   21   22   23   24
25   26   27   28   29   30   31   32
33   34   35   36   37   38   39   40
41   42   43   44   45   46   47   48
49   50   51   52   53   54   55   56
57   58   59   60   61   62   63   64
65   66   67   68   69   70   71   72
73   74   75   76   77   78   79   80
81   82   83   84   85   86   87   88
89   90   91   92   93   94   95   96
97   98   99   100
```

See if you can modify this to give five items per line.

Number input

Numerical input is handled by the previously explained character and string input words. Once you have the ASCII string of the characters, you may need to perform mathematical functions upon them, so Forth provides two basic words to enable the programmer to convert the pictured input into binary data.

DIGIT `c base -- n t/f`

Simple to use word that is useful when parsing simple numerical input. **DIGIT** converts the ASCII character given to a number, if possible, in the base supplied, but returns only the false flag if it failed to convert the characters. Try this small word:

```
: get-#          \ --
  cr ." Enter a number : " key 10 digit
  if  cr ." The number was : " .
  else cr ." That was not a number !!!"
  then
;
```

NUMBER? `addr -- nl..nn n2`

Simple to use word that is useful when parsing numerical string input. **NUMBER?** converts the counted string into a number if possible. If conversion was successful, then the number of words generated, as well as a number of that size is left on the stack. If conversion was unsuccessful then a zero, false, flag is left on the stack:

```
No conversion    -- 0
Single number    -- n 1
Double number    -- d 2
```

If a double number is encountered the user variable **DPL** will contain the number of digits after the decimal point.

MPEism: *The ANS standard specifies that a double number is indicated by a '.' character in the number. Because this confuses people used to entering floating point numbers with a dot, and because there is considerable variation within Europe on the use of commas and dots in numbers, MPE Forths default to using a comma to indicate double numbers. MPE Forths use the variable* ***DP-CHAR*** *to hold the character which signifies a double number.*

Redirecting KEY and EMIT

Many systems permit **KEY**, **EMIT** and friends to be redirected to device other than the console. Desktop systems use this for windows, serial lines, memory buffers, files and so on. Embedded systems can use redirection to switch output from the development port to an LCD display or a ticket printer. Redirection is usually set independently for each task.

Because I/O redirection is not defined by the ANS standard, the implementation of redirection varies widely between Forth systems.

8 Defining with CREATE ... DOES>

This section is devoted to a facility which distinguishes Forth from almost all other languages. To use this facility you will only need to learn one new Forth word. The concepts involved with this word however are profound. If you can master it you will have become a true Forth programmer!

Most languages distinguish a ``compile time" phase from a ``run time" phase. For example when declaring an array in Pascal, at compile time space must be reserved and at run time the array elements will be accessed.

Forth does not include a standard method for defining arrays. Instead it gives you the tools to define compile time and run time behaviour, so you can create your own ``defining words" to set up arrays and many other ``classes" of word. Standard Forth has some basic ``defining words" already supplied. For example: **CONSTANT**, **CREATE** and **VARIABLE**. We will now have a closer look at **CONSTANT** to see how such defining words can themselves be defined.

All words defined by **CONSTANT** have a similar behaviour. Suppose we define:

```
3 CONSTANT FRED
7 CONSTANT JIM
```

When Forth reads the definition of **FRED**, a dictionary entry for **FRED** is created and the value 3 is associated with it. We call this the compile time action for **FRED**. When the **FRED** is subsequently invoked, as in:

```
FRED .
```

the run time action for **FRED** is executed, and this pushes the value 3 onto the stack. **FRED** is called a **child** of **CONSTANT**. A possible definition of **CONSTANT** is:

```
: CONSTANT      \ n -- ; -- n
  CREATE        \ compile time, create new data word
    ,                       \ lay down the value n
  DOES>         \ what the child does, get data address
    @                       \ fetch contents
;
```

CREATE <name> makes a new entry in the dictionary. The run time action of **<name>** is to return the address at which any data (laid down by **,**) may be found.

The phrase **CREATE ,** is the compile time action which takes place when a constant is defined. At this point **CREATE** builds the dictionary entry for the new constant and its value is compiled into the next free dictionary location.

The words **,** and **C,** are used to compile a cell- or character-sized unit in the dictionary.

, `\ x --`

Place the value x into the dictionary at **HERE** and update the pointer the pointer by the size of a cell.

C, `\ char --`

Place the character-sized data (usually a byte) into the dictionary at **HERE** and update the pointer the pointer by the size of a character.

HERE `\ -- addr`

Return the address of the next free space in the dictionary data area. You will come across this word later.

ALLOT `\ n --`

Reserve space in the dictionary data area without initialising it. You will come across this word later.

DOES> separates the compile time action from the run time action, returning the address of the data at run time. The run time action for a constant is simply to perform a ``fetch'' with the command but a fetch from which address? Recall that words defined by **CREATE** return the address of the next free dictionary location at the time they were created. This idea still holds, but now we can add words following **DOES>** which act on this address.

Traditionally the address left for the run time code is called the “parameter field address” of the word. All constants perform the same run time action, but on different parameter fields.

***MPEism**: A defining word has two stack comments. The first is for the stack action when the defining word runs and the second is for when the child is run.*

Arrays

```
: Array        \ size -- ; [child] n -- addr ; cell array
  Create                  \ create data word
    cell *                \  calculate size
    Here over erase       \  clear memory
    allot                 \  allocate space
  Does>                   \ run time gives address of data
    Swap cell * +         \  index in
;

: 2Array       \ n1 n2 -- ; [child] n1 n2 -- addr ; 2-D array
  Create                  \ create data word
    Over ,                \   save width for run-time
    cell * *              \   calculate size
    Here over erase       \   clear memory
    allot                 \   reserve n cells
  Does>                   \ run time gives address of data
    Dup @ Rot * Rot + 1+
    cell *
    +
;

: Message-array      \ len mesgs -- ; [child] n -– addr
\ generate an array for mesgs messages of size len
  Create                  \ create data word
    Over ,                \   save the size of the array
    *                     \   calculate space needed
    Here over erase       \   clear memory
    allot                 \   reserve space for it
  Does>                   \ run time gives address of data
    Dup @                 \   get length
```

```
    Rot *                   \   calculate offset
    +                       \   add to base
    cell +                  \   skip length
;
```

Structures

A structure is a template for the layout of data. An instance of a structure may be global data or a record in memory or a peripheral in a microcontroller. A field in a structure is a data unit of a certain size. The core action of a field is to add an offset (position) in the template to the base address (where the physical structure is located). When a structure is referred to by name in the code, we need the size of the structure.

When implementing a structure notation in Forth we are using defining words to extend the compiler. The version here provides the required functionality with a little bit of syntactic sugar to make the code easier to read.

```
0 constant [struct   \ -- 0 ; start
\ Start a structure definition, returning zero as
\ the initial offset.


: field        \ offset n -- offset+n ; addr -- addr+offset
\ When defining a structure, a field of size n starts
\ at the given offset, returning the next offset. At
\ run time, the offset is added to the base address.
  create
    over ,  +        \ lay offset and update
  does>
    @ +              \ addr offset to address
;

: struct]      \ offset -- ; -- size ; end
\ End a structure definition by naming it.
  constant
;
```

The first and third words are just syntactic sugar to make the code easier to read. For example, we can define a simple array of integers as follows:

```
: int          \ addr – addr+offset
  cell field
;

[struct        \ -- 0
  int link           \ link to previous structure
  int ident          \ identifier
  int data           \ data value
  16 field name      \ name string
struct] record \ -- size
```

When we need a global instance of this structure we can use:

```
Create FirstRec      \ -- addr
  Record allot
```

Or, more prettily:

```
Record buffer: FirstRec    \ -- addr
```

To write a value into the data field, we can use:

```
55 FirstRec data !
```

Although this is a very simple example, it provides what we need. However, to some eyes it would be easier to read if the structure is named first and defined afterwards. Many Forth compilers, even optimising ones, do not expand children of defining words and the code may be somewhat inefficient. Solutions to these problems are provided in the later chapter “Extending the compiler”.

9 Diary and Phone Book Examples

This chapter gives two worked examples. For array definitions see the defining word chapter. These examples are not written in the most “Forth-like” style, they provide solutions that will be familiar in other languages. The phone book example is revisited in a later chapter to show how to take more advantage of Forth.

Diary

Specification

- Stores x entries in the format:
  day, month, year, hour, min, user message.

- Validates all fields before storage, including leap years...

- Provide words that :-

  Store-Entry `\ dd mm yy hh mm <message> --`
  To-Do `\ dd mm yy -- ; display messages for day`

- Shall provide array-building words.

Implementation

```
13 constant cr-chr              \ code for carriage return

10 Constant Max-entries    \ maximum number of entries allowed
20 Constant Max-message-len     \ maximum message length

Variable #entries               \ number of entries in diary

Max-message-len Max-entries     \ array of max-entries and
  Message-array Messages        \ max-message-len long
Max-entries 2 2Array Times      \ array for holding times
  0 Constant Hour               \   index for hours
  1 Constant Minute             \   index for minutes
Max-entries 3 2Array Dates      \ array for holding dates
  0 Constant Day                \   index for hours
  1 Constant Month              \   index for minutes
  2 Constant Year               \   index for years

: $move        \ src$ dest$ -- ; copy counted string
  over c@ 1+ move
;

: Save-message \ message$ entry# -- ; save message$ at entry#
  Messages $Move                \ copy message to array
;

: Save-time    \ hh mm entry# -- ; save time (hh:mm) in array
  Tuck Minute Times !           \ save minutes
  Hour Times !                  \ save hours
;
```

```
: Save-date    \ dd mm yy entry# -- ; save date
  Tuck Year Dates !              \ save year
  Tuck Month Dates !             \ save month
  Day Dates !                    \ save day
;

: Valid-message?     \ message$ -- t/f ; is message valid?
  Dup c@ 0<>                     \ must be non zero
  Swap c@ Max-message-len <= and \ and must fit in array
;

: on           \ -- ; just added for readability
;

: at           \ dd mm yy -- t/f ; save date if valid
  3Dup Valid-date?               \ is date valid?
  If                             \ Yes,
    #entries @ Save-date         \   save date
    True                         \   return true
  Else                           \ No,
    2Drop Drop                   \   clear stack
    False
  then
;

: ?Valid-message     \ message$ -- t/f ; validate and save
  Dup Valid-message?             \ is message valid?
  If                             \ Yes,
    #entries @ Save-message      \   save message
    True
  Else                           \ No,
    Drop                         \   clear stack
    False
  then
;

: ?Valid-time  \ hh mm -- t/f ; validate and save time
  2Dup Valid-time?               \ is message valid?
  If                             \ Yes,
    #entries @ Save-time         \   save message
    True
  Else                           \ No,
    2Drop                        \   clear stack
    False
  Then
;

: Write        \ hh mm <name> -- t/f t/f ; validate and save
  Ascii . Word ?Valid-message    \ validate and save message
  -Rot ?Valid-time               \ validate and save time
;

: Diary-full?  \ -- t/f ; is Diary full?
  #entries @ Max-entries =       \ at maximum?
;

: To-diary     \ t/f t/f t/f -- ; accept entry?
  And And                        \ all ok?
```

```
  Diary-full? Not And           \ and is diary is not full?
  If                            \ Yes,
    #entries incr               \   accept entry
    cr ." Set entry in diary"   \   tell user
  Else                          \ No,
    cr ." Can't set diary entry" \   tell user
  Then
;

: This-date?   \ dd mm yy entry# -- t/f ; entry has date?
  Tuck Year Dates @ =                \ year match?
  -Rot Tuck Month Dates @ = Rot and  \ and month match?
  -Rot Day Dates @ = and             \ and day match?
;

: .Time        \ entry# -- ; display time
  Dup Hour Times @ 2digits       \ display hour
  Ascii : Emit                   \ display colon
  Minute Times @ 2digits         \ display minutes
;

: .Message     \ entry# -- ; display message for entry
  Messages count type
;

: .entry       \ entry# -- ; display entry
  cr
  Dup  .Time                     \ display time
  5 Spaces                       \ leave a gap
  .Message                       \ display message
;

: Diary        \ dd mm yy -- ; display messages for this date
  #entries @ 0
  ?Do
    3Dup i This-date?            \ a matching date?
    If                           \ Yes,
      i .entry                   \   display message
    Then
  Loop
  2Drop Drop
;


\ Example diary entries

on 3  12 1992 at 10 30 write message 1. to-diary
on 3  11 1992 at 10 30 write message 2. to-diary
on 3  12 1992 at 12 30 write message 3. to-diary
on 2  12 1992 at 10 30 write message 4. to-diary
on 2  12 1992 at 10 30 write message 4. to-diary
on 3  12 1991 at 10 30 write message 5. to-diary
on 3  12 1992 at 11 15 write message 6. to-diary
on 29  2 1988 at 10 30 write message 7. to-diary
on 24 12 1992 at 10 30 write message 8. to-diary
on 3  12 1992 at 01 00 write message 9. to-diary


0 [if]
```

```
\ Example invalid diary entries
on 32 1  1991 at 10 30 write message 5. to-diary
on 29 2  1991 at 10 30 write message 5. to-diary
on 2  13 1991 at 10 30 write message 5. to-diary
on 2  12 2100 at 10 30 write message 4. to-diary
on 2  12 1992 at 24 30 write message 4. to-diary
on 2  12 1992 at 10 70 write message 4. to-diary
on 2  12 1992 at 10 70 write message this message is far, far
far too long. to-diary
[then]
```

An Internal Phone Book

Specification

You are required to create an application to allow the storing and querying of an electronic phone book for internal calls.

The data to be recorded is the surname of the employee:

```
1 <= length(surname) <= 15
```

and his or her telephone number:

```
0 <= number <= 9999
```

There may be up to twenty entries in the phone book at any one time. Apart from 0 itself, telephone numbers may not begin with a 0. The user interface requirement is as follows.

Adding

To add a number to the book the user should type in the following format:

```
nnnn calls name<Cr>
```

Where `nnnn' is a phone number in the correct range (0 <= number <= 9999) and `name' is the surname of the employee (not more than 15 characters long). <Cr> represents the pressing of the "Return" or "Enter" key.

Examples:

```
3270 calls prefect
4298 calls slartibarfast
0    calls switchboard
42   calls dent
```

Querying

The phone book may be queried in three ways.

Who is at this number?

```
nnnn calls?<Cr>
```

Enquires which person a number calls. The computer should reply with the surname of the individual or give an appropriate error message.

```
42 calls? Dent
25 calls? Nobody        \ error case
```

What is X's number?

```
phone name<Cr>
```

Inquires the telephone number of the person named. The computer should respond with the number of the person named or give an appropriate error message.

```
phone Prefect<Cr>
3270
phone Marvin
Marvin Has No Phone!
```

List the book

```
entries?<Cr>
```

List the contents of the phone book.

```
entries?<Cr>
Prefect          -- 3270
Slartibartfast   -- 4298
Switchboard      -- 0
Dent             -- 42
```

Some design notes

The first thing we will need for an answer to the problem is a data structure to contain the names, and another to contain the phone numbers. Thinking back to the notes on arrays, we can see that the best way to hold the phone numbers will be an array of twenty entries. This is fairly straightforward. However, a little thought will be required to hold the names in a data structure.

We will need to define a data structure that consists of `slots' of 16 bytes each (one for the 1-15 characters of the name, and one for the count byte of the counted string). It is suggested that the data structure that holds these names has the following stack effect:

```
\ entry# -- $addr
```

which will take the number of an entry, and return the address in memory of the text. We will also need to know which is the next free entry in the book so that when we add another, we will add it into the next free slot. This we can do with a variable.

When we add an entry, we define the name and the phone number. The word **CALLS** will therefore take a number off the stack and store it in the next free slot in the array of phone numbers. The free slot will be determined from the variable which contains this, and will start at zero. Before storing the phone number, the condition that it does not start with a zero, unless it is zero will have to be tested. This test will run along the lines that if the number is zero, it is acceptable. However, if the number is non-zero, the nature of the Forth interpreter is such that leading zeros are removed from a number. This means that any number typed will not have a leading zero on it by the time it is on the stack. This meets the condition required. Having stored the telephone number, the **CALLS** word must also extract the name of the called from the command line. This may be done using the phrase '**BL WORD**'. This will return the address of the counted string so gathered. This string may be trimmed to 15 characters if it is too long, by fixing the count byte. The text and count byte must then be moved into the array containing the names. Having done so, the variable indicating the next free slot must be updated.

The **CALLS?** word is used to find out who a telephone number calls. This will take the phone number off the stack as a parameter, and test each entry in the array of phone numbers (up to the first free slot). If the number is found, then the appropriate name will be extracted from the name array, and printed. If the phone number is not found, the error message will be printed.

The word **PHONE** returns the phone number for the person named on the command line. This word will take the name from the command line, much like **CALLS**, and will then look up all entries in the array of names until one matches. If no match is found, the error message will be printed. If a match is found, then the equivalent phone number will be extracted from the array of phone numbers, and will be printed.

Listing the phone book (**ENTRIES?**) will index through the array of names, and print each name out as it finds it. As it finds a name, it will look in the array of phone numbers for the equivalent entry, and will print out the phone number. It will the print a <Cr> and go on to the next one.

Implementation

This version is a simple classical implementation using an array of records. A later chapter discusses an implementation that takes advantage of Forth's internal structures.

Structures

```
0 constant struct{   \ -- 0
\ Start a structure.

: fld          \ len size "<name>" -- len' ; addr -- addr+len
\ Define a field in a structure, given the current length
\ of the structure and the size of the field. The name follows
\ in the input stream.
  create
    over ,  +
  does>
    @ +
;

: }struct      \ len "<name>" -- ; -- len
\ End a structure definition and give it a name.
\ At run time <name> returns the length of the structure.
  constant
;
```

Strings

```
[undefined] UPC [if]
: UPC           \ char -- char'
\ Convert supplied character to upper case if it was
\ alphabetic otherwise return the unmodified character.
  dup [char] a >= if
    dup [char] z <= if
      $DF and
    then
  then
;
[then]

[undefined] UPPER [if]
: UPPER         \ addr len --
\ Convert the ASCII string described to upper-case. This
\ operation happens in place.
  bounds ?do
    i dup c@ upc swap c!
  loop
;
[then]
```

Miscellaneous

```
[undefined] place [if]
: place        \ caddr len dest --
\ Store the string given by caddr/len as a counted string
\ at dest.
  2dup 2>r                 \ write count last
  1 chars + swap move
  2r> c!                   \ to avoid in-place problems
;
[then]

[undefined] >pos [if]
: >pos          \ +n --
\ Place cursor on current line to column n if possible.
  out @ - spaces
;
[then]

[undefined] bounds [if]
: bounds \ addr len -- addr+len addr
\ Convert an address and length to a form suitable for use
\ with DO...LOOP structures.
  over + swap
;
[then]
```

Phone Book data

```
struct{
  cell fld PB.Phone#             \ phone number
  #16  fld PB.Name               \ name
}struct PBentry            \ -- len
\ The structure that defines a record in the phone book.

20 constant MaxEntries     \ -- n
\ Maximum number of entries in the phone book.

MaxEntries PBentry * constant /PhoneBook     \ -- len
\ Size of the phone book data.

create PhoneBook     \ -- addr
\ The phone book data.
  /PhoneBook allot
```

Processing the phone book

```
: PB[]         \ n -- addr
\ Return address of nth entry in phone book.
  PBentry * PhoneBook +
;

: CheckNumber  \ n --
\ Error if n is outside the range 0..9999.
  0 9999 within?
  0= abort" Telephone number outside range 0..99999"
;
```

```
: CheckName    \ caddr len --
\ Apply checks to a name.
  dup 15 > abort" Name too long"
  2drop
;

: FindNumber   \ num -- n true | false=0
\ Given a number, find the slot it occupies in the phone book
\ and return the slot number and true. If the number cannot be
\ found, just return false.
  MaxEntries 0 do                \ -- num
    dup i PB[] PB.Phone# @ = if
      drop  i true  unloop  exit \ -- n true
    then
  loop
  drop 0                         \ -- 0
;

#32 buffer: N1
#32 buffer: N2

: SameNames?   \ caddr1 len1 caddr2 len2 -- flag
\ Return true if the names given by caddr1/len1 and
\ caddr2/len2 are the same. Why is this word provided?
\ How can you compare names in different CaSes?
  N2 place  N1 place
  N1 count upper  N2 count upper
  N1 count N2 count  compare 0=
;

: FindName     \ caddr len -- n true | false=0
\ Given a name, find the slot it occupies in the phone book
\ and return the slot number and true. If the number cannot be
\ found, just return false.
  MaxEntries 0 do                \ -- caddr len
    2dup i PB[] PB.Name          \ -- caddr len caddr len an
    count SameNames? if
      2drop  i true              \ -- n true
      unloop  exit
    then
  loop
  2drop 0                        \ -- 0
;

: FindFree     \ -- n true | false=0
\ Find a free slot in the phone book and return the
\ slot number and true. If there is no room, just return
\ false.
  MaxEntries 0 do                \ --
    i PB[] PB.Phone# @ -1 = if
      i true  unloop  exit       \ -- n true
    then
  loop
  0                              \ -- 0
;

: EraseEntry   \ n --
\ Wipe nth entry in phone book.
```

```
  PB[] dup PBentry erase
  -1 swap PB.Phone# !
;

: EraseBook    \ --
\ Wipe the phone book
  MaxEntries 0
  do  i EraseEntry  loop
;
EraseBook

: MakeEntry    \ num caddr len n --
\ Add the entry whose phone number is num and name is
\ given by caddr/len to slot n in the phone book.
  PB[] >r
  2dup CheckName  r@ PB.name place
  dup CheckNumber r> PB.Phone# !
;

: .PBname      \ slot --
\ Display name in entry.
  PB.name count type
;

: .PB#         \ slot --
\ Display name in entry.
  PB.Phone# @ .
;

: ShowEntry    \ n --
\ Display the contents of slot n in the phone book
  PB[] dup PB.name c@ if         \ if slot has name
    cr  dup .PBname              \ display name
    #16 >pos  ." -- "            \ step to col 16
    .PB#                         \ display number
  else
    drop
  then
;
```

Application words

```
: Calls        \ num "<name>" --
\ Make a new entry in the phone book in the form:
\   <nnn> Calls <name>
  dup FindNumber abort" Number in use"
  bl word count 2dup FindName abort" Name in use"
  FindFree 0= abort" Phone book full"
  MakeEntry
;

: Calls? \ num --
\ Report who is called by number num. Use in the form:
\   <nnn> Calls?
  FindNumber if
    PB[] .PBname
  else
    ." Nobody"
```

```
  then
;

: Phone        \ "<name>" --
\ Give the phone number for name. Use in the form:
\   Phone <name>
  bl word count 2dup FindName if
    cr PB[] .PB#  2drop
  else
    cr type ."  has no phone."
  then
;

: Entries?     \ --
\ Display the contents of the phone book.
  MaxEntries 0                  \ --
  do  i ShowEntry  loop
;
```

10 Execution Tokens and Vectors

When the Forth text interpreter or the Forth compiler looks up a word in the dictionary they find the ``execution token'' associated with the word. An execution token, often referred to as an **xt**, is a unique identifier for a Forth word. It is usually, but not always, the address of the code to call.

In this section we see how the ability to handle an execution token can be useful in an application.

Input the following command line:

```
1 2 ' + {ok}
```

There are now three items on the stack, which are the values 1 and 2, and the execution token for **+**. The single apostrophe (referred to as ``tick'') looks up the following word and leaves its execution vector. If you need to refer to the xt of a word inside a colon definition use the phrase **['] <word>**.

We can execute the token:

```
EXECUTE . {3 ok}
```

By definition, the phrase **' <word> EXECUTE** has the same action as **<word>**, so this will have the same overall effect as performing:

```
1 2 + . {3 ok}
```

The word **EXECUTE (i*x xt -- j*x)** just performs the xt passed to it. It is, if you like, an indirect call to **xt**. The notation **i*x** and **j*x** indicates that the stack action is defined by that of **xt** and not by **EXECUTE**.

Exercise: Predict what Forth will print in response to:

```
100 20 ' * EXECUTE .
```

Execution vectors

We can treat an execution token like any other data item, pass it as a stack parameter, save it in a variable or in a table and so on. In the following example we define a variable **OPERATION** which will be used to hold an execution token. **DO-IT** is defined to execute the operation, and **SET-OP** finds a new token and stores it in **OPERATION**.

```
VARIABLE OPERATION              \ -- addr

: DO-IT   ( i*x – j*x )  OPERATION @ EXECUTE ;

: SET-OP  ( -- )         ' OPERATION ! ;
```

We can use this as follows.

```
SET-OP /  \ store xt of / in OPERATION

9 3 DO-IT . {3 ok}
```

Note that when “tick” is embedded in another definition, as it is in **SET-OP**, this definition takes on the property of looking up the next word in the dictionary. It is

also possible to look up a word that occurs within the compiled text. This is done by the word **[']**.

For example the following definition will set the execution token of **+** in the variable **OPERATION**.

```
: SET+  ['] +  OPERATION !  ;
```

Now we can enter:

```
SET+  9 3 DO-IT .  {12 ok}
```

Exercise: Define a word **KEYS** which inputs character codes from the keyboard and emits them to the screen, terminating when a carriage return key is received. **KEYS** is to work in several different modes.

NORMAL KEYS will echo all characters without changing them.
UPPER KEYS will echo only upper case letters, with a dot printed for all other characters.
CAP KEYS will convert lower case letters to upper case, then output.
ENCODE KEYS will add one to the ASCII code of each key before output.

Complete the following outline code to implement the application. Note that the design allows additional modes to be added without changing the existing code! Note also that you can define a character inside a colon definition using: **[CHAR] A (-- 65)**

```
VARIABLE KEY-OPERATION

: TRANSFORM-KEY
  KEY-OPERATION @ EXECUTE
;

: KEYS           \ --
  BEGIN
    KEY  DUP 13 <>
   WHILE
    TRANSFORM-KEY EMIT
  REPEAT
  DROP                          \ drop CR key code
;

: NORMAL  \ --
  ['] NO-OP KEY-OPERATION !
;

: ENCODE  \ --
  ['] 1+ KEY-OPERATION !       \ 1+ is short for 1 +
;

: <UPPER> \ char1 – char2
  your code goes here   \ if not A..Z, replace by '.'
;

: <CAP>          \ char1 – char2
  your code goes here   \ if a..z, replace by A..Z
;

: UPPER          \ --
  your code      \ store xt of <UPPER> in KEY-OPERATION
;
```

```
: CAP           \ --
  your code goes here
;
```

Most Forth systems provide a defining word called **DEFER** which creates a variable (holding an xt) which executes the xt when the word is referenced.

```
DEFER <name>
```

A common notation for setting the action is:

```
' <action> IS <name>
```

although several others are also used.

MPEism: *The ANS standard does not define* **DEFER** *or a means of setting an action. As well as supporting* **IS** *MPE code uses the form:*

```
ASSIGN <ACTION> TO-DO <NAME>
```

Execution arrays

There are cases when the action of a system is dependent of the value of the input data. An example of this is handling the keystrokes for an editor. Many editors are programmable for the actions of key strokes.

A naive but eventually complex method would be to hold the keys for specific actions in a set of variables and to check each key value against each of these. This eventually becomes tedious and long winded in the source code, and slow to execute. This may not be important in human terms, but can significantly impact the performance when replaying a macro. In other applications that process a serial data stream, performance may be important.

In our editor example, we will assume for the moment that we are handling 8 bit characters. This means that we have 256 characters to choose from. We can store the xt of the word that handles character n in the nth slot in an array. Since many of the characters are handled in the same way (displayed) we can simplify life by passing the character code to the word that handles it, and give all the character-handling words the stack action:

```
char --
```

To define the array:

```
create KeyTable  \ -- addr
  256 cells allot             \ reserve space for 256 xts
```

Now we need to initialise the table with the default action, which is to display the character. We assume that this word is called **DisplayChar**.

```
: InitTable             \ --
  KeyTable 256 cells bounds
  do  ['] DisplayChar i !  cell +loop
;
```

Now we need a word to set a character's action:

```
: SetKey         \ char "name" --
  '                          \ find xt of the next word
  swap cells KeyTable + !    \ index into the table
;
```

Now we can define the action of each keystroke by using a simple script. If we keep this script in a file, we can make the editor completely configurable, we can preserve the settings, and we can use the Forth interpreter itself to load our editor's settings. Since we have an interactive language, there is no reason not to use the interpreter and/or compiler at run-time!

```
13 SetKey doCR
10 SetKey doLF
...
```

In a fixed application, e.g. serial line processing, we may only need to handle the control keys (0..31) in this way, and with fixed actions **(--)**. We can produce a predefined table as follows:

```
Create KeyTable  \ -- addr
  ‘ action0 ,    ‘ action1 ,
  ‘ action2 ,    ‘ action3 ,
  ...

: ProcessKey            \ char --
  dup #32 u< if
    cells KeyTable + @ execute
  else
    DisplayChar
  then
;
```

11 Extending the compiler

Forth is an extensible language. In Forth you can add to the compiler itself to produce compiler macros and new control structures.

Immediate words

Forth has a compiler that provides sequential composition of operations. Some aspects of the compilation process require more than this however, and these situations are dealt with by ``immediate words".

Immediate words occur in Forth definitions in the normal way but they are executed during compilation instead of being compiled. Suppose we enter the definition:

```
: BEEP   7 EMIT  ;  IMMEDIATE
```

BEEP emits an ASCII code 7, which causes a beep to be sounded by the console. The word **IMMEDIATE** flags the most recently defined word as ``immediate". Now if we invoke **BEEP** during compilation of another word it will be executed immediately rather than being composed as part of the new definition. For example enter:

```
: TEST   100 . BEEP 200 .  ;
```

The beep sounds as the definition of **TEST** is being compiled. It is not compiled as part of **TEST**.

Immediate words are used to provide all the extensions to the basic Forth compiler. The following words are all immediate.

```
." ; IF  ELSE THEN BEGIN WHILE REPEAT UNTIL AGAIN
```

For a more serious example we start by studying the definition of Forth's **IF ... THEN** control structure. Note that the techniques we describe enable you to add new control structures to Forth in an incremental fashion.

First we need to look a little more closely at the compiled form of a Forth definition.

The following session assumes you are seated at your Forth terminal. The results are taken from VFX Forth for Windows, which generates optimised native code. Other compilers will generate different results. Enter the definition:

```
: ABS   DUP  0<  IF  NEGATE  THEN  ;
```

Now examine the code generated by the compiler:

```
see abs
ABS
( 00495EC4    0BDB )                  OR      EBX, EBX
( 00495EC6    0F8D02000000 )          JNL/GE  00495ECE
( 00495ECC    F7DB )                  NEG     EBX
( 00495ECE    C3 )                    NEXT,
( 11 bytes, 4 instructions )
 ok

' abs 10 dump
0049:5EC4  0B DB 0F 8D 02 00 00 00  F7 DB C3 04 44 55 4D 50
 ok
```

Here, the phrase **' ABS** looks up **ABS** in the dictionary and returns its execution address (as discussed in the previous section on execution vectors). The body of the definition (usually) follows this address, so the dumped values must represent the operations **DUP 0<** etc.

The guts of the definitions of **IF** and **THEN** in VFX Forth are as follows:

```
: IF            \ C: -- orig ; Run: x --
\ *G Mark the start of an IF..[ELSE]..THEN conditional
block.
  s_?br>,
; IMMEDIATE

: THEN          \ C: orig -- ; Run: --
\ *G Mark the end of an IF..THEN or ..ELSE..THEN
conditional.
  s_res_br>,
; IMMEDIATE

: AHEAD         \ C: -- orig ; Run: --
\ *G An unconditional forward branch resolved later.
  s_br>,
; IMMEDIATE
```

They simply provide access to compilation primitives which start and resolve forward branches, producing or consuming branch target addresses. The internals of each Forth compiler will be different and hence are not portable between Forth implementations. What is required is a way to use words such as **IF** and **THEN** inside other compiling words. This facility is provided by the word **POSTPONE**, which is always used inside a colon definition in the form **POSTPONE <name>** to delay execution or compilation of **<name>**. If **<name>** is immediate, it is not executed, but compiled into the colon definition. If **<name>** is not immediate, it would normally be compiled but now the colon definition will compile **<name>** when the colon definition is executed.

Now we have the mechanics of using the existing compilation structures. As an example, let us define a way of compiling a “short circuited” conditional. When we want to perform an action as result of conditions a, b and c being true we can define it two ways. The first way is:

```
A B AND C AND IF
  ...
THEN
```

If **A** returns false, execution of **B** and **C** is redundant. To avoid this, we might code it a second way with several conditionals:

```
A IF
  B IF
    C IF
      ...
    THEN
  THEN
THEN
```

But this is tedious and takes a lot of source code space. A neater notation which generates the same code as the second solution might be a third solution, originatated by Wil Baden:

```
A ANDIF  B IF  C IF
  ...
THENS
```

ANDIF starts the structure and **THENS** resolves the foward branches. Now we go back and see that when **IF** executes (during compilation) it produces an **orig** on the compilation stack (usually the data stack). An **orig** is a reference to where the forward branch has been compiled. This is consumed by **THEN** which patches up the forward reference. Assuming that **ANDIF** also produces an orig, when **THENS** is reached it will find three **origs** on the compilation stack. We will use a marker of 0 to indicate that there are no more **origs** to be resolved.

```
: ANDIF          \ C: -- 0 orig ; Run: x --
  0  POSTPONE IF
; IMMEDIATE

: THENS          \ C: 0 orig1..orign -- ; Run: --
  BEGIN
    DUP
   WHILE                \ if not the marker
    POSTPONE THEN               \ resolve the latest orig
  REPEAT
  DROP                          \ discard the marker
; IMMEDIATE
```

As you can see, if takes much longer to describe this technique than to use it. Once these words have been defined, they can be used as described above. This is an example of a key technique in Forth, which is to change the notation to suit the problem at hand.

Immediate words help us to change the compilation notation.

Cautionary notes

The ANS Forth standard defines a compile-time control-flow stack, as indicated by the `C:` in the stack comments for the words above. The majority of Forth systems use the data stack for the control-flow stack. The examples above will not work on systems that do not use the data stack for the control-flow stack. Note also that the standard does not define the size of `orig` or `dest`. We have seen systems with these ranging from one to three cells, varying according to circumstance.

Accessing the compiler

Forth is composed of words and numbers. You can cause a word to be compiled in a definition using the word **COMPILE, (xt --)** just as you can force it to be executed using **EXECUTE (xt --)**, where xt is an execution token as returned by **'** or **['**].

Similarly (but only inside a colon definition) you can force a value (number) to be compiled into the **current** definition using **LITERAL (x --)**. To use **LITERAL** as the compiling primitive you must **POSTPONE** it – **LITERAL** is almost always **IMMEDIATE**.

For example to lay down code that adds a number to a base value, we can define a word **+LIT, (n --)** which compiles code to add a literal to the current top of the data stack. By convention, words whose names end in a comma usually compile code or data into the dictionary.

```
: +lit,         \ n --
```

```
\ Lay down code to add a literal to the top stack item.
  postpone literal  [‘] + compile,
;
```

If your system has an immediate version of **+**, you will have to postpone it. In most cases **POSTPONE** will do what you want, but **COMPILE,** requires you to be careful.

```
: +lit,         \ n --
\ Lay down code to add a literal to the top stack item.
  postpone literal  postpone +
;
```

In many systems, **POSTPONE** and **COMPILE,** will give an error if the compiler has not been turned on. The compiler is turned on by **] (--)** and turned off by **[(--)** which is immediate.

You can use **[** and **]** inside a colon definition to calculate an expression and compile the result as a literal. This was common practice before optimising Forth compilers were available and practice is still used for portable code in some shops.

```
: foo           \ --
  ...
  [ const 55 * ] literal   \ equivalent to “const 55 *”
  ...
;
```

Note that because **LITERAL** is immediate, the result of interpreting the expression is compiled inside **foo**.

In some cases, you will need to know whether the system is compiling or interpreting. The contents of the variable **STATE** indicate whether Forth is compiling (non-zero) or interpreting (zero). Words which have different behaviours in the two cases are referred to as “state-smart”. The use of state-smart words is deprecated by some Forth experts (see below), but it is a convenience to avoid having to know the carnal details of your system. Incorrect use of state-smart words can lead to bugs which are very hard to trace. Just be sensible and cautious when using them. The general rule to avoid problems is never to tick, **POSTPONE** or **[COMPILE]** state-smart words. Although **[COMPILE]** is standardised, you can normally use **POSTPONE** instead.

Be careful with over-enthusiastic use of compilation techniques to improve performance. It can lead to code that is hard to maintain, and in most cases optimising the algorithm can lead to much higher performance than tuning the compiler. The real virtue of these techniques is in changing the notation of your code to suit the application.

Structures revisited

The structures example in the earlier chapter “Defining words” does not provide efficient code in many Forth systems. This version solves the problem of naming the structure first and provides better compilation of fields. We use defining words, **IMMEDIATE** words and the compiler to gain efficiency at the expense of complexity.

Note that the internal mechanisms of Forth compilers vary greatly and that for best results knowledge of the specific compiler is required. The efficiency is gained by using the variable **STATE** to determine whether interpretation of compilation is required.

```
: struct        \ -- addr 0 ; -- size
\ Begin definition of a new structure. Use in the form
\   STRUCT <name>
\ At run time <name> returns the size of the structure.
  create
    here 0 ,  0           \ mark stack, lay initial offset
  does>
    @                     \ get size of structure
;
```

To avoid the overhead of field calculations, the new version of **FIELD** accesses the compiler to lay down the code for **lit +**.

```
: field  \ offset n <"name"> -- offet' ; Exec: addr -- 'addr
\ Create a new field within a structure definition of
\ size n bytes.
  create  immediate       \ the child is IMMEDIATE
    over , +
  does>
    @  state @ if         \ compile
      postpone literal  postpone +
    else
      +
  then
;

: end-struct    \ addr n --
\ Terminate definition of a structure.
  swap !                  \ set size of structure
;

: int          \ <"name"> -- ; Exec: addr -- 'addr
\ Create a new field within a structure definition
\ of size one cell.
  cell field
;

struct record  \ -- addr 0 ; -- size
  int link           \ link to previous structure
  int ident          \ identifier
  int data           \ data value
  16 field name      \ name string
end-struct
```

As in the earlier chapter, when we need a global instance of this structure we can use:

```
Create FirstRec      \ -- addr
  Record allot
```

Or, more prettily:

```
Record buffer: FirstRec    \ -- addr
```

To write a value into the data field, we can use:

```
55 FirstRec data !
```

Because field accesses are passed to the compiler rather than as calls to the field words, optimising compilers have more optimisation opportunities, resulting in smaller and faster code. In writing the definition of **STRUCT** we had to use a defining

word with its albeit small run time overhead when referring to the size of a structure. By applying the technique we used in **FIELD** we can overcome even this.

```
: struct        \ -- addr 0 ; -- size
\ Begin definition of a new structure. Use in the form
\   STRUCT <name>
\ At run time <name> returns the size of the structure.
  Create  immediate
    here 0 ,  0           \ mark stack, lay initial offset
  does>
    @  state @            \ compile literal?
    if  postpone literal  then
;
```

Cautionary notes

"State-smart words are evil"

A full proposition of this statement may be found at http://www.complang.tuwien.ac.at/papers/ as ertl98.ps.gz.

Anton Ertl commented as follows on the examples above: I have revised only the formatting of his replacement code.

FIELD: What's worse than a state-smart word? A defining word that defines **STATE**-smart words. If you want fields to be efficient, define **FIELD** like this:

```
: field  \ offset1 n "name" -- offset2 ; addr1 -- addr2
  over >r
  :  r> postpone literal  postpone +  postpone ;
  +
;
```

and leave it to the inliner to optimize this; this is also much shorter. Another way that field can be optimized is for offset 0:

```
: field  \ offset1 n "name" -- offset2 ; addr1 -- addr2
  over >r
  :
  r> dup if
     postpone literal  postpone +
  then
  postpone ;
  +
;
```

STRUCT has the same problem with STATE-smartness; this could be fixed by letting **END-STRUCT** define a constant with the name given by **STRUCT**. This also can lead to unexpected behaviour, but it's better than STATE-smartness. These problems are what you get for working against the grain of Forth.

So why use state-smart words?

There are indeed problems with state-smart words and the problems they can introduce are often hard to find. Their particular advantage is in reducing the number of words to be defined, and so improving ease of use. If you are introducing a facility that is used under well defined circumstances, state-smart words have their place in your collection of techniques. If you are just using them to provide a performance micro-optimisation, they should be avoided.

For most current Forth compilers, the state-smart versions of the examples above will generate faster code.

Remember, never tick, **POSTPONE** or **[COMPILE]** state-smart words.

12 Errors and exception handling

ABORT, QUIT and ABORT"

ABORT (i*x --) resets the data stack and runs the Forth text interpreter, usually by calling **QUIT**. **ABORT** is essentially a warm restart of the system and in some systems the action of **ABORT** can be modified by applications.

```
<fatal error check>      \ return non-zero on error
IF  ABORT  THEN
```

QUIT resets the return stack, forces input to come from the system console, and runs the Forth text interpreter loop.

ABORT" (x "<text>" --) tests x, and if it is non-zero resets the data stack, usually displays the text as a message, and then runs **QUIT**. If x is zero, the message text is ignored and execution continues.

```
<fatal error check>      \ return non-zero on error
ABORT"Fatal error"
```

In most modern Forth systems **ABORT** and **ABORT"** are implemented using **THROW** below.

CATCH and THROW

Before the ANS specification, Forth lacked a portable nested exception handler. The design of **CATCH** and **THROW** is excellent, and I recommend that they be used to replace the use of **ABORT** and **ABORT"**, which can be defined in terms of **CATCH** and **THROW**.

Description

The following description of the words **CATCH** and **THROW** was written by Mitch Bradley.

CATCH is very similar to **EXECUTE** except that it saves the stack pointers before **EXECUTE**ing the guarded word, removes the saved pointers afterwards, and returns a flag indicating whether or not the guarded word completed normally. In the same way that a Forth word cannot legally play with anything that its caller may have put on the return stack, and also is unaffected by how its caller uses the return stack, a word guarded by **CATCH** is oblivious to the fact that **CATCH** has put items on the return stack.

Here's the implementation of **CATCH** and **THROW** in a mixture of Forth and pseudo-code:

```
VARIABLE HANDLER \ Most recent error frame

: CATCH          \ cfa -- 0|error-code
  <push parameter stack pointer on to return stack>
  <push contents of HANDLER on to return stack>
  <set HANDLER to current return stack pointer>
  EXECUTE
  <pop return stack into HANDLER>
  <pop & drop saved parameter stack ptr from return stack>
```

```
  0
;

: THROW         \ error-code --
  ?DUP
  IF
    <set return stack pointer to contents of HANDLER>
    <pop return stack into HANDLER>
    <pop saved parameter stack pointer from return stack>
    <back into the parameter stack pointer>
    <return error-code>
  THEN
;
```

The description as written implies the existence of a parameter stack pointer and a return stack pointer. That is actually an implementation detail. The parameter stack pointer need not actually exist; all that is necessary is the ability to restore the parameter stack to a known depth. That can be done in a completely standard way, using **DEPTH**, **DROP**, and **DUP**. Likewise, the return stack pointer need not explicitly exist; all that is necessary is the ability to remove things from the top of the return stack until its depth is the same as a previously-remembered depth. This can't be portably implemented in high level, but I neither know of nor can I conceive of a system without some straightforward way of doing so.

Sample implementation

In most Forth systems, the following code will work:

```
VARIABLE HANDLER \ Most recent exception handler

: CATCH         \ execution-token -- error# | 0
                ( token ) \ Return address already
                          \ on data stack
  SP@ >R        ( token ) \ Save data stack pointer
  HANDLER @ >R  ( token ) \ Previous handler
  RP@ HANDLER ! ( token ) \ Set current handler to this
one
  EXECUTE       ( )       \ Execute the word passed
  R> HANDLER !  ( )       \ Restore previous handler
  R> DROP       ( )       \ Discard saved stack pointer
  0             ( 0 )     \ Signify normal completion
;

: THROW         \ ?? error#|0 -- ?? error# ;
        \ Returns in saved context
  ?DUP
  IF
    HANDLER @ RP! ( err# ) \ Back to saved R.
                           \ stack context
    R> HANDLER !  ( err# ) \ Restore previous handler
                  ( err# ) \ Remember error# on
                           \ return stack before
                  ( err# ) \ changing data stack ptr.
    R> SWAP >R    ( saved-sp ) \ err# is on return stack
    SP!           ( token) \ switch stacks back
    DROP          ( )
    R>            ( err# ) \ Change stack pointer
  THEN
\ This return will return to the caller of catch, because
\ the return stack has been restored to the state that
\ existed when CATCH began execution.
;
```

Features

In particular the following features of **CATCH** and **THROW** should be noted.

- **CATCH** and **THROW** do not restrict the use of the return stack
- They are neither **IMMEDIATE** nor "state-smart"; they can be used interactively, compiled into colon definitions, or **POSTPONE**d without strangeness.
- They do not introduce any new syntactic control structures (i.e. words that must be lexically "paired" like **IF** and **THEN**)

To handle the case where there is no **CATCH** to handle a **THROW**, it is wise to **CATCH** the main loop of the application. A different solution, if you don't want to modify the loop, is to add this line to **THROW**:

```
HANDLER @ 0= ABORT" Uncaught THROW"
```

Stack rules for CATCH and THROW

Let's suppose that we have the word **FOO** that we wish to "guard" with **CATCH**. **FOO**'s stack diagram looks like:

```
FOO       \ a b c -- d
```

Here's how to **CATCH** it:

```
<prepare arguments for FOO> ( -- a b c )
['] FOO  CATCH
IF                           ( -- x1 x2 x3 )
  <some code to execute if FOO caused a THROW>
ELSE                         ( -- d )
  <some code to execute if FOO completed normally>
THEN
```

Note that, in the case where **CATCH** returns non-zero (i.e. a **THROW** occurred), the stack **depth** (denoted by the presence of x1, x2 and x3) is the same as before **FOO** executed, but the actual **contents** of those three stack items is undefined. N.B. items on the stack underneath those three items should not be affected, unless the stack diagram for **FOO**, showing three inputs, does not truly represent the number of stack items potentially modified by **FOO**.

In practice, about the only thing that you can do with those "dummy" stack items x1, x2 and x3 is to **DROP** them. It is important, however, that their number be accurately known, so that you can know how many items to **DROP**. **CATCH** and **THROW** are completely predictable in this regard; **THROW** restores the stack depth to the same depth that existed just prior to the execution of **FOO**, and the number of stack items that are potentially garbage is the number of inputs to **FOO**.

Some more features

THROW can return any non-zero number to the **CATCH** point. This allows for selective error handling. A good way to create unique named error codes is with **VARIABLE**s as they return unique addresses without having to worry about which number to use, e.g.

```
VARIABLE ERROR1
VARIABLE ERROR2
```

creates two words, each of which returns a different unique number. For selective error handling, it is convenient to follow **CATCH** with a **CASE** statement instead of an **IF**. Here's a more complicated example:

```
BEGIN
  ['] FOO CATCH
  CASE
    0       OF  ." Success; continuing"   TRUE   ENDOF
    ERROR1  OF  ." Error #1; continuing"  TRUE   ENDOF
    ERROR2  OF  ." Error #2; retrying"    FALSE  ENDOF
    ( default ) ." Propagating error upwards"  THROW
  ENDCASE                   ( retry? )
UNTIL
```

Note the use of **THROW** in the default branch. After **CATCH** has returned, with either success or failure, the error handler context that it created on the return stack has been removed, so any successive **THROW**s will transfer control to a **CATCH** handler at a higher level. It is good practice to define a top level handler for all tasks. For an interactive Forth, the terminal task includes this inside its main loop.

The **CATCH** and **THROW** scheme appealed to people because it is simpler than most other schemes, as powerful as any (and more powerful than most), is easy to implement, introduces no new syntax, has no separate compiling behaviour, and uses the minimum possible number of words (two).

Error codes and return results

Forth words that perform heap requests and I/O to files and other devices, e.g. **OPEN-FILE**, return an "I/O result", abbreviated to **ior** in the ANS Forth documentation. An ior is zero for success. **THROW (n --)** does nothing if n=0. Your application should decide how to handle this ior if it is non-zero. Whether to handle the error locally, e.g. retry, or whether to treat failure as a fatal error is your decision.

If a memory request via **ALLOCATE (size -- addr ior)** fails, it may be that another task is temporarily using memory. In this case it may be worth trying every 20 ms for 100 ms to get the memory before triggering a fatal error. If a fatal error occurs, you can **THROW** up a level. This leads in turn to the question of what error code to use for the fatal **THROW**. In our experience, the best tactic is to use the same codes for iors and for throw codes.

Let us define two versions of a protected version of **ALLOCATE**. The first treats all failures as fatal, whereas the second tries to recover before causing a **THROW**.

```
: ProtAlloc     \ size -- addr
  allocate throw        \ all errors fatal
;

: ProtAlloc     \ size -- addr
  0 0                   \ dummy addr and ior
  5 0 do                \ -- addr ior
    2drop               \ discard previous results
    dup allocate        \ -- size addr ior
    dup 0=              \ exit loop on success
    if  leave  endif
    20 ms               \ wait
  loop                  \ -- size addr ior ; try again
  rot drop              \ -- addr ior
```

```
    throw                \ -- addr
;
```

These examples illustrate that, providing there are no side effects, you can radically change the architecture of a word with **no** impact on the software outside it. By **THROW**ing using an ior as a throw code, your upper level error handler performs the same action in both cases.

The general rule is that you try to handle recoverable errors locally, and fatal errors cause a **THROW**. The handler for the next level up will **THROW** again if it cannot handle the error. The top level error handler, such as the one in **QUIT**, processes all errors.

The ANS standard reserves negative throw codes for the Forth system. In particular, -1 and –2 are reserved for **ABORT** and ABORT" so that they can be implemented in terms of **THROW**. Most modern Forths take this approach. A side effect of this is that **ABORT** is no longer a system-wide warm restart.

Always clean up

Many operation perform a sequence that can be described as:

```
OPEN PROCESS CLOSE
```
or
```
SETUP OPERATE CLEANUP
```

If **PROCESS** or **OPERATE** encounter a fatal error, they may **THROW**. If you do not perform the termination operations, your application may become unstable. This is particularly true of long-running embedded systems and server applications such as firewalls. We used to reboot our email server PC every month or so to recover from memory leaks (unreleased memory allocations). The following code ensures that the **CLOSE** operation above is always performed:

```
: DOIT
  ...
  OPEN  ['] PROCESS CATCH  CLOSE  THROW
  ...
;
```

The ior returned by **CATCH** is consumed by the **THROW** after **CLOSE** has been executed, and if the ior is non-zero another **THROW** will occur. In this way, the **CLOSE** operation is always performed, and problems such as memory leaks and open files will be removed.

13 Files

ANS File Access Wordset

The basis for all file operations comes from the ANS specification wordset for Files. The following group of definitions are implementations of the ANS standard set.

The following data types are used:

fam	"File Access Method", describes read/write permission etc.
ior	"IO Result", A return result from most IO calls, this value is 0 for success or non-zero as an error-code.
fileid	"File Identifier", a handle for a file.

Table 1 : File access data types

Bin `\ fam -- fam'`

Modify a file-access method to include BINARY.

r/o `\ -- fam`

Get ReadOnly fam.

w/o `\ -- fam`

Get Writeonly fam.

r/w `\ -- fam`

Get ReadWrite fam

CREATE-FILE `\ c-addr u fam -- fileid ior`

Create a file on disk, returning a 0 ior for success and a file id.

OPEN-FILE `\ c-addr u fam -- fileid ior`

Open an existing file on disk.

CLOSE-FILE `\ fileid -- ior`

Close an open file.

WRITE-FILE `\ caddr u fileid -- ior`

Write a block of memory to a file.

WRITE-LINE `\ c-addr u fileid -- ior`

Write data followed by EOL. IOR=0 for success.

READ-FILE `\ caddr u fileid -- u2 ior`

Read data from a file. The number of character actually read is returned as u2, and ior is returned 0 for a successful read.

READ-LINE `\ c-addr u1 fileid -- u2 flag ior`

Read an ASCII line of text from a file into a buffer, without EOL. Read the next line from the file specified by fileid into memory at the address c-addr. At most u1 characters are read. Up to two line-terminating characters may be read into memory at the end of the line, but are not included in the count u2. The line buffer provided by c-addr should be at least u1+2 characters long.

If the operation succeeds, flag is true and ior is zero. If a line terminator was received before u1 characters were read, then u2 is the number of characters, not including the line terminator, actually read (0 <= u2 <= u1). When u1 = u2, the line terminator has yet to be reached.

If the operation is initiated when the value returned by **FILE-POSITION** is equal to the value returned by **FILE-SIZE** for the file identified by fileid, flag is false, ior is zero, and u2 is zero. If ior is non-zero, an exception occurred during the operation and ior is the I/O result code.

An ambiguous condition exists if the operation is initiated when the value returned by **FILE-POSITION** is greater than the value returned by **FILE-SIZE** for the file identified by fileid, or if the requested operation attempts to read portions of the file not written.

At the conclusion of the operation, **FILE-POSITION** returns the next file position after the last character read.

FILE-SIZE `\ fileid -- ud ior`

Get size in bytes of an open file as a double number, and return ior=0 on success.

FILE-POSITION `\ fileid -- ud ior`

Return file position, and return ior=0 on success.

REPOSITION-FILE `\ ud fileid -- ior`

Set file position, and return ior=0 on success.

RESIZE-FILE `\ ud fileid -- ior`

Set the size of the file to ud, an unsigned double number. After using **RESIZE-FILE**, the result returned by **FILE-POSITION** may be invalid.

FLUSH-FILE `\ fileid –- ior`

Attempt to force any buffered information written to the file referred to by fileid to be written to mass storage. If the operation is successful, ior is zero.

DELETE-FILE `\ c-addr u -- ior`

Delete a named file from disk, and return ior=0 on success.

INCLUDE-FILE `\ file-id --`

Include source code from an open file whose file-id (handle) is given. After **INCLUDE-FILE** has executed the file will have been closed.

INCLUDED `\ c-addr u --`

Include source code from a file whose name is given by c-addr/u.

INCLUDE \ "<name>" --

A more convenient form of **INCLUDED**. Use in the form "**INCLUDE <name>**". This word is **not** part of the ANS standard, but is widely available. Also called **FLOAD** in some systems. Under operating systems that support spaces in file names, you may be able to use the forms:

```
INCLUDE “my file.f”
INCLUDE ‘my file.f’
```

Simple file tools

This example is derived from code by Wil Baden in his ToolBox series. It covers what is required when loading a complete file into memory and writing a memory block to a file.

```
\ *************
\ *S File tools
\ *************

: FILE-CHECK   \ n --
  ABORT" File Access Error "
;

: MEMORY-CHECK \ n --
  ABORT" Memory Allocation Error "
;

: rewind-file  \ file-id -- ior
  0 0 rot reposition-file
;

0 value pData  \ -- addr ; poi to data block
0 value /Data  \ -- n ; size of data block
0 value hData  \ -- handle ; handle of data file

: InitReadFile \ handle -- size
\ *G Reset the file to the start and return its size.
  dup rewind-file file-check
  file-size file-check  drop
;

: OpenMouth    \ caddr len --
\ *G Open the file for read only.
  r/o open-file file-check  dup to hData
  InitReadFile  to /Data
;

: guzzle       \ file-id -- addr length
\ *G Reads file from disc to HERE without ALLOTing space.
\ ** The file is left open.
  dup InitReadFile              \ -- handle len
  here swap rot read-file file-check   \ -- #read
  here swap                     \ -- addr #read
;

: slurp        \ file-id -- addr length
\ *G Reads the contents of a file into ALLOCATEd memory
\ ** and returns the address and length. Release the
\ ** memory using BURP.
```

```
  dup InitReadFile              \ -- handle len
  dup allocate memory-check     \ -- handle len addr
  dup to pData  dup >r  swap rot \ -- addr len h ; R: -- addr
  read-file file-check          \ -- #read
  r> swap                       \ -- addr #read
;

: Hiccup       \ --
\ *G Close the file opened by OpenMouth.
  hData close-file file-check
;

: BURP         \ --
\ *G Release memory ALLOCATEd by SLURP.
  pData free memory-check
;

: Spit         \ caddr len name namelen --
\ *G Write the memory defined by caddr/len to the file
\ ** whose name is given by name/namelen. A THROW occurs
\ ** on any error.
  r/w create-file throw >r      \ create
  r@ write-file throw           \ write
  r> close-file throw           \ close
;
```

14 Common extensions

Multitasking

Multitasking has been part of Forth for a very long time, and support for it is built into the kernel at the lowest levels. Many systems even reserve a CPU register as a task pointer.

Unfortunately, the ANS 1994 Forth standards team was unable (worn out, not enough time) to reach a consensus on multitasking and therefore this chapter can only sketch out the principles and use vendor-specific examples. Despite this, the approach taken by most vendors and implementers is broadly similar.

Cooperative and Preemptive taskers

In general, hosted systems for operating systems such as Windows, Linux and other Unices take advantage of operating system services. Commercial vendors who also support embedded systems often provide a wrapper layer to provide some source code compatibility with embedded system practice. The multitaskers for operating systems are nearly always pre-emptive - triggered by an interrupt.

Forth embedded systems mostly use cooperative (sometimes called round-robin) multitaskers. In these, programmers have to call a word, usually called **PAUSE**, which schedules the next task. Some Forth programmers do use pre-emptive multitaskers, and these are usually based on cooperative schedulers extended to limit the time slot used by a single task. A few fully preemptive systems have been written.

The choice between the two types comes down to the application domain. Embedded systems can have very heavy CPU loadings and the overheads of a preemptive scheduler may not be suitable. In a heavily loaded system, the simplicity (and hence performance) of a cooperative scheduler can lead to higher service levels than are available from a preemptive scheduler. However, this leads to lack of features which an operating system programmer feels naked without. Provided that you take the trouble to learn the Forth cooperative multitasker and apply a few simple rules, the Forth cooperative multitasker is a wonderfully effective lightweight scheduler.

A four-axis bomb-disposal machine ran twelve tasks using the standard MPE scheduler, and we have heard of systems with up to 400 tasks.

USER variables

The equivalent of what Windows programmers call “thread-local storage” is provided by **USER** variables, usually defined in the form:

```
offset USER <name>      \ -- addr
```

A **USER** variable returns its address. Each task has its own **USER** variable area. Consequently, commonly used system variables such as **BASE** can be set independently for each task. Similarly, each task can have its own communications channel, e.g. in an intelligent serial card handling different protocols. To ease definition of task-specific buffers, and to reduce programmer thinking time, many systems maintain a count of the used size of the **USER** area and provide a facility to add a buffer.

```
size +USER <name>       \ -- addr
```

USER variables are often also provided for high-level interrupt handlers in embedded systems and for operating system callbacks (user-supplied routines called by the operating system). Many Forth implementers consider **USER** variables to be so important that they dedicate a machine register to hold the base address of the **USER** area.

Simple Forth tasks

Before any tasks can be activated, the multitasker must be initialised. This is done with the following code:

```
INIT-MULTI
```

The word **INIT-MULTI** initialises all the multitasker's data structures and starts multitasking. This word need only be executed once in a multitasking system and is usually done automatically at power up.

```
#1000 VALUE DELAY       \ -- n ; time delay between #'s

: ACTION1        \ -- ; task to display #'s
  [CHAR] $ EMIT  \ Display a dollar ($)
  BEGIN                 \ Start continuous loop
    [CHAR] # EMIT       \ Display a hash (#)
    DELAY 0             \ Reschedule Delay times
    ?DO  PAUSE  LOOP
  AGAIN                 \ Back to the start ...
;
```

If we wanted to wait a specific time there is a word **MS (u --)** which waits the given number of milliseconds. In this case we replace the code

```
  DELAY 0               \ Reschedule Delay times
  ?DO  PAUSE  LOOP
```

with

```
  DELAY MS              \ wait x milliseconds
```

To activate (run) the example task, type:

```
TASK TASK1
' ACTION1 TASK1 INITIATE
```

This will activate **ACTION1** as the action of task **TASK1**. Immediately you will see a dollar and a hash ($#) displayed. If you press <return> a few times, you notice that the Forth interpreter is still running. After a while another hash will appear. This is the example task working in the background. You can kill the task using

```
TASK1 TERMINATE
```

Most Forth systems provide facilities to stop a task temporarily, to restart it and to send messages to tasks.

I/O and PAUSE

Any I/O operation that has to wait should call **PAUSE** inside its polling loop so that other tasks are not stalled. The input word **KEY (-- char)** nearly always contains **PAUSE**. Whether **KEY? (-- flag)** calls **PAUSE** is system dependent.

My preference is only to call **PAUSE** when no input is available. This optimises system throughput under heavy load by reducing the overall number of **PAUSE**s, but may require additional code if an I/O activity can cause overload. The sample implementation of **KEY** uses **KEYPRIM (-- char)**, a primitive word that blocks.

```
: key          \ -- char
  begin
    key? 0=
   while
    pause
  repeat
  keyprim
;
```

Error checking

The most common fault is a stack fault. Since a task is an endless loop it is simple to put stack depth checks in the main loop. A simple task with checking is shown below.

```
: TASK-ACTION    \ --
  sp@ s0 !              \ store stack base
  <initialisation>
  BEGIN
    <body of task>
    depth IF            \ non-zero if anything there
      s0 @ sp!
      <warn programmer!>
    ENDIF
    PAUSE
  AGAIN
;
```

You should also consider checking for **THROW**s.

```
: StackCheck     \ ?? --
  depth if              \ non-zero if anything there
    s0 @ sp!  <warn programmer!>
  then
;

: TASK-ACTION    \ --
  sp@ s0 !              \ store stack base
  <initialisation>
  BEGIN
    ['] BodyOfTask catch if
      <warn programmer!>
    then
    StackCheck
    PAUSE
  AGAIN
;
```

Floating point

ANS Forth standardises floating point handling within a framework which leaves some flexibility for the implementer. See the ANS Forth standard for more details.

The internal storage format of floating point numbers is **not** defined by the standard as being IEEE format because of the number of non-IEEE formats in use, especially by DSP (Digital Signal Processing) chips.

Many implementations do not keep floating point numbers on the Forth data stack. Instead they use a separate floating point stack in memory to avoid stack gyrations or they use the internal stack of a hardware floating point engine such as the NDP of Intel desktop PCs. This is referred to as a **separate** float stack. Using the Forth data stack is referred to as a **combined** stack.

Traditionally Forth programmers have used floating point less than C programmers. Forth's facilities for mixed and double integer made it easy to construct multiple precision operators. This situation has changed after the floating point performance of desktop PCs improved to the point that some floating point operations are now faster than some integer operations. However, this does not mean that floating point is suitable for all applications requiring a very wide dynamic range. Floating point calculations can suffer from imprecise calculation and rounding errors. During the construction of the Hong Kong airport, it was reported that the cost difference in the estimation for one part of the concrete was over ten million dollars between a calculation using 128 bit integers and a calculation using 64 bit floats. The integer calculation was the accurate one.

Anton Ertl had this to say about the concrete calculation. His comments emphasize the point that use of floating point can require caution with any programming language.

"About the concrete calculations: The total volume of the project was probably on the order of 1,000-10,000 million dollars, so an error of 10 million is 0.1%-1%. That's not the result of plain floating-point rounding errors (which are on the order of 1e-16 for 64-bit FP operations), or even accumulated errors (it would need 1e13 FP operations with worst-case rounding error accumulation to get there, or 1e26 FP operations with expected error accumulation). It's the result of a numerically unstable algorithm.

FP is probably more surprising to programmers in that respect than fixed point, though; e.g., if you calculate using integer numbers, you know that you will never get any fractional result, whereas it surprises programmers when this happens with FP numbers."

The Forth Scientific Library (FSL) project originally coordinated by Skip Carter provides a huge range of scientific calculation routines, mostly using floating point. The FSL is supplied as part of several systems.

Local variables

A set of words for local variable handling is provided by ANS Forth. Sadly, the standard does not provide what is required for interfacing with operating systems such as Windows, especially in handling local arrays. It also uses a very clumsy notation that is hard to read. However, it did follow practice of the time. The most popular alternative notation is described below. Not all systems provide it and several variants exist.

***MPEism**: The description below is of the MPE implementation, which is broadly similar to many others.*

The sequence

```
: <name> { ni1 ni2 ... | lv1 lv2 ... -- o1 o2 }
  …
;
```

defines named inputs, local variables, and outputs. The named inputs are automatically copied from the data stack on entry. Named inputs and local variables

can be referenced by name within the word during compilation. The output names are dummies to allow a complete stack comment to be generated.

- The items between **{** and **|** are named inputs.
- The items between **|** and **--** are local variables.
- The items between **--** and **}** are outputs.

Named inputs and locals return their values when referenced, and must be preceded by **TO** to perform a store, or by **ADDR** to return the address.

Arrays may be defined in the form:

```
arr[ n ]
```

Any name ending in the '[' character will be treated as an array, the expression up to the terminating ']' will be interpreted as the size. Arrays always return their base addresses, all operators are ignored.

In the example below, a and b are named inputs, **a+b** and **a*b** are local variables, and **arr[** is a 10 byte array.

```
: foo           { a b | a+b a*b arr[ 10 ] -- }
  arr[ 10 erase
  a b + to a+b
  a b * to a*b
  cr a+b .  a*b .
;
```

Cautionary notes

In the following discussion the term “Forth locals” refers to both named inputs and local variables.

Writing C in Forth: Although use of Forth locals can be valuable for local arrays and readability, there is a great danger for C programmers learning Forth to overuse them. There are some in the Forth community who believe that local variables have no place in a course or text until the second or third level. I have seen enough Forth that looks like C to know that there is a real problem. Because of the benefits when interfacing to modern operating systems and in certain classes of problems, I reluctantly decided to include them.

Excessive use of Forth locals inhibits learning how to use the data stack efficiently and reduces the incentive to factor into small definitions. In turn, that leads to "cut and paste" errors and to bigger code, which further leads to difficulties in maintenance and debugging. I also note that programmers who use Forth locals heavily tend not to use defining words and other more advanced Forth techniques.

Performance: The Forth virtual machine is optimised for two stacks, and the code generation of modern Forth compilers reflects this. Especially on CPUs with more than eight registers, good stack code is faster and smaller than code with heavy use of local variables.

I recently overhauled parts of a TCP/IP stack and removed Forth locals where possible. After testing on an ARM (16 registers) embedded system, the size of rewritten words reduced by 20% and performance improved by 50%. In particular cases code size reduced by 50% or more. Code size improves because the compiler makes better use of CPU registers and performance improves because of smaller code and reduced memory traffic. Although less dramatic we have similar results for

most CPUs. For CPUs with 32 or more registers, e.g SPARC and PowerPC, Forth compiler writers can easily use registers for local variables.

Writers of Forth compiler are unlikely to put in a huge effort to optimise bad code. After an earlier release of this book, the following comments were made on the comp.lang.forth newsgroup comparing C and Forth compilers.

Anton Ertl: *"You might be surprised; as long as the bad code occurs in sufficiently important benchmarks, they are very likely to put in a huge effort. However, in the case of Forth, optimizing stack code will benefit all the code out there (including code using locals), whereas optimizing locals will only benefit a minority of code; it should come as no surprise that Forth compiler implementors will concentrate on optimizing stack accesses first."*

Andrew Haley: *"Well, it's not just that: we can't tell whether the 'bad' code has been written by a programmer or is the output of a previous compilation pass. So, we optimize everything we can, even if it's something no sane programmer would ever write."*

Avoiding locals: The main reason that people feel they need locals is having too many items in use on the data stack. There are three ways to avoid this:

1) Re-factor into small definitions that use fewer items on the stack,

2) Use the return stack to hold the least commonly used items,

3) Where items are pairs or larger, for example x/y coordinate pairs for points or x/y/w/h sets for a rectangle description, consider keeping them in structures and passing pointers to the structures. Although pointers increase memory traffic, they considerably reduce stack traffic at word entry and exit.

We rewrote an embedded GUI package after a client requested changes. The original code had been written with extensive use of Forth locals. After reorganisation and overhaul, only one word used local variables. The code is shorter both in terms of lines of source code and in terms of compiled code size. The code is easier to maintain.

When to use locals:

1) To avoid stack repetition of complex calculations. Some calculations have common sub-expressions with reused intermediate results. Storing these on either stack can lead to "stackrobatics". In performance and code size, local variables are cheap compared to named inputs. Defining local variables for these can be very effective.

2) For temporary small buffers. The alternatives to local buffers are a heap, global or task-based (thread local) structures. Although heap functions are widely available and standardised, they require code, have performance and reliability penalties (heap leaks are not unknown in any language) and require great care in exception handling. Local buffers are automatically discarded on exit from a word, and, in every Forth implementation I have inspected, they are completely compatible with **`CATCH`** and **`THROW`**.

Object oriented programming

A plethora of object oriented extensions exist for Forth ranging from 16 to several thousand lines of source code. The Neon (later Actor) OO programming language was itself written in Forth. This model has probably been the most widely used and is

available for several desktop Forth systems. Embedded system models also vary widely.

There is not yet enough common practice to make further examples and discussion worthwhile.

Integrated assembly

The majority of Forth systems written for a specific CPU (not written in another high level language) include an assembler for the CPU and facilities to create words written entirely in assembler. Such words are called **CODE** words. As with C compilers there is a wide variety of notations and operand orders.

CODE definitions were widely used by indirect (ITC) and direct-threaded (DTC) systems to improve the performance of performance-critical inner loops. With optimising native code compilers this is far less necessary. Our Windows Forth uses only about 30 routines coded in assembler, mostly to do with start up and interfacing with the operating system. In embedded systems, **CODE** routines are reserved for accessing special CPU registers such as the status register (e.g. for interrupt enabling and disabling), for the occasional interrupt handler, and for the scheduler in the multitasker.

The embedded systems chapter contains several examples of **CODE** definitions.

Source location

As applications get larger, the ability to browse source code rapidly becomes more important. Many Forth systems provide a way to find the source code of a word, typically using a phrase such as

```
LOCATE <name>
SEE <name>
VIEW <name>
```

where **<name>** is the name of the word whose source code is required. The source code is usually displayed by an internal or external editor. If the source code is not available, the code may be disassembled or decompiled.

Mixed language programming

It is a sin to reinvent the wheel unless it is just for the pleasure of exploration. Perfectly good code exits in a wide range of languages. Reusing code is usually cheaper than rewriting it. For Forth systems running under an operating system the most common solution is to provide an interface to shared libraries or Windows DLLs, commonly the same mechanism required to access the operating system itself. For embedded systems, the issues revolve around object file formats.

Parameter passing

The first issue is to work out how to translate Forth stack items into the form required by the other language. We will use the basic Windows **SendMessage** API call as an example.

```
int PASCAL SendMessage(
  HWND hwnd, WORD msg, WPARAM wParam, LPARAM lParam
);
```

To call this from Forth, we can define it in two ways.

```
SendMessage      \ hwnd msg wparam lparam -- int
```

or

```
SendMessage      \ lparam wparam msg hwnd -- int
```

Which choice you make affects how the Forth parameters have to be organised when they are passed to the C system. Depending on the operating system and compiler, parameters are passed in registers or on a stack. If parameters are passed in registers, e.g. ARM or SPARC CPUs, there will be a limit on the number that are passed in registers, and additional parameters will be passed on a stack. The order in which parameters are passed will depend on the language.

The next stage is to write an interface routine which takes the parameters from the Forth data stack and an address to call, performs the call, and finally returns the result if there is one. We usually call this routine **XCALL**. These interface routines require the use of specific CPU registers and so are best written in assembler. It is well worth while writing a Forth defining word or custom parser to handle defining the Forth side of the call. How you do this will influence how you write the interface routine or routines and you should design and code them together.

For readability, it is common to keep the Forth stack order and C parameter order visually the same in order to reduce programmer typing errors.

DLLs and shared libraries

Once you have decided on the parameter passing and notation, you then have to determine how to load the library, find the address of a routine in a shared library, and release the library. This is operating system dependent and often requires a special section in the executable file. Operating system documentation at this level is often badly organised, tedious and obscure. For Windows, the API routines you need are called **LoadLibrary**, **GetProcAdress** and **FreeLibrary**.

Once you have these three functions, you can now make a routine that searches for functions by name (probably case sensitive) and saves the function addresses in the right places. Your defining word or parser can produce the right data for the XCALL routine above. A typical Forth definition for **SendMessage** above could be:

```
4 1 libfunction: SendMessage
```

which defines that there are four input parameters and one output. Since most C functions always return a value it is not uncommon to omit the output parameter. If you are using a custom parser to handle interfacing, you can go as far as:

```
EXTERN: int PASCAL SendMessage(
  HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam
);
```

which is the MPE VFX Forth description of the **SendMessage** API call.

Static linking

In an embedded system with no operating system, you will have to do the work of linking and relocation yourself. This requires processing compiler object files and link maps. It can be done, but is compiler specific and always tedious. It is only recommended if you can only get a library as an unlinked object file. The general process is:

1) Define the parameter passing and interface routine as above,

2) Work out how to perform relocation,

3) Work out how to get to the procedure names and find the function addresses,

4) Define the Forth interface to each function,

5) Sort out the start up requirements of the library – this can be a major pain.

6) Combine the Forth and C binaries.

The biggest maintenance issue here is that if you receive a new version of the object library, you will have to go through stages 3) to 5) all over again. Far and away the simplest approach is use the jump table approach below.

Jump tables

The basis of this approach is to provide a table of function addresses in the C module and link it at a fixed address. The Forth interface requires the same **XCALL** routines as in the previous approaches. Forth calls the relevant fixed address in the table. No relocation is required as the C linker has done this for you. In many cases, calling a dummy **main** function from Forth at power up will avoid most of the initialisation problems. You can always add explicit initialisation into **main**.

What you lose from this approach is memory because you have separated and fixed the Forth and C binaries. You lose memory for the table itself and you may lose a few clock cycles more in performing the call. To avoid compiler issues, you are likely to have to write the function table in assembler. However, simplicity and reliability make jump tables the approach of first choice for embedded systems.

15 Embedded Systems

The major difference between embedded system programming and programming for an operating system is that you have to do everything yourself. You have to be able to specify exact memory locations for ROM, Flash, RAM and peripherals. You are also likely to be tightly constrained by limited RAM and an underpowered CPU. You will become very good at re-reading large datasheets and treating their contents with some scepticism.

Code for embedded systems is usually cross-compiled on a PC and the final code is downloaded to the target system. Forth cross compilers vary widely in capabilities and code generation quality. However, the commercial vendors supply cross compilers for a very wide range of CPUs and hardware implementations. Forth cross compilers are also used when porting hosted Forths between operating systems, e.g. from Windows to Linux. Cross compilers can usually generate a complete Forth interpreter and compiler on your target system, or can compile a cut-down system which is interactively debugged over a serial line or other channel. These Umbilical (or tethered) systems are discussed later in this chapter.

Embedded systems programming is an area in which Forth scores very well. The use of interactive debugging directly on the target permits very rapid development and considerably reduces the need for additional (and expensive) tools such as In-Circuit Emulation (ICE). We brought up a large StrongARM SA-1110 system using a JTAG Flash programmer, an oscilloscope and a soldering iron. With more modern CPUs that incorporate a boot loader, the JTAG programmer can be dispensed with.

Apart from controlling a mass spectrometer, this StrongARM system has an Ethernet connection and runs a full TCP/IP stack and multi-threaded web server with CGI and ASP. Embedded Forth is not just for little systems.

Although Forth cross compilers vary widely, there is a draft ANS Forth cross compiler standards proposal by Elizabeth Rather which has been adopted by MicroProcessor Engineering and Forth Inc and some others. The proposal was the result of both companies working on parts of the same project. This chapter assumes the use of the proposed standard. This text is somewhat informal, so for definitive definitions consult the PDF files on the CD.

Defining and using memory

Embedded systems require at least three **types** of memory; code in ROM or Flash, normally read-only; RAM that is initialised at power up or reset; and RAM that is uninitialised at power up or reset. These three types are known as **CDATA**, **IDATA**, and **UDATA** respectively in ANS terminology.

xDATA \ --

Select this type of memory as the one to operate on next. This will affect the operation of the words listed below.

, (comma) **ALIGN ALIGNED ALLOT C, HERE UNUSED**

The defining word **SECTION (start end --)** creates a word known only to the cross-compiler (not to the target code) that names a piece of memory.

```
HEX
8000 FFFF CDATA SECTION PROGRAM  \ Program in Flash
```

```
0800 08FF IDATA SECTION IRAM     \ Initialized RAM
0900 0BFF UDATA SECTION URAM     \ Uninitialized RAM
0400 07FF UDATA SECTION EEPROM   \ Uninitialised EEPROM
0000 01FF UDATA SECTION PERIPHS  \ peripherals
DECIMAL
```

Most cross compilers attach the used portion of the **IDATA** images to the end of a **CDATA** section and the start up code will copy it into RAM. **UDATA** sections will not be modified at start up unless you place stacks and other data there.

Other vendor-specific functions are available to support pages/banks of memory for CPUs such as the 9S12 and Rabbit 8/16 bit CPUs.

Referencing a section name during interpretation makes that section the current section of that type and selects the type as the current type.

```
DECIMAL
IRAM
42 value answer      \ -- n ; from previous question
URAM
#100 buffer: databuff      \ -- addr ; uninitialised
EEPROM
variable BaudRate    \ -- addr ; holds baud rate
variable UPort       \ -- addr ; holds UART port address
CDATA
Create Powers2       \ -- addr ; powers of two
  $01 c,  $02 c,  $04 c,  $08 c,
  $10 c,  $20 c,  $40 c,  $80 c,
```

The example above declares different variables in specific RAM sections, and a data table in the current **CDATA** section. If your cross compiler forces **VARIABLE**s to be in **IDATA**, try either of the following:

```
create BaudRate  cell allot      \ -- addr ; holds baud rate
cell buffer: BaudRate            \ -- addr ; holds baud rate
```

Harvard targets

Some CPUs, notably 8051, Z8 and many DSPs use a Harvard architecture in which code and data are in separate address spaces. For some of these architectures, e.g. 8051, code space is not writable by CPU instructions. For these systems **CDATA** corresponds to the code address space, and **IDATA** and **UDATA** to the data address space.

Memory access words that access code space are usually called
c@c w@c @c c!c w!c !c cmovec

Compiler and Interpreter extensions

Forth cross-compilers allow you to extend the compiler/interpreter itself by controlling where new words are placed. After cross-compilation is started, all new words are placed by default into the target image. The following directives control where new words are placed.

Directive and corresponding vocabulary	Action
TARGET ***TARGET**	New words are placed in the target image Conceptual search order: ***TARGET**
COMPILER ***COMPILER**	New words are added to the cross-compiler's compile time behaviour. These words act like **IMMEDIATE** words in conventional Forth, but are not available during interpretation. All memory access words refer to the target. Conceptual search order: ***COMPILER *HOST**
INTERPRETER ***INTERPRETER**	New words are added to the cross-compiler's interpret time behaviour. These words are not available during compilation. All memory access words refer to the target. See the next section on defining words for details of the actions for defining words using **CREATE … DOES>** or **CREATE … ;CODE.** Conceptual search order: ***INTERPRETER *HOST**
ASSEMBLER ***ASSEMBLER**	New words are added to the cross-compiler's assembler. This directive is usually used to add macros to the assembler. Also searches the **INTERPRETER** words. Conceptual search order: ***ASSEMBLER *INTERPRETER *HOST**
HOST ***HOST**	Exposes the underlying host portion of the cross-compiler so that utility words can be added that will be used later by words defined using **COMPILER INTERPRETER** or **ASSEMBLER**. Use of this mode is at your own risk. Finish this mode with **TARGET**. Conceptual search order: ***HOST**

Table 2: Compiler extension directives

It is a convenient conceptual model to regard these directives as corresponding to vocabularies called ***TARGET *COMPILER *INTERPRETER *ASSEMBLER** and ***HOST**. The table shows the conceptual search order generated by the directives.

Defining words

Defining words can be handled in two ways, in some cases automatically by the cross-compiler, or explicitly using the extension mechanism discussed above. The objectives behind the two mechanisms are different.

The automatic mechanism aims to be transparent, so that code for the cross-compiler can be the same as that for a hosted Forth. This encourages portability and makes the cross-compiler easier to use for the majority of defining words. The automatic mechanism copes with the majority of defining words.

The explicit mechanism provides very fine control of the host and target environments, but can be more confusing to use.

Automatic handling

***MPEism**: Note that not all cross compilers support automatic handling of defining words. The examples here are written for the MPE Forth 6 cross compiler.*

The cross-compiler will automatically build an analogue of the defining word in the host's conceptual ***INTERPRETER** vocabulary up to the terminating **; DOES>** or **;CODE**. This is triggered by the word **CREATE**. Consequently, any code between the **:** and the **CREATE** will not have a host analogue. The words between **CREATE** and the terminating **DOES>** or **;CODE** must either be in the ***INTERPRETER** vocabulary or must be target constants or variables, which allows construction of linked lists that refer to target variables.

A target version of the defining portion up to **DOES>** or **;CODE** is built if the target words has heads.

The run-time portion of the code is always placed in the target.

Construction of the host analogue is inhibited between the directives such as **TARGET-ONLY** and **HOST&TARGET**.

Both the defining words below can be handled automatically by MPE cross-compilers

```
: CON            \ n -- ; -- n ; a constant
  CREATE
    ,
  DOES>
    @
;

VARIABLE LINKIT                \ exists in target

: IN-CHAIN       \ n -- ; -- n ; constants linked in a
chain
  CREATE
    ,                                  \ lay down value
    HERE  LINKIT @ ,  LINKIT ! \ link to previous
  DOES>
    @
;
```

Explicit handling

Explicit handling uses the compiler directives discussed in the previous section to control how defining words are created. This is particularly useful for more complex words, and where no target version of the defining word is required, as is often the case when an Umbilical Forth target is being used.

The examples from the automatic handling section are repeated here using the explicit mechanism. The words **@(H)** and **!(H)** are used because **LINKIT2** is in the host.

```
INTERPRETER

: CON            \ n -- ; -- n ; a constant
  CREATE
    ,
```

```
  DOES>
    @
;

VARIABLE LINKIT                \ exists in target

: IN-CHAIN      \ n -- ; -- n ; constants linked in a
chain
  CREATE                       \ only in host
    ,                          \ lay down value
    HERE  LINKIT @ ,  LINKIT ! \ link to previous
  DOES>                        \ run time in target
    @
;

HOST

VARIABLE LINKIT2               \ exists in host

INTERPRETER

: IN-CHAIN2     \ n -- ; -- n ; link variable in host
  CREATE                       \ in host
    ,
    HERE  LINKIT2 @(H) ,  LINKIT2 !(H)
  DOES>
    @
;

TARGET
```

As can be see from the examples above, the automatic handling mechanism is simpler, but the explicit handling mechanism permits finer control over where code is generated, which may be useful when defining words are required and the absolute minumum of target memory is to be used.

Compiler macros

There are often occasions in which short pieces of code are repeated, but should really be compiled inline to allow the code generator to optimise them further or to avoid a call/return performance penalty. The following example is useful for stepping through an array of integers.

```
: @++           \ addr -- addr+cell x
\ Fetch the contents of addr and increment addr.
  dup @  swap cell + swap
;

: !++           \ addr x -- addr+cell
\ Store x at addr and increment addr.
  over !  cell +
;
```

You can extend the compiler to treat these as macros as follows:

```
compiler

: @++           \ addr -- addr+cell x
  dup @  swap cell + swap
;

: !++           \ addr x -- addr+cell
  over !  cell +
```

```
;

target
```

When the following code is compiled, the macros will be expanded, regardless of whether or not target definitions of **@++** and **!++** already exist.

```
: CopyInts      \ src dest #ints --
  0 ?do                 \ -- src dest
    swap @++ rot !++
  loop
  2drop
;
```

I/O ports

Memory mapped I/O is handled using the normal Forth memory operators. In some cases, peripherals are reset by reading a register and discarding the result:

```
<address> @ drop
```

Extremely aggressive compilers, e.g. the MPE 68xxx and 386 VFX compilers, can optimise such phrases away. In these cases, synonyms such as **P@** in place of **@** ('P' for Port) will be available that cannot be optimised away.

Where code such as device drivers is intended to be portable across different hardware, use of port access words makes portability much easier when code is moved to a CPU with a separate I/O space, e.g. Rabbit or 386. The standard port access words are:

```
PC@ PW@ PL@ P@ PC! PW! PL! P!
```

The C/W/L denote 8/16/32 bit access and **P@** and **P!** denote a native cell. For memory mapped systems you can always alias these words to an existing word using compiler macros.

All Forth cross compilers we have come across (at least for embedded work) provide a facility to name a number. This permits the names of numbers to be held in the compiler without using any space in the target except when used, in which case they will treated just like the number they represent. They are equivalent to conventional assembly language equates and are defined like **CONSTANT**s.

```
<value> EQU <name>
```

Interrupt handlers

Because of the wide variety of CPU architecture and peripherals, it is in practice almost impossible to provide a universal word set for interrupt handlers. For example, vectored interrupt controllers for ARMs come in at least four forms. However, the principles are essentially the same for all CPUs.

For any interrupt you must first of all decide whether to use an assembler coded interrupt service routine or a high level Forth routine. High-level interrupt service routines take more instructions (and CPU time) to set up, but are very convenient to use.

The phrases of the form:

```
[ TASKING? ]
```

in a definition temporarily turn off the compiler to interpret **TASKING?** in the middle of the word. The flag returned is used the immediate words **[IF] [ELSE]** and **[THEN]** for conditional compilation.

Assembler interrupt

The following example is for an MSP430 running at 32kHz.

```
variable rx0-char    \ -- addr ; low byte holds KEY character
rx0-char 1+
  equ rx0-avail      \           high bye holds KEY? flag

proc Rx0-isr   \ --
  mov .b  & u0rxbuf  & rx0-char  \ stash character
  mov .b  # -1  & rx0-avail      \ flag character available
  bic     # _cpuoff 0 (sp)       \ return with cpu active
  reti
end-code
rx0-isr USART0rx_vec !           \ set vector

code init-ser  \ --
  or .b   # $c0 & me1            \ enable USART0 tx and rx
  or .b   # $30 & u0tctl         \ baud rate source is smclk
  mov .b  # $a1 & u0br0          \ 9600 baud
  mov .b  # $01 & u0br1
  mov .b  # $03 & u0mctl
  mov .b  # $30 & u0ctl          \ 8N2, clear reset
  or .b   # $30 & p3sel          \ P3.4,5 = USART0 TXD/RXD
  or .b   # $10 & p3dir          \ P3.4 output dirction
  mov     # 0 & rx0-char         \ no characters received yet
  bis .b  # $40 & ie1            \ enable UART 0 RX interrupt
  bic .b  # $80 & BCSCTL1        \ XT2 = HF XTAL
  bis .b  # $88 & BCSCTL2
  eint                           \ global enable interrupts
  ret
end-code

: key?0        \ -- flag ; true if USART0 received a character
  rx0-avail c@ 0<>
;

: key0         \ -- char ; wait for character from USART0
  begin
    di  rx0-avail c@ 0=
   while
[ tasking? ] [if]
    ei pause                     \ schedule
[else]
    \ cpu to sleep, GIE set
    [asm  bis # _cpuoff _gie + sr  asm]
[then]
  repeat
  rx0-char c@  0 rx0-avail c!
  ei
;

: emit0        \ char -- ; send character through USART0
  begin  u0tctl c@ 1 and  until  \ wait for empty buffer
  u0txbuf c!                   \ write buffer
```

```
;

: type0        \ c-addr len --
  bounds
  ?do  i c@ emit0  loop
;

: cr0          \ --
  $0D emit0  $0A emit0
;
```

High level interrupts

The following code is taken from the serial driver for an Analogue Devices AduC7020 serial driver.

```
: >RxQ         \ char queue --
\ *G Put character from UART into queue.
  dup (cqFull?)                  \ if queue full
  if  2drop  exit  endif         \  discard character
  (>cqueue)                      \  queue character
;

: FIFO>RxQ      \ base queue --
\ *G Given a UART base address and a character queue,
\ ** empty the UART FIFO into the queue.
  begin
    over ComSta0 + @ $01 and
   while
    over ComRx + @ $FF and over >RxQ
  repeat
  2drop
;

: ser-isr      \ --
\ *G UART0 high level ISR.
  _INTCON IrQSta + @ SerialBit and if
    case  _UART dup ComIId0 + @ 7 and
      0 of  ComSta1 + @ drop  endof    \ MSR for UART
      2 of  endof                      \ Tx interrupt
      4 of  SerInpQ FIFO>RxQ  endof    \ read characters
      6 of  ComSta0 + @ drop  endof    \ Line Status interrupt
            nip
    endcase
  endif
;
' ser-isr add-irq

: seremit      \ char --
\ *G Transmit a character on UART.
  _UART
  begin
    dup ComSta0 + @ %00100000 and
  until
  ComTx + !
;

: serkey?      \ -- t/f
\ *G Return true if the UART has a character available.
  SerInpQ cqnotempty?
```

```
;

: serkey       \ -- char
\ Wait for a character on the UART and return it.
  SerInpQ cqueue>
;

: serTYPE      \ c-addr len --
\ *G Type a string to the UART.
  bounds
  ?do  i c@ seremit  loop
;

: serCR        \ --
\ *G Issue a CR/LF pair to the UART.
  $0D seremit  $0A seremit
;
```

Interlocks

The MSP430 example illustrates how to avoid a common source of error when programming code that runs in the main code and extracts information from data stored by an interrupt handler. If you do not **carefully** control the interrupts, the interrupt handler may modify data while the main code assumes that it is stable. The MSP430 example is repeated below.

```
: key0         \ -- char ; wait for character from USART0
  begin
    di  rx0-avail c@ 0=
   while
[ tasking? ] [if]
    ei pause                     \ schedule
[else]
    \ cpu to sleep, GIE set, in assembler
    [asm  bis # _cpuoff gie + sr  asm]
[then]
  repeat
  rx0-char c@  0 rx0-avail c!
  ei
;
```

The following code is for the four words used to control interrupts. The words **DI** and **EI** are simple global interrupt enables and disables. The words **[I** and **I]** define a section in which interrupt status is preserved to and restored from the return stack. Inside the section interrupts are disabled. These four words are present in all MPE systems. Equivalents will exist in systems from other vendors. Because these words are CPU and implementation dependent, they must be written in assembler. In some systems, optimised versions may be available in the code generator.

```
code ei         \ -- ; enable interrupts
\ *G Global enable interrupts.
  eint    ret
end-code

code di         \ -- ; disable interrupts
\ *G Global disable interrupts.
  dint   ret
end-code
```

```
code [I        \ R: -- x
\ *G Preserve status on return stack, disable interrupts.
\ ** The state is restored by I].
  pop    r14
  push   sr
  dint
  br     r14
end-code

code I]        \ R: x -- ; restore I status from r. stack
\ *G Restore status saved by [I from the return stack.
  pop    r14
  pop    sr
  br     r14
end-code
```

Block I/O

Forth handles files on any given computer in much the same way as any other language running on that computer. The words used to open and close files are discussed elsewhere. However, Forth also has a system called `blocks'.

Although blocks are no longer often used by Forths hosted on modern operating systems, blocks can be very useful on embedded systems for data logging, especially in a 16 bit Forth. Blocks can be implemented in paged or linear memory, serial EEPROM or data Flash, over a serial line or network to a host, or a physical disc.

Blocks are units of memory loaded from and saved to what is notionally mass storage. The data size of a block is normally 1024 bytes (1K). The system can work with only one block buffer, but two or more are better. Blocks are accessed by a number 0..u.

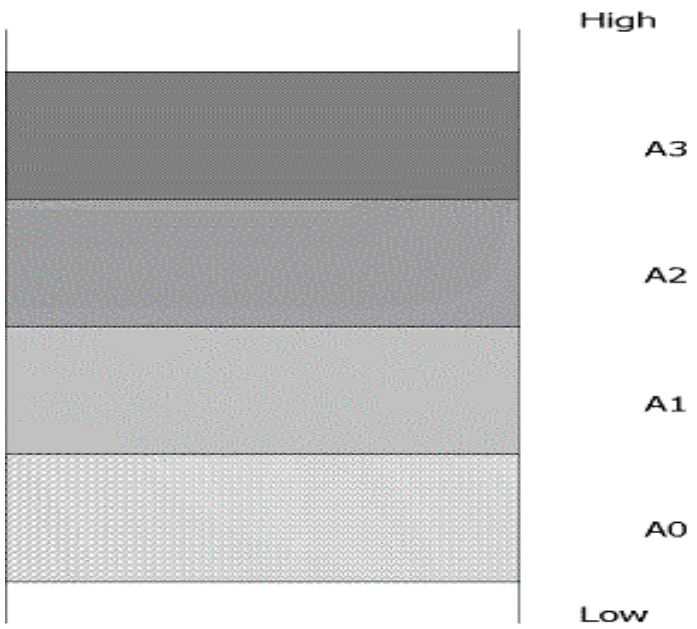

Figure 4: Block buffers

The addresses A0 to An can be obtained by entering:

```
0 block  .
1 block  .
etc.
```

These each return the address of the first byte in their respective buffers. Block buffers act as a memory cache between the mass storage and the Forth system. Different Forth systems use various strategies for allocating and reusing buffers. The most common are round robin and least recently used (LRU).

If your data can be neatly partitioned into 1kb units, blocks can provide extremely good performance. Blocks can be implemented in a filing system or can simply be treated as raw sectors on a disc. Even above a filing system, blocks can give a surprising performance improvement at the expense of a little management code.

BLOCK `\ u -- a-addr`

Load the uth block of 1K from mass storage into a buffer, and return the address of the buffer.

BUFFER `\ u -- a-addr`

Reserve a data buffer, and return the address of the buffer. The contents of the buffer are undefined. If you write this buffer back to mass storage, the previous contents in storage will be lost.

The block data can now be used and modified. When the data has been modified, you must mark the block as modified using **UPDATE**.

UPDATE `\ --`

Mark the current buffer as modified. If the current buffer has to be reused for another block, the contents will be written out before reuse.

The system knows that an update has occurred because an update flag is set. **UPDATE** does not actually save the block to disc, but tells the system that it ought to be written next time the buffer is reallocated, i.e. next time a block is written to that buffer.

In most systems, only one buffer will be written to disc each time and so a disc or file will never be closed neatly. However, there are three words to do the job nicely.

FLUSH `--`

This will save any **UPDATE**d buffers to disc and unassign all the buffers. This means that the next time a block is loaded into a buffer, the computer takes the buffer as being empty. Flush is made up of the words **SAVE-BUFFERS** and **EMPTY-BUFFERS**.

SAVE-BUFFERS `--`

This will write to any disc any buffers marked as **UPDATE**d, but will not unassign them. If the block is accessed again it will not be loaded into memory, because the computer will still think the block is there and in fact it will be.

EMPTY-BUFFERS `--`

This word actually unassigned the buffers, and after this word has been executed, any reference to a block will cause the block to be loaded into memory first.

Source in blocks

Before GUI operating systems and cheap PCs, many Forth systems kept source code in blocks. The source code was displayed on-screen as 16 rows of 64 column lines, matching the 1024 byte storage.

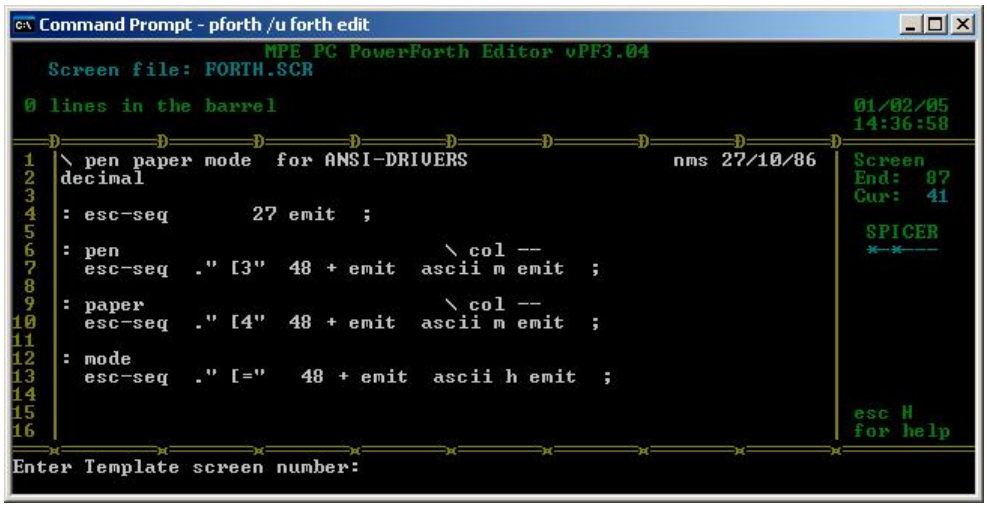

Figure 5: Source in blocks

The use of blocks for source code is rare in modern systems except for embedded systems which require local configuration and editing without the presence of a PC. By changing the default block size, some ingenuity with keys, and familiarity with mobile phone text messaging, it is perfectly possible to edit source code using a hexadecimal keypad and a 4 by 16 character LCD display. Desperate problems lead to desperate solutions.

Umbilical Systems

A conventional Forth system includes its own text compiler and interpreter. In smaller embedded systems, there is not enough Flash or RAM for this, or cheaper chips can be used if less Flash and RAM is required. In addition, a PC can compile much faster than an 8051. To avoid losing the interactivity that makes Forth so attractive, the standalone compiler/interpreter on the target is replaced by a message passer which executes commands (words) as instructed by the host PC.

Now all the interaction is provided by the cross-compiler running on the PC. The cross compiler can take advantage of its own code generator and integrated assembler and disassembler. This arrangement is called an **umbilical** or **tethered** system. In practice, the only disadvantage of these is that debugging and maintenance operations require exactly the same source code as generated the code in the target. This is as it is during development, but may cause problems for maintenance. Another disadvantage is that the Forth interpreter/compiler cannot be used for production test and configuration.

Using an Umbilical Forth, it is quite possible to run two or more tasks on an 8051 single chip device using only the 256 bytes of internal RAM.

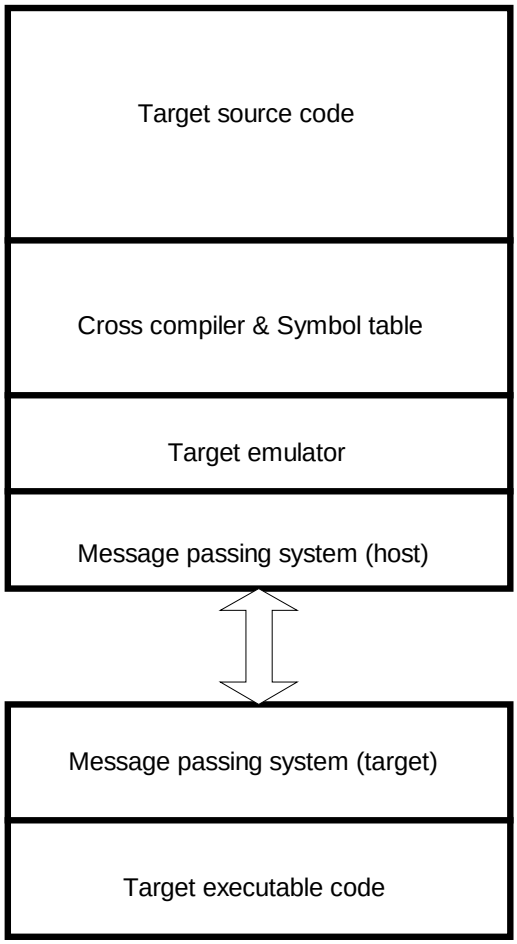

Figure 6: Umbilical Forth model

Probably the smallest CPU we have used this technique on was a 4-bit CPU from the Hitachi HMCS400 range.

The link between the host and target is often called the XTL for cross target link. It is normally a serial line. Some cross compilers can be extended by the user with new drivers. We have used other links such as I2C, SPI, BDM and a JTAG interface.

16 Forth Internals

Anatomy of a Forth system

A Forth system consists of the following:

- Dictionary in which to store the definitions of words
- Parameter stack to pass information to and from words
- Return stack to hold return addresses
- Input and output buffers
- User variables

A minimum system will occupy around 8K bytes (4K cells). The maximum addressable memory size on a 16 bit Forth system is 64kb and on a 32 bit Forth is 4Gb. A typical memory map on a conventional CPU is shown below:

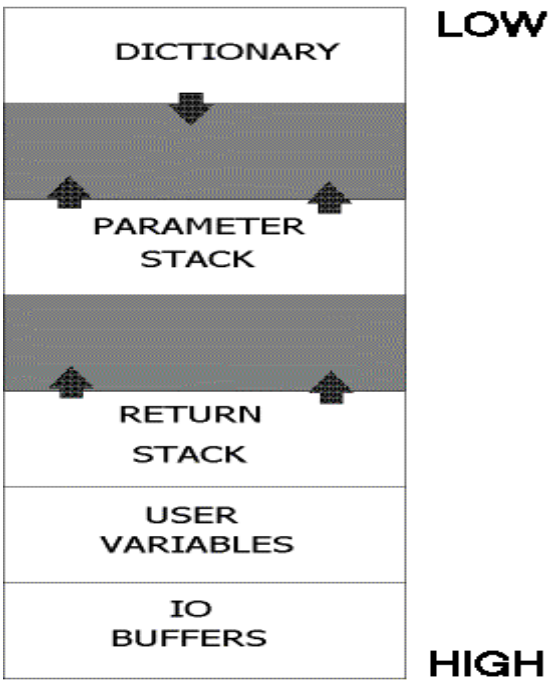

Figure 7: Typical Forth memory model

The picture below indicates a conventional layout of a Forth word.

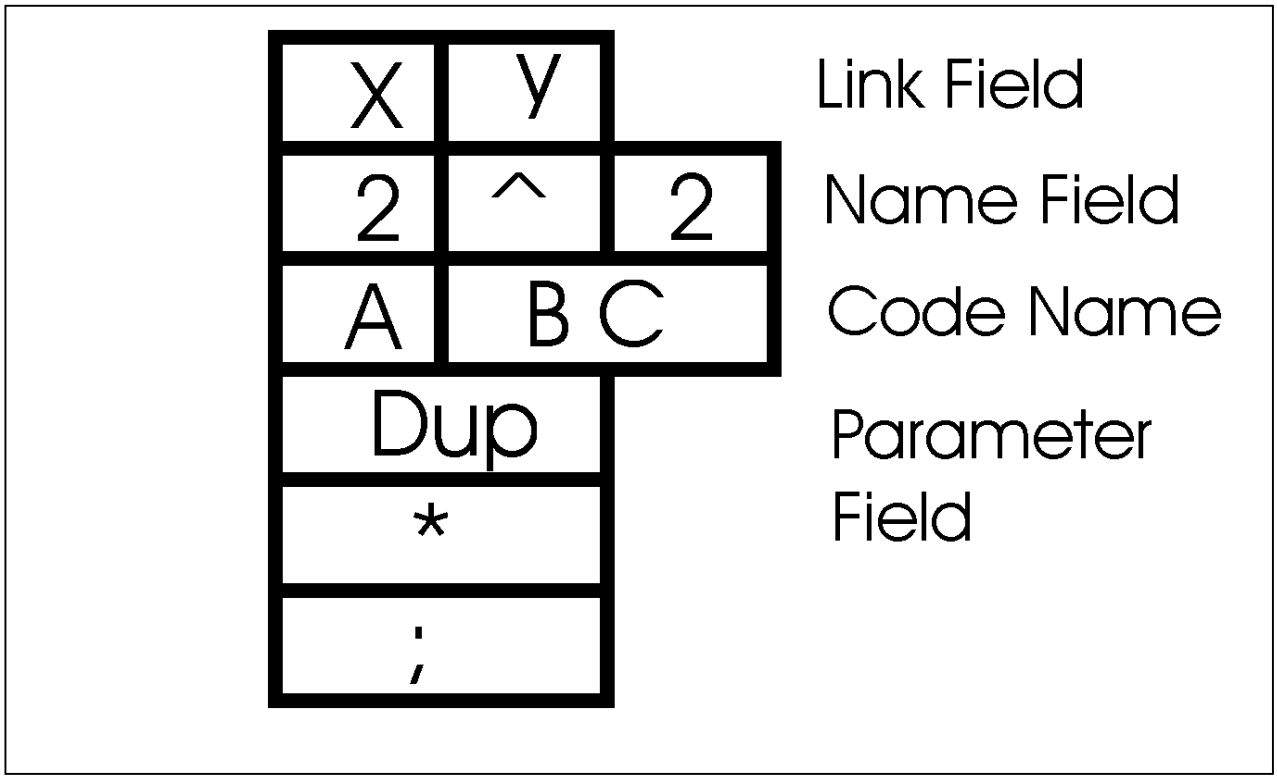

Figure 8: Dictionary entry

The word described is:

```
: 2^2           \ n -- n*n
  dup *
;
```

There is great variety in dictionary implementation schemes. In general, the link and name fields (the dictionary header) are together, and the code name and parameter fields (the code body) follow. However, some systems keep the headers in a separate database, and in systems such as Umbilical Forths (see the embedded systems chapter) the header and the code body may even be on separate CPUs.

Link field: X and Y in the figure above. Forth word names are chained in a linked list structure, anchored in a wordlist. The linking schemes range from single linked lists on embedded systems to fully hashed schemes under operating systems.

Name field: Includes the text of the word's name, and some other information controlling visibility of the word.

Code name field: A, B and C in the figure above. The field contains compiler-specific information generated by the compiler. In threaded code systems, this will contain a pointer or code. In modern systems, the field may not present at all for colon definitions, but will be present for children of **CREATE**.

Parameter field: Words built by **CREATE** contain data. The parameter field address, sometimes known as the PFA, is the start address of this data area. Colon definitions do not have parameter fields and this region usually just contains compiled code.

Navigating the dictionary structure

Because of the variety of implementation techniques, and because the ANS Forth standard carefully avoids implying any underlying implementation techniques, dictionary navigation is not standardised. However, many systems provide a range of words for stepping from one field to another for use within their own tools. You will have to read the manual for the Forth you use to find out what is available. Commonly available words are shown here. They mostly assume that the xt of the word is the starting point since this is the address returned by **FIND** and **SEARCH-WORDLIST**.

>BODY `xt -- pfa`

Step from the xt to the PFA. In ANS systems this operation is only specified for children of **CREATE**.

BODY> `pfa -- xt`

Step from the PFA to the xt.

>NAME `xt -- name`

Step from the xt to the name field.

NAME> `name -- xt`

Step from the name field to the xt.

Structure of compiled code

This section deals with compiling code on conventional CPUs. Dedicated two-stack machines are covered in the next section. The code examples are for colon definitions on the Intel IA32 instruction set used from the 80386 upwards. This was chosen for its wide familiarity, not for any endorsement of the architecture. The choice of compiled code structure depends on the required trade off between code size, performance and complexity. The implementations are described in the order of fastest first.

Forth reg.	Function
IP	Instruction/Interpretation Pointer
RSP	Return Stack Pointer
PSP	Data(Parameter) Stack Pointer

Native Code Compilation (NCC)

This is the term usually given to systems that generate optimised machine code. The code quality of the best of these is as good as that produced by compilers for other languages such as C/C++. NCC systems have much in common with the STC systems below. Tests show that code size need be no greater than for DTC systems below, although the complexity of the code generator is very much higher. NCC is currently (2005) the implementation of choice for most 16 and 32 bit CPUs.

Forth reg.	CPU reg.
IP	EIP (PC)

RSP	ESP
PSP	EBP

Calling a word:

```
CALL address
```

Entry/exit code:

```
… RET
```

Subroutine Threaded Code (STC)

These systems use the CPU call and return system. They are typically over twice as fast as the DTC systems below. Some versions generate short code sequences for simple words to avoid call/return overhead, but do not do enough to be classified as NCC systems.

Forth reg.	CPU reg.
IP	PC
RSP	ESP
PSP	EBP

Calling a word:

```
CALL address
```

Entry/exit code:

```
RET
```

Note that on RISC CPUs such as ARM and MIPS, the return address may be placed by the BL instruction in a register and then a further save of the return address may be required in the entry code.

Direct Threaded Code (DTC)

A short machine code sequence is followed by data or a list of word addresses for colon definitions. The word addresses point to the machine code sequence for the word. Coded words end by jumping to or inlining the **next** sequence below.

Forth reg.	CPU reg.
IP	ESI
RSP	EBP
PSP	ESP

Calling a word:

```
address
```

Entry/exit code:

```
JMP docolon … exit
```

```
docolon:
  SUB   EBP, 4
  MOV   0 [EBP], ESI
next:
  MOV   EAX, 0 [ESI]
  ADD   ESI, 4
  JMP   EAX

exit:
  MOV   ESI, 0 [EBP]
  ADD   EBP, 4
  JMP   next
```

Indirect Threaded Code (ITC)

The start of a colon definition contains the address of the docolon code routine. It is followed by a list of addresses finishing with the address of the exit routine, which is coded.

Forth reg.	CPU reg.
IP	ESI
W	EBX – Work register used by some system code
RSP	EBP
PSP	ESP

Calling a word:

```
address
```

Entry/exit code:

```
docolon … exit

docolon:
  DD    cocoloncode
docoloncode
  SUB   EBP, 4
  MOV   0 [EBP], ESI
next:
  MOV   EBX, 0 [ESI]
  ADD   ESI, 4
  MOV   EAX, 0 [EBX]
  JMP   EAX

exit:
  MOV   ESI, 0 [EBP]
  ADD   EBP, 4
  JMP   next
```

Token Threaded Code (TTC)

Token threaded code is used when code density is the overriding requirement. TTC systems use token numbers to represent Forth words by small integers. Often the most common words are coded as 8 bit tokens, and 16 bit tokens are used for other words. Token Threaded Code has been used on payment terminals to provide binary

code portability of banking applications across terminals from many different suppliers. It has also been used to reduce the code size of games in a mobile phone.

Because of the wide variety of implementation techniques on the same CPU (some TTC systems are written in C), no sample implementation is given. TTC is far and away the slowest technique shown, but for applications which spend the majority of their time in core words or operating system calls, application performance may be considerably better than expected.

Other forms

Other forms that have been seen include segment threaded code for 80x86 real-mode applications. The ingenuity of implementers in exploring the speed/size trade-off is impressive.

The Open Firmware system used by Sun Microsystems, IBM and Apple to initialise hardware and boot the operating system uses tokenised Forth **source** code which is compiled at power-up.

Forth engines and stack machines

Commercially, microprocessors designed for a specific programming language have not been successful. CPUs designed for efficient execution of Pascal, LISP, C, Forth and Occam have all been produced as single chip microprocessors. Ultimately, none have been a success in the long term. The exception in this list may be the Transputer, which was originally designed to support both Occam and parallel processing. Once an efficient C compiler for the Transputer was available, it became a mild commercial success.

Stack machines have returned to current compiler and silicon design through execution of the Java Virtual Machine (JVM). Add to this the realisation that the fashions in current language selection are shorter than the commercial lifetimes of CPU architectures. The current focus in stack machine design includes careful design to support efficient execution of languages other than Forth.

Although hardware support for preserving return addresses on a stack has been common since the 1960s, support for two or more stacks has been less common. When using a data stack transient data is ordered on the stack rather than in a register file, so one feature that distinguishes CPUs designed to support two stacks is a very limited number of registers that have to be saved when performing a context switch. Thus interrupt response is very good. Such CPUs are usually designed to be deterministic, i.e. the time taken for execution of an instruction sequence is completely predictable and stable. The RTX2000 running at 10 MHz instruction rate has a 400ns interrupt response time before useful work can be performed. One of our clients used it for video data acquisition using a 1 MHz interrupt rate. Another used it as a timing generator for medical imaging systems.

It is interesting to note that despite the enourmous improvement in straight-line performance of CPUs over the last 10-15 years, there has been very little if any improvement in interrupt response times. This is discussed in more detail in a following section.

Deterministic execution rules out caching and pipelining beyond the fetch/execute level. Because of the low number of internal registers and the use of zero operand instructions reducing instruction decode complexity, stack machines tend to be very simple (low gate count, silicon area) by comparison with current RISC and CISC architectures. In turn this makes them fast for the given process technology. Their

application domain has been mostly in the area of hard real-time ("late answers are wrong answers").

First published in 1989, "Stack Computers: the new wave" by Philip J. Koopman, Jr. is still the definitive work for an introduction to stack machines. The book may be downloaded from http://www.ece.cmu.edu/~koopman/stack_computers/index.html

Commercial devices

Commercially available chips based on a Forth execution model included the Novix NC4000, the Harris/Intersil RTX2000/01/10 series, the IX1 from Delta-t and the Silicon Composers SC32.

http://www.inscenes.com/siliconcomposers.shtml?siliconcomposers

Several others have been prototyped without achieving commodity silicon status. Of the named CPUs above, the RTX2000 family became a workhorse for space and satellite applications as it was available in a radiation-hardened process. It survives as the RTXcore VHDL implementation from MicroProcessor Engineering. Both Forth and C compilers are available. The IX1 sold in considerable volume into a niche market as a fieldbus processor, capable of supporting most serial protocols at up to 1 Mbps.

A group of devices including the Patriot Semiconductor PTSC1000 and the Lightfoot family have been produced. These are essentially two-stack machines enhanced for C and direct support of Java bytecode execution. At present their level of commercial success is unknown.

With the explosion in the capability of FPGAs (Field Programmable Gate Arrays), the current crop of available stack machines are mostly focussed on implementation in FPGAs from VHDL or Verilog sources. Apart from the RTXcore from MPE, an interesting machine is the MicroCore (see http://www.microcore.org) which has a particularly interesting exception handling mechanism in hardware. With the enourmous costs of one turn of a System-on-Chip design to silicon, the FPGA approach to CPU design is the most likely to produce commercial success.

A Google search for stack machines will reveal the current state of the art.

Prototype and research machines

Chuck Moore and his associates have produced a range of devices which have achieved limited production. His approach has been to produce an integrated tool chain from silicon modelling to tiling. The results show that this approach can a give a performance gain of over ten times compared to layout through an industry standard tool chain. The downside of this approach has been that each silicon process change requires considerable work on the tool chain, so reducing the rate at which new processes can be adopted. While (Gordon) Moore's law holds, it is unlikely that this approach will lead to commodity silicon, but it can produce stunning performance at very low cost for high-volume niche products.

Several EuroForth papers, especially from Bernd Paysan, describe the use of small stack machines inside embedded products. In these designs the stack machine is tuned in terms of instruction set, performance and silicon area for each application.

Chris Bailey (now at York University, UK) has explored the possibilities of superscalar execution in stack machines with some success. His research suggests that stack machines are suitable for superscalar execution and shows how it can be

done. In the long term this opens the door to the use of stack machines in high performance environments outside the traditional hard real-time domains.

Notes on embedded real-time

These notes are taken from a paper by Chris Bailey and Stephen Pelc at the EuroForth 2004 conference.

There are four main areas in which embedded systems differ from desktop computing.

1) Importance of interrupt response time

2) Importance of deterministic response time

3) Economics of code size

4) Importance of branch behaviour

In the case of stack machines, all comparisons are against an RTX2000 CPU at 10MHz (circa 1988). The state of the art for an embedded system is taken from a 60MHz Philips LPC2106 ARM-based microcontroller without caches but with a memory accelerator (2003/4).

Interrupt response time

We define this as the time taken for the CPU hardware and software to save context before starting useful work. To this must be added the context restore time after performing useful work. The code below shows the figures for a 10MHz RTX against a 60MHz ARM7TDMIS.

```
Entry
RTX2000          400ns  0 bytes       0 instructions
ARM              432ns  20 bytes      5 ins, 27 cycles @
16ns

Exit
RTX20000  200ns  2 bytes       1 instruction
ARM              400ns  16 bytes      4 ins, 25 cycles
```

Note that these figures exclude any overheads for compiler-generated code.

The figures show that despite a 6:1 to 40:1 clock rate increase in 15 years, where typical top-end embedded CPUs run at 200-800MHz, interrupt response times have not improved. The ARM is among the better performers in embedded RISC CPUs. As shown by Koopman and others, interrupt response (entry) times in current CISC CPUs can exceed 400 clock cycles. When caches and MMUs are involved, the situation becomes even worse.

```
PROC IRQ_entry \ --                \ 4 cycles
\ --- save state ---
  stmfd rsp ! { r0-r12, link }     \ 3 + 14 mems
  mrs   r3, SPSR                   \ 1
  stmfd rsp ! { r3 }               \ 2 + 1 mem
\ --- call high level handler ---
l: IRQ_call
  bl    <action>                   \ 3
\ --- restore state ---
```

```
  ldmfd rsp ! { r3 }              \ 3 + 1 mem
  msr   SPSR, _c _f r3            \ 1
  ldmfd   rsp ! { r0-r12, link }  \ 3 + 14 mems
  sub .s  pc, link, # 4           \ 3
end-code
```

Determinism

Many embedded applications sample regular signals (heartbeat, 50/60Hz mains, audio etc.). It is imperative that sampling periods are at fixed intervals to reduce phase jitter in the sampling. Modern CPUs achieve high clock speed using caches and long pipelines. Both of these have adverse impact on determinism. See Koopman et al for the numbers.

It should be noted that a heavy interrupt load affects both entry and exit performance. In the worst case, the whole of the interrupt routine must be loaded from main (slowest) memory and displaces the background task which in turn must be reloaded from main (slowest) memory.

Economics of code size

Silicon costs go up as the fourth power of dimension (yield, chips/wafer etc) and power consumption goes up with the number of active transistors per clock within the same geometry. In the single chip microcontroller area, code size affects memory in terms of on/off chip Flash, and also in terms of RAM usage. Many die photos demonstrate that the silicon area of on-chip memory exceeds that of the CPU and peripherals.

Importance of branch behaviour

Typical microcontroller code shows that branches and flow of control occur roughly every 5 instructions (approximately 20% of the code). A Pentium 4 with a 20 stage pipeline, and misses in all caches and branch prediction buffers can suffer a 30 cycle penalty for a missed branch (20 cycles in the pipeline, 10 in the memory). The ARM above has a 4 cycle worst case (1 for decode, 3 for memory), whereas the RTX has a fixed 2 cycle overhead in all cases.

The Pentium 4 figures given above are probably best case. The actual timings depend heavily on the chip-set and memory system. One correspondent reports that he has measured a delay of 385 cycles on a Pentium 4.

17 Using the Forth interpreter

Forth is an interactive extensible language. You can use the interpreter at compilation time and during program execution as well as for program testing. You can also use the compiler at run time. Careful use of the interpreter, compiler and Forth system can give big dividends in terms of code size and timescales.

The first example shows how configuration code can be used to set defaults at compile time, and to provide configuration tools for the application.

The second example uses the compiler to provide storage of named items, and explores a number of issues raised by this.

Configuration example

Application and design

The application is from a serial device driver used in an embedded PC running DOS and VFX Forth for DOS. Not all the code is provided here, so you'll have to use this code as an illustration from a real-world example – it is not complete in itself. The point is to illustrate how the Forth interpreter can be used both at compile time and at run time.

To configure a serial port, not only do you need to specify the baud rate, parity and so on at run time, but you also need to be able to specify the I/O port address, interrupt number and so on. The hardware configuration for the first three serial ports is (mostly) standard and can be done at compile time. Industrial applications can require a very large number of serial ports and since their configurations may change (it is difficult to buy the same PC motherboard for more than two or three years in succession), the configuration may have to be done during installation and commissioning of the equipment. This configuration information can be held in a file which is **INCLUDE**d at program start. It uses the Forth interpreter to set up the system.

The Forth system accesses I/O devices through a "Generic I/O structure". This is a data structure whose first part is common to all devices, and the following parts are data private to each device. For each serial device, a configuration string is used to the data into the data structure. The strings are similar to those used by the DOS MODE command, e.g.

```
COM2: 38400 baud  N,8,1  8 kb 4 kb buffers
```

specifies that the device uses the hardware for COM2, runs at 38400 baud, no parity, eight data bits, one stop bit and has input and output buffers of 8 and 4 kilobytes respectively.

Interpretation of the string starts with the address of a structure on the data stack. Each word that processes part of the configuration, such as **BAUD** and **N,8,1**, consumes relevant data and leaves the structure address on the stack ready for the next word.

Implementation

The code is taken directly from one of the VFX Forth for DOS source files. The comment lines that start '\ *x' are processed by documentation tools to produce HTML and PDF documentation directly from the source code.

```
struct /ComData      \ -- n ; size of uart/queue structure
\ *G The /ComData structure holds the data needed to process
\ ** a channel, including the UART address, queue addresses,
\ ** and transmit flags. This is a Generic I/O structure.
\ GENIO data
  int C.handle       \ device handle or -1
  int C.vectors      \ pointer to Gen I/O vectors
\ UART configuation
  int C.mode         \ mode flags, b0=16450, b1=16550,
  int C.baud         \ baud rate
  byte C.int#        \ DOS interrupt#
  byte C.#data       \ #data bits, 5, 7, 8
  byte C.parity      \ parity
  byte C.#stop       \ stop bits
\ buffer configuation
  int C.port         \ UART port base address
  int C./inpQ        \ size of input queue
  int C.inpQ         \ input queue (allocated from heap)
  int C./outQ        \ size of output queue
  int C.outQ         \ output queue
  int C.flag         \ true if TX in progress
\ PIC configuation
  int C.pic0         \ first PIC port (must be set)
  int C.pic1         \ second PIC port (0=unused)
  byte C.irqPic0     \ PIC0 IRQ#
  byte C.enPic0      \ PIC0 channel enable mask
  byte C.disPic0     \ PIC0 channel disable mask
  byte C.irqPic1     \ PIC1 IRQ#
  byte C.enPic1      \ PIC1 channel enable mask
  byte C.disPic1     \ PIC1 channel disable mask
  aligned
\ Interrupt counters and state
  int C.#ints        \ number of times interrupts checked
  int C.#MSRints     \ number of MSR interrupts
  int C.#RXints      \ number of receive interrupts
  int C.#TXints      \ number of transmit interrupts
  int C.#LSRints     \ number of LSR interrupts
  byte C.lastMSR     \ last value from MSR
  byte C.lastLSR     \ last value from LSR
  aligned
\ Saved UART configuation
  /UARTstate field C.US    \ UART state save area (all bytes)
  aligned
end-struct


\ ************************
\ *S Configuring serial ports
\ ************************
\ *P Serial ports are initalised and opened using command
\ ** strings similar to those used in DOS commands. The
\ ** commands are Forth words in a private vocabulary.

vocabulary COMsettings     \ --
\ *G Used for serial port set up commands. All words in this
\ ** vocabulary must return the /Comdata structure passed
\ ** to them, e.g.
\ *C   115200 baud   ( struct n -- struct )
```

```
\ ******************************
\ *N Words used in command strings
\ ******************************
\ *P The /ComData structures are initialised using command
\ ** strings held in a private vocabulary. The same strings
\ ** are used when opening the device.
\ *C   COM2: 38400 baud  N,8,1  8 kb 4 kb buffers

also COMsettings definitions

: baud         \ struct baudrate -- struct
\ *G Set the structure's baudrate.
\ ** The default is 9600.
  over C.baud !  ;

: port         \ struct port -- struct
\ *G Set the structure's I/O port base address.
\ ** No default unless SET by COM1..COM4 below.
  over C.port !  ;

: int#         \ struct int# -- struct
\ *G Set the structure's DOS interrupt number.
\ ** No default unless SET by COM1..COM4 below.
  over C.int# c!  ;

\ LCR - Line control register

: #data        \ struct #bits -- struct
\ *G Set the structure's number of data bits. Defaults to 5.
  5 - %00000011 and              \ LCR bits 1:0
  over C.#data c!  ;

: #stop        \ struct #stop -- struct
\ *G Set the structure's number of stop bits. Must be 1 or 2.
\ ** The default is 1.
  1- 0<> $04 and  over C.#stop c!  ;   \ LCR bit 2

$00 constant no                        \ LCR bits 5:3
$08 constant odd
$18 constant even
$28 constant par1
$38 constant par0

: parity       \ struct #bits -- struct
\ *G Set the structure's parity usage. #bits must be one of
\ ** NO, ODD, EVEN, PAR1 or PAR0. The default is NO.
  over C.parity c!
;

: buffers      \ struct #in #out -- struct
\ *G Set the input and output queue buffer sizes, which
\ ** must be a power of 2. If unused, defaults to 1024
\ ** and 256.
  rot tuck C./outQ !  tuck C./inpQ !
;

: PIC0         \ struct bit# port -- struct
```

```
\ *G Set the bit# and base address of the PIC that will be
\ ** used first to clear/enable/disable an interrupt.
\ ** Defaults to the COM1 standard settings or as set
\ ** by COM1..4.
  ...
;

: PIC1         \ struct bit# port -- struct
\ *G Set the bit# and base address of the PIC that will be
\ ** used second to clear/enable/disable an interrupt.
\ ** Use 0 0 PIC1 if the second PIC is unused.
\ ** Defaults to the COM1 standard settings or as set
\ ** by COM1..4.
  ...
;

: COM1:        \ struct -- struct
\ *G Set the default conditions for COM1:
  $03F8 port  $0C int#  4 $20 PIC0  0 0 PIC1
;

: COM2:        \ struct -- struct
\ *G Set the default conditions for COM2:
  $02F8 port  $0B int#  3 $20 PIC0  0 0 PIC1
;

: COM3:        \ struct -- struct
\ *G Set the default conditions for COM3:
  $03E8 port  $0C int#  4 $20 PIC0  0 0 PIC1
;

: COM4:        \ struct -- struct
\ *G Set the default conditions for COM4:
  $02E8 port  $0B int#  3 $20 PIC0  0 0 PIC1
;

: N,8,1        \ struct -- struct
\ *G Set the structure to no parity, 8 data bits, 1 stop bit.
  no parity  8 #data  1 #stop
;

previous definitions


\ ***************************
\ *N Handling command strings
\ ***************************

also COMsettings
: ([COM) \ struct -- struct
  -1 over C.handle !             \ mark as closed
  COM1:  #9600 baud  N,8,1       \ default initialisation
  /ComRxQ /ComTxQ buffers
;
previous

: [COM         \ struct -- struct
\ *G Starts a definition of a /ComData structure.
\ ** [COM may be followed by any of the words above up to
```

```
\ ** the closing COM]. Unless overriden, sets
\ ** COM1:9600,N,8,1. Use as indicated
\ ** in the example below:
\ *C   struct [COM  COM2: 38400 baud  N,8,1  COM]
  state @
  if  postpone ([COM)  else  ([COM)  endif
  also COMsettings
; immediate

: COM]         \ struct --
\ *G Closes the definition started by *\fo{[COM} above.
  state @
  if  postpone drop  else  drop  endif
  previous
; immediate

SysErrDef err-COMstring "Error in COM command/open string"

: (SetComData) \ caddr len struct --
  depth >r
  [COM  -rot evaluate  COM]
  depth r> swap - 3 <> err-COMstring ?throw
;

: SetComData   \ caddr len struct --
\ *G The string caddr/len is processed by EVALUATE as
\ ** between [COM string COM]. Any error causes a THROW.
\ ** Zero length strings are ignored so that previously
\ ** set data is used.
  over if
    ['] (SetComData) catch err-COMstring ?throw
  else
    drop 2drop
  endif
;
```

A serial port device is created by the word **`SerDev:`** which can then be set up using the same command strings.

```
SerDev: COM1dev      \ -- sid
\ *G Standard PC COM1 port device set by default to:
\ *C  COM1: #115200 baud
  COM1dev [COM  COM1: #115200 baud  COM]

SerDev: COM2dev      \ -- sid
\ *G Standard PC COM2 port device set by default to:
\ *C  COM2: #38400 baud
  COM2dev [COM  COM2:  #38400 baud  COM]
```

Phone book revisited

The phone book example comes from an MPE course that has been run for many years. While giving the course at a customer site, I realised that the implementation shown in a previous chapter takes no advantage of Forth itself. I then rewrote the exercise to take full advantage of Forth itself.

The original implementation in the course file requires about 270 lines of source code. The implementation presented here takes about 80 lines but takes some liberties with the intent of the specification. It uses Forth's dictionary structure and extensibility as part of the application.

Design

An entry in the phone book is a combination of a string and an extension number. From the specification, the string is a single name. The association of the two is performed by defining a constant, e.g.

```
10 constant benedict
```

To avoid conflicts with words of the same name in the underlying Forth system, we keep the names in a separate wordlist (nameless vocabulary) as **CONSTANT**s. By creating and searching for names in this wordlist we can use the Forth interpreter itself to create the entries and the standard dictionary search tools to find the names. Once we have found the name we can **EXECUTE** the constant to return its value.

Because ANS Forth does not specify either the implementation structure of the dictionary or the tools required to walk a wordlist, I have chosen to make an assumption (see the words **CALLS** and **CALLS?** below) about the dictionary and data layout of the underlying Forth system. This technique will not work on Umbilical systems. If you find other Forth systems for which the assumption fails, please let me know. The assumption is required to find a name from a number. I assume that the data value defined for a **CONSTANT** is at the very end of the word's data structure and that if I "comma" more data into the dictionary the new data follows on immediately. I also assume the existence of the word **BODY> (pfa -- xt)** which exists on most hosted systems.

Implementation

```
only forth definitions  decimal

wordlist constant widPhone \ -- wid
\ *G Make the wordlist for the PhoneBook.

0 value LastEntry    \ -- addr
\ *G Holds linked list of entries.

: FindName     \ "<name>" -- xt nz | 0
\ *G Return the xt and non-zero of the name that follows.
\ ** If the name cannot be found just return zero.
  bl word count widPhone search-wordlist
;

: CheckNumber  \ n --
\ *G Error if n is outside the range 0..9999
  0 9999 within? 0=
  abort" Telephone number outside range 0..9999"
;

: CheckName    \ caddr len --
\ *G Apply checks to a name
  dup 15 > abort" Name too long"
  2drop
;
```

```
: Calls        \ n "<name>" --
\ *G Make a new entry in the phone book in the form:
\ *C  <nnn> Calls <name>
  dup CheckNumber
  >in @  bl word count CheckName  >in ! \ abort if exists
  get-current >r  widPhone set-current  \ private wordlist
  constant                      \ make entry
  here LastEntry ,  to LastEntry \ link in chain
  r> set-current                \ restore
;

: Calls?       \ n --
\ *G Report who is called by number n.
  LastEntry
  begin                    \ -- n addr
    dup
   while                   \ -- n addr
    2dup cell - @ = if     \ check value of constant
      body> >name .name
      drop  exit
    endif
    @
  repeat
  2drop ." Nobody"
;

: Phone        \ "<name>" --
\ *G Display the number of the name that follows.
  FindName if              \ -- xt |
    execute cr .           \ execute to return value
  else
    cr ." has no phone"
  then
;

: Entries?     \ --
\ *G Display the phonebook names and numbers.
  LastEntry
  begin
    dup                    \ address of link
   while
    cr  dup body> >name .name
    17 >pos  ." -- " dup cell - @ .
    @
  repeat
  drop
;
```

Deviations, issues and lessons

This implementation deviates from the original specification in a number of ways:

- the number of entries is not defined,
- the display format is slightly different,
- changed extension numbers are not removed.

In effect, the implementation is a quick and dirty hack to get something running. However, it raises several issues about the specification:

- Why are names restricted in size?
- How should the phone book be saved?
- Can two people have the same name?
- Do people share extensions?

In my career it has been very rare to receive a complete or good specification. In practice specifications evolve by refinement – the "spiral lifecycle" rather than the "waterfall" model. However, an implementation architecture that makes a false assumption about the long-term evolution of the system will eventually fail and have to be discarded. Fixing architectural faults is often more expensive than starting again.

When you are faced with a new specification, you have to question it. There are often assumptions made by the authors about software that are invalid. Similarly, as a software engineer, you are likely to make assumptions about the application that are also invalid. The fine (and often critical) details about the process being automated are all too often unstated in the specification.

Another crucial part of the initial stages is to simplify as much as possible. Separate the main objectives from the "wouldn't it be nice ifs" (WIBNIs). In the later stages of a project WIBNIs often introduce complexity, consume more time than the original objectives, and delay product launch. Although the marketing department and salesmen are key friends early on, their enthusiasm can generate unexpected problems later on. In most projects, and especially embedded ones, the software people are the last in the chain, and thus are most visible for late delivery, even if late delivery of out-of-specification mechanical systems is the cause of your grief. On one project I worked on, the mechanics were taken away for overnight painting and were delivered back four days later, leaving less than eight hours for software testing before the system was delivered for trials.

Analysis of large software projects shows that frequent deliveries of software modules that can be tested contributes enourmously to project success rate. If the architecture is right, additional features can be added later. Giving the end-user something to try contributes to the sense of progress and reveals any design problems early. Fixing something costs as at least three times as much as getting it right first time.

One of the key benefits of Forth is that the interactivity allows for very rapid investigation of faults. You can use the interactivity not only for debugging but also to demonstrate possibilities to people in other disciplines. In order to do this successfully you have to have an initial chassis to work from. So deliver something early and use it to test not only software, but also the specifications and assumptions you are working from – not only other peoples' assumptions, but also your own.

18 Code Layout

Code layout is a vexed topic in any programming language. Many programmers, especially “guru” class programmers, have a style of their own and are reluctant to change it. However, from my own observation of large projects, the use of a “house style” for source code produces significant benefits over the lifetime of a product. The larger the team, the more important this becomes.

This does not mean to say that a house coding style should not evolve. My own programming style has changed as I have learnt, and in some cases it changes it according to the application domain and who is going to read the code.

The key features for a source coding standard are

- Consistency from one programmer
- Consistency between many programmers
- The standard is easy to follow and understand
- The resulting code is easy to follow and understand
- Code which is difficult to get wrong because of layout
- A layout which is also visually pleasant

A secondary (and increasingly important) benefit is that a consistent style makes it much easier to use automated source maintenance and documentation tools. This may not appear worth much for a quick and dirty project, but it is invaluable for large projects. There are current Forth projects approaching 1,000,000 lines of Forth source code.

A description of the requirement of the standard will be followed by an example. In the example, the relevant code fragments will be printed in UPPER CASE. The comments in the example will be in lower case. However, the case of the characters used in real code is not specified. Some compilers may be case-sensitive, others not. Also different programmers may have case preferences. The code examples in this document will assume that the left-hand end of the line comes directly below the left-hand end of this type of text. Indented code will be relative to this column position:

```
\ the beginning of the line
 \ indented by one space
```

Why a standard?

There are several reasons for producing and following a standard for anything. This standard is produced for the following reasons.

In any organisation, if many programmers are all using the same language, they will inevitably be sharing source code, or working in teams on a given project. If every programmer writes code to his own or no standard, then communication will be difficult, and programmers will tend to prefer to re-invent the wheel rather than try to understand and use another person’s code. This is both costly and time-consuming. By following a standard - any standard - each programmer’s code will be meaningful to the others.

When a programmer starts to learn a language, he or she will not know how best to either write or lay out the code being written. Tutorials abound to teach the syntax, structure and word-set of Forth, but there is little advice given on layout and practice. This document seeks to offer the collected experience of staff at MPE so that other programmers learn an accepted and clear standard.

However, the standard as presented here should not be taken as a diktat. If an organisation has its own specific layout or documentation requirements, these should not be ignored. A standard is there to help, not hinder in the production of source code.

Other Forth coding standards exist. A good place to start looking for them is at http://www.forth.org.

Implications of editors

Programmers vary widely in their attitudes to editors. Some can (and will) use almost any editor to hand. Others are lost without an (inevitably obscure and obsolete) editor and a set of idiosyncratic macros. Over the years I have come to the conclusion that trying to impose a particular editor will only spark a long and expensive religious war. Editors are cheap compared to programmer salaries and PC costs.

The use of an editor has many implications. One of these is the amount of code visible on the monitor at any one time. Another is the use of tabs and spaces in white space. Another is syntax colouring, which has a real impact; I am always surprised when I read my code in uncoloured form. Yet another is the decision of how the code is fragmented - how many words in a section or in a file - and how many files in the application or project.

The number of lines visible on the screen, coupled with the speed of cursor movement almost dictates the size of any word or procedure produced. This is of direct relevance to the style of code eventually written: whether the code is very vertical with lots of white space, or is very horizontal with much code on every line. This is discussed later.

The decision of how many words will be placed on one page or in one file depends on the nature of the compiler and editor in use. If the editor can have many windows onto many files, then many files may be used easily - with the code factored out on a per-file basis. However, if the editor does not support multiple files, then there will be a tendency to place all code in one file.

Tabs

The use of tabs to space out code and comments has a direct relevance to the amount of white space between elements of the source file. If spaces must be used, there will be little white space (as its production is tedious), but if tabs are used, there may be quite a lot of space in the file, and it will be uniform, but will be at the whim of any tool used with the file. A good example of this is the fact that DOS (and many Windows command-line tools) tabs are preset to 8 columns (1,9,17, etc.) for programs such as PRINT and TYPE, but editors generally have programmable tab-stops.

It is therefore important to consider the tab spacing used in conjunction with the tools likely to be used with the source files. If an editor is capable of smart indenting, the amount of indent has to be considered. If the indent is set too deep, very few structures will be easily nestable (sometimes an advantage), but if the indent is too small, then the indent will not stand out as such.

Our house rule is that if you edit with tabs set to anything other than every eight characters, the editor must be set to insert spaces rather than hard coded tabs. This lets us use some old but powerful command line tools while retaining flexibility in the editor.

When you receive third party code, you will often find that it has hard tabs set to some value other than the one you use. A quick Google search will find TAB.EXE or a similar tool that will substitute spaces for tabs. Some will even replace spaces by the appropriate tab settings.

Horizontal and Vertical layouts

There are two main styles of source-code layout in use. One is vertical, as used in assembler source, and the other is horizontal, as used by most C programmers (and others).

Vertical code:

```
mov ax, bx
add bx, 3inc di
```

Horizontal code:

```
 for (i=0; i++, i<=20) printf (%d, i);
```

The horizontal layout leads to a high code density and minimal eye movement to read. The vertical layout encourages in-line comments and improved visibility due to white space. Both have their benefits and disadvantages.

Forth programmers usually prefer in-line comments along the definition of a word. This undoubtedly leads to more comprehensible code as it is both read and written, but relies on a rather vertical code layout. However, the novice programmer is likely to extend the vertical layout to the extreme of a typical assembler layout, and thus lose the high-level structure and flow of Forth source.

This generally derives from the phrasing of code - writing meaningful phrases or fragments of code on one line such that the comment describes the overall effect of the line, not the actions of individual words such as **@** and **+**. Phrasing also helps point out code fragments which could be written more efficiently as separate words - being factored out.

Making Forth code legible is a compromise between vertical and horizontal layouts - with code well phrased or factored, but with structures spread out for easy checking or modifying. As discussed in the section Control Structure Layout, however, even these are open to debate, as short bodies within structures are often best on the same line as the entire structure.

Comments

There have been two standards of Forth source code comments. One is the in-line comment, the other is an additional block of comment before or after in the file, or in another file.

The former has the advantage of being parallel to the code to which it relates whilst the latter reads as consistent English, or other human language, and is more descriptive. In a text file, both forms of comment may be supported. Because the page is at least 80 characters wide, each line may include a good in-line comment. Because the monitor screen is at least 25 lines deep, a definition may also have a header block of comment above or below it. This comment could potentially be in

another file, perhaps a documentation file, but is far more relevant and useable if it is in the same file, and in the same section as the code it explains.

Wisdom should be used in the wording in comments. The comment should not be so trivial that it is pointless (fetch the contents of the variable), but should not be so removed from the code that it conveys no information (reads data structure). It should indicate clearly, assisting the Forth itself (get the pointer value):

```
: EXAMPLE \ -- ; comments example
                \ pointless ...
  DATA 4 + @           \ get the contents of the variable
                \ no information ...
                \ read data structure
                \ useful ...
                \ get the second pointer value
 @ EXECUTE  ;
```

Naturally the use of the phrase **4 +** in this code is deprecated. The use of structures or named offsets is much to be preferred.

File layout

Our files tend to consist of several sections. The first section is a header section, which is followed by one or more sections of related code, followed finally by file test code.

Some program editors permit you to enter page breaks as section breaks and display them as horizontal lines. Some years ago this was popular but the practice has now almost died out. Our experience was that printed copies (still useful) of source code did not look good unless programmers rigidly maintained a maximum of (say) 66 lines per page. A few days of late night commissioning on site ruined it all. We replaced the pages with sections delimited by particular commenting conventions, e.g.

```
\ *******
\ Section
\ *******

\ ==========
\ SubSection
\ ==========

\ -------------
\ SubSubSection
\ -------------
```

Header Section

The first section of a file should include any copyright, author, or other specific information. This may also include project details or other information relevant to the use of the code. If there are specific hardware dependencies, these should be outlined on the first page - this page is the one usually first seen when the file is browsed or edited.

```
\ Code for Front panel control
\ Customer: xxxxxxxx
\ Project: yyyyyyyy
\ (c) MicroProcessor Engineering Ltd and xxxxxxxxx
```

```
\ 133 Hill Lane
\ Southampton  SO15 5AF
\ UK
\ Phone +44 (0)23 8063 1441
\ Note: this code requires the zzzzzzzz
\ interface card, and MPE 8031 Cross-compiler
\ and interrupt handling code.
```

Do not forget to use full international addresses and telephone numbers. You'll be amazed where your code gets to after a few years.

Many Forth compilers have extended comment operators which ignore parentheses. This allows header comments to be written without the use of a backslash comment on every line. MPE and others use the form:

```
(( can extend over many lines and include () chars ))
```

If you do not have these, you can use the ANS conditional compilation form for long comments:

```
0 [IF]  ...  [THEN]
```

The standard MPE header looks like this:

```
\ Filename – description

((
Copyright (c) 2005
MicroProcessor Engineering
133 Hill Lane
Southampton SO15 5AF
England

tel: +44 (0)23 8063 1441
fax: +44 (0)23 8033 9691
net: mpe@mpeltd.demon.co.uk
     tech-support@mpeltd.demon.co.uk
web: www.mpeltd.demon.co.uk

Short description of the function of the code in this
file.

To do
=====

Change history
==============
20050131 SFP002 Converted for Program Forth
20041205 RBM001 Removed potential insult
))

((
Put the design description here. Design information
is really important!
))
```

We provide a change history in the form of a date code that can be automatically sorted, then a change code of three initials for the author and a three digit change number, followed by a brief description of the change. Each changed line is marked with the change code. Changes can be quickly monitored by searching for the relevant change code.

Where the design information is non-obvious, put an overall design in separate comment block – it's easier to find that way. Be kind to your fellow programmers, they may not be as bright as you are and you do not want to be pestered by them long afterwards. More importantly, putting down design information here clarifies the mind and reduces the bug level in your code.

Code sections

The rest of the file, either the code after the project information, or the subsequent sections, will contain the code for the project, or the portion of the project.

It is normal to split the code for an entire system into several files. These are normally grouped into related areas: all the data structures in one file, all the serial i/o in another, and so on. If the project is small enough not to warrant many source files, then the code areas will be grouped on different sections. If the project is big, then some parts of the code will inevitably start small, and then evolve into a monster. Split these into easily identifiable sections.

```
\ *******
\ Section
\ *******

\ Design info

: foo           \ a b – c
  ...
;
```

Since we started using automated tools to extract documentation from the source code, we have noticed that we tend to stick to our house coding style much more closely. Our clients and contractors have noticed this and say that the ability to produce HTML and PDF documents quickly has impacted their coding style too. Because design information can be extracted quickly and us up to date, it is read more. Consequently there is more positive feedback that encourages programmers to produce this documentation. Early documentation reduces bugs.

Test Section

Although much ignored, test code can be invaluable when the file has been broken. When you have finished with initial test code, comment it out and/or move it to the end of the relevant section or the end of the file. Do try to document it to some level, otherwise your successors will throw your work away.

Base and numbers

If the number base for the code on a section is important, the base should be specified at the top of the section. Our habit is always to revert to **DECIMAL** afterwards.

Many recent Forth systems permit you to use a prefix character to indicate the number base. The most common are:

$55AA hexadecimal
#1000 decimal
%1010 binary

If the compiler to be used supports this feature, then it is good practice to use it, as there can then be no mistake which number is meant at any time. If the compiler does

not support the temporary base definition, then it is best to always prefix a hex number with a zero:

```
HEX
0100      \ hex 100 = decimal 256
0ADD      \ hex ADD = decimal 2781
ADD       \ the word ADD
DECIMAL
```

Vocabularies and wordlists

If a vocabulary or context switch is to be made in the source code, the vocabulary should be the same at the end of a section as at the beginning. This means that if a new section is inserted afterwards, the search order and defining vocabulary will be known:

```
\ ************
\ I/O handling
\ ************

ALSO IO
.........
PREVIOUS
```

Layout of a definition

It is acknowledged that a Forth definition should be as short as possible. This may be 2 or 3 lines, or it may be 15 or 20 lines. The actual size will depend on circumstances, but should always be as short as possible. Forth minimalists say that good Forth definitions contain no more than seven source tokens, which is very difficult to achieve in practice, especially when coding for operating systems such as Windows. This recommendation comes for several reasons.

- Short code fragments are easy to test and hence are reliable
- Short code fragments promote code reuse
- Short code is understandable

If you reuse code wherever possible, it can then be possible to make major architectural changes to a software subsystem without introducing bugs all over the rest of the system.

Header comments

One method for writing a lengthy descriptive comment for a Forth word is to use a header block. This is a block of comments just above the start of the word, which describes the function of the word in detail. This is normally detail or description which would not fit well in the in-line comments down the right hand side of the page:

```
\ this word ...
\ ...
\ ...

: word1          \ a b -- c ; does ...
 ...                      \ ...
;
```

Because of the way our documentation tools work, MPE puts these comments after the definition line. What matters is to be consistent.

We tend to use header blocks before a group of words for design information that covers the group.

Name and stack comment

The first line of a definition will consist of the start of the definition - either a colon **:** or a **CODE**, or label, etc. and the name of the procedure. This will then be followed by the stack effect for the word. In our standard, stack comments start at column 17 (second tab stop).

```
: WORD1          \ n1 -- n2 n3 ; description
 ...      \ ...
;

: WORD2          ( n1 -- n2 ; description )
 ...      \ comment
;

CODE WORD3       \ n1 -- n2 n3 ; description
  ...
END-CODE
```

The **:**, etc. will start at the very left-hand end of the line. There will be one space between this and the name of the word.

The stack comment and description will start some way across the line - but further towards the left and the word name than the in-line comments. **THERE WILL ALWAYS BE A STACK COMMENT**. It is a major sin if the stack comment is incorrect. Within the stack effect, execution will be identified by one of the recognised marks:

```
-- --- -->
```

The first of these is our standard. The description is recommended, as formal source-scanning tools will look for this rather than the others. If it becomes necessary to also document the return stack effect, use

```
 R: a -- bc
```

Some people use **++** in place of **--** for return stack comments. However, many people use **F:** to indicate the floating point stack, so R: is consistent with this. If you need special additional stacks you can extend the notation, e.g. with **S:**, to indicate these. If you need multiple stack effects separate them with " **;** ", e.g.

```
: >R      \ x -- ; R: -- x ; copy one item to r.stack
  ...
;
```

It is good practice to follow the stack effect with a short description of the action of the word - about three or four words:

```
: D*      \ d1 d2 -- d3 ; double multiply
```

If there is a short description of the word, it should be separated from the stack effect by a semicolon (;) or other obvious character. This will distinguish the description from a stack effect consisting of descriptive names for the stack items. Using a

standard semicolon, other formal tools such as source analysers can correctly handle the source code and the comments.

Indenting and phrasing

The body of the word - the words it calls, or the assembler mnemonics it uses will be indented from the left-hand end of the line. This indent will be uniform throughout the file, and will normally be two spaces in MPE code, but Forth Inc and others often use three.

Each line of code in a definition should constitute a readable and meaningful phrase. If you have more than one phrase on a line separate them by two spaces. Forth should not be laid out so vertically that each line is individually meaningless. A single phrase will consist of enough code to perform some appropriate part of the application:

```
VAR @ 10 +
OVER 4 <
SWAP 3 + BILL +!
```

End of definition

At the end of the definition, the final word, the semicolon or **END-CODE** will not be indented. This ensures that the end of the definition can be found easily. It also helps when code is added to the end of a word, by avoiding the possibility of having several semicolons at or near the end of the word.

```
: WORD1          \ n1 n2 -- n3 ; function to ...
 ...
 ...
;

CODE WORD2       \ n1 n2 -- n3 ; function to ...
 ...
 ...
END-CODE
```

If a word is to be made **IMMEDIATE**, the word to make it so should appear just after the semicolon or **END-CODE**:

```
: WORD1          \ n1 n2  n3 ; function to ...
 ...
 ...
; IMMEDIATE
```

If the word were to be placed on the line following the end of the definition, though legal, there would be a possibility of another word being inserted between the two, and the first word then losing its immediate status. Similarly, an assignment of the action of a **DEFER**red word to resolve a forward reference should appear on this line

Comments

Definitions should be commented as well as possible. In a text file there is no excuse for not having enough room to write comments, and so comments should be used liberally. In a definition, there should be comments down the right-hand side of the page, in parallel with the code. These comments should start in a uniform column, which should as far as possible be consistent throughout the file. This column should be further to the right than the starting column for the stack comment and short description, usually about half way across the page. We default to column 41.

```
: WORD1          \ n1 n2 -- n3 ; function to ...
 ...                            \ get the pointer
 ...                            \ modify the address
;
```

Line comments are best started with the **** word - comment to end of line. This is in preference to the **(** word, which must be terminated with a **)**. This last is easily forgotten. These comments should not be on the stack-detail level, though this may be appropriate in certain cases. They should, however, give descriptive information on the state of the system at that point - describing the overall action of the line of code, of the phrase. Needless to say, comments should also be correct.

On a point of style, it is better if the editor inserts tabs between the code and the comment than a series of spaces. This leaves less tidying-up to do after small changes to a line of code. It also makes the source file more compact on disc and faster to load, although given the speed of modern desktop computers and the size of their discs, this is now of secondary importance.

Defining words

Defining words present a special case of definition. This is because, as the word breaks down into two parts, more care should be given both to indentation and to commenting:

```
: WORDN          \ n1 -- ; -- n3 ; function to ...
  CREATE  \ defining portion
    ...          \ lay down data
  DOES>          \ execution portion
    ...          \ process data ...
;
```

The **CREATE** and **DOES>** words should be indented to below the name of the word. The code in the **CREATE** and **DOES>** portions should then be indented by further spaces. The layout of **DOES>** and **;** also applies to **;CODE** and **END-CODE**. It is also often found useful to document the stack action of the relevant portion of the word on the line with the **CREATE** and **DOES>** words:

```
: WORDN          \ n1 n2 -- ; -- n3 ; function to ...
  CREATE  \ n1 n2 -- ; defining portion
    ...          \ lay down data
  DOES>          \ -- n3 ; execution portion
    ...          \ process data ...
;
```

Control Structure layout

Control structures should be laid out for ease of understanding, and to easily spot overlapping or incomplete structures. To this end, indenting and the use of many lines makes the layout easy, especially for inexperienced Forth programmers.

Flags and limits

As Forth uses a postfix notation, the flag used to control program flow is specified before the structure or test which uses it. The flag should be identified on the line immediately preceding the test which will use it, as should loop limits:

```
VAR @                \ get flag
IF                           \ if set  ...
  ...
THEN
```

```
VAR @ 0               \ make loop limits
DO                           \ for each ...
 ...
LOOP
VAR1 @ VAR2 @ AND            \ this and this
VAR3 @ OR                    \ or this
IF
 ...
THEN
```

Indenting

For ease of reading, the start and end words of a control structure should be placed on lines by themselves. This makes them easy to spot - for presence or absence. The code within the structure should then be indented by a uniform amount:

```
...
DO
  ...
LOOP
...

...
IF
  ...
ELSE
 ...
THEN
...

...
BEGIN
  ...
 WHILE
  ...
REPEAT
...

...
CASE
  ... OF ... ... ENDOF       \ case 1
  ... OF ... ENDOF    \ case 2
  ... OF ...          \ big case 3
         ...
      ENDOF
      ...                    \ default
ENDCASE
...
```

At the end of a control structure, the structure termination word will be without indentation and back below the start of the structure, ensuring that starts and ends of structures are vertically aligned, it is easy to see an unbalanced structure or piece of code. See above for examples.

Short Structures

If the code within a control structure is very short, then it is good practice to leave the start and end of the structure on one line, with the body of the structure. However, what constitutes a short structure is very subjective.

```
...
DO  I .  LOOP
```

Note that there is more than one space between the **DO** and the **I .**, and again to the **LOOP**. This helps the code to retain phrasing.

I've changed my mind

The above layout for control structures was defined in the late 1980s when we trained a large number of people with very little programming experience. We were teaching them Forth as a first programming language. Nowadays, the majority of the people we train have had some exposure to programming. Teaching a Java programmer the meaning of an address and I/O programming can be a frightening experience just after doing a bomb-disposal machine!

Over the years, my coding style has changed, for the better I hope. I have also been able to follow the code produced by several good Forth programmers over a period of several years, as they move from novice Forth programmers to very good Forth programmers. Once a programmer becomes fluent in Forth, he (and there are very few shes in programming, but that's a separate topic) tends to write longer definitions. This is a mistake, because code reuse goes down and "call by text editor" goes up (a bane of C programmers), leading to less maintainable and less modifiable code.

Particularly for **IF ... ELSE ... THEN** and **DO ... LOOP** structures, I now recommend that **IF** and **DO** are at the **end** of the line that generates the flag or limits.

```
... DO          \ -- a b
  ...
LOOP
...

... IF          \ -- c d
  ...
ELSE
  ...
THEN
```

There are two reasons for this. The first reason is that I am now writing shorter words, and so I want to see more words on the screen at the same time, and so I want fewer lines per word without sacrificing layout. The second reason is that I **know** that **IF** consumes a flag and that **DO** consumes an index and a limit. What matters, both for me and for less experienced programmers, is to remember the stack condition at the repetition of the loop structure or the entry to the conditional clause. My thanks to Rick van Norman for my introduction to (and his persistent promotion of) this format.

Layout of code definitions

The layout of code definitions will be slightly different from the layout of high-level definitions. For a start, the layout will be more vertical than the corresponding high-level code. If a word is being defined, the top line of the definition will reflect that of any other word - ie it will have a stack comment and a brief description. If a label is being defined, then there may not be a stack effect, but there will still be a brief description of the function of the procedure or sub-routine. The code that then follows may be very vertical, or may be phrased more:

```
CODE WORD1        \ n1 -- n2 ; word to ...
  MOV AX, BX
  ADD BX, # 03
  XCHG BP, SP  INC BP  INC BP  XCHG BP, SP
```

```
   ...
END-CODE
```

Of course, there will still be plenty of in-line comments.

Constants, Values and Variables

Constants require an input value as part of their definition. This value should appear at the left-hand end of the line. If several constants are being defined at one time, the word **CONSTANT** should line up vertically, especially if the values correspond to items of the same size, such as 8/16/32 bit masks.

It is good practice to define all constants, values and variables in one place in the file, or in one file in the set. If a variable or constant is only used in one section of a file, of course it may be defined in that section. However, the appearance of such variables may indicate an unwillingness to use the Forth data stack because it is getting too deep. This is often a sign of badly factored code.

They all require stack comments! This is not just because the stack comment is a place to indicate what data type is being returned, but also because they need documentation too.

If variables are to be pre-initialised to anything other than zero, the initialisation value should follow the definition of the variable:

```
VARIABLE BILL    \ -- addr ; holds what he has
   25 BILL !
VARIABLE BEN            \ -- addr ; holds what he has
  FLOWERPOT BEN !
```

Buffers

A buffer may be defined and require more than a cell of dictionary space. This space may be pre-initialised, or it may be a scratch area, or otherwise filled by the application. The buffer should be defined and any pre-initialisation should immediately follow its definition:

```
CREATE BILL      \ -- addr ; the ...
  10 ALLOT                    \ for a hosted Forth in RAM
  S" this" BILL PLACE   \ preset to this
```

If you are writing for both hosted and embedded systems, RAM-based buffers can be defined using **BUFFER: (len -- ; -- addr)**, e.g.

```
#12 BUFFER: BILL \ -- addr
```

BUFFER: is defined in the draft ANS cross compiler proposal. If your hosted system does not provide **BUFFER:** you can define it very easily as:

```
: BUFFER: \ len -- ; -- addr ; create len-byte buffer
  CREATE  ALLOT
;
```

Data Tables

A table may be predefined - such as a look-up table. This will usually be created in the dictionary, and will include its data. The important point is consistency and ease of reading:

```
CREATE TABLE     \  -- addr ; bit-pattern table
    1 C,   2 C,   4 C,   8 C,  \ b0..b3
   16 C,  32 C,  64 C, 128 C,  \ b4..b7
```

Note that the numbers and the commas line-up. This makes reading easy. Note however that these are bit patterns. The more pedantic may prefer the following:

```
CREATE TABLE     \  -- addr ; bit-pattern table
  $01 C,  $02 C, $04 C,  $08 C,       \ b0..b3
  $10 C,  $20 C, $40 C,  $80 C,       \ b4..b7
```

Case questions

The case of the words used in a Forth application is a very delicate issue as different programmers have different preferences. All MPE Forth systems are case-insensitive, and so the case used is only a recommendation, not a requirement.

It is recommended that lower case be used throughout. Firstly, this is easier to type, and secondly it is easier to read. Using upper case throughout is **not** recommended.

However, it may be found that certain classes of words are better capitalised. These might be control structures or constants, or the name of the word as it is defined. Each organisation will find any preference and use it. An important point to remember with any decision made is to be consistent throughout all source code. The following examples are deliberately in mixed case, and do not follow previous convention.

```
: WORD1          \ n1 n2 -- ; word does ...
  ...                   \ lower case code
  ...
;

: word2          \ n1 -- n2 ; word to ...
 ... IF                 \ lc code, UC structure
   ...
   ...
 THEN
;

23 CONSTANT BILL \ n – avoids magic number
: word3          \ n1 -- ; function to ...
  ...                   \ lower case code
  BILL +         \ upper case constant
  ...                   \ lower case code
;
```

19 Exercises

Stack operations

In general these little exercises are best done by dealing with the bottom of the stack first and reordering the top afterwards.

Exercise 1.1: Define the word **2DUP** in terms of stack operations on single numbers.

```
: 2DUP          \ x1 x2 -- x1 x2 x1 x2
  <your code>
;
```

Exercise 1.2: Define **`NIP ( x1 x2 -- x2 )`**

Exercise 1.3: Define **`TUCK ( x1 x2 -- x2 x1 x2 )`**

Exercise 1.4: Define the word **3DUP** in terms of stack operations on single and double items.

```
: 3DUP          \ x1 x2 x3 -- x1 x2 x3 x1 x2 x3
  <your code>
;
```

Exercise 1.5: Define the word **`4DUP`** in terms of stack operations on single and double items.

```
: 4DUP          \ x1 x2 x3 x4 -- x1 x2 x3 x4 x1 x2 x3 x4
  <your code>
;
```

Arithmetic

Exercise 2.1: Give Forth commands that would print in decimal the largest -ve number and the largest +ve number that can be held as signed 16 bit numbers.

Hint: The hexadecimal representation of these numbers is -8000 and 7FFF for a 16 bit Forth.

Exercise 2.2: Would you expect the command sequence **`-15000 4 * .`** to give the correct answer on a 32 bit Forth and on a 16 bit Forth? If not why not?

Exercise 2.3: Consider the evaluation of the following expressions:

```
100 0 /
-10 4 /
-32768 -1 /
```

The ANS Forth specification gives two preconditions for the division operation for **`( n1 n2 -- n3 )`**. Firstly, the denominator n2 must not be zero, and secondly n1/n2 must lie within the range of signed single-cell numbers. If either of these conditions is violated, an error condition exists, but the specification does not tell what will happen in such cases.

Are the pre-conditions for **`/ ( n1 n2 -- n3 )`** satisfied in these expressions? If so, what are the values of n3? Would there be a difference between the results on 16 and 32 bit systems? How will the results vary for floored and symmetric division?

Exercise 2.4: Define the word **FUNC2** which has the following specification:

```
FUNC2     a b c -– d
d = a + 7*(b+c)
```

Exercise 2.5: Here are the specifications of some functions that can be coded in Forth using standard arithmetic operators along with **DUP** and **SWAP**. Edit their Forth definitions onto a file, compile it and test the definitions.

```
FUNC5     x y –- z
z = x^2 + 5y

FUNC6     x y –- z
  z = 3*x^2 + 2y + y^2

FUNC7     x y –- z
z = 2 + x + x^2 + y + y^2

FUNC8     a b c –- d
d = a^2 + b^2 + c^2

FUNC9     x y –- z
z = x^2 + 2*x*y + y^2
```

Hint: this one may need a bit of lateral thinking. The definition consists of just 3 Forth words!

Exercise 2.6: Define Forth words with the following specifications.

```
FUNC10    x y -– z
z = x+xy

FUNC11    a b c -- a^2 b^2 c^2
```

Expects 3 stack items, removes them from the stack, and replaces them with their squares.

Exercise 2.7: Define words **OCTAL** and **BINARY** which will set the system to base 8 and base 2 respectively. Try them out doing some octal and binary arithmetic. What do you expect the system's response will be to these commands?

```
HEX     BASE @ .
OCTAL   BASE @ .
BINARY  BASE @ .
```

(If you think it will be to print 16, 8 and 2, think again!)

Define a word **.BASE** that will print the current number base in decimal. **.BASE** should leave the number base as it finds it.

Define a word **.BIN** that prints the top item on the stack as an unsigned binary number. **.BIN** should leave the number base as it finds it.

Exercise 2.8: The word **+!** is present in all Forth all systems, but can you define your own version using the Forth words described so far? Call your version **MY+!** and test it as shown above.

Hint: A solution can be obtained using the words

```
@ ! + OVER SWAP
```

An alternative solution can be obtained using

```
@ ! + DUP ROT SWAP
```

Exercise 2.9: Define variables **TEMP** and **PRESSURE** and a word **CHECK** with the following specification.

```
-- flag
```

Flag is true if the value stored in **TEMP** is less than 700 and the value stored in **PRESSURE** is between 1200 and 2200 inclusive. Make sure that your definition of **CHECK** produces the following results.

```
600 TEMP !  1200 PRESSURE !  CHECK . ( -1 ok )
2200 PRESSURE !  CHECK . ( -1 ok }
2201 PRESSURE !  CHECK . ( 0 ok )
2200 PRESSURE !  700 TEMP !  CHECK . ( 0 ok )
1199 PRESSURE !  -1 TEMP +!  CHECK . ( 0 ok )
```

Exercise 2.10: Define the word **TEST** with the following specification.

```
n -- flag
```

Leaves a true flag if n < 50 or n >100, and leaves a false flag otherwise.

Exercise 2.11: Define the words **>=** and **<=** with the following specs.

```
>=          \ n1 n2 -- flag
```

Flag is true if n1 is greater than or equal to n2

```
<=          \ n1 n2 -- flag
```

Flag is true if n1 is less than or equal to n2 and false otherwise.

Exercise 2.12: Trace the execution of **20 10 MAX**.

Code and test a definition for **MIN**. This has the specification:

```
MIN       n1 n2 -- n3
```

Where n3 is the signed minimum of n1 and n2.

Exercise 2.13: Define versions of **MAX** and **MIN** with an **IF ... THEN** construct, i.e., without using an **ELSE**.

An **IF** construct need not have an **ELSE**. For example, consider a possible definition for the standard word **ABS**. This takes a signed number from the stack and replaces it with a positive number of the same magnitude, as in:

```
100 ABS . <cr> 100 ok
-200 ABS . <cr> 200 ok
```

The definition of **ABS** requires a word we have not introduced yet, which is **NEGATE**. It just changes the sign of the top stack item.

```
: ABS ( n -- abs[n] )
    DUP 0<
    IF NEGATE THEN
;
```

Exercise 2.14: Define words to meet the following specifications.

```
TEST1     n1 -- n2

If $n_1 > 100$
then $n_2 = 2 times n1
else n2 = n1 - 20

TEST2     n1 n2 -- n3

If n1 > n2
then n3 = 1000
else n3 = n1
```

Exercise 2.15: Suppose the variables **TEMP** and **HEATER-SWITCH** have been declared as part of an industrial simulation. **TEMP** holds a simulated temperature and **HEATER-SWITCH** holds a value that indicates whether a simulated heater is on or off. Code a word **STC** (``simulate temperature change'') which has the following spec.

If the value held in **HEATER-SWITCH** is non zero, add 2 to the value held in **TEMP** otherwise subtract 1 from the value held in **TEMP**.

Exercise 2.16: Define a word **.B** (``print boolean'') which removes the top stack item and prints ``true'' if its value is non zero and ``false'' if its value is zero. The word can be used like this:

```
5 6 < .B <cr> true ok
```

Note that although the definition of **.B** contains an **IF** construct it does not need to contain a test!

Exercise 2.17: Studies have suggested that Forth applications use ``if'' structures much less frequently than C or Pascal. If this is the case it is probably because Forth encourages a style which ``hides'' decisions within other words. For example the definition of **ABS** can be written without an **IF**, can you see how?

Hint: The definition is three words long, and the word you need to make the hidden decision is **MAX**.

Exercise 2.18: A very early description of an algorithm for computing a mathematical result is Euclid's method for computing the greatest common divisor of two numbers. This algorithm is presented at the start of Euclid book 7, along with a proof that it will always generate the correct result.

Recall that the GCD of two whole numbers is the greatest whole number that can be divided into both of them with no remainder. So the GCD of 16 and 24 is 8 for example. In Euclid's time (around 300B.C.) numbers were reasoned about in terms of lines of a certain length. Instead of using x to represent a quantity they would talk about a straight line AB. The length of AB would represent the quantity under discussion. Give two lines AB and CD, each an exact number of units in length, the problem is to find the longest line that can be used to measure both AB and CD

exactly. For example, given lines AB and CD of lengths 26 and 16 units, a line of 8 units will fit into AB three times and into CD two times. Thus 8 becomes the greatest common "measure" of 24 and 16.

An important idea associated with common measure came from Eudoxus, a pupil of Plato and a renowned mathematician active around 400 B.C. He noted that the greatest common measure of lines AB and CD is the same as that of lines AB and AB + CD. To see this, we first reason that the greatest common measure of AB and CD must be a common measure of AB and AB + CD, for since it can measure both AB and CD it can measure AB + CD. Secondly, we reason that it must be the greatest common measure, for if there were a greater common measure for AB and AB + CD, this would measure AB, hence it would measure the AB part of AB + CD, and hence it would have to be able to measure CD since it is a measure for AB + CD, and thus it would be a common measure for AB and CD, which contradicts our assertion that it is greater than the greatest common measure of AB and CD.

Euclid inverted this. Let EF be the same length as AB + CD, and consider the lines (numbers) AB and EF. Subtract the smaller from the larger (i.e., AB from EF) and we obtain a line of length CD. Now lines AB and CD have the same common measure as lines AB and EF. The importance of inverting Eudoxus' procedure is that it gives a process that when repeated, must finish, since we are generating pairs of lines that are getting shorter by some units of length at each stage.

This suggests the following algorithm, which is a slight simplification of that found at the start of Euclid book 7. To find the GCD of any two numbers:

1) Using any unit measure draw two lines whose measures are equal to the two numbers (e.g., for numbers 24 and 16 draw lines 24 units and 16 units in length).

2) Label the larger line AB and the smaller CD.

4) Subtract CD from the length of AB to give lines CD and AB - CD.

5) If these two lines are equal in length, this length is the greatest common measure of the original two lines. If not take CD and AB – CD as a new pair of lines for the next stage of the method.

6) Continue at step 2.

Code Euclid's algorithm in Forth, with the following specification:

```
GCD       x y -- z
```

z is the greatest common divisor of x and y.

Hint: Define a word **ORDER** which removes two numbers from the stack and replaces them in the order, smaller larger. Use **ORDER** within a **BEGIN ... WHILE ... REPEAT** loop.

Exercise 2.19: Code a version of GCD using Euclid's algorithm and a **BEGIN ... UNTIL** loop. Perform at least the following tests on your definition:

```
16 24 GCD .<cr> 8 ok
24 16 GCD . <cr> 8 ok
 8  8 GCD .<cr> 8 ok
```

Exercise 2.20: The combined electrical resistance R offered by two resistors with values R1 and R2 connected in parallel is given by:

```
R = {{R1 * R2} / {R1 + R2}}
```

Define a word **//RES** with the following spec:

```
//RES             R1 R2 -- R
```

Leaves R, the resistance obtained by connecting R1 and R2 in parallel.

Hint: The best solution is three words long.

Define a word **///RES** which calculates the value of three resistors connected in parallel.

Hint: The best solution is two Forth words long and uses **//RES**.

Exercise 2.21: **MOD** can be used to define the greatest common divisor function. Given any numbers x and y we can obtain new values x' and y' from:

```
x' = y
y' = x mod y
```

If we carry on generating new pairs of numbers like this until the y value is zero, the corresponding x value will be the greatest common divisor of the original x and y. For example if the original values of x and y are 24 and 16, we would get the following x and y values during the calculation:

x	y	Notes
16	24	16 mod 24 = 16, this will be the next y value.
24	16	24 mod 16 = 8, this will be the next y value.
16	8	16 mod 8 = 0, this will be the next y value.
8	0	Since y = 0 we are finished, result is 8.

Using the method illustrated here define a **GCD** function using a **BEGIN ... WHILE ... REPEAT** loop and the **MOD** function. **GCD** should remove two numbers from the stack and leave their greatest common divisor.

Input, output and loops

Exercise 3.1: Define a word **X** that waits for a character to be input at the keyboard, and then outputs the character and its ASCII code. Using **[CHAR]**, define a word **STAR** that outputs an asterisk. **STAR** could be used as follows:

```
STAR <cr> *ok
```

Exercise 3.2: Define a word **TEST1** that reads characters from the keyboard and for each key entered displays both the character and its ASCII code. Terminate on receiving a carriage return code.

Define a word **TEST2** that reads characters from the keyboard, ignores all characters that are not upper case letters, and for upper case letters converts A to B, B to C, ..., Z to A and then outputs the resulting codes to the screen.

Hint: use **WITHIN** to test whether a character is an upper case letter.

Define a word **TEST3** that reads characters from the keyboard, converts upper case characters to lower case, and outputs to the screen.

Exercise 3.3: Define a word **EVENS** with the following spec:

```
EVENS     n --
```

Prints all even numbers between 0 and $2n$.

Hint: The loop limit n is on the stack when you enter **EVENS**.

Exercise 3.4: Define a version of **SPACES (n --)** called **MY-SPACES** that deals with the 0 case correctly. i.e., **0 MY-SPACES** should print 0 spaces. What would/should such a word do if n is less than 0?

Exercise 3.5: Using the word **STAR**, which you can define as:

```
: STAR  ( -- ) ASCII * EMIT  ;
```

Define a word **STARS** that takes a number from the stack and prints that many asterisks. For example:

```
5 STARS <cr> *****ok
7 STARS <cr> *******ok
```

Exercise 3.6: In the following exercises make use of the words **STAR** and **STARS** defined in the previous exercise.

Define a word **RECT** that prints an 8 by 5 rectangle of asterisks as follows:

```
RECT <cr>
  ********
  *      *
  *      *
  *      *
  ********ok
```

Hints: To keep the code looking as simple as possible use the words **STAR** and **STARS** defined in the previous exercise, and define and use the following words:

MARGIN output carriage return/line feed and 5 spaces.

HORIZONTAL output a margin and 8 stars.

VERTICALS print the vertical sides of the rectangle

Exercise 3.7: Define a word **CHARACTERS** with the following specification.

```
CHARACTERS      n --
```

Input n characters from the keyboard (where n>0) and output each one to the screen before the next is input.

Example usage could be:

```
6 CHARACTERS <cr> QWERTYok
```

Exercise 3.8: Define a word **SMALL** with the following spec.

```
SMALL     n --
```

Input characters from the keyboard, convert upper case alphabetic characters to lower case, and display the characters on the screen until an ASCII space is received or n characters have been input.

Memory

Exercise 4.1: How do you suppose **ALLOT** is defined?

How do you suppose **VARIABLE** is defined?

Exercise 4.2: Suppose we define a 16 byte buffer with:

```
CREATE BUFF1  16 ALLOT
```

Describe the effect of executing

```
HEX 1234 BUFF1 C!
```

Make sure you check your answer by actually performing the operation (in hex) and observing its effect on locations BUFF1 and BUFF1 + 1. The word **DUMP (addr len --)** is usually available.

Exercise 4.3: Define you own version of **DUMP**. Call it **MY-DUMP**. Don't worry too much about the finer details like formatting the output in columns and displaying the ASCII characters.

Define **WDUMP** which takes the same arguments as **DUMP** but displays the 16 bit contents of count words starting at addr.

Hint: Use the control structure

```
DO ... 2 +LOOP
```

at **+LOOP** the top stack item is removed and added to the index, so in this case the index will go up in steps of 2. This kind of loop terminates when the index reaches or passes the limit.

If you are using a 32 bit Forth system define **LDUMP** which takes the same arguments as **DUMP** but displays the 32 bit contents of count words starting at addr.

Exercise 4.4: Bytes of data are compiled with **C,**. Write a specification for this word, and show how you would use it when setting up a 4 byte table called **NAME** containing the ASCII codes for ``J'' ``O'' ``H'' and ``N''.

Suggest how the word ``comma'' is defined.

Hint, the words **HERE** and **ALLOT** will be useful.

Exercise 4.5: Show how **FILL (addr len char --)** could be defined using **CMOVE**, not as a block move, but in the ``wrong direction''. Then use **FILL** to define the words **ERASE** and **BLANK**. The words **FILL**, **ERASE** and **BLANK** have all been described earlier.

Exercise 4.6: Consider a temperature sensor connected via an A to D converter and giving readings that vary between 0 at 5 C and 255 at 160 C. Assuming that the variation in the reading is linear with temperature define a word called **DEGREES** that will convert a temperature value in degrees C to the equivalent A to D converter reading.

Exercise 4.7: Define a word **OF-TEMP-TABLE** with the following spec.

```
OF-TEMP-TABLE    n -- t
```

t is the temp of element n in the table (0 <= n < 5).

Example usage:

```
2 OF-TEMP-TABLE . <cr> 3349 ok
```

Exercise 4.8 Non linear temperature conversion via table lookup and interpolation.

Suppose a temperature sensor connected to an A to D converter has been calibrated with the results as shown in the following table.

```
600   12.36
1100  23.42
1600  33.49
2100  42.02
2600  50.10
```

We consider how to define a word **READING->DEG** which converts a reading taken from the converter to the equivalent temperature. First we see how to do the conversion by hand, without any particular reference to Forth. Suppose an A to D reading of 1507 is received. This is between the values 1100 and 1600 in the table, so the required temperature will be between 23.42 and 33.49.

To estimate the temperature we use a technique called ``linear interpolation''. The reading value 1507 is 81.4% (407/500) of the way between table values 1100 and

1600, and with linear interpolation we estimate the corresponding temperature to be 81.4% of the way between 23.42 and 33.49.

```
temp = 23.42 + {407 / 500} * ( 33.49 - 23.42 )
```

Since we are limiting ourselves to integer arithmetic and temperatures are given to two decimal places of accuracy, we will scale the temperatures by a factor of 100 and hold them in a table defined as follows:

```
CREATE TEMP-TABLE    \ -- addr
   1236 ,  2342 ,  3349 ,  4202 ,  5010 ,
```

Note that we have not bothered to record the A to D readings in a table. This is because they occur at regular intervals and the data they provide can easily be encoded in the logic of our program.

We will define a word to read values from the temperature table. You were asked to provide this definition in the previous exercise.

Now here is the start of one possible way to define **READING->DEG**.

```
: READING->DEG ( n – t )
\ convert a-d reading to scaled temperature
  600 -
  500 /MOD ( r q )
  ...
;
```

To see what is happening here, consider the execution of the example we took before:

```
1507 READING->DEG
```

The values shown as r and q in the comment that follows /MOD will then be r=407 and q=1. The value of q tells us that the required temp is between **1 OF-TEMP-TABLE** and **2 OF-TEMP-TABLE**. The value of r tells us that the required temp will be r 500ths of the way between these two values.

Complete the definition of **READING->DEG** using the comments in the following outline code. Note that in the comments t[q] is used to represent the temp at position q of the temperature table (q=0 to 4).

```
: READING->DEG ( n -- t )
\ convert a-d reading to scaled temperature
  600 -
  500 /MOD     ( -- r q )
   ( -- r q q+1 )
               ( -- r t[q] t[q+1] )
   ( -- r t[q] t[q+1]-t[q] )
( -- t[q] {t[q+1]-t[q]}*r/500 )
( -- t[q]+{t[q+1]-t[q]}*r/500 )
;
```

Note the use of the **{}** brackets in place of the **( )** within the comment. This is because the **)** is used to indicate the end of the comment. The use of the backslash comments would have avoided this problem.

Defining words

Exercise 5.1: Define a word **+CONSTANT** which defines words which record a 16 bit value at compile time and add this word to the value at the top of the stack at run time. Example usage:

```
 5 +CONSTANT PLUS5
10 +CONSTANT PLUS10
100 PLUS5 PLUS10 . <cr> 115 ok
```

Exercise 5.2: Complete the following definition of BYTES.

For our next example, we create a defining word **BYTES** which can be used to set up tables of 8 bit values. Example usage of **BYTES** is:

```
10 BYTES BTABLE1
20 BYTES BTABLE2
77 0 BTABLE1 C! ( store 77 in byte 0 of BTABLE1 )
40 1 BTABLE1 C! ( store 40 in byte 1 of BTABLE1 )
```

So **0 BTABLE1** returns the address of byte 0 in **BTABLE1**, **1 BTABLE1** returns the address of byte 1 in **BTABLE1**, etc.

Hints: You need to add just one word of compile time action and just one word of run time action.

The comment that follows **DOES>** shows what is on the stack when the code following **DOES>** commences execution.

```
 : BYTES
CREATE ...
   DOES> ( -- index addr ) ... ;
```

Exercise 5.3: Provide definitions for **WITEMS** and **LITEMS** which can be used like **BYTES** but defines tables whose elements are 16 and 32 bits wide. For example **10 WITEMS TABLE1** would define an array of 10 16-bit items called **TABLE1**.

Exercise 5.4: Another use of **CREATE ... DOES>** is to define the opcode classes for an assembler. The simplest class of opcodes for an assembler to handle are those which have a fixed opcode. Here are some such opcodes for the 8086 instruction set

```
Opcode    Mnemonic

C3 RET
FA CLI
FB STI
9B WAIT
F0 LOCK
```

In a Forth assembler each of the mnemonics such as **RET** and **CLI** would be defined as a Forth word. When executed these words compile their associated 1 byte opcode into the next free byte of dictionary memory.

Suppose the word **CLASS1** is used to define the given instructions. Usage will be:

```
HEX
C3 CLASS1 RET
```

```
FA CLASS1 CLI
...
```

The compile time action for **CLASS1** is to compile the 1 byte opcode into the next free dictionary location. The run time action is to retrieve the opcode from the parameter field and compile it into the next free byte of the dictionary. For example the action of **RET** is to compile the byte C3, which is the opcode for the 8086 RET instruction.

Now see if you can define **CLASS1**.

Miscellaneous

Exercise 6.1: Suppose the following definitions are loaded.

```
: TEST1 ." old version " ;

: TEST2 TEST1 ;

: TEST1 ." new version " ;
```

What will be the output generated by:

```
TEST1 TEST2
```

20 Solutions to Exercises

Stack operations

Solution 1.1: Define the word **2DUP** in terms of stack operations on single numbers.

```
: 2DUP         \ x1 x2 -- x1 x2 x1 x2
  OVER OVER
;
```

Exercise 1.2: Define **`NIP ( x1 x2 -- x2 )`**

```
: nip    \ x1 x2 -- x2
  swap drop
;
```

Solution 1.3: Define **`TUCK ( x1 x2 -- x2 x1 x2 )`**

```
: tuck   \ x1 x2 -- x2 x1 x2
  swap over
;
```

Solution 1.4: Define the word **3DUP** in terms of stack operations on single and double items.

```
: 3DUP           \ x1 x2 x3 -- x1 x2 x3 x1 x2 x3
  dup 2over rot
;
```

Solution 1.5: Define the word **`4DUP`** in terms of stack operations on single and double items.

```
: 4DUP         \ x1 x2 x3 x4 -- x1 x2 x3 x4 x1 x2 x3 x4
  2over 2over
;
```

Arithmetic

Solution 2.1: Give Forth commands that would print in decimal the largest -ve number and the largest +ve number that can be held as signed 16 bit numbers.

```
HEX -8000 DECIMAL .
HEX 7FFF DECIMAL .
```

Solution 2.2: Would you expect the command sequence **`-15000 4 * .`** to give the correct answer on a 32 bit Forth and on a 16 bit Forth? If not why not?

No, because the ``correct'' result is -60000 which cannot be expressed as a 16 bit 2's complement number and so fails on a 16 bit system.

Solution 2.3: Consider the evaluation of the following expressions:

```
100 0 /
  n2=0 => fail
-10 4 /
  Symmetric: n2<>0 => ok, result –2 in range
  Floored  : n2<>0 => ok, result –3 in range
-32768 -1 /
  n2<>0 => ok, result = 32768, fail on 16 bit system.
```

Solution 2.4: Define the word **`FUNC2`** which has the following specification:

```
     FUNC2     a b c -- d
     d = a + 7*(b+c)

: FUNC2  \ a b c -- d
  +  7 *  +
;
```

Solution 2.5: Here are the specifications of some functions that can be coded in Forth using standard arithmetic operators along with **DUP** and **SWAP**. Edit their Forth definitions onto a file, compile it and test the definitions.

```
     FUNC5     x y -- z
     z = x^2 + 5y

: FUNC5  \ x y -- z
  5 *  SWAP  DUP *  +
;

     FUNC6     x y -- z
       z = 3*x^2 + 2y + y^2

: FUNC6  \ x y -- z
  DUP DUP       \ -- x y y y
  *  SWAP       \ -- x y^2 y
  2 *  +        \ -- x y^2+2y
  SWAP DUP *    \ -- y^2+2y x^2
  3 * +         \ -- z
;

     FUNC7     x y -- z
     z = 2 + x + x^2 + y + y^2

: FUNC7  \ x y -- z
  DUP DUP * +         \ -- x y+y^2
  SWAP                \ -- y+y^2 x
  DUP DUP * +         \ -- y+y^2 x+x^2
  +  2+
;

     FUNC8     a b c -- d
     d = a^2 + b^2 + c^2

: FUNC8  \ x y -- z
  DUP *         \ -- a b c^2
  SWAP DUP *    \ -- a c^2 b^2
  +             \ -- a b^2+c^2
  SWAP DUP *    \ -- b^2+c^2 a^2
  +
;

     FUNC9     x y -- z
     z = x^2 + 2*x*y + y^2

     Use x^2 + 2xy + y^2 = (x+y)*(x+y).

: FUNC9  \ x y -- z
  + DUP *
;
```

Solution 2.6: Define Forth words with the following specifications.

```
     FUNC10    x y -- z
     z = x+xy

: FUNC10 \ x y - z
  OVER * +
;
```

```
FUNC11   a b c -- a^2 b^2 c^2
```

Expects three stack items, removes them from the stack, and replaces them with their squares.

```
: FUNC11 \ x y -- z
  DUP *              \ -- a b c^2
  ROT DUP *          \ -- b c^2 a^2
  ROT DUP *          \ -- c^2 a^2 b^2
  ROT                \ -- a^2 b^2 c^2
;
```

Solution 2.7: Define words **OCTAL** and **BINARY** which will set the system to base 8 and base 2 respectively.

```
: OCTAL  ( -- )  8 BASE !  ;
: BINARY ( -- )  2 BASE !  ;
: .BASE  \ --
  BASE @  DUP DECIMAL .  BASE !
;
: .BIN   \ n --
  BASE @ SWAP        \ -- base n
  BINARY U.
  BASE !
;
```

Solution 2.8: The word **+!** is present in all Forth all systems, but can you define your own version using the Forth words described so far? Call your version **MY+!** and test it.

```
: MY+!   \ n addr --
  SWAP OVER @  +  SWAP !
;

: MY+!   \ n addr --
  DUP @ ROT +  SWAP !
;
```

Solution 2.9: Define variables **TEMP** and **PRESSURE** and a word **CHECK** with the following specification.

```
-- flag
```

Flag is true if the value stored in **TEMP** is less than 700 and the value stored in **PRESSURE** is between 1200 and 2200 inclusive. Make sure that your definition of **CHECK** produces the following results.

```
600 TEMP !  1200 PRESSURE !  CHECK . ( -1 ok )
2200 PRESSURE !  CHECK . ( -1 ok }
2201 PRESSURE !  CHECK . ( 0 ok )
2200 PRESSURE !  700 TEMP !  CHECK . ( 0 ok )
1199 PRESSURE !  -1 TEMP +!  CHECK . ( 0 ok )
```

Solutions:

```
VARIABLE TEMP        \ -- addr
VARIABLE PRESSURE    \ -- addr

: CHECK        \ -- flag
```

```
  TEMP @ 700 <                    \ -- flag1
  PRESSURE @ 1200 1201 WITHIN     \ -- flag1 flag2
  AND
;
```

Alternative solution:

```
: CHECK         \ -- flag
  TEMP @ 700 <
  PRESSURE @ 1199 >         \ -- flag1 flag2
  PRESSURE @ 2201 <         \ -- flag1 flag2 flag3
  AND AND
;
```

Solution 2.10: Define the word **TEST** with the following specification.

```
     n -- flag
```

```
: TEST          \ n –- flag
\ Leaves a true flag if n < 50 or n >100, and leaves
\ a false flag otherwise.
  DUP 50 <      \ -- n flag1
  SWAP 100 >    \ -- flag1 flag2
  OR
;

: TEST          \ n -- flag
  50 101 WITHIN 0=
;
```

Solution 2.11: Define the words **>=** and **<=** with the following specs.

```
     >=          \ n1 n2 -- flag
```

Flag is true if n1 is greater than or equal to n2

```
     <=          \ n1 n2 -- flag
```

Flag is true if n1 is less than or equal to n2 and false otherwise.

```
: >=     ( n1 n2 -- flag )  < 0=  ;
: <=     ( n1 n2 -- flag )  > 0=  ;
```

Solution 2.12: Code and trace the execution of **`20 10 MAX`**, where **MAX** is defined as:

```
: MAX           \ n1 n2 –- flag
  2dup >
  if  drop  else  swap drop  then
;
```

You may have to perform the trace manually, depending on the facilities of your Forth system. For example on an optimised Forth, the following machine code may result.

```
dis max
MAX
( 0049EB70    3B5D00 )                CMP     EBX, [EBP]
( 0049EB73    0F8D0B000000 )          JNL/GE  0049EB84
( 0049EB79    8B5D00 )                MOV     EBX, [EBP]
( 0049EB7C    8D6D04 )                LEA     EBP, [EBP+04]
```

```
( 0049EB7F    E903000000 )            JMP     0049EB87
( 0049EB84    8D6D04 )                LEA     EBP, [EBP+04]
( 0049EB87    C3 )                    NEXT,
( 24 bytes, 7 instructions )
 ok

2DUP           -- 20 10 20 10
>              -- 20 10 -1
IF (true)      -- 20 10
DROP           -- 20
THEN           -- 20
```

Code and test a definition for **MIN**. This has the specification:

```
MIN       n1 n2 -- n3
```

Where n3 is the signed minimum of n1 and n2.

```
: MIN          \ n1 n2 -- flag
  2dup <
  if  drop  else  swap drop  then
;
```

Solution 2.13: Define versions of **MAX** and **MIN** with an **IF ... THEN** construct, i.e., without using an **ELSE**.

```
: MAX          \ n1 n2 -- flag
  2DUP <
  IF  SWAP THEN
  DROP
;
: MIN          \ nl n2 -- flag
  2DUP >
  IF  SWAP THEN
  DROP
;
```

Solution 2.14: Define words to meet the following specifications.

```
TEST1    n1 -- n2

If $n_1 > 100$
then $n_2 = 2 times n1
else n2 = n1 - 20

TEST2    n1 n2 -- n3

If n1 > n2
then n3 = 1000
else n3 = n1
```

```
: TEST1        \ n1 -- n2
  DUP 100 >
  IF  2 *  ELSE  20 -  THEN  ;

: TEST2        \ n1 n2 -- n3
  OVER <
  IF  DROP  1000  THEN
;
```

Solution 2.15: Suppose the variables **TEMP** and **HEATER-SWITCH** have been declared as part of an industrial simulation. **TEMP** holds a simulated temperature and **HEATER-SWITCH** holds a value that indicates whether a simulated heater is on or off. Code a word **STC** (``simulate temperature change'') which has the following spec.

If the value held in **HEATER-SWITCH** is non zero, add 2 to the value held in **TEMP** otherwise subtract 1 from the value held in **TEMP**.

```
: STC          \ --
  HEATER-SWITCH @
  IF  2  ELSE  -1  THEN
  TEMP +!
;
```

Solution 2.16: Define a word **.B** (``print boolean'') which removes the top stack item and prints ``true'' if its value is non zero and ``false'' if its value is zero. The word can be used like this:

```
     5 6 < .B <cr> true ok
```

```
: .B           \ flag --
  IF  ." true"  ELSE  ." false"  THEN
;
```

Solution 2.17: The definition of **ABS** can be written without an **IF**, can you see how?

```
: ABS          \ n1 -- n2
  DUP NEGATE MAX
;
```

You can also define **ABS** without **IF** and with simpler operations than **MAX** (which often contains an internal branch):

```
: abs
  dup 0< tuck xor swap –
;
```

However, that depends on 2s-complement arithmetic and, on many CPUs, **0<** requires an internal branch.

Solution 2.18:

Code Euclid's algorithm in Forth, with the following specification:

```
     GCD       x y -- z
```

z is the greatest common divisor of x and y.

```
While they are not equal
 Arrange pair so smaller is first      16 24       8 16
 Put copy of the first after second.   16 24 16    8 16 8
 Subtract the two rightmost            16 8        8 8
Endwhile
Delete the second number                          8
```

This can be coded in Forth as:

```
: ORDER        \ n1 n2 -- smaller larger
  2DUP >
```

```
  IF  SWAP  THEN
;

: GCD          \ n1 n2 -- n3
  BEGIN
    2DUP <>
   WHILE
    ORDER  OVER -
  REPEAT
  DROP
;
```

Solution 2.19: Code a version of GCD using Euclid's algorithm and a **BEGIN ... UNTIL** loop.

```
: ORDER        \ n1 n2 -- smaller larger
  2DUP >
  IF  SWAP  THEN
;

: GCD          \ n1 n2 -- n3
  BEGIN
    ORDER  OVER -
    DUP 0=
  UNTIL
  DROP
;
```

Solution 2.20: The combined electrical resistance R offered by two resistors with values R1 and R2 connected in parallel is given by:

```
R = {{R1 * R2} / {R1 + R2}}
```

Define a word **//RES** with the following spec:

```
//RES            R1 R2 -- R
```

Leaves R, the resistance obtained by connecting R1 and R2 in parallel.

Define a word **///RES** which calculates the value of three resistors connected in parallel.

```
: //RES  ( R1 R2 -- R )  2DUP +  */  ;
: ///RES ( R1 R2 R3 -- R ) //RES //RES  ;
```

Solution 2.21: Define a **GCD** function using a **BEGIN ... WHILE ... REPEAT** loop and the **MOD** function. **GCD** should remove two numbers from the stack and leave their greatest common divisor.

```
: GCD    \ x y -- z ; z is the gcd of x and y
  BEGIN        ( -- x y )
    DUP        ( -- x y y )
   WHILE       ( -- x y )
    SWAP OVER  ( -- y x y )
    MOD        ( -- y mod )
  REPEAT
  DROP
;
```

Input, output and loops

Solution 3.1: Define a word **X** that waits for a character to be input at the keyboard, and then outputs the character and its ASCII code. Using **[CHAR]**, define a word **STAR** that outputs an asterisk. **STAR** could be used as follows:

```
       STAR <cr> *ok
```

```
: X      \ --
\ Wait for char then output char and its ASCII code
  KEY  DUP EMIT  .
;

: STAR   \ --
  [CHAR] * EMIT
;
```

Solution 3.2: Define a word **TEST1** that reads characters from the keyboard and for each key entered displays both the character and its ASCII code. Terminate on receiving a carriage return code.

```
: TEST2        \ --
  BEGIN
    KEY DUP 13 <>
   WHILE
    DUP EMIT  .
  REPEAT
  DROP
;
```

Define a word **TEST2** that reads characters from the keyboard, ignores all characters that are not upper case letters, and for upper case letters converts A to B, B to C, ..., Z to A and then outputs the resulting codes to the screen.

```
: TEST2        \ --
  BEGIN
    KEY DUP
    [CHAR] A  [CHAR] Z 1 +  WITHIN IF ( upper case )
      DUP ASCII Z = IF  ( Z code )
        DROP ASCII A
      ELSE              ( not Z code)
        1 +
      THEN
      EMIT
    ELSE
      DROP
    THEN
  REPEAT
;
```

Define a word **TEST3** that reads characters from the keyboard, converts upper case characters to lower case, and outputs to the screen.

```
CHAR a  CHAR A -  CONSTANT CASE-SHIFT

: TEST3        \ --
```

```
  BEGIN
    KEY DUP  13 <>
   WHILE
    DUP [CHAR] A [CHAR] Z 1+ WITHIN
    IF  CASE-SHIFT +  THEN
    EMIT
  REPEAT
;
```

Solution 3.3: Define a word **EVENS** with the following spec:

```
    EVENS     n --
```

Prints all even numbers between 0 and 2n.

```
EVENS ( n -- )
  0 DO  I 2 * .  LOOP
;
```

Solution 3.4: Define a version of **SPACES (n --)** called **MY-SPACES** that deals with the 0 case correctly. i.e., **0 MY-SPACES** should print 0 spaces. What would/should such a word do if n is less than 0?

```
: MY-SPACES    \ n --  ; print n spaces
  DUP IF
    0 DO SPACE LOOP
  ELSE
    DROP
  THEN
;

: MY-SPACES    \ +n -- ; print n spaces
  0 MAX  0
  ?DO  SPACE  LOOP
;
```

Solution 3.5: Using the word **STAR**, which you can define as:

```
       : STAR  ( -- ) [CHAR] * EMIT  ;
```

Define a word **STARS** that takes a number from the stack and prints that many asterisks.

```
: STARS  ( n -- )  0 ?DO  STAR  LOOP  ;
```

Solution 3.6: In the following exercises make use of the words **STAR** and **STARS** defined in the previous exercise.

Define a word **RECT** that prints an 8 by 5 rectangle of asterisks as follows:

```
       RECT <cr>
         ********
         *      *
         *      *
         *      *
         ********ok
```

```
: MARGIN  ( -- )  CR 5 SPACES  ;
: HORIZONTAL  ( -- )  MARGIN  8 STARS ;
: VERTICALS  ( -- )
  3 0
  DO  MARGIN  STAR  6 SPACES  STAR  LOOP
;
: RECT  ( -- )  HORIZONTAL  VERTICALS  HORIZONTAL  ;
```

Solution 3.7: Define a word **CHARACTERS** with the following specification.

```
CHARACTERS       n --
```

Input n characters from the keyboard (where n>0) and output each one to the screen before the next is input.

```
: CHARACTERS   \ n --
  0 DO  KEY  EMIT  LOOP  ;
```

Solution 3.8: Define a word **SMALL** with the following spec.

```
SMALL     n --
```

Input characters from the keyboard, convert upper case alphabetic characters to lower case, and display the characters on the screen until an ASCII space is received or n characters have been input.

```
CHAR a  CHAR A  -  CONSTANT CASE-SHIFT

: SMALL        \ n --
  0 DO
    KEY  DUP 13 =
    IF  DROP  LEAVE  THEN
    DUP  [CHAR] A  [CHAR] Z 1 + WITHIN
    IF  CASE-SHIFT +  THEN
    EMIT
  LOOP
;
```

Memory

Solution 4.1: How do you suppose **ALLOT** is defined?

How do you suppose **VARIABLE** is defined? Many systems contain a variable **DP** which contains the address of the next free space in the dictionary.

```
: ALLOT ( n -- )  DP +!  ;
: VARIABLE  ( -- ; -- addr )  CREATE  1 CELLS ALLOT  ;
```

Solution 4.2: Suppose we define a 16 byte buffer with:

```
CREATE BUFF1  16 ALLOT
```

Describe the effect of executing

```
HEX 1234 BUFF1 C!
```

Hex value 34 is stored at the byte address **BUFF1**. The following bytes are not changed. The top 8 bits in the value 1234 are ignored, only the low 8 bits are used.

Solution 4.3: Define you own version of **DUMP**. Call it **CDUMP**.

Define **WDUMP** which takes the same arguments as **DUMP** but displays the 16 bit contents of count words starting at addr.

```
: CDUMP        \ addr n --
  0 DO
    DUP  I +  C@ .
  LOOP
  DROP
;
: WDUMP        \ addr n --
  0 DO
    DUP  I +  W@ U.        \ use @ on 16 bit Forths
  2 +LOOP
  DROP
;
```

If you are using a 32 bit Forth system define **LDUMP** which takes the same arguments as **DUMP** but displays the 32 bit contents of count words starting at addr.

```
: LDUMP        \ addr n --
  0 DO
    DUP  I +  @ U.
  4 +LOOP
  DROP
;
```

Further improvements can be made by using **?DO** rather than **DO** (why) and by using .R to provide fixed-width output.

Solution 4.4: Bytes of data are compiled with **C,** . Write a specification for this word, and show how you would use it when setting up a 4 byte table called **NAME** containing the ASCII codes for ``J'' ``O'' ``H'' and ``N''.

Suggest how the word ``comma'' is defined.

ALLOT **c –**

Allot 1 byte of space and store c in the location thus reserved.

```
CREATE NAME    \ -- addr
  CHAR J C,  CHAR O C,  CHAR H C,  CHAR N C,
```

Solution 4.5: Show how **FILL (addr len char --)** could be defined using **CMOVE**, not as a block move, but in the ``wrong direction''. Then use **FILL** to define the words **ERASE** and **BLANK**. The words **FILL**, **ERASE** and **BLANK** have all been described earlier.

```
: FILL   \ addr n byte --
  ROT             ( n byte addr )
  SWAP            ( n addr byte )
  OVER            ( n addr byte addr )
  C!              ( n addr )
  DUP 1 +         ( n addr addr+l )
  ROT 1 – CMOVE
;

: ERASE ( addr n -- )  0 FILL  ;
: BLANK ( addr n -- )  BL FILL  ;
```

Solution 4.6: Consider a temperature sensor connected via an A to D converter and giving readings that vary between 0 at 5 C and 255 at 160 C. Assuming the variation in the reading is linear with temperature define a word called **DEGREES** that will convert a temperature value in degrees C to the equivalent A to D converter reading.

```
: DEGREES      \ y -- x ; convert y to machine units
  5 -  255 155  */
;
```

Solution 4.7: Define a word **OF-TEMP-TABLE** with the following spec.

```
     OF-TEMP-TABLE    n -- t
```

t is the temp of element n in the table (0 <= n < 5).

```
: OF-TEMP-TABLE ( n -- t )
  CELLS TEMP-TABLE + @
;
```

Solution 4.8 READING->DEG

```
: READING->DEG ( n -- t )
\ convert a-d reading to scaled temperature
  600 -
  500 /MOD           ( r q )
  DUP 1+             ( r q q+1 )
  OF-TEMP-TABLE SWAP
  OF-TEMP-TABLE SWAP ( r t[q] t[q+1] )
  OVER -             ( r t[q] t[q+1]-t[q] )
  ROT  500  */       ( t[q] {t[q+1]-t[q]}*r/500 )
  +                  ( t[q]+{t[q+1]-t[q]}*r/500 )
;
```

Defining words

Solution 5.1: Define a word **+CONSTANT** which defines words which record a 16 bit value at compile time and add this word to the value at the top of the stack at run time. Example usage:

```
     5 +CONSTANT PLUS5
    10 +CONSTANT PLUS10
   100 PLUS5 PLUS10 . <cr> 115 ok

: +CONSTANT    \ n -- ; a -- a+n
  CREATE
    ,
  DOES>
    @ +
;
```

Solution 5.2: Complete the following definition of **BYTES**.

For our next example, we create a defining word **BYTES** which can be used to set up tables of 8 bit values. Example usage of **BYTES** is:

```
   10 BYTES BTABLE1
   20 BYTES BTABLE2
```

```
   77 0 BTABLE1 C! ( store 77 in byte 0 of BTABLE1 )
   40 1 BTABLE1 C! ( store 40 in byte 1 of BTABLE1 )
```

So **0 BTABLE1** returns the address of byte 0 in **BTABLE1**, **1 BTABLE1** returns the address of byte 1 in **BTABLE1**, etc.

```
: BYTES  \ +n -- ; addr –- addr+n
  CREATE  ALLOT  DOES> +
;
```

Solution 5.3: Provide definitions for **WITEMS** and **LITEMS** which can be used like **BYTES** but defines tables whose elements are 16 and 32 bits wide. For example **10 WITEMS TABLE1** would define an array of 10 16-bit items called **TABLE1**.

```
: WITEMS       \ +n -- ; addr –- addr+2n
  CREATE  2* ALLOT  DOES> SWAP 2* +
;

: LITEMS       \ +n -- ; addr –- addr+4n
  CREATE  ALLOT  DOES> SWAP 4 * +
;
```

Solution 5.4: Another use of **CREATE ... DOES>** is to define the opcode classes for an assembler. The simplest class of opcodes for an assembler to handle are those which have a fixed opcode. Here are some such opcodes for the 8086 instruction set

```
Opcode    Mnemonic

C3 RET
FA CLI
FB STI
9B WAIT
F0 LOCK
```

In a Forth assembler each of the mnemonics such as **RET** and **CLI** would be defined as a Forth word. When executed these words compile their associated 1 byte opcode into the next free byte of dictionary memory.

Suppose the word **CLASS1** is used to define the given instructions. Usage will be:

```
HEX
C3 CLASS1 RET
FA CLASS1 CLI
...
```

Define **CLASS1**.

```
: CLASS1
  CREATE
    C,
  DOES>
    C@ C,
;
```

Note. This definition would be for an assembler which assembled code into Forth's memory space. Modern Forth systems often assemble code into a different area of memory to conserve the Forth memory space. The definition of **CLASS1** would then be slightly different. The **C,** in the run time action would be replaced by a non standard word to compile the opcode into the next free byte in the code area.

Miscellaneous

Solution 6.1: Suppose the following definitions are loaded.

```
: TEST1 ." old version " ;

: TEST2 TEST1 ;

: TEST1 ." new version " ;
```

What will be the output generated by:

```
TEST1 TEST2

 TEST1 TEST2 <enter>  new version old version
```

21 Adopting and managing Forth

This question and response was posted on the comp.lang.forth newsgroup and expresses many of the concerns of potential users, plus some typical responses from users.

```
>The main thing I'm trying to get a better idea of here:
>In what cases should Forth be seriously considered?

My opinion:

1. Limited memory.
2. Lack of need for certain language-pervasive features
(garbage collection, strong typing).
3. Management willingness to train people rather than
expecting them to hit the ground running (this is actually a
fundamental software engineering principle, but it's
especially important to Forth if the reason discussed in #4 is
true in your case, and can be important due to the scarcity of
pre-trained Forth programmers).
4. The problem being solved has its own well-understood
language and terminology, so that Forth can be used to create
a computer language which resembles the problem's description
language.
5. The problem being solved is not well understood, but can be
explored interactively (for example, a hardware component for
which a controller is being written).
```

I saved this post as it raises so many issues. The rest of this chapter is an attempt to pick out some answers and expansions on the points raised here. Apart from discussing the benefits (and perhaps some limitations) of Forth, I also want to indicate how to sell Forth into an environment that is possibly resistant to change.

The essence of what makes Forth great is interactivity and extensibility in a small package. A particular difficulty in talking to C/C++ programmers who have not experienced true interactivity or extensibility is that the programmers don’t know that they might need interactivity or extensibility.

Interactivity and exploration

Embedded systems are often hard to debug without interactivity. In this environment, the ability to test “bottom up” makes a significant contribution to program reliability, eliminating the collection of hacks and kluges required at higher layers to overcome limitations at lower levels. A secondary benefit is smaller code.

Interactivity can replace a large collection of expensive debugging tools. We once were asked to find a problem in a bank note sorting machine. The company’s engineers had spent two weeks with in-circuit emulators and their non-interactive development system. We installed a simple Forth system in EPROM just using the serial line, CPU and memory and started investigating. We wrote a very simple loop that polled an I/O device until a key was pressed.

```
VARIABLE ADDRESS

: T            \ --
  begin  adress @ c@ drop  key? until
  key drop
;
```

Running this word and using a simple oscilloscope probe revealed that there was a small glitch in one of the address decoders. The in-circuit emulator previously used had slightly different timing which moved the problem elsewhere. A quick fix and the problem had gone. After fitting the Forth (one hour) the problem was fixed in 40 minutes.

Interactivity provides tools for rapid debugging and focussing on the correct issues. You have to experience it to believe it.

In a desktop PC environment, once you are in a place where a canned library (Windows DLL, Linux shared library) cannot fix your problem, you are faced with understanding the operating system API. Despite the best efforts of all concerned, API documentation is all-to-often incomplete and sometimes just wrong. Operating system bugs do exist. Interactive exploration in the PC environment is much faster for the same reasons as in the embedded environment. A modern Forth for Windows can provide a very rich debugging environment, not only for Forth programs, but also for debugging DLLs.

Exploration (unfortunately called “playing” by far too many engineers) is a vital part of producing better systems. In an unfamiliar world, searching for better techniques and algorithms pays big dividends. Interactivity provides tools to do this.

Extensibility and notation

In many of the jobs I’m involved with, I work with electronic and mechanical engineers and bomb disposal officers on embedded systems, or with construction engineers, video people or ambulance crews on desktop PCs. They all have their own jargon and terminology.

Forth’s ability to provide a language tuned to the way end-users think, plus it’s ability to provide rapid prototyping, enables me to use code to talk to them about what they really want, and then to provide the facilities they need in the final application.

The “spiral lifecycle” approach that is so common encourages this approach. We simply use extensibility and notation to improve the speed at which each turn of the spiral is performed.

Limited memory

Forth encourages the use of small definitions, if for no other reason than that having too many items on the stack makes a definition hard to understand and difficult to maintain. The Forth execution model means that additional words do not cost much at execution time, and modern Forth compiler technology can even eliminate that by inlining small definitions.

Small definitions are easy to reuse and each reuse saves memory. The embedded PowerNet TCP/IP stack is considered by one company to be at least half the size of any other unmodified commercial stack. So much so, that it will fit in the boot ROM of a custom CPU with an open Forth system.

This is not just a trait of small scale embedded systems. In the mobile phone arena, one company saved 30% of their ROM space by converting the games to a custom Forth.

If you are delivering code electronically for remote updates, the cost of delivery may be dominated by phone bills or on-line costs. This is an increasingly common requirement. It may not seem much at each upload, but when you are updating thousands of units, it is not uncommon to have to transfer several gigabytes to update all units. Not all units are connected by fixed lines or broadband connections. The tokenised Forth used by Europay's Open Terminal Architecture was estimated to pay for itself for only one update per year.

In the desktop environment, low memory footprint can have a surprising impact on performance. Despite the huge amount of memory that is commonly fitted to desktop PCs, first and second level cache is very restricted. Commonly, the difference between CPU and main memory speeds is about 10:1. On desktop PCs, the ratio can be 100:1. When a Forth application can fit entirely into cache, even an old-fashioned threaded Forth implementation can run faster than a large compiled application. This has been observed at Sun Microsystems. The tradeoffs between application performance, interpretation and memory/CPU bandwidths have been explored in some detail in "Interpretation and Instruction Path Coprocessing" by Eddy H. Debaere and Jan M. Van Campenhout, MIT Press, ISBN 0-262-04107-3.

Why not the common language?

The simplest answer to this question is that one size does not fit all situations. The arguments are not really about Forth, the reasons to use the common language are also used against Pascal, Modula-2, Smalltalk, Eiffel and even Delphi. These are often called minority languages, and managers tend to be scared of them. The issue of programmer training and management is discussed later. For the moment we will stick to more technical issues, particularly those affecting time and staff costs.

Languages from the C, C++ and Pascal/Delphi groups (sometimes called the Pasgol group) do not have interactivity or extensibility. The interactivity is mostly provided by invasive tools that upset performance, especially in real-time systems of any description. Debugging sealed remote applications is vastly more difficult than connecting to a running Forth application over a serial line or a Telnet session. Extensibility in the Pasgol languages is largely confined to compile-time use of the preprocessor. This contributes in turn to the use of additional external tools (Awk, Perl, M4 and so on) to provide facilities that can be integrated very easily into Forth.

An example of this was the need to provide a scripting tool for a predefined language. For many years we have used a derivative of Brad Rodriguez' excellent BNF parser implementation if Forth. A BNF parser is a tool that allows you to define the grammar of a language together with the code required to handle it. The base BNF parser is less than 100 lines of source code. This small size makes it perfectly feasible to embed a scripting system inside an application and then define custom languages for end-users. Doing this with a language that is neither interactive nor extensible requires the production of custom parsers and compilers.

In embedded systems, memory is expensive. If you are making 10,000 units per year, the cost of using more memory is significant. If you are making 100,000 or 1,000,000 units per year, the cost of additional memory may make the difference between commercial success and failure.

If you are making low volume systems which interact with a poorly defined or understood outside world, an interactive language can be a lifesaver. In one machine

control application, we rewrote the motor drivers three times in a few days as we increasingly understood what impact the working environment (water, mud and grit) had on the mechanics (air lines, motors, leadscrews and so on) and how to compensate for variability as the machine bedded in.

The essence of what makes Forth great is interactivity and extensibility in a small package. If your application can take advantage of or requires interactivity, you can see improvements in productivity of two or three times.

We used Forth 15 years ago, but ...

There are really two reasons for encountering this kind of objection:

1) We use C/C++ nowadays – it's the common language. This has already been discussed.

2) Someone at the company had a bad experience with a Forth project long, long ago.

Recovering a brand from a bad reputation is far harder than selling to a new client. If you have to go down this route be prepared for a difficult sell. The common opinions and assumptions we see as part of the second response are:

1) Our programmer wrote his own Forth. When he left we couldn't maintain the Forth or the code.

2) Forth is interpreted and slow.

3) The system we bought then was not good enough for today's requirements. Yours must be the same.

4) Forth people are all unmanageable.

5) You can't do X, Y, or Z in Forth.

6) You can't write and maintain large programs in Forth.

7) Nobody uses Forth any more.

Our programmer wrote his own Forth. Some 15 or 20 years ago people built their own hi-fi systems. The technical hobbies of today are very different from the technical hobbies of then. There are now many good Forths from commercial or open-source suppliers. Management must decide whether the benefits and associated costs of tool-making justify keeping tool-making in-house. In the vast majority of cases, buying in a compiler and perhaps getting a few days consultancy is the cheapest option. Documentation is a management issue, and these days people know a lot more about it than they did 15 years ago.

Forth is interpreted and slow. Modern Forths from commercial suppliers use aggressive code generators. Some are based on very similar techniques to those used by many C compilers and the benchmark results reflect this. The only way to persuade some people is to provide some numbers. Our benchmarks and some tests by users show that the current batch of Forth compilers are over 10 times faster than those based on threaded code.

The system we bought then was not good enough for today's requirements. The "bad then, bad now" argument is similar to a proof by assertion. The easiest way to overcome it is to ask them if they would use a 15 year-old C compiler. Part of the

problem is that some people do continue to use 15 year-old obsolete and unsupported C compilers.

Forth people are all unmanageable. We just have to face facts. Programmers (in any language) have a reputation of being difficult to manage. Managing guru level programmers is compared with herding cats. About ten years ago I heard the manager of a large Japanese company's research labs say:

"The problem with C++ is that it requires guru programmers: and guru programmers don't do maintenance."

I have worked on Forth projects with up to 30 programmers. Whether such projects work is a matter of management, a topic we will come to later. In practice, many managers are much better at software management than they used to be. You have to be aware that software management is a vital part of constructing software regardless of the language involved.

You can't do X, Y, or Z in Forth. It may be true. But it may also be ignorance. I come across this opinion far too often. When I show them that it is no longer true, the problem goes away. As with many other hostile opinions, the best way to deal with them is to show them the numbers or give references to successful projects that actually do what they say cannot be done.

You can't write and maintain large programs in Forth. Yet again, point to large Forth projects. The construction estimation package produced by Construction Computer Software plan huge construction projects such as the new Hong Kong airport. It contains over 850,000 lines of Forth source code. See http://www.ccssa.com for more details. How they do it is very much based on the techniques described in "The Mythical Man-Month" by Fred Brooks. Big projects need management. If you don't do it, they **will** fail.

Nobody uses Forth any more. Apart from people like IBM, Sun MicroSystems, Apple, and NASA. Side issues for this discussion are that programming languages are not the topic of discussion that they used to be and that some large and successful companies using Forth consider it to be a trade secret of their success. At least one of our clients sold their company and became very rich on the back of technologies developed with Forth.

Managing Forth projects

If you do not believe how important software project management is, read "The Mythical Man-Month" by Fred Brooks. There are also several books available about software disasters and what to learn from them. Until fairly recently, many companies treated software projects differently from those in other engineering disciplines. Managing a project written in Forth is no different from managing any other software project.

One of the considerable benefits of the surgical team approach promoted in "The Mythical Man-Month" is that the project manager can retain technical competence (and hence respect) of the programmers. The team administrator is the one who does the spreadsheets.

Managers

A split between "the management" and the engineering staff is never productive. It is part of the process for one to educate the other. The effective power is usually in the hands of the management because they control the budgets. Experienced managers have been scammed by technical staff many times in their careers and are rightly

cautious on occasion. What convinces them are cost and time numbers. Convincing them by assertion rarely works. Indicating what you have done and why it works may be effective.

In many departments there will be an opinion-maker in the engineering team. Such people are valuable allies, but it is often not obvious who they are. Making contact with them is important.

In leading or bleeding edge projects, ensuring the social cohesion of a programming team is very important. In one multinational project is was involved in, our (Irish) project officer remarked one evening that projects that started with a good party were more likely to succeed.

Programmers

When programming was the new technology of the time, very few people knew anything about it. We learned by doing and made a vast number of mistakes, some of which still live to haunt us. What characterised programmers then was an eagerness to do and learn something new. We had to be sociable because we had to learn from our peers. We also had to be capable of locking ourselves into a room with a computer for hours and hours – there may have been nobody to teach us what we needed to know. Managers should be aware of the dynamics of their team and select staff accordingly to produce a balanced team.

Many projects in the embedded arena were done by engineers who had learnt programming as an adjunct to their main job. Many of them took on more than they could chew, and management are now justly cautious of letting a non-specialist have their head.

Many current Forth programmers also come from an electronics or scientific background. Forth's good reputation in embedded systems is largely a result of how it grew, rather than in any truth about what it is good for.

Nowadays, programming for many people is just a job. It also covers a huge range of applications. Putting a Java programmer onto a hard real-time application happens, but it is a risk.

Like much other engineering and professional practice, software is as much a craft as anything else. Experience counts for a lot. Even more valuable is learning from that experience.

Training

When people start using Forth, there is a learning curve. Because Forth is a different type of language compared with the ones they have probably used before, the learning curve converting from C/C++ to Forth is larger than that in converting from C/C++ to Delphi. A typical Forth training course needs from three to five days, mostly depending on how good the programmers are and how much time they have available. The commercial vendors can provide standard or tailored courses.

After such a course, it will take a month or so to become comfortable in Forth. After that productivity rises. The important point of a course is to do enough coding in Forth so that they "get it". A programmer used to C who does not "get" Forth will probably write Forth in a C style and will show little or no productivity gains in the areas where Forth could otherwise show major gains. Exposure to other Forth programmers helps here. An interested programmer will learn much faster than someone who has just been sent on the course. Consequently, motivation on the benefits of Forth is an essential part of courses.

The costs of training are often raised as an issue, especially for newly hired programmers. Learning a programming language takes much less time than learning the business of a company. Over the lifetime of a project, the cost of training (like the cost of the software tools) is all but lost in the noise.

Portability

Since the introduction of ANS Forth, the portability of code has improved considerably. However, just as in any other language, porting code between different Forth systems requires attention to detail. After being involved with several large ports of up to 800,000 lines of Forth source code, the key issues are separation and management.

Separation involves deciding which code can be changed and which code cannot. During the early stages of the project you will want to run the new system alongside the old. This involves splitting the code into several layers.

1) **Application** – this code is left unchanged

2) **Tools** – you leave this code unchanged if at all possible, but depending on the underlying kernels, some change may be required. Careful use of conditional compilation helps here, but too much use of it leads to maintenance issues.

3) **Harnesses** – provide one for each Forth system you are using. The job of the harness layer is to translate from the underlying Forth kernel so that the tool and application layers require (ideally) no change. When you decide to abandon the earlier kernel, only then should you permit further changes to the tool and application layers.

4) **Forth kernel** – leave it alone! If you need it to be modified consult the supplier. It is a major problem if the kernel you use is not the one that your supplier is shipping. If you must have changes, make sure that the supplier is prepared to take over maintenance of the changes.

The management issues in porting are ensuring that you do one thing at a time and ensuring people do not get too enthusiastic. Making a major feature enhancement during a port is a recipe for disaster. Doing the port and then doing the feature enhancement will be quicker. Porting puts the team manager as a barrier between the differing objectives of the software development team and the sales and marketing group.

Lifecycle

Successful software products evolve, so change is a constant of their lives. Changing software means that someone has to back and revisit code written several years ago. Code must be documented to cope with this. An in-house style greatly eases the problems of code maintenance.

Be prepared for the issues involved in porting your code. Hardware platforms do not last for ever, and neither do operating systems. The product you are developing may have a commercial life well beyond that of the hardware and software platforms you are currently using.

Some of the products I am currently maintaining started life under CP/M, were ported to 16 bit DOS, 32 bit DOS, Windows 3.1 and then to the Windows NT based operating systems. The code involved has evolved over a period of nearly 25 years and is still working. It has outlived changes in CPU, operating system, and host Forth system.

22 Legacy issues

As with any programming language, Forth has evolved over time. This chapter discusses some of the legacy issues.

Forth Standards

Four Forth standards have been significant: FIG-Forth, Forth-79, Forth-83 and ANS Forth. ANS Forth is the current standard. FIG-Forth and Forth-79 are unlikely to be encountered, so this section deals with converting code from Forth-83 to ANS Forth. Because the ANS standard very deliberately avoids specifying implementation technique, there has been an explosion in Forth compiler technology since the introduction of the ANS standard in 1994. This section also discusses the impact of converting from a threaded code implementation to a modern optimising code generator such as MPE's VFX systems.

Native Code Compilers

Carnal knowledge is dangerous

Extra care should be exercised with any source code which requires knowledge of the underlying architecture. This will particularly impact definitions which cause compilation, and assembler fragments.

Comma does not Compile

Many Forth-83 implementations allowed compilation by "comma-ing" a CFA into the dictionary. This is no longer a valid method of generating code. The ANS word **COMPILE,** should be used instead. Also the system must be in "compile state" when **COMPILE,** is used.

Converting from Forth-83

MPEism: *COMPILE* ***is now*** *IMMEDIATE*

Previous MPE implementations have used a non-immediate version of ***COMPILE*** *which has "unpicked" the following CALL instruction at run-time. This behaviour has now been changed. Most uses of* ***COMPILE*** *and* ***[COMPILE]*** *can be replaced by* ***POSTPONE****.*

COLON and Current

Under ANS Forth, the **CONTEXT** definitions wordlist (the used when looking up word names) is **not** modified by **:** (colon). The impact of this change is that you do not need to add an extra **ALSO** to protect the search order. Also note that **:** (colon) is no longer immediate in most implementations, which may affect some compile-time error checking.

ANS Error Handling

Error handling is done using the ANS specified **CATCH** and **THROW** mechanism. The definitions **ERROR** and **?ERROR** from Forth-83 are now aliases for **THROW** and **?THROW**. Please read the section on exception handling both in this manual and the ANS Forth Standard.

Screen files

A screen (also called a block) is a unit of Forth source code that consists of 16 lines of 64 characters. The reasons for this unit are almost lost in the depths of computing history, but stem from using a size that fitted efficiently onto a drum store, and also fitted onto current screen sizes (most people used mechanical teletypes as terminals). This convention has been retained, partly by convention, and partly because it helps to enforce writing code in small chunks.

***MPEism**: The description in the remainder of this section is specific to MPE products.*

If you still need to maintain screen files, MPE's Forth Starter Kit includes PC PowerForth Plus and Modular Forth. These provide a specialised screen file editor called FRED, which is the editor you get by typing **EDIT** or **FRED**. You can see what is in a screen file by using a Forth convention. According to this convention the top line of every screen (a unit of source code) is used only as a comment line describing the contents of a screen.

The word **INDEX** can then be used to display all these comment lines. The phrase:

```
0 100 INDEX
```

will index the comment lines of screens 0 to 100.

You can then see what is in screen 23 by typing:

```
TOOLS     ( loads module TOOLS.MD3 )
23 LIST    ( lists screen 23 )
```

Notice how the word **TOOLS** is used to load the module TOOLS.MD3 that contains many handy utilities. The word **N** will show you the Next screen, and the word **P** will show you the Previous screen. Screens are organised as 16 lines of 64 characters, and are edited using the full screen editor. When you find a screen you want to compile (say screen 7), you can compile it using the word **LOAD**:

```
7 LOAD
```

Screen 7 will be compiled. If screen 7 contains words that load other screens, they too will be compiled.

Files

If a screen file was specified when Forth was loaded, it will be opened as Forth signs on. To change to another file, use:

```
USING NEW.SCR
```

If the file NEW.SCR does not exist the system will ask you if you want the file created, and if so, the file will be created and opened. The screen file is used by all the words affecting source code, and when you type **EDIT** or **FRED**, the editor is loaded and acts on the current screen file. After the word **EDIT** screen 0 is edited, or if the editor has been used before, the screen last edited is edited again.

```
 EDIT            ( edits screen 0 or last screen )
25 FRED   ( edits screen 25 )
```

23 Other Books and Resources

I learned Forth by using it, talking to other people and by reading everything about it that I could lay my hands on and afford. As with every new subject, I found that I needed more than one book, and often I needed different books at different stages in my learning. I cannot pretend that this book is the only one you will need, so here are some of them that I have found useful.

The first two books are by Leo Brodie and are classics.

Starting Forth – Leo Brodie

Sadly out of print, but if you find a copy, especially of the second edition, buy it. A web version by Marcel Hendrix is described as a tribute to this great book and is available at:

`http://home.iae.nl/users/mhx/`

Thinking Forth – Leo Brodie

A PDF of this wonderful book is available from:

`http://thinking-forth.sourceforge.net/`

The following is taken from the description there.

Thinking Forth is a book about the philosophy of problem solving and programming style, applied to the unique programming language Forth. Published first in 1984, it could be among the timeless classics of computer books, such as Fred Brooks' The Mythical Man-Month and Donald Knuth's The Art of Computer Programming.

Many software engineering principles discussed here have been rediscovered in eXtreme Programming, including (re)factoring, modularity, bottom-up and incremental design. Here you'll find all of those and more - such as the value of analysis and design - described in Leo Brodie's down-to-earth, humorous style, with illustrations, code examples, practical real life applications, illustrative cartoons, and interviews with Forth's inventor, Charles H. Moore as well as other Forth thinkers.

If you program in Forth, this is a must-read book. If you don't, the fundamental concepts are universal: Thinking Forth is meant for anyone interested in writing software to solve problems. The concepts go beyond Forth, but the simple beauty of Forth throws those concepts into stark relief.

So flip open the book, and read all about the philosophy of Forth, analysis, decomposition, problem solving, style and conventions, factoring, handling data, and minimizing control structures. But be prepared: you may not be able to put it down.

Forth Programmer's Handbook – Conklin & Rather

A second level book well regarded by many.

`http://www.forth.com/forth/fph.html`

Forth Application Techniques – Rather

"From first-day Forth exercises to advanced techniques many programmers never learn on their own, this course notebook is filled with pithy, succinct discussion and exercises developed and refined over the years to quickly teach, test, and reinforce Forth language skills."

http://www.forth.com/forth/fat.html

Other Resources

Forth Interest Group

The Forth Interest Group web site at:

http://www.forth.org

contains links to a huge amount of information about Forth. It also has a repository of compilers, source code and documentation which you may well find useful. You can also find links to local support groups in several countries.

Usenet news groups

```
comp.lang.forth
comp.lang.forth.mac
```

Conferences

Many of the smaller Forth conferences have been superseded by the news groups. The annual EuroForth conference is the best of the current conferences. It is linked at

http://www.forth.org

and is held in nice places around Europe. Several smaller conferences are held in Silicon Valley, Holland and Germany.

Amazon

The online bookseller has a searchable section under Programming:Languages:Forth

24 Index