tForth Manual

Documentation for the Programming Language of the Canon Cat

By the Staff of
Information Appliance, Inc.

Copyright Information

Canon Cat by Jef Raskin, Dr. James Winter, Terry Holmes, Minoru Taoyama, Jonathan Sand, John Bumgarner, Paul Baker, Jim Straus, Dave Boulton, Charlie Springer, Scott Kim, Ralph Voorhees, Richard Kraus, Kouji Fukunaga, Kazuhiro Nakamura, Naohisa Suzuki, Shigeru Ishida, Susumu Takase.

Manual by David Alzofon, Lori Chavez, Jim Winter, David Caulkins, Terry Holmes, Minoru Taoyama, Jonathan Sand, John Bumgarner, Scott Kim.

TABLE OF CONTENTS

HOW TO INTEGRATE SOFTWARE INTO THE CANON CAT i

INTRODUCTION/BACKGROUND 1
- Background of Forth 1
- Organization of This Manual 2
- Enabling Forth 3
- Talking to tForth 5
- A Brief Introduction to Forth 7

tFORTH PROGRAMMER USER MANUAL 11
- Moving Around in tForth -- Vocabularies 12
- The Parameter Stack 19
- Stack Notation 27
- Integers and Memory Operators 29
- Program Control Structures 35
- Character and String I/O 41
- Numeric I/O and Number Formatting 46
- Local Variables 53

tFORTH TECHNICAL REFERENCE MANUAL 55
- System Memory Usage 56
- The tForth Dictionary Structure 61
- Vocabularies 69
- Running tForth 73
- The Basics of tForth Compilation 77
- Execution of Token Threaded Code 85
- Implementation of Integers 96
- Implementation of Local Variables 104
- Implementation of Program Control Structures 115

tFORTH 68000 ASSEMBLER 142
- A Brief Overview of the 68000 Microprocessor 143
- Using the tForth Assembler 146
- Specifying Assembler Operands 148
- Structured Assembly Language Programming Support 153
- The movem Instruction 158
- tForth Assembler Words 159

GLOSSARY (tFORTH KERNEL WORDS ARRANGED BY FUNCTION) 160
- Arithmetic Operators 160
- Logic Operators 163
- Comparison Operators 165
- Stack Manipulation Operators 166
- Integer and Local Variable Words 169
- Memory Operators 171
- Program Control Structures 173
- Character I/O Words 179
- Numeric I/O Words 183
- Defining Words 185
- Dictionary Management Words 186
- Compilation Words 189
- Disk I/O Words (High Level) 195
- Disk I/O Words (Low Level) 197

GLOSSARY (CONT.)

CRT Words	199
Sound Generator Words	200
Keyboard Words	201
Modem and Serial I/O Words (High Level)	204
Modem and Serial I/O Words (Low Level)	206
tForth System Integers	208
tFORTH GLOSSARY (Alphabetical Listing)	212
tFORTH SYSTEM INTEGERS (Alphabetical Listing)	241
APPENDIX: DEFINING WORDS	245

ILLUSTRATIONS

USER MANUAL

tForth Vocabularies 13
The tForth Parameter Stack 20
Manipulating the Parameter Stack 24
Number Formatting 48

TECHNICAL MANUAL

System Memory Map 57
tForth RAM Memory Map 59
Structure of a Dictionary Header 62
Close-up of the Dictionary Header Area 64
Structure of the Length Byte 66
Close-up of the Dictionary Code Area 68
Opening and Closing Vocabularies 71
Dictionary Entries: Address Threaded
Versus Token Threaded 78
Token Table 80
Token Threaded Execution 87
Integers Execution 98
System Integer Table 101
Local Variable Return Stack Usage 105
A 'begin...until' Loop 116
A 'begin...while...again' Loop 119
An 'if...else...then' Loop 121
A 'do...loop' 124
A 'do...while...loop' 126
A 'do...if...leave...then...loop' 129

ASSEMBLER MANUAL

68000 Execution Environment 144
68000 Instruction Set Table 145
An Example of a tForth Assembly Language Word Listing 146
tForth Assembler Register Symbols Table 148
68000 Address Mode Categories Table 150-51
Condition Code Symbol Table 153
Status Register 154
tForth Assembler Words Table 159

HOW TO INTEGRATE SOFTWARE INTO THE CANON CAT

Products designed by Information Appliance Inc. (IAI), such as the Canon Cat, have a number of unique features. One of them that directly affects third-party software development is the principle of editor-based software.

In most microprocessor-based products, the user shifts between applications by returning to the operating system, indicated by a menu with a number of choices (or, equivalently, a window with a number of icons.) Then the user chooses the next application. Once having entered the application, the user gets the data on which to work.

In an IAI interface, the data stays in place at all times so that the user can concentrate on content rather than on the system. As commands are given, different "applications" come to bear on the user's text or graphics (there are graphics primitives in the Cat, although the built-in software does not use them). This is possible due to our unified data structure which is -- all at the same time -- a text, a data base, a spreadsheet, and a programming environment.

The user has a much simpler mental model with the IAI interface than with traditional products, since invoking an application looks just like another simple editor command. The user does not have to work with a number of different editors, one for each application. This is an improvement over the Macintosh, for example, in that with the Macintosh model each application must recreate (using provided routines) an interface that is similar to that of other applications.

When developing new applications for the Cat, it is easiest, both on the programmer and the user, to make your application look just like the existing built-in software. When your application needs to get information from the user, it generally asks a question. This can be done by sending the question to the screen, perhaps surrounded by a few blank lines so that it is visible. If the user finds that the question has come out in an awkward place (say, in the middle of a letter), then the user can always delete the question or move it elsewhere.

A typical question for an accounting package might be:

Name of account?

When this appears, the application should wait for a response to be sent to it by the ANSWER command (USE FRONT-ERASE). Thus the user is free to employ any and all the features of the Cat in creating the answer, for example, they might leap to their account area, or even change disks or perform a calculation to find the information they need. The idea here is to leave the full power of the Cat available at all times.

When the user has formulated the answer to the question your application has asked, they highlight it and use the ANSWER command. At this point, your application is in control again and can do what it wishes until it asks its next question.

This "loss of control" after a question has been asked will disturb some designers who are used to a forcefully directed dialog with the user. However, research has shown that users work better if they can do tasks at their own speed, and if they are in control. There is nothing more annoying than a program that demands an answer and won't let you use the system (say for looking up a phone number you need **right now**) until you are finished answering the computer's question -- a task that might take a few minutes if you have to look up something that's in a file cabinet somewhere.

One secret of the Cat's utility is that all abilities are available simultaneously and instantaneously. If your application has a number of features or areas, then allow the user to create a message which activates them when desired (the messages sent to your application via the ANSWER command, of course. One set of messages might be: "AR" to activate the accounts receivable package, "AP" to activate the accounts payable package, and "GL" to run the general ledger package. Once in any of these packages, the dialog would work as already described.

Notice that you do not have to write any I/O editing routines. You can simply send strings to the screen, and receive strings (edited by the user). Naturally, your application may need to do error checking, but when an error is detected, you can just send a string to the screen with the message, the user can edit their previous response using the Cat's built-in editor, and resend it to your application.

Following this protocol will keep the Cat feeling like a Cat, and will be least disruptive to a user's habits. It is also very easy and quick to create application interfaces this way.

Jef Raskin
13 September 1988

INTRODUCTION/BACKGROUND

BACKGROUND OF FORTH

The Forth language was developed in the early 1970's by Charles "Chuck" Moore. It was designed for control applications in an astronomical laboratory environment. Forth's interactive nature and its extremely small "kernel" of basic words (the Forth kernel typically requires only 2-5K bytes of memory) made it ideal for machine control using the very limited minicomputers of the time.

tFORTH

The intent of this manual is to describe the "tForth" ("token-threaded Forth") implementation of Forth designed specifically for use in the Information Appliance Inc. Cat project. The basics of Forth and Forth programming are not covered in a comprehensive manner. Starting Forth includes very good explanations of basic Forth programming and good descriptions of the inner structure of a simple Forth.

ORGANIZATION OF THIS MANUAL

This manual is organized as follows:

Introduction:	A very brief, general overview of the Forth language. This section tries to give the reader a feel for the Forth language by presenting examples and discussion of interactive and compiled execution of Forth words and parameter stack usage.
tForth Programmer User Manual	How to program in tForth. Examples are used to demonstrate how common programming tasks (arithmetic, memory access, character and numeric I/O, control structures, constants, variables, etc.) are performed in tForth. This section will give the reader a quick introduction to the use and power of tForth.
tForth Technical Reference Manual	Implementation-specific information required by those who intend to change or extend the tForth system. Topics covered include system memory usage, the vocabulary and dictionary structure, compilation and token-threading specifics.
tForth 68000 Assembler	How to use the tForth 68000 Assembler.
Glossaries	Stack notation and short descriptions of the words included on the tForth source disks. The words are grouped according to function (there is a list of functional groups at the start of the Glossary). The words are arranged alphabetically within in each group.
Appendices	Program listings.

ENABLING FORTH IN THE CAT

Forth is normally hidden away, inaccessible in the Cat. However, with a simple incantation you can "enable Forth," making it possible to switch from the Cat's editor to a Forth programming environment, or to run Forth programs from the Cat's editor with the ANSWER command. Forth enablement is associated with a given disk and text. If you enable Forth, record the text, change to a non-enabled disk, then Forth will no longer be enabled.

Remember to exercise caution whenever Forth has been enabled. For example, a nonprogrammer may be trapped in Forth if they accidentally press the key combination SHIFT-USE FRONT-SPACE BAR while editing the text on a Forth-enabled disk. The key combination USE FRONT-SEMI-COLON will erase the disk in the drive if Forth is enabled. Other pitfalls exist. SO, PROCEED WITH CAUTION IF YOU ENABLE FORTH. READ THE DISCLAIMER AT THE BEGINNING OF THE MANUAL.

How to Turn on Forth

We will now explain how to turn on Forth, and, equally important, how to turn it off:

1. To turn on Forth in a Cat, type the following phrase (be sure to capitalize "E", "F", and "L"):

   Enable Forth Language

2. Highlight these three words.

3. Hold down the USE FRONT key and, while holding it, tap the ANSWER key (ERASE). Then let go. This executes the ANSWER command, enabling Forth. You are not yet in Forth.

4. Now hold down the USE FRONT key AND the SHIFT key, and, while holding BOTH keys, tap the SPACE BAR. You are now in the Cat's Forth editor.

5. Type the following and press the RETURN key (the letters will automatically appear in boldface):

   -1 wheel! savesetup re

   This step allows you to enter Forth simply by pressing SHIFT-USE FRONT-SPACE BAR from now on.

   To enable easy access to Forth with Step 4 only, make some change to a Setup parameter, then use the DISK command. This will save the Forth enabling information on the disk. Whenever you play back this disk, you can then enter Forth using only the procedure of Step 4.

6. To turn off Forth, type the following and press RETURN key:

   Forth? off 0 wheel! re

   Make some change to a Setup parameter, then use the DISK command. This restores the Cat to normal operation, meaning that you will have to start over at step 1 again to invoke Forth. Normal Cat users will not be trapped in Forth in case they happen to accidentally press SHIFT-USE FRONT-SPACE BAR.

TALKING TO tFORTH

tForth is hiding in the background of every Cat system. It is very easy and convenient to communicate with tForth from within the editing environment.

Sending Commands to tForth

Once Forth has been enabled (see the previous page), commands and programs can be sent to tForth from the editor by highlighting the desired command string or program listing and pressing [ERASE] while holding the [USE FRONT] key down. tForth's responses will be printed out in the editor.

All examples in this manual are expected to be typed into the editor and "sent" to tForth in this manner. All examples presented are set off from the body of the text by two blank lines and are indented:

```
3 dup . . 3 3
```

A section of the above example was underlined. In an example, the underlined sections are the sections of the text which should be highlighted and passed to tForth by pressing the [USE FRONT][RETURN] key combination. After the above example was sent to tForth, tForth responded by printing two 3's on the screen.

Using the Calc Command to Talk to tForth

Commands and programs can also be sent to tForth with the use of the [USE FRONT] [CALC] key combination. When this method is used, all command strings or program listings sent to tForth must be preceded by a "]" character:

```
]3 dup . . 3 3
```

The above example produced the same results as the [USE FRONT] [RETURN] example. The [USE FRONT][CALC] method is not used in this manual.

Errors

The [USE FRONT][RETURN] is used to let Forth know it should start 'processing' any highlighted words. If Forth ever has a problem processing an input, a beep will be issued. To see the error message press the [EXPLAIN] key while holding the [USE FRONT] key down. For example, if tForth is sent the following input:

```
How now brown cow?
```

it will beep and [USE FRONT][EXPLAIN] will reveal a "can't use" message. This is the error message which occurs when tForth is sent a command it does not recognize.

CAUTION: ALWAYS RECORD YOUR EDITOR TEXT ON DISK BEFORE DIRECT EXECUTION OF tFORTH WORDS. IT IS VERY EASY TO MAKE PROGRAMMING MISTAKES WHICH COULD PERMANENTLY DAMAGE THE DOCUMENT.

A BRIEF INTRODUCTION TO FORTH

The Forth language is comprised of many "words" (commands). This collection of words is referred to as the "Forth dictionary." The tForth dictionary contains approximately 600 words. The list below shows a few Forth words and the actions they perform:

emit	Takes a number and displays the corresponding ASCII character on the screen.
+	Adds two numbers together and returns the result.
words	Produces a listing of all available words.
if then	Words used to implement the IF...THEN program control construct.
@	Fetches a 32-bit value from memory.

As the list shows, a Forth word can either have the format of a 'normal' word (a sequence of letters), or it can be a punctuation mark, a sequence of punctuation marks, or a mixture of punctuation marks and characters. In a Forth program, all words must be separated from each other by at least one space, tab, or carriage return. In this document Forth commands will be shown in boldface. For example:

"The Forth word **words** will produce a listing of all available words."

Note: tForth is case-sensitive. This means that tForth thinks a capital W is different than a lowercase w. Thus tForth will think **Words** is a different command than **words**.

If the pronunciation of a Forth word is unclear, it's first usage in the text will be followed by the natural language pronunciation enclosed in quotes and parentheses. For example:

> To take a number off of the parameter stack and display it, use the word . ("dot").

Executing a Forth Word

Most of the words in the Forth dictionary may be executed directly and immediately, from the keyboard. The example below shows how the Forth word **emit** could be used to display an asterisk character on the screen. In the example, the underlined type is used to indicate which commands should be highlighted and sent to tForth. The normal type is used to show Forth's responses.

Note: Do not confuse the underlined commands in the examples with the underlined Forth words in the text. In the examples the underlined commands are those commands which should be highlighted and sent to tForth with the ANSWER command.

```
42 emit *
```

emit , as was described above, is a Forth word which will display the character which corresponds to the ASCII value passed to it.

Compiling Forth Words

The interactive execution of **emit** in the previous example did not cause any code to compiled. The Forth word : ("colon") is used to turn the Forth compiler on:

```
: printstar 42 emit ;
```

The above example shows how a new word may be added to the Forth dictionary. The word which immediately follows : (**printstar** in the above example) is the name which will be assigned to the new word. The Forth words following the name and preceding the ; will be compiled into the new definition; these are the words which define the actions of the new word. Since the action words for **printstar** are **42 emit**, **printstar** will print an asterisk when executed. The word ; ("semi-colon") is used to turn the compiler off and return to the interactive execution mode.

Note that in this example, sending the input to Forth did not cause the asterisk to be displayed. Since the Forth compiler was "on" when the "**42 emit**" was typed, the **42 emit** was compiled rather than executed. Forth was able to successfully compile the new definition so no error beep was issued. Forth is an "incremental compiler"; code is compiled definition by definition; compilation is triggered by each reception of a line of input.

The Forth Parameter Stack

Forth is a stack-based language. Any Forth word which takes an input will expect to find its input parameter on the Forth parameter stack when it executes. Any Forth word which returns a value will leave the value on the parameter stack when it completes execution.

The parameter stack, and stacks in general, are functionally similar to the spring-loaded stack of plates which can be found at most institutional kitchens. Whenever a plate is taken from the stack, it is always taken from the top of the stack of plates. Whenever a plate is added to the stack, it is always added to the top of the stack of plates. A person who does not want the steaming hot plate on top of the stack must remove the top plate before the second plate can be accessed. If no plates are available, the stack is empty.

The Forth parameter stack works the same way as the stack of plates, except the Forth parameter stack is set up to hold numeric values rather than plates. Also, just as the kitchen stack was designed for a certain plate size, the Forth parameter stack is designed for a certain numeric value size (the plate size of the tForth parameter stack will be discussed later).

Interacting With the Parameter Stack

To put a number on the parameter stack, send the number to Forth:

```
34
```

To take a number off the parameter stack, use the word **drop**. To take a number off the parameter stack and display it, use the word . ("dot"):

```
. 34
```

To place more than one number at a time on the stack, send the numbers, separated from each other by a space or spaces (so that Forth knows they are distinct numbers), to Forth:

```
3 6 8
```

Now there are three numbers on the stack. If . is used, it will take the top number off the stack and display it. Since the 8 was the last value placed on the stack, it will be the top value on the stack:

```
. 8
```

To place more than one number on the stack at a time, the numbers were separated by spaces and sent to Forth. This is the same way Forth commands (words) work. To take both of the remaining numbers off the stack, the word . can be used twice on the same line:

```
. . 6 3
```

Forth's response should be read left to right. The 6 is the result of the first use of . The 3 is the result of the second use of .

Note what happens if . is used again:

```
. 0
```

You should hear a beep as . tried to remove a value from an empty stack and Forth responded by displaying a zero, beeping and issuing a "stack is empty" error message.

Passing Parameters to Forth Words on the Stack

Many Forth words take input parameters from the stack and return results on the stack. The Forth word + ("plus") is a good example of such a word:

```
5 4 + . 9
```

+ takes two numbers from the stack (the 5 and the 4 in the above example), adds them together and returns the single number result on the stack. In the example, . was use to display the result returned by +

Summary

* Forth programs are developed by creating new words out of previously existing words.

* The parameter stack is the primary means of communication among Forth words.

* The Forth language does not have many syntax requirements. This gives the experienced programmer great control over the computer but can make it difficult for beginning programmers to locate mistakes.

* The interactive abilities of Forth make it a hard-to-beat debugging environment. Each word can be tested individually and interactively.

This is the end of our brief introduction to the Forth language. For more introductory Forth reading, refer to the first chapter of Starting Forth, by Leo Brodie (Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, 1981).

INTRODUCTION

Now it's time to actually try some tForth programming. tForth contains words for performing many types of programming tasks. The available tForth words may be grouped into 19 functional categories: arithmetic words, stack manipulation words, character I/O words, numeric I/O words, structured programming words, etc. A complete list of these categories is shown at the start of the tForth glossary section, and the words in the glossary are grouped according to these functions.

This section of the manual will concentrate on describing how a few words from the most important functional categories are used. The categories covered will be:

* Vocabularies
* Stack Operators
* Integers
* Program Control Structure Words
* Character I/O Words
* Numeric I/O Words
* Local Variable Words

MOVING AROUND IN tFORTH -- VOCABULARIES

Before tForth programming can commence, the 'layout' of the tForth dictionary should be explained from a user's point of view. The words in the tForth dictionary are arranged into four groups of words called 'vocabularies'. The names of the four initial tForth vocabularies are **forth** , **user** , **function** , and **arithmetic** . The diagram on the following page demonstrates the relationships between the four initial tForth vocabularies. The **forth** vocabulary is the main or 'root' vocabulary. The three other vocabularies branch out from **forth** (the new vocabulary should be ignored for now), i.e., **forth** is the parent vocabulary of **user** .

existing is a Forth word which will print out the names of all existing vocabularies, the names of their parent vocabularies, and a count of how many vocabularies may still be added to the system:

```
existing
function (in forth)  arithmetic (in forth)  user (in forth)
forth (in forth)   12 free
```

Note: There is also a fifth initial vocabulary which is invisible to the user and is named, appropriately, **hidden** . The **hidden** vocabulary contains the words used to implement the Cat editor.

The Vocabulary Search Order

In order for tForth to compile or execute a word, it must be able to find the word in the tForth dictionary. The programmer helps tForth find words by setting up a 'vocabulary search order'. The vocabulary search order is a list which tells tForth which vocabularies it should search through and in which order the vocabularies should be searched. The word **searched** displays the current search order:

```
searched  user  forth  arithmetic  function
```

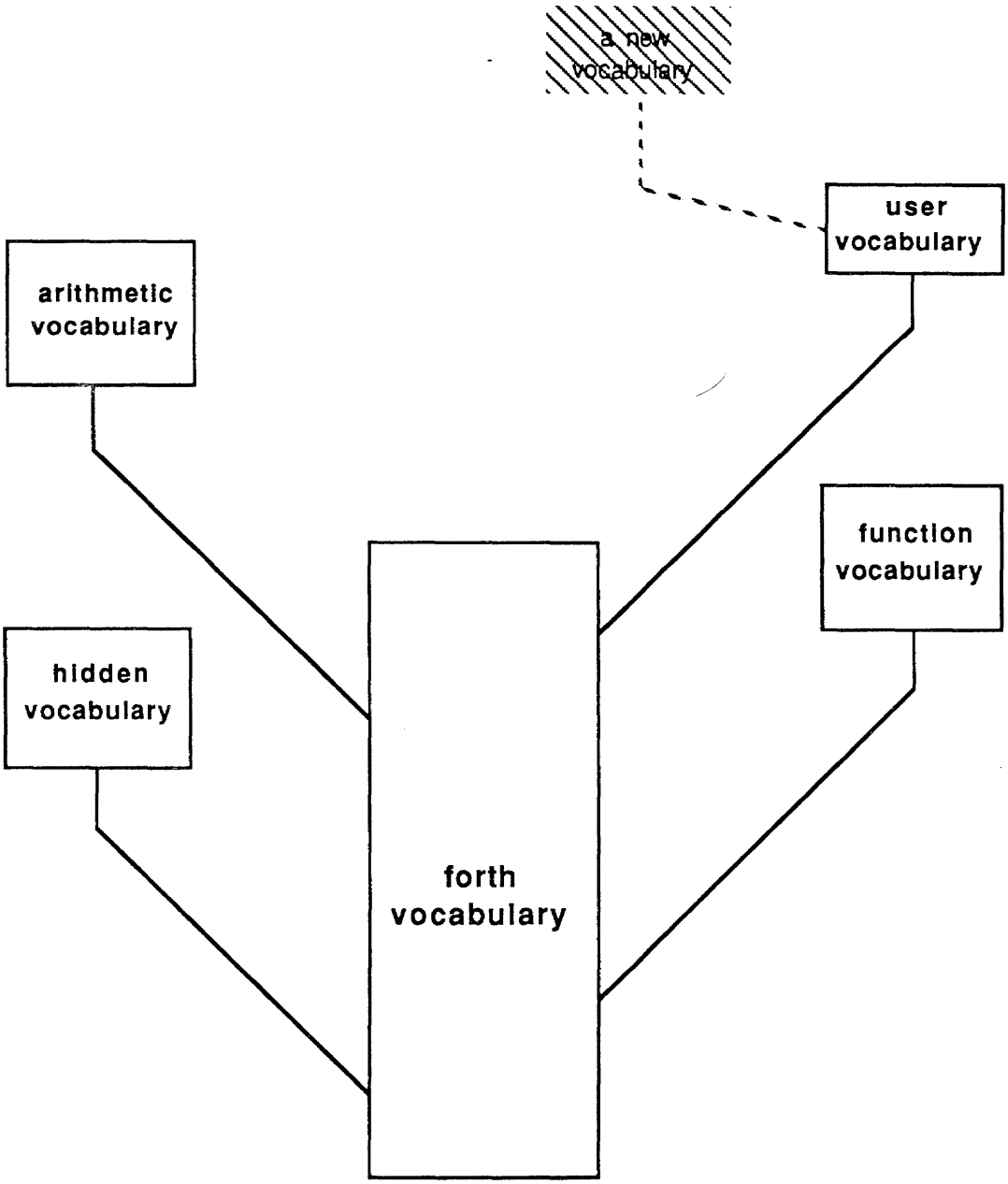
The names indicate which vocabularies are being searched, and the order of the names, read from left to right, indicates the order in which the vocabularies are searched.

Modifying the Search Order

A vocabulary may be added to the front of the search order by executing its name. If the vocabulary was already in the search order, executing its name will place it first in the search order. For example, to have the **forth** vocabulary searched first:

```
forth
searched  forth  user  arithmetic  function
```

The tForth Vocabularies



To remove a vocabulary from the search order use the word **deactivate**

```
deactivate arithmetic
searched forth user function
```

Listing the Words in a Vocabulary

The tForth command **words** can be used to print a listing of all words in the first vocabulary in the search order:

```
forth

words
! !char !csp !ptr " "to # ##ichrs ##ind ##wide ##lt
##sp ##tabs #wd #> #ab #be #chars #chrs #cmptabs
#count #ctrl #formats #guard #ichrs #indent #iwide
#learns #left #limit #line #lmar #lnloc #long #nextwrap
<cr> ok
```

The word **forth** was used to place the **forth** vocabulary first in the search order.

The **words** listing may be terminated by pressing any key. In the example, the carriage return key was used to prematurely terminate the listing. Since most of tForth's 600 words are located in the **forth** vocabulary a complete listing of all words in the vocabulary would be dull reading (masochists, however, are encouraged to display the complete listing at their terminals).

The words in a tForth vocabulary are arranged alphabetically. This allows tForth to locate words in the vocabulary with a very quick binary search algorithm. In most Forths the words are arranged in a chronologically ordered linked list. New words are added to the beginning of the vocabulary list. Locating a word in linked list requires that the list of words be searched linearly, starting from the newest word and progressing through the list to the oldest added word.

If you have been reading Starting Forth you should recognize a few of the words (! and #>) in the listing above. The unfamiliar words are additional Forth words which were not described in Starting Forth.

Adding New Words to a Vocabulary

New words may only be added to the current "open" vocabulary. Only one vocabulary may be open at a time. The word **addto** is used to open a vocabulary so that words may be added to the vocabulary. When addto is used, it is followed by the name of the desired vocabulary; in each case, **here** points to the next available byte of dictionary space (see page 70 for more on **addto**).

Note: The **user** vocabulary is the vocabulary to which all new "user-defined" words should be added. The **function** and **arithmetic** vocabularies are used by the editor so they should not be altered. The words in the **forth** vocabulary are located in EPROM so it is not possible to add words to the **forth** vocabulary.

The example below shows how a new word is added to the **user** vocabulary. The phrase '**addto user**' opens the **user** vocabulary so that new words may be added. The previous open vocabulary is closed.

printchar is a word which performs the same functions as the **printstar** word defined in an example in the introduction.

To reiterate the earlier description of compilation, the Forth word : turns the Forth compiler on. The word **]which** immediately follows : will be the name for the new word. The characters between the left and right parens form a comment string (Forth commenting style will be discussed later). All other words between the definition name and the final semicolon are compiled into the new definition. When the definition is later executed, these compiled words will be run.

```
addto user

: printchar ( -> | Prints a character. ) 42 emit ;
```

Now that **printchar** has been compiled, **words** can be used to ensure that **printchar** was really added to the **user** vocabulary:

```
user            ( Make user the current vocabulary )
                ( and then use words to list the words )
words           ( in the current vocabulary. )
printchar
```

The word **printchar** is the only word in the **user** vocabulary. (Note: If you have been experimenting with your system, your **user** vocabulary may have additional words). The new word may be executed interactively by typing its name followed by a carriage return:

```
printchar *
```

A Short Program

The program below is taken from page 13 of Starting Forth. It prints a large letter "F" using asterisk characters.

The word **decimal** tells Forth that all numbers input from this point on are to be treated as decimal numbers. **addto user** opens the **user** vocabulary. The program is comprised of the five definitions above, and the **printchar** definition which was compiled earlier. The program was developed by first writing the three lowest level words: **printchar** , **printchars** , and **margin.** Next, two intermediate words, **blip** and **bar** , which use the three

lower level words, are defined. Finally, the highest level word, **F** , which uses the two intermediate words, is defined. Since Forth programs continually build upon themselves, the order in which words are defined is extremely important. A word cannot be used in a new definition unless the word has been previously defined.

The program is run be executing the highest level word **F** (use uppercase "F"!).

```
addto user
decimal

: printchars ( -> n | Prints n asterisks. )
      0 do printchar loop ;

: margin ( -> | Prints a carriage return and 30 spaces. )
      cr 30 spaces ;

: blip ( -> | Prints 30 spaces followed by an asterisk )
      margin printchar ;

: bar ( -> | Prints 30 spaces followed by 5 stars. )
      margin 5 printchars ;

: F ( -> | Prints a large letter 'F'. )
      bar blip bar blip blip cr ;

F
                    *****
                    *
                    *****
                    *
                    *
```

Redefining a Word (Changing the Actions of a Word)

After a word has been defined, the action of the word can be altered by 'redefining' the word, i.e., entering a new colon definition which has the same name as the word to be replaced. For example, to change the action of **printchar**:

```
: printchar ( -> | Prints an '@'. )
      64 emit ; redefining printchar
```

Whenever tForth compiles a new definition, it looks at the names of all other words in the open vocabulary to see if a word with the same name already exists. If a word with the same name does exist, the compiler knows that the word is being redefined. Instead of creating a new entry in the vocabulary for the word, the compiler will replace the old actions of the word with the new actions. The message **redefining <name>** will be issued whenever a word is redefined.

This new version of **printchar** will print a '@' instead of a '*' when executed. All other words which referenced **printchar** will also be affected by this change:

```
F
                @@@@@
                @
                @@@@@
                @
                @
```

How Words Are Redefined in Other Forths

In most Forth's the redefinition of a word causes a complete new entry to be added to the dictionary. Because the vocabulary list is searched from newest entry to oldest entry, the redefined version of the word will be found before any previous versions of the word in all future dictionary searches. For example, if the word **printchar** had been redefined in most other Forths, any word defined later which referenced **printchar** would always use the redefined version of **printchar**. However, any words defined BEFORE **printchar** was redefined would ALWAYS reference the original, obsolete version of **printchar**.

In tForth, programmers can alter the actions of definitions without leaving unused, obsolete code in the dictionary. Every word in a program will always reference the most up-to-date versions of other words in the program.

Purging a Word From the Dictionary

The tForth word **purge** can be used to remove any word, regardless of vocabulary, from the dictionary (remember that the words in the **forth** vocabulary cannot be altered because they are in EPROM). The example demonstrates how **printchar** could be removed:

```
purge  printchar
```

What about the words which referenced **printchar**? Let's execute **F** to see what happens:

```
F
                (X)
```

Your cursor should have stopped at the point marked by the '(X)' above (you shouldn't see the '(X)' though) and a beep and the error message "unassigned token" should have been issued.

The first word run when **F** was executed above was **bar** (refer to the program listing). The first word in **bar** was **margin** . **margin** did not reference **printchar** so it executed without error and printed a carriage return and 30 spaces. The next word in **bar** was **printchars** , a word which did reference **printchar** . As soon as **printchars** tried to execute **printchar** , tForth displayed an error message which indicated that it could not find the word

it was supposed to execute next. This situation can be remedied by defining a new word named **printchar** :

```
: printchar ( -> | Print a character. )
    70 emit ;
```

This time the **redefining printchar** message was not issued because there was no word named **printchar** in the vocabulary at the time the definition was compiled. Now **F** can be successfully executed:

```
F
                FFFFF
                F
                FFFFF
                F
                F
```

Creating New Vocabularies

The word **vocabulary** is used to create new, named vocabularies. The new vocabulary will be empty and inactive (closed). The parent vocabulary for the new vocabulary will be the vocabulary which was open when the new vocabulary was created:

```
addto user
vocabulary testvocab

existing testvocab (in user) function (in forth)
arithmetic (in forth) user (in forth) forth (in forth)   11 free

testvocab words
```

In the example, a new vocabulary named **testvocab** was created. Since the **user** vocabulary was open when **testvocab** was created, the **user** vocabulary is the parent vocabulary of **testvocab** . This relationship was verified above by using **existing** to print a listing of all of the vocabularies and their parent vocabularies.

Next, **words** was used to verify that **testvocab** was empty when it was created.

THE PARAMETER STACK

In Forth, the programmer places the parameters on the stack and then executes the word. The word is responsible for taking the parameters it requires from the stack. If the word returns any parameters, it will leave them on the parameter stack. The programmer is responsible for removing any parameters returned from the stack. For example, consider the addition of 3 and 4, and the display of the sum:

```
3 4 + .  7
```

First, the parameters 3 and 4 are placed on the stack by entering the numbers separated by spaces. Next, the addition command + ('plus') is executed. + takes the 3 and the 4 off the stack, adds them together, and leaves the result (7) on the stack. The word . takes the result off the stack and displays it.

Structure of the Parameter Stack

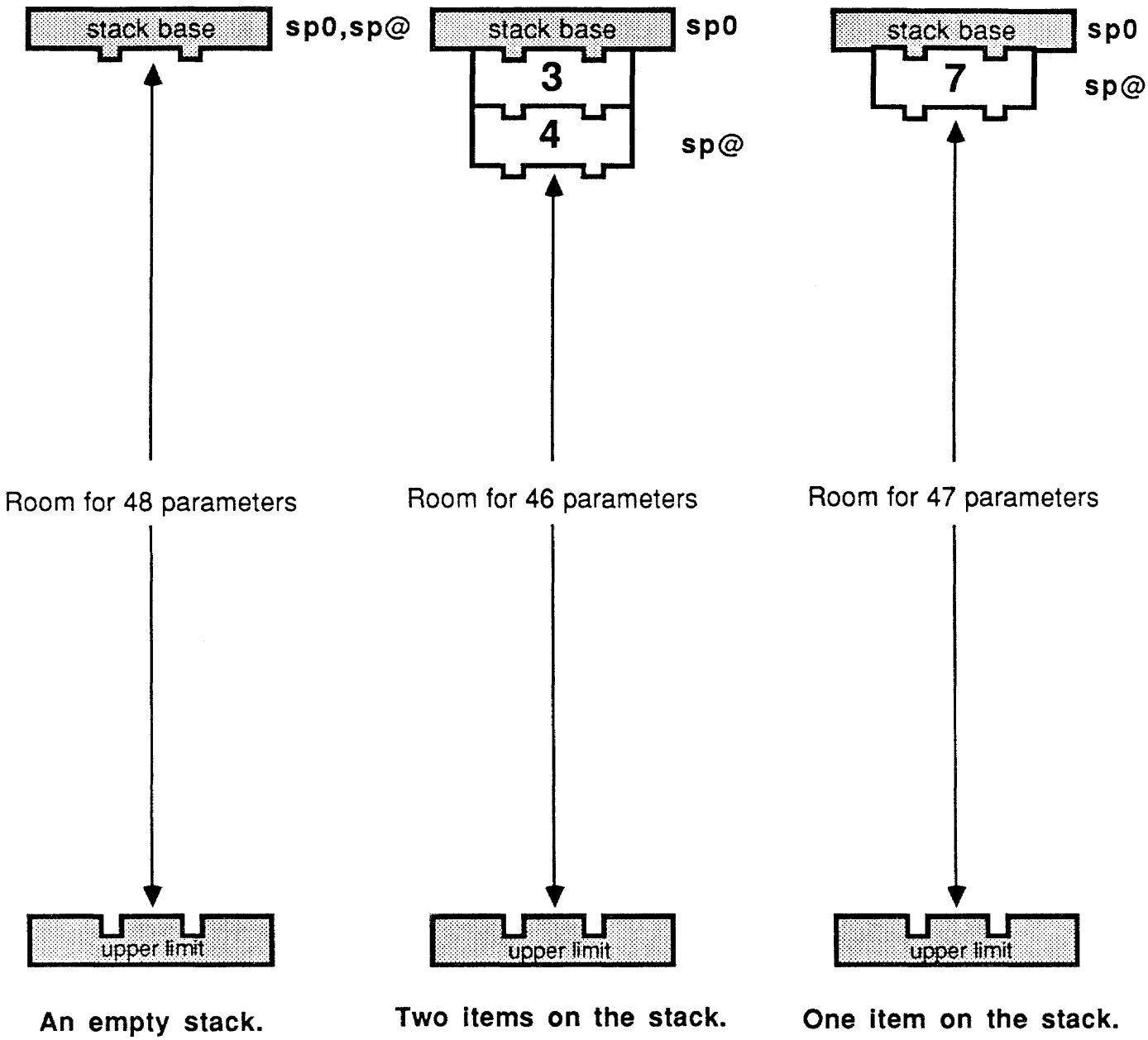
The diagram on the following page uses interlocking blocks to depict the functioning of the tForth parameter stack. The tForth parameter stack can hold up to 48 parameters. During execution of a program, only 5-10 parameters are typically on the stack at one time.

The parameter stack grows downward in memory. The word **sp0** ('s-p-zero') returns the address of the base of the parameter stack. The word **sp@** ('s-p-fetch') returns the address of the top item on the stack. Each parameter placed on the stack is placed in successively lower memory locations.

```
sp0 . 286748    ( The stack is empty so both )
sp@ . 286748    ( sp0 and sp@ return the )
                ( same address, the address )
                ( of the base of the stack. )

3               ( Place one item on stack. )

sp@ . 286744    ( Now sp@ points at the top item )
                ( on the stack, which is located )
                ( 4 bytes lower in memory than )
                ( the base of the stack. )

4               ( Place another item on the stack. )

sp@ . 286740    ( sp@ has been decremented by )
                ( 4 bytes again. )
```

Notice that each time a parameter was added to the stack, the address returned by **sp@** (also called the 'stack pointer') is decremented by 4. This is because the tForth parameter stack is 4 bytes wide. Each item on the stack is a 4 byte, or 32-bit, value. This means the largest signed number which can be placed on the tForth parameter stack is 7FFFFFFF (hexadecimal).

The tForth Parameter Stack



Since one bit is used for the sign bit, this is the largest number which can be expressed with 31 bits.

tForth Is a 32-bit Forth Implementation

Because of the stack width, tForth is categorized as a 32-bit Forth implementation. A 32-bit Forth fits well on the 68000 microprocessor with its internal 32-bit wide data path and 32-bit general purpose registers. Most of the current Forths, including the Forth described in Starting Forth, are 16-bit Forths since 32-bit microprocessors have only recently become widely available.

Observing the Stack

Most Forth words either put values on the stack or remove items from the stack. **.s** is a word which displays the contents of the stack without disturbing the contents:

```
3 4 5 6 .s  3 4 5 6
```

Since **.s** does not disturb the stack in any way, it is a very handy tool for checking results. Another useful stack checking word is **depth** . **depth** returns a count of the number of items currently on the stack:

```
depth  .  4              ( There are four items on the stack. )
.  .  .  .  6 5 4 3     ( Take the four values off of the stack. )
depth  .  0              ( Now the stack is empty. )
```

tForth Words Which Operate on the Stack

In the glossary, the tForth words which operate on the parameter stack are grouped together under the 'Stack Manipulation' heading. The stack manipulation words are used to rearrange, to duplicate, to remove, and to check items on the parameter stack.

Here are some examples of the use of some of these stack manipulation words. The diagram on page 23 has a visual demonstration of the effects of these examples on the stack:

```
2 3 4                   ( Put three items on the stack. )

                        ( CHECKING THE STACK )
.s  2  3  4             ( Display the items on the stack )
                        ( without removing them from the stack. )

                        ( REARRANGING STACK ITEMS )
swap  .s  2  4  3       ( Move the second stack item to the top. )
rot  .s  4  3  2        ( Move the third stack item to the top. )

                        ( DUPLICATING STACK ITEMS )
over  .s  4  3  2  3 ( Copy the second item on the stack. )
                        ( Leave the copy on top of the stack, )
dup  .s  4  3  2  3  3  ( Copy the top item on the stack. )
                        ( Leave the copy on top of the stack. )

                        ( REMOVING STACK ITEMS )
drop  .s  4  3  2  3 ( Discard the top stack item. )
```

Simple Words Which Use the Stack

The words which perform the basic arithmetic operations: addition, subtraction, multiplication, and division, are all simple words which use the stack. These simple operators have been grouped under the "Arithmetic Operators" headings in the glossary. Here are some examples of their use:

Here is the name, pronunciation, and stack notation for each word used below:

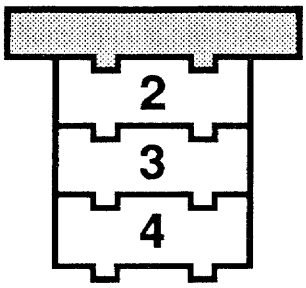
```
+  ( n1 n2 - n3 )     ('plus')
1+ ( n1 - n2 )        ('one-plus')

-  ( n1 n2 - n3 )     ('minus')
2- ( n1 - n2 )        ('two-minus')

*  ( n1 n2 - n3 )     ('times')
2* ( n1 - n2 )        ('two-times')

/  ( n1 n2 - n3 )     ('divide')
2/ ( n1 - n2 )        ('two-divide')
mod ( n1 n2 - n3 )

                        ( ADDITION )
2  3                    ( Put two numbers on the stack. )
+                       ( Use  +  ['plus'] to add the numbers. )
.s 5                    ( Display the result. )

1+                      ( Add 1 to the number on the stack. )
.s 6                    ( Display the result. )
```

(cont.)

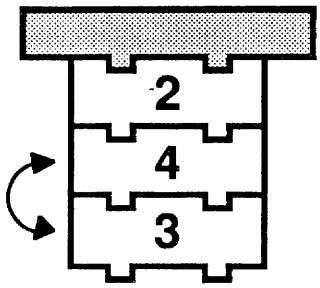
```
                    ( SUBTRACTION )
9  6                ( Put two numbers on the stack. )
-                   ( Use  -  ['minus'] to subtract the top )
                    ( number on the stack from the second )
                    ( number on the stack:  9 - 6 = 3 . )
.s 3                ( Display the result. )

2-                  ( Subtract 2 from the number on the stack. )
.  1                ( Remove the result from the stack and )
                    ( display it. )

                    ( MULTIPLICATION )
2  4                ( Put two numbers on the stack. )
*                   ( Multiply the two numbers. )
.s 8                ( Display the result. )

2*                  ( Multiply the number on top of the stack )
                    ( by 2. )
.  16               ( Remove the result from the stack and )
                    ( display it. )

                    ( DIVISION )
50  3               ( Put two numbers on the stack. )
/                   ( Divide the second number on the stack )
                    ( by the number on top of the stack: )
                    ( 50 / 3 = 16 . )
.s 16               ( Display the result. )

2/                  ( Divide the number on top of the stack )
                    ( by 2. )
.s 8                ( Display the result. )

5 mod               ( Divide the number on top of the stack )
                    ( by 5 and return the remainder. )
.  3                ( Remove the remainder from the stack and )
                    ( display it: 8 / 5 = 1 , remainder = 3 . )
```

Manipulating the Stack

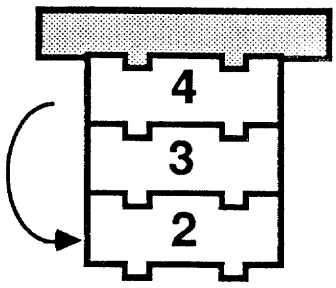
Examples are cumulative



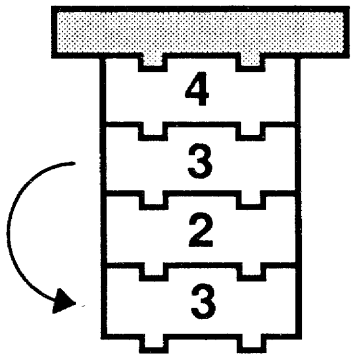
initial stack



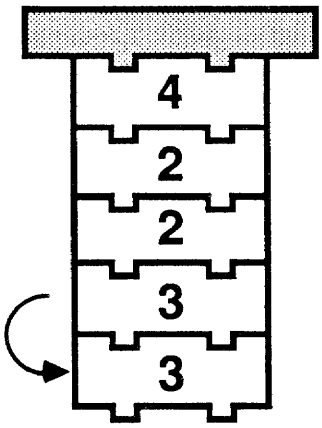
after swap



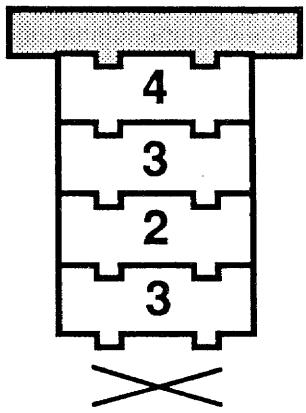
after rot



after over



after dup



after drop

All of these words take one or two numerical inputs, perform an arithmetic operation upon the input (s), and return a numerical result on the stack. However, as will be shown below, the stack does not have to be used for numerical values only.

Comparison Operators and Flags

The tForth comparison operators are another group of simple words which use the stack. The comparison operators treat their inputs as numbers and return a flag as a result. A flag is a value which may only represent one of two states: "true" or "false". In general, tForth treats any non-zero flag as a true flag and any flag with a value of 0 as a false flag. All of the words listed under the "Comparison Operators" section of the glossary, except the words **max** and **min** , will return a specific non-zero value, '-1', if the result of their operation is true and will return '0' if the result of their operation is false. Here are some examples of comparison operator use.

Here is the name, pronunciation, and stack notation for each comparison operator used below:

```
0<  ( n - f )          ('zero-less-than')
0=  ( n - f )          ('zero-equal')

=   ( n1 n2 - f )      ('equal')
<>  ( n1 n2 - f )      ('not-equal')

<   ( n1 n2 - f )      ('less-than')
>   ( n1 n2 - f )      ('greater-than')

inrange  ( n1 n2 n3 - f )

max   ( n1 n2 - n3 )
min   ( n1 n2 - n3 )

                       ( SINGLE PARAMETER COMPARISONS )
3  0=   . 0            ( False, 3 is not equal to 0. )
-2  0<  . -1           ( True, -2 is less than 0. )

                       ( DOUBLE PARAMETER COMPARISONS )
3  3  =   . -1         ( True, 3 is equal to 3. )
3  3  <>  . 0          ( False, 3 and 3 are equal. )
6  3  <   . 0          ( False, 6 is not less than 3. )
6  3  >   . -1         ( True, 6 is greater than 3. )

                       ( TRIPLE PARAMETER COMPARISONS )
5 2 8 inrange . -1     ( True,  2<=5<=8 )

4 8 max . 8            ( 8 is the larger of 4 and 8. )
4 8 min . 4            ( 4 is the lesser of 4 and 8. )
```

Note that the **max** and **min** comparison operators are the only ones which return numbers instead of flags.

Forth Is Not a "Typed" Language

In many languages, the type of each program parameter must be declared. If a parameter is a number it must be declared to be of type byte, integer, long, real, signed, unsigned. If a parameter is declared to be of type address or flag, it can only be used by functions which operate on addresses or flags. A language which requires typed parameters can help the programmer avoid mistakes since it is constantly cross checking actual input parameter types with the allowed input parameter types for a given operation.

The Forth language does not enforce typed parameters. Any type of item (number, address, flag) may be placed on the parameter stack. Since all Forth words can use the parameter stack, it follows that all Forth words can accept any type of input parameter.

There are both advantages and disadvantages to non-typed languages:

DISADVANTAGES

* A non-typed language cannot help the programmer by double checking all input parameters used.

* It is difficult for code written in non-typed languages to be shared since the types of the input and output parameters being used is usually not easily determined by the reader of the program listing.

ADVANTAGES

* A program written in a non-typed language should execute faster than a program written in a typed language because all of the code required for parameter checking is removed.

* A non-typed language gives the programmer the extra control over the language which is often required to get more performance out of a computer.

In Forth, as in any language, the second disadvantage above can be overcome by interspersing useful, thoughtful comments throughout the program code. The Forth community has taken the commenting solution a step farther by developing a suggested Forth commenting style which has been widely accepted. This commenting style, called 'stack notation' is discussed next.

STACK NOTATION

Stack notation is a standard method of commenting the stack usage of Forth words. For example:

```
<     ( n1  n2  -  f  )
```

This is the stack notation for the word < ('less-than'). The word (('left-paren') is a Forth commenting word. Because (is a Forth word, it must be surrounded on either side by at least one space or tab. Any characters in a Forth program which lie between parentheses are considered to be comments and are ignored by the Forth compiler. The characters between the parentheses above comprise the stack notation for < .

In stack notation, characters to the left of the '-' are used to indicate the inputs a Forth word expects to find on the parameter stack when it starts execution. Characters to the right of the '-' are used to indicate the outputs a Forth word will leave on the parameter stack when it completes execution.

In stack notation, the following codes are used to indicate parameter types:

CODE	MEANING	HEXADECIMAL RANGE
f	Boolean flag	0 = false, non-zero = true
c	7-bit ASCII character	0...7F
b	unsigned 8-bit number	0...FF
w	unsigned 16-bit number	0...FFFF
n	signed 32-bit number	-80000000...7FFFFFFF
u	unsigned 32-bit number	0...FFFFFFFF
a	32-bit address	0...FFFFFFFF

Here are other examples of stack notation. Note that a digit suffix is used to differentiate multiple parameters of the same type:

```
words  ( - )          ( words takes no inputs and returns no )
                      ( outputs. )

fill   ( a u b - )    ( fill takes three inputs and returns no )
                      ( outputs. )
```

```
key  ( - c )        ( key takes no inputs and returns one )
                    ( output. )

*/mod  ( n1 n2 n3 - n4 n5 )
                    ( */mod is a word which accepts multiple )
                    ( numeric parameters. Digits are used )
                    ( the 'n' code to differentiate the )
                    ( parameters. )

-trailing  ( a n - a n' )
```

When an output parameter is followed immediately by an apostrophe character it means the output parameter is a slightly modified version of an input parameter, rather than a completely new parameter. For example, -trailing takes as inputs a string address (a) and the length of the string (n). **-trailing** strips any trailing spaces from the string and returns the new, adjusted length of the string (**n'** , pronounced 'n-prime').

tForth Stack Notation

In tForth, the stack notation structure has been slightly extended to include comments:

```
c,  ( b - | Compile byte b at here. )
```

The '|' marks the end of the normal stack notation and the start of the comment field. The comment field can be as long as necessary (multi-line) as long as it is terminated by a closing paren.

INTEGERS AND MEMORY OPERATORS

Variable data is program data whose value changes during execution of a the program. Constant data is program data whose value will remain constant throughout program execution. For example, the equation used to calculate the area of a circle is a familiar equation which makes use of both constant and variable data:

```
(PI)        *  (radius, squared)     = area

(100*PI)    *  {[radius (meters)] ^2}   = area (cm ^2)
```

The '(100*PI)' or '(100*3.14 = 314)', is the constant in the circle area equation. The radius is the variable data. Scaling is used (the PI value is multiplied by 100 to eliminate fractional values) to ensure that only integer values are required.

Forth Note: Most Forth implementations do not support floating point number input/output or floating point math calculations. Many Forth designers/programmers feel that any floating point operation can be implemented using integer math with the proper scaling and that the integer math operations will be faster and more compact than their floating point counterparts.

Declaring Constant and Variable Program Data

The tForth word integer is used to define and name both constant and variable program data. The general format for the use of integer is:

<value> integer <name>

<value> is the 4 byte value for and <name> is the name of the constant or variable data. integer makes <name> an executable Forth word. Whenever <name> is executed it will put its associated value on top of the parameter stack.

Forth Note: If Forth were a strictly postfix language the syntax for integer would be:

<value> <address of name string> integer

The constant data for the circle area equation is defined as follows:

```
314   integer  pi*100
```

The variable data could be defined as follows:

```
1  integer  smallradius
7  integer  mediumradius
15 integer  largeradius
```

When one of the above names is executed, it will push its associated data onto the stack:

```
pi*100  . 314

smallradius  . 1
mediumradius  . 7
largeradius  . 15
```

Forth Note: The following colon definition performs the same action as the integers above when executed:

```
: mediumradius  7  ;
```

When mediumradius is executed, it will push a '7' onto the stack. The drawback of the colon definition is that the '7' is "hardcoded" into the definition. If mediumradius must put a different value on the stack, the definition would have to be recompiled.

Integers, on the other hand, were designed so that their contents could be easily modified during program execution and thus are ideal for use as program variables. The operators used to alter the contents of integers are discussed below.

Now let's make the circle area equation part of a Forth word which, when passed a radius value (expressed in meters) will return the corresponding area (expressed in centimeters squared):

```
: circlearea  ( n1 - n2 )
     dup            ( Make a copy of the radius. )
     *              ( Multiply:  radius*radius , square it. )
     pi*100  *      ( Multiply the radius squared by the )
;                   ( pi*100  constant. )

smallradius  circlearea  . 314
mediumradius  circlearea  . 15386
largeradius  circlearea  . 70650
```

Altering Integer Data

The tForth words to , +to , on , and off are used to modify integer data:

```
23  largeradius  to   ( Put a '23' in the  largeradius  integer. )
largeradius  . 23     ( Get and display the contents of  )
                      ( largeradius  .  )

5  largeradius  +to   ( Add '5' to the contents of  largeradius  .  )
largeradius  . 28     ( Get and display the contents of )
                      ( largeradius  .  )

largeradius  off      ( Put a 'false' flag, '0', in  largeradius .  )
largeradius  . 0
```

(cont.)

```
largeradius on       ( Put a 'true' flag, '-1', in largeradius . )
largeradius . -1
```

to is used to change the contents of an integer to a specified value. +to is used to add a 4 byte value to the contents of an integer. off and on are boolean integer operators, usually used on integers which are being used as flags. off is used to turn an integer value 'off', i.e. to set the integer's value to 'false' (0). on is used to turn an integer value 'on', i.e. to set the integer's value to 'true' ('-1' or 'non-zero').

The Use of Integers Versus the Direct Alteration of Memory

tForth's generic integer data structure (can be used to hold either constant or variable data) frees the programmer from having to treat variable data differently than constant data. They also free the programmer from having to remember the 'type' of a particular piece of memory. This is especially convenient in a multi-programmer programming environment where each programmer may not be intimately familiar with the constants and variables being used by another programmer. Any tForth programmer is able to get the value of any integer without having to know whether another programmer is using the integer as a constant or variable.

A limitation of integers, however, is that they only support interaction with 4 byte data values. Handling of data sizes which are smaller (byte, word) or larger (arrays, data structures) than 4 bytes requires that the contents of memory be accessed directly.

Directly Accessing the Contents of Memory

The following words are the main words used for direct manipulation of data in memory. The stack notations for these words are:

WORD	PRON.	STACK NOTATION
!	'store'	(n a - | Stores the 4-byte value on the parameter stack, 'n', into memory starting at the address 'a' .)
@	'fetch'	(a - n | Fetches the 4-byte value 'n' stored in memory starting at address 'a' and returns it on the parameter stack.)
w!	'w-store'	(w a - | Stores the lower 2 bytes of the 4-byte value on the parameter stack into memory starting at address 'a'.)

w@	'w-fetch'	(a - w | Fetches the 2-byte value 'w' stored in memory starting at address 'a' and returns it in the lower two bytes of the number on top of the parameter stack, the upper two bytes are set to zero.)
c!	'c-store'	(c a - | Stores the lowest order byte of the 4-byte value on top of the parameter stack into memory starting at address 'a'.)
c@	'c-fetch'	(a - c | Fetches the 1-byte value 'c' stored in memory at address 'a' and returns it in the least significant byte of the number on top of the parameter stack. The upper three bytes are set to zero.)
+!	'plus-store'	(n a - | Adds the 4-byte increment value 'n' to the 4-byte value located in memory starting at address 'a'.)

Directly Altering Integer Data

The execution of an integer variable name puts the value of the integer variable directly on the stack. To get the address of the location where an integer's data is stored, use the word addr ("adder") immediately after the name of the integer:

```
hex
largeradius  addr  . 478FE ( The contents of the integer )
                           ( largeradius  are located in )
                           ( starting at address '478FE'. )
```

To directly access the contents of largeradius , without using to or executing largeradius , the direct memory access words described above may be used to directly access the memory location ('479F6') where qty's contents are stored:

```
12345678  479F6  !    ( Store the 4 byte value '12345678' in )
                      ( memory starting at the address '479F6' . )
479F6  @  . 12345678  ( Fetch and display the 4 byte value )
                      ( residing in memory starting at address )
                      ( '479F6'. )

9876  479F6  w!        ( Store the 2 byte value '9876' in memory )
                      ( starting at the address '479F6'. )

479F6  w@  . 9876     ( Fetch and display the 2 byte value )
                      ( residing in memory starting at address )
                      ( '479F6'. )

FF  479F6  c!         ( Store the 1 byte value 'FF' into memory )
                      ( starting at the address '479F6'. )
479F6  c@  . FF       ( Fetch and display the 1 byte value )
                      ( residing in memory starting at address )
                      ( '479F6'. )
```

```
5  479F6  !          ( Store the 4 byte value '12345678' in )
                     ( memory starting at the address '479F6' . )
1  479F6  +!         ( Add 1 to the 4 byte value located at )
                     ( address '479F6'. )
479F6  @  . 6        ( Fetch and display the 4 byte incremented )
                     ( result. )
```

Other Useful Direct Memory Access Operators

and! , **or!** , **not!** , and **xor!** are memory operators which perform logical operations with data stored somewhere in memory.

fill , **move** , and **cmove** are memory operators which perform memory operations on large sections of memory. See the "Memory Operators" section of the Glossary for more information of these words.

Starting Forth Note

Chapter 8 of Starting Forth discusses how the defining words variable and constant are used to create named variable locations and constant values in Forth. In tForth **variable** and **constant** have been replaced by the single word **integer.** The following table compares **integers**, **variables**, and **constants**:

ACTION	INTEGER	VARIABLE/CONSTANT
Create a named variable location:	5 integer fred	variable fred
Store a value into a variable:	7 fred to	7 fred !
Fetch a value from a variable:	fred	fred @
Increment the contents of a variable:	1 fred +to	1 fred +!
Get the address of a variable location:	fred addr	fred
Create a named constant value:	12 integer dozen	12 constant dozen
Get the constant value:	dozen	dozen

As the table shows, there is no substantial difference between

constant and integer when the purpose is the creation of constant data. The difference between integer and variable when creating variable data is that integer allows the creation of initialized variable data and the value of a variable created with integer can be obtained by simply executing the name of the integer variable. Variables created with variable must use the @ operator to obtain their values. If a program works mainly with 4 byte variables and never needs the addresses of the variable locations, the use of integer to create those variables can save many 'fetch' operations during the execution of a program. The integer operators to and +to are also more readable commands than the memory operators ! and +! .

Displaying the Contents of Memory

The tForth utility word **dump** is used to display the contents of memory:

```
479F6  10  dump
479F6  FF 76 56 78 00 00 00 DA 81 00 07 2F 83 63 6E 74
          .vVx......./.cnt
```

dump has the following stack notation:

dump (a n -)

dump displays 'n' bytes of data starting at address 'a' in memory. The start address of the memory dump is shown on the far left of the display. **dump** displays memory in 16-byte chunks. Each byte in the memory dump is separated from the next byte by a space. The ASCII equivalents of the 16 byte values shown in the memory dump are listed on the far right side of the display. Here is how dump would be used to display 40 (hex) bytes of memory:

```
479F6  40  dump
479F6  FF 76 56 78 00 00 00 DA 81 00 07 2F 83 63 6E 74  .vVx......./.cnt
47A06  07 37 89 63 75 72 72 65 6E 63 79 24 07 35 85 64  .7.currency$.5.d
47A16  61 74 65 24 07 30 88 64 65 63 69 73 69 6F 6E 07  ate$.0.decision.
47A26  31 89 64 65 63 69 73 69 6F 6E 32 07 38 8C 6D 61  1.decision2.8.ma
```

PROGRAM CONTROL STRUCTURES

There are four major types of program control structures:

1. Conditional Execution

   Code within a conditional execution structure is executed only if certain conditions are met. The tForth words used for the implementation of conditional execution structures are:

   if **else** **then**

2. Definite Loops

   Definite, or counted, loops are used to cause a set of instructions to be executed a specific number of times. The number of times the loop is to be executed is known prior to the start of the loop. The tForth words used for the implementation of definite loops are:

   do **+loop** **loop** **i**

3. Indefinite Loops

   Indefinite loops are used to cause a set of instructions to be executed an unknown number of times. These types of loops are termed 'indefinite' because the number of times the loop will be executed is determined during execution of the loop. The tForth words used for the implementation of indefinite loops are:

   begin **until** **again** **while**

4. Forced Execution

   Forced execution words are used when program execution must be unconditionally redirected to another section of the code. The tForth forced execution words are:

   abort **abort"** **exit** **leave**

A Special Note about tForth Program Control Structures

In most Forth systems, program control structures can only be used within colon definitions. In tForth, program control structures can be used interactively. This is very useful for testing out ideas since the extra work required to create a new definition is eliminated and the dictionary doesn't become cluttered with test definitions. Several of the examples presented in the section are to be executed interactively. Interactive execution of a program control structure commences when the final word in the program control structure is entered (try the examples below).

Conditional Execution

The conditional execution structures allow programs to make decisions. In the following example, the word decision decides whether a 5 should be displayed by examining the flag passed to it:

```
: decision  ( f - ) ( Create a definition named decision . )
     if             ( Check the flag. )
          5 .       ( IF it is true, non-zero, display a 5. )
     then ;         ( Mark the end of the 'if...then' structure. )

0 decision          ( 0 is false so the 5 is not displayed. )
1 decision 5        ( 1 is true so the 5 is displayed. )
```

When the flag passed to **if** is true (non-zero) the code between the **if** and the **then** will be executed. When the flag is false (0), **if** will reroute program execution to the code which immediately follows the **then** .

The 'if...then' structure can be extended by inserting an **else** in the middle:

```
: decision2 ( f - )
     if
          5 .       ( IF the flag is true, display a five... )
     else
          6 .       ( ELSE the flag is false, display a six. )
     then ;

0 decision2 6       ( Flag was false so a 6 was displayed. )
1 decision2 5       ( Flag was true so a 5 was displayed. )
```

Definite Loops -- 'Do ... Loop'

The following interactive example shows a 'do...loop' being used to display the numbers from 0 through 9:

```
10                  ( Place the limit on the stack. )
0                   ( Place the index on the stack. )
do                  ( Start the loop. )
     i .            ( This is the code to be executed )
                    ( each time through the loop. )
loop 0 1 2 3 4 5 6 7 8 9   ( End of the loop. )
```

Execution of a counted 'do...loop' always requires the specification of the number of times the loop is to be executed. The count is specified by placing two numbers on the stack. These two numbers are referred to as the loop "limit" and the loop "index". The number of times the loop will be executed is determined by subtracting the index value from the limit value. So, in the above example, the loop will be executed 10 times.

When the word **do** executes it moves the limit and index values from the parameter stack to the return stack. The index value

will be the top item on the return stack and the limit value will be the second item on the return stack. **do** is executed only once in a 'do...loop'. When the word **loop** executes it subtracts one from the index value on the return stack and compares the new index value to the limit value. **loop** is executed each time through the loop. When the index value equals the limit value, the loop is immediately terminated (this is why the limit value, 10, was not displayed).

The word **i** copies the top item on the return stack and places the copy on top of the parameter stack. **i** is normally used during execution of a 'do...loop' to get the value of the current loop index (which is the top item on the return stack during a 'do...loop'). In the example, **i** was used to get the current loop count each time through the loop and . was used to take the number off of the parameter stack and display it.

Note that the code in a definite loop will always be executed at least once since the loop termination check occurs at the end of the loop.

Definite Loops -- 'Do ... +Loop'

The 'do...+loop' definite loop structure is used when there is a need for a counted loop which "counts" by a value other than one. For example, to display the even numbers between 0 and 10:

```
10  2  do            ( Pass the loop limit and index to  do .  )
     i  .            ( Get the current index and display it. )
2                    ( Place the loop increment value on the )
                     ( stack for  +loop  .  )
+loop 2 4 6 8
```

The main difference between 'do...loop's and 'do...+loops' is that **+loop** is passed the desired increment value for the loop index. The number of times a 'do...+loop' will execute is determined using the following equation (where square brackets indicate "integer part of"):

[(limit-index)/increment] = number of times loop will be executed

So, the loop above was executed (10-2)/2 = 4 times. **+loop** also accepts negative increment values. When a negative increment value is used, the loop will not terminate until the index becomes less than the limit so the equation for calculating loop execution cycles becomes:

[(limit-index)/increment]+1 = number of times loop will be executed

```
10  20  do
     i  .
-2
+loop 20 18 16 14 12 10
```

A mistake such as will result in a seemingly infinite loop:

```
0  10  do      ( Start at '10' and count to '0'. )
       i .
3
+loop
```

The loop will eventually terminate. The initial loop index value '10' will be continually incremented by 3 until at some point, it gets so large that it will not be able to be expressed as a 32-bit value. When the index value reaches this 32-bit "cut-off" the value will appear to change from a very large postive number to a very large negative number. This large negative index value will continue to be incremented by 3 and eventually will reach zero.

Indefinite Loops

The most common indefinite loop structure is the 'begin...until' loop. In a 'begin...until' loop, the code between the begin and the until is executed until the flag passed to until is true (non-zero). If the flag passed to until is false (0), until will reroute program execution back to the code which immediately follows the begin. For example, the 'begin...until' loop below will not terminate until the user presses the 'a' key. The example uses the word key , which has not yet been discussed, to obtain the user's input and to place the ASCII value of the character pressed on the parameter stack. Since the ASCII value for 'a' is 97, a comparison is made to determine whether the key pressed was the 'a' key:

```
decimal
begin
      key  97  =
until
```

After this example is entered the text will remain highlighted UNTIL the lowercase 'a' key is pressed.

Placing Conditionally Executed Code in an Indefinite Loop

while is a conditional-test word which may be included in any indefinite looping structure. while allows code which is to be conditionally executed to be included in an indefinite loop. If the flag passed to while is true (non-zero), the code following the while will be executed. If the flag passed to while is false (0), then the code after the nearest following while , again , or loop will be executed. again always reroutes program execution back up to the code which immediately follows the begin .

To use the example below, send the underlined text to tForth and then press the 'a' key six times. Each time the 'a' key is pressed (after the 'a' key is pressed) the current index value will be displayed and incremented. Press any other key to

terminate the loop:

```
decimal                 ( change to base 10 )
0  integer  index       ( Define an integer variable to be )
                        ( used to hold an index value. )
begin                   ( Start the loop. )
      key  97  =        ( Did the user press the 'a' key ? )
while                   ( WHILE the user did press the 'a' )
                        ( key execute the following code. )
      index  .          ( Display the current index value. )
      1  index  +to     ( Increment the index by one )
                        ( and go back to the top of the loop. )
again 0 1 2 3 4 5
```

Any number of **while** decision points may be inserted into an indefinite looping structure.

Forced Execution

The forced execution words will immediately and unconditionally redirect program execution when they are executed.

The Word **leave**

leave is a forced execution word used to immediately leave from any definite or indefinite looping structure. When used inside of a definite looping structure, **leave** is responsible for removing the loop limit and index from the return stack:

Note: A "nested" program control structure is a control structure which contains another program control structure. The leave example below uses nested control structures; an 'if...then' structure is used inside of a 'do...loop' structure. A control structure may contain any number of nested control structures. However, words which leave control structures (leave and while) will only leave from the current control structure to the next outer control structure.

```
:  shortened-loop       (  -  )
      10  0  do         ( We seem to want to run the loop )
                        ( 10 times. )
           i  .         ( Print the current index value. )
           i  7  >      ( Is the loop index greater than 7 ? )
           if           ( If it is... )
                leave   ( leave this loop. )
           then
      loop  ;

shortened-loop 0 1 2 3 4 5 6 7 8
```

leave will always reroute program execution to a point right outside of the nearest following until , again , loop , or +loop . If leave is used inside a set of nested looping structures, it will only leave the current loop.

The Word **Exit**

exit is a forced execution word which must be used within a colon definition. Whenever exit is encountered in a colon definition it will immediately terminate execution of that colon definition and will redirect program execution back to the word which originally called the definition:

```
: unfinished  ( - )
     1  .          ( Display a '1'. )
     2  .          ( Display a '2'. )
     3  .          ( Display a '3'. )
     exit          ( Terminate execution of this definition. )
     4  .          ( The '4' will not be displayed. )
     5  . ;        ( The '5' will not be displayed. )

unfinished 1 2 3
```

As soon as the exit in unfinished was reached, execution of unfinished was terminated.

The Words **abort** and **abort"**

abort will cause a Forth system abort. In a Forth system abort the return stack and parameter stack are cleared and Forth is restarted. **abort** may be used interactively or within a colon definition.

abort" is a version of **abort** which accepts a flag and, if the flag is true (non-zero), aborts, issues a beep, and displays an error message on the "explain" screen ([USE FRONT][EXPLAIN]). The error message for **abort"** immediately follows **abort"** and is terminated with a trailing quote. Note that there must be at least one space or tab between **abort"** and the start of the error message. **abort"** may only be used within a colon definition:

```
: testabort  ( f - )
     abort" Error Error" ;

0 testabort          ( Nothing should happen. )
1 testabort          ( The system should beep and the )
                     ( error message "Error Error" should )
                     ( be displayed on the explain screen. )
```

CHARACTER AND STRING I/O

Character input

ascii

ascii is a character manipulation word which returns the ascii value of the single character which follows it:

```
ascii s  . 115        ( The ASCII code for an 's' is decimal 115. )
115 . 115             ( This is another way to put the ASCII code )
                      ( for an 's' on the stack. )
```

Note that 'ascii s' has the same effect as '115', both command sequences place the decimal ASCII value for 's' on the stack. The '115' is a more meaningful result when it is viewed as the result of ascii .

ascii can be use to make all single character comparisons in your program much more readable:

```
115  116  =  . 0      ( Comparing the ASCII codes for 's' and 't' )
                      ( in an unreadable fashion. )

ascii s  ascii t  =  . 0   ( Does the ASCII code for 's' equal the )
                      ( ASCII code for 't' ?  The false [0] flag )
                      ( returned shows that they are not equal. )
```

?t

?t is a character input word which checks to see if any characters input by the user are available. If the user has typed a character, a true (non-zero) flag will be returned. If no user input characters are waiting, a false (0) flag will be returned. The word keypress below spins in a loop, printing a message, until the user presses any key:

```
: keypress (  -  | Displays a message until any key is pressed. )
     begin
          cr
          ." Press any key to terminate."
     ?t
     until
     cr  ." Done."  ;

     keypress
     Press any key to terminate.
     Press any key to terminate.
     Done.
```

The Word **key**

While **?t** only reports on the presence of user input, **key** is a character input word which waits until the user presses a key and then returns the ASCII code for the key pressed. **key** forces the system to wait until the user inputs a character. **key** will be incorporated into the **keypress** example to obtain a more specific response from the user. In the new example, the user must press a certain key, an 's', to terminate the message printing loop, but can press any other key to print the message out:

```
: keypress ( - | Displays a message whenever any key except an
's' is pressed.  When an 's' is pressed, the loop terminates.  )
      begin
            cr       ( Print a carriage return followed )
                     ( by the message. )
           ." Press an 's' to stop or any other key to continue."
      key  ascii s  =    ( Did the user press the 's' key ? )
      until
      cr  ." Stop."  ;  redefining keypress

keypress
Press an 's' to stop or any other key to continue.
Press an 's' to stop or any other key to continue.
 Stop.
```

Note that when this new version of **keypress** was sent to tForth, a "redefining keypress" message was issued.

Character Output

emit takes a number and outputs the corresponding ASCII character. All of the other character output words are built using emit . cr uses emit to output a carriage return and a linefeed. space uses emit to output a space. The following demonstration uses cr twice to produce two carriage returns, uses space twice to produce two spaces, and uses emit three times to output three asterisks:

```
: demo  ( - )
      cr  cr
      space  space
      42 emit  42 emit  42 emit  ;

demo


  ***
```

emit is a vectored output routine. When **emit** executes it checks the state of four output device flags. There is a flag for the screen (crt), the printer (lp), the editor (edde), and the modem/serial port (ser). If a flag is set, it means that the corresponding output device is currently enabled. Whenever emit is used, it must output the character to all output devices which are currently enabled. Since emit is smart enough to know how to talk to all of the devices mentioned above, the

programmer does not have to worry about the idiosyncracies of each device.

String Creation

The string manipulation word " ('quote') will construct either a temporary or permanent string in memory and will return the address and length of the string on the stack:

```
" This is a test."  .  . 31 471930
```

" will place all characters between itself and the trailing quote into the string being constructed. Because " is a Forth word, it must be surrounded on both sides by at least one space or tab. The text to be included in the string should immediately follow the " and the space. The text should be terminated with a closing quote. The closing double quote is used only as a delimiter, so it does not have to be separated from the rest of the text. Double quotes may not be used within the string text.

String Output

type is a string output word which, when passed the address and length of a string, will output the string to the screen (and/or any other current devices).

```
" This is a string."  .s  type 746BD 25 This is a string.
```

.s was used in the above example, between the string construction and type , to show that " does leave the address and length of the string on the stack for type . type was then used to output the string to the screen.

String Integers

string is the string-handling equivalent to integer . In the following example the text within the quotes is the string data. mystring is the name assigned to the string data. string makes mystring an executable word which will put the address and length of its associated string data on the stack when executed.

This is how **string** is used:

```
" This is string data."  string  mystring
```

This is how the **mystring** string may be displayed:

```
mystring  typeThis is string data.
```

An empty string is created when no characters are included between the quotes (when quote-space-quote is used):

```
" "  string  emptystring
```

Changing String Data

The word "to is used to alter string data. The use of "to is similar to the use of the integer manipulation word to . "to is smart enough to handle changing string sizes.

```
" "  string  stringinteger      ( Create an empty string integer. )

" abcdefg" stringinteger "to    ( Store some text in the string )
                                ( integer. )

stringinteger  type abcdefg     ( Display the contents of the )
                                ( string integer. )
```

String Input

The tForth word **query** takes string input from the editor environment and passes the input to tForth. When **query** is executed, it waits until it is passed a string from the editor. Any string may be passed from the editor to **query** by selecting the desired string and pressing [ANSWER] while holding the [USE FRONT] key down.

This is the stack notation for **query** :

```
query  ( - a n )
```

query returns on the stack the address 'a' and length 'n' of the string passed to it.

In the example below, **query** is used to obtain a line of user input. The address and length of the user's input, returned by **query** , are passed on to **type** so that the input will be immediately displayed. To use the example below perform the following steps:

- Send the word stringinput to tForth.
- Enter the text you would like to return to query . In the example, 2 spaces were typed, followed by the words "Hello there.", followed by 2 more spaces.

o Highlight your text and return the text to **query** with the use of [USE FRONT][ANSWER].

```
: stringinput  (  -  )
      space        ( Put a space in front of the response string. )
      query        ( Wait for string input. )
      type         ( Display the input string. )
;

stringinput   Hello there.    Hello there.
```

The first instance of " Hello there. " above was typed in response to **query** . The second instance of " Hello there. " was output by **stringinput** .

NUMERIC I/O AND NUMBER FORMATTING

Numeric input involves the conversion of ASCII strings which represent numbers to numbers which may be placed on the parameter stack. Numeric output involves the conversion of numbers to strings of ASCII characters which may be displayed using the string I/O words. There are four categories of numeric I/O words: words used to control the numeric conversion base, words used to handle numeric input conversion, words used to handle formatted numeric output, and words used to handle standard numeric output.

Numeric Conversion Base

Numbers are always converted with respect to the current numeric base. The current number base is controlled by the system integer base . The two most commonly used number bases, base ten (decimal) and base 16 (hexadecimal), may be chosen with the special words decimal and hex :

```
( The system was in hexadecimal base when these definitions
  were defined. )

: decimal ( - ) 0a base to ;

: hex ( - ) 10 base to ;
```

The examples below demonstrate how you may set or change the current numeric base:

```
decimal                 ( Set the base to decimal. )
10  15  20              ( Put some numbers on the stack. )

hex                     ( Change the base to hexadecimal. )
.s A F 14               ( Display the hexadecimal equivalents )
                        ( of the numbers. )

2 base to               ( Change base to base 2, binary. )
.s 1010 1111 10100      ( Display the binary equivalents )
                        ( of the numbers. )

7 base to               ( Non-standard bases are also allowed. )
.s 13 21 26             ( Display the base 7 equivalents. )

decimal                 ( Change the base back to decimal )
. . . 20 15  10         ( and remove the numbers from the stack. )
```

Note that both the number input words and the number display words are affected by the current base setting.

Numeric Input Conversion

The word **number** takes the address and length of a string, the desired conversion base, and tries to convert the string to a number:

```
decimal
" 1234"  10  number  .  . -1 1234

" 123X4"  10  number  . 0
```

In the examples above, " was used to create a temporary string. The address and length of the temporary string, and a '10' for base 10 were passed to **number**. The '10' was treated as a decimal '10' since **decimal** was used above to set the system base to decimal.

In the first use of **number** the temporary string contained a sequence of valid ASCII numerical characters. **number** was able to successfully convert the string to a number and returned two values, a true (non-zero) flag and the converted numerical value, on the parameter stack.

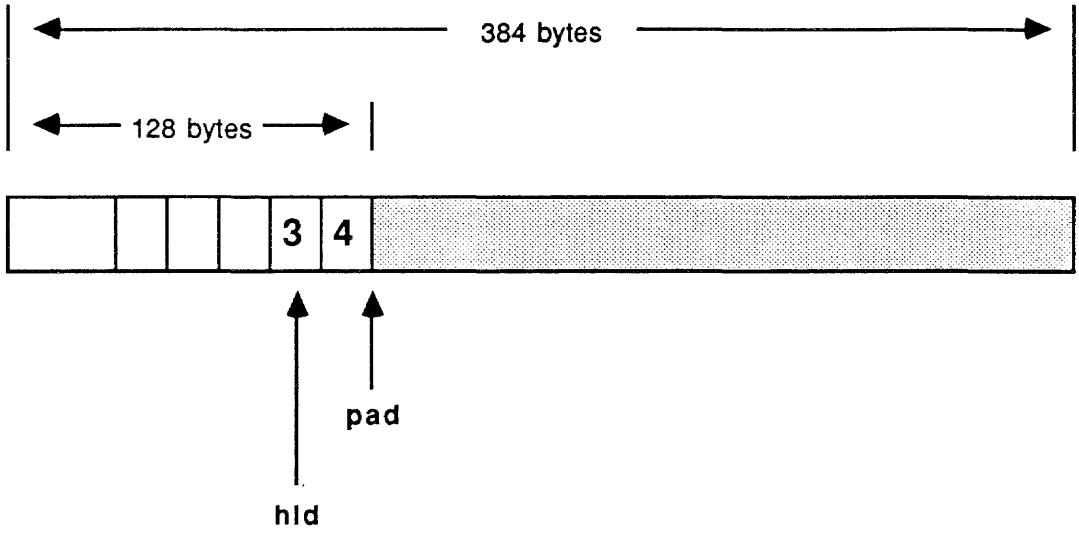
In the second use of **number** the temporary string contained an invalid numerical character (the 'X'). number was not able to convert the string to a number and returned only one result, a false (0) flag, on the stack.

Number Output Conversion and Number Formatting

The number formatting words are used to convert binary numbers to printable strings of ASCII numerals. 128 bytes of a 384-byte scratch area called 'the pad' (see diagram on the following page) are used by the number formatting words to hold the output string as it is being constructed. Executing the word **pad** will cause the address of a location 128 bytes into the pad to be placed on the parameter stack. An integer named **hld** is used to hold a pointer to the spot in the string where the next character will be inserted.

Number Formatting

(decimal)



These are the names and functions of the number formatting words to be used in the example:

<# (n - n)
('less-sharp')
Must always be used at the start of a number conversion process. Initializes the hld pointer with the address of pad (the number to be converted, 'n', should be on the stack, although <# does not use it):

```
: <# ( - ) pad hld to ;
```

hold (c -)
Lower level word used by # . Decrements the hld pointer by one and then takes the ascii value from the stack and places it in the next available spot in the string:

```
: hold ( c - )
     -1 hld +to   ( Decrement the hld pointer. )
     hld c! ;     ( Store ASCII value into string. )
```

digit (n1 n2 - n1' c)
Lower level word used by # . Extracts one digit from the number being converted 'n1' using the specified base 'n2' and converts the digit to its corresponding ASCII value. Returns the remainder of the number 'n1'' and the ASCII value 'c' .

(n - n')
('sharp')
Uses digit to extract one digit from the number on top of the stack (the number being converted) and then uses **hold to]** insert the ASCII code for the digit into string being constructed in the pad:

```
: # ( n - n' ) base digit hold ;
```

#> (n1 - a n2)
('sharp-greater')
Removes the remainder of the number being converted from the stack (the number should be zero if it was completely converted), and returns the address 'a' and length 'n2' of the output string:

```
: #> ( n1 - a n2 )
     drop  ( Drop the remainder. )
     hld   ( Put string start address on stack. )
     pad   ( Put end address of string on stack.  )
     over  ( Put copy of start address on top. )
     - ;   ( Subtract to get string length. )
```

The simple example below shows how number formatting words could be used to convert a 3-digit number to a string:

```
234 <# # # # #> space type 234
```

The diagram of the pad showed how the example string above looked while it was under construction. When the diagram was drawn, only 2 of the 3 digits had been added to the string. Note that the string is constructed from right to left. The least significant digits are inserted into the string first, and the most significant digits last. The hld pointer is always pointing to the current last character in the string.

u.

The word u. , which is used to display unsigned numeric values, solves the example task in a more generic manner:

```
: u. ( n - )
     <# #s #> space type ;

hex
FFFFFF34 u. FFFFFF34
```

u. uses #s , an extended version of the # formatting word, to extract all the digits from any size number:

#s (n - 0)
('sharp-s')
Continually extracts digits from the number being converted and inserts the ASCII values into the string being constructed until the number being converted is reduced to zero:

```
: #s ( n - 0 )
     begin
        #      ( Get one digit at a time. )
        dup      ( Copy the remaining value. )
        0=     ( Is it zero yet ? )
     until  ;     ( Go until the number is 0. )
```

Dot (.)

The word . , which is used to display unsigned or signed numerical values, uses the number formatting word **sign** :

sign (n -)
If the number on the stack is negative sign will insert a minus sign (ASCII value = hex 2D) into the string being constructed in the pad:

```
: sign ( n - )
     0<       ( Is 'n' negative ? )
     if
        2D hold     ( If it is, insert a '-'. )
     then  ;
```

(cont.)

```
: .  ( n - )
     dup      ( Duplicate number to convert. )
     abs      ( Take absolute value of copy. )
     <#       ( Start number formatting. )
        #s    ( Convert all of the digits. )
        swap     ( Put original number on top. )
        sign     ( If negative, insert '-' in string. )
     #>       ( End conversion. )
     space
     type  ;     ( Display string. )

hex
FFFFFF34  . -CC
```

Note that . , which is affected by the sign bit on a number, displayed the value 'FFFFFF34' in a different manner than u. did previously.

Inserting Special Characters Into a Formatted String

The following example shows how number formatting can be used to convert a 6-digit number to a common date format (mm/dd/yy):

Note: It is customary, but not necessary, to use a '$' at the end of string names and string-handling word names. For example, date$ is a word which creates and displays formatted date strings.

```
: date$  ( n - | Takes 6-digit number, converts it to mm/dd/yy
date format string, and displays string.  )
     <#            ( Start the number conversion process. )
        #          ( Convert least significant digit of year. )
        #          ( Convert most significant digit of year. )
        ascii  /  hold ( Insert a '/' in the string. )
        #          ( Convert least significant digit of day. )
        #          ( Convert most significant digit of day. )
        ascii  /  hold ( Insert a '/' in the string. )
        #          ( Convert least significant digit of month. )
        #          ( Convert most significant digit of month. )
     #>            ( End conversion process. )
     space
     type  ;       ( Display date string. )

100961  date$ 10/09/61
```

The date string is constructed from right to left. Note that the phrase 'ascii / hold' was used in place of the equivalent phrase '92 hold' for readability.

Storing Formatted Strings in String Variables

The following example shows how a value with any number of digits can be converted to the United States currency format ($dddd.cc). A string variable is created for use as a storage for the currency string which can be printed out at a later time:

Note: Since the pad is used as a scratch area by many tForth words, important data, such as formatted strings, should not be kept in the pad.

```
"  "  string  currency$ ( Create an empty string variable location. )

   : makecurrency  ( n - | Converts a number to $dddd.cc format
   string and saves string away in string variable. )
        <#              ( Start number formatting process. )
           #            ( Convert least significant 'cents' digit. )
           #            ( Convert most significant 'cents' digit. )
           ascii  .  hold ( Insert decimal point. )
           #s           ( Insert all of the 'dollars' digits. )
           ascii  $ hold  ( Insert dollar sign. )
        #>              ( End number formatting process. )
        currency$  "to ( Copy string into variable )
        ;              ( for later use. )

   1257595 makecurrency
   currency$  space  type $12575.95
```

The string creating word **string** and the string operating word "to were discussed in the section on string I/O.

.r and u.r

.r and u.r are formatted versions of the words . and u. These words print signed and unsigned values, right justified, in a field with a specified width:

```
   fixedfont          ( For these words to print with proper )
                      ( alignment in the editor, a non-proportional )
                      ( or fixed, font must be used. )
   : signed-aligned  ( - )
      cr                    ( Output a carriage return. )
123         10  .r   cr ( Print 123 in 10 char field )
      123456      10  .r   cr ( followed by carriage return. )
      FFFFFF34    10  .r   cr  ; ( Repeat for 2 other numbers. )

   signed-aligned
    123
 123456
   -CC

   : unsigned-aligned  ( - )
      cr                    ( Output a carriage return. )
      123         20  u.r  cr ( Print 123 in 20 char field )
      123456      20  u.r  cr ( followed by carriage return )
      FFFFFF34    20  u.r  cr  ; ( repeat for 2 other numbers. )
   unsigned-aligned
                    123
                 123456
              FFFFFF34
   variablefont          ( To return to a proportional )
                         ( font, if you'd like. )
```

LOCAL VARIABLES

A major drawback of the Forth parameter stack is that with several parameters on the stack, the manipulations required to get at a certain parameter become cumbersome and the resulting code unreadable. For example, this is a routine which sums the numbers between 0 and n where n is an arbitrary limit. In this routine, the arbitrary limit is reached when the user hits a key:

```
: summation ( -> n | Spins in a loop, summing the loop count,
until the user hits a key. Returns the summation on the stack. )

0                       ( The initial sum is 0. )
0                       ( The initial loopcount is 0.)

begin                   ( Start the loop. )
      1+                ( Increment the loop count. )
      dup               ( Copy the loop count. )
      rot               ( Rotate the current sum to the top of )
                        ( the stack. )
      +                 ( Add the copy of the loop count to the )
                        ( current sum. )
      swap              ( Put the loop count back on top. )
?t                      ( Has the user pressed a key ? )
until                   ( Go until ?t reports that the )
                        ( user has pressed a key. )
drop                    ( Drop the loop count but leave the )
;                       ( summation on the stack. )

summation . 4367490
```

The inner loop of the above routine, the words between the begin and until (two program control structures which will be explained later), is very difficult to comprehend at first glance. The stack notation indicates that the routine takes no inputs and returns one output but it does not provide any information regarding the use of the stack within the routine. Even though this example has only a maximum of three items on the stack, it is very difficult to work through without resorting to pencil and paper to keep track of the stack usage.

A Description of tForth's Local Variables

In tForth programmers are allowed to create 'local variables', that is, variables which are valid only during execution of the word in which they are defined. The main advantage of local variables is that they help produce more readable code by eliminating confusing stack manipulations.

This is how local variables are used in a definition:

```
: <name>  (  -  )
      local  n1
      local  n2
         .
         .
      local  ni
;
```

The word **local** is used to create and name a local variable. The local variable is not initialized to any value. Any number of local variables may be created in a definition. Since the variable is a local variable, references to **<name>**'s local variables are only valid within **<name>** None of the words which call **<name>** and none of the words which **<name>** calls can reference **<name>**'s local variables.

The next example shows the **summation** routine after it has been rewritten to take advantage of local variables:

```
: summation  (  ->  n  | Spins in a loop, summing the loop count,
until the user hits a key.  Returns the summation on the stack. )

   local sum          (  Creating and naming a local variable. )
0  sum  to         (  Initializing  sum  with a zero. )

   begin
        1  sum  +to (  Increment the contents of  sum  by one. )
?t
   until              (  Go until keypress. )
   sum                (  Leave the result on the stack. )
;
```

The use of a local variable makes this version of summation much easier to read and understand.

Local Variable Operators

The two integer operators **to** and **+to** are used to change and add to the contents of local variables. The use of these two operators is demonstrated in the example above.

INTRODUCTION

The tFORTH Technical Reference Manual contains implementation-specific information about the tFORTH FORTH implementation. The following topics are covered:

* SYSTEM MEMORY USAGE
  Includes a ROM/RAM memory map of the entire system and a close-up memory map of the tFORTH RAM execution area.

* DICTIONARY STRUCTURE
  Detailed memory maps showing the layout of the tFORTH dictionary header and dictionary code areas.

* VOCABULARY STRUCTURE
  The search order and the active array, open and closed vocabularies, creating/removing vocabularies and the extant array.

* "RUNNING" tFORTH
  How interpret works.

* COMPILATION
  Structure of a dictionary header and code entry.
  Token threading versus address threading.

* EXECUTION OF TOKEN THREADED CODE
  Names of the various pointers and register usage.
  Nested execution levels.

* IMPLEMENTATION OF INTEGERS
  The 'iv' pointer and integer tables.
  System integers.

* IMPLEMENTATION OF LOCAL VARIABLES
  How local variables are compiled and executed.

* IMPLEMENTATION OF PROGRAM CONTROL STRUCTURES
  How program control structures are compiled and how they are interactively executed.

SYSTEM MEMORY USAGE

The System Memory Map on the following page gives a general overview of the memory layout of the 'V777' system, including ROM/RAM memory specifications. A second memory map, shown a couple of pages later, contains a close-up memory map of the tFORTH RAM area and lists the tFORTH words commonly used to traverse memory.

ROM

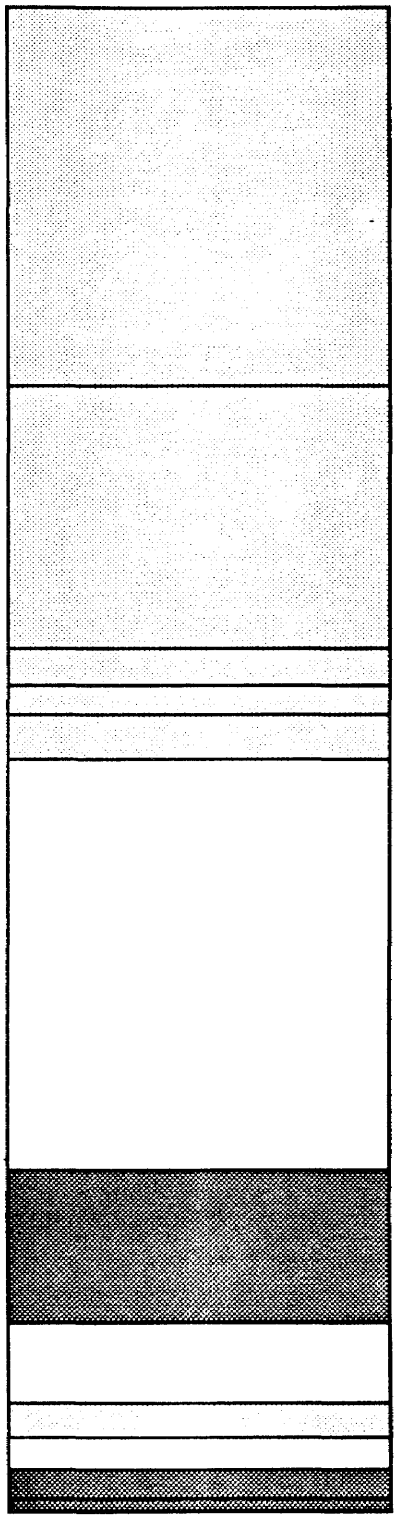
128K of ROM is located starting at address $00000. The lowest 1K bytes in the ROM are used to hold the system reset exception vector and the rest of the 68000 exception vector table (exception handling and interrupt handling routines are discussed later). The ROM'ed tFORTH and 'V777' editor code fill up the rest of the low memory ROM space.

RAM

The 'V777' system has 256K to 512K bytes of RAM located starting at address $400000. tFORTH uses half of the RAM area (128K bytes) for its alterable vocabularies and other dynamic data areas. The remaining 128K bytes of RAM are used by the editor. The boundary between these two areas is alterable.

System Memory Map

Addresses in hexadecimal



440000 Last RAM location

420000-43FFFF Editor RAM

4108A0-41FFFF 'tFORTH' dictionary

40EC00-41089F Token table

40A280-40EBFF Variable area

400000-40A27F Display memory

100000-15FFFF Spelling Checker ROM

040000-043FFF User Dictionary RAM

000400-01FFFF 'tFORTH' editor code

000000-0003FF 68000 Exception Vector Table

ROM RAM

The tForth RAM Memory Map

The following page contains a close-up view of the tFORTH RAM area.

Display Memory

The first 28K bytes of the tFORTH RAM area is used for display memory. The V777 screen has 672 pixels horizontally and 344 pixels vertically. ramstart will return the address of the start of the screen display memory when executed. The first byte in the display memory area corresponds to the 8 pixels on the far left of the top line on the screen.

Variable Memory

Following the display memory is the 4-5K area used for array variables and for the return and parameter stacks. sp0 will return the address of the base of the parameter stack when executed. sp@ returns the address of the top of the parameter stack when executed.

Token Table

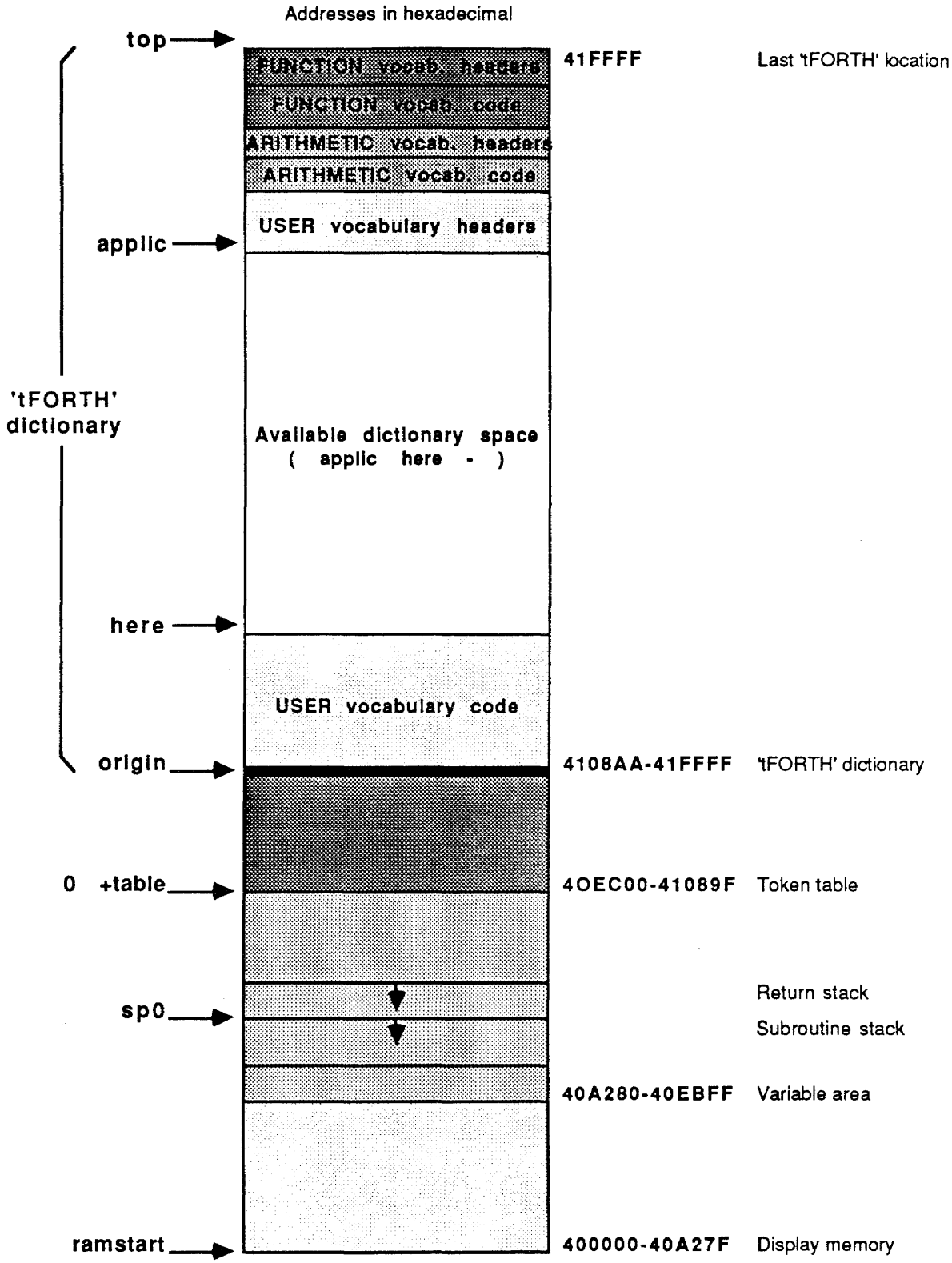
The address of the start of the 2K byte token table is obtained by asking for the address of the first token in the table:

```
hex
0   +table    .    40EC00
```

The token table contains an ordered list of the addresses of most of the words in the system. The layout and use of the token table are explained in the section covering the tFORTH compiler.

The rest of tFORTH's RAM allotment is used by the dictionary.

tForth memory Map (RAM)



Dictionary

origin will return the address of the start of the tForth dictionary space when executed. **top** returns the address of the first byte BEYOND the end of the tForth dictionary space.

Initially, the tForth dictionary contains three vocabularies of words. The approximate locations of the header and code areas (explained later) for each of the three vocabularies are shown in the diagram. The tForth word **here** is used to return the address of the next available byte location in the code area of the current 'open' vocabulary. applic will return the address of the next available byte in the header area of the current 'open' vocabulary (+1). The following calculation is used to determine the amount of remaining tForth dictionary space:

```
hex
applic   here   -   .   186C9   ok
```

THE tFORTH DICTIONARY STRUCTURE

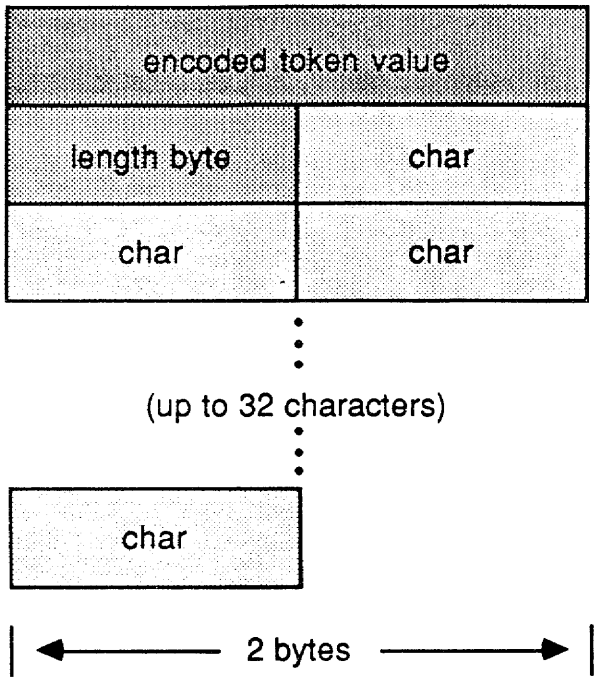
As the previous memory map illustrates, the tForth dictionary space contains two types of data areas: the 'header' area and the 'code' area. The header area is where the header portions of all words in a vocabulary are stored. The code area is where the code portion of all words in a vocabulary is stored. Each vocabulary is given its own header and code area.

The Dictionary Header Area

The diagram on the following page contains a close-up view of an dictionary header area. The first four bytes in a dictionary header area contain a 32-bit value which indicates how many bytes of individual header entries this header area currently contains.

The first header entry in every vocabulary belongs to an invisible word used to mark the start of the individual dictionary header entries. This 'stub' word is 1 character in length with a name of 'null' . The ASCII code for the null character is 00. The entry for the stub word is not a complete dictionary header, it contains only the length byte and the single character in the name.

The last header entry in every vocabulary belongs to another invisible word named 'del' (ASCII code = hexadecimal 7F). 'del' is used to mark the end of the dictionary header entries. The 'del' entry is a complete header entry. The encoded token value used in the 'del' header entry is the highest possible encoded token entry.



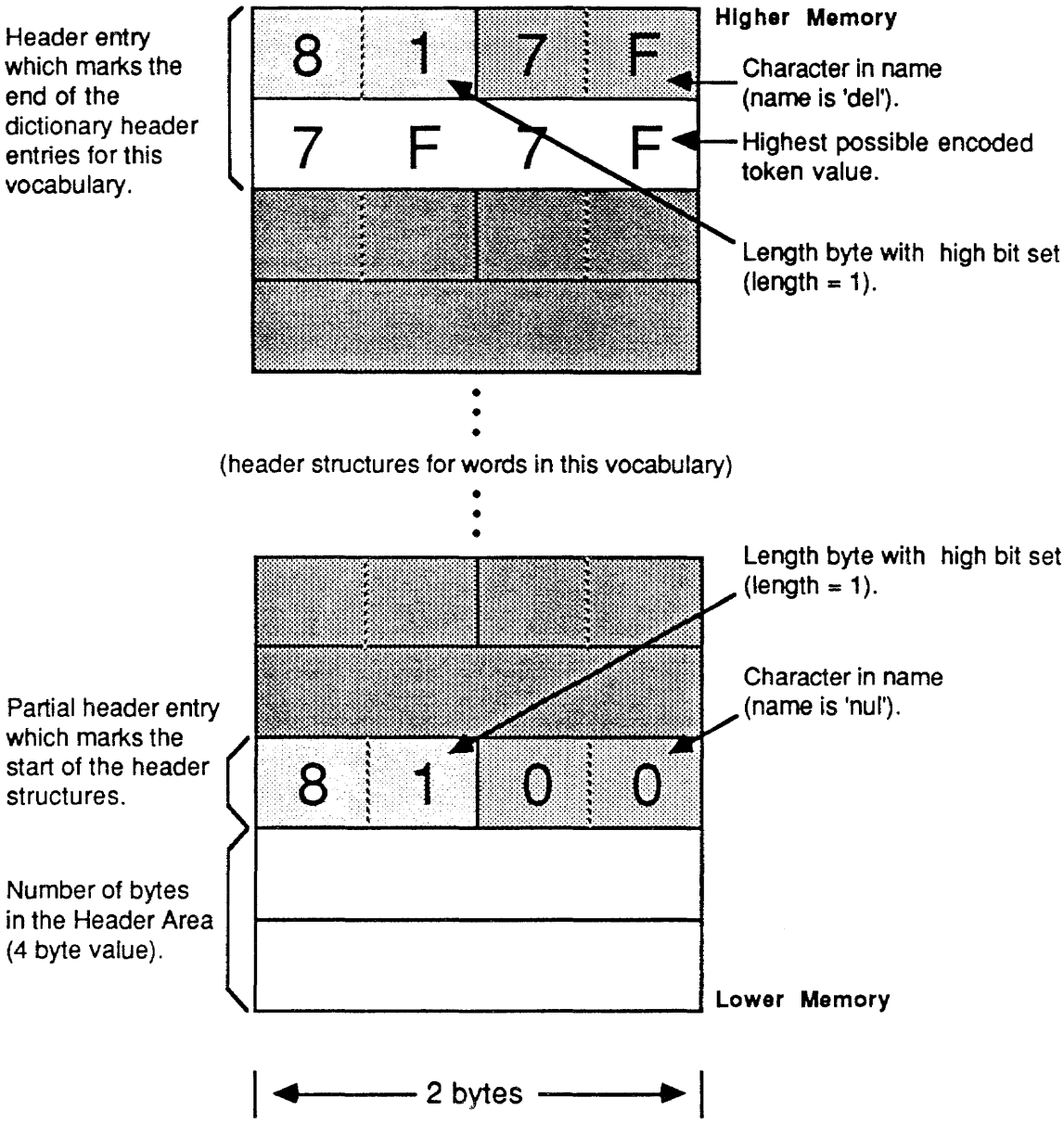
Structure of a Dictionary Header

Adding Entries to the Header Area

The dictionary header area grows downward in memory. Any new entries added to this vocabulary area will be placed between the 'nul' and 'del' header entries (the 'del' header entry is moved to a lower memory location to accommodate the new entry).

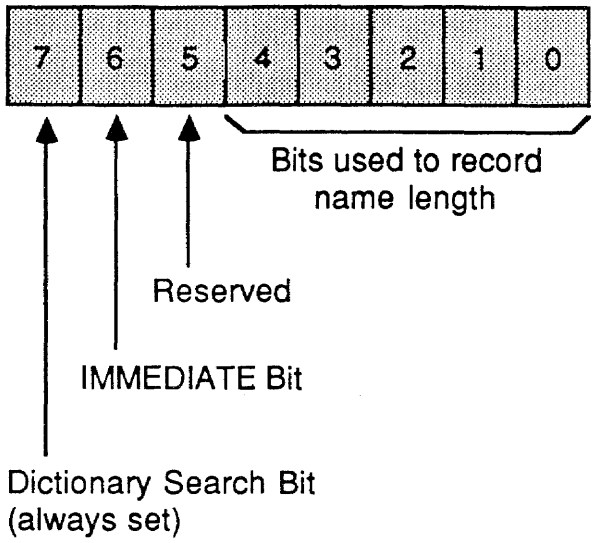
Structure of an Individual Dictionary Header Entry

The structure of a tForth dictionary header entry is shown in the diagram on the following page. The first two bytes in the header entry contain an encoded version of the token value assigned to the word. The third byte in the header structure is a length byte. The bytes following the length byte contain the ASCII codes for the characters which make up the word's name.



Close-Up of the Dictionary Header Area

The diagram on the following page gives a close-up view of the length byte in a header entry. Bit 7 is used during dictionary searches, bit 6 is used to mark IMMEDIATE words, and bit 5 is reserved for future use. Bits 4 through 0 are used to record the length of the word's name. Since only 5 bits in the length byte are available for recording the length of a word's name, only the first 32 characters in a word's name are significant.



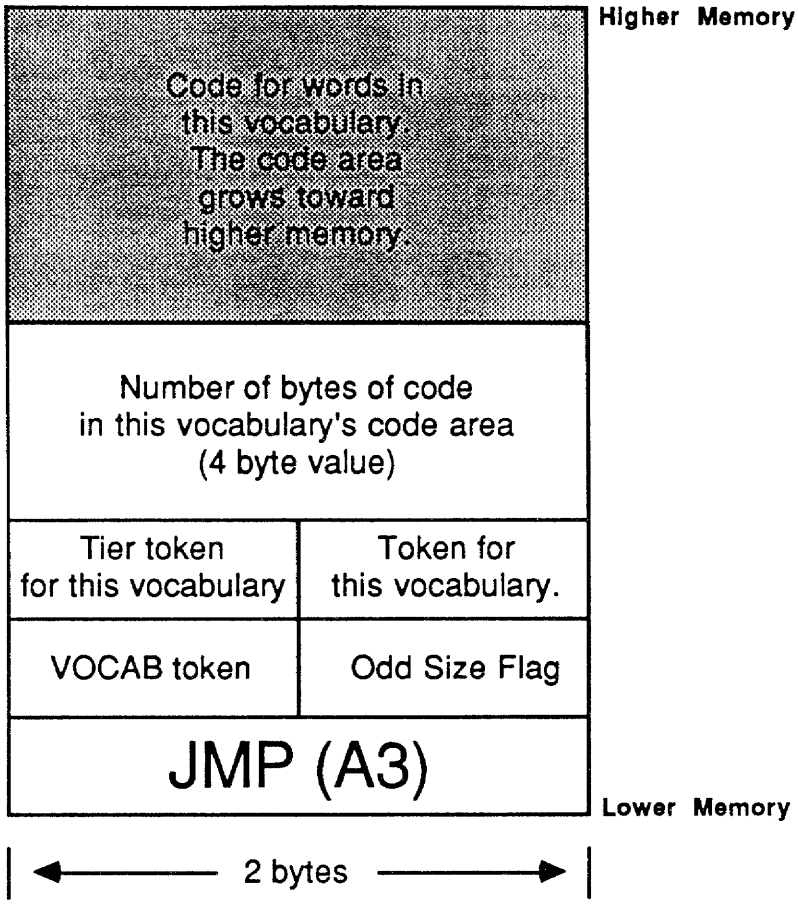
Structure of the Length Byte

The Dictionary Code Area

The following diagram shows a close-up of the dictionary code area. 10 bytes of data are located at the start of every tForth code area. The first two bytes contain an 68000 assembly language 'JMP (A3)' instruction. The next byte is the actual token for the VOCAB defining word. The byte is a flag which indicates whether the code space contains an odd or even number of bytes of code. The next two bytes contain the tier and token information for the vocabulary to which this code area belongs. The final four bytes of this 10 byte data structure are used to hold a 32-bit value which indicates how many bytes of code this code area currently contains.

Adding Code to the Code Area

As new definitions are added to the system, the code portions of the new words which belong to this vocabulary will be placed in successive memory locations in the code area. The code area size field will be incremented accordingly as code is added. The code area grows toward higher memory locations. The odd size flag will be set whenever the vocabulary is closed if the vocabulary contains an odd number of code bytes. The words which reopen a vocabulary will check a vocabulary's odd size flag whenever the vocabulary is opened.



Close-Up of the Dictionary Code Area

VOCABULARIES

The words in a FORTH dictionary are usually subdivided into several smaller groups of words called 'vocabularies'. The 500 or so words in the tForth dictionary are located in four different vocabularies:

forth vocabulary	Contains all of the 'standard' FORTH FORTH words supported by tForth and all tForth FORTH extension words. These words are located in ROM and may not be altered (the token table may be 'patched' to point to a new RAM definition of a ROM word if necessary)
user vocabulary	The user vocabulary is used to hold the user's definitions.
arithmetic vocabulary	Contains the code which corresponds to the user's calculations in the text.
function vocabulary	Contains the code which corresponds to the functions to be used in calculations in the text.

There is also an invisible vocabulary which is used to hide all of the editor words.

Vocabularies help arrange the words in the dictionary into smaller groups of related words. During compilation, the programmer can help the compiler by specifying in which of the vocabulary subsets of words the next word, or group of words to be compiled, is located. This can speed up the compilation process since the compiler performs less time searching the dictionary. How to specify a 'vocabulary search order' is discussed next.

The Vocabulary Search Order

The vocabulary search order determines which of the available vocabularies in the system are searched whenever the compiler or interpreter need to find a word. A list of the vocabularies contained in the current search order is kept in an array named 'active'.

The Word active

active is a tForth word which returns the address of the start of the active array when executed. The active array is 32 bytes in length. The first entry in the active array contains the token corresponding to the vocabulary which is first in the search order. The second entry in the active array contains the token corresponding to the second vocabulary in the search order, and so on. Up to 16 vocabularies may be included in the search order list at one time. The active array is traversed by **find** until

either 16 vocabularies are searched or until a word-length (16-bit) value (hexadecimal FFFF) is encountered.

Specifying the Search Order

A vocabulary may be placed first in the search order by executing its name. If the vocabulary was already included in the search order, its token will be moved from its current position in the active array to the first spot in the array and the rest of the array will be adjusted to close the gap. If the vocabulary is new to the search order its token will be inserted at the start of the array.

Adding New Words to a Vocabulary

New definitions may only be added to the vocabulary which is currently 'open'. In the tForth RAM Memory Map diagram the user vocabulary is the current open vocabulary. When a vocabulary is 'open' there is a memory gap between the top of the vocabulary's code area and the bottom of the vocabulary's header area. When new definitions are added to the open vocabulary, the memory required for the new definition's code and header is taken from the 'open' pool of memory.

The Word **addto**

The tForth word **addto** is used to open a vocabulary:

addto <name>

Only one vocabulary may be open at once. **addto** will close the current open vocabulary before it opens the vocabulary specified by <name>.

The diagram on the following page shows how the vocabulary closing and opening process works. A vocabulary is closed by closing the gap between the vocabulary's header and code areas in memory. A vocabulary is opened by creating a gap between its code and header areas.

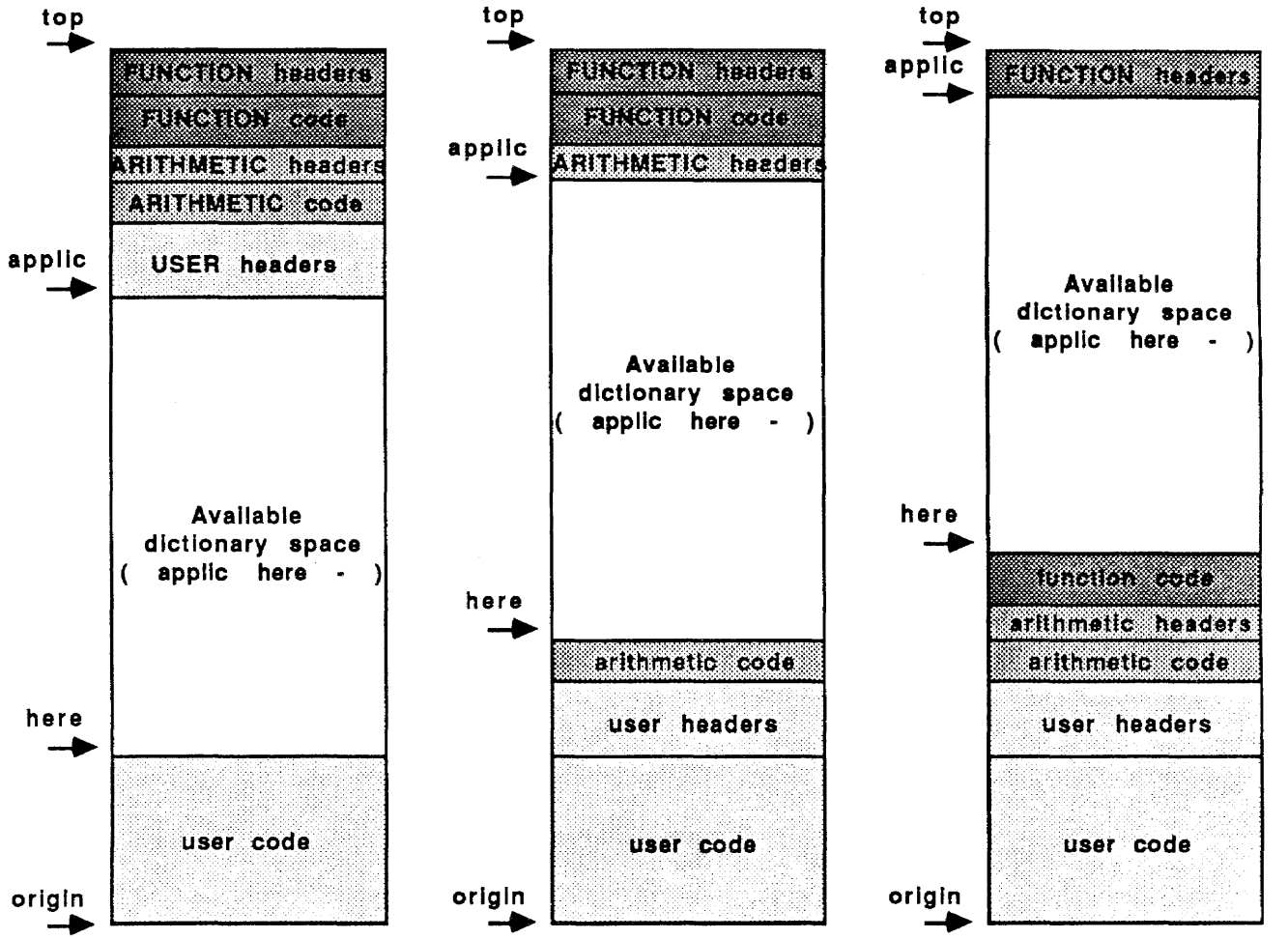
Creating New Vocabularies

The Forth-83 defining word **VOCABULARY** is used to add new vocabularies to the system. The following actions are used to add a new vocabulary:

1. The new vocabulary's header and code are placed in the header and code areas of the currently open vocabulary (the new vocabulary's parent vocabulary).

2. The contents of memory between 'applic' and 'top-1' are shifted downwards by 20 bytes to make room for the 10 bytes of data stored at the start of every dictionary header area and for the 10 bytes of data stored at the start of every dictionary code area. (cont.)

Opening and Closing Vocabularies



" addto user "

user vocabulary is open.

" addto arithmetic "

arithmetic vocabulary is open.

" addto function "

function vocabulary is open.

3. Finally, the new vocabulary's token and its parent token are stored in the extant array.

The Extant Array

Each vocabulary in the dictionary has its token and the token of its parent vocabulary in an array called 'extant'. Each entry in the extant array is 4 bytes in length and consists of a 2-byte token for a vocabulary followed by a 2-byte token for the vocabulary's parent vocabulary. The extant array is 64 bytes long and therefore has enough room for 16 child-parent vocabulary pairs. Execution of the word **extant** will return the address of the extant array.

The extant array is used by tForth words which must remove vocabularies from the system.

The first part of the tForth technical manual dealt with the data structures and memory usage and layout of the tForth system. This part of the technical manual will cover the dynamic functioning of the tForth system.

The Word **quit**

The main word which 'runs' FORTH is **quit** . These are the basic actions performed by **quit** :

```
: quit  ( - )
        begin
                ( clear the return stack )
                ( get a block of user input text )
                ( interpret the user input text )
                ." ok"  cr
        again ;
```

Clearing the return stack is simply a matter of returning the return stack pointer to its base position. **query** , which was discussed previously in the string I/O section, is used to get a line of input text from the user. The most important word used by **quit** is **interpret** . **interpret** is responsible for parsing the input stream.

The tForth definition for **interpret** is shown on the following page. **interpret** is passed the address and length of the input text on the parameter stack. In the first two lines of **interpret** the end address of the input text is stored in the system integer **limit** and the start address of the input text is stored in the system integer **in** . The **in** integer is used to mark **interpret**'s progress through the input text string.

These are the actions which occur in the main 'begin...while...again' loop in interpret :

1. **word** is used to extract the next word (sequence of characters delimited by spaces or tabs) from the input text. word will leave the address of the extracted word in the system integer **str**, the length of the extracted word in **len** , reposition the **in** integer to point to the next character to be examined in the input text.

```
: interpret ( a l - )
        over + limit to
        in to
        begin                       ( begin the interpretation loop )
                word                ( grab a word from the input text )
        len
        while                       ( while the length is nonzero )
                locals              ( is this word a local variable? )
                if                  ( IF it is, perform special local variable )
                        doloc       ( actions, described later )
                else
                        -1
                then

                if                  ( if the word is not a local variable, )
                                    ( try to find it in the dictionary )
                        str len find
                        ?dup
                        if          ( if the word was found in the dictionary... )
                                    0< state nesting or and
                                    if ( and if the system is in the )
                                       ( compiling state, compile the word )
                                       compile,
                                    else ( otherwise, execute the word )
                                       sw execute sw
                                    then
                        else
                                    ( if the word was not found in the )
                                    ( dictionary, try to convert it to a number )
                                    str len base number
                                    if ( if it is a number... )
                                       state nesting or
                                       if
                                    ( and compiling state is on )
                                    ( compile # as a literal )
                                          [compile] literal
                                       else
                                          ( leave # on param stack )
                                       then
                                 else ( if its not a #, check for targeting )
                                       targeting
                                       if
                                          ( if targeting is occurring... )
                                          forward
                                       else
                                          ( leave the loop and abort )
                                          leave
                                       then
                                 then
                        then
                then
        ?stackerr                   ( check the parameter stack state )
        again
        len abort" can't use" ; ( can't use this word )
```

2. If **word** is able to extract a word from the input text, it will drop into the while section of the **begin...while...again** loop. The first test performed in the while section is a test to see if the extracted word is a local variable. If the word is a local variable, local variable type actions occur (described later) and a false (0) flag is left on the parameter stack. If the word is not a local variable, a true (nonzero) flag is left on the stack.

3. If the word was not a local variable, interpret checks next to see if the word can be found in the dictionary using the current search order. find is used to locate words, specified by the address and length of their name strings, in the dictionary. If find finds a word it returns the token for the word and a true (nonzero) flag on top of the parameter stack. If the word found is an 'immediate' word the flag returned will be a '1'. Otherwise, the flag returned will be a '-1'. If find cannot located a word, it returns a false (0) flag.

4. If the word was found in the dictionary, **interpret** must next decide whether the word should be compiled or executed. Two tests must be true in order for the word to be compiled. First, the word found must not be an immediate word. Second, one or both of the two system integers **state** or **nesting** must contain a nonzero value. If the **state** system integer contains a nonzero value it means the system is in the compilation state. If the **nesting** system integer contains a nonzero value it means the system is currently compiling the temporary code required for interactive execution of program control structures (discussed later). So, if the word found is not immediate AND if the system is either compiling real definitions OR compiling temporary code for interactive execution of program control structures, the token for word will be compiled.

5. If the conditions for compilation are not met, **execute** will be used to execute the word corresponding to the token.

6. If the extracted word was not found in the dictionary, **interpret** will next try to convert the string to a number. If the string can be converted to a number, and the system is either in the compiling state OR compiling temporary code, the number will be compiled into the current definition as a literal.

7. If the string was converted to a number and the system is not compiling, the number will be left on the parameter stack.

8. If the extracted word was not a word in the dictionary and could not be converted to a number **interpret** will check to see if the system is currently in the target compiling state by checking the contents of the **targeting** system integer for a nonzero value. If target compilation is occurring (target compilation will be discussed in a separate document), the

extracted word will be compiled into the target system image under construction. Otherwise, **leave** will be used to exit the loop, **interpret** will **abort** , and an error message will be issued.

Now that the general actions of a running tForth system have been described, individual aspects may be discussed in detail. The following aspects of the tForth system will be covered in the final sections of this technical reference manual:

* The Basics of tForth Compilation
* Execution of Token-Threaded Code
* The Implementation of tForth's Integers
* The Implementation of tForth's Local Variables
* The Implementation of tForth's Program Control Structures

THE BASICS OF tFORTH COMPILATION

Structure of a Dictionary Entry

In the "Under the Hood" chapter of Leo Brodie's book Starting Forth, the structure of a dictionary entry in an address-threaded FORTH implementation is described. Since the original implementation of FORTH, and most FORTH implementations for many years after, were address-threaded implementations, the address-threaded model of FORTH is generally considered to be the 'standard'. In recent years, FORTH implementors have devised many different threading schemes. The token-threading scheme used in tForth is one of the most popular of the new FORTH threading schemes due to its conservative use of memory. In this section, tForth's token threading scheme will be explained by comparing it with the 'standard' address threading scheme as described in Starting Forth.

The diagrams on the following page show the dictionary entries which would be created for the definition below in a 32-bit address-threaded implementation and in the 32-bit tForth token-threaded implementation:

```
:  newword  ( n -  )
     3  *  dup  +  .  ;
```

Certain areas in the diagrams have been given the following labels since, according to Starting Forth, all Forth definitions share these common parts:

DEFINITION HEADER FIELDS	DEFINITION CODE FIELDS
name field	code pointer field
link field	parameter field

The 'definition header fields' are used to find definitions in the dictionary (in order to execute or compile the definition). The 'definition code fields' are used when a definition is executed. In the tForth implementation the definition header fields and the definition code fields are stored in different memory areas. The token field and the token table, which will be described in more detail later, are used to link a definition's header and code fields together.

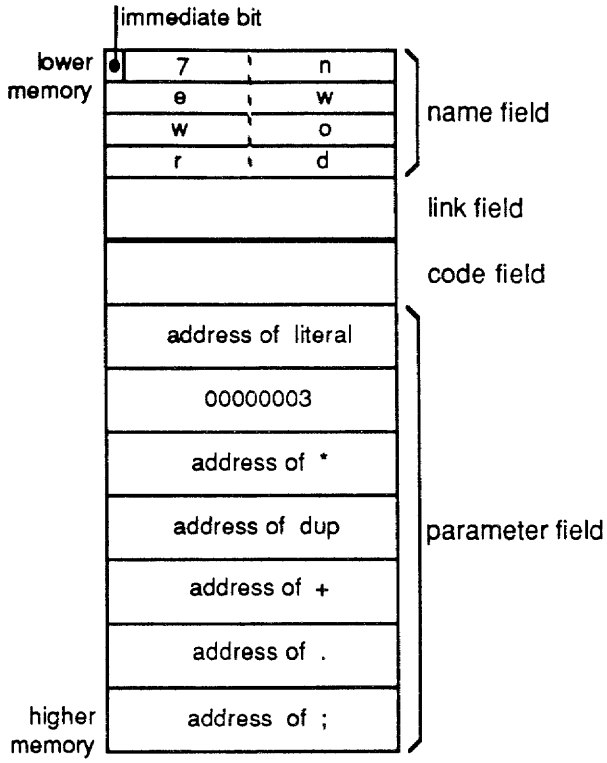
The 'code pointer' field is used to specify where the 'code' for the definition is located. The parameter field contains either data, addresses (in an address threaded implementation), tokens (in a token threaded implementation), or machine code instructions.

Dictionary Entries: Address Threaded versus Token Threaded

: newword (-) 3 * dup + . ;

Address Threaded

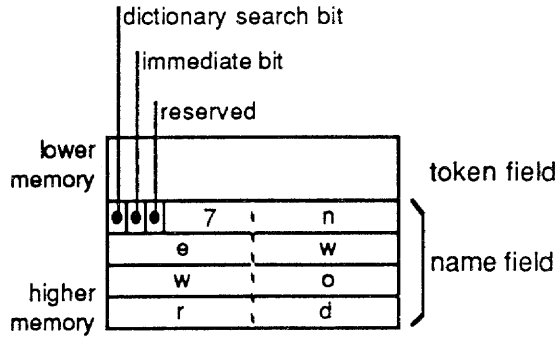
Definition Header Fields:



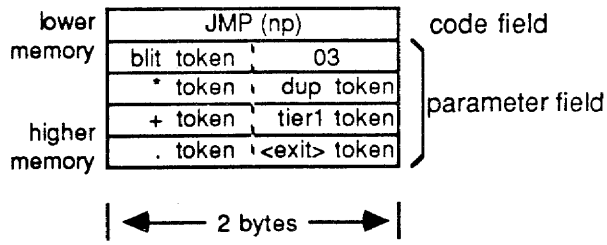
Definition Code Fields:

Token Threaded

Definition Header Fields:



Definition Code Fields:



In an address threaded implementation, the dictionary is organized into groups of linked lists (one for each vocabulary). The 'link' field for each word in the dictionary points to (holds the address of) the previous dictionary entry in a particular vocabulary list.

In the tForth implementation, the link field has been eliminated because the words in each vocabulary are arranged alphabetically.

Examining the Definition Header and Code Areas

The tForth compilation word **n'** ('n-tick') can be used to find the address of the definition header for a word:

```
n'  newword  10  dump
5FFBA  07 2C 87 6E 65 77 77 6F 72 64 7F 7F 81 7F 4E D3 .,.newword....N.
```

The two leftmost bytes are the encoded token value for **newword**. The next byte is the length byte for **newword**'s name. The length byte shows that there are 7 characters in 'newword'. The length byte looks like '87' because the most significant bit is set for dictionary searches (described later). The next seven bytes contains the ASCII codes for the characters in the word **newword**. **n'** can only be used on words in the current open vocabulary.

The tForth compilation word **c'** ('c-tick') can be used to find the address of the code for a definition:

```
c'  newword  10  dump
478EC  4E D3 0A 03 77 2D 54 01 0C 10 00 00 00 1A 81 00  N...w-t.........
```

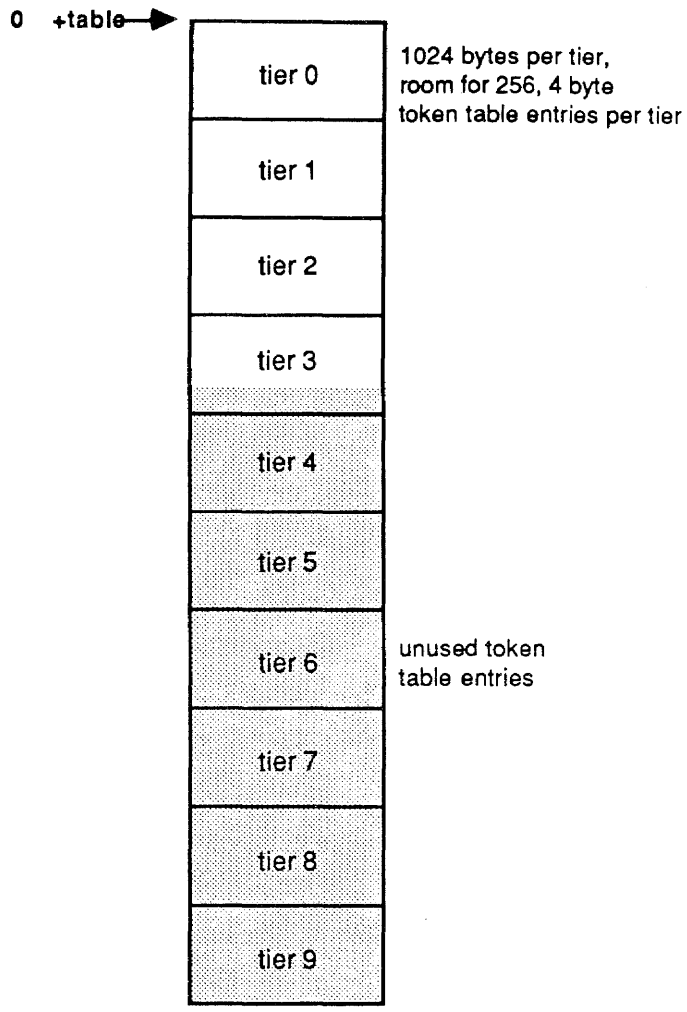
The contents of the code field for **newword** are located in the two leftmost bytes in the display. The parameter field contents occupy bytes 3 through 10 in the display.

Tokens and the Token Table

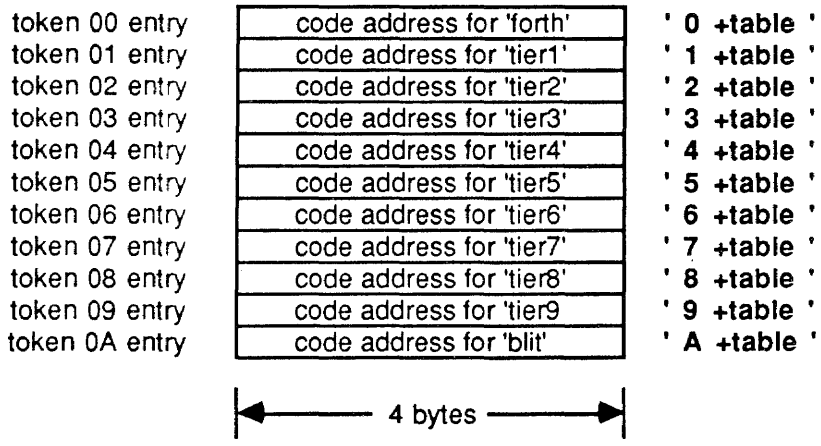
A token is a one byte number whose value can be any number between 0 and 255 decimal. Each word in tForth is represented by either 1 or 2 tokens. The token field in a tForth definition header contains an encoded version of the definition's token value. To find the code which corresponds to a particular definition header, the encoded token value is decoded to its corresponding token value and then multiplied by four (since each entry in the table is 4 bytes long) and used as an offset into the 'token table' (see the diagram on the following page). The token table is a table which contains the 32 bit code addresses for all definitions known to tForth.

Token Table

Tiers:



Close up of the start of tier 0:



Useful Token-Related Words

The words ' ('tick') and **[']** ('bracket-tick-bracket') can be used to find the token value for a word. ' is used outside of colon definitions and **[']** is used within colon definitions. The word **name** will take a token value and display the name of the corresponding FORTH word. The word **encode** encodes a token value (for placement into the token field of a word's header) and the word **decode** decodes an encoded token value (so the decoded token can be used as an index into the token table). The word **+table** takes a decoded token value and returns the address of the location in the token table where the corresponding code address is stored.

```
' newword . 3AC        ( get newword's token value )
3AC name newword       ( find which FORTH word belongs to )
                       ( the token 3AC )
3AC encode . 72C       ( encode the token value )
72C decode . 3AC       ( decode the token value )
3AC +table . 46EB0     ( get the address of newword's entry in )
                       ( the token table )
46EB0 @ . 478EC        ( fetch the contents of newword's entry )
                       ( in the token table: the address where )
                       ( the code for newword is located. )
c' newword . 478EC     ( compare the address returned by c' )
                       ( with the address returned above, they )
                       ( are the same )
```

Note that the encoded value of **newword**'s token is the same value stored in **newword**'s token field (shown above in the **n'** example). Also note that the code address returned by **c'** is the same as the code address stored in **newword**'s token table entry. (Note, try using **name** to determine which FORTH words belong to the tokens in the parameter field area of the code portion of **newword**. See the **c'** example above.)

Tiers

Since a byte size token can only assume 256 distinct values, the use of single tokens only would limit the system to a maximum of 256 words. To overcome this limitation, the 'tiered' arrangement shown on the diagram was introduced. Each of the 10 tiers can hold 256 code addresses. This means the system can potentially accomodate 256*10=2560 words, although currently there are only about 1000 words. To locate a code address stored in tiers other than the base tier (tier0) requires the specification of the tier level and the token value.

Example: Anatomy of **newword**

Let's examine the tokens compiled into the **newword** definition:

```
c' newword 10 dump
478EC  4E D3 0A 03 77 2D 54 01 0C 10 00 00 00 1A 81 00  N...w-t.........
```

4ED3 Contents of code field, to be discussed later.

0A Token for blit
03 Literal data used by blit
77 Token for *
2D Token for dup
54 Token for +
01 Tier token 1, indicates that the following token value is located in tier 1 in the token table
0C Token for . , located in tier 1
10 Token for **<exit>**

References to all words except for . were compiled as single-byte tokens. Words compiled as single byte tokens are located in tier0 in the token table. The word . is located in tier1 in the token table. Words located in tiers other than tier0 will be compiled as 2 byte token combinations. Token values 01 , 02 , 03 , 04 , 05 , 06 , 07 , 08 , and 09 are all 'tier tokens' which correspond to tier1 , tier2 , tier3 , tier4 , tier5 , tier6 , tier7 , tier8 , and tier9 in the token table.

Compilation Size Considerations

The compiled code for **newword** required 10 bytes of code space and 28 bytes (7 tokens * 4 bytes per token table entry) of token table space. This means the token threaded version of **newword** requires a total of 38 bytes of memory.

In an address threaded version of **newword** each of the 5 words referenced by **newword** would cause a 32 bit, or 4 byte, address to be compiled. This means the compiled code for an address threaded version of **newword** could require up to 24 bytes of code space (including the space required for the literal data).

If 5 words very similar to **newword** were to be compiled in the tForth token threaded system the memory requirements would be 50 bytes of code space (10*5=50) and still only 28 bytes of token table space for a total of 78 bytes of memory.

The same 5 words compiled in an address threaded system would require 120 bytes of code space (24*5=120) because in each definition the complete 4 byte addresses for each referenced word would have to be compiled.

Overall, tForth's token threaded approach saves code space because the memory intensive data, the 4 byte code addresses for each word in the dictionary, are only located in one spot in memory - in the token table. Each compiled reference to a word only generates a one or two byte token in the code space. In an address threaded version the 4 byte code address for a word is repeated each time a reference to the word is compiled.

The Compilation Process

To study the compilation process, we will reconstruct the tForth and interpreter's and compiler's actions during compilation of **newword** . Here, once again, is the definition of **newword** :

```
: newword ( n - ) 3 * dup + . ;
```

For the sake of simplicity, assume the line containing the definition of **newword** has just been sent to tForth. The FORTH interpreter will start "walking" through the input line, separating out words (sequences of characters delimited by spaces or tabs) and executing them. The first word encountered by the interpreter will be the word : . This is what happens when : is executed:

: 1. Calls **create** . **create** aligns the **here** pointer (which points to the location in the code space where the code for **newword** will be placed) on an even word boundary. Next, **create** uses **word** to get the next word from the input stream, which will be the name to be assigned to the new definition ("newword", in this case). Finally, **create** assigns a token to the new definition and creates a new dictionary header.

2. Calls] .] saves the current contents of the **nesting** and **state** system integers and places zeros in both integers. The purpose of the **nesting** system variable will be discussed later in the technical discussion on program control structures. **state** is the system integer used to record the current 'state' of the system. If **state** holds a '0', the system is in the compiling state. If **state** holds a '1', the system is in the interpreting state. The net effect of] is to place the system in the compiling state.

3. The final important action of : is to lay the first two bytes of code into the code area for **newword** . These two bytes contain the opcode for the assembly language instruction 'JMP (np)' (4ED3) or, in human language, "execute the **nest** code definition". All definitions created by the defining word : begin with a 'JMP (np)' instruction.

At this point, execution of : terminates and control returns to the interpreter. The interpreter grabs the words **3** , ***** , **dup** , **+** , and . from the input line. Since the system is now in the compiling state, due to the actions of **]** , the the tokens for these words are compiled rather than executed. The final word on the input stream is ; . If the definition just compiled contains local variables, ; will compile the token for **<;lp>** into the definition. Otherwise, ; will compile the token for **<;>**. The implementation of local variables will be discussed in the technical discussion on local variables. At this point the interpreter has reached the end of the input line and **newword** has been compiled.

EXECUTION OF TOKEN-THREADED CODE

How is a tForth word executed? Explanation of the tForth execution process requires that the tForth 68000 register usage is discussed and that some terminology is established.

tForth Register Usage

tForth uses 10 out of the 16 available 68000 registers to hold addresses it requires as it operates.

REGISTER	SYMBOL	USAGE
D7	bp	Holds the address of the base of the token table. "base pointer"
D6	iv	Holds the address where the value of the current integer is stored. "integer value"
D5	sa	Holds the address of the start of the definition currently being executed. "start address"
D4	ct	Holds the address of the token table entry for the definition currently being executed. "current token"
A7	sp	Parameter stack pointer.
A6	rp	Return stack pointer.
A5	ip	Holds the address of the next token to be executed in the current definition. "interpretation pointer"
A4	nx	Holds the address of the code for **next** . "next pointer"
A3	np	Holds the address of the code for **nest** . "nest pointer"
A2	vp	Holds the address of the run-time code for **integer** .

The FORTH instructions generated by the tForth compiler are not executable 68000 instructions. The tForth compiler creates a virtual processor, with its own instruction set, which runs on top of the 68000 microprocessor. Since all 68000 instructions are at least 2 bytes in length and the tForth tokens are at most two bytes in length (the most popular tForth words are 1 byte tokens), the instruction set for this virtual processor allows tForth to produce more compact code than it would if it always used the 68000 instruction set. Only the lowest level tForth words, the code definitions, consist of actual 68000 machine code instructions.

The Word **execute**

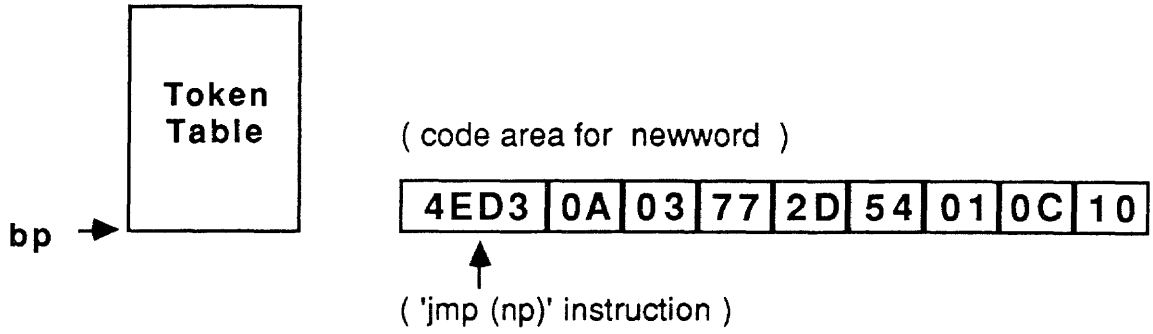
The word **execute** is used by **interpret** to execute FORTH words. **execute** executes the word corresponding to the token passed to it. To demonstrate how tForth words are executed, the steps taken by **execute** as it executes the word **newword** will be examined:

```
: newword  ( n - )          ( here is the definition of  newword )
     3 * dup + . ;          ( once again )
     2 newword C            ( test  newword  out )

     ' newword . 3AC        ( get the token for  newword  )
     2                      ( newword expects to be passed a )
                            ( number on the stack when it executes )
     3AC execute C          ( use  execute  to execute  newword  )
```

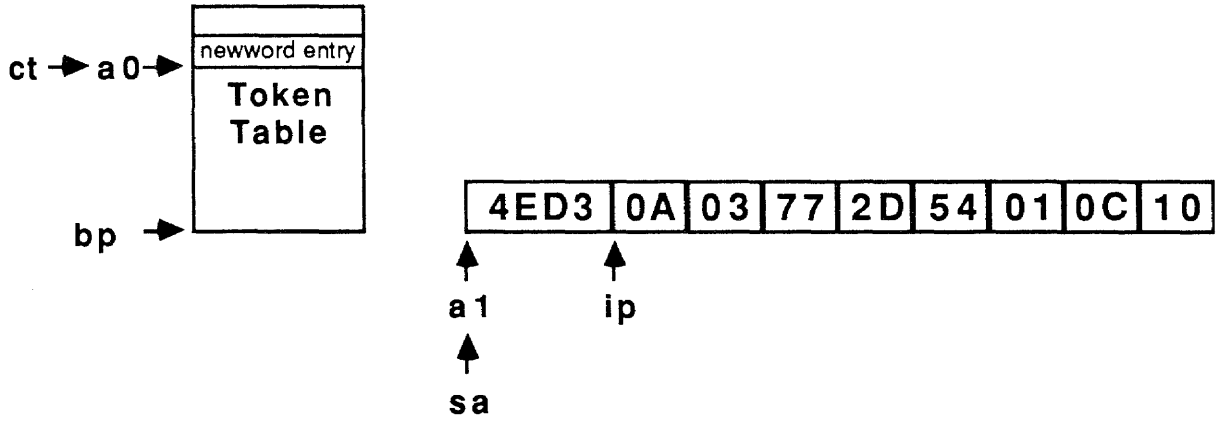
The code area for the **newword** definition is shown on the following page. The locations of some of the pointers mentioned above are shown in the positions they would hold just before the word **newword** is executed.

Token Threaded Execution

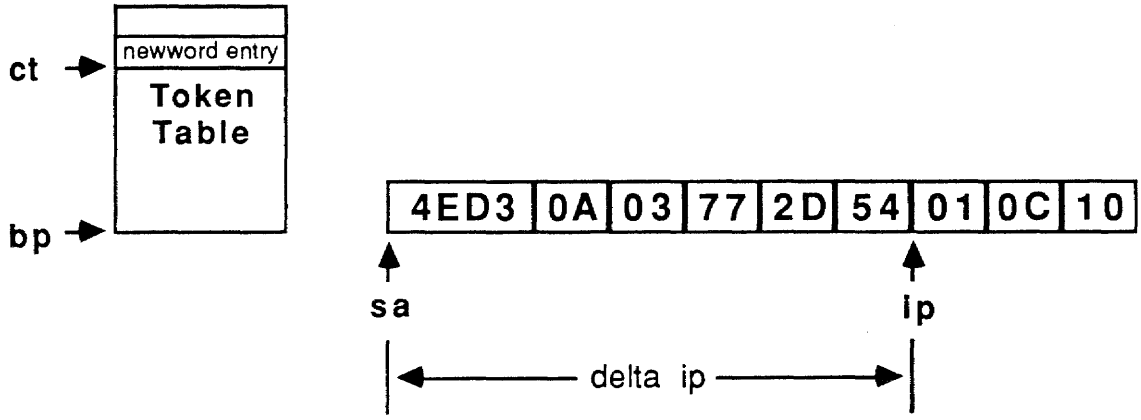
Before execute :



After execute :



Before . :



Here is the code definition for **execute** :

```
      code  execute  (  n  -  )
            sp  )+  d0     move,

            ( code which checks to see if the token on the stack )
            ( belongs to a system integer goes here )
            if,
                  ( special system integer )
                  ( handling code, which will )
                  ( be explained in the next )
                  ( section, goes here. )
            then,

( 1 )       bp  .b  clr,          ( clear the lowest byte of the )
                                  ( base pointer )
( 2 )       bp  a0  move,         ( move a copy of the base pointer )
                                  ( to the A0 register )
( 3 )       d0  a0  .w  add,      ( multiply the token value times 4 )
( 4 )       a0  a0  .w  add,      ( and add the result to the copy )
( 5 )       a0  a0  .w  add,      ( of the base pointer to calculate )
                                  ( the address of this token's token )
                                  ( table entry )
( 6 )       a0  )  a1  move,      ( put the contents of this token's )
                                  ( token table entry-the code addr )
                                  ( for the word corresponding to the )
                                  ( token-in the A1 register )
( 7 )       a1  )  jmp,           ( jump to the first instruction in the )
                                  ( code area for this word )
      ;c
```

The first part of **execute** checks to see if the token to be executed belongs to a system integer. Execution of system integers will be discussed later. The second half of **execute** contains the code responsible for the execution of tForth words. Since **newword** is not a system integer, the code in the second half of **execute** will be used.

In the line marked line (1) above, the lowest order byte of the base pointer, which points to the start of the token table in RAM, is cleared. During tForth execution, the lowest byte in the base pointer is altered (as will be shown later). Clearing the lowest byte of the base pointer actually puts the base pointer back in its correct position.

In line (2), a copy of the corrected base pointer is placed in the A0 register.

In line (3), the word length decoded token value is added to the base pointer address.

In lines (4) and (5) the decoded token value ($3AC for **newword**) is added twice to the lower word of the copy of the base pointer. This has the net effect of multiplying the token value by 4 (4 bytes per token table entry) and adding the result to the

token table base address. The resulting address in the A0 register is the address of the token table entry for the word to be executed (the address of **newword**'s token table entry). Note that all of the addition operators use the '.w' (word length) suffix so that the upper word of the base pointer address is not affected by the addition operation.

In line (6) the code address stored in the token table entry (the code address for **newword**) is placed in the A1 register.

In line (7) program execution is vectored to the code area for **newword** by means of a 'JMP (A1)' or, "jump to the code for this word" instruction.

Nesting

As the diagram shows, the first field in **newword**'s code area is a 2-byte code field. The instruction in the code field is a 'JMP (np)' instruction. **newword** , and all other tForth definitions created by the defining word : will have a 'JMP (np)' instruction in their code field. In tForth the np (A3) register is used to hold the address of the tForth nesting routine (described below). The nesting routine is always the first routine run whenever a colon definition is executed.

In the 68000, each time a 'JSR' (jump to subroutine) instruction is executed, the address of the instruction at which execution should resume after the subroutine completes execution (the address of the instruction which immediately follows the 'JSR' instruction) is placed on the 68000's system stack. The tForth processor is similar to the 68000 microprocessor in that 'return information' is saved away each time a 'jump' down to a lower level FORTH word occurs. The process of saving FORTH return information and 'jumping' down to a lower FORTH execution level is called "nesting" (don't confuse this term with the tForth system integer named **nesting**). In FORTH, the nesting process stops when a directly executable code definition is reached.

The tForth nesting routine is used to make the transition between tForth execution levels. Each time a tForth word references another tForth word which is not a code definition, the system drops down to another execution level ('nests'). Any execution level can be completely described by two pieces of information: the delta distance from the start address of the definition currently being executed (kept in the sa register) to the address of the token currently being executed (see the diagram), and the address of the token table entry for the definition currently being executed (kept in the ct register). Here is a listing of the tForth nesting routine:

```
        frag  .nest  to    ( create a code fragment, store the address of )
                            ( the code fragment into the  .nest  system integer )
( 1 )         sa  ip  .w  sub,        ( calculate the distance between )
                                      ( the start of the current code )
                                      ( area to the token in the code )
                                      ( area which is currently being )
                                      ( executed )
( 2 )         ip  rp  -)  .w  move,   ( save this delta away )
( 3 )         ct  rp  -)  .w  move,   ( save the lower word of the )
                                      ( current token pointer away )
( 4 )         a1  2  )d  ip  lea,     ( put the address of first )
                                      ( token to be executed in the )
                                      ( lower level definition in the )
                                      ( instruction pointer )
( 5 )         a1  sa  move,           ( A1 points to the start of the code )
                                      ( area of the lower level  definition )
( 6 )         a0  ct  move,           ( A0 points to the token field )
                                      ( entry for the lower level defn. )

              ( ** the 'next' code fragment starts here ** )

( 7 )         ip  )+  bp  .b  move,   ( place the token pointed to by )
                                      ( the ip pointer into the lower byte )
                                      ( of the base pointer )
( 8 )         bp  a0  move,           ( calculate the address of the )
( 9 )         a0  a0  .w  add,        ( token table entry for the token )
( 10 )      a0  a0  .w  add,          ( just fetched )
( 11 )      a0 )  a1  move,           ( get code addr. from token table )
( 12 )      a1)  jmp,                 ( jump to the start of the code )
            c;                        ( area for the new token )
```

The nesting routine code performs three functions. In the first half of the nesting routine, information about the previous execution level is saved away on the return stack (lines 1-3) and the current execution level information is set up (lines 4-6). The second half of the nesting routine (lines 7-12, also known as the 'next' routine), is responsible for starting the execution process going at the newly set up execution level (starting with the first token in the current definition).

At the end of the **execute** routine (described previously), the A1 register was left pointing at the code field for **newword** and the A0 register was left pointing at the token table entry for **newword** . The state of these registers is reflected in the diagram. Discussion of the nesting routine as it relates to execution of **newword** will start at line (4) above. In line (4) the interpretation pointer, 'ip', is set up to point at the first token in **newword**'s parameter field area. This is accomplished by using the **lea**, instruction to add 2 to the address in the A1 register. The nesting routine assumes that the code field in tForth is a 2 byte field.

In line (5) **newword**'s code field address is placed in the start address pointer, 'sa'. In line (6) the address of the token table entry for **newword** is placed in the current token pointer,

'ct'. The 'ct' and 'sa' registers always hold addresses which pertain to the instruction which is currently being executed, in this case, **newword** . At this point, a new execution level has been completely established and the tokens in **newword**'s code area may be executed.

The last half of the nesting routine (the next routine) performs the same functions as the last half of the **execute** routine described above. Both sections of code take a token, calculate its token table entry address, fetch its code address from token table, and vector execution to the first instruction in the code area for the token. The only difference between the two pieces of code is that in **execute** the token to be executed is passed in on the parameter stack and in the nesting routine the token to be executed is pointed to by the interpretation pointer.

Executing a tForth Code Definition

The first token to be executed in **newword** belongs to **blit** (token $0A). **blit** is a code definition, so the code area for **blit** contains machine code:

```
c' blit 10 dump
11E6  10 1D 48 80 48 C0 2F 00 4E D4 70 00 12 1D 10 1D  ..H.H./.N.p.....

101D     ip )+  d0  .b  move, ( get the byte value which follows )
                               ( the  blit  token and increment the )
                               ( ip )
4880     d0  .w  ext,          ( extend it to a word value )
48C0     d0  .l  ext,          ( extend it to a long value )
2F00     d0  sp  -)  move,     ( push the value on the stack )
4ED4     nx  )  jmp,           ( exit to FORTH )
```

Code definitions do not cause a change in execution level so the code field in code definition does not contain a jump to the nesting routine. The machine code instructions in **blit**'s code and parameter fields are executed straight through. Since execution of a code definition did not cause any nesting to occur, termination of a code definition does not require any "unnesting". Program execution simply continues with the next token in the definition. For this reason, a 'JMP (nx)' instruction is used at the end of a code definition. This instruction causes the second half of the nesting routine, the part of the nesting routine which starts execution of the "next" token in the definition, to be executed.

During execution of **blit** the ip was 'bumped' over the byte length literal data, $03. Therefore, the ip is currently pointing at the * token . * , **dup** , and + are also code definitions so this same cycle (jump to the code address, execute the machine code instructions, return to next) will be repeated three more times. Only when the token for **tier1** is executed does the execution cycle get more involved.

Tier Tokens

The tForth interpreter steps through tForth definitions executing a single byte token at a time. The interpreter does not know how to execute 2 byte tokens. Single byte execution works fine with tokens numbered 00-FF hex (the tokens located in tier0 of the token table). But what about the tokens in the rest of the tiers? Tokens in the other tiers would have token numbers like $123, $4A0, or $5FF. These numbers cannot be expressed as single byte values. The 'tier words', **tier1** through **tier9** (which have token numbers $01 through $09 and are located in tier 0) are special versions of **execute** which know how to execute words represented by tokens located in tiers 1 through 9, respectively.

The token for the word . is located in tier 1 of the token table. References to . are compiled as a two-byte sequence: '01 0C' (see the previous diagram of the code area for newword). The '01' is the **tier1** tier token. **tier1** 'knows' that the token which immediately follows it in memory, the '0C', is a token table entry number offset into tier 1 of the token table (rather than an actual token table entry number). This token table entry number offset must be added to the base token number for its tier to calculate an actual token entry number. For example, the base token number for tier 1 is $100 (for tier 2 it is $200, for tier 3 it is $300, etc). The actual token table entry number for . would then be: $100 + $0C = $10C. Once a tier word has determined the actual token table entry number for a token, the token's token table entry address can be calculated and the code corresponding to the token can be executed.

All of the 'tier words' are code definitions. The code definition for **tier1** is listed below:

```
     code  tier1  (  -  )
( 1 )        .tbl  100  +  #n  d0  move,    ( put address of base of token )
                                             ( table, plus 100 hex which is the )
                                             ( first token value in tier1, in the )
                                             ( d0 register )
( 2 )        ip  )+  d0  .b  move,          ( get the next token, the token for  .  , )
                                           ( $0C, and place in the lowest byte of the )
                                           ( value in the d0 register.  now the lowest )
                                           ( 12 bits of the d0 register contain $10C )( (
3 )       d0  a0  move,                    ( move the d0 to the A0 register )
( 4 )        a0  a0  .w  add,              ( calculate the address of the token table )
( 5 )        a0  a0  .w  add,              ( entry for  .  )
( 6 )        a0  )  a1  move,              ( fetch the code address for  .  from  . 's )
                                           ( token table entry )
( 7 )        a1  )  jmp,                   ( jump to  . 's code area )
```

When a tier token is executed it first adds the base token number for its tier to the base address of the token table (line 1 below, the system integer **.tbl** holds the token table address). Next, the token table entry offset into the tier (i.e. the token value which immediately follows the tier token in the definition's code area, $0C in this example) is added to the previous result (line 2) . The last 5 lines of code in **tier1** are used to calculate the address of the token table entry in tier 1 for . , to get the code field address for . , and to vector program execution to the code area for . :

Words whose tokens are located in tiers 1 through 9 are compiled as a two byte token sequence. To conserve program space, the words in tForth have been arranged so that the most often used words are located in tier0.

Nesting Down a Level

Here is a listing of the code area for . :

```
c'  .  20  dump
3E30  4E D3 2D 57 01 04 01 06 31 01 08 01 07 01 01 24  N.-W....1......$
3E40  10 FF 4E D3 01 04 01 06 01 07 01 01 24 10 20 1F  ..N.........$. .

2D      name dup
57      name abs
0104    name <#
0106    name #s
31      name swap
0108    name sign
0107    name #>
0101    name space
24      name type
10      name <exit>
```

. is a colon definition so its code field contains a 'jump to the nesting routine' instruction. Execution of the nesting code will cause a change in execution level to occur. A change in execution level at this point means that the system will stop

executing tokens in **newword** and will start executing tokens in .

Let's follow the transition between execution levels (refer back to the listing of the nesting code fragment).

Line 1: The delta between the start address of **newword**'s code area, held in the sa register, and the address of the token to be executed next, held in the ip register, is calculated and the result is left in the ip register.

Line 2: The word length delta value is store in the return stack.

Line 3: The lower word of the token table entry address for **newword** is saved on the return stack.

Line 4: Put the address of the first token in . in the ip register.

Line 5: Put the code field address of . in the sa register.

Line 6: Put the address of the token table entry for . in the ct register.

Line 7: Fetch a copy of the first token in . , put the byte length token value in the bp register, and bump the ip pointer ahead one byte.

Lines 8-12: Calculate the token table entry address for the token and vector program execution to the token's code area.

Un-Nesting

When . has completed execution, the final word to be executed in **newword** is **<exit>** (also called <;>). **<exit>** takes two word length pieces of 'return' information off of the return stack and restore the execution-related registers so that execution may continue at a previous execution level. Here is the listing for **<exit>** :

```
code  <exit>  Parameter:  (  -  )  Return:  (  n1  n2  -  )
      rp  )+  ct  .w  move,    ( replace the lower word of the )
                               ( current token table entry )
                               ( address with the lower word of )
                               ( of the token table address being )
                               ( used at the previous execution )
                               ( level )
      rp  )+  a0  .w  move,    ( remove the delta ip value from )
                               ( the return stack )
      ct  a1  move,            ( move the new token table entry )
                               ( address to the a1 register )
      a1  )  sa  move,         ( put the code field address found )
                               ( in the new token table field into )
                               ( the sa register )
      a0  sa  0  xl)d  ip  lea,  ( add the code field address to the )
                               ( delta ip value and place the )
                               ( resulting address in the ip reg. )
      next;
```

Since **newword** was executed interactively using **execute** , the above use of **<exit>** will 'unnest' and allow execution of the main loop in **quit** , the 'interpret loop', to continue.

THE IMPLEMENTATION OF INTEGERS

Important Integer-Related Registers

There are two registers which are directly related to the functioning of tForth's integers: the 'iv' and the 'vp' register (see below). The exact usage of these registers will be explained later in this section.

REGISTER	SYMBOL	USAGE
D6	iv	Holds the address where the value of the current integer is stored. "integer value"
A2	vp	Holds the address of the run-time code for **integer** .

Creating Integers

The defining word **integer** is used to create new integers. **integer** is a defining word because it creates named (words with dictionary headers) child words and assigns run-time actions (the run-time action of a child word created by **integer** is to place its value on the stack) to the child word. Here is the definition of **integer** :

```
: integer ( Compile time: n - | Run-time:  - n )
      create          ( create a dictionary header for the )
                      ( new integer )
      4ED2  w,        ( lay a 'jump to the address stored in )
                      ( the vp register' instruction in the code )
                      ( field of the child words code area )
      ,               ( lay the 4 byte initial value for the integer )
                      ( into memory immediately following the )
                      ( 'jump' instruction )
;

 12  integer  fred          ( create a new integer named  fred )
 '  fred  . 3B1             ( this is the token assigned to  fred  )

 c'  fred  10  dump         ( display the contents of  fred's code area )
47938  4E D2 00 00 00 12 31 84 66 72 65 64 07 32 83 6A  N.....1.fred.2.j

 : test1  ( - )  fred  .  ;  ( use  fred  in a colon definition )

                            ( show how a reference to  fred  is )
 c'  test1  10  dump        ( compiled into a colon definition )
4793E  4E D3 03 B1 01 0C 10 00 6B 81 00 07 36 85 64 6F  N.......k...6.do
```

In the first example above integer was used to create a new integer named fred . Then, ' was used to check the token number assigned to fred . Finally, c' was used to display the code area for fred . The code area for fred contains a '4ED2', or "jump to the address in the vp (a2) register" and the 4 byte value , '00000012', of fred .

The second example above uses fred inside of a simple colon definition and then displays the contents of the colon definition's code area. The code area dump shows that integer references are compiled the same way as a references to other colon definitions are compiled, the token for the integer is laid into the code area of the colon definition.

Execution of Words Created By **integer**

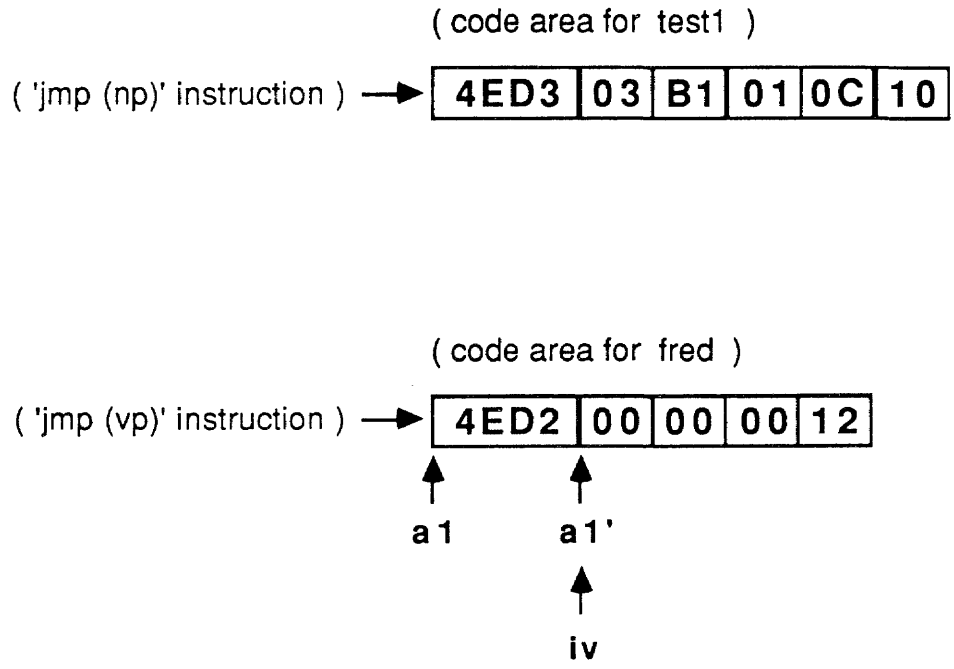
The discussion of the tForth execution process showed that every colon definition contains a two byte machine code instruction, a "jump to the address in the np (a3) register (opcode = 4ED3)", in its code field. During execution of colon words, execution of this special instruction would cause the tForth nesting routine to be run.

The execution process for integers is very similar to the execution process for colon definitions. The vp, or a2, register in the V777 system is used to hold the address of the code fragment shown below. Whenever the token corresponding to an integer is executed, program execution will be vectored to the code field in the integer's code area. This will cause the 'JMP (vp)' instruction to be executed and the code below will be run:

```
      frag  .ramint  to  (  -  n  )
( 1 )       2  #n  a1  addq,          ( 'point' a1 at the parameter field/ )
                                      ( data for the integer )
( 2 )       a1  )  sp  -)  move,      ( push the integer's data on the )
                                      ( parameter stack )
( 3 )       a1  iv  move,             ( put the address of the integer's )
                                      ( data in the iv register )
( 4 )       ip  )+  bp  .b  move,     ( keep tForth execution going )
( 5 )       bp  a0  move,             ( by causing the next token in )
( 6 )       a0  a0  .w  add,          ( the word which 'called' the )
( 7 )       a0  a0  .w  add,          ( integer to be executed... )
( 8 )       a0  )  a1  move,
( 9 )       a1  )  jmp,
      ;c
```

When execution of the .ramint code fragment starts, the a1 register will be pointing to the code field for the integer (see the diagram on the following page). This position of the a1 register was described in detail in the section on the execution of token threaded code. In line 1 of the integer code above the address in the a1 register is repositioned so that it points at the start of the parameter field in the integer's code area. As the diagram shows the parameter field in an integer's code area is four bytes long and holds the current value of the integer. In line 2, the familiar run-time action of integers is performed: the integer's value is placed on the parameter stack. In line 3, the address where the integer's value is stored is placed in the iv register. Lines 4 through 9 are identical to the last 6 lines in the nesting routine. These lines are responsible for causing the next token in test1 , the token for . , to be executed.

Integers Execution



How the Integer Operators Work

Here is the code definition for and an example usage of **to** :

```
code  to  ( n1  n2  -  )
   iv  a0  move,          ( put the address of the integer's data )
                          ( in the a0 register )
   4  #n  sp  addq,       ( drop the top item, n2 - the integer's )
                          ( current value, from the param stack )
   sp  )+  a0  )  move,   ( store the new value, n1, into the )
                          ( integer's storage location )
next;

5  fred  to
```

to should always be executed immediately after an integer (or a local variable) name has been executed. After an integer is executed, the integer's value will be on top of the stack and the address of the integer's storage location will be in the iv register. Since **to** is also passed the new desired value for the integer, there will be two items on the stack when **to** is executed, the current integer value will be on top of the stack and the desired new value will be second on the stack.

In the first line of **to** the address of the integer's storage location is moved into the a0 register. In the second line the integer's current value is dropped from the stack. In the third line the new value on top of the stack is stored into the integer's storage location.

+to is very similar to **to** :

```
code  +to  ( n1  n2  -  )
   iv  a0  move,       ( put the address of the integer's storage )
                       ( location in the a0 register )
   4  #n  sp  addq,    ( drop the current integer value from the )
                       ( parameter stack )
   sp  )+  d0  move,   ( put the increment value in the d0 reg )
   d0  a0  )  add,     ( add the increment value to the current )
                       ( value of the integer )
next;

4  fred  +to
```

addr is even simpler than to :

```
code  addr  ( n  -  a  )
   iv  sp  )  move,    ( put the address of the integer's storage )
                       ( location in the top position on the param )
                       ( stack, write over the integer's current )
                       ( value )
next;
```

System Integers

tForth and V777 system integers, integers which are used during operation of tForth or the editor, are implemented and executed in a completely different manner than integers created with the use of **integer** .

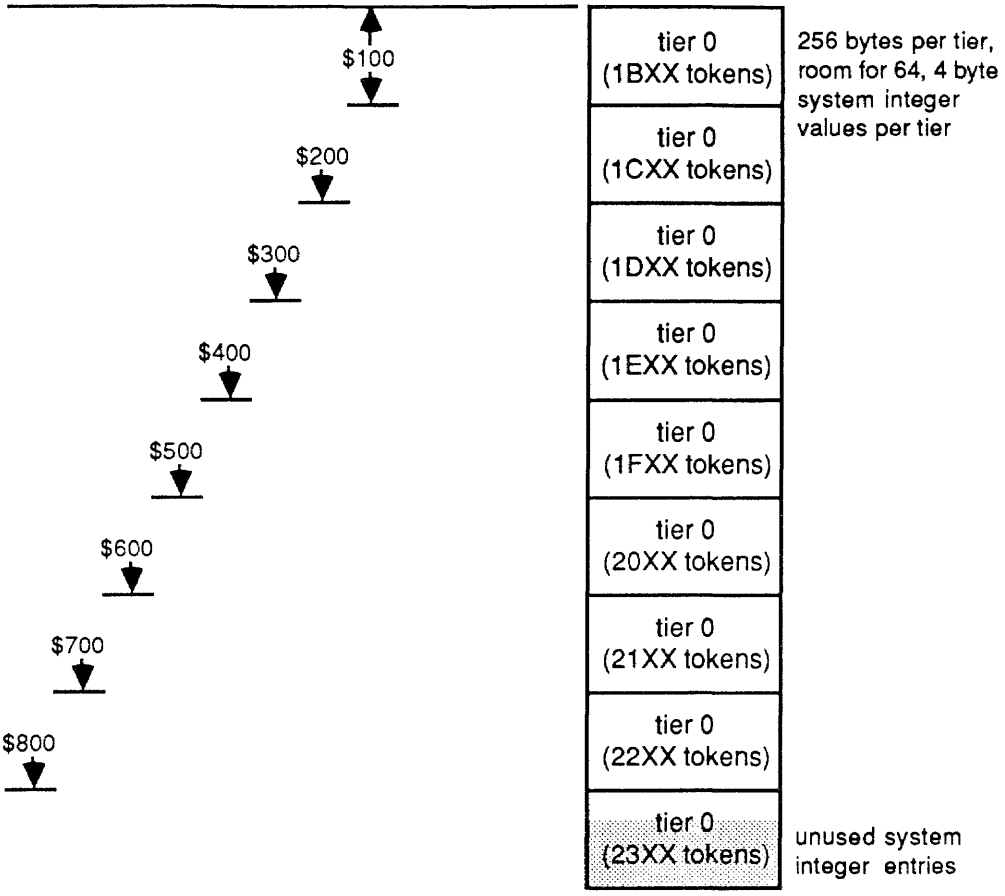
The first difference between system integers and regular integers is that system integers are not created with **integer** . System integers are created with a special integer creation word which is not available for use in the editing environment. A second difference is that the values of system integers are located in a special 'tiered' integer data table. The final difference is that system integers have dictionary header entries for their names but do not have any corresponding dictionary code areas. The reasons for these differences are described below.

The System Integer Tier Table

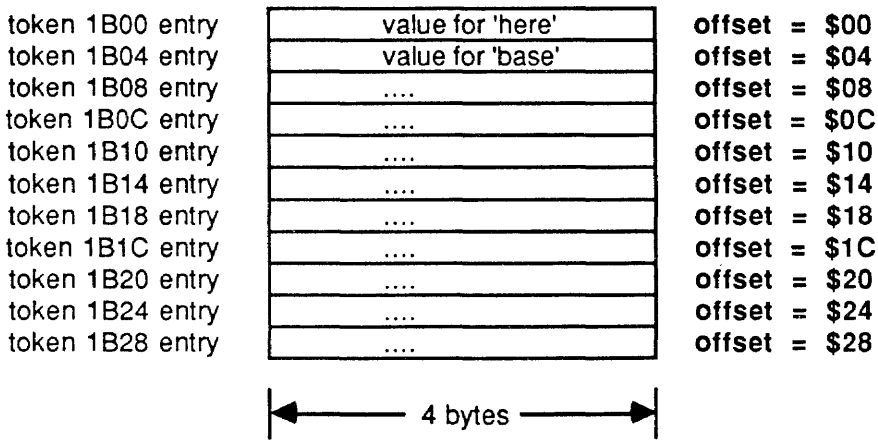
The diagram on the following page shows how the system integer tier table is arranged. The tiered integer table is very similar to to the tiered token table. The system integer table has 9 tiers, numbered 0 through 8. Each tier is 256 (decimal) bytes in size (each tier in the token table is 1024 bytes in size). 64 4-byte system integer values can be stored in each tier (64*4=256 bytes). The system integer table can hold a maximum of 576 (64*9=576) system integer values.

System Integer Table

Tiers:



Close up of the start of system integer tier 0:



System Integer Token Assignments

The value of the system integer **here** is located in the very first position in the system integer table. The value of the system integer **base** is located in the second position in the system integer table. The memory dump below shows how references to system integers are compiled into colon definitions:

```
: test2 ( - a n ) here base ;

c' test2 10 dump
47940 4E D3 1B 00 1B 04 10 00 6B 81 00 07 36 85 64 6F  N.......k...6.do
```

System integers are always compiled as a 2 byte token sequence (even if the system integer is located in tier 0 of the system integer table). **here** has been assigned the tokens '1B' and '00'. **base** has been assigned the tokens '1B' and '04'. Token values '1B' through '22' hex are assigned to the lower level system integer execution words **int0** through **int8** . The system integer execution words perform functions which are analogous to those performed by the tier words (**tier1** through **tier9**) discussed in the tForth execution section. Here is the definition of the system integer execution word **int0** :

```
code int0 ( - n )
   .int 000 +          ( add the offset to system integer tier 0, )
                       ( which is zero, to the base address of the )
                       ( system integer table )
   #n iv move,         ( put the address of the start of system )
                       ( integer tier 0 in the iv register )
   ip )+ iv .b move,
                       ( get this system integer's offset into tier 0 )
                       ( and add it to the address in the iv register )
                       ( to determine the address of this system )
                       ( integer's entry in the system integer table )
   iv a0 move,         ( put a copy of the iv in the a0 register )
   a0 ) sp -) move, (  fetch the system integer's data from its )
                       ( entry in the system integer table )
next;
```

int0 uses a very simple equation to calculate the address of a system integer's storage location in the system integer table. The offset to tier 0 in the system integer table is added to the base address of the system integer table (obtained by executing the word **.int** , which is not available for use in the editor environment). The offset to tier0 is 0, the offset to tier 1 is 100 hex, the offset to tier 2 is 200 hex, etc. The system integer's offset into a particular tier in the table is added to the base address of its tier to calculate the exact address of its table entry. When a reference to a system integer is compiled the first byte ('1B' through '22') is the token for the system integer execution word which corresponds to the integer's tier and the second byte ('00' for **here** and '04' for **base**) is the offset from the start of the tier to the integer's storage location.

Execution of System Integers

The word **execute** contains special provisions for the execution of system integers. Here is the top half of the **execute** routine listed previously:

```
code  execute  ( n - )
   sp )+ d0 move,          ( get the token from the stack )
   \int 8 shl              ( form the value '1B00' hex )
   #n d0 .w cmp,           ( if the token value is greater than )
                           ( or equal to 1B00 then the token )
                           ( belongs to a system integer )
   nc if,
         .int \int 8 shl -   ( subtract 1B00 from the base )
                           ( address of the integer table )
         #n d0 add,        ( now add the token value to the )
                           ( previous result to calculate the )
                           ( address of the integer's storage )
                           ( location )
         d0 iv move,
         iv a0 move,          ( fetch the system integer's value )
         a0 ) sp -) move, ( and place it on the parameter )
         next,               ( stack )
   then,
   ....
   ....
```

If **execute** is passed the token value for a system integer, it performs the system integer functions immediately (place the system integer value on the stack and place the address of the system integer storage location in the iv register).

THE IMPLEMENTATION OF LOCAL VARIABLES

The word **test3** below is a simple example word which demonstrates the usage of tForth local variables. In **test3** three local variables, named 'one' , 'two' , and 'three' , are created and then referenced. The dump of the code area for **test3** will be used as a reference during this discussion of the implementation of local variables:

```
: test3 ( - n1 n2 n3 )
      local one  local two  local three   ( create 3 local variables )
      1 one to                           ( put a 1 in one   )
      2 two to                           ( put a 2 in two   )
      3 three to ;                       ( put a 3 in three )

c' test3 20 dump
478D0  4E D3 14 0C 41 15 4F 0A 02 16 4F 0A 03 13 08 4F  N...A.O...O....O
478E0  11 0C 13 08 11 0C 00 00 00 3A 81 00 07 36 85 64  .........:...6.d
```

```
4ED3              This is the machine code instruction compiled at
                  the start of all colon definitions.
14 0C             Token for <locals> , data for <locals> .
41 15 4F          Tokens for 1 (puts a '1' on the stack), <loc0> , to
0A 02 16 4F       Tokens for blit , data for blit , <loc1> , to
0A 03 13 08 4F    Tokens for blit , data for blit , <local> , data
                  for <local> , to
11 0C             Token for <;lp> , data for <;lp>
```

Execution of Words Which Contain Local Variables

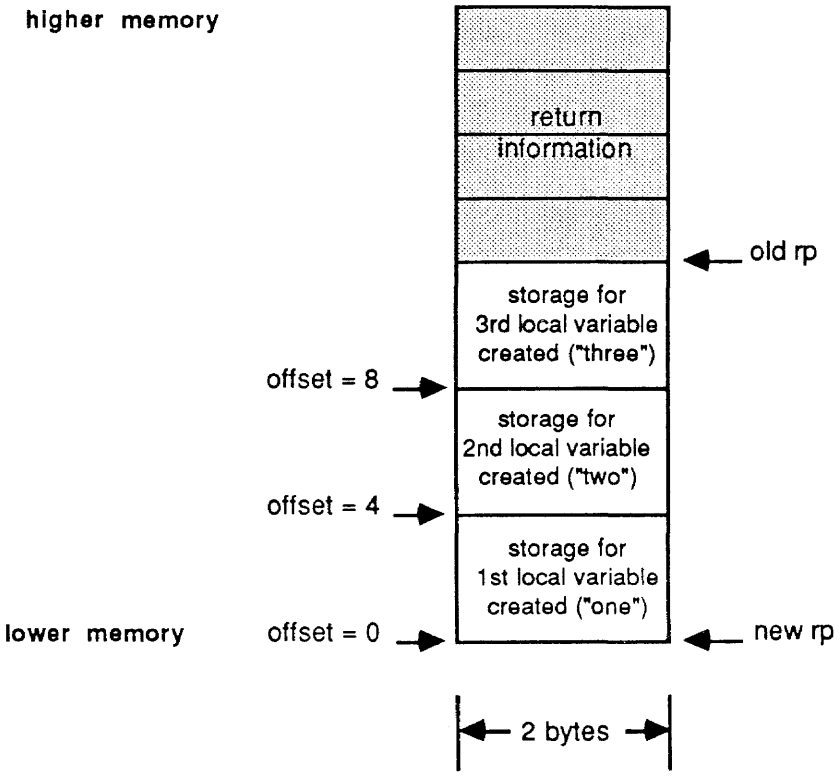
This section will briefly discuss how words which contain local variables are executed. **test3** will be used as an example.

The declaration of the first local variable in a word causes the token for the word **<locals>** to be compiled. The byte location which immediately follows **<locals>** is used to hold data used by **<locals>** . The listing below shows the run-time actions of **<locals>** :

```
code <locals> ( - )
     0 #n  d0  moveq,        ( clear the d0 register )
     ip )+ d0  .b  move,     ( get the byte of data which )
                             ( immediately follows <locals> )
     d0  rp  sub,            ( create a local variable storage )
                             ( area on the return stack )
next,
```

At execution time, **<locals>** is responsible for initializing the storage area to be used by the local variables used in the definition. The storage space for local variables is located on the return stack. When **<locals>** is executed, it subtracts the byte value which immediately follows it in memory, the byte value indicates how much local variable storage is required by this definition, from the current return stack pointer address (see the diagram on the following page).

Local Variable Return Stack Usage



This repositioning of the return stack pointer creates a "hole" in the return stack which will be used as the local variable storage area while the definition executes. The contents of the local variable storage area are not initialized to any value. Since **test3** uses 3 local variables, which each require 4 bytes of storage, a 12 decimal byte storage location is set aside on the return stack.

The next 3 tokens in **test3** will place a '1' into the **one** local variable. '41' is the token for the word **1** . 1 will put a '1' on top of the parameter stack when executed. '15' is the token for the local variable word **<loc0>** . The run-time actions of **<loc0>** are:

```
code  <loc0>  (  -  n  )
      rp  iv  move,          ( put the address of the storage )
                             ( location for the first local )
                             ( variable in the iv register )
      rp  )  sp  -)  move,   ( put the contents of the first )
                             ( local variable on top of the )
                             ( param stack )
next;
```

one was the first local variable declared in **test3** . The storage location for the first local variable is always located at an offset of 0 from the top of the return stack. Local variables act like integers. When a local variable is executed, its contents are placed on the parameter stack and the address of its storage location is placed in the iv register. Because local variables act like integers, the word **to** , whose token will be executed next, will be able to store the value '1' into the local variable's storage location. The diagram reflects the effects of the execution of these three tokens.

The next line to be executed is '2 two to'. The corresponding 4 tokens to be executed are '0A 02 16 4F'. '0A' is the token for **blit** . When **blit** is executed it will place the byte length data which immediately follows it in memory (the '02') on the parameter stack. '16' is the token for the local variable word **<loc1>** . These are the run-time actions of **<loc1>** :

```
code  <loc1>  (  -  n  )
      rp  4  )d  a0  lea,    ( get the address of the second )
                             ( storage location in the local )
                             ( variable storage area )
      a0  )  sp  -)  move,   ( put the contents of the 2nd )
                             ( storage location on the stack )
      a0  iv  move,          ( put the address of the 2nd )
                             ( storage area in the iv register )
next;
```

The second local variable declared in a definition is always given storage at an offset of 4 bytes from the top of the return stack. Aside from the use of a different offset, the actions of **<loc1>** are very similar to those of **<loc0>** . The contents of the

second storage location are placed on the parameter stack and the address of the second storage location is placed in the iv register.

The next line of code to be executed is '3 three to'. The tokens for this line are '0A 03 13 08 4F'. The '0A 03' causes a 3 to be placed on the parameter stack during execution. The '13' is the token for the local variable word **<local>** :

```
code  <local>  (  -  n  )
      0  #n  iv  moveq,          ( clear the iv register )
      ip  )+  iv  .b  move,      ( put the byte of data which )
                                 ( immediately follows <local>  )
                                 ( into the iv register )
      rp  iv  add,               ( calculate the address of this )
                                 ( local variable's storage location )
                                 ( and leave it in the iv register )
      iv  a0  move,
      a0  )  sp  -)  move,       ( put the contents of the local )
                                 ( variable's storage location on )
                                 ( the parameter stack )
next;
```

The third local variable, and all subsequent local variables declared in a definition, have the offset to their storage location on the return stack compiled into the definition immediately after the token for **<local>** . The storage location for the **three** local variable is located at an offset of '8' from the top of the return stack. The first two local variables declared in a definition use special words (**<loc0>** and **<loc1>**), which take advantage of fast addressing modes, to access their contents. The contents of all other local variables are accessed using the more generic word **<local>** .

The final two tokens in test3 , '11 0C', are used to clean up after, and terminate execution of test3 . '11' is the token for <;lp> , which is the special version of <;> compiled when a word which contains local variables is being terminated:

```
code  <;lp>  (  -  )
      0  #n  d0  moveq,
      ip  )+  d0  .b  move,      ( find out how much local variable )
                                 ( space this definition used )
      d0  rp  add,               ( reclaim the return stack space )
      rp  )+  ct  .w  move,      ( take the return information off )
      rp  )+  a0  .w  move,      ( of the return stack and use it )
      ct  a1  move,              ( to restore the ip and sa registers )
      a1  )  sa  move,           ( used during the previous )
      a0  sa  0  x1)d  ip  lea,  ( execution level )
      next;
```

When a word which uses local variables is compiled, the last byte in the word's code area will contain a byte value which indicates how much local variable storage the word uses during execution. <;lp> uses this data to reclaim the return stack space when the word has completed execution.

Compilation of Words Which Contain Local Variables

local is the main word responsible for the creation of local variables:

```
: local  (  -  )
     locals  0=          ( have any local variables been created )
                         ( yet?  if not, perform these special local )
                         ( variable initialization processes... )
          applic  0A  -  localvoc  to
          emptyvoc  0A  +  localvoc  0A  cmove
          compile  <locals>
          here  location  to
          0  c,          ( compile a 'spacer' byte )
     then
     tokens  >r             ( save  tokens  value away )
     locals  tokens  to     ( put return stack offset into  tokens  )
     word                   ( get name for this local variable )
     localvoc  addr  str  len  assign ( create header entry and )
                            ( put offset in token field )
     4  locals  +to         ( increment offset )
     r>  tokens  to  ;  immediate  ( restore  tokens  value )
```

locals is the system integer used to keep track of how much local variable storage space is required for the definition currently being compiled (the colon definition defining word : will always set the contents of locals to zero at the start of compilation of a new colon definition). When compilation of a colon definition terminates, locals will hold a value which specifies the complete local variable return stack storage requirements of the definition. During compilation of a colon definition, the value in locals is used to determine the return stack offset for the local variable currently being declared.

When **local** is used to create a local variable, its first action is to check the current value of **locals** . If **locals** contains a zero, it means that **local** is being asked to create the first local variable for a definition. The creation of the first local variable for a definition requires that the special 'first-local' initialization code, contained between the **if** and **then** in the definition of **local**, must be executed. This special 'first-local' initialization code performs the following functions:

1. Constructs an empty, temporary vocabulary, which is invisible to the rest of the system, and places it immediately below the current location of applic (see the "Opening and Closing Vocabularies" diagram).

2. Stores the address of this special vocabulary in the system integer localvoc .

3. Compiles the token for the word **<locals>** into the definition currently being created.

4. Compiles a byte length 'spacer' (with a value of 0) into the definition. This spacer area will later be used to hold a value which indicates how much local variable storage is required by this definition. The address of the spacer location is stored in the system integer location .

This initialization code is performed only when the first local variable is created for a definition. The rest of the code in the local definition performs the normal functions of local :

1. assign is used to create a dictionary header entry (str and len hold the address and length of the name for the local variable) for the local variable in the special local variable vocabulary.

2. The contents of the locals integer are incremented by four to indicate that four more bytes of local variable storage space are required for this definition.

The Words **assign**, **tokens**, and **recycledtoken**

Local variables do not have any associated code area because the words <loc0> , <loc1> , and <local> are used to perform the run-time actions of local variables. For this reason, the token field in a local variables header field is not used in a standard manner. Rather than holding an encoded token value, the token field in a local variable dictionary header is used to hold the offset to the local variable's storage location on the return stack.

To use the token field in this non-standard manner, local must perform some non-standard manipulations of the contents of the tokens system integer. tokens holds the next token value available for assignment to a new definition. The word assign is used by local to create a dictionary header entry for the local variable in the special, invisible dictionary header area. To fill in the token field of the dictionary entry assign calls recycledtoken . recycledtoken must perform four tests before it can decide which token value to return to assign :

1. Is the system is undergoing target compilation?
   If it is, recycledtoken will return the current value of tokens to assign (the next token value available for assignment) and then will increment the tokens value by one.

2. Is the value in tokens greater than the value in ramtoken0?
   The system integer ramtoken0 holds the token value of the first word in the dictionary which is not a rom word. If the

value in tokens is greater than the ramtoken0 value, the system is attempting to add a new word to the dictionary. Rather than immediately returning and telling assign to assign the value in tokens to the new definition, recycledtoken will first check all of the previously assigned ram token entries (starting at the ramtoken0 entry and continuing to on to the last known assigned token table entry, the entry corresponding to the token value which is one less than the value in tokens) to see if any of the entries have been freed up due to the removal of a word from the system (if a token has been freed, its corresponding token table entry will hold the code address of the word freetoken). If a token in this region is available for re-use, recycledtoken will return its value to assign , otherwise, it will go ahead and return the value in tokens .

3. Are all available token table entries in the token table in use? If the value in tokens corresponds to a token table entry which would not fall within the known token table area, no more tokens are available for assignment to new words, the dictionary is full. recycledtoken should return a '0' if this situation occurs (although it seems to return a '-1' in the current listing).

4. Is the system trying to use the token field in a non standard manner?
   Tokens assigned to words in the roms cannot be purged or reassigned. Therefore, if the current value in tokens is less than the token value found in ramtoken0 , the system is trying the dictionary header fields in some non-standard manner (as is the case with local variable headers where the token field in the header entry is used to hold the local variable's execution time return stack offset). In this case, recycledtoken will immediately return whatever value is currently in tokens to assign .

The listing for recycledtoken is shown on the following page. The word i' ('integer-tick') is a target compiler word which returns the storage address for the system integer whose name immediately follows it. The word tc' ('tee-see-tick') is a target compiler word which returns the code address for the word whose name immediately follows it. endtable is the system integer which holds the address of the end of the token table.

```
: recycledtoken ( - token )
        i' tokens d1 move,          ( get current contents of tokens )
        i' ramtoken0 d0 move,       ( get value of ramtoken0 )
        tc' freetoken #n d3 move,   ( get code address for freetoken )
        i' targeting tst,           ( if targeting is on, return and )
        1 ne bra,                   ( increment tokens )
```

```
d0  d1  cmp,                    ( tokens value  - ramtoken0 value )
1  lt  bra,                     ( if tokens value is less than  )
                                ( ramtokens value, return and )
                                ( increment tokens )
d0  d1  sub,                    ( tokens value - ramtoken0 value )
1  #n  d0  subq,

begin,                          ( look for a token to be recycled )
      1  #n  d0  addq,          ( check all token table entries )
      bp  d2  move,             ( from the start of the token table )
      d2  .b  clr,              ( to the entry corresponding to )
                                ( token value in  tokens )
      d0  d2  add,              ( is the execution address of )
      d2  d2  .w  add,          ( freetoken  in any of the token )
                                ( table entries? )
      d2  d2  .w  add,
      d2  a0  move,             ( a0 holds token table entry addr )
      a0  )  d3  cmp,           ( does entry hold freetoken's code )
                                ( address? )
      eq  if,                   ( if so, recycled the token )
            d0  sp  -)  move,   ( by returning its value to  assign  )
            next,
      then,                     ( used recycled token )

d1  nt  -until,                 ( check all token table entries )
                                ( until the token table entry )
                                ( corresponding to the value in )
                                ( tokens  is reached )

i'  endtable  d1  move,         ( get the address of the end of )
                                ( the token table )
a0  d1  cmp,                    ( has search reached the end of )
                                ( the table? )
lt  if,
      -1  #n  sp  -)  move,     ( out of tokens, return 0? )
else,
      d3  a0  )  move,          ( put the address of freetoken )
                                ( in this token's entry )
1  :l i'  tokens  #n  a0  move,  ( return the value in  tokens  )
      a0  )  sp  -) move,       ( new token )
      1  #n  a0  )  addi,       ( increment tokens )
then,
next;
```

local takes advantage of the relationship between tokens and assign . Before local uses assign , it saves away the current value of tokens and places the current value of locals (which holds the return stack offset for the current local variable) into tokens . The largest possible return stack offset which would be stored into tokens would be 252 decimal because only 64 local variables are allowed per colon definition. The current value of ramtoken0 is approximately 939 decimal. Therefore, when assign is used by local , it does not assign a new token for the local variable. Instead, assign places the offset found in tokens directly into the

token field. After **local** has used **assign**, the previous value of **tokens** is restored.

At the end of compilation of the first line in **test3** (the line where all the local variable declarations are made) the **test3** code area would look like this:

```
4ED3  14  00
```

4ED3	This is the machine code "jump to the nesting routine" instruction which is compiled at the start of all colon definitions.
14	This is the token for <locals> .
00	This is a hole which will be filled in later with the value which indicates how much local variable storage is used by this definition.

The locals system integer would hold a 12 decimal at this point because 3 local variables, which each require 4 bytes of storage, were declared for this definition. The location integer would hold the address of the spacer, and the localvoc integer would hold the address of the temporary dictionary header area where the special local variable headers are located. The token field for one would hold a '0' since its storage location is located at an offset of '0' on the return stack. The token field for two would hold a '4' because its storage area is second on the return stack and the token field for three would hold an '08'.

The Word **doloc**

The remaining lines of test3 contain references to the newly created local variables. As interpret 'processes' these remaining lines (please refer the previous discussion on 'Running tForth), it is constantly using the word doloc to see if the word just encountered is the name of a local variable. Here is the listing for doloc :

```
code doloc ( - f )
   localvoc str len <find> ( can this word be found in the )
                           ( local variables vocabulary? )
   if
      loops + ?dup         ( if so, have any local variables )
                           ( or *loops* been used yet in this )
                           ( definition )
      if
            dup 4 =        ( if they have, was this the second )
                           ( local variable created in this )
                           ( definition? )
            if
               drop        ( if it is the second, compile a )
               compile <loc1>  ( special, fast local variable )
                           ( reference )
            else
               compile <local>  ( otherwise, compile the normal )
               c,               ( local variable reference )
```

```
            then
        else
            compile  <loc0>    ( if this was the first local variable )
                               ( created for this definition compile )
                               ( a special, fast, local variable )
                               ( reference )
        then
        drop  0              ( return a false flag to indicate )
                             ( that  doloc  handled this word )
                             ( so the word needs no further )
                             ( processing by  interpret  )
    then  ;
```

In the first line, doloc uses <find> to check the special local variable vocabulary to see if the name specified by the string address and length found in str and len is the name of a local variable. This is the stack notation for <find> :

```
<find> (  vocabaddr  str  len  -  addr  token  trueflag | If found )
       (  vocabaddr  str  len  -  addr  falseflag  | If not found. )
```

The flag returned by <find> is checked first by doloc to see if the word was the name of a local variable. If the word is the name of a local variable doloc checks the token value, or for a local variable, the return stack offset (the reasons for the reference to the loops system integer will be explained in the next section on the implementation of program control structures). If the offset is 0, doloc 'knows' that the first local variable declared is being referenced so it branches down and compiles the token for <loc0>. If the offset is 4, the second local variable declared is being referenced so the token for <loc1> is compiled. If the offset is greater than 4, the token for <local> , and the byte length offset, are compiled.

Terminating the Compilation of a Word Which Contains Local Variables

The word ; compiles the final two tokens in test3 , '11' and '0C', and backfills the spacer left in test3 :

```
  :  ;  (  -  )
        locals  loops  +  ?dup  ( if  locals  contains a nonzero )
                                ( value then this word uses local )
                                ( variables )
        if
            compile  <;lp>      ( if it does, compile the token for )
            c,                  ( followed by the amount of return )
                                ( stack storage space required )
        else
            compile  <;>        ( if this word doesn't use local )
                                ( variables compile the token for )
                                ( the normal colon definition )
                                ( termination word )
```

(continued)

```
then
state  off                ( return to the interpretation state )

locals                    ( if this word used local variables )
if
      locals  location  c! ( store the amount of storage )
                          ( space required in the spacer )
                          ( location )
then
locals  off  ;  immediate
```

IMPLEMENTATION OF PROGRAM CONTROL STRUCTURES

The lists below show all tForth words involved with program control structures. The first list below shows all of the low level program control primitives which define the run-time actions of the program control words:

<0bran>	**<bran>**	**<loop>**	**<+loop>**
<leave>	**<0leave>**	**<branl>**	**<0branl>**
<leavel>	**<0leavel>**	**<do>**	

The next group of words are those which execute during compilation and cause the words listed above to be compiled:

back			
?pairs	**{loop}**	**{while}**	**nest**
unnest	**if**	**else**	**then**
do	**loop**	**+loop**	**begin**
again	**until**	**while**	**leave**

Execution of Program Control Structures

'begin...until' loops

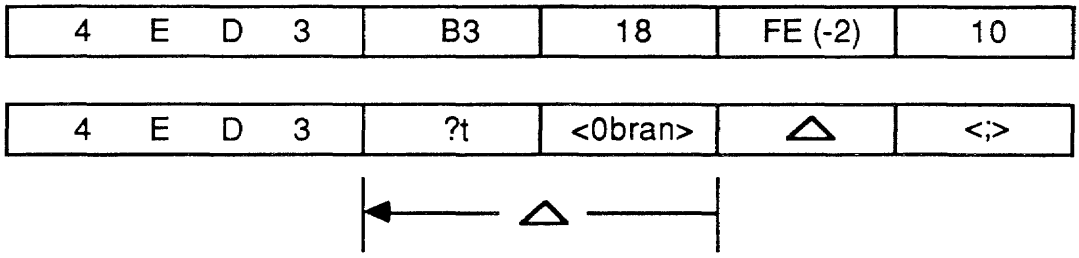
The memory dump below shows the code area for the definition pcs1 , which contains a simple 'begin...until' loop control structure:

```
: pcs1  (  -  )
      begin
            ?t
      until  ;

c'  pcs1  10  dump
478EA  4E D3 B3 18 FE 10 07 36 85 64 6F 7A 65 6E 07 31  N......6.dozen.1

4ED3      'jump to the nesting routine' machine code instruction
B3        token for  ?t
18  FE    token for  <0bran>  ,  data for  <0bran>
10        token for  <;>
```

Note that **begin** does not cause any tokens to be compiled. During compilation, **begin** leaves its address on the stack so that the **until** or **again** which will eventually follow can determine how far back they must branch during execution. This delta backwards jump distance is compiled into the definition immediately after the token for the low level conditional branching primitive **<0bran>** ('bracket-zero-bran'). The diagram on the following page gives a pictorial representation of **pcs1** .

A 'begin...until' Loop

: pcs1 (-) begin ?t until ;



Here is the listing for **<Obran>** :

```
code  <Obran>  ( f  -  )
      sp  )+  tst,                ( test the flag on the stack )
      eq  if,                     ( if the flag is zero execute the )
                                  ( following instructions )
            ip  )  d0  .b  move,  ( get the byte branch data )
            d0  .w  ext,          ( extend it to word length )
            d0  .l  ext,          ( extend it to long word length )
            d0  ip  add,          ( add the delta distance to the )
                                  ( current instruction pointer )
                                  ( address, execution will resume )
                                  ( at the new address )
            next,                 ( exit )
      then,
      1  #n  ip  addq,            ( flag was true, leave the loop )
                                  ( and continue execution at the )
                                  ( token which immediately follows )
                                  ( the byte branch data )
      next;
```

<Obran> is a conditional branching primitive because it will only cause a branch under certain conditions. During execution, <Obran> checks the flag passed to it on the parameter stack. If the flag is false (0), a program branch will occur. Branching in high level tForth code involves modifying the instruction pointer (ip) so that it points at a different section of code. <Obran> uses the byte length data which immediately follows it in memory to determine the destination point for the branch. To calculate the destination address, the byte data is added to the address currently in the ip register. Before the byte data is added to the ip address, it is sign-extended to a long word.

If the flag passed to **<Obran>** is true (nonzero), a program branch will not occur. The code between the if, and then, above, the code responsible for performing the branch, is not executed. Instead, the code which follows the then, is executed. This code adds 1 to the instruction pointer so that the byte branch data is skipped. When a true flag is passed to <Obran> , the instruction pointer will be left pointing to the token which immediately follows the branch data.

During execution of **pcs1** , **<Obran>** will check the flag left on the stack by ?t . If the flag is true (nonzero), the jump back to the beginning point will be skipped and execution will continue immediately outside of the loop. If the flag is false (0), the branch back to the start of the loop will be taken.

begin...while...again Loops

The definition **pcs2** below contains a 'begin...while...again' loop:

```
: pcs2  ( n  -  )
     begin                ( begin... )
          dup 1-  swap    ( duplicate the number on top of )
                          ( the stack, subtract 1 from the )
                          ( duplicate, swap the values )
     while                ( while the value is not zero...)
          dup .           ( duplicate it and display it )
     again                ( back to the top... )
     drop  ;              ( drop the seed )
```

```
c'  pcs2  10  dump
478F0  4E D3 2D 3B 31 18 06 2D 01 0C 17 F7 2F 10 6E 07  N.-;1..-..../.n.
```

```
4ED3          'jump to the nesting routine' machine code instruction
2D 3B 31      tokens for  dup  ,  1-  ,  swap
18 06         token for  <0bran>  ,  data for  <0bran>
2D 01 0C      tokens for  dup  ,  .
17 F7         token for  <bran>  ,  data for  <bran>
2F            token for  drop
10            token for  <;>
```

When **while** is used in a begin loop it also causes the low level word **<0bran>** to be compiled (see the diagram on the following page). If the flag passed to **<0bran>** at execution time is false (0), the branch out of the loop ('<0bran> delta1') will be taken. If the **<0bran>** flag is true (nonzero), the code which immediately follows the **<0bran>** data, the code between the **while** and the **again**, will be executed.

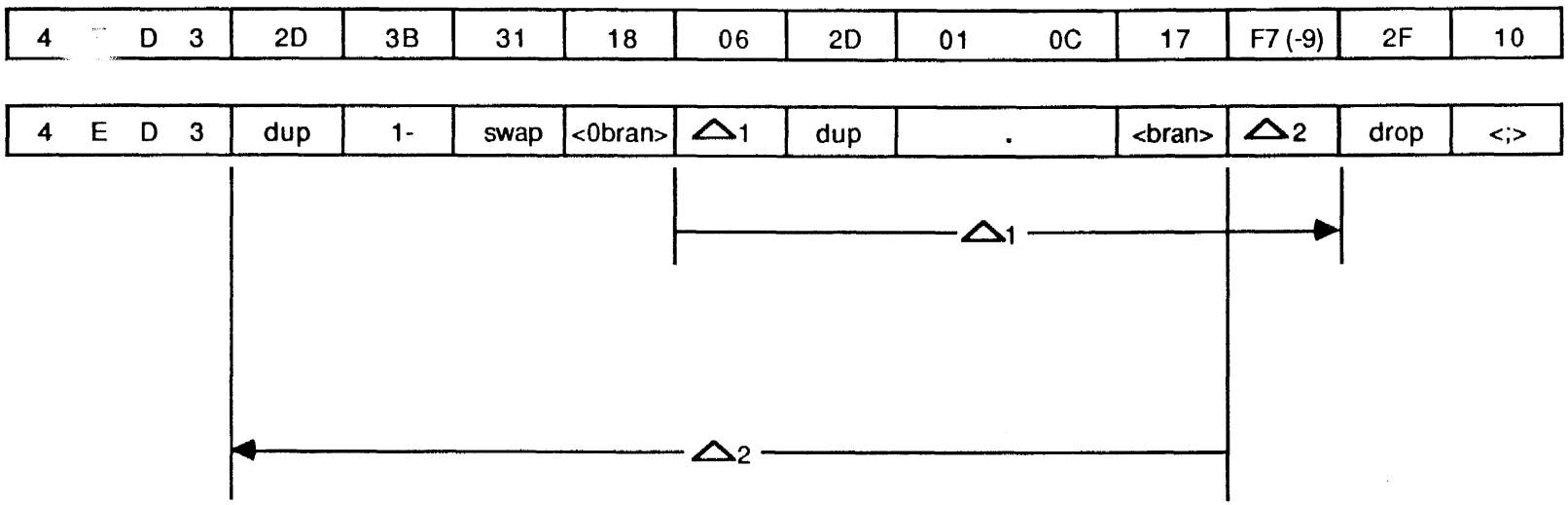
The **again** causes a new unconditional branching primitive, **<bran>** ('bracket-bran'), to be compiled:

```
code <bran>  (  -  )
     ip  )  d0  .b  move,  ( get the byte delta branching data )
     d0  .w  ext,          ( extend to a word delta value )
     d0  .l  ext,          ( extend to a long word delta value )
     d0  ip  add,          ( add the delta offset to the )
                           ( instruction execution address to )
                           ( calculate the address at which )
                           ( execution should continue after )
                           ( the branch )
next;
```

Execution of **<bran>** will always cause a branch in execution to occur. The delta branch distance to be used for the branch is located in byte in memory which immediately follows the **<bran>** token. **<bran>** sign extends the byte value to a long word value and adds the sign extended value to the instruction pointer address to calculate the destination address for the branch.

A 'begin...while...again' Loop

: pcs2 (-) begin dup 1- swap while dup . until drop ;



if...else...then Conditonal Structures

The word **pcs3** uses the 'if...else...then' program control structure:

```
: pcs3  ( f - )
     if              ( if the flag is true... )
          3 .        ( ... display a 3 )
     else            ( otherwise... )
          5 .        ( ... display a 5 )
     then ;          ( and always display a 7 )
```

```
c' pcs3 10 dump
478FE  4E D3 18 07 0A 03 01 0C 17 05 0A 05 01 0C 10 07  N...............
```

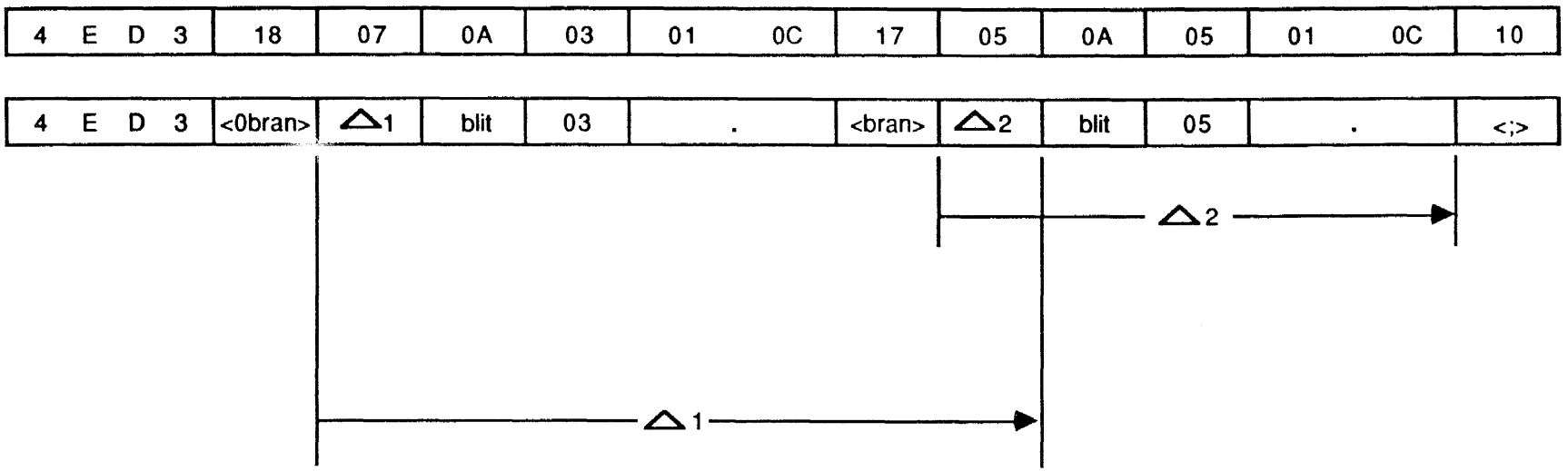
```
4ED3      'jump to the nesting routine' machine code instruction
18 07  token for  <0bran>  , data for  <0bran>
0A 03  token for  blit  , data for  blit
01 0C  token for  .
17 05  token for  <bran>  ,  data for  <bran>
0A 05  token for  blit  ,  data for  blit
01 0C  token for  .
10        token for  <;>
```

The diagram on the following page demonstrates how the 'if...else...then' structure works. The 'if...else...then' structure uses the familiar <0bran> and <bran> branching primitives. if causes a conditional forward branch to be compiled. If the flag passed into pcs3 is true (nonzero), the first branch shown in the diagram will not be taken and the code between the if and the else will be executed. else causes the unconditional <bran> instruction to be compiled. The <bran> instruction terminates execution of the code between the if and the else and unconditionally reroutes program execution to the code which follows the then .

If the flag passed into **pcs3** is false (0), the first branch in the diagram will be taken and program execution will be rerouted to the code which immediately follows the '<bran> delta2' tokens compiled by **else** .

An 'if...else...then' Loop

: pcs2 (-) if 3 . else 5 . then ;



do...loop and do...+loop Control Structures

The pcs4 definition below uses the 'do...loop' program control structure:

```
: pcs4 ( - ) 10 0 do i drop loop ;

c' pcs4 10 dump
4790E  4E D3 0A 10 40 0D 39 2F 0E 10 7A 65 6E 07 31 84  N...@.9/..zen.1.
```

```
4ED3      'jump to the nesting routine' machine code instruction
0A 10     token for blit , data for blit
40        token for 0
0D        token for <do>
39 2F     tokens for i , drop
0E        token for <loop>
10        token for <;>
```

The 'do...loop' structure does not required the compilation of branching data. The branching data used by a 'do...loop' is passed on the stack at execution time. This is the code definition for <do> ('bracket-do'):

```
code <do> ( n1 n2 - )
     ip rp -) move, ( save branching data, the address of the )
                    ( start of the 'do...loop', on the ret. stack )
     sp )+ d0 move, ( get the start value for the loop )
     sp )+ d1 move, ( get the limit value for the loop )
     d1 rp -) move, ( move the limit value to the return stack )
     d1 d0 sub,     ( calculate the number of times the loop )
                    ( should be executed: start - limit , a )
                    ( negative value )
     d0 rp -) move, ( move the loop count to the return stack )
next;
```

During execution, the <do> program control primitive takes two numbers from the parameter stack, the loop limit and start value, and places three items on the return stack, the start loop address, the loop limit value, and the negative loop count. The values on the return stack are used by the corresponding <loop> ('bracket-loop') or <+loop> ('bracket-plus-loop'). The listing for <loop> is :

```
code <loop> ( - )
     1 #n rp ) addi, ( Increment the negative loop count by 1 )
     eq if,
          6 #n rp addq, ( get rid of the 12 bytes of loop )
          6 #n rp addq, ( info on the return stack )
          next,
     then,
     rp 8 )d ip move, ( fetch the address of the start )
                      ( of the 'do...loop' out of the )
                      ( return stack frame and place )
                      ( the address in the ip register )
     next;
```

<loop>'s first action is to subtract one from the number on top of the return stack, the loop count. If the loop count has reached zero, the code between the if, and the then, , which is responsible for removing the 12 bytes of data placed on the return stack by <do> from the return stack, is executed. If the loop count has not reached zero the code which follows the then, , which gets the address left on the return stack by <do> and places it in the ip register, is executed. The address left on the return stack by <do> is the address of the start of the 'do...loop'. Placing this address in the ip register causes program execution to be rerouted back to the start of the 'do...loop'. A picture of the pcs4 execution time return stack frame and code area are shown on the following page.

A 'do...loop'

: pcs4 (-) 10 0 do i drop loop ;

4 E D 3	0A	10	40	0D	39	2F	0E	10

4 E D 3	blit	10	0	<do>	i	drop	<loop>	<;>

Using **while** in **do...loops**

The word **while** can be used in either the **begin** looping structure or in the **do** looping structure. When **while** was used in a **begin** loop above, it caused the **<0bran>** conditional branching primitive to be compiled. Since a conditional exit from a 'do...loop' is more involved than a conditional exit from a **begin** loop (the return stack must be cleaned up when exiting a 'do...loop', see the listing for **<loop>**), **while** will compile a different conditional branching primitive when used inside of a 'do...loop'. **pcs5** shows how **while** could be used in a 'do...loop' program control structure:

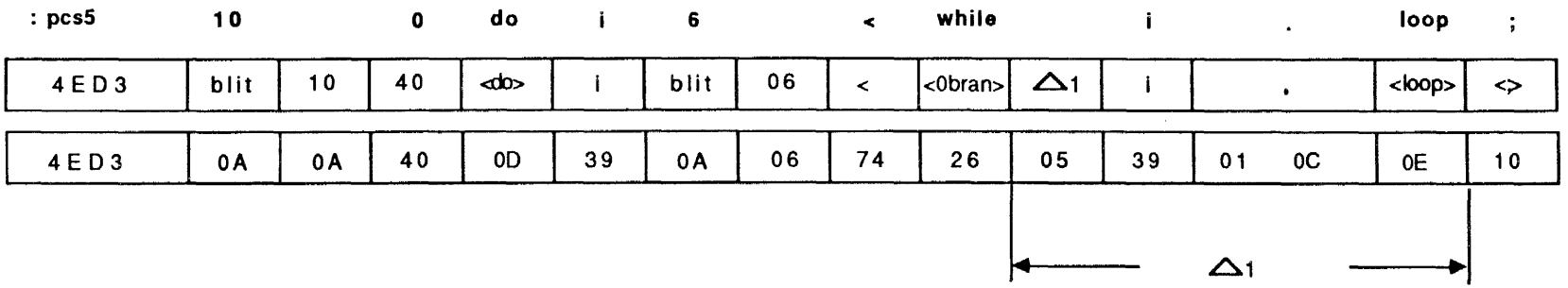
```
: pcs5  (  -  )
      10  0  do       ( go from 0 to 10 )
           i 6 <      ( is the current loop index less than 6? )
      while           ( while it is less than 6... )
           i .        ( print the current loop index )
      loop  ;
```

```
c'  pcs5  20  dump
47918  4E D3 0A 0A 40 0D 39 0A 06 74 26 05 39 01 0C 0E  N...@.9..t&.9...
47928  10 66 72 65 64 07 32 83 6A 6F 65 07 2C 84 70 63  .fred.2.joe.,.pc
```

```
4ED3           'jump to the nesting routine' machine code instruction
0A 0A 40       token for  blit  ,  data for  blit  ,  token for  0
0D             token for  <do>
39 0A 06 74    tokens for  i  ,  blit  ,  data for  blit  , token for  <
26 05          token for  <0leave>  , data for  <0leave>
39 01 0C       tokens for  i  ,  .
0E             token for  <loop>
10             token for  <;>
```

The word <0leave> ('bracket-zero-leave') is compiled when a conditional exit out of a 'do...loop' structure is required. <0leave> , like <0bran> and <bran> , expects a byte of branch data to immediately follow it in memory (see the diagram on the following page). If the flag passed to <0leave> is true (nonzero), this byte of data will be skipped over (see the last line in the <0leave> code definition below) and the code which immediately follows the branch data will be executed (the code between the while and the loop in the pcs5 example). If the flag passed to <0leave> is false (0), the code between the if, and the then, below will be executed. This code adds the byte branch data to the instruction pointer, to calculate the branch destination address, and then removes the 12 bytes of looping information from the return stack. In the pcs5 example, the branch destination is the code which lies just outside of the 'do...loop' control structure.

A do...while...loop structure



```
code <0leave>  (  f  -  )
      sp  )+  d0  move, ( get flag from the parameter stack )
      eq  if,           ( if the flag is false, a branch will occur )
            ip  )  d0  .b  move  ( get the byte length branch )
                           ( offset from the next byte )
                           ( location in memory )
            d0  ip  add,   ( calculate the destination address )
                           ( for the branch )
            6  #n  rp  add,   ( remove 12 bytes of information )
            6  #n  rp  add,   ( from the return stack )
            next,
      then,
      1  #n  ip  addq,        ( 'bump' the ip pointer )
next;
```

leave'ing From Program Control Structures

The word **leave** is used to exit immediately and unconditionally from the current 'begin' or 'do...loop' program control structure. When **leave** is used inside of a **begin** loop it compiles the unconditional branching primitive **<bran>** . When **leave** is used inside of a 'do...loop' however, it must use the special **<leave>** unconditional branching primitive:

```
: pcs6  (  -  )
      10  0  do          ( perform this loop ten times )
            i  7  =      ( does the loop index equal 7? )
            if
                  leave  ( if it does, leave this loop )
            then
      loop  ;            ( if it doesn't, loop back to the top of )
                         ( the loop )
```

```
c'  pcs6  20  dump
4792A  4E D3 0A 10 40 0D 39 0A 07 70 18 03 25 02 0E 10  N...@.9..p..%...

4ED3           'jump to the nesting routine' machine code instruction
0A 10 40       token for  blit  , data for  blit  , token for  0
0D             token for  <do>
39 0A 07 70    tokens for  i  and  blit  , data for  blit  , token for  =
18 03          token for  <0bran>  , data for  <0bran>
25 02          token for  <leave>  , data for  <leave>
0E             token for  <loop>
10             token for  <;>
```

Here is the listing for <leave> :

```
code  <leave>  (  -  )
      0  #n  d0  moveq,
      ip  )  d0  .b  move,  ( get the branch data )
      d0  ip  add,          ( calculate the branch destination )
                            ( address )
      6  #n  rp  addq,      ( remove the 12 bytes of )
      6  #n  rp  addq,      ( 'do...loop' data from the stack )
next;
```

<leave> points the instruction pointer at the first instruction outside of the 'do...loop' and clears the 12 bytes of 'do...loop' data from the return stack.

The diagram on the following page illustrates the execution process for **pcs6** . If the flag passed to **<0bran>** is false (0), the **<0bran>** branch will occur and the instruction pointer will be routed past the **<leave>** token to the **<loop>** token. If the flag passed to **<0bran>** is true (nonzero), the **<0bran>** branch will be skipped over and the **<leave>** branch will occur. Execution of the **<leave>** token will cause the instruction pointer to be altered to point at the <;> token, which is the first token outside of the current 'do...loop'.

A Note About Long Branches

In the examples above, the compiled branch data used by the branching primitives words was 1 byte in length. This means that these primitives may only be used to branch to locations that are within 256 bytes of the start location for the branch. <branl> , <0branl> , <leavel> , and <0leavel> are versions of the branching primitives described above which support the use of longer branch distances. Each of these 'long' versions of the branching primitives will be followed in memory by 2 bytes of branching data. Therefore, the long forms of the branching primitives support branches to locations which are up to 32K bytes away from the branch start address.

The listing for the <0branl> primitive should adequately demonstrate the functioning of all of the long branching primitives:

```
code  <0branl>  (  f  -  )
      sp  )+  d0  move,        ( get the flag from the stack )
      ne  if,                  ( if the flag is false... )
            2  #n  ip  addq,   ( ... skip over the 2 byte )
            next,              ( branch data )
      then,
      ip  )+  d0  .b  move,    ( if the flag is true get the 1st )
      8  #n  d0  .w  lsl,      ( byte of the branch data and )
                               ( shift it into the 2nd byte of )
                               ( the d0 register )
      ip  )  d0  .b  move,     ( get the second byte of the )
                               ( branch data and put it in the )
                               ( least significant byte of d0, )
                               ( now the 2 bytes of branch data )
                               ( are in the lower 2 bytes of the )
                               ( d0 register )
      d0  .l  ext,             ( extend the branch data to a )
                               ( long word )
      d0  ip  add,             ( add the branch data to the )
      next;                    ( instruction pointer )
```

The main difference between the short and long versions of the branching primitives is that the long versions work with two byte branching data and the short versions work with one byte branching data.

Interactive Execution of Program Control Structures

When a program control structure, or a set of nested program control structures, is executed interactively (when it is used outside of a colon definition) execution will commence as soon as the outermost program control structure is closed. For example, the interactive execution of a single level 'do...loop' would begin as soon as the closing loop is entered. Interactive execution of a 'do...loop' nested within an 'if...else...then' program control structure would begin as soon as the then which closes the outer 'if...then' structure is entered.

A do...if...leave then...loop

: pcs6	10		0	do	i	7		=	If		leave	then	loop	;
4ED3	blit	10	0	<do>	i	blit	07	=	<0bran>	△1	<leave>	△2	<loop>	<>
4ED3	0A	0A	40	0D	39	0A	07	70	18	03	25	02	0E	10

During the interactive use of program control structures, the system goes into a 'temporary compilation' state. Code is compiled in the code area of the dictionary until the outermost control structure is closed. As soon as the outermost control structure is closed, the temporary code is moved from the dictionary to an interactive program control structure execution buffer and executed. The address of the interactive execution buffer is kept in the system integer execbuf .

tForth uses two system integers to determine if program control structures are being used interactively, nesting and state . nesting is used to hold a count of how many program control structures have been nested inside of the outermost structure. Each time a word which starts a program control structure (if , do , begin) is used, the contents of the nesting system integer are incremented by one. Each time a word which ends a program control structure (then , loop , +loop , again , until) is used, the contents of the nesting system integer are decremented by one. If program control structures are used correctly, nesting should always hold a zero when the outermost structure is closed.

state is used to keep track of whether the system is in the interpretation or compiling state. If state holds a zero, the system is in the interpretation state. Whenever the nesting integer holds a nonzero value while the state integer holds a zero, the system is performing the temporary compilation required for the interactive execution of program control structures.

nest

As the definition of nest shows, execution of nest will always cause the nesting contents to be incremented by one. If nest is used when both the nesting and state integers hold a zero, it means that temporary compilation of a program control structure has just started. In this situation, nest will save the address of the start of the temporary compiled code in the bound system integer and then will compile the 2 byte machine code instruction which tells the system to 'jump to the nesting routine' (during program execution) at the start of the temporary code (please do not confuse the nesting system integer used by the program control structures with the nesting routine used during the execution of token threaded code).

```
: nest ( - )
     nesting state or 0=  ( check the values of nesting )
                          ( and state )
     if                   ( if both nesting and state are )
                          ( 0, we are starting the compilation )
                          ( of a control structure which is to )
                          ( be executed interactively )
          here bound to   ( save away the address of the )
                          ( start of the interactive code )
          4ED3 W,         ( lay down a 'jump to the nesting )
                          ( routine' instruction at the start )
                          ( of the interactive code )
     then
     1 nesting +to ;      ( always increment the contents of )
                          ( the nesting system integer )
```

unnest

unnest is the complement of the word **nest** :

```
: unnest ( - )
     local oldhere
     local size
     -1 nesting +to      ( decrement the nesting value )
     nesting state or 0=
     if                  ( if the contents of nesting have )
                         ( been reduced to zero and the )
                         ( system is in the interpretation )
                         ( state, it's time to execute the )
                         ( temporary control structure code )
          [compile] exit
          bound oldhere to ( get the address of the start of )
                           ( the temporary code )
          here oldhere - size to    ( determine the size )
                                    ( of the temporary )
                                    ( code )
          oldhere execbuf size cmove ( move the code to )
                                     ( the execution )
                                     ( buffer )
          oldhere here to    ( move the here pointer back so )
                             ( the temporary code will be )
                             ( overwritten in the dictionary )
          execbuf            ( get the address of the execution )
                             ( buffer )
          size execbuf +to   ( temporarily move the start of the )
                             ( execution buffer to the location )
                             ( just past the current temporary )
                             ( code )
          goto               ( execute the code at the execbuf )
                             ( address, the temporary code )
          size negate execbuf +to ( set the start address of )
                             ( the execution buffer back to its )
                             ( original address )
     then ;
```

unnest will always decrement the contents of the nesting system integer. Then, if nesting has been reduced to zero and state holds a zero, the code between the **if** and the **then** in unnest will see to it that the temporary code is executed. Note that after the code has been copied up to the execution buffer the here pointer is moved back to the position it held before the start of the temporary compilation process. This ensures that the code area does not become cluttered with unused code.

goto is used by unnest to directly execute the code located starting at the execution buffer address:

```
: goto ( a - )
     ['] temp +table ! ( borrow temp's token table entry )
                       ( put the address of the code to )
                       ( be executed in its token table )
                       ( field )
     temp              ( now execute temp )
;
```

The code which **temp** points to must end with a **next** in order to properly terminate execution.

Compilation of Program Control Structures

The **if...then** Control Structure

Here is the definition of the word if :

```
: if ( - a n )
     nest              ( add one to the nesting level )
     compile <Obran>   ( compile the <Obran> token )
     here              ( return the address where <Obran>'s )
                       ( branch data should be located )
     0 c,              ( reserve a spot for <Obran>'s data )
     2 ;               ( 2 means 'if...then' structure )
immediate              ( immediate means if is executed )
                       ( at compile time )
```

if increments **nesting** , compiles the token for **<Obran>** , and compiles a byte length zero spacer in the location where <Obran>'s data will later be placed. **if** leaves the address of the **<Obran>** data location and a 2 on the parameter stack during compilation. The 2 indicates that the next piece of data on the parameter stack belongs to an 'if...then' control structure.

else is used to mark the end of the if code and the start of the else code in an 'if...else...then' program control structure. else has three compile time duties. First it must verify that if was used previously. To perform this function else searches through the parameters on the stack. If it encounters a '4', which identifies while or leave data, it moves both the identifier and the associated data over to the return stack. As soon as else has removed all while or leave data from the parameter stack it expects to find the '2'

which identifies if data. If else does not find the '2' 'if' data identifier at this point, a program control structures pair mismatch has occurred and the system will abort.

Once **else** has found the **if** identifier it can perform its remaining two duties. First, else must compile an unconditional branching primitive (<bran>) and a byte-length '0' branch data spacer at the end of the if code. At execution time, this unconditional branch will terminate execution of the if code by routing execution past the else code to the code which immediately follows the then (see the previous diagram for pcs3). Next, else must calculate and backpatch the branch data for if so that the if branch points to the start of the else code. Because else did find the '2' on the stack, it knows that the next address on the stack is the if data, the address where the if branching data should be stored. else uses back to calculate the distance between this data location and the start of the else code and to insert the delta branch distance into the <0bran> data spot:

```
: back ( a - )
     here over -              ( calculate delta distance )
     dup -80 7F inrange not ( make sure data is in the byte )
                              ( range )
     abort" use long branch"  ( abort if delta is too large )
     swap c! ;                ( store delta in correct spot )
```

else's final actions are to leave the address of its branch data location and a '2' 'if..else..then' control structure identifier on the parameter stack, and then to transfer any data left on the return stack back to the parameter stack. Here is the definition of **else** :

```
: else ( a n - )
     0 >r                  ( put a marker on the return stack )
     begin                 ( move all the leave and while  )
          dup 4 =          ( data from the parameter stack )
     while                 ( to the return stack )
          >r >r
     again

     2 ?pairs                 ( was there an if which matches )
                              ( this else? )
     compile <bran> 0 c,  ( compile an unconditional branch )
                          ( and a byte length spacer )
     back                 ( backpatch the previous if data )
     here 1- 2            ( leave the address of the else  )
                          ( data and the 'if...then' identifier )
     begin
          r> ?dup         ( move all of the leave and while )
     while                ( data back to the parameter stack )
          r>
     again ; immediate
```

then is a simpler version of **else** . **then** is only responsible for checking for a previous **if** or **else** and backfilling the delta branch data for the previous **if** or **else** . **then** will also use **back** to take the data address left on the stack by **if** or **else** and to calculate the delta branch distance. If an **else** was used last, **back** will fill in the data for the **<bran>** token. If an **if** was used last, **back** will fill in the data for the **<0bran>** token. Since then closes the 'if...then' and 'if...else...then' program control structures, it uses **unnest** :

```
: then  ( a n - )
     0 >r                ( put a marker on the return stack )
     begin
          dup 4 =        ( move all of the leave and while data )
     while               ( from the parameter stack to the return )
          r> r>          ( stack )
     again

     2 ?pairs       ( check for an 'if...else...then' pair match )
     back           ( backpatch the previous if or else )
                    ( branching data )
     begin
          r> ?dup   ( move all of the leave and while data )
     while          ( back to the return stack )
          r>
     again
     unnest ;       ( we've just closed a program control )
immediate           ( structure so unnest... )
```

The **do...loop** Control Structure

Return Stack Usage

During the compilation of colon definitions, the system integer loops is used to keep track of the 'do...loop' return stack usage for the definition. Whenever a 'do...loop' is started, 12 decimal is added to the contents of loops (each 'do..loop' uses 12 bytes of return stack space during execution: 4 for the 'do...loop' start address, 4 for the limit value, and 4 for the loop count value). Whenever a 'do...loop' is terminated, 12 decimal is subtracted from the contents of **loops** .

The 'do...loop' return stack usage is monitored because two other words, doloc and exit are very return-stack-usage dependent. doloc , described previously, is the word responsible for compiling references to local variables. Local variable storage space is kept on the return stack and is located using an offset from the current top of the return stack.

Although each local variable "tells" doloc what its storage location offset into the return stack should be, doloc will check the contents of loops to determine if the offset needs to be adjusted to account for extra 'do...loop' data which will be on the return stack (see the listing for doloc in the technical local variable discussion) at execution time.

exit is a word which may be used at any time to unconditionally terminate execution of the current colon definition. If the colon definition uses local variables or 'do...loops', exit is responsible for proper clean-up of the return stack so that the return information required upon exit from the definition may be accessed.

```
: exit  ( - )
      locals  loops  +      ( does this word use the return )
                            ( stack? )
      ?dup
      if                    ( if it does, compile a special form )
            compile  <exitlp> ( of  exit  which knows how to )
                            ( clean up the return stack )
            c,              ( compile the number of return )
                            ( stack bytes which are used by )
                            ( this word )
      else
            compile  <exit>    ( otherwise, compile the normal )
      then  ;  immediate       ( exit  )
```

do, loop, and +loop

The word **do** performs four compile time actions. First, since it is a word which starts a new program control structure, it uses the word nest . Second, since it requires 12 decimal bytes of return stack storage space, it increments the contents of loops by 12. Next, it compiles the token for <do> . Finally, it places the contents of the nestype system integer on the parameter stack, places the 'do...loop' identifier, '3', on the parameter stack, and places a '-1' in nestype .

nestype is a system integer which holds a value which distinquishes between 'do...loops' (-1) and 'begin' loops (0). Since 'do...loops' and 'begin' loops perform many similar compile time activities, tForth conserves code space by sharing compiling words between the two control structures. Since the shared words must still perform some loop-specific actions, the nestype integer is used to indicate which type of loop is currently being compiled.

loop and +loop

{loop} is used by all program control words which terminate program control structures (loop , +loop , again , until). {loop} is passed two parameters, the structure identifier value for the type of structure being compiled (3 for 'do...loops' and 1 for 'begin' loops), and the token for the control structure primitive compiled by the word which called {loop} (loop compiles <loop> , +loop compiles <+loop> , again compiles <bran> , until compiles <0bran>). The structure identifier value is stored in the system integer temp0 . The token value is stored in the system integer temp1 (because the program control structures compiling words make

extensive use of the return stack they cannot use local variables). The main actions of both loop and +loop are performed by the shared compiling word **{loop}** as follows:

```
: loop ( n1 n2 n3 n4 - )
     temp0 to              ( get the token from the stack )
     temp1 to              ( get the identifier from the stack )
     0 >r                  ( put a marker on the return stack )
     begin
          dup 4 =          ( move all of the leave and while )
     while                 ( data to the return stack )
          swap             ( swap the data during transfer so )
          >r >r            ( that the identifier is on top of the )
     again                 ( address )

     temp1 ?pairs          ( check for a pair mismatch )
     temp0 c,              ( compile the token for this type )
                           ( of looping structure )
     temp0 ['] <bran> =    ( again compiles a <bran> ... )
     temp1 ['] <0bran> =   ( until compiles a <0bran> ... )
  or                       ( is this an again or until? )
     if                    ( if it is, fill in the required )
          here - c,        ( branch data )
     then

     nestype to            ( restore the previous nestype )
                           ( value, left on the stack by either )
                           ( do or begin )
     begin
          r>               ( is there any leave or while )
     while                 ( data on the return stack? )
          here r@ -        ( if there is, calculate and backfill )
          r> c!            ( the leave or while branching )
     again ;               ( data )
```

After **{loop}** moves all **leave** and **while** data to the parameter stack it compares the structure identifier passed in on the parameter stack to the structure identifier stored in **temp0** to see if a pair mismatch has occurred. If all control structures are matched up properly **{loop}** will compile the token held in **temp1** and, if the token compiled was the token for **<bran>** or **<0bran>** , the delta branch data will be calculated and compiled. Next, the loop type is restored by storing the loop type value left on the stack by a previous **do** or **begin** into **nestype** .

The final action of {loop} involves backfilling the delta branch data information for all unresolved leave and while tokens. Both leave and while compile branches to a location just outside of the current program control structure. Since this location cannot be determined until the program control structure has been closed, {loop} , which is only called by words which close program control structures, is used to perform this function.

begin, again, until

begin is a much simpler version of **do**:

```
: begin  (  -  a  n  )
     nest              ( indicate that a new program control )
                       ( structure is being compiled )
     here  1           ( leave the address of the start of the )
                       ( 'begin' loop code and a 'begin' loop )
                       ( identifier )
     0  nestype  to    ( store the 'begin' loop type identifier )
                       ( into  nestype  )
;  immediate
```

begin calls **nest** , leaves the address of the start of the 'begin' loop and a '1' to identify the 'begin' data, and sets nestype to '0' (identifies 'begin' type loops).

again and **until** are also very straightforward control structure operators. again always takes an unconditional branch back to the start of the loop:

```
: again  (  n  a  -  )
     1                 ( pass the 'begin' structure identifier ... )
     [']  <bran>       ( and the token for the primitive compiled )
                       ( by  again ...  )
     {loop}            ( ...to  {loop}  )
     unnest            ( this is the end of the structure so unnest )
;  immediate
```

again expects the 'begin' structure identifier and the address of the start of the 'begin' loop to be somewhere on the parameter stack when it is called. again will pass an additional 'begin' structure identifier, '1', and the token for the unconditional branching primitive <bran> to {loop} . {loop} will compile the <bran> token, and calculate and compile the delta distance between the <bran> token and the start of the 'begin' loop.

The only difference between **until** and **again** is that until causes a conditional branch to the start of the loop to be compiled:

```
: until  (  n  a  -  )
     1
     [']  <0bran>
     {loop}
     unnest  ;
```

while and leave

while and **leave** are used to conditionally or unconditionally exit from the current loop control structure:

```
: while  (  -  )
     [']  <0bran>    (  while  compiles  <0bran>  if it is used )
```

```
                    ( inside of a 'begin' loop )
      ['] <0leave>  ( while compiles <0leave> if it is used )
                    ( inside of a 'do' loop )
      {while}       ( {while} makes the decision )
;  immediate

:  leave ( - )
      ['] <bran>    ( leave compiles <bran> if it is used )
                    ( inside of a 'begin' loop )
      ['] <leave>   ( leave compiles <leave> if it is used )
                    ( inside of a 'do' loop )
      {while}       ( {while} makes the decision )
;  immediate
:  {while} ( n1 n2 - )
      nestype       ( is this a 'begin' loop or a 'do' loop? )
      if            ( if its a 'do' loop, dispose of the second )
            swap    ( token on the stack. the second token )
      then          ( is the token to be used inside of 'begin' )
      drop          ( loops )
      c,            ( compile the remaining token )
      here 4        ( leave the address of the branching data )
                    ( for the compiled branching primitive and )
                    ( leave a while / leave data identifier )
      0 c, ;        ( compile a byte length spacer for the )
                    ( while or leave data )
```

If **while** or **leave** are used inside of a 'begin' loop either a conditional or unconditional branch out of the loop will be compiled. If **while** or **leave** are used inside of a 'do' loop a conditional or unconditional branch out of the loop with return stack clean up will be compiled. The word **{loop}** , described above, is used to backfill the branching data for **while** and **leave**

Some Notes on Initial Program Control Structures

Any number of tForth control structures may be nested within a program control structure. The program control structures described only support branches up to 256 bytes in length (short branches). They are currently being modified to support branches which are up to 32K bytes in length (word branches). See below.

Program Control Structures Which Support Byte and Word Length Branching

These 13 words are used to implement the new program control structures:

if backelse {elsethen} else then

{while2} while leave
{loop2} until again loop +loop

if...else...then Control Structure Words

Only the 'if...else...then' program control words can cause code movement to occur. The word **if** will always leave a single byte spacer for its associated data. If **else** or **then** later determine that the code between the **if** and the **else** , or between the **if** and the **then** take up more than 7F hex bytes, the **if** data space will have to be expanded to word length. The code between the **if** and **else** or between the **else** and **then** will have to shifted one byte towards higher memory. **if** temporarily stores the byte length hexadecimal code FF in its data area during compilation. This code is used to inform **else** and **then** that the data are belongs to an **if**.

The new version of **if** is very similar to the previous version. The only difference is that the new version leaves a hex FF, rather than a 0, in its byte length data area.

else always leaves a 2-byte spacer for its associated data. **else** temporarily stores the offset back to the **if** data in this 2-byte space during compilation. If the offset back to the **if** data area is short, the first byte in the **else** data area will be the hexidecimal code FE and the second byte will be the byte length offset. If the **if** offset is long, greater than 7F hex bytes, the entire word offset is stored in the **else** data area. If **else** determines that the offset back to the **if** data is greater than hex 7F bytes, it will shift the code between the **if** and **else** by one byte to make room for 2 bytes of **if** data, store the word length delta branching distance in the enlarged **if** data area, and will change the **<0bran>** instruction compiled by **if** to a **<0branl>** instruction.

If **then** is used after **if** , with no intermediate **else** , **then** performs functions which are very similar to **else** . If the distance between the **if** and the **then** is less than 7F hex bytes, **then** will simply store the delta distance in the **if** data area. If the distance is greater than 7F hex bytes, **then** will shift the code between the **if** and the **then** up in memory by one byte, store the word length delta branching distance in the **if** data area, and will replace the **<0bran>** token with the token for **<0branl>**.

If **then** is used after **else** it will first calculate the distance between the start of the **else** data area and the **then** destination point. If this distance is short, **then** will store the byte length branching distance in the first byte of the two byte **else** data area and then will move the code between the **else** and the **then** down in memory by one byte to recover the second unused byte of the **else** data area. Before performing this code movement, **then** will snake back up to the **if** data area, using the offset stored temporarily in the **else** data area, and reduce the **if** branching distance by one byte (since the start of the **else** code is to be moved down in memory by one byte). If the distance between the **else** and **then** is long, **then** will store the word length delta branching distance directly in the word length **else** data area and will change the **<bran>** token to a **<branl>** token.

{elsethen}

Since **else** and **then** perform many similar functions, a common lower level word {**elsethen**} has been defined to conserve program space. {**elsethen**} performs three major actions. First, it transfers all of the **leave** and **while** **data** to the return stack. Next, it takes care of all **if** or **else** backpatching and code movement. Finally, it takes the **leave** and **while** data off of the return stack and adjusts the addresses as necessary. The backpatching code will return a '+1' value if the code had to be moved towards higher memory, a '-1' if the code had to be moved towards lower memory, or a '0' if the code was not moved. {**elsethen**} will use this value when it adjusts the **leave** and **while** addresses. {**elsethen**} is passed a flag which tells it which conditional word, **else** or **then** , is using it.

backelse

backelse is a word used by {**elsethen**} when backpatching **else** data is required.

begin and **do** Loop Structures

The words **begin** , **again** , **until** , **loop** , and **+loop** have only been modified so that they use the newer versions of the words {**while**} and {**loop**}.

while and **leave** now always use a 2-byte branching distance. until and again will use either a byte or 2 byte branching distance.

Size Considerations

There are currently approximately 160 uses of **while** and 20 uses of **leave** in the Cat code. Use of these new program control structures will cause these words to use 3 bytes of code space each rather than the 2 bytes of code space they each used previously so the code space will increase by 180 bytes.

The code used to implement these new versions of the control structures is 280 bytes larger than the older control structures code.

Therefore, these modifications will cause the Cat program to take up 480 more bytes of program space.

THE tFORTH 68000 ASSEMBLER

INTRODUCTION

This section of the manual will explain the syntax and usage of the tForth 68000 assembler. The architecture, addressing modes, and instruction set of the 68000 microprocessor will be discussed briefly. For more detailed information on the 68000 microprocessor please refer to the Motorola Microprocessor Reference Manual, 4th Edition.

Any differences between the standard Motorola-format 68000 assembler syntax, as presented in the examples in the Motorola Microprocessor Reference Manual, hereafter referred to as the M68000 manual, and the syntax required by the tForth assembler will be noted in the text.

BRIEF OVERVIEW OF THE 68000 MICROPROCESSOR

Internally, the 68000 has 32-bit data and address paths. Externally the 68000 has 16 data lines and 24 address lines. The 24 external address lines allow the 68000 microprocessor to directly access 16 megabytes of address space.

Execution Environment

The 68000 executes in one of two 'modes': user mode and supervisor mode. "Cat" code always executes in supervisor mode. In user mode a program is only allowed to execute a subset of the available 68000 program instructions. The prohibited instructions are those which are usually used by systems level software.

The diagram on the following page shows that eight 32-bit data registers, eight 32-bit address registers, a 32-bit stack pointer, a 32-bit program counter, and a 8-bit condition code register are available during user mode program execution.

In supervisor mode the 32-bit supervisor stack pointer and the 16-bit status register (an extension of the 8-bit condition code register) are also available.

Data Registers

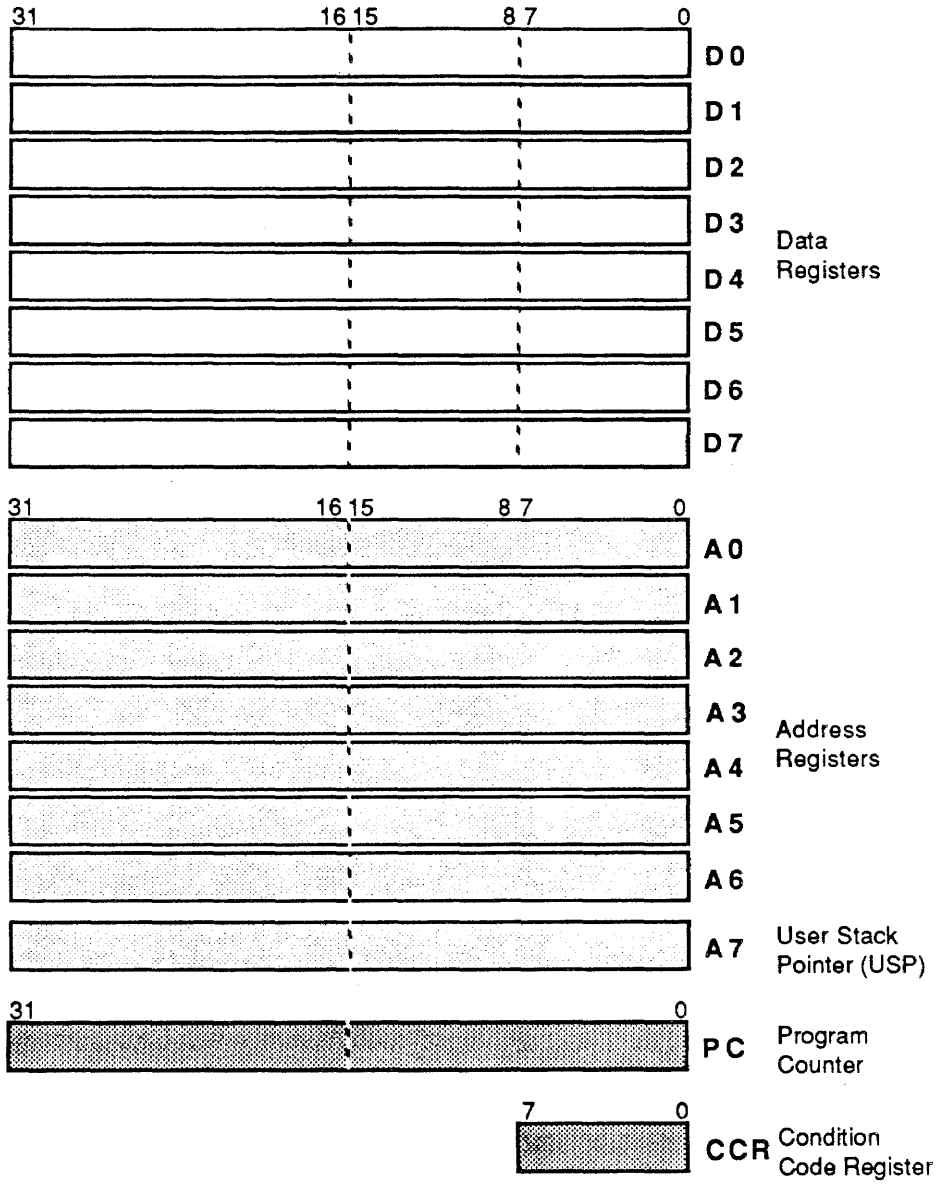
Each data register supports data operands of 1, 8, 16, or 32 bits. Byte operands occupy the low order 8 bits, word operands the low order 16 bits, and long word operands the entire 32 bits. The least significant bit is addressed as bit zero; the most significant bit is addressed as bit 31. When a data register is used as either a source or destination operand, only the appropriate low order portion is changed; the remaining high-order portion is neither used nor changed (i.e. if a data register is used to hold a byte sized operand, only the lower 8 bits of the data register are affected by the operation).

Address Registers

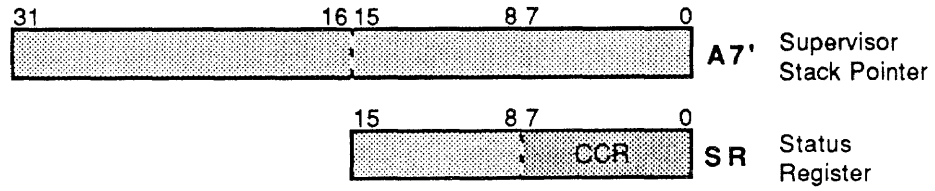
All address registers and both stack pointers are 32 bits wide and hold full 32 bit addresses. Address registers do not support byte sized operands. Therefore, when an address register is used as a source operand, either the low order word or the entire long word operand is used depending upon the operation size. When an address register is used as the destination operand, the entire register is affected regardless of the operation size.

68000 Execution Environment

User Programmer Execution Environment:



Supervisor Programmer Execution Environment:



THE 68000 INSTRUCTION SET

The 68000 instruction set allows the following eight types of operations to be performed:

OPERATION	INSTRUCTION (tFORTH ASSEMBLER)		
Data Movement	exg,	lea,	link,
	unlk,	movem,	movep,
	moveq,	move,	pea,
	swap,		
Integer Arithmetic	add,	addx,	addi,
	addq,	clr,	cmp,
	cmpa,	divs,	divu,
	ext,	muls,	mulu,
	neg,	negx,	sub,
	subi,	subq,	subx,
	tas,	tst,	
Logical	and,	andi,	or,
	ori,	eor,	eori,
	not,		
Shift and Rotate	asr,	asl,	lsr,
	lsl,	roxr,	roxl,
	ror,	ror,	
Bit Manipulation	bchg,	bclr,	bset,
	btst,		
Binary Coded Decimal	abcd,	sbcd,	nbcd,
Program Control	bra,	bsr,	dbra,
	jmp,	jsr,	rtd,
	rtr,	rts,	set,
System Control	rte,	reset,	stop,
	trapv,	chk,	trap,

Note: The 68000 instruction names used in the tForth assembler are very similar to those presented in the M68000 manual. The only differences are that in the tForth assembler the instruction names must be written in lowercase and must be immediately followed by a comma.

tForth Assembler Syntax

The tForth assembler, unlike the Motorola assembler, uses a postfix syntax. This means the operands precede the instructions,

```
sp  )+   d0                    .l  move,
<operands>                     <instruction>
```

(or, more specifically)

```
sp  )+         d0              .l  move,
<source>    <destination>  <instruction>
```

the register/data specifications precede the address mode specification,

```
sp                                )+
<register/data specification> <address mode specification>
```

and the instruction size precedes the instruction:

```
.l                                move,
<size>                            <instruction>
```

Code Definitions

In most FORTH implementations, a FORTH definition is composed of intermediate threading information (FORTH instructions) instead of directly executable machine code instructions. The term 'code definition' is used to refer to those definitions which consist of machine code instructions.

The word **code** is used to create named tForth assembly definitions. **code** (i) creates a name in the dictionary for <name> (so that the code definition can be found in the future), (ii) tells the system that all subsequent words should be compiled rather than executed, (iii) and puts the assembler vocabulary **asm68** (the vocabulary which contains all of the assembler instructions) first in the search order.

Here is an example of a tForth code definition:

```
code  swap  ( n1  n2  -  n2  n1  |  Swaps the top two stack items. )
        sp  )+  d0  move,        ( take n2 off of the stack and put )
                                 ( it in the d0 register )
        sp  )+  d1  move,        ( take n1 off of the stack and  put )
                                 ( it in the d1 register )
        d0  sp  -)  move,        ( put the contents of the d0 )
                                 ( register on the stack )
        d1  sp  -)  move,        ( put the contents of the d1 )
                                 ( register on the stack )
next;                            ( end this assembly routine )
```

The standard formats for a code definition are:

```
code <name>  ...  ...  next;
code <name>  ...  next,  ...  next,  ...  next;

code <name>  ...  ;c
code <name>  ...  next,  ...  next,  ...  ;c
```

The word next; ('next-semi-colon') inserts an instruction into the code definition which during execution will (i) help the interpreter switch between the execution of machine language instructions and the exection of FORTH instructions, (ii) check the return stack to make sure it has not been corrupted, and (iii) remove asm68 from the search order. next; and next, ('next-comma', described below) are used in code definitions that are called from FORTH and return to FORTH when they terminate execution.

The word next, is used when more than one exit from a code definition is required. next, is similar to next; in that it also inserts a 'return to FORTH' instruction into the code definition being constructed. next, does not check the return stack or remove the assembler vocabulary from the search order.

The word ;c ('semi-colon-c') is used in code definitions which do not exit at the end or in code definitions which should not return to FORTH after execution of the code definition terminates (perhaps a code definition which is called from another code definition instead of called from FORTH). ;c (i) checks the return stack and (ii) removes the assembler vocabulary from the search order. The word **next;** is defined:

```
: next;  (  -  )
      next,          ( compile an 'exit to FORTH' instruction )
      ;c  ;          ( check the stack and deactivate  asm68  )
```

Creating Unnamed Assembly Code Fragments

The tForth assembler also includes provisions for the creation of assembly language code fragments. A code fragment is an assembly language routine which has no dictionary entry, and thus cannot be referenced by name. A common format for the use of code fragments is shown below:

```
frag <integername> to  ...  c;
```

frag puts the address where the first instruction in the code fragment will be located on the stack. The sequence '<integername> to', although not required, is usually used to save the address of the code fragment away for future reference. c; is used to terminate a code fragment.

Note: A tForth code definition (a set of assembly language instructions) must always be preceded by either code or .frag and must always be followed by either next; or ;c .

SPECIFYING ASSEMBLER OPERANDS

A complete assembler instruction consists of a 68000 assembly instruction and the source and/or destination operand on which the instruction will operate. A complete assembler source or destination operand consists of a register or data specification and an addressing mode specification.

Register Specification

The following symbols are used in source and destination operands for register specification in both the tForth assembler and in the M68000 manual:

SYMBOL	MEANING
an	Address register, 'n' specifies the register number
dn	Data register, 'n' specifies the register number
rn	Address or data register, 'n' specifies register #
pc	Program counter
sr	Status register
ccr	Condition code half of status register
sp	Active stack pointer (user or supervisor)
usp	User stack pointer
ssp	Supervisor stack pointer
d8	8 bit displacement (-80...7F hex)
d16	16 bit displacement (-8000...7FFF hex)
d32	32 bit displacement (-8000000...7FFFFFFF hex)
xxxx	Number, size determined by instruction size.
addr16	16 bit address.
addr32	32 bit address.

SPECIAL NOTE: During execution, tForth uses certain 68000 registers for special purposes. For code readability, the following registers have been assigned special symbols which are recognized by the tForthassembler (the usage of these registers is explained in more detail in the compilation discussion included in the technical reference section of this manual):

REGISTER	SYMBOL	CONTENTS
d7	bp	Address of the base of the token table.
d6	iv	Address of the value of the current integer.
d5	sa	Zeroth nesting starting address.
d4	ct	Address of the current token.
a7	sp	Parameter stack pointer.
a6	rp	Return stack pointer.
a5	ip	Interpretation pointer.
a4	nx	Next pointer.
a3	np	Nest pointer.
a2	vp	Pointer to the code for integer .

In the swap code definition presented at the start of this discussion the symbols 'd0' and 'd1' were used to specify data register zero and data register one and the symbol 'sp' was used to specify the stack pointer (the parameter stack pointer in tForth).

Address Modes

Address modes are used to specify the location of instruction operands or data to the microprocessor. The 68000 supports 12 address modes. The table below lists each of the 12 address modes and the Motorola and tForth assembler syntax used for each address mode:

#	MOTOROLA SYNTAX	tFORTH SYNTAX	DESCRIPTION
1.	Dn	dn	Data register direct.
2.	An	an	Address register direct.
3.	(An)	an)	Address register indirect.
4.	(An)+	an)+	Address register indirect with postincrement.
5.	-(An)	an -)	Address register indirect with predecrement.
6.	d16(An)	an d16)d	Address register indirect with displacement.
7.	d8(An,Rn.W) d8(An,Rn.L)	an rn.w d8 xw)d an rn.l d8 xl)d	Address register indirect with index.
8.	xxx.W	addr16	Absolute short address.
9.	xxx.L	addr32	Absolute long address.
10.	d16(PC)	d16 pc)d	Program counter with displacement.
11.	d16(PC,Rn.W) d16(PC,Rn.L)	rn.w d16 pc,xw)d rn.l d16 pc,xl)d	Program counter with index.
12.	#xxxx	xxxx #n	Immediate data.

Examples

Since the 68000 'move' instruction allows its source operand to be specified with the use of any of the address modes listed above, it is a good instruction to use when providing examples of the usage of the address modes (the source operand is the leftmost operand in the tForth assembler syntax):

```
Address mode #1:      d0                  sp )+ .l move,
Address mode #2:      a0                  sp )+ .l move,
Address mode #3:      a0 )                sp )+ .l move,
Address mode #4:      a0 )+               sp )+ .l move,
Address mode #5:      a0 -)               sp )+ .l move,
Address mode #6:      a0 4 )d             sp )+ .l move,
Address mode #7:      a0 d2 4 xw)d        sp )+ .l move,
Address mode #8:      7ce0                sp )+ .l move,
Address mode #9:      420000              sp )+ .l move,
Address mode #10:     4e00 pc)d           sp )+ .l move,
Address mode #11:     d4 7F8 pc,xl)d      sp )+ .l move,
Address mode #12:          400 #n         sp )+ .l move,
```

Address modes 1 and 2 are 'register direct addressing modes'. These addressing modes are used when the operand is in either an address or data register.

Address modes 3 through 7 are 'memory address modes'. These address modes are used when the operand is located somewhere in memory. These address modes are evaluated to produce the address in memory where the operand is located.

Address modes 8 and 9 are used when the address of the operand is specified explicitly to the instruction.

Address modes 10 and 11 are special versions of the 'memory address modes'. Although they function similarly to address modes 6 and 7, they are put in a special class because it is assumed that these addressing modes will be used to access locations in the program code area rather than in the program data area.

Address mode 12 is used when the operand is specified explicitly to the instruction.

For more detailed information on how address modes are evaluated, please refer to the 'Program/Data References' (section 2.7) of the M68000 manual.

Address Mode Categories

Certain 68000 instructions can only use a subset of the available addressing modes. On the individual instruction glossary pages in the M68000 manual, the following classifications are used to categorize the addressing modes which a particular instruction may use:

1. DATA ADDRESSING ADDRESS MODES
   If an effective address mode may be used to refer to data operands, it is considered a data addressing effective address mode.

dn	an)
an)+	an -)
an d16)d	an rn.w d8 xw)d
an rn.1 d8 xl)d	addr16
addr32	d16 pc)d
rn.w d16 pc,xw)d	rn.1 d16 pc,xl)d
xxxx #n	

2. MEMORY ADDRESSING ADDRESS MODES

If an effective address mode may be used to refer to memory operands, it is considered a memory addressing effective address mode.

```
an )                    an )+
an -)                   an d16 )d
an rn.w d8 xw)d         an rn.l d8 xl)d
addr16                  addr32
d16 pc)d                rn.w d16 pc,xw)d
rn.l d16 pc,xl)d        xxxx #n
```

3. ALTERABLE ADDRESSING ADDRESS MODES

If an effective address mode may be used to refer to alterable (writable) operands, it is considered an alterable addressing effective address mode.

```
dn                      an
an )                    an )+
an -)                   an d16 )d
an rn.w d8 xw)d         an rn.l d8 xl)d
addr16                  addr32
```

4. CONTROL ADDRESSING ADDRESS MODES

If an effective address mode may be used to refer to memory operands without an associated size, it is considered a control addressing effective address mode.

```
an )                    an d16 )d
an rn.w d8 xw)d         an rn.l d8 xl)d
addr16                  addr32
d16 pc)d                rn.w d16 pc,xw)d
rn.l d16 pc,1)d
```

Operand Size

The size of the operand to be used by a 68000 instruction can be specified with the use of the assembler words .b , .w , and .l . .b means the source and destination operands are 1 byte in size. .w means the source and destination operands are 2 bytes in size. .l means the source and destination operands are 4 bytes in size. If no operation size is specified, the assembler assumes the operands are 4 bytes in size.

HOW OPERAND SIZE AFFECTS REGISTER OPERATIONS

```
d0   d1   .b move,   ( Move the lowest order byte of )
                     ( register d0 to register d1. )
d0   d1   .w move,   ( Move the lowest order word of )
                     ( register d0 to d1. )
d0   d1   .l move,   ( Move the entire 4 bytes in register d0 )
                     ( to register d1. )
d0   d1     move,    ( Same as '.l move,'. )
```

How Operand Size Affects Memory Operations

```
a1  ) d1     .b move,    ( Move the byte of data located at )
                         ( address a1 into the lowest order )
                         ( byte of register d1. )
a1  ) d1     .w move,    ( Move two bytes of data into register d1. )
                         ( The byte at address a1 goes into the )
                         ( second lowest order byte in d1 and the )
                         ( byte a address a1+1 goes into the lowest )
                         ( order byte of d1. )
a1  ) d1     .l move,    ( Move the four bytes located in memory )
                         ( starting at address a1 into the d1 )
                         ( register.  The byte at a1 goes into the )
                         ( highest order byte of d1 and the byte )
                         ( at address a1+3 goes into the lowest )
                         ( order byte of d1. )
```

STRUCTURED ASSEMBLY LANGUAGE PROGRAMMING SUPPORT

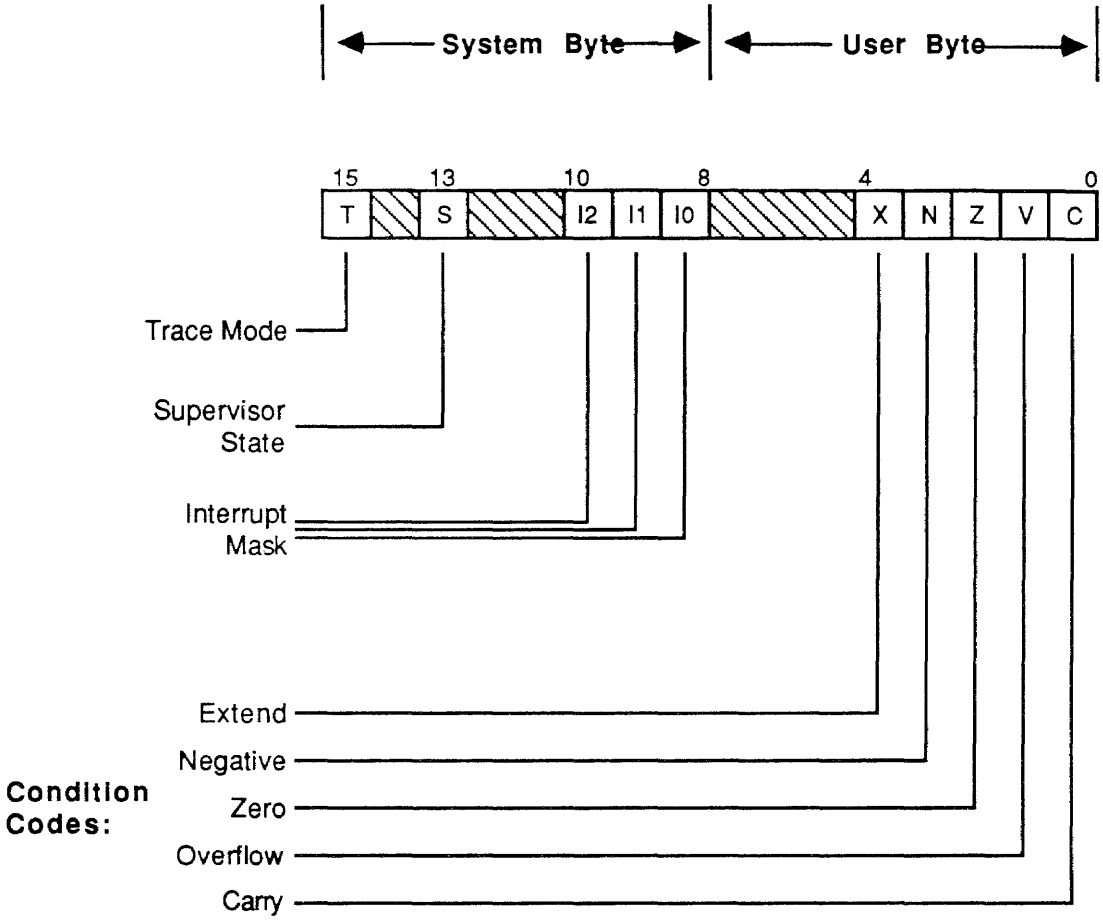
The tForth assembler allows special versions of the conditional and indefinite looping high-level FORTH program control structures to be included in assembly language code definitions. The assembler versions of the program control structures make decisions based on the microprocessor condition code state.

The Condition Code Register

The 'condition code register' (ccr), which is located in the lower order byte of the status register (see diagram on the following page), holds the condition code information. The 5 bits which represent the five possible condition codes (negative, zero, overflow, carry, and extend) are also shown. Certain 68000 instructions modify the condition codes to reflect the outcome of their operation. The condition codes can be combined, or used individually, to perform the following conditional tests:

tFORTH CONDITION CODE SYMBOL	CONDITIONAL TEST
tr	always true
nt	always not true or false
hi	high
ls	low or same
nc	carry clear, no carry
cs	carry set
ne	not equal
eq	equal
nv	overflow clear, no overflow
vs	overflow set
pl	plus
mi	minus
ge	greater or equal
lt	less than
gt	greater than
le	less or equal

Status Register



Using Conditional Test Structures in Assembly Language Words

The 'if...else...then' conditional program control structure has the following format when used in code definitions:

```
cc if, ... then,
cc if, ... else ... then,
```

The 'cc' is used to denote a condition code symbol. The word @ , which is used to fetch a four byte value from a memory address, is a tForth code definition which uses the 'if,...then,' assembly language program control structure:

```
code @ ( a - n )
   sp ) a0      move,     ( put address in the a0 register )
   a0 d0      .w move,    ( move the lower word of the address )
                          ( into the d0 register )
   1 #n d0    .b lsr,     ( shift the least significant bit out of )
                          ( the d0 register and into the condition )
                          ( code 'carry' bit )
   cs if,                 ( IF the bit was a '1', the address was )
                          ( odd so the long word must be fetched )
                          ( one byte at a time)
      a0 )+ sp )      .b move, ( if the condition was met )
      a0 )+ sp 1 )d .b move, ( these instructions will be )
      a0 )+ sp 2 )d .b move, ( executed. )
      a0 )+ sp 3 )d .b move,
      next,               ( intermediate exit to FORTH )
   then,                  ( if the condition was not met above )
                          ( the next instruction will be executed )
   a0 ) sp )       move,  ( move 4 bytes at once since data is on )
                          ( even byte address )
next;                     ( check return stack, deactivate )
                          ( asm68 , and exit to FORTH )
```

All of the FORTH comparison operators also use the 'if,...then,' assembly language conditional program control structure. Here is the code definition for the word max :

```
code max ( n1 n2 - n3 | Compare n1 to n2, return the greater. )
      sp )+ d0     move,    ( get parameter 'n2' )
      sp ) do      cmp,     ( subtract n2 - n1 )
      gt if,                ( IF the condition codes indicate that )
                            ( n2 is greater than n1, put n2 in )
                            ( the top stack position )
         d0 sp )   move,
      then,                 ( otherwise, leave n1 on top of stack )
next;                       ( terminate code definition )
```

Using Indefinite Loop Structures in Assembly Language Words

The assembly language versions of the indefinite looping program control words are used as follows:

```
begin,  ...  again,
begin,  ...  cc  while,  ...  [  leave,  ]  ...  again,

begin,  ...  [  leave,  ]  ...  cc  until,
begin,  ...  cc  while,  ...  [  leave,  ]  ...  cc  until,

begin,  ...  [  leave,  ]  ...  dn  cc  -until,
begin,  ...  cc  while,  ...  [  leave,  ]  ...  dn  cc  -until,
```

The words in square brackets ([ ]) denote optional program control words which may be used. All of these constructs, except for the 'begin,...dn cc -until' construct, should be familiar. -until, is a special assembly language program control word which utilizes the 68000 'DBcc' (decrement and branch) instruction. -until, takes two inputs, a data register specification and a condition code specification. Each time through the loop, if the condition is NOT met, the contents of the specified data register will be decremented. A -until, loop will continue until either the count in the data register reaches -1, or until the condition is met.

The tForth word fill uses a -until, loop:

```
code fill  (  a  n  b  -  )
   sp  )+  d0     move,        ( put fill char in the d0 register )
   sp  )+  d1  .w move,        ( put high word of count in d1 register )
   sp  )+  d2  .w move,        ( put low word of count in d2 register )
   sp  )+  a0     move,        ( put address in a0 register )
   0          .b bra,          ( one-time branch down to the label '0' )
   begin,                      ( perform outer loop until the upper )
                               ( word of the count is reduced to -1 )
      begin,                   ( perform inner loop until the lower )
                               ( word of the count, in d2, is -1 )
            d0  a0  )+  .b move,
            0 :1               ( create a label #0, discussed below )
      d2  nt
      -until,
   d1  nt
   -until,
next;
```

fill code definition demonstrates that assembly language program control structures may be nested. fill used nested loops because the 'DBcc' instruction can only work with a 16 bit count value in the data register. The nested loops allow the user to pass a 32 bit count value to fill . Since the nt condition code always evaluates to false, the -until, loop never terminates due to the condition code. When the nt condition code is used in an -until, loop, the loop will only terminate when the count in the data register is reduced to -1.

Labels

Ten local labels are allowed within a single code definition. Labels are defined using :1 :

```
0 :1
3 :1
9 :1
```

Labels are defined in the code definition at the spot which the label should mark. In the definition of fill above, a label was placed inside of the inner 'begin,...-until,' loop.

The number passed to :1 is used to identify the label. Label numbers must be between 0 and 9. Three 68000 instructions may be passed label numbers: bra, , bsr, , and cc bra, :

```
2 :1 d3 clr, ... 2 bra,     ( unconditional branch back to )
                            ( the clr, instruction located at )
                            ( label 2 )

2 :1 d3 clr, ... 2 eq, bra, ( conditional branch back to the )
                            ( clr, instruction, branch only )
                            ( occurs if the condition is met )

2 :1 d3 clr, ... 2 bsr,     ( unconditional branch to a )
                            ( subroutine )

2 bra, ... 2 :1 d3 clr,     ( forward and backward branching are)
                            ( allowed )
```

THE **movem,** INSTRUCTION

The 68000 **movem,** instruction is used to move multiple values to and from registers at once. For example, to move several registers onto the parameter stack:

```
(regs  d4  d5  d6  d7  a3  a4  a5  a6  to)  sp  -)  movem,
```

To restore the contents of the registers from the parameter stack:

```
(regs  d4  d5  d6  d7  a3 a4  a5  a6  from)  sp  )+  movem,
```

tFORTH ASSEMBLER WORDS

These are the available tForth assembler words (all are located in the **asm68** vocabulary):

#n	(regs	)	)+	)d
-)	-until,	.b	.l	.w
:1	;c	a0	a1	a2
a3	a4	a5	a6	a7
abcd,	add,	adda,	addi,	addq,
addx,	again,	and,	andi,	asl,
asr,	bchg,	bclr,	begin,	bp,
bra,	bset,	bsr,	btst,	ccr
chk,	clr,	cmp,	cmpa,	cmpi,
cmpm,	cs	ct	d0	d1
d2	d3	d4	d5	d6
d7	dbra,	divs,	divu,	else,
eor,	eori,	eq	exg,	ext,
from)	ge	gt	hi	if,
ip	iv	jmp,	jsr,	le
lea	leave,	link,	ls	lsl,
lsr,	lt	mi	move,	movem,
movep,	moveq,	muls,	mulu,	nbcd,
nc	ne	neg,	negx,	next,
nop,	not,	np	nt	nv
nx	or,	ori,	pc)d	pc,xl)d
pc,xw)d	pea,	pl	reset,	rol,
ror,	roxl,	roxr,	rp,	rtd,
rte,	rtr,	rts,	sa	sbcd,
set,	sp,	sr	stop,	sub,
suba,	subi,	subq,	subx,	swap,
tas,	then,	to)	tr	trap,
trapv,	tst,	unlk,	until,	usp
vp	vs	while,	xl)d	xw)d

ARITHMETIC OPERATORS

***** (n1 n2 - n3)
('times')
Multiplies n1*n2 and leaves the 32-bit result on top of the stack.

***/** (n1 n2 n3 - n4)
('times-divide')
First, n1 is multiplied by n2, leaving a 64-bit intermediate result on the stack. The intermediate result is then divided by n3, leaving the 32-bit quotient, n4, on the stack. The 64-bit intermediate result allows this operation to respond with greater precision than the equivalent sequence: n1 n2 * n3 / .

***/mod** (n1 n2 n3 - n4 n5)
('times-divide-mod')
First, n1 is multiplied by n2, leaving a 64-bit intermediate result on the stack (the intermediate result occupies two stack positions). The intermediate result is then divided by n3, leaving the 32-bit remainder, n4, and the 32-bit quotient, n5, on the stack.

+ (n1 n2 - n3)
('plus')
Adds n1 plus n2 and leaves the 32-bit result on the stack.

- (n1 n2 - n3)
('minus')
Subtracts n1 minus n2 and leaves the 32-bit result on the stack.

-1 (- -1)
('minus-one')
Puts the commonly used constant value '-1' on top of the parameter stack.

/ (n1 n2 - n3)
('divide')
Divides n1 by n2 and leaves the 32-bit quotient on the stack.

/mod (n1 n2 - n3 n4)
('divide-mod')
Divides n1 by n2 and leaves the 32-bit remainder, n3, and the 32-bit quotient, n4, on the stack.

0 (- 0)
('zero')
Puts the commonly used constant '0' on top of the parameter stack.

1 (- 1)
Puts the commonly used constant '1' on top of the parameter stack.

1+ (n1 - n2)
('one-plus')
Adds one to the number on top of the stack.

1- (n1 - n2)
('one-minus')
Subtracts one from the number on top of the stack.

2* (n1 - n2)
('two-times')
Multiplies the number on top of the stack by two.

2+ (n1 - n2)
('two-plus')
Adds two to the number on top of the stack.

2- (n1 - n2)
('two-minus')
Subtracts two from the number on top of the stack.

2/ (n1 - n2)
('two-divide')
Divides the number on top of the stack by two.

abs (n - |n|)
('absolute')
Returns the absolute value of the number on top of the stack.

mod (n1 n2 - n3)
n1 is divided by n2 and the 32-bit remainder, n3, is left on top of the stack.

negate (n - -n)
Returns the two's complement of n, i.e. n is subtracted from zero (0-n).

shl (n1 n2 - n3)
('shift-left')
Shifts the bits in 'n1' 'n2' bits to the left. Leaves the 32-bit result, 'n3', on the parameter stack.

shr (n1 n2 - n3)
('shift-right')
Shifts the bits in 'n1' 'n2' bits to the right. Leaves the 32-bit result, 'n3', on top of the parameter stack.

um* (u1 u2 - u3)
('u-m-times')
Multiplies the unsigned values u1*u2 and returns the 32-bit unsigned result, u3, on top of the stack.

um/mod (u1 u2 - u3 u4)
('u-m-divide-mod')
The 32-bit unsigned value u1 is divided by the 32-bit unsigned value u2. The 32-bit unsigned remainder, u3, and the 32-bit unsigned quotient, u4, are left on top of the stack.

and (n1 n2 - n3)

Performs a bit-by-bit logical and using n1 and n2. returns the 32-bit result (n3) on the parameter stack. The FORTH code definition for **and** is shown below:

```
code and ( n1 n2 - n3 )
     sp )+ d0 move,     ( take the 32-bit value [n2] off )
                        ( the top of the parameter stack )
                        ( and put in the d0 register. )
     d0 sp ) and,       ( perform an and operation. )
                        ( using the 32-bit value on top of )
                        ( the stack [n1] and the value in )
                        ( the d0 register [n2]. replace )
                        ( the value on top of the stack )
                        ( with the result )
     next;              ( return )
```

not (n1 - n2)

Takes the ones complement of the 32-bit value on top of the parameter stack. Returns the 32-bit result on top of the parameter stack. The FORTH code definition for **not** is shown below:

```
code not ( n1 - n2 )
     sp ) not,          ( take the ones complement of )
                        ( the 32-bit value on top of the )
                        ( parameter stack. )
     next;              ( return )
```

or (n1 n2 - n3)

Performs a bit-by-bit logical or using n1 and n2. Returns the 32-bit result (n3) on the parameter stack. The FORTH code definition for **or** is shown below:

```
code or ( n1 n2 - n3 )
     sp )+ d0 move,     ( take the 32-bit value [n2] off )
                        ( the top of the parameter stack )
                        ( and put in the d0 register. )
     d0 sp ) or,        ( perform an or operation. )
                        ( using the 32-bit value on top of )
                        ( the stack [n1] and the value in )
                        ( the d0 register [n2]. replace )
                        ( the value on top of the stack )
                        ( with the result )
     next;              ( return )
```

xor (n1 n2 - n3)

Performs a bit-by-bit logical xor using n1 and n2. Returns the 32-bit result on the parameter stack. The FORTH code definition for **xor** is shown below:

```
code  xor  ( n1  n2  -  n3  )
                sp  )+  d0  move,    ( take the 32-bit value [n2] off )
                                     ( the top of the parameter stack )
                                     ( and put in the d0 register. )
                d0  sp  )  eor,      ( perform an exclusive or operation. )
                                     ( using the 32-bit value on top of )
                                     ( the stack [n1] and the value in )
                                     ( the d0 register [n2].  replace )
                                     ( the value on top of the stack )
                                     ( with the result )
           next;                     ( return )
```

COMPARISON OPERATORS

0< (n - f)
('zero-less-than')
Returns a true (-1) flag if n is less than zero.

0= (n - f)
('zero-equal')
Returns a true (-1) flag if n is equal to zero.

< (n1 n2 - f)
('less-than')
Returns a true (-1) flag is n1 is less than n2.

= (n1 n2 - f)
('equal')
Returns a true (-1) flag if n1 is equal to n2.

> (n1 n2 - f)
('greater-than')
Returns a true (-1) flag if n1 is greater than n2.

<> (n1 n2 - f)
('not-equal')
Returns a true (-1) flag if n1 is not equal to n2.

inrange (n1 n2 n3 - f)
Returns a true (-1) flag if the value n1 is greater than or equal to the lower limit n2 and less than or equal to the upper limit n3 (i.e. n2 < n1 < n3).

max (n1 n2 - n3)
Compares n1 and n2 and returns the greater value.

min (n1 n2 - n3)
Compares n1 and n2 and returns the lesser value.

u< (u1 u2 - f)
('u-less-than')
Returns a true (-1) flag if the unsigned value u1 is less than the unsigned value u2.

.s (-)
('dot-s')
Prints a nondestructive display of the number of items on the parameter stack:

```
3 5 4  .s  3 5 4  ok
```

2drop (n1 n2 -)
Discards the top two items on the parameter stack. The FORTH code definition for **2drop** is shown below:

```
code 2drop   ( n1 n2 -   )
     8 #n sp addq,     ( increment the parameter stack )
                       ( pointer by 8, i.e. skip over the )
                       ( top two items on the stack and )
                       ( point at the previous item )
next;
```

2dup (n1 n2 - n1 n2 n1 n2)
Duplicates the top two items on the parameter stack. Leaves the duplicates on top of the parameter stack. The FORTH code definition of **2dup** is shown below:

```
code 2dup   ( n1 n2 - n1 n2 n1 n2 )
     sp 4 )d sp -) move,       ( put a copy of the second  )
                               ( 32-bit value on the stack )
                               (  on top of the stack )
     sp 4 )d sp -) move,       ( do the same thing again. )
next;
```

>r (n1 - | return stack: - n1)
('to-r')
Removes n1 from the parameter stack and places it on the return stack.

?dup (n - n n) or (0 - 0)
('question-dupe')
Duplicates the value on top of the stack if it is nonzero.

?stack (- f)
('question-stack')
Checks the status of the parameter stack. A false (0) flag will be returned if the stack is ok. A -1 will be returned if the stack is empty (if a stack underflow condition exists) and a 1 will be returned if the stack is full.

?stackerr ('question-stack-error)
(-)
Uses **?stack** to check for stack underflow or overflow. If one of these conditions has occurred **?stackerr** will issue an appropriate error message and abort.

depth (- n)
Returns the number of items on the stack.

drop (n1 -)
Discards the top item from the stack. The FORTH code definition for **drop** is shown below:

```
code drop ( n1 - )
     4 #n sp addq,     ( increment the parameter stack pointer )
                       ( by four, i.e. skip over the top item on )
                       ( the stack and point at the previous
                       ( item )
next;
```

dup (n1 - n1 n1)
Duplicates the value on top of the parameter stack. Leaves the copy on top of the stack. The FORTH code definition for **dup** is shown below:

```
code dup ( n1 - n1 n1 )
     sp ) sp -) move,          ( move a copy of the 32-bit )
                               ( value on top of the )
                               ( stack, onto the stack )
next;                          ( return )
```

i (- n)
Puts a copy of the top item on the return stack on top of the parameter stack. During execution of a do...loop, the top item on the return stack is the index for the current loop.

over (n1 n2 - n1 n2 n1)
Places a copy of the second item on the stack on top of the stack.

r> (n - | return stack: - n)
('r-from')
Transfers the top item on the parameter stack to the top of the return stack.

r@ (- n)
('r-fetch')
Puts a copy of the top item on the return stack on top of the parameter stack. **r@** performs the same function as **i** but **r@** is normally used outside of do...loops.

rot (n1 n2 n3 - n2 n3 n1)
('rote')
Rotates the third item on the stack to the top of the stack.

swab (n1 - n2)
Exchanges the lower two bytes of the top value on the stack. Example:

```
hex
12345678   swab  .  12347856
```

swap (n1 n2 - n2 n1)

Exchanges the top two items on the parameter stack. The FORTH code definition for **swap** is shown below:

```
code swap   ( n1 n2 - n2 n1 )
     sp )+ d0 move,   ( take n2 off the stack )
                      ( and place in the d0 register )
     sp )+ d1 move,   ( take n1 off the stack )
                      ( and place in the d1 register )
     d0 sp -) move,   ( put n2 on the stack )
     d1 sp -) move,   ( put n1 on top of the stack)
next;                 ( return )
```

+to (n1 n2 -)
('plus-to')
Format: n1 <integer or local variable name> +to
Adds n1 to the current value of the integer or local variable specified by name. +to discards the value of the integer or local variable, n2, which was place on the stack when the name of the integer or local variable was executed and uses the address in the iv register (see the integer section in the Technical Reference Manual for more information) to find the location where the current value of the integer or local variable is stored.

<loc0> (- n)
('brac-loc-zero')
A special fast word used to access the first local variable on the return stack. Moves a copy of the return stack pointer into the iv register and then places a copy of the item on top of the return stack, the contents of the first local variable, on top of the parameter stack.

<loc1> (- n)
('brac-loc-one')
A special fast word used to access the second local variable on the return stack. Moves a copy of the return stack pointer 4+ into the iv register and then places a copy of the second item on the return stack, the contents of the second local variable, on top of the parameter stack.

<local> (- n)
Generic word used to access the third, and all subsequent local variables on the return stack. Uses the offset pointed to by the ip register to index into the return stack to find the contents of the desired local variable. Puts the address of the local variable in the iv register and puts the value of the local variable on top of the parameter stack.

<locals> (-)
First local variable word compiled into a tForth word which uses local variables. Creates a storage area on the return stack which the local variables will use to temporarily hold their values.

addr (n - a)
('adder')
Format: <name of integer> addr
Returns the address of the storage location for the integer specified by name.

int0 (- n)
Runtime code for integers located in integer tier 0. Puts the address of the integer's storage location in the iv register and places the current value of the integer on top of the parameter stack.

int1 (- n)
Runtime code for integers located in integer tier 1. See int0.

int2 (- n)
Runtime code for integers located in integer tier 2. See int0.

int3 (- n)
Runtime code for integers located in integer tier 3. See int0.

int4 (- n)
Runtime code for integers located in integer tier 4. See int0.

int5 (- n)
Runtime code for integers located in integer tier 5. See int0.

int6 (- n)
Runtime code for integers located in integer tier 6. See int0.

local (-)
Format: local <name for local variable>
Creates a named local variable. The local variable is not initialized to any value. Executing the name of the local variable will place the value of the local variable on top of the parameter stack.

off (n -)
Format: <name of local variable or integer> off
Sets the value of the local variable or integer specified by name to zero. The value of the integer or local variable placed on the parameter stack when the local variable or integer name was executed is discarded.

on (n -)
Format: <name of local variable or integer> on
Sets the value of the local variable or integer specified by name to negative one. The value of the integer or local variable placed on the parameter stack when the local variable or integer name was executed is discarded.

to (n1 n2 -)
Format: n1 <name of local variable or integer> to
Replaces the current value of the integer or local variable specified by name with the 32-bit value n1. The value placed on the stack when the local variable or integer name was executed is discarded.

MEMORY OPERATORS

! (n a -)
('store')
Stores the 32-bit value n into memory starting at address a.

+! (n a -)
('plus-store')
Adds the 32-bit value n to the 32-bit value located in memory starting at address a. The 32-bit value located in the memory location is replaced with the 32-bit addition result.

-?&set (a - f)
('minus-test-and-set')
Sets the sign bit on the byte located in memory starting at address a. This makes the byte a negative value. If the byte was already a negative value before **-?&set** , a true (-1) flag is returned. If the byte was a positive value, a false (0) flag is returned. The 68000 'TAS', 'test and set', instruction is used to implement this function. The 'TAS' instruction is special because it was designed such that the microprocessor cannot interrupt it between the testing and setting parts of its operation.

0?&set (a - f)
('zero-test-and-set')
Clears the sign bit on the byte located in memory starting at address a. This makes the byte a positive value. If the byte was already a positive value before **0?&set** , a true (-1) flag is returned. If the byte was a negative value, a false (0) flag is returned. The 68000 'TAS' instruction is used to implement this function (see **-?&set**).

@ (a - n)
('fetch')
Places a copy of the 32-bit value located in memory starting at address a on top of the parameter stack.

and! (b a -)
('and-store')
Performs a bit-by-bit logical AND operation using b and the byte located in memory starting at address a. The byte length result is stored into memory at address a.

c! (b a -)
('c-store')
The least significant 8 bits of the 32-bit value, b, on the parameter stack are stored into to memory starting at address a.

c@ (a - b)
('c-fetch')
Places the 8-bit value located in memory starting at address a in the least significant byte of a 32-bit value on top of the parameter stack. The upper three bytes (24 bits) are set to zero.

cmove (a1 a2 u -)
('c-move')
Moves the u bytes located starting at the source address a1 to the memory location starting at destination address a2. The general format is: 'source address' 'destination address' 'number of bytes to move' **cmove**.

dump (addr len -)

fill (a u b -)
Replaces the u bytes located in memory starting at address a with the byte value b. The general format is: 'start address' 'count' 'fill character' **fill**.

move (a1 a2 u -)
Special version of **cmove**.

not! (a -)
('not-store')
Takes the one's complement of the 8 bits of data located in memory starting at address a. The byte length result is stored into memory at address a.

or! (b a -)
('or-store')
Performs a bit-by-bit logical OR operation using b and the byte located in memory starting at address a. The byte length result is stored into memory at address a.

tip (a -)
Performs a byte write operation to the specified address. Used for toggling soft switches.

w! (w a -)
('word-store')
The least significant 16 bits of the 32-bit value, b, on the parameter stack are stored into to memory starting at address a.

w@ (a - w)
('word-fetch')
Places the 16-bit value located in memory starting at address a in the least significant word of a 32-bit value on top of the parameter stack. The upper 2 bytes (16 bits) are set to zero.

xor! (b a -)
('exclusive-or-store')
Performs a bit-by-bit logical XOR operation using b and the byte located in memory starting at address a. The byte length result is stored into memory at address a.

+loop Compiling: (-)
('plus-loop')
Executing: (n -)
Format: do ... n +loop
Program control structure used to implement definite loops. During execution, **+loop** adds 'n' to the current loop index.

<+loop> (n -)
('brac-plus-loop')
Run-time code for **+loop**. Adds the decrement value 'n' to the current loop count and then decides whether the loop should be continued or terminated.

<0bran> (f -)
('brac-zero-bran')
Run-time conditional branching primitive. A branch will occur if the flag passed to **<0bran>** is false (zero). Can only handle short (-81<n<80 hex) branching distances. Used by **while**, **until**, and **if**.

<0branl> (f -)
('brac-zero-bran-long')
Run-time conditional branching primitive. A branch will occur if the flag passed to **<0branl>** is false (0). Can be used for short and word (-8001<n<8000 hex) branching distances. Used by **while**, **until**, and **if**.

<0leave> (f -)
('brac-zero-leave')
Run-time code used to conditionally leave from a 'do...loop' or 'do...+loop' program control structure. The branch out of the program control structure will occur if the flag passed to **<0leave>** is false (0). Can only be used to branch forward short distances (n<80 hex). Currently, all **leave** and **while** branches use the long version of **<0leave>**. Also cleans up the return stack by reclaiming all of the return stack space used by the loop. Used to by used by **while**.

<0leavel> (f -)
('brac-zero-leave-long')
Run-time code used to conditionally leave from a 'do...loop' or 'do...+loop' program control structure. The branch out of the program control structure will occur if the flag passed to **<0leavel>** is false (0). Can be used to branch forward word length distances (-8001<n<8000 hex). Also cleans up the return stack by reclaiming all of the return stack space used by the loop. Used by **while**.

<abort"> (f a n -)
('brac-abort-quote')
Run-time code used by **abort"**. Expects to be passed the address 'a' and length 'n' of an error message string and a flag 'f' indicating whether or not the message should be displayed in the explain screen.

<bran> (-)
('brac-bran')
Run-time unconditional branching primitive. Always causes a branch to occur. Can only handle short (-81<n<80 hex) branching distances. Used by **again** and **else**.

<branl> (-)
('brac-bran-long')
Run-time unconditional branching primitive. Always causes a branch to occur. Can handle short <-81<n<80 hex) and word (-8001<n<8000) length branching distances. Used by **leave** and **else**.

<do> (n1 n2 -)
('brac-do')
Run-time code for **do**. Expects to be passed a loop index, n2, and limit, n1, on the parameter stack. Takes both values off of the parameter stack and then pushes first the limit onto the return stack and then the count (limit-index).

<leave> (-)
('brac-leave')
Run-time code used to unconditionally leave from a 'do...loop' or 'do...+loop' program control structure. Can only be used to branch forward short distances (n<80 hex). Currently, all leave and while branches use the long version of **<leave>**. Also cleans up the return stack by reclaiming all of the return stack space used by the loop. Used by **leave**.

<loop> (-)
('brac-loop')
Run-time code for **loop**. Subtracts one from the value on top of the return stack (the count value for a 'do...loop') and then checks to see if the count has reached zero. If the count has reached zero, **<loop>** removes the limit and count from the return stack and terminates the loop by allowing program execution to continue on the the code which follows the 'do...loop'. If the count has not reached zero, "jumps" back to the code which immediately follows the **do**.

<quit> (-)
('brac-quit')
Low-level word used by **quit**.

again (-)
Format: begin ... again
Used to implement endless loops. All code between the **begin** and **again** will be executed endlessly (**leave**, **while**, and **exit** can be used to terminate 'begin...again' endless loops).

abort" (f -)
('abort-quote')
Format: f abort" ccc"
If the flag passed to abort" is true (nonzero), a forced system abort process will occur. A beep will be issued, the message between the quotes will be displayed on the explain screen, the parameter stack will be cleared, and **quit** will be executed (to start FORTH running again). **abort"** must be used within a colon definition.

begin (-)
Format: begin ... again
begin ... until
Used to mark the start of an endless or indefinite program loop.

do Compiling: (-)
Executing: (n1 n2 -)
Format: n1 n2 do ... loop
n1 n2 do ... n3 +loop
Marks the start of a definite program loop. During execution, **do** takes the index 'n2' (start count) and limit 'n1' (end count) for the loop from the parameter stack and transfers the limit and the loop count (limit-index) to the return stack.

else Compiling: (-)
Executing: (f -)
Format: if ... else ... then
Inner decision point in the 'if...else...then' conditional program control structure. During execution, if the flag passed to **else** is true (nonzero), the code between the **else** and the then will be executed. Otherwise, program execution will continue on to the code which immediately follows the **then**.

execute (n -)
Executes the word corresponding to the token 'n' passed on the stack. Example:

```
' .s execute empty
```

exit (-)
Immediately and unconditionally terminates execution of the current definition and transfers control to the definition which contains the current definition.

if

Compiling: (-)
Executing: (f -)
Format: if ... then
if ... else ... then
Marks the start of the 'if...then' or 'if...else...then' conditional program control structures. During execution, if the flag passed to **if** is true (nonzero), the code between the **if** and the **then**, or the code between the **if** and the **else** will be executed. Otherwise, program execution will be routed to the code which immediately follows the **then** (if the 'if...then' structure is being used) or to the code between the **else** and the **then** (if the 'if...else...then' structure is being used).

interpret (a l -)

interpret is the main word involved in the running FORTH. **interpret** performs the following actions:

1. Takes as inputs the address and length of a block of user input text.
2. Advances through the text, word by word. The word **word** is used to isolate individual input "words" (a sequence of characters surrounded by spaces or tabs). Each time **word** is used it will return the address and length of the next word in the input text block to **interpret**. The **in** system variable is used to mark **word**'s progress through the input text.
3. Next, **interpret** passes the address and length returned by **word** to **find**. **find** will check to see if the string represented by the address and length contains the name of a word which can be found in the dictionary using the current vocabulary search order. If the system is in the compiling state, the word will be compiled into the definition currently being constructed. If the system is not in the compiling state, the word will be executed immediately (using **execute**).
4. If the string represented by the address and length does not contain the name of a FORTH word, **interpret** will pass the string address and length to **number**. **number** will try to convert the string to a number. If the number conversion process is successful and the system is in the compiling state, the converted number will be compiled as a literal into the definition currently being compiled. If the system is not in the compiling state, it will be placed immediately on the parameter stack.
5. If the string cannot be found in the dictionary, and cannot be converted to a number, **interpret** will issue an error message to indicate that it does not recognize the input.
6. If there is more user input text to process, **interpret** will repeat the steps above. If the user input text has been exhausted, **interpret** will terminate execution and let **quit** (the word which calls **interpret**) get more user input.

leave (-)
Immediately and unconditionally reroutes program execution out of the current "looping" program control structure. May be used in 'begin' loops or in 'do' loops.

loop (-)
Format: do ... loop
Marks the end of the 'do...loop' definite loop program control structure. During execution, **loop** will decrement the loop count by one and compare the new count to zero. If the count has reached zero, **loop** will terminate the loop by routing program execution to the code which immediately follows it. Otherwise, **loop** will route program execution back to the code which immediately follows **do**.

nest (-)
Used by all words which start program control structures. If a program control structure is being used interactively, **nest** compiles an assembly language "jump to the nesting routine" instruction, records the address of the instruction, and increments the nesting level by one. This address will be used later when the temporarily compile code must be moved to the execution buffer for immediate execution. If a program control structure is not being used interactively, **nest** will simple increment the nesting level, stored in the system integer **nesting**, by one. See **unnest**.

quit (-)
quit is the word which runs FORTH. Clears the return stack and puts the system in the interpreting state. After **quit** is executed the system will be waiting for user input to interpret and execute. A high-level definition of **quit** is:

```
: quit ( - )
    begin
              ( clear the return stack )
              ( get a block of user input text )
              ( interpret the user input text )
              ." ok" cr
    again ; ( do this endlessly )
```

then (-)
Format: if ... then
if ... else ... then
Marks the end of the 'if...then' or 'if...else...then' conditional program control structures.

unnest (-)

Used by all words which end program control structures. Decrements the **nesting** system integer by one and, if **nesting** has been reduced to zero and the system is not in the compilation state, moves the temporarily compiled program control structure code up to the execution buffer and causes it to be executed immediately. If the system is in the compilation state, **unnest** simple decrements **nesting** by one.

until (f -)

Format: begin ... f until

Conditional exit/branching word used at the end of the 'begin...until' indefinite loop program control structure. If the flag passed to **until** is true (nonzero), **until** will terminate execution of the loop by allowing program execution to continue on to the code which immediately follows itself. If the flag is false (0), **until** will reroute program execution back to the code which immediately follows the **begin**.

while Compiling: (-)
Executing: (f -)

```
Format: begin ... while ( ... while ) ... again
        begin ... while ( ... while ) ... until
        do ... while ( ... while ... ) ... loop
        d0 ... while ( ... while ... ) ... +loop
```

Inner decision/branching point in the 'begin...until' , 'begin...again', 'do...loop', or 'do...+loop' program control structures. During execution, if the flag passed to **while** is true (nonzero), the code between the **while** and the next **while** **until**, **again**, **loop**, or **+loop** will be executed. If the flag is false, **while** will immediately reroute program execution out of the current loop (to the code which follows the next **until**, **again**, **loop**, or **+loop**).

{loop} (n1 n2 -)
('curly-loop')

Shared routine used by the loop termination words **loop**, **+loop** **until** and **again**. Used during compile time to compile the lower level branching primitives used by the loop termination words and to resolve and compile the delta branching distances used by the lower level branching primitives.

{while} (-)
('curly-while')

Shared routine used by the words used to exit from loop program control structures: **while** and **leave**. Compiles the lower level branching primitive used by the exit word and reserves a two byte space for the delta branch distance used by the branching primitive

"

Compile time: (-)
Run-time: (- addr len)
('quote')
Format: " ccc"
When used during compilation, lays the string between quotes, and the runtime code <"> , into the definition being compiled. At run time the address and length will be left on the parameter stack. When used interactively leaves the string characters in the 'tib' and returns the address and length on the stack. The first " must be surrounded on both sides by at least one space or tab. Caution: Always double-check for the presence of the closing " . If the closing " is missing, the compiler will continue appending program text into the string being constructed in the definition until either the dictionary fills up or until some other error message is generated.

"to

(addr1 n1 addr2 n2 -)
('quote-to')
Format: " ccc" <string name> "to
Stores the string specified by the address and length (addr1 and n1) into the string integer specified by name. The address and length (addr2 and n2) of the current string stored in the string integer, which were placed on the stack when the name of the string integer was executed, are discarded. "to adjusts the string integer's storage area size to accomodate the length of the new string data.

(

(-)
('paren')
Format: (ccc)
(is the FORTH commenting word. All characters between the starting left paren and the closing right paren are considered to be comments and are ignored by the FORTH compiler. (must be surrounded on both sides by at least one space or tab. Comments may not be nested, i.e., don't use parentheses within comment statements.

+bit7

(char - char')
('plus-bit-7')
Sets the seventh bit in the character byte.

-trailing

(addr len - addr len')
('minus-trailing')
Strips the trailing spaces from the string located at address

." (-)
('dot-quote')
Format: ." ccc"
May be used interactively or compiled into a definition. The compile-time action of ." is to lay the string between quotes into the definition being compiled. The run-time action of ." is to type the string between quotes out to the current output device. The ." must be surrounded by at least one space or tab. Example:

```
." Hello" Hello                    ( used interactively )

: SayHi ( - ) ." Hello again." ; ( compiled into a )
SayHi Hello again.                 ( definition. )
```

<"> (- addr len)
('brac-quote')
<"> is the run-time code for the word " . Pushes the address and length of the string on the stack.

<"to> (addr1 n1 addr2 n2 -> addr3 n3)
('brac-quote-to')
Run-time code for "to .

<demit> (char x y -)
('brac-display-emit')
Draw the character at position x,y on the screen.

<remit> (char -)
('brac-raw-emit')
Raw emit to the screen.

<word> (addr1 addr2 - addr3 n addr4)
('brac-word')
Lower level routine used by **word** . Looks for the next word in the input stream which is surrounded by at least one space. Takes the start address, addr1; and end address, addr2; of a region of text to search. Returns the address where the next search should commence, addr3; the length of the word found, n; and the address where the word found is located, addr4.

ascii (- n)
Format: ascii <char>
Returns the ASCII value of the single character which immediately follows it.

becomes
(-)

check (addr n -)
Format: <string integer name> check
Prints the ASCII values for each character in the string currently stored in the string integer specified by name. If the string integer is empty, an error message is displayed.

cr (-)
('c-r')
Emit a carriage return/linefeed to the current active output devices.

crlfscroll (-)
Emit a carriage return and linefeed. Also blank the new line out and scroll if necessary.

ctl (-)
('control')
Format: ctl <char>
Turns the character which immediately follows it into a control character by setting the three most significant bits in the character byte to zero.

demit (c -)
('display-emit')
Emit the character to the screen. If the character is a cr perform a carriage return/linefeed and scroll if necessary. If the character is the 'del' (delete) character erase the previous character on this line (if any).

eemit (c -)
('editor-emit')
Emit the character to the editor.

emit (c -)
Output the character to all active output devices. The allowable output devices are the screen (see **demit**), the parallel port (see **pemit**), the editor (see **eemit**), and the serial port (see **semit**).

key (- c)
Waits until a printable character (8<ascii code<80 hex) is typed at the keyboard. Returns the ASCII value of the character on the stack.

pemit (char -)
('parallel-emit')
Send the character out through the parallel port.

rub (-)
Erase the previous character on the current line (if any).

scanfor (c -)
Looks for the next word in the current input stream which is surrounded by the delimiter character, c. Sets the **in**, **str**, and **len** system variables.

space (-)
Emit a space to the current active output devices.

spaces (n -)
Emit 'n' spaces to the current active output devices.

word (-)
]] Looks for the next word in the current input stream which
] is surrounded by at least one space. Sets the **in**, **str**, and **len** system variables accordingly.

NUMERIC I/O WORDS

(n1 - n2)
('sharp')
Format: n <# ... # ... #>
Extracts the lowest order digit from the number on top of the stack and inserts it into the formatted numeric string being constructed in the pad.

#> (n1 - a n2)
('sharp-greater')
Format: n <# ... #>
Removes the number from the top of the stack and returns the address and length of the formatted numeric string which has been constructed in the pad (prepares the formatted numeric string for **type**).

#s (n - 0)
('sharp-s')
Format: n <# ... #s ... #>
Calls # until the number on top of the stack has been reduced to zero.

. (n -)
('dot')
Prints the signed value on top of the stack followed by a trailing space. The definition of . provides a good example of the use of the pictured numeric output operators:

```
: . ( n - | Output n as a signed or unsigned number
with 1 trailing space. )
     dup                ( duplicate the number )
     abs                ( get absolute value of number )
     <#                 ( start number formatting... )
              #s        ( convert all digits in number to )
                        ( ascii characters and insert in )
                        ( the string )
              swap      ( check the sign of the original number )
              sign      ( if it was negative, insert a '-' here )
     #>                 ( clean up stack, set stack for  type  )
     type               ( display the string )
     space     ;        ( follow numeric string by one space )
```

.r (n w -)
('dot-r')
Prints the signed value 'n' in a field which is 'w' spaces wide.

<# (n - n)
('less-sharp')
Format: n <# ... #>
Marks the start of a pictured numeric conversion process. The words **#**, **#>**, **#s**, **<#**, **hold**, and **sign** are all used to construct the formatted string in the **pad**.

decimal (-)
Selects base ten (decimal) as the base used for all numeric input/output conversions.

digit (n1 n2 - n3 c)
Extracts the least significant digit from the number, n1, on the stack (using the specified base, n2) and leaves ascii value for the digit, c, and the remaining number, n3 on the stack. digit performs the following actions: 1. takes the number n1 from the stack and divides it by the base, n2 2. leaves the quotient of the division, n3, and the ASCII value of the remainder, c, on top of the stack.

hex (-)
Selects base sixteen (hexadecimal) as the base used for all numeric input/output conversions.

hold (c -)
Format: <# ... ascii c hold ... #>
Inserts the character (represented by the ASCII value) on top of the stack into the formatted numeric string currently being constructed in the pad.

number (a n1 n2 - f | If conversion is not successful.)
(a n1 n2 - n3 f | If conversion is successful.)
Converts the string of length n1 located starting at address a to a number, n3, using base n2. If the string-to-number conversion is successful, the converted number and a true (-1) flag will be left on the stack. If the string-to-number conversion is not successful (non-numeric characters in the string) a false (0) flag will be left on the stack.

sign (n -)
If the number on top of the stack is negative, **sign** will insert a minus sign into the formatted numeric string being constructed in the **pad**.

u. (n -)
Prints the unsigned value on top of the stack followed by a trailing space.

u.r (n w -)
Prints the unsigned value 'n' in a field which is 'w' spaces wide.

<string> (- addr len)
('brac-string')
Run-time code for string integers created with the defining word **string**. Pushes the address and length of the string stored in the string integer on the stack.

: (-)
('colon')
Format: : <name> ...words... ;
Defining word used to create new definitions. Puts the system in the compiling state, creates a new dictionary header using <name>, sets the **smudge** bit in the dictionary header so the definition will not be visible until completed. All words between the <name> and the ; will be compiled into the definition. The run-time action of words created by : is to execute the words which comprise the definition.

array Compiling: (n -)
Executing: (- a)
Format: n array <arrayname>
During compile-time, **array** allocates 'n' bytes in the dictionary for an array of data and creates a header to mark the start of the data area. The run-time action of the child words created by array is to push the address of the start of the array data area on the stack.

```
64 array message
message . 293036
```

integer Compiling: (n -)
Executing: (- n)
Format: n integer <integername>
At compile-time **integer** creates a named 4-byte data location and initializes the location with the value 'n'. The run-time action of the child words created by integer is to push the current contents of their 4-byte storage location on the stack.

string Compiling: (a n -)
Executing: (- a n)
Format: " ccc" string <stringname>
At compile-time **string** creates a named, multi-byte string storage area in the dictionary and initializes the storage area with the characters between the quotes. The runtime action of the child words created by **string** is to push the address and length of the the string currently stored in the string storage area on the stack.

vocabulary (-)
Format: vocabulary <vocabname>
Create a new but inactive vocabulary. The name for the new vocabulary will reside in the vocabulary which was open when the new vocabulary was created. When the child word created by **vocabulary** (**<vocabname>**) is executed, it will place itself first in the vocabulary search order.

DICTIONARY MANAGEMENT WORDS

<addto> (n -)
('brac-addto')
Close the current open vocabulary and open the vocabulary specified by the token 'n'.

<becode> (n -)
('brac-becode')
Remove the code corresponding to the token 'n'.

<behead> (a -)
('brac-behead')
Remove the header located at address 'a'.

<bevoc> (n -)
('brac-bevoc')
Completely eliminate the vocabulary specified by the token 'n'.

<csize> (a - n)
('brac-code-size')
Returns the code size 'n', in bytes, of the word whose code is located at address 'a'.

<deactivate> (n -)
('brac-deactivate')
Removes the vocabulary specified by the token 'n' from the current search order (removes its token from the 'active' array, see **active**).

<empty> (n -)
('brac-empty')
Purges all words from the vocabulary specified by the token 'n'.

<eta> (a n - 0 | If token 'n' is not found.)
(a1 n - a2 | If token 'n' is found.)
Takes the vocabulary address 'a1' and the encoded token value 'n' and, if successful, returns the encoded token address.

<purge> (n -)
('brac-purge')
Removes the word corresponding to the token 'n' from the dictionary.

addto (-)
Format: addto <vocab-name>
Opens the vocabulary whose name immediately follows **addto**.

behead (-)
Format: behead <name>
Remove the header of the definition whose name immediately follows **behead**.

bevoc (-)
Format: bevoc <name of vocabulary>
Removes the vocabulary specified by name, and all words in the vocabulary, from the dictionary.

createvoc (a1 n - a2)
Create an empty vocabulary using the image of an empty vocabulary located starting at address 'a1' and assign it the token 'n'. Return address 'a2' is unused.

csize (- n)
('code-size')
Format: csize <name>
Returns the code size of the word specified by <name>.

deactivate (-)
Format: deactivate <vocab-name)
Removes the vocabulary whose name immediately follows **deactivate** from the current search order.

empty (-)
Purges all words from the current vocabulary. The words in the **forth** vocabulary cannot be purged.

emptyvoc (- addr)
Returns the address of the 18 decimal byte image of an empty vocabulary.

eta (token - addr f)
Tries to return the address of the token table entry for the token. If successful returns the token table entry address and a true (nonzero) flag. Otherwise, returns a false (0) flag.

existing (-)
Displays the names of and parents of all existing vocabularies.

forth (-)
This is the main 'tFORTH' vocabulary. It contains all of the 'standard' FORTH words supported by 'tFORTH' and all of the 'tFORTH' FORTH extension words. Execution of **forth** will cause the **forth** vocabulary to become the first vocabulary in the search order (its token will be placed first in the 'active' array).

invoc (a - n)
Returns the token 'n' of the vocabulary which contains address 'a'.

name (n -)
Print the name of the definition which corresponds to the token 'n'.

purge (-)
Format: purge <name>
Removes the word specifed by **<name>** from the dictionary.

recycle (n -)
Reclaims the token table space for the token 'n'.

retop (a -)
Lower level word used to open a vocabulary. Moves the upper half of the dictionary up so that the new top of dictionary is at address 'a'.

safety (a -)
Reclaim the token table space for the token whose header is located at address 'a'.

searched (-)
Display the vocabulary search order.

setcodesize (-)
Set the code size field for the current open vocabulary. Set the odd size flag if necessary.

vocab (-)
Move the current execution vocabulary to the top of the search order by placing its token at the start of the active **array**.

vocab? (token - f)
Returns a true (nonzero) flag if the token on top of the stack it the token for a vocabulary. Returns a false (0) flag otherwise.

vopen (token - addr)
Returns the address of the opening point for the vocabulary which corresponds to the token.

words (-)
Displays a list of all words in the vocabulary which is first in the search order.

COMPILATION WORDS

!csp (-)
('store-csp')
Used to save the return stack pointer value away before a compilation process occurs.

' (- token)
('tick')
Format: ' <name>
Returns the token for **<name>**:

```
' words . 1A5 ok
```

, (n -)
('comma')
Lays the 32-bit value 'n' into the next free location in the code area. The **here** pointer always points at the next free location in the code area. The here pointer is incremented by 4 bytes.

+table (n - a)
('plus-table')
Takes a token table entry number and calculates and returns the address of the corresponding token table entry field.

; (-)
('semi-colon')
Used to terminate colon definitions. If the colon definition does not use local variables, ; causes the word <;> to be compiled into a definition. If the colon definition does use local variables, ; causes the word **<;lp>** to be compiled into a definition.

<;> Parameter: (-) Return: (n1 n2 -)
('brac-semi')
Run-time word compiled by ; . Pops two word length return values off of the return stack. The first value popped, 'n2', is used to reconstruct the ip register. The second value popped is used to reconstruct the ct register.

<;lp> (-)
('brac-semi-local')
Run-time exit word compiled at the end of colon definitions in which local variables are used. Compiled by ; . Pops two word length return values off of the return stack (see <;>) and then reclaims all return stack local variable storage.

<find> (a1 a2 n1 - a2 f | If not found)
(a1 a2 n1 - a3 n2 t | If found)
Searches for the name specified by the string at address 'a2' of length 'n1' in the vocabulary which starts at address 'a1'. If the word is found in the vocabulary, <find> will return the dictionary header address 'a3' for the word, the token for the word 'n2' and a true flag (nonzero). If the word is not found in the vocabulary **<find>** will return the original name string address 'a2' and a false (zero) flag.

?csp (-)
('question-csp')
Compares the current return stack pointer to the return stack value saved away previously in the **csp** system integer. If the two addresses are not equal the system will abort with an " unpaired" message. The return stack pointer address is saved away at the start of the compilation of a colon definition (in :) and is checked at the end of compilation (by ;).

?pairs (-)
('question-pairs')
Checks for properly paired conditional statements. Aborts and issues an error message if it senses an improperly paired conditional.

align (-)
Aligns the **here** pointer to an even address boundary.

allot (n -)
Tries to allocate 'n' bytes in the code area of the currently open vocabulary. If no vocabularies are currently open, or if 'n' bytes are not available in the open vocabulary, the system will abort. **allot** allocates space by adding 'n' to the address stored in the **here** system integer.

assign (a1 a2 n -)
Assigns a token to and builds a header for a new definition in the vocabulary specified by the address 'a1' using the name located at the address 'a2' with the length 'n'.

backelse (n1 - n2 n1)
Used by **then** to backpatch a forward **else** branch offset. If the delta branch distance is short (-81<delta<80), the code between the **else** and the **then** will be shifted one byte towards lower memory and the shift distance, -1, will be returned as the second item on the stack, n2. If the delta branch distance is word length (-8001<delta<8000) no code movement will occur and a shift distance of 0 will be returned as the 'n2' parameter.

blit (- n)
('byte-literal')
Code definition which transfers the byte-length literal value pointed to by the instruction pointer to the parameter stack and increments the instruction pointer by one byte. Used by **literal**.

c' (- a)
('c-tick')
Format: c' <name>
Returns the address of the code field (code area) of the definition specified by **<name>**.

c, (c -)
('c-comma')
Compiles the byte length value 'c' into the next available location in the code area (at the address pointed to by the **here** pointer.)

compile, (n -)
('compile-comma')
Lays the token value passed on the stack into the dictionary at the current **here** address. Checks the size of the value. If the token value is greater than $100 (bigger than one byte), compile will **w,** the token value into the dictionary. If the token value is less than $100, **compile** will use **c,** to place the token into the dictionary.

create (-)
Format: create <name>
Assigns a token to and creates a header entry for <name> in the current open vocabulary.

decode (n - token)
Takes the encoded token number from the top of the stack, decodes it, and returns the decoded token number on top of the stack.

diff? (a1 a2 n -> 0 1 If strings match)
(a1 a2 n -> a3 -1 1 If strings don't match)
Compares the first 'n' characters in the strings located at addresses 'a1' and 'a2'. If the first 'n' characters in the two strings match, a false (0) flag is returned. If the first 'n' characters in the two strings do not match, a true (-1) flag and a pointer to the first dissimilar character in the first string (the string pointed to by 'a1'), 'a3', is returned.

doloc (- f)
Used by **interpret**. Only used within a colon definition. Checks to see if the word just extracted from the input stream belongs to a local variable. If the word is the name of a local variable, compiles the code which will place the value of the local variable on the stack during execution into the definition and returns a false (0) flag. If the word is not the name of a local variable, returns a true (nonzero) flag.

encode (n1 - n2)
Takes the decoded token number from the top of the stack, encodes it, and returns the encoded token number on top of the stack.

find (a n1 - n2 true 1 If found in search order)
(a n1 -> false 1 If not found in search order)
Searches the through the dictionary (uses the current search order) looking for the definition whose name matches the name at the address 'a' with length 'n1'. If a match is found, **find** will return a true (nonzero) flag and the token which corresponds to the definition. If a match is not found, **find** will return a false (0) flag. **find** uses the lower level word **<find>** .

fnderr (-)
('find-error')
Prints a "can't find" error and aborts.

forward (-)

freetoken (-)
Prints an "unassigned token" message and aborts.

immediate (-)
Sets the **immediate** bit (bit 6) of the most recently defined colon definition so that whenever the word is encountered during compilation, it will be compiled rather than executed. The address of the header entry for the most recently defined colon definition is kept in the newest system integer.

lit (- n)
Code definition which transfers the long-word (32-bit) literal value pointed to by the instruction pointer to the parameter stack. The instruction pointer, ip, is incremented by by 4 bytes. Used by **literal**.

literal (n -)
literal is used to compile constant data into a definition. **literal** will also compile the token of a word which will push the constant data onto the parameter stack when the definition is later executed. If the value can be represented with one byte of data, **literal** will compile the token for **blit** into the new definition. If the value can be represented with two bytes of data, **literal** will compile the token for **wlit** into the new definition. If the value can only be represented with 4 bytes of data, **literal** will compile the token of **lit** into the new definition.

n' (- a)
('n-tick')
Format: n' <name>
Returns the address of the dictionary header area for the word specified by **<name>**.

raddr (- a)
Copies the return information stored on the return stack. Uses the return information to calculate the address where the next token to be executed in the definition at the next higher execution level is located (calculates the previous location of the ip pointer). Used by **compile**.

recycledtoken
(- token)
recycledtoken checks to see if any previously assigned tokens are now available for re-assignment. If a previously assigned token is available, **recycledtoken** will return the token value on the stack. If no previously assigned tokens are available a token value of 0 will be returned.

same? (a1 a2 n -> f)
Returns a true (nonzero) flag if the first 'n' characters in the strings located at 'a1' and 'a2' are the same.

stub (-)
Format: stub <name>
Uses **create** to assign a token to and create a dictionary header for **<name>**. Stores a 0 in **<name>**'s token table entry so **<name>** will not have any corresponding code area.

w, (w -)
('w-comma')
Stores the word length value 'w' into the next available spot in the code area of the currently open vocabulary.

wlit (- n)
('w-lit')
Code definition which transfers the word-length (16-bit) literal value pointed to by the instruction pointer to the parameter stack and increments the instruction pointer by two bytes. Used by **literal**.

[(-)
('left-bracket')
Turns the FORTH compiler on.

['] (- token)
('brac-tick-brac')
Format: : <name> ... ['] <definition-name> ;
['] must be used within a colon definition. **[']** will return the token for the definition whose name immediately follows it in the colon definition.

```
: test ( - ) ['] words . ;
test 1A5 ok
```

[compile] (-)
('brac-compile-brac')
Compiles the token of the word which immediately follows it into the definition currently being constructed.

] (-)
('right-bracket')
Turns the FORTH compiler off.

!ptr (n delta -)
Store the value n into the save block area.

<load> (n -)
Reads block 'n' from disk into memory and interprets its contents.

<rblock> (addr b# - flag)
Read block number 'b' into the buffer located at address 'addr'. If no error occurs during read, the flag returned will be false (0).

<wblock> (addr b# - f)
Write the block of data contained in the buffer located at address 'addr' to block number 'b' on the disk. If no error occurs during the write operation the flag returned will be false (0).

?diskerror (n -)
?diskerror will take the error code from the parameter stack, analyze it, and print an error message which tells the user what type of disk error occurred.

@ptr (delta - ptr)
Get a pointer from the system id area.

block (n -)
Tries to read the contents of block number 'n' on the disk into the block buffer in memory. If block 'n' has already been read into the block buffer, **block** will do nothing. If block 'n' is not currently in the block buffer, **block** will read the contents of block 'n' into the buffer and overwrite the current block buffer contents.

copy (n1 n2 n3 -)
Copy blocks number 'n1' through 'n2' to the blocks starting at block number 'n3'.

copy0>0 (n1 n2 n3 -)
Copy blocks number 'n1' through 'n2' from the source disk to the blocks starting at block number 'n3' on the destination disk.

doff (-)

don (-)
Tries to turn the disk drives on.

drive0 (-)

drive1 (-)

ebuf (-)

format (-)
Formats a disk using the IAI disk format.

idblock (- f)
Read one of the two edde (i/o flag) id blocks. The flag returned will be true (nonzero) if an error occurs during the read.

load (b -)

rblock (addr b -)
Reads block # 'b' from disk to the RAM buffer located starting at address 'addr'.

rblocks (n b - n m)
Read 'n' blocks, starting at block number 'b', from disk into memory starting at the current location of the **here** pointer.

recal (-)

rsector (a sector# - errorcode)

rtrk (a track - errorcode)

save? (-)
Aborts if the disk is write-protected.

side0 (-)

side1 (-)

thru (b1 b2 -)
Loads block number 'b1' through block number 'b2' from disk.

vsector (a sector# - errorcode)

wblock (addr b -)
Write the block of data located in RAM starting at address 'addr' to block number 'b' on the disk.

wblocks (n b - n b)
Write 'n' blocks, starting with block 'b', to disk from memory starting at the address of the **here** pointer.

wsector (addr sector# - errorcode)

wtrk (addr track - errorcode)

.<restore> (-)
Restore subroutine.

.<save> (-)
Save subroutine address.

<rdata> (d2 = -1 if invalid crc or data field not found)
(d3 = low word is crc read. high word is crc calculated)
(a6 = address of rbyte)
(a5 = address of disk status register)
(a4 = disk data register address)
(a3 = CRC table address)
(a2 = return address)
(a1 = buffer address)

<rheader> (d2 = returns with the address of info or -1 if not found.)
(a6 = address of rbyte)
(a5 = address of disk status register)
(a4 = disk data register address)
(a3 = CRC table address)
(a2 = return address)

<rsector> (a n - n)

<step> (-)
Step the drive head with interrupts off. Saves and restores the status register.

<trackdump> (-)

<vdata> (d2 = -1 if invalid crc or data field not found)
(d3 = low word is crc read. high word is crc calculated)
(a6 = address of rbyte)
(a5 = address of disk status register)
(a4 = disk data register address)
(a3 = crc table address)
(a2 = return address)
(a1 = buffer address)

<vsector> (a n - n)

<wdata> (a1 = pointer to data)
(a2 = return address)
(a3 = pointer to crc table)
(a4 = pointer to disk data register)
(a6 = pointer to wbyte routine)
Writes a data field onto the disk using the table pointed to by the contents of the A5 register.

<wsector> (a n - n)

```
<wtrack>    ( a1 = pointer to data )
            ( a2 = return address )
            ( a3 = pointer to crctable )
            ( a4 = pointer to disk data register )
            ( a5 = pointer to format information )
            ( a6 = pointer to wbyte )
            ( d2 = starting address information )
            Writes one track of data to the disk.

?diskrdy    ( - f )
            Returns a true (-1) flag if the disk is ready.

?trk0       ( - f )
            Returns a true (-1) flag if on track 0.

?wprot      ( - f )
            Returns a true (-1) flag if write protected.

crc         ( n1 n2 - n3 )

crctable

iai-trk

rbyte       ( a3 = pointer to crctable )
            ( a4 = pointer to disk data register )
            ( d3 = contains the current crc value )
            Writes a byte of data to the disk.

rheader     ( - n )

rtrack      ( a n n - n )

stepin      ( - )
            Set dir signal to step in.

stepout     ( - )

trackdump   ( - )

wbyte       ( a3= pointer to crctable )
            ( a4 = pointer to disk data register )
            ( d0 = the byte to be written with upper bits =0 )
            ( d3 = contains the current crc value )
            Writes a byte of data to the disk.

wsync       ( d0 = number of times to be written )
            ( a4 = pointer to disk data register )
            Write n bytes of zeros to the disk.

wtrack      ( a n - )
            Write track using IAI format to disk.

~rimage     ( - )
~wimage     ( - )
```

CRT DISPLAY WORDS

cls (-)
Clear the display screen.

home (-)
Positions the cursor in the first column of the first row on the screen (in the upper left hand corner).

page (-)
If the screen is the current output device, clears the screen and places the cursor in the upper left corner of the screen.

setcur (x y -)
Position the curson at x,y.

window (n -)
Set FORTH's bottom display line to 'n' where 1<=n<=1D.

voff (-)
Turn the video display off.

von (-)
Turn the video display on and off, decide in high level.

SOUND GENERATOR WORDS

beep (-)
Make a beep.

ringoff (-)
Turns off timer interrupts.

thp (n -)
Set up sound generator frequency.

toff (-)
Turn sound generator off.

ton (-)
Turn sound generator on.

tone (pitch duration -)
Emit sound with the specified pitch for the specified duration.
The duration is specified in ticks.

!char (char -)
Takes a character, as returned by **<?k>** , stores the character code in the system integer **char** , and stores a true (nonzero) value in the system integer **char?** (to indicate that a character is available. If the character is one of the "special" keys on the keyboard (KB1/2 , left shift , right shift , caps lock , left use-front , right use-front , left leap or right leap) **!char** will perform some special tests before storing the character code in **char** . If the special key is going down while one of the use-front keys is already down, and the special key is not the caps lock key, the special key will be marked as "down" in the modifiers array. If the special key is a caps lock key, the state of the modifiers array will not be affected. If the special key is going down while neither use-front key is down, the special key is marked as "down" in the modifiers array and, if the special key is one of the shift keys, the caps lock key is marked as "up" (off). If the special key is going up and it is a caps lock key, the state of the modifiers array is not changed. If the special key is going up and it is not a caps lock key, it is marked as "up" in the modifiers array. The final special key test checks to see if the caps lock key is currently down. If it is, the LED on the caps lock key will be lit. Otherwise, the LED will be unlit.

<?k> (- f)
Uses **<<?k>>** to see if a key is available and returns a true (nonzero) flag if a character is available.

<<?k>> (- flag)
Returns a true (nonzero) flag if a key is available. First, checks to see if a key is currently available. If a key is already available, will exit immediately and return a true (nonzero) flag. If a key is not currently available, will spin in a loop calling **do-event** until either a key is available or until there are no more key events in the event loop.

<key> (- char)
Get a key, set **char?** to zero to indicate that no keys are currently available, and, if the system is in the middle of recording a learn sequence, record the character.

?auto (- f)
Returns a true (nonzero) flag if it is time to autorepeat the current character.

?ctl (- f)
Returns a true (nonzero) flag if one of the USE FRONT keys is currently down.

?ev (- f)
Returns a true (nonzero) flag if the keyboard event queue is not empty, if keyboard events are available.

?k (- f)
Return a true (nonzero) flag if the current character is not a special key.

?kstat (- n)
Returns the keyboard status.

?kval (- c)
Returns the character code stored in **char** . Used to "peek" at the current character without affecting its current character status.

?lex (- f)
Returns a true (nonzero) flag if the left leap key is currently down.

?panic (- f)
Returns a true (nonzero) flag if the user hits the panic stop key.

?rex (- f)
Returns a true (nonzero) flag if the the right leap key is currently down.

?shift (- f)
Returns a true (nonzero) flag if either of the SHIFT keys is down.

?t (- f)
Returns a true (nonzero) flag if a character is currently available. The character is consumed by **?t** .

@k (- c)
Returns the next 'physical' character (the character code as returned by **do-event**).

clear-auto (-)
Turn off autorepeating.

clear-special
(-)
Clears out the shift state array to indicate that all of the special keys are up.

clr-kbd (-)
End playback of a learn sequence.

do-event (-)
Removes a key event from the keyboard event queue and converts the event code into offset. The offset is used to index into a table which converts key press information into character information. Stores the character information into the system integer **kval** and stores a true (nonzero) flag into the system integer **kstat** to indicate that a key is available. If the character is one of the special keys, performs tests and actions similar to those performed by **!char** (except **do-event**'s actions affect the **shiftstate** array instead of the **modifiers** array).

down? (n - f)
Checks to see if the special key corresponding to the number 'n' is currently down. Returns a true (nonzero) flag if the key is down.

keyboardoff (-)
Turn keyboard scan off.

keyboardon (-)
Turn keyboard scan on.

playback? (- f)
Returns a true (nonzero) flag if there is a character to play back.

playback (- c)
Return the next character to be played back.

record (c - c)
Insert the character in the learn string currently being recorded.

set-auto (-)
Turn on autorepeating for the last key returned.

sync-shiftkeys
(-)
Store the actual physical states of the special keys, as stored in the system integer **shiftstate** , into the modifiers system integer.

thislearn (- addr n)
Return the address and length of the current learn string.

MODEM AND SERIAL I/O WORDS (HIGH-LEVEL)

squish (byte1 byte2 byte3 byte4 - longword)
Uses the lowest order byte from each of the four values on the stack to create one **longword** (32-bit) which is returned on the stack. The byte taken from the value on top of the stack will end up in the most-significant byte position of the **longword** and the byte taken from the fourth value on the stack will end up in the least-significant byte position.

talk (-)
Connect phone to line.

thres.43 (-)
Set energy detect threshold to -43 dBm.

thres.48 (-)
Set energy detect threshold to -48 dBm.

tt.disable (-)
Disable touchtone encoder.

tt.enable (-)
Enable touchtone encoder.

txcr.disable
(-)
Disable modem carrier.

txcr.enable (-)
Enable modem carrier.

valid.tone.table
(-)

wordlen (n -)
Sets the number of bits per word.

<dial> (addr len -)
Dial the string pointed to by 'addr' and 'len'.

char>tone (char -)
Send DTMF if valid tone found.

char>pulses (char -)
Send pulses.

dialchar (char -)
Dials an ASCII char.

getover (-)

getport# (- n)

initmodem (-)
Reset the modem.

initphone (-)
Initialize the modem and phone ACIA.

initrs232 (-)
Initialize the serial port.

port>mem (addr len -)
Get a string of stuff into memory.

pulses (n -)
Send 'n' pulses.

send.tone (n -)
'n' is the row/col data. Send a 50 ms DTMF.

MODEM AND SERIAL I/O WORDS (LOW-LEVEL)

<box> (x1 y1 x2 y2 flag -)

<line> (x1 y1 x2 y2 flag -)

<point> (x y flag -)

analog.loop.off
(-)
Disable analog loopback.

analog.loop.on
(-)
Enable analog loopback.

digital.loop.off
(-)
Disable digital loopback.

digital.loop.on
(-)
Enable digital loopback.

filter.high (-)
Set filter for normal operation.

filter.low (-)
Set filter for call progress detection.

init.ph.acia
(-)
Set hpne UART to 1200 baud, 8 data bits, 1 stop bit, and no parity.

modem.ans (-)
Set to answer mode.

modem.fsk (-)
Set to 300 bits per second (bps) FSK.

modem.orig (-)
Set to originate mode.

modem.psk (-)
Set to 300 bits per second (bps) PSK.

mute (-)
Mute phone.

offhook (-)
Place the phone off hook.

onhook (-)
Place the phone on hook.

ph.rx (- byte)
Receive a byte from the telephone's rs232 port.

ph.tx (byte -)
Send a byte to the telephone's rs232 port.

pll.fast (-)
Set PLL to fast response.

pll.slow (-)
Set PLL to slow response.

row/col.table
(-)

scrambler.disable
(-)
Disable modem scrambler.

scrambler.enable
(-)
Enable modem scrambler.

ser.rx (- byte)
Receive a byte from the rs232 port.

ser.tx (byte -)
Send a byte to the rs232 port.

sp1.off (-)
Set pin 13 low.

sp1.on (-)
Set pin 13 high.

sp2.off (-)
Set pin 16 low.

sp2.on (-)
Set pin 16 high.

tFORTH SYSTEM INTEGERS

active Holds the address of the 'active' vocabulary array.

applic Holds the address of the next available location in the header area of the current open vocabulary.

auto

base Holds the number used to indicate the numeric base currently being used for all number I/O.

blk

bound During the interactive execution of program control structures, bound is used to hold the start address of the program control structures code which is to be executed interactively.

cbuff Holds the address of the keyboard input circular buffer.

char

char?

clock0

clock1

crt

csp Used to hold the return stack pointer which is saved away before compilation and checked after compilation.

diskerror# Holds the most recent disk error number.

drive Holds the number used to specify the drive type.

dticks Holds count for the disk countdown timer.

edde I/O flag. If true (nonzero) output should be sent to the editor.

endtable Holds the end address of the RAM token table.

execbuf

extant Holds the address of the vocabulary 'extant' array.

gticks Holds count for a general countdown timer.

gvect Used as a general execution vector.

here Holds the address of the next available location in the code area of the current open vocabulary.

hld During number formatting, holds the current offset into the string being constructed in the pad.

in Pointer used to mark word's progress through the input stream. in always holds the address of the next byte to be examined by word in the input stream.

intexecvecs Interrupt execution vectors.

inuse

itx Holds the address of the current input text.

jdn

kev

kstat

kval

last4thline

lasttok

len Holds the length of the word most recently extracted from the input stream by **word** .

limit Used to hold the end address of the block of text to be examined by the word **interpret**.

locals Used during the compilation of local variables to keep track of the amount of local variable return stack storage space which is required by the definition currently being compiled. Used primarily by the words **doloc** , **local** , and ; .

localvoc Holds the address of the temporary hidden vocabulary used to hold the names of local vocabularies.

location Used during the compilation of local variables to hold the address of the special, invisible vocabulary used to hold the names of the local variables used by the word currently being compiled.

loops System integer used during the compilation of a 'do' loop program control structure to hold the amount of return stack space currently required by the definition being created. Used primarily by the words **do** , **loop** , **+loop** , **doloc** , and ; .

lp I/O flag. If true (nonzero) output should be sent to the line printer.

maxblks

modifiers

nesting System state flag, if true (nonzero) the system is in a temporary compiling state (for interactive execution of program control structures). If false (0) the system is in the interpreting state.

nestype The **nestype** system integer is used to hold a flag which, during the compilation of program control structures, holds a '-1' if the program control structure currently being compiled is a 'do' loop or holds a '0' if the program control structure being compiled is a 'begin' loop. Used by the compiling word **{while}** .

newest Holds the header address of the most recently defined colon definition.

origin Holds the address of the start of the 'tFORTH' dictionary.

pad Holds the address of a location 128 (decimal) bytes from the start of a 384 (decimal) byte scratch location. The pad area is used by the number formatting operators and by the editor [CALC] function.

panicked

ramend Holds the address of the end of RAM memory.

ramstart Holds the address of the start of RAM memory.

ringsoundaddr
Holds the address of a general ring sound routine.

savenest

savestate

scontcopy

screen Holds the address of the start of display memory.

screensize

ser I/O flag. If true (nonzero) output should be sent to the serial port.

soundaddr Holds the address of a general sound routine .

soundcount Holds general sound count.

sp0 Holds the address of the base of the parameter stack.

special Holds the address of the keyboard 'special' array.

state System state flag, if true (nonzero) system is in the compiling state. If false (0) the system in the interpreting state.

str Each time **word** gets the next word fro[illegible] input string, it places the address of the character string in the **str** system integer. See the description for the system integer **len** also.

strings

targeting Flag. If true (nonzero) target compilation is occurring.

ticks Holds time ticks.

tokens One of two system integers used to help in the assignment of tokens to new words (**lasttok** is the other system integer used for this purpose). See the techical discussion on local variables.

top Holds the address which is one byte beyond the top of 'tFORTH' memory.

vdelay Holds the value used to specify how long the video screen should stay 'on' on an unused terminal.

vticks Holds count for the video countdown timer.

x Holds 'tFORTH's column output position.

y Holds 'tFORTH's row output position.

! (n a -)
('store')
Stores the 32-bit value n into memory starting at address a.

!char (char -)
Takes a character returned by **<?k>** , stores the character code in the system integer **char** , and stores a true (nonzero) value in the system integer **char?** to indicate that a character is available.

!csp (-)
('store-csp')
Saves return stack pointer during compilation.

!ptr (n delta -)
Store the value n into the save block area.

" Compile time: (-)
('quote')
Run-time: (- addr len)
Format: " ccc"
Compiling, lays the string between quotes, and the runtime code <">, into the definition being compiled. At runtime the address and length will be left on the parameter stack.

"to (addr1 n1 addr2 n2 -) ('quote-to')
Format: " ccc" <string name> "to
Stores the string specified by the address and length (addr1 and n1) into the string integer specified by name. addr2 and n2 are discarded.

(n1 - n2)
('sharp')
Format: n <# ... # ... #>
Extracts the lowest order digit from the number on top of the stack and inserts it into the formatted numeric string being constructed in the pad.

#> (n1 - a n2)
('sharp-greater')
Format: n <# ... #>
Prepares a formatted numeric string for type.

#s (n - 0)
('sharp-s')
Format: n <# ... #s ... #>
Calls # until the number on top of the stack has been reduced to zero.

' (- token)
('tick')
Format: ' <name>
Returns the token for **<name>**:

((-)
('paren')
Format: (ccc)
All characters between the starting left paren and the closing right paren are considered to be comments and are ignored by the FORTH compiler.

***** (n1 n2 - n3)
('times')
Multiplies n1*n2 and leaves the 32-bit result n3.

+ (n1 n2 - n3)
('plus')
Adds n1 plus n2 and leaves the 32-bit result n3.

+! (n a -)
('plus-store')
Adds the 32-bit value n to the 32-bit value located in memory starting at address a. Memory at a is modified.

+bit7 (char - char')
('plus-bit-7')
Sets the seventh bit in the character byte.

+loop Compiling: (-)
('plus-loop')
Executing: (n -)
Format: do ... n +loop
Program control structure used to implement definite loops. During execution, **+loop** adds 'n' to the current loop index.

+table (n - a)
('plus-table')
Converts a token table entry number to the token table entry fields address.

+to (n1 n2 -)
('plus-to')
Format: n1 **<integer** or local variable **name>** **+to**
Adds n1 to the current value of the integer or local variable specified by name.

, (n -)
('comma')
Lays the 32-bit value 'n' into the next free location in the code area. The here pointer always points at the next free location in the code area. The here pointer is incremented by 4 bytes.

- (n1 n2 - n3)
('minus')
Subtracts n2 from n1 and leaves the 32-bit result on the stack.

-1 (- -1)
('minus-one')
Puts the constant value '-1' on top of the parameter stack.

-trailing (addr len - addr len')
('minus-trailing')
Strips the trailing spaces from the string located at address.

-userounded

. (n -)
('dot')
Prints the signed value on top of the stack followed by a trailing space. Prints in the current radix.

." (-)
('dot-quote')
Format: ." ccc"
May be used interactively or compiled into a definition. The compile-time action of ." is to lay the string between quotes into the definition being compiled. The run-time action of ." is to type the string between quotes out to the current output device.

.r (n w -)
('dot-r')
Prints the signed value 'n' in a field which is 'w' spaces wide.

.s (-)
('dot-s')
A nondestructive display of the items on the parameter stack.

/ (n1 n2 - n3)
('divide')
Divides n1 by n2 and leaves the 32-bit quotient on the stack.

/mod (n1 n2 - n3 n4)
('divide-mod')
Divides n1 by n2 and leaves the 32-bit remainder, n3, and the 32-bit quotient, n4, on the stack.

0 (- 0)
('zero')
Puts the constant '0' on top of the parameter stack.

0< (n - f)
('zero-less-than')
Returns a true (-1) flag if n is less than zero.

0= (n - f)
('zero-equal')
Returns a true (-1) flag if n is equal to zero.

1 (- 1)
Puts the constant '1' on top of the parameter stack.

1+ (n1 - n2)
('one-plus')
Adds one to the number on top of the stack.

1- (n1 - n2)
('one-minus')
Subtracts one from the number on top of the stack.

2* (n1 - n2)
('two-times')
Multiplies the number on top of the stack by two.

2+ (n1 - n2)
('two-plus')
Adds two to the number on top of the stack.

2- (n1 - n2)
('two-minus')
Subtracts two from the number on top of the stack.

2/ (n1 - n2)
('two-divide')
Divides the number on top of the stack by two.

2drop (n1 n2 -)
('two drop')
Discards the top two items on the parameter stack.

2dup (n1 n2 - n1 n2 n1 n2)
('two dup')
Duplicates the top two items on the parameter stack.

3dup (n1 n2 n3 - n1 n2 n3 n1 n2 n3)
('three-dup')
Duplicates the top three items on the parameter stack.

: (-)
('colon')
Format: : <name> ...words... ;
Defining word used to create new definitions. Puts the system in the compiling state, creates a new dictionary header using <name>, sets the smudge bit in the dictionary header so the definition will not be visible until completed.

; (-)
('semi-colon')
Used to terminate colon definitions. If the colon definition does not use local variables, : causes the word <;> to be compiled into a definition. If the colon definition does use local variables, ; causes the word **<;lp>** to be compiled into a definition.

< (n1 n2 - f)
('less-than')
Returns a true (-1) flag is n1 is less than n2.

<"> (- addr len)
('brac-quote')
<"> is the run-time code for the word " . Pushes the address and length of the string on the stack.

<"to> (addr1 n1 addr2 n2 -> addr3 n3)
('brac-quote-to')
Run-time code for "to .

<# (n - n)
('less-sharp')
Format: n <# ... #>
Marks the start of a pictured numeric conversion process.

<+loop> (n -)
('brac-plus-loop')
Run-time code for **+loop** .

<0bran> (f -)
('brac-zero-bran')
Run-time conditional branching primitive. A branch will occur if the flag passed to **<0bran>** is false (zero). Can only handle short (-81<n<80 hex) branching distances. Used by **while**, **until**, and **if**.

<0branl> (f -)
('brac-zero-bran-long')
Run-time conditional branching primitive. A branch will occur if the flag passed to **<0branl>** is false (0). Can be used for short and word (-8001<n<8000 hex) branching distances. Used by **while** , **until** , and **if** .

<0leave> (f -)
('brac-zero-leave')
Run-time code used to conditionally leave from a 'do...loop' or 'do...+loop' program control structure.

<0leavel> (f -)
('brac-zero-leave-long')
Run-time code used to conditionally leave from a 'do...loop' or 'do...+loop' program control structure.

<;>
Parameter: (-)
('brac-semi')
Return: (n1 n2 -)
Run-time word compiled by ; .

<;lp>
(-)
('brac-semi-local')
Run-time exit word compiled at the end of colon definitions in which local variables are used. Compiled by ; .

<<?k>>
(- flag)
('brac-brac-question-k')
Returns a true (nonzero) flag if a key is available.

<>
(n1 n2 - f)
('not-equal')
Returns a true (-1) flag if n1 is not equal to n2.

<?k>
(- f)
('brac-question-k')
Uses **<<?k>>** to see if a key is available and returns a true (nonzero) flag if a character is available.

<abort">
(f a n -)
('brac-abort-quote')
Run-time code used by **abort"** .

<addto>
(n -)
('brac-addto')
Close the current open vocabulary and open the vocabulary specified by the token 'n'.

<avg>

<becode>
(n -)
('brac-becode')
Remove the code corresponding to the token 'n'.

<behead>
(a -)
('brac-behead')
Remove the header located at address 'a'.

<bevoc>
(n -)
('brac-bevoc')
Completely eliminate the vocabulary specified by the token 'n'.

<bran>
(-)
('brac-bran')
Run-time unconditional branching primitive. Always causes a branch to occur. Can only handle short (-81<n<80 hex) branching distances. Used by **again** and **else** .

<branl> (-)
('brac-bran-long')
Run-time unconditional branching primitive. Always causes a branch to occur. Can handle short <-81<n<80 hex) and word (-8001<n<8000) length branching distances. Used by **leave** and **else** .

<csize> (a - n)
('brac-code-size')
Returns the code size 'n', in bytes, of the word whose code is located at address 'a'.

<deactivate>
('brac-deactivate')
(n -)
Removes the vocabulary specified by the token 'n' from the current search order (removes its token from the 'active' array, see **active**).

<demit> (char x y -)
('brac-display-emit')
Draw the character at position x,y on the screen.

<do> (n1 n2 -)
('brac-do')
Run-time code for **do** .

<empty> (n -)
('brac-empty')
Purges all words from the vocabulary specified by the token 'n'.

<eta> (a n - 0 | If token 'n' is not found.)
(a1 n - a2 | If token 'n' is found.)
Takes the vocabulary address 'a1' and the encoded token value 'n' and, if successful, returns the encoded token address.

<exit> ('brac-exit')

<exitlp> ('brac-exit-lp')

<find> (a1 a2 n1 - a2 f | If not found)
(a1 a2 n1 - a3 n2 t | If found)
('brac-find')
Searches for the name specified by the string at address 'a2' of length 'n1' in the vocabulary which starts at address 'a1'. If the word is found in the vocabulary, **<find>** will return the dictionary header address 'a3' for the word, the token for the word 'n2' and a true flag (nonzero). If the word is not found in the vocabulary **<find>** will return the original name string address 'a2' and a false (zero) flag.

<key> (- char)
('brac-key')
Get a key, set **char?** to zero to indicate that no keys are currently available, and, if the system is in the middle of recording a learn sequence, record the character.

<leave> (-)
('brac-leave')
Run-time code used to unconditionally leave from a 'do...loop' or 'do...+loop' program control structure.

<leavel>

<load> (n -)
('brac-load')
Reads block 'n' from disk into memory and interprets its contents.

<loc0> (- n)
('brac-loc-zero')
A special fast word used to access the first local variable on the return stack.

<loc1> (- n)
('brac-loc-one')
A special fast word used to access the second local variable on the return stack.

<local> (- n)
('brac-local')
Generic word used to access the third, and all subsequent local variables on the return stack.

<locals> (-)
('brac-locals')
First local variable word compiled into a tForth word which uses local variables.

<loop> (-)
('brac-loop')
Run-time code for **loop**.

<purge> (n -)
('brac-purge')
Removes the word corresponding to the token 'n' from the dictionary.

<quit> (-)
('brac-quit')
Low-level word used by **quit**.

<rblock> (addr b# - flag)
('brac-r-block')
Read block number 'b' into the buffer located at address 'addr'. If no error occurs during read, the flag returned will be false (0).

<step> (-)
('brac-step')
Step the drive head with interrupts off. Saves and restores the status register.

<string> (- addr len)
('brac-string')
Run-time code for string integers created with the defining word **string**. Pushes the address and length of the string stored in the string integer on the stack.

<sum>

<sumrounded>

<wblock> (a n - f)
('brac-write-block')
Write the block of data contained in the buffer located at address 'a' to block number 'n' on the disk.
If no error occurs during the write operation the flag returned will be false (0).

<word> (addr1 addr2 - addr3 n addr4)
('brac-word')
Lower-level routine used by **word**.

> (n1 n2 - f)
('greater-than')
Returns a true (-1) flag if n1 is greater than n2.

= (n1 n2 - f)
('equal')
Returns a true (-1) flag if n1 is equal to n2.

>r (n1 - | return stack: - n1)
('to-r')
Removes n1 from the parameter stack and places it on the return stack.

?auto (- f)
('question-auto')
Returns a true (nonzero) flag if it is time to autorepeat the current character.

?csp (-)
('question-csp')
Compares the current return stack pointer to the return stack value saved away previously in the **csp** system integer. If the two addresses are not equal the system will abort with an "unpaired" message.

?ctl (- f)
('question-control')
Returns a true (nonzero) flag if one of the use-front keys is currently down.

?diskerror (n -)
('question-disk-error')
?diskerror will take the error code from the parameter stack, analyze it, and print an error message which tells the user what type of disk error occurred.

?dup (n - n n) or (0 - 0)
('question-dupe')
Duplicates the value on top of the stack if it is nonzero.

?ev (- f)
('question-event')
Returns a true (nonzero) flag if the keyboard event queue is not empty, if keyboard events are available.

?k (- f)
('question-key')
Return a true (nonzero) flag if the current character is not a special key.

?keystep

?kval (- c)
('key-value')
Returns the character code stored in **char** . Used to "peek" at the current character without affecting its current character status.

?lex (- f)
('question-left-leap')
Returns a true (nonzero) flag if the left leap key is currently down.

?pairs (-)
('question-pairs')
Checks for properly paired conditional statements. Aborts with an error message if conditionals are improperly paired.

?panic (- f)
('question-panic')
Returns a true flag if the user hit the panic stop key.

?rex (- f)
('question-right-leap')
Returns a true flag if the the right leap key is down.

?shift (- f)
('question-shift')
Returns a true flag if either of the shift keys is down.

?stack (- f)
('question-stack')
Checks the status of the parameter stack. A false (0) flag will be returned if the stack is ok.

?stackerr (-)
('question-stack-error)
Uses **?stack** to check for stack underflow or overflow.

?t (- f)
('question-terminal')
Returns a true (nonzero) flag if a character is currently available. The character is consumed by **?t** .

@ (a - n)
('fetch')
Places a copy of the 32-bit value located in memory starting at address a on top of the parameter stack.

@k (- c)
('fetch-key')
Returns the next 'physical' character (the character code as returned by **do-event**).

@ptr (n - a)
('fetch-pointer')
Get a pointer from the system id area.

aabs

abort

abort" (f -)
('abort-quote')
Format: f abort" ccc"
If the flag passed to **abort"** is true (nonzero), a forced system abort process will occur.

abs (n - |n|)
('absolute')
Returns the absolute value of the number on top of the stack.

addr (<name> - a)
('adder')
Format: <name of integer> addr
Returns the address of the storage location for the integer specified by name.

addto (-)
Format: addto <vocab-name>
Opens the vocabulary whose name immediately follows **addto** .

afilter

again (-)
Format: begin ... again
Used to implement endless loops.

aint

align (-)
Aligns the **here** pointer to an even address boundary.

allot (n -)
Tries to allocate 'n' bytes in the code area of the currently open vocabulary.

and (n1 n2 - n3)
Performs a bit-by-bit logical AND using n1 and n2. Returns the 32-bit result (n3) on the parameter stack.

and! (b a -)
('and-store')
Performs a bit-by-bit logical AND operation using b and the byte located in memory starting at address a. The byte at address a is modified.

arithmetic (-)
Name of the vocabulary in which the basic arithmetic functions are located.

array Compiling: (n -)
Executing: (- a)
Format: n array <arrayname>
Create an array of length n and name <arrayname>. Later use of <arrayname> will return the address of the start of the array.

ascii (- n)
Format: ascii <char>
Returns the ascii value of **<char>**.

assign (a1 a2 n -)
Assigns a token to and builds a header for a new definition in the vocabulary specified by the address 'a1' using the name located at the address 'a2' with the length 'n'.

asqrt

backelse (n1 - n2 n1)
Used by **then** to backpatch a forward **else** branch offset.

becomes (-)

beep (-)
Make a beep.

begin (-)
Format: begin ... again

begin ... until
Used to mark the start of an endless or indefinite program loop.

behead (-)
Format: behead <name>
Remove the header of the definition whose name immediately follows **behead** .

bevoc (-)
Format: bevoc <name of vocabulary>
Removes the vocabulary name and all its words.

blit (- n)
('byte-literal')
Byte length version of **lit**.

block (n -)
Tries to read block 'n' into memory. If block 'n' has already been read into the block buffer, **block** will do nothing.

c! (b a -)
('c-store')
The least significant 8 bits of the 32-bit value, b, on the parameter stack are stored into memory starting at address a.

c' (- a)
('c-tick')
Format: c' <name>
Leaves the address of the code field of **<name>** .

c, (c -)
('c-comma')
Compiles the byte value 'c' into the next available location in the code area.

c@ (a - b)
('c-fetch')
Places the 8-bit value from address a on the stack.

call

check (a n -)
Format: <string integer name> check
Prints the ascii values for each character in the string currently stored in the string integer specified by name. If the string integer is empty, an error message is displayed.

clear-auto (-)
Turn off autorepeating.

clr-kbd (-)
('clear-keyboard')
End playback of a learn sequence.

cls (-)
('clear-screen')
Clear the display screen.

cmove (a1 a2 u -)
('c-move')
Moves the 'u' bytes located starting at the source address 'a1' to the destination address 'a2'.

compile

compile, (n -)
('compile-comma')
Lays the token value passed on the stack into the dictionary at the current **here** address.

copy (n1 n2 n3 -)
Copy blocks number 'n1' through 'n2' to the blocks starting at block number 'n3'.

copy0>0 (n1 n2 n3 -)
Copy blocks number 'n1' through 'n2' from the source disk to the blocks starting at block number 'n3' on the destination disk.

cr (-)
('c-r')
Emit a carriage return/linefeed.

create (-)
Format: create <name>
Assigns a token to and creates a header entry for <name> in the current open vocabulary.

createvoc (a1 n - a2)
Create an empty vocabulary using the image of an empty vocabulary located starting at address 'a1' and assign it the token 'n'. Returned address 'a2' is unused.

crlfscroll (-)
Emit a carriage return and linefeed. Blank the new line out and scroll if necessary.

csize (- n)
('code-size')
Format: csize <name>
Returns the code size of the word specified by **<name>** .

ctl (-)
('control')
Format: ctl <char>
Make **<char>** a control character.

deactivate (-)
Format: deactivate <vocab-name)
Removes the vocabulary whose name immediately follows **deactivate** from the current search order.

decimal (-)
Set the base to ten.

decode (n1 - n2)
Turns an encoded token number into a decoded token number.

demit (c -)
('display-emit')
Emit the character to the screen. If the character is a cr perform a carriage return/linefeed and scroll if necessary. If the character is the 'del' (delete) character erase the previous character on this line (if any).

depth (- n)
Returns the number of items on the stack.

diff? (a1 a2 n -> 0 1 If strings match)
(a1 a2 n -> a3 -1 1 If strings don't match)
Compares the first 'n' characters in the strings located at addresses 'a1' and 'a2'.

digit (n1 n2 - n3 c)
Extracts the least significant digit from the number, n1, on the stack (using the specified base, n2) and leaves ascii value for the digit, c, and the remaining number, n3 on the stack.

do Compiling: (-)
Executing: (n1 n2 -)
Format: n1 n2 do ... loop
n1 n2 do ... n3 +loop
Marks the start of a definite program loop.

do-event (-)
Removes a key event from the keyboard event queue and converts the event code into offset. The offset is used to index into a table which converts key press information into character information.

doff (-)
Tries to turn the disk drive(s) off.

doloc (- f)
('do-local')
Used by **interpret** . Checks to see if the word just extracted from the input stream belongs to a local variable.

don (-)
Tries to turn the disk drive(s) on.

down? (n - f)
Checks to see if the special key corresponding to the number 'n' is currently down. Returns a true (nonzero) flag if the key is down.

drop (n1 -)
Discards the top item from the stack.

dump (a n -)
Displays the contents of 'n' bytes of memory starting at address 'a'.

dup (n1 - n1 n1)
Duplicates the value on top of the stack.

ebuf

eemit (c -)
('editor-emit')
Emit the character to the editor.

else Compiling: (-)
Executing: (f -)
Format: if ... else ... then
Inner decision point in the 'if...else...then' conditional program control structure.

emit (c -)
Output the character to all active output devices.

empty (-)
Purges all words from the current vocabulary. The words in the **forth** vocabulary cannot be purged.

emptyvoc (- addr)
Returns the address of the 18 decimal byte image of an empty vocabulary.

encode (n1 - n2)
Takes the decoded token number from the top of the stack, encodes it, and returns the encoded token number on top of the stack.

error

eta (n - a f)
Tries to return the address of the token table entry for the token. If successful returns the token table entry address and a true (nonzero) flag. Otherwise, returns a false (0) flag.

exa

execute (n -)
Executes the word corresponding to the token 'n' passed on the stack.

existing (-)
Displays the names of and parents of all vocabularies.

exit (-)
Terminates execution of the current definition and transfers control to the definition which contains the current definition.

fill (a u b -)
Format: 'start address' 'count' 'fill character' fill
Replaces the u bytes located in memory starting at address a with the byte value b.

find (a n1 - n2 true 1 If found in search order)
(a n1 -> false 1 If not found in search order)
Searches the through the dictionary (uses the current search order) looking for the definition whose name matches the name at the address 'a' with length 'n1'.

finish-lex

fnderr (-)
('find-error')
Prints a "can't find" error and aborts.

format (-)
Formats a disk using the IAI disk format.

forth (-)
This is the main tFORTH vocabulary. It contains all of the 'standard' FORTH words supported by tFORTH and all of the tFORTH FORTH extension words. Execution of **forth** will cause the **forth** vocabulary to become the first vocabulary in the search order.

forward

freetoken (-)
Prints an "unassigned token" message and aborts.

froom?

function

get(

getphrase

goto

hex (-)
Selects base sixteen (hexadecimal) as the current radix.

hidden (-)
Vocabulary which contains all of the editor words.

hold (c -)
Format: <# ... ascii c hold ... #>
Inserts the character (ascii value) on top of the stack into the formatted numeric string currently being constructed in the pad.

home (-)
Positions the cursor in the first column of the first row on the screen (in the upper left hand corner).

i (- n)
Puts a copy of the top item on the return stack on top of the parameter stack. During execution of a do...loop, the top item on the return stack is the index for the current loop.

idblock (- f)
Read one of the two edde id blocks. The flag returned will be true (nonzero) if an error occurs during the read.

if Compiling: (-)
Executing: (f -)
Format: if ... then
if ... else ... then
Marks the start of the 'if...then' or 'if...else...then' conditional program control structures.

immediate (-)
Sets the 'immediate' bit (bit 6) of the most recently defined colon definition so that whenever the word is encountered during compilation, it will be compiled rather than executed.

inrange (n1 n2 n3 - f)
Returns a true (-1) flag if the value n1 is greater than or equal to the lower limit n2 and less than or equal to the upper limit n3 (i.e. n2 < n1 < n3).

int0 (- n)
('int-0')
Runtime code for integers located in integer tier 0.

int1 (- n)
('int-1')
Runtime code for integers located in integer tier 1. See **int0**.

int2 (- n)
('int-2')
Runtime code for integers located in integer tier 2. See **int0**.

int3 (- n)
('int-3')
Runtime code for integers located in integer tier 3. See **int0**.

int4 (- n)
('int-4')
Runtime code for integers located in integer tier 4. See **int0**.

int5 (- n)
('int-5')
Runtime code for integers located in integer tier 5. See **int0**.

int6 (- n)
('int-6')
Runtime code for integers located in integer tier 6. See **int0**.

int7 (- n)
('int-7')
Runtime code for integers located in integer tier 7. See **int0**.

int8 (- n)
('int-8')
Runtime code for integers located in integer tier 7. See **int0**.

integer Compiling: (n -)
Executing: (- n)
Format: n integer <integername>
At compile-time integer creates a named 4-byte data location and initializes the location with the value 'n'. The run-time action of the child words created by integer is to push the current contents of their 4-byte storage location on the stack.

interpret (a l -)
interpret parses words in the input stream and either executes them (if they are executable Forth words), places them on the stack (if they or numbers), or aborts if it does not know what to do with the word.

interpretphrase

invoc (a - n)
Returns the token 'n' of the vocabulary which contains address 'a'.

ioff (-)
Turns interrupts off.

ion (-)
Turns interrupts on.

key (- c)
Waits until a printable character (8<ascii code<7F) is typed at the keyboard. Returns the ascii value of the character on the stack.

learnstrings

leave (-)
Immediately and unconditionally reroutes program execution out of the current "looping" program control structure. May be used in 'begin' loops or in 'do' loops.

lit (- n)
Code definition which transfers the long-word (32-bit) literal value pointed to by the instruction pointer to the parameter stack. The instruction pointer, ip, is incremented by 4 bytes. Used by **literal** .

literal (n -)
literal is used to compile constant data into a definition. literal will also compile the token of a word which will push the constant data onto the parameter stack when the definition is later executed.

load (n -)
Loads block 'n' from the disk.

local (-)
Format: local <name for local variable>
Creates a named local variable. The local variable is not initialized to any value. Executing the name of the local variable will place the value of the local variable on top of the parameter stack.

loop (-)
Format: do ... loop
Marks the end of the 'do...loop' definite loop program control structure.

max (n1 n2 - n3)
Compares n1 and n2 and returns the greater value.

min (n1 n2 - n3)
Compares n1 and n2 and returns the lesser value.

mod (n1 n2 - n3)
n1 is divided by n2 and the 32-bit remainder, n3, is left on top of the stack.

move (a1 a2 u -)
Special version of **cmove** .

ms (n -)
Wait 'n' milliseconds.

n' (- a)
('n-tick')
Format: n' <name>
Returns the address of the dictionary header area for the word specified by **<name>**.

name (n -)
Print the name of the definition which corresponds to the token 'n'.

needforth

needtext

negate (n - -n)
Returns the two's complement of n, i.e. 'n' is subtracted from zero (0-n).

nest (-)
Used by all words which start program control structures.

nip

noop

noroom

not (n1 - n2)
Takes the ones complement of the 32-bit value on top of the parameter stack. Returns the 32-bit result on top of the parameter stack.

not! (a -)
('not-store')
Takes the one's complement of the 8 bits of data located in memory starting at address a. The byte length result is stored into memory at address a.

number (a n1 n2 - f | If conversion is not successful.)
(a n1 n2 - n3 f | If conversion is successful.)
Converts the string of length n1 located starting at address a to a number, n3, using base n2.

numerical

oddadjust

off (n -)
Format: <name of local variable or integer> off
Sets the value of the local variable or integer specified by name to zero. The value of the integer or local variable placed on the parameter stack when the local variable or integer name was executed is discarded.

on (n -)
Format: <name of local variable or integer> on
Sets the value of the local variable or integer specified by name to negative one.

open?

or (n1 n2 - n3)
Performs a bit-by-bit logical or using n1 and n2.
Returns the 32-bit result (n3) on the parameter stack.

or! (b a -)
('or-store')
Performs a bit-by-bit logical OR operation using b and the byte located in memory starting at address a. The byte length result is stored into memory at address a.

outofroom

over (n1 n2 - n1 n2 n1)
Places a copy of the second item on the stack on top of the stack.

packforth

page (-)
If the screen is the current output device, clears the screen and places the cursor in the upper left corner of the screen.

pemit (char -)
('parallel-emit')
Send the character out through the parallel port.

playback (- c)
Return the next character to be played back.

playback? (- f)
Returns a true (nonzero) flag if there is a character to play back.

purge (-)
Format: purge <name>
Removes the word specifed by **<name>** from the dictionary.

quit (-)
quit is the word which runs FORTH. Clears the return stack and puts the system in the interpreting state. After **quit** is executed the system will be waiting for user input for user input to interpret and execute.

r> (n - | return stack: - n)
('r-from')
Transfers the top item on the parameter stack to the top of the return stack.

r@ (- n)
('r-fetch')
Puts a copy of the top item on the return stack on top of the parameter stack. **r@** performs the same function as **i** but **r@** is normally used outside of do...loops.

raddr (- a)
('return-address')
Copies the return information stored on the return stack. Uses the return information to calculate the address where the next token to be executed in the definition at the next higher execution level is located (calculates the previous location of the ip pointer). Used by **compile** .

ramchecksum

recal (- n)
Recalibrate the disk drive to track 0.

record (c - c)
Insert the character in the learn string currently being recorded.

recycle (n -)
Reclaims the token table space for the token 'n'.

recycledtoken
(- token)
recycledtoken checks to see if any previously assigned tokens are now available for re-assignment.

restore

retop (a -)
Lower level word used to open a vocabulary. Moves the upper half of the dictionary up so that the new top of dictionary is at address 'a' .

ringoff (-)
Turns off timer interrupts.

romchecksum

rot (n1 n2 n3 - n2 n3 n1)
('rote')
Rotates the third item on the stack to the top of the stack.

rp!

rub (-)
Erase the previous character on the current line (if any).

safety (a -)
Reclaim the token table space for the token whose header is located at address 'a'.

same? (a1 a2 n -> f)
Returns a true (nonzero) flag if the first 'n' characters in the strings located at 'a1' and 'a2' are the same.

save? (-)
('save-question-mark')
Aborts if the disk is write-protected.

scanfor (c -)
Looks for the next word in the current input stream which is surrounded by the delimiter character, c. Sets the **in** , **str** , and **len** system variables.

searched (-)
Display the vocabulary search order.

semit (c -)
('serial-emit')
Emit the character to the serial port.

set-auto (-)
Turn on autorepeating for the last key returned.

setcodesize (-)
Set the code size field for the current open vocabulary.
Set the odd size flag if necessary.

setcur (x y -)
Position the cursor at x,y.

shl (n1 n2 - n3)
('shift-left')
Shifts the bits in 'n1' 'n2' bits to the left. Leaves the 32-bit result,'n3', on the parameter stack.

shr (n1 n2 - n3)
('shift-right')
Shifts the bits in 'n1' 'n2' bits to the right. Leaves the 32-bit result, 'n3', on top of the parameter stack.

side0 (-)
Select side 0.

side1 (-)
Select side 1.

sign (n -)
If the number on top of the stack is negative, **sign** will insert a minus sign into the formatted numeric string being constructed in the **pad** .

sp!

sp@

space (-)
Emit a space to the current active output devices.

spaces (n -)
Emit 'n' spaces to the current active output devices.

stepin (-)
Set drive to step in.

stepout (-)
Set drive to step out.

string Compiling: (a n -)
Executing: (- a n)
Format: " ccc" string <stringname>
At compile-time **string** creates a named, multi-byte string storage area in the dictionary and initializes the storage area with the characters between the quotes. The run-time action of the child words created by **string** is to push the address and length of the string currently stored in the string storage area on the stack.

stub (-)
Format: stub <name>
Uses **create** to assign a token to and create a dictionary header for <name>. Stores a 0 in <name>'s token table entry so <name> will not have any corresponding code area.

sw

swab (n1 - n2)
Exchanges the lower two bytes of the top value on the stack.

swap (n1 n2 - n2 n1)
Exchanges the top two items on the parameter stack.

sync-shiftkeys
(-)
Store the actual physical states of the special keys, as stored in the system integer **shiftstate** , into the modifiers system integer.

temp

then (-)
Format: if ... then
if ... else ... then
Marks the end of the 'if...then' or 'if...else...then' conditional program control structures.

thislearn (- a n)
Return the address and length of the current learn string.

thp (n -)
Set up sound generator frequency.

thru (n1 n2 -)
Loads block number 'n1' through block number 'n2' from disk.

tier (-)
The first byte of a multi-byte token is called a **tier**. The tiertokens have to be in order. Every 256 definitions in the dictionary takes up a new tier in the token table. Those tiers are named tier1 tier2 tier3 up to tier9 (10 times 256 words in the dictionary). **tier** is never executed alone.

tip (a -)
Toggles memory or I/O port location.

to (n1 n2 -)
Format: n1 **<name** of local variable or **integer>** to
Replaces the current value of the integer or local variable specified by name with the 32-bit value n1. The value placed on the stack when the local variable or integer name was executed is discarded.

toff (-)
Turn sound generator off.

ton (-)
Turn sound generator on.

tone (n1 n2 -)
Emit sound with the pitch 'n1' for the duration 'n2'.
The duration is specified in ticks.

type (a n -)
Types the 'n' characters located in memory starting at address 'a' out to the current output device.

u. (n -)
Prints the unsigned value on top of the stack followed by a trailing space.

u.r (n w -)
Prints the unsigned value 'n' in a field which is 'w' spaces wide.

u< (u1 u2 - f)
('u-less-than')
Returns a true (-1) flag if the unsigned value u1 is less than the unsigned value u2

unnest (-)
Used by all words which end program control structures.

unpackforth

until (f -)
Format: begin ... f until
Conditional exit/branching word used at the end of the 'begin...until' indefinite loop program control structure.

user (-)
Vocabulary to which all user defined words are added.

vocab (-)
Move the current execution vocabulary to the top of the search order by placing its token at the start of the active array.

vocab? (n - f)
Returns a true (nonzero) flag if the token on top of the stack it the token for a vocabulary. Returns a false (0) flag otherwise.

vocabulary (-)
Format: vocabulary <vocabname>
Create a new, but inactive, vocabulary.

voff (-)
('video-off')
Turn the video display off.

von (-)
('video-on')
Turn the video display on and off.

vopen (n - a)
Returns the address of the opening point for the vocabulary which corresponds to the token.

w! (w a -)
('word-store')
The least significant 16 bits of the 32-bit value, b, on the parameter stack are stored into to memory starting at address a.

w, (w -)
('w-comma')
Stores the word length value 'w' into the next available spot in the code area of the currently open vocabulary.

w@ (a - w)
('word-fetch')
Places the 16-bit value located in memory starting at address a in the least significant word of a 32-bit value on top of the parameter stack. The upper 2 bytes (16 bits) are set to zero.

wblock (addr b -)
('write-block')
Write the block of data located in RAM starting at address 'addr' to block number 'b' on the disk.

wblocks (n1 n2 - n3 n4)
('write-blocks')
Write 'n1' blocks, starting with block 'n2', to disk from memory starting at the address of the **here** pointer.

while

Compiling: (-)
Executing: (f -)
Format: begin ... while (... while) ... again
begin ... while (... while) ... until
do ... while (... while ...) ... loop
dO ... while (... while ...) ... +loop
Inner decision/branching point in the 'begin...until' , 'begin...again', 'do...loop', or 'do...+loop' program control structures.

window

(n -)
Set FORTH's bottom display line to 'n' where 1<=n<=1D.

wlit

(- n)
('w-lit')
Code definition which transfers the word-length (16-bit) literal value pointed to by the instruction pointer to the parameter stack and increments the instruction pointer by 2 bytes. Used by **literal**.

word

(-)
Looks for the next word in the current input stream which is surrounded by at least one space. Sets the **in**, **str**, and **len** system variables accordingly.

words

(-)
Displays a list of all words in the vocabulary which is first in the search order.

wtrack

(a n -)
('write-track')
Write track using IAI format to disk.

xor

(n1 n2 - n3)
Performs a bit-by-bit logical **xor** using n1 and n2. Returns the 32-bit result on the parameter stack.

xor!

(b a -)
('exclusive-or-store')
Performs a bit-by-bit logical XOR operation using b and the byte located in memory starting at address a. The byte length result is stored into memory at address a.

[

(-)
('left-bracket')
Turns the FORTH compiler on.

[']

(- token)
('brac-tick-brac')
Format: : <name> ... ['] <definition-name> ;
['] must be used within a colon definition. **[']** will return the token for the definition whose name immediately follows it in the colon definition.

[compile] (-)
('brac-compile-brac')
Compiles the token of the word which immediately follows it into the definition currently being constructed.

] (-)
('right-bracket')
Turns the FORTH compiler off.

{elsethen} ('curly-else-then')
Lower-level compiling word used by **else** and **then**.

{loop} (n1 n2 -)
('curly-loop')
Shared routine used by the loop termination words **loop** , **+loop**, **until** and **again**.

{while} (-)
('curly-while')
Shared routine used by the words used to exit from loop program control structures: **while** and **leave**.

tFORTH SYSTEM INTEGERS
(Alphabetical Listing)

active Holds the address of the 'active' vocabulary array.

applic Holds the address of the next available location in the header area of the current open vocabulary.

auto

base Holds the number used to indicate the numeric base currently being used for all number I/O.

blk

bound During the interactive execution of program control structures, **bound** is used to hold the start address of the program control structures code which is to be executed interactively.

cbuff Holds the address of the keyboard input circular buffer.

char

char?

clock0

clock1

crt

csp Used to hold the return stack pointer which is saved away before compilation and checked after compilation.

diskerror# Holds the most recent disk error number.

drive Holds the number used to specify the drive type.

dticks Holds count for the disk countdown timer.

edde I/O flag. If true (nonzero) output should be sent to the editor.

endtable Holds the end address of the RAM token table.

execbuf

extant Holds the address of the vocabulary 'extant' array.

gticks Holds count for a general countdown timer.

gvect Used as a general execution vector.

here Holds the address of the next available location in the code area of the current open vocabulary.

hld During number formatting, holds the current offset into the string being constructed in the pad.

in Pointer used to mark **word**'s progress through the input stream. **in** always holds the address of the next byte to be examined by **word** in the input stream.

intexecvecs Interrupt execution vectors.

inuse

itx Holds the address of the current input text.

jdn

kev

kstat

kval

last4thline

lasttok

len Holds the length of the word most recently extracted from the input stream by **word**.

limit Used to hold the end address of the block of text to be examined by the word **interpret** .

locals Used during the compilation of local variables to keep track of the amount of local variable return stack storage space which is required by the definition currently being compiled. Used primarily by the words **doloc**, **local**, and ; .

localvoc Holds the address of the temporary hidden vocabulary used to hold the names of local vocabularies.

location Used during the compilation of local variables to hold the address of the special, invisible vocabulary used to hold the names of the local variables used by the word currently being compiled.

loops System integer used during the compilation of a 'do' loop program control structure to hold the amount of return stack space currently required by the definition being created. Used primarily by the words **do** , **loop** , **+loop** , **doloc** , and ; .

lp I/O flag. If true (nonzero) output should be sent to the line printer.

maxblks

modifiers

nesting System state flag, if true (nonzero) the system is in a temporary compiling state (for interactive execution of program control structures). If false (0) the system is in the interpreting state.

nestype The **nestype** system integer is used to hold a flag which, during the compilation of program control structures, holds a '-1' if the program control structure currently being compiled is a 'do' loop or holds a '0' if the program control structure being compiled is a 'begin' loop. Used by the compiling word **{while}** .

newest Holds the header address of the most recently defined colon definition.

origin Holds the address of the start of the 'tFORTH' dictionary.

pad Holds the address of a location 128 (decimal) bytes from the start of a 384 (decimal) byte scratch location. The pad area is used by the number formatting operators and by the editor [CALC] function.

panicked

ramend Holds the address of the end of RAM memory.

ramstart Holds the address of the start of RAM memory.

ringsoundaddr
Holds the address of a general ring sound routine.

savenest

savestate

scontcopy

screen Holds the address of the start of display memory.

screensize

ser I/O flag. If true (nonzero) output should be sent to the serial port.

soundaddr Holds the address of a general sound routine .

soundcount Holds general sound count.

sp0 Holds the address of the base of the parameter stack.

special Holds the address of the keyboard 'special' array.

state System state flag, if true (nonzero) system is in the compiling state. If false (0) the system in the interpreting state.

str Each time **word** gets the next word from the input string, it places the address of the character string in the **str** system integer. See the description for the system integer **len** also.

strings

targeting Flag. If true (nonzero) target compilation is occurring.

ticks Holds time ticks.

tokens One of two system integers used to help in the assignment of tokens to new words (**lasttok** is the other system integer used for this purpose). See the techical discussion on local variables.

top Holds the address which is one byte beyond the top of 'tFORTH' memory.

vdelay Holds the value used to specify how long the video screen should stay 'on' on an unused terminal.

vticks Holds count for the video countdown timer.

x Holds 'tFORTH's column output position.

y Holds 'tFORTH's row output position.

APPENDIX: DEFINING WORDS

Defining words are the most powerful of the Forth words because they allow the programmer to create new Forth words. The defining words which have been used so far (although they weren't categorized as defining words previously) are **:** , **integer** , **string** , and **vocabulary** .

CREATING NEW DEFINING WORDS

The words **<builds** and **does>** are used to create new Forth defining words. The word **does>** (described in "Starting Forth") was not included in tForth since the ability to create new defining words was not required in the Cat system. However, the second listing in the appendix shows three words which, when included in the tForth dictionary, add the ability to create defining words to the Cat system.

The format for creating defining words with these extension words is:

```
: <name>
    <builds    ...compile time actions...
    does>      ...execution time actions...
    ;
```

(Note that in Chapter 11 of "Starting Forth" the word **create** is used in place of the word **<builds**.) Here is how the 'characters' example found on page 293-295 of "Starting Forth" could be implemented using **<builds** and **does>** :

```
: characters ( n - )
    <builds                ( Creates a new dictionary entry. )
          dup ,            ( Compile the count into the first. )
                           ( Position in the child word's. )
                           ( Parameter field for future )
                           ( reference. )
          allot            ( Allocate count bytes in the code )
                           ( area for the string characters. )
    does>                  ( Marks the beginning of the run- )
                           ( time code, leaves the parameter )
                           ( field address of the child word )
                           ( on the stack at run-time. )
          dup              ( Copy the parameter field address. )
          4 +              ( Advance the address past the )
                           ( four byte count value to the )
                           ( actual start of the string. )
          swap @           ( Swap the string address with the )
                           ( count addr and fetch the count )
                           ( value from the count address. )
    ;

    20 characters me
    me . . 20 479A0
```

When the child word created by **characters** (**me** in the above example) is executed, it returns the length of its character array and the address of its character array on the stack.

Defining the word **words**:

```
hex

0  integer  doesptr        ( holds the address of where to patch token )

:  <builds  ( - )
            create          ( create the new name )
            4ED3 w,         ( it will do a nest )
            here doesptr to          ( remember where the token should go )
            0  w,           (  and make space for it )

;
:  <does>>  ( - )
            raddr           ( terminate this word and get token addr )
            w@              ( find the compiled token )
            doesptr  w!     ( and back patch it )
;

:  does> (  - )
            recycledtoken   ( allocate a token )
            dup  -1  =
                 abort" out of tokens"    ( and test it first )
            dup  100 <
                 abort" must be a 2 byte token"
            lasttok to          ( remember which token it was  )
            compile <does>>     ( compile the runtime version of does> )
            lasttok w,          ( and the token for the does> code )
            align               ( align here pointer on even address )
            here lasttok +table !  ( and store address of does> part in table )
            4ED3 w,             ( nest the does part )
            compile raddr       ( so doesn't return and so data address is )
                                ( on stack )
;
immediate

:  test  ( n  --  | n  --  addr \  compiles an array with n entries.
runtime takes an entry number and returns the addr for it )
            <builds 4  * allot  ( allocate space )
            does>
                 swap           ( calculate address )
                 4 * + ;
```