Chapter 2
Introduction to JForth

JForth is a programming language that allows you to interact with your Amiga. When you are programming in JForth, you are "inside" the language. You can access any data structure, test any routine, or use any development tool, right from the keyboard. This direct communication with the computer can improve your productivity, giving you additional time to improve the quality of your software products.

JForth is based on the 1983 standard Forth language. Forth was first conceived by Charles Moore when he wanted a new language for controlling telescopes. He developed a language based on a dictionary of words. This dictionary can be extended by writing new words based on the old ones. Since Mr. Moore's original version, Forth has been translated to almost every type of computer from the biggest mainframes to the smallest microcomputers.

Since a minimal Forth can be implemented in just a few kilobytes of memory, it is often used in very small embedded computer systems for process control and robotics. Forth, however, is equally appropriate for larger, more advanced, computers like the Amiga. Forth is a very flexible language and can be adapted to larger computers without losing the flavor of the original language.

JForth is an implementation of Forth designed specifically for the Commodore Amiga. JForth uses a 32-bit stack and compiles directly to 68000 machine code. This makes JForth faster than most Forths. JForth also provides an extensive set of tools for accessing the special features of the Amiga. You can call any Amiga Library routine by name and reference any Amiga structure using constructs similar to 'C'. JForth also has some special toolboxes that support simple graphics, Intuition menus, IFF files, and other Amiga features. These toolboxes can be used directly to simplify your Amiga programming. Source code for all these toolboxes is provided so you can customize them if needed or study them as examples of Amiga programming. JForth also provides over a dozen small sample programs for those, like me, who learn best by example.

JForth also allows you to do things that are unique in the Forth experience, the most dramatic being CLONE. This exeptional utility allows you to create a totally independant, standalone version of your program of minimal size that is entirely royalty free!

Major Functional Systems

For a full list of the tools that JForth offers, you should read the next chapter about the various files on the disks. The systems mentioned here are ones that we feel are major ones not to be missed.

Amiga Library Calls

JForth provides a system for easily calling any Amiga Library routine by name. It will figure out what 68000 registers the parameters go into and build the appropriate code. Thus you can simply pass parameters on the stack in the order described in the Amiga documentation. A number of toolboxes have been written for supporting specific parts of the Amiga Library including EZMENUS, Graphics, the Serial Device, ANSI codes, and much more. Chapter 18 covers the details of Amiga library access.

Amiga Structure Support

JForth provides the equivalent to all of the ".h" include files from 'C'. These ".j" files define all of the necessary structures and constants for passing information to the Amiga Library routines. Structures can be dumped interactively with all members shown by name and value for debugging by using DST. Chapter 18 further discusses this helpful feature.

ARexx Support

ARexx is a language that helps various applications communicate with each other. A spread sheet program, for example, could communicate with a data base program. These tools help you write ARexx compatible programs. Chapter 23 further discusses AREXX.

Assembler

JForth support two 68000 assemblers, one with Reverse Polish Notation (RPN) and one with Motorola like syntax. The RPN assembler can be used to create macros but the Motorola assembler is easier to read. We also offer a Disassembler. All are covered in depth in Chapter 14.

Block Support and SCRED

For those who prefer the old fashioned BLOCK or SCREEN environment, we provide LIST, LOAD and the standard line editor. We also provide a WYSIWYG SCREEN editor called SCRED. (We recommend the use of normal text files with JForth). Chapter 15 discusses the BLOCK environment.

Clone

Clone can be used to generate small, royalty free, executable images of your JForth programs. Clone will take a compiled JForth program, extract out all of the code and data needed to run it, and reassemble a smaller version of it. All of the JForth development tools, the name fields and link fields and any other unused words are left behind. The final image size is comparable to images created using a 'C' compiler and linker. Images can be saved with a symbol table for use with WACK or other debuggers (if needed). The JForth Source Level Debugger can also be used with Cloned programs. Most programs will Clone without modification if they follow a few simple rules regarding run time initialization of variables or arrays containing Forth addresses. See Chapter 7 for further information.

Debugger

JForth provides a source level debugger that allows you to single step through your code. At every step you can see what is on the stack. You can also examine the return stack, dump memory, set breakpoints, control execution, or even enter Forth commands. Chapter 13 fully documents the source level debugger.

Floating Point

JForth support both the single precision Fast Floating Point and the IEEE double precision. These words conform to the Forth Vendors Group Standard. Chapter 9 describes the Floating Point implementation.

IFF Support

JForth provides general purpose tools for reading and writing IFF files. It also provides a toolbox specifically for ILBM graphics files. This allows you to use pictures, brushes, anims and animbrushes from a paint program, or other source, in your programs. JForth also provides tools for animation and presentation such as, blit, wipe, fade in, fade out, etc. Powerful graphics and animation programs can be created using these tools. Chapter 21 will further guide you in this area.

Local Variables

Local variables can simplify the definition of complex words by eliminating much of the stack manipulation. Local variables are fast self fetching variables that allow reentrant recursive code to be written. Using regular VARIABLEs can make a word non reentrant. Local variables are specifically covered in Chapter 11.

Modules

JForth's precompiled modules provide a method for rapidly accessing code that is used during

complation. This includes the Assemblers and Disassembler, and the Amiga include files. These modules are dynamically linked into the dictionary when needed. This way, they do not take up space in your normal dictionary, yet are instantly available. See Chapter 16 for more information about modules.

MultiStandard

This system allows you to switch easily between JForth and the major standards - FIG , '79 , and '83. This is handy for compiling code written using other Forths. Multistandard is discussed in Chapter 17.

ODE

ODE is an Object Oriented Development Environment similar in concept to SmallTalk. It allows you to define classes of intelligent data structures, then create as many copies of these data structures as needed. This technique can simplify programming immensely by making it easier to write reusable code. Chapter 10 is devoted to ODE.

Profiler

This optimization tool monitors the operation of a program to find out where it is spending its time. Most programs spend most of their time in a small portion of the code By identifying this code, one can focus efforts at optimization where they are most needed. The profiler is documented in Chapter 17.

Textra

Textra is a powerful, yet easy to use, multi-window text editor designed for programmers. It uses the mouse to select text and allows Cut, Copy and Paste, operations between its windows. It also incorporates ARexx hooks that allows you to use provided macros (.textra files) for text processing (or write your own). Textra can communicate with JForth such that you can compile directly from the editor. If an error is encountered while compiling, Textra will highlight the error so the you can fix it quickly. The documentation for Textra may be found in the JTX:DOCS directory. To access the '.textra' REXX scripts enter:

```
COPY JRX:SCRIPTS/#? REXX:
```

Grand Tour of JForth

The section describes some of the special features of JForth. If you have never used any Forth, you may want to try the tutorials first. Then come back to this chapter because it describes features you should know about.

JForth has so many tools and utilities that it is easy to miss something useful. We recommend studying this manual carefully so that you can take advantage of all that JForth has to offer. You should also read some of the source code files where you may find some undocumented goodies.

Let's start with a grand tour of the main JForth tools. Bring up JForth and try out these commands as we discuss them. (If you are unclear about how to read stack diagrams, please try the tutorial first.)

`WORDS ( -- , list the words currently available )`

This command lists the words in the dictionary that you can call. Although thousands of words may be listed, you will only need to learn a few. You can stop the output of this list by hitting a key. Hit return to continue listing or enter "quit" to stop. You could also enter a Forth command while WORDS is paused. This pause feature is provided by ?PAUSE which is described in the glossary. VLIST is a synonym for WORDS.

`WORDS-LIKE ( <word-fragment> -- , list all with fragment )`

Use this if you want to see all of the words that have some word fragment in their name. Enter:

```
WORDS-LIKE  EMIT  ( all words with EMIT )
WORDS-LIKE  +     ( all words with '+' )
```

As a shortcut, if you hit <Function-Key-6> it will insert WORDS-LIKE into your input stream. This will save you from having to type it in. Hit the <HELP> key to find out about other function key assignments. Now hit the <Up-Arrow> key repeatedly to scroll through your previous commands just like in the Amiga DOS shell. You can look in the section on Command Line History for more information about this.

FILE? (<name> -- , show file and source code)

If you are curious about how a word is defined you can use FILE? to see what file it is from. It will then ask you if you want to see everywhere it is referenced or defined in that file. This will work for every word in JForth except those from the Assembly Language Kernal which is proprietary. (Hey! We gotta have some secrets!)

```
FILE? ANEW
```

EACH.FILE? (<name> -- , show every file and source code)

If a word has been redefined, this command will find every occurrence of it in the dictionary and show it to you.

```
EACH.FILE?  AUTO.INIT
EACH.FILE?  [FORGET]
```

DEF (<name> -- , disassemble word)

Here is a handy word for those of you who are familiar with 68000 assembly language. DEF will disassemble any Forth word. (See the chapter on the 68000 assemblers for more info.)

```
DEF 1+
DEF CMOVE
```

The first time you enter DEF it will load a precompiled MODULE containing the disassembler. This gives you quick access to DEF without taking up room in the dictionary or having to compile it yourself.

DOS (<command-line> -- , pass command to Amiga DOS)

You can execute almost any Amiga DOS command line except those that require input, and CD. For CD, just enter CD without using the DOS keyword (CD has been provided for you in the JForth dictionary). Some common DOS commands have been predefined. (See the file JF:DOSCOMMANDS for more information.)

Execute arbitrary DOS command

```
DOS TYPE S:STARTUP-SEQUENCE OPT H
```

For CD, use CD directly.

```
CD RAM:
```

Use predefined command.

```
DIR DF0:
```

Pass command in string.

```
" RENAME RAM:MOO RAM:GOO"  $DOS
```

MEASURE (<Forth-line> -- , time Forth commands)

This is handy when you are trying to speed up your code. This command will tell you how long things take. It will take the commands that follow, execute them and tell you how long it took.

```
MEASURE DIR
```

BENCH (<Forth-line> -- , time commands with overhead)

MEASURE and BENCH are only accurate to about 1/50th of a second. If something takes less than 1/2 a second you should call it N times in a loop then divide by N to find out the actual time. If you want to take the loop overhead into account, use BENCH and BENCH.WITH.

```
: TDO  ( N -- )
    0 DO LOOP
;
: TSWAP ( a b count -- a b | b a )
    0 DO SWAP LOOP
;
1,000,000 BENCH.WITH TDO
11 22 33 1,000,000 BENCH TSWAP
```

INCLUDE (<filename> -- , compile source code from a file)

You can specify a complete pathname or use the current directory. INCLUDE can be nested so you can call INCLUDE in the file you are currently INCLUDEing. (See INCLUDE? in the glossary.) INCLUDE is a very important word so we have assigned it to <Function-Key-1>.

```
INCLUDE JU:RANDOM
6 CHOOSE .  ( random number, 0 <= R < 5 )
6 CHOOSE .

TYPEFILE JU:LOGTO
MEASURE INCLUDE JU:LOGTO
```

JForth uses a HASHED dictionary to speed up compilation. This is a special way of looking words up directly instead of scanning the entire dictionary. If we turn off hashing we can see how hashing speeds up the compile process...

```
HASH.OFF
MEASURE INCLUDE JU:LOGTO
```

Notice the difference in compile speed. Hashing does not, however, come without a price. JForth has to build a table of pointers for hashing to work. This rehashing is required whenever you FORGET code or do a GETMODULEs. If we recompile LOGTO with hashing on we will have to rehash.

```
HASH.ON
MEASURE INCLUDE JU:LOGTO
```

This is still faster than compiling with hashing off. Hashing is most effective with large files when the dictionary has lots of words already defined. Some people have reported an 8-9 times speed up. Hashing is least effective when you recompile small files but they are pretty fast anyway. Hashing will speed up compilation in almost every instance so we usually leave it on.

MAP (-- , display system status)

This word tells you all about the current system, how many files are open, how much room is left in the dictionary, etc. This is assigned to a function key. Hit <HELP> to find out which one.

SAVE-FORTH (<filename> -- , save current JForth system)

If you compile some code and want to save the dictionary in its current state, use SAVE-FORTH to write it to a file. You can run this image later, just like you did COM:JForth. If you try this you should write it to a work disk (NEVER write to the original JForth disks). If you want the new image to have more dictionary space available, increase the value of the variable #K before you SAVE-FORTH...

```
#K @ .   ( see how much is there )
```

```
50 #K +! ( increase it by whatever you think you need )
SAVE-FORTH  WORK:MY4TH
```

Be mindful that when you SAVE-FORTH, any opened files or allocated memory will not be preserved in the saved-image, so your programs should re-establish such resources in an INIT-type word. Unless your program meticulously cleans-up after itself, you should SAVE-FORTH after immediately compiling and before running it so that you save a pristine image. See the glossary for more information.

JForth Compared to other Forths

This section describes some of the ways in which JForth differs from other Forths.

32 Bit Stack

Forth uses what is called a "data stack" for passing values between words. It is a place to put numbers, addresses or other values. Values can be added to the stack and removed just like on a stack of plates. Most Forth implementations use a 16 bit stack. This means that the numeric range in a standard Forth is -32,768 to 32,767. JForth and a few other Forths, use a 32 bit stack which gives you a numeric range of -2,147,483,648 to 2,147,483,647. Since JForth addresses are also 32 bit, you can address a much larger memory space than a 16 bit Forth.

Jump Subroutine Threading

Another major difference between JForth and most other Forths is that JForth is a true compiler. In most Forths,when you "compile" a word, you actually end up creating a table of tokens or pointers to the other words that are called. If your word calls '+' it will put a pointer to the plus function in your word. When you execute your word, a program called the inner interpreter examines these pointers and then executes the appropriate functions That is the traditional method.

In JForth we use what is called Jump Subroutine, or JSR, Threading. That is where the 'J' in JForth comes from. In JSR threading, actual 68000 machine code is compiled into your word. If you call a word, JForth will compile a 68000 JSR instruction to make that call. This eliminates the need for an inner interpreter. If the subroutine is small enough, JForth will copy the called routine inline into your routine thus avoiding the small overhead of the JSR instruction. Since the 68000 executes code directly, JForth will run 2-3 times faster than a traditional Forth on the same machine.

Relative Addressing

In JForth, addresses are expressed as offsets relative to the base of the JForth image in memory. This is important because AmigaDOS will load JForth at different places in memory at different times. Despite this, your programs can use the same relative addresses at different times. This can simplify Forth programming because variables, CFAs, dictionary links, and other addresses keep the same relative addresses even thought their absolute addresses may change.

Amiga Libraries uses absolute addresses. You will, therefore, have to convert relative addresses to absolute before passing them to the Amiga. This is a simple operation. On the whole, we feel using relative addressing is an advantage over absolute addressing.

Here are the words used to convert addresses:

```
>REL  ( absolute-address -- relative-address , convert )

>ABS  ( relative-address -- absolute-address , convert )
```

(Also remember that the addresses on the return stack will be absolute as well. We have provided words for accessing "inline" data based on reading the return stack. Please see the appendix on Transportability.)

Text File Input

JForth will compile programs from "normal" ASCII text files created using standard Amiga text editors. Thus you can use you favorite editor with JForth instead of the TEXTRA editor provided. (Traditional Forths use a system of 1024 byte source code blocks. We provide support for this system for those who want it.)

NOT versus 0=

In the Forth '83 standard, NOT performs a 1's complement operation. Since NOT is usually used for negating a logical value, this can sometimes give surprising results. In Forth, any non zero value will be considered true by IF and other conditional words. The traditional NOT only works with a true value of -1. For that reason, JForth defines NOT as 0= which negates any true value.

```
In Forth '83
        -1 NOT ( is false )
         1 NOT ( is true!!, = -2 )
         1 0=  ( is false )

In JForth
        -1 NOT ( is false )
         1 NOT ( is false )
         1 0=  ( is false )
```

The JForth word COMP will perform a 1's complement operation like the '83 NOT. If you want the traditional NOT you can redefine it or use the Multistandard package which gives you strict conformance with '83, FIG or '79 standards.

Floored Division

JForth division is not floored. This means that the results of a division are rounded up if they are negative. In floored division the result is rounded toward negative infinity.

```
In JForth:   -13 2 / .  ( gives -6 )
Floored:     -13 2 / .  ( gives -7 )
```