Pocket Forth, version 0.5 and 1.5
by Chris Heilman 6/30/91

Contents

Introduction

Forth is a beautiful language. It is easily spoken and source code looks like poetry. Oh yes, Forth produces fast compact code. Pocket Forth is a small Forth system for the Macintosh computer, founded on well documented principals(1,2,3). While most of Pocket Forth is 'standard' usage, it follows no standard rigorously.

Pocket Forth is based on Flint by G. Yates Fletcher(4), figFORTH(5,6) and the Forth described in Starting FORTH by Leo Brodie(7). I wrote Pocket Forth to experiment with the Macintosh toolbox. Pocket Forth can produce applications and desk accessories that run on any Macintosh.

Programs are easily modified and you are encouraged to adapt Pocket Forth into your own program. Toward this end, you own your copies of Pocket Forth. This means that programs may be distributed with no royalties, homage, or other restrictions, even if the Forth system is available to the end user.

> **NOTE**: Pocket Forth is not warranted for any particular purpose. That means you assume all of the responsibilities in the use of Pocket Forth.

I have attempted to make Pocket Forth free of bugs, but low level languages, such as Forth, allow you access to the operating system. Therefore, use Pocket Forth with care in conjunction with other programs or files. This is especially true when writing programs which access the disk, or when experimenting while using other programs that do. Always open a backup of any file you may want to keep, including Pocket Forth itself. Never run originals!

If you are new to Forth, read Starting FORTH to learn the language. Chapters 1, 2, 4 through 8, parts of 9, 11 and 12 apply to Pocket Forth, but read the whole book.

Preparation

There are two Pocket Forth programs: a small application and a desk accessory. To use the Pocket Forth application, follow this procedure:

- Back up Pocket Forth before running it. Put the original away.
- Double click the program's icon. If everything is ok, a window will appear and say so. The 'ok' prompt and cursor indicate that Pocket Forth is ready for input.

To prepare the desk accessory:

- Backup the file named PocketDA and put the original away.
- With System 6 or less, install the desk accessory into the system file with the Font/DA Mover. System 7 users should open the suitcase file and drag the icon inside of it to the Apple Menu Items folder.
- Select Pocket Forth from the Apple menu (or open it normally under System 7). Its window will appear with the 'ok' prompt, awaiting your input.

Alternatively, launch the desk accessory with a utility program such as Suitcase. Follow the directions of your program to install Pocket Forth. This method isolates your System file from alteration.

Using other programming utilities, your environment can be as rich as you like.

About Pocket Forth

Pocket Forth is a simple interpreter that can execute code or compile it into a dictionary for later execution. Source code text is typed from the keyboard, pasted from the clipboard or loaded from a file. The text is interpreted one line at a time. When the line is completed an 'ok' prompt appears and the input sequence restarts.

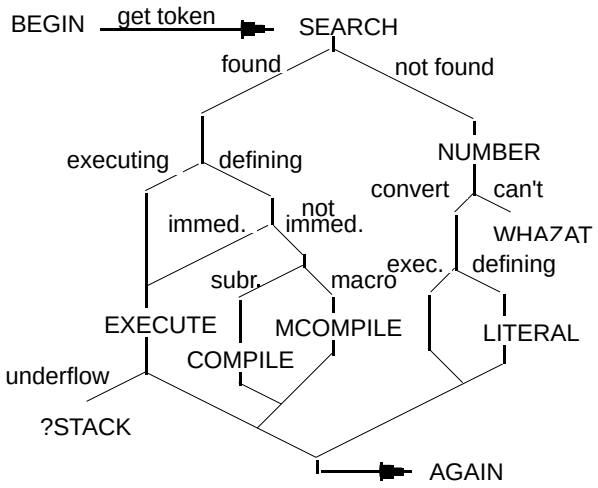
The text interpreter parses tokens from the input line (also called the input stream). Tokens are any group of characters, excluding space, carriage return and null. Individual tokens are separated from each other by one or more spaces. Carriage return or null signal the end of the line.

The dictionary is a list of named subroutines, called words. Each token of a line is matched to a word found in the dictionary. In executing mode, the word's subroutine is run and the interpreter continues with the next token. Tokens not found in the dictionary are converted to a numeric value by the routine named "number". If the token is not a valid number, an error is signaled. See the error section.

Some words place the interpreter into compile mode. In compile mode, most words are written into the dictionary as machine language subroutines. Thus, Pocket Forth produces subroutine threaded code.

Here is the structure of the interpreter:

```
FORTH PSEUDO CODE

BEGIN  get token  SEARCH
  found IF
    defining IF
      immediate IF
        EXECUTE  ?STACK
      ELSE
        macro IF  MCOMPILE
        ELSE  COMPILE  THEN
    ELSE  EXECUTE  ?STACK  THEN
  ELSE  NUMBER
    not IF  WHAZAT  THEN
    defining IF  LITERAL  THEN
  THEN
AGAIN
```



Text Files

Typing into the Pocket Forth window supplies text to be interpreted. None of what is typed is saved, so write your programs with an editor.

Pocket Forth does not use Forth's traditional block input, rather it interprets normal text files. This allows you to prepare program source files with your favorite text editor and tools. Which editor you choose will depend on the memory available, which Pocket Forth type you are running (application or DA) and your personal preference.

On Macintoshes with MultiFinder or System 7, the Pocket Forth application runs in multitasking mode. A 32K byte segment is the minimum required, but 128K bytes or more may be utilized. Accessory editors can be used with Pocket Forth on even the smallest Macintosh.

To accomplish multitasking on an older Macintosh or System, the DA version of Pocket Forth can be run within an editor, the Finder or any program that allows desk accessories. As long as the host program has an Edit menu, an accessory editor can be run simultaneously. The DA version also works well in multitasking environments.

Text is interpreted by Pocket Forth either from the clipboard or directly from a file. Select and copy text from a source file and paste it into the Pocket Forth window. The pasted text is printed and interpreted just as if it had been typed in. The 'ok' prompt returns when the paste is complete.

Files are interpreted directly with the word "-->" (pronounced “load”). Follow "-->" with the name of the file. If the file is in an HFS folder, a path name must be used. Here is an example of the use of "-->":

```
--> diskName:folderName:fileName
```

This will load the file called 'fileName' in the folder 'folderName' from the disk 'diskName'. If the files to be loaded are in the default or System folders, only the file name is necessary. Launch a program from a folder to make it the default folder.

File, folder and disk names that are to be used with "-->" must not contain any spaces because the text interpreter stops at spaces, causing the entire name not to be found. This or other disk errors print an error message and return control to the keyboard. See the error section for more information.

The word "open" presents a standard file open dialog for your perusal. With it you can select a file to interpret. There are no restrictions on the filename to be used with "open". The application's File menu has the Open… command available on it for this use. Although desk accessories do not have an Open menu item, the command key and ‘O’ combination is available in the DA version.

Files should be saved from the editor prior to pasting or loading into Pocket Forth. Text files require a carriage return ($13) at the end of each line, and only the first 80 characters of a line are considered.

A file stack is maintained so that files can load other files. Files can 'nest' up to five levels deep. When a file is finished interpreting, the input stream returns to the previous file and begins with the line after the "-->" was encountered. When all levels have been completed, control returns to the keyboard. If more than 5 files are nested, an error may result.

The text of the current file and all pending files is kept in memory. While files can be any size, the memory available will limit how large a file will load. Use short files that deal with single aspects of a program. Reusable code in small files can be loaded by other files. Make source code simpler by loading a list of modular files and code that is specific to your particular program.

Comments document a program within the source code. Create comments by enclosing text within parenthesis. Two main types of comments are stack comments and general comments.

A stack comment is a one line diagram of the effect of a word's execution on the parameter stack. Stack comments may also show compile mode or the return stack. The purpose of a stack comment is to unambiguously show the effect of a word. See the stack section.

General comments tell what a word does, or why it has been defined. A short explanation comment should accompany each word defined. Long definitions may contain many comments. By commenting well, your program will be easier for others to understand and easier for you to maintain. See the example files for the use of both types of comments.

Memory

Pocket Forth uses memory in six ways. There are two stacks, variables, the dictionary, free memory available to the the dictionary and absolute addresses.

Stack addressing is implied, that is, you push and pop, and the stack determines what address these actions access. Variable, dictionary, and free memory addressing are all done with a 16 bit relative address. Absolute addresses cover the entire addressing range of the 680x0 with a 32 bit double number.

Understanding the use of relative and absolute addressing is vital to programming Pocket Forth.

Absolute Addressing

An absolute address is the actual address as seen by the microprocessor. Pocket Forth treats absolute addresses as double numbers. A double number is two consecutive 16 bit numbers or a 32 bit number. The double number words (such as "2swap" "d+" and "2@") can manipulate absolute addresses. Absolute addressing allows access to any address whether 24 bit or 32 bit addressing mode is used.

In addition to double number words, Pocket Forth has some words to deal directly with absolute addresses. The words "l@", "dl@", "l!" and "dl!" read and write absolute addresses outside of the relative address range.

See the glossary for the specifics of using double number and absolute addressing words. Abbreviate absolute addresses as 'dabs.addr' in comments.

Relative Addressing

The addresses used within Pocket Forth are relative addresses, that is, 16 bit numbers, relative to a base address. Pocket Forth can be anywhere in memory and relative addresses will remain constant.

Relative addressing uses the 'indirect with displacement' addressing mode of the 680x0. Addresses within ±32K bytes of the base address can be accessed using a 16 bit relative address. Pocket Forth's dictionary starts at zero and uses only positive relative addresses, so the maximum dictionary size is 32K bytes.

Free memory, that is, memory available for dictionary expansion, is determined with the word "room". As distributed, 4K bytes of free space are provided. The default amount can be set by typing "save" (or using the application's Save Dictionary menu item) after adjusting the allocated memory with "grow".

The word 'address' or the abbreviation 'addr' usually means a relative address.

Relative and absolute addresses can be converted from one to the other. The word ">abs" converts a relative address to an absolute address. The result is a double number.

The word ">rel" converts an absolute address to a relative address. Only absolute addresses actually within the relative addressing space will be correctly converted. Absolute addresses outside of this range are mapped into the relative addressing space, causing the wrong address to be calculated.

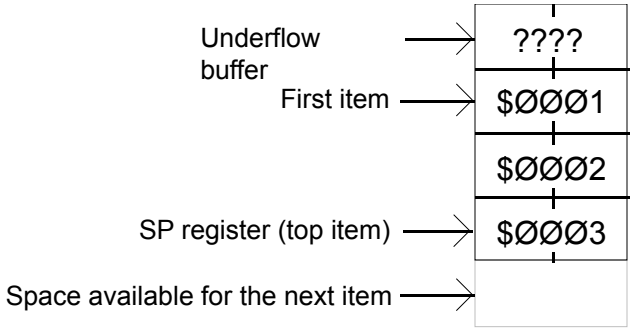
Stacks

The stack data structure is used throughout the computer world, but most languages keep their stacks hidden. On the other hand, Forth stacks are open, allowing complete access.

Like most Forths, Pocket Forth uses a parameter stack and a return stack. The term 'the stack' used by itself usually means the parameter stack. In contrast, 'the stack' in other languages means the system stack, which is Pocket Forth's return stack.

The parameter stack, used to pass values to and from routines, is under your full control. It holds up to 256 sixteen bit numbers. See the error section about under- and overflow conditions.

This is a memory diagram of the parameter stack after the numbers 1, 2, and 3 (in hex) have been placed on the stack in that order. Notice that, although the stack grows downward in memory, the lowest item, the last one added to the stack, is called the 'top' of the stack. Visualize a stack of dishes from a cafeteria to get the idea.



Because the return stack is the system stack, it is used for subroutine threading by Pocket Forth and other programs, including the operating system. Runtime threading by pushing and popping addresses to the return stack is handled automatically by the processor.

The return stack can also be used explicitly to store temporary data and to interface with toolbox routines. Used for data storage, the return stack acts as a local variable, which is invisible outside of the definition where it is used. For example, here are two versions of a word that returns a product and a quotient:

```
: PANDQ ( n1 n2 n3 -- n1*n3  n2/n3 ) dup rot swap / rot rot * swap ;
: PANDQ ( n1 n2 n3 -- n1*n3  n2/n3 ) >r swap r *  swap r> / ;
```

In the first definition, "pandq" is written using parameter stack manipulation only. Holding n3 temporarily on the return stack makes the second definition smaller. It is also faster because return stack words, such as "r", "r>" and ">r", compile inline code instead of subroutine calls.

Used this way, the return stack must be carefully balanced between the beginning and end of a routine. An unbalanced parameter stack can cause an incorrect result, but a single number out of place on the return stack causes an immediate or delayed crash.

Toolbox parameters and results are passed by pushing and popping the return stack. In addition to the words ">r"and "r>", Pocket Forth adds the words "2>r", "2r>" and "a>r" to facilitate calling toolbox routines. Refer to the glossary, the example programs and the toolbox section of this manual for further information on the use of these words.

Dictionary

Pocket Forth's dictionary is a linked list of subroutines and other objects. Each object in the dictionary is named so that it may be matched to a token from the input stream. This makes the interpreter seem conversational.

An algorithm named "search" provides the ability to quickly locate any named routine. Given a token, "search" returns the address of the routine. Execution, compilation or some other action may be taken with the address.

Normally words which are typed (or loaded, or pasted) are executed immediately. The interpreter can also be in compile mode. The word ":" (pronounced “colon”) engages the compiler to add a new definition to the dictionary.

> **NOTE**: Actually the word "]" (“right bracket”) enters compile mode by setting the byte variable in "cstate" to 128. “Colon” creates the dictionary header, that is, the name and link fields, then calls "]". Like “colon”, ";" (pronounced “semi-colon”) isn't the actual word that leaves compile mode. The word "[" (“left bracket”) is called by “semi colon” to do that.

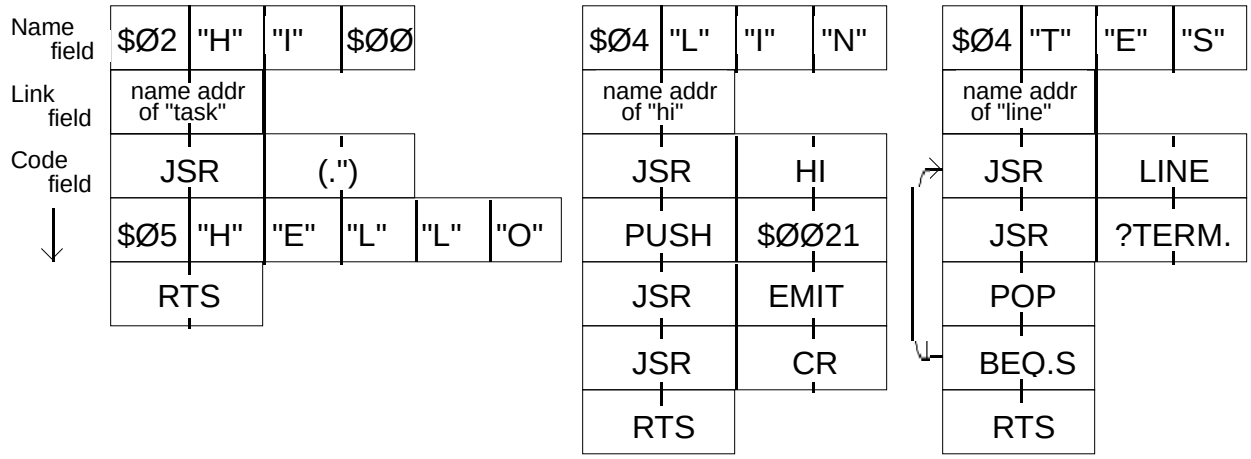
Anatomy of Dictionary Entries

“Colon” definitions consist of a name, a link address for searching and an executable subroutine. The subroutine can contain any code. Because definitions are themselves subroutines, definitions can call other definitions to any depth of nesting.

This simple example program illustrates the dictionary's structure:

```
: HI  ." HELLO" ;  ( print "hello" on the screen )
: LINE  hi  33 emit cr ;  ( print hello! and go to next line )
: TEST  BEGIN line  ?terminal UNTIL ;  ( continue 'til key press )
```

The following figure shows the memory used in the example to illustrate how definitions are constructed:



The three fields common to all definitions can be seen, as well as some special features in each one. Notice that dictionary memory is commonly used in 2 byte groups. These groups are called cells.

Name Field

The first part of a dictionary entry is its name field. A name field is four bytes long: the first three characters of the name are preceded by the number of characters in the name. Names such as "hi", with less than three characters, are padded with zeros so the name field is always four bytes long. Names with more than three characters are truncated.

Name collisions occur if the name field of two words is the same. For example, a word called "lint" would have the same name field as the word "line". The effect of a name collision is that references to the collided word are interpreted as the later defined word. References already compiled are executed normally. Redefining a word does not cause an error message because the dictionary is not searched during compilation, so choose names carefully.

Link Field

After the name field is the link field. This two byte field holds the name address of the word defined just prior to the current word. "Search" uses the link address to scan through the dictionary from word to word. A dictionary search starts with the last word defined, whose name field address is returned by the word "latest".

Code Field

The code for a word begins at its code field address, usually refered to as the address of the word. The address of a word can be found with the word " ' " (pronounced “tick”). Provided with the code address, the words ">name" and ">link" return the name and link addresses.

The code field consists of a subroutine, ending either with an RTS instruction or a jump to another subroutine. Most words compile to the form: JSR d(BP), where the displacement, d, is the relative address of the word.

Other features besides subroutine calls are found in the code field. Some words write code into the dictionary, or take some other action while in compile mode. These words are called immediate words, because they execute instead of compile when used within definitions. Some uses of immediate words are branching, literal numbers, strings and compile/interpret control.

Immediate words used by Pocket Forth are identified in the glossary. To make a new definition immediate, follow the definition with the word "immediate". To compile an immediate word, precede it with the word "[compile]".

Strings, such as 'HELLO' in the example are compiled as ASCII characters prefaced with a length byte. The word "(.")" (pronounced “paren-dot-quote”) prints the string data, then adjusts the return address to skip over the string. The word “dot-quote” compiles “paren-dot-quote” and a string into the dictionary, but does not print the string.

NOTE: Pocket Forth's “dot-quote” is not 'state smart'. Its behavior outside of a definition is the same as its compiling behavior, that is “dot-quote” compiles a call to “paren-dot-quote” and the string data. To have a word that prints interactively, define “dot-paren”:

```
: .(  41 word  here count type ;
.( This is a test) ( will print the enclosed text )
```

Numbers compile inline code that pushes a value to the stack. In the example above, in the word "line", the value, 33, is compiled as MOVE #33,-(PS) that is, push 33 to the parameter stack. Since this code is compiled, 33 is put on the stack when the code runs.

The word "literal" compiles a 'literal' value as described above, from data on the stack rather than from the input stream. Numbers are compiled as literals within definitions and pushed to the stack outside of definitions. The word "dliteral" works like "literal" but compiles a 32 bit push instruction from a double number.

Branching is controlled by immediate words that place short test and branch routines into the code. These branching structures are supported by Pocket Forth:

BEGIN ... (flag) UNTIL	loop until flag is true
BEGIN ... (flag) WHILE ... REPEAT	loop until flag is false
BEGIN ... AGAIN	loop until aborted or quit
(limit start) DO ... LOOP	do code (limit-start) times
(limit start) DO ... (count) +LOOP	do code (limit-start)/count times
(flag) IF ... THEN	do enclosed code if flag is true
(flag) IF ... ELSE ... THEN	do 1st part if true, 2nd if false

Branching words must be used in the correct combination, no compile time error will occur if they are not. These words are described further in the glossary.

NOTE: Back branches, that is branches toward lower memory are done by the word "back". "Back" will compile short branches if the displacement is less than 128 bytes, otherwise, long branches (four bytes) and compiled. Back is called by "until", "repeat", "again", "loop" and "+loop" so you usually don't need to call it yourself.

Macro Definitions

Another class of definition, similar to immediate words, are macro definitions. A compiled subroutine is four bytes long, and the JSR/RTS combination has some execution overhead. Routines that are only two or four bytes long are defined as macros. Macros compile their code inline rather than as subroutines. Inline words are the same size or smaller and execute a lot faster than subroutine words. (2,8). When not in compile mode, macro words function as subroutines.

These words are predefined as macro definitions:

2 BYTE MACROS			4 BYTE MACROS		
,	2*	DUP	!PEN	CBLK	OR
>LINK	2/	LATEST	+	CSTATE	OVER
>NAME	2>R	MON	+MD	D+	PMODE
>R	2DROP	NEGATE	-TO	DL!	R0@
0	2DUP	R	2OVER	DL@	S0@
1+	2R>	R>	AND	L!	TIB
1-	DNEGATE	RP@	BASE	L@	XOR
2+	DROP	SP@			

You may also define your own macros. Declare a word to be a macro definition by following the word's definition with the word "macro". A major caveat applies to user defined macros. Definitions become very large if macros longer than four bytes are used. This means that macros should be machine code only. If speed is the only concern, macros can be made from code longer than four bytes. See the Sieve file for an example of a longer macro built for speed.

Here are some useful macro definitions:

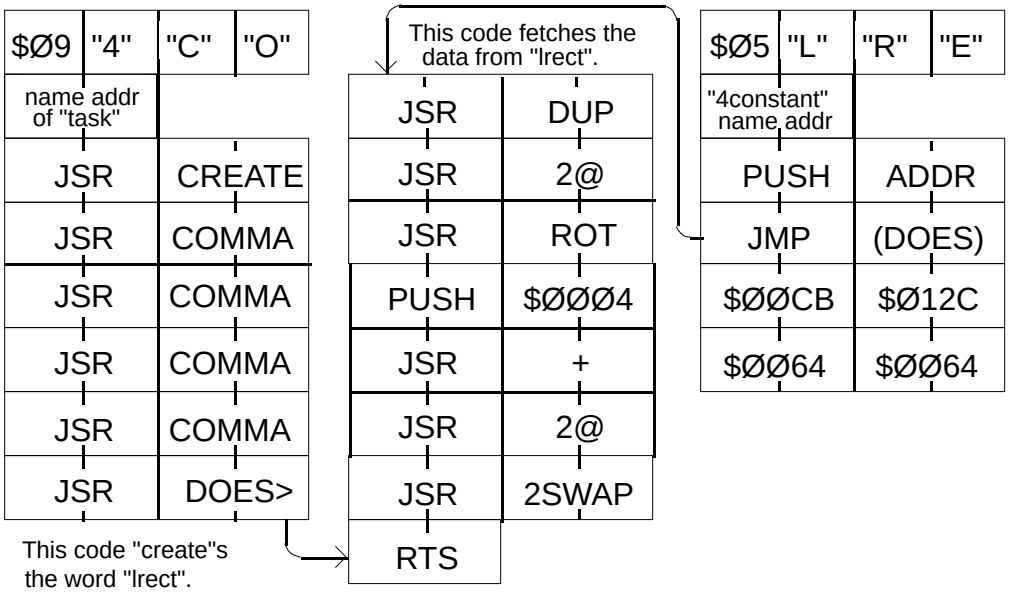
```
: 00>R ( rstack: -- 0 0 ) ( push a long zero to the return stack )
  ,$ 42A7 ; MACRO  ( clr.l -[rs] ) ( NOTE: rs is register A7 )
: 2R ( -- d ) ( rstack: d -- d ) ( get double from the return stack )
  ,$ 2D17 ; MACRO  ( move.l [rs],-[ps] ) ( ps is A6 )
```

Parameter Field

The parameter field is present only in words defined using "variable", "create" or other 'defining' words. Defining words build new words, called 'daughter' words, using the next token from the input stream for the name. When the daughter word later executes, it returns the address of its parameter field to the stack. If the "does>" follows "create" in a definition, the daughter word executes the code after "does>".

Words following "create" in the definition, build a data structure during compilation. Words after "does>" operate on the data at runtime. This is illustrated with the following example:

```
: 4CONSTANT  create , , , ,  ( stack at compile time:  n1 n2 n3 n4 -- )
   does>  dup 2@ rot 4 + 2@ 2swap ;  ( at run time:  -- n1 n2 n3 n4 )
100 100 300 200 4constant LRECT  ( create a 4 cell constant: "lrect" )
LRECT  ( put 100, 100, 300, 200 on the stack )
```



Variables are defined with the word "variable". Use "variable" to create data structures if "does>" is not needed. Variables are smaller and faster than 'created' data structures.

Memory Management

In a Macintosh, certain actions cause functional blocks of memory to move around. This is called memory management and is done by the memory manager in the ROM. (Brodie's fans should imagine a cartoon character for the manager.) Because of this, Macintosh programs must run from any memory location.

Blocks move only under certain conditions(9). It is likely that you are running Pocket Forth with other programs and can't control what those programs do, so the dictionary's block is locked.

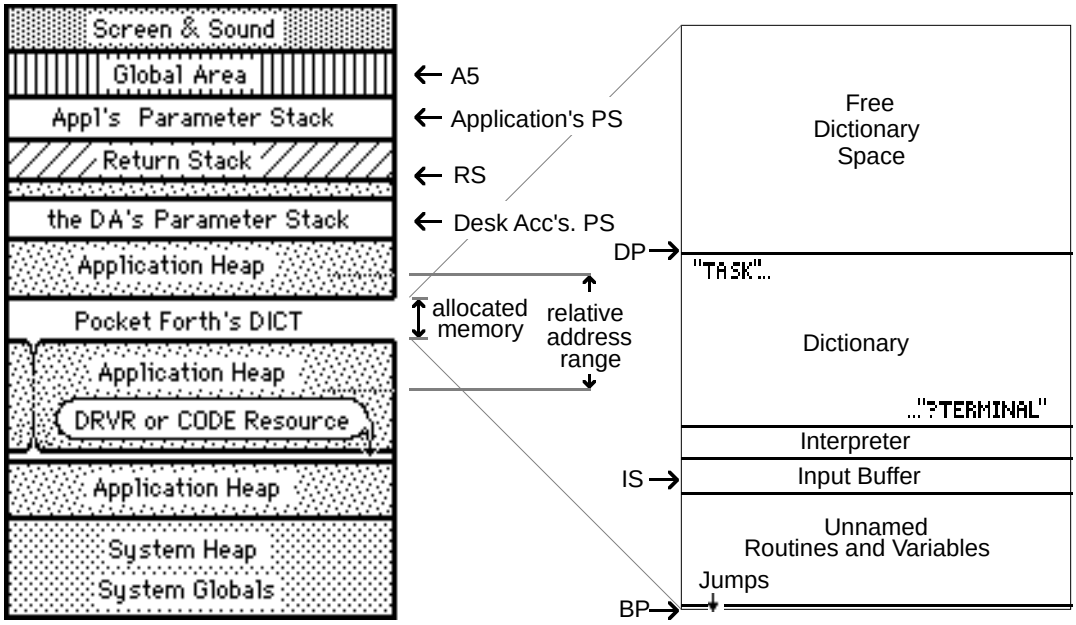
Memory available for a program is allocated above the dictionary. Do not confuse this with addressable memory. Allocated memory is memory you are allowed to use by the memory manager. Addressable memory is any address that can be accessed with a relative address. Programs which alter memory should respect the limit of the allocated memory when using relative addressing.

The word "grow" changes the amount of allocated memory. To change the amount of free memory place the number of bytes to increase it by onto the stack, then type "grow". To shrink the allocation, supply "grow" with a negative argument. You program should always have at least 80 bytes free. Pocket Forth comes with 4K allocated above the dictionary. Do not attempt to allocate more than 32K.

Fragmentation of the application heap is minimized by unlocking the dictionary when "grow" is called, so the dictionary may move in memory. For this reason any absolute addresses calculated before "grow"ing should be recalculated. Relative addresses are not affected. The same precaution should also be used with "save", which calls "grow". If a memory error occurs during "grow", you will get an alert (a red alert if you have a color monitor) and Pocket Forth will quit.

The dictionary is loaded from the disk as a 'DICT' resource. The entire dictionary is written to the disk by the word "save", replacing the previous DICT resource. This makes extensions and modifications that you have made a permanent part of your copy of Pocket Forth. "Save" can't be undone.

The memory map, with high memory at the top:



Other programs that may be in memory are not shown in this diagram.

Errors

Three types of error are reported: stack errors. numeric conversion errors and disk errors. All of these errors use the general purpose error word "abort". "Abort" prints a question mark, clears the file, parameter and return stacks and goes back to the prompt.

Stack errors occur when items are removed from the parameter stack when it is already empty. Since the parameter stack is often 'under flowed' this way, it is buffered so that no harm will be done by this condition. A stack error report looks like this: *?. Make a stack error happen by typing "drop" when you know the stack is empty.

Stack overflow on the other hand is not reported nor buffered. A runaway loop is the most often cause of stack overflow. The symptoms differ in each occurrence from a bomb box to extreme graphics and rude noises. Check your loops carefully.

Tokens not found in the dictionary are assumed to be numbers. If the conversion routine, "number" finds any characters greater than the value of "base", the token is not a number either. Unconverted tokens are reported by printing the token in capital letters, then doing "abort". Type "xx" to see this error.

NOTE: Since the stacks are cleared by "abort", typing "xx" (or any other non-word) is an easy way to clean up a stack with a lot of unwanted entries on it.

Disk errors are reported by printing "I/O error:" and the number of the error encountered. Look up the error numbers in Inside Macintosh(10) or other text. "Abort" is executed after the message is printed. Most disk errors result from incorrectly designating the path name. Using "open" avoids this problem.

Common disk errors include:

-35 No such volume error: disk name is mispelled or disk is not mounted.
-36 General purpose I/O error: cause unknown.
-43 File not found: pathname is mispelled or includes a space.

Error handlers can be written using "abort". Type a message before calling "abort" as in this example:

```
: BASIC ( -- ) ( demonstrate user error message )
  here 20 expect  ( get a line of text )
  ." Syntax Error" abort  ( report error and abort )
  beep ;  ( NOTE: this never happens )
```

Errors reported by some other Forth systems are not reported by Pocket Forth. Compiler or branching security is not provided, nor is free space checked before compiling. Multiple definitions of the same word are allowed. The lack of these messages can sometimes cause mysterious problems.

If the dictionary runs out of free space while compiling, compilation continues on, right into whatever is right after the DICT block in the heap. If you are lucky, its only a free block so no harm is done. From experience it is more likely some vital part of the operating system. Prevent this problem by using grow before compiling long programs. See the first few lines of the Sieve example file.

Toolbox

The Macintosh toolbox is a collection of hundreds of routines within the ROM and System file that create the distinctive 'Mac' interface present in most programs. Pocket Forth encourages your use of toolbox routines in your programs.

Toolbox routines are called by executing a 16 bit instruction called a trap word. Each routine has its own trap word that executes and returns as if it had been called as a subroutine. Compile trap words inline into Pocket Forth code.

A word ",$" (pronounced “comma hex”) compiles a 16 bit hexadecimal number into the dictionary from the input stream. If the token after “comma hex” is not a hex number, a number conversion error is signaled. The word "," (pronounced “comma”) compiles a number into the dictionary from a value on the stack instead of the input stream.

Most toolbox arguments are passed on the system stack. Because Pocket Forth's return stack is the system stack, return stack words can be used to setup for a toolbox call. The words ">r" and "2>r" move 16 and 32 bit values to the return stack. "R>" and "2r>" move 16 and 32 bit numbers from the return stack to the parameter stack. The word "a>r" performs a double duty: it converts a relative address on the stack to an absolute address which it pushes to the return stack.

To call a toolbox routine, setup the system stack with "a>r", ">r" and "2>r". If the toolbox call is a function, push a zero (single or double, as appropriate) to the return stack ahead of any arguments. Then compile the trap word with “comma hex”. Get function results with "r>" or "2r>". Many examples of toolbox calls are in the example programs.

The operating system keeps information for it's use in low memory global variables. These globals can be accessed using absolute addressing with the words "l@" and "dl@".

Machine Language

Because Pocket Forth uses executable code, it is easy to include your own machine code. Insert it as hex data directly with the word “comma hex”. Machine code and Forth code can be mixed as much as you like. Be sure that all words are executable and end with an RTS instruction or JMP to another word.

Machine code must respect the contents of most of the 680x0's registers. Save and restore registers that are used by Pocket Forth. The following table describes the register use:

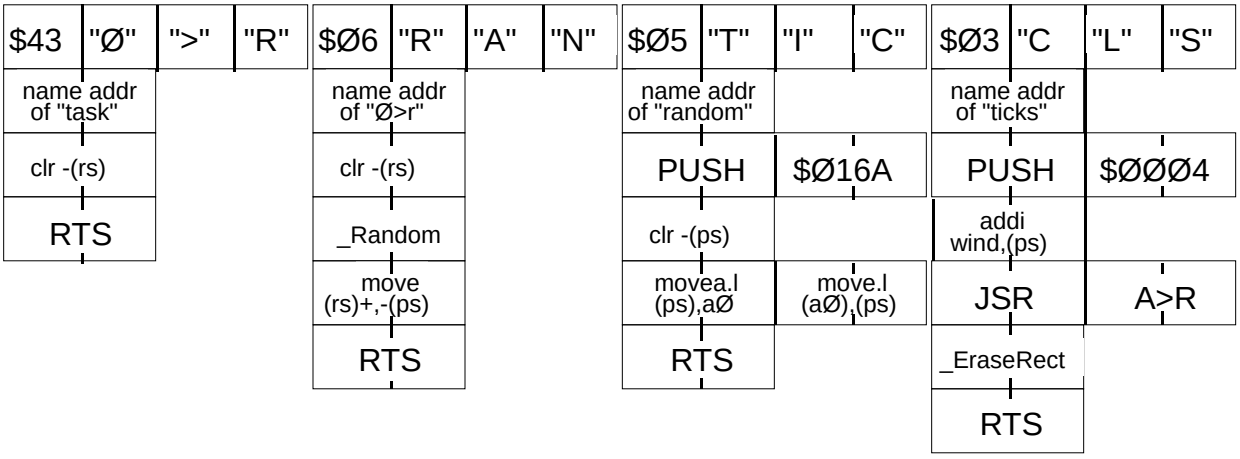
Register	Comments
D0-D1/A0-A1	Used and changed by many words; use freely
D2, D3	Preserved by all words; restore if used
D4	Parameter block absolute address in the DA only
D5	Internal use by the DA only
D6.W	Name address of the last defined word (see "latest")
D7.W	Input character counter
A2 (DP)	Start of free memory absolute address (see "here")
A3 (BP)	The base pointer absolute address (see ">rel" and ">abs")
A4 (IS)	The input stream pointer absolute address
A5	Macintosh operating system use
A6 (PS)	The parameter stack pointer absolute address
A7 (RS)	The return (and system) stack pointer absolute address

The word "mon" executes the trap word _Debugger. If a debugger, such as TMON or MacsBug is installed,it is engaged, otherwise, the system bomb window appears. If you get the bomb, click the “Continue” (or “Resume”) button to quit the application.

Here are examples of toolbox calls, system global variables and machine language:

```
: 0>R ( return stack: -- 0 ) ( push a zero to the return stack )
   ,$ 4267 ; MACRO  ( clr -[rs] )
: RANDOM ( -- u ) ( u is a random number, 0 to 65535 )
   0>r ,$ A861  r> ;  ( _Random )
: TICKS ( -- d ) ( double number = ticks since startup )
   362 0 dl@ ;  ( fetch double from absolute address 362 )
: CLS ( -- ) ( clear the window, leaving the cursor alone )
   4 +md a>r ,$ A8A3 ;  ( _EraseRect the window content rect )
```

And the memory used for these words:



Events

Macintosh programs interact with users and other programs by means of 'events'. For example, clicking on a partially hidden window generates mouse down, mouse up, activate, deactivate and update events in order to bring the window to the front. Pocket Forth handles events internally so you need only deal with those that are pertinent to your program.

A table of unnamed variables in the dictionary contains the relative addresses of event handlers, that is, routines that handle events. Access them with the word "+md", which adds the address of the table to an offset on the stack. The names listed are not in the dictionary, but are shown for clarity.

High level events are new to the System 7 operating system. For use in Pocket Forth, high level events are considered to be any event with an event number greater than 16. Actual high level events have an event number of 23.

Some of the variables in the table hold other important data instead of handlers. Here is a list of the available variables (These names can be added to Pocket Forth by loading the file “Names”):

Name	Offset	Name	(type)	Offset
Window	0	MyID	(DA only)	30
WRect	4	Keydown	(DA only)	32
WSize	8	Cursor	(DA only)	34
Activate	12			
Update	14	AppleMenu	(Appl. only)	30
Button	16	FileMenu	(Appl. only)	34
Menus	18	EditMenu	(Appl. only)	38
Idle	20			
Close	22	Events	(Both)	116
Version	24	EventRecord	(Appl. only)	148
Start	26	WhichWindow	(Appl. only)	164
Echo	28	Clicks	(Appl. only)	168
		MFlag	(Appl. only)	186
		HLEvent	(Appl. only)	188
File Stack Components				
		FStack	(Both)	54
OText	42	FSOffsets	(Both)	74
EText	46	FSEnds	(Both)	94
HText	50	FSPointer	(Both)	114

The first group of variables are those in common to both types.

- **Window** contains a pointer to Pocket Forth's window. Get the absolute address of the window's data structure with "0 +md 2@".

- **WRect** is the content rect of the window in local coordinates and **WSize** is the last four bytes of the rect, the width and height of the window. The values in wrect are used by "cr" and "page" and the window update and button routines.

- **Activate** expects a flag on the stack, true for window on, false for window off. The default handler is the word "drop". Your substitute handler should also take a number off the stack.

- **Update** draws invalidated portions of the window if automatically invalidated by uncovering it or explicitly invalidated by a program. The default handler draws an underscore at the current text cursor position, and has no stack effect.

- **Button** executes when the mouse button is pressed inside the window. The word "beep" is the default handler. Your handlers can make use of the words "@mouse" and "?button" to create more complex mouse button actions. The button handler has no stack effect.

- **Menus** is a variable holding the relative address of a list of menu lists. See the menu section.

• **Idle** is executed periodically. The application's Idle is executed as often as possible. The DA calls Idle every 30th of a second. The period can be altered by setting the drvrDelay variable in Pocket Forth's DRVR's header with ResEdit. The default Idle handler does nothing.

• **Close** executes in response to a click in the window's close box. The Application's handler quits Pocket Forth by executing the toolbox routine ExitToShell. The DA executes its Close handler after the window has been disposed of by the driver. Don't attempt to output to the DA's window within a Close handler. The DA's default handler does nothing.

• **Version** is a handler that identifies the version of the program running. The DA prints a string and the application displays its About… box from the Apple menu.

• **Start** is executed when the program starts. See the turnkey section.

• **Echo** is a variable that contains a flag. If the flag is true then text is printed when pasting or loading. If it's false then printing isn't done and interpretation is faster.

The next group is unique to either the DA or the application. The DA's variables are:

• **MyID** contains the resource ID of the DICT and other owned resources. See the resource section.

• **Keydown** is a handler that executes when a key is pressed. It contains the text interpreter, so change it carefully if you want to return to Forth. See the turnkey section.

• **Cursor** executes periodically like Idle. It should be used for changing the cursor shape if necessary. By default, Cursor does nothing.

The application's variables are:

• **AppleMenu**, **FileMenu** and **EditMenu** are double variables that hold menu handles.

• **WhichWindow** holds a pointer to the last window clicked on. This is useful if your program creates additional windows.

• **Clicks** is a list of the various window part handlers. They handle desktop, menubar, desk accessory, window content region, drag region, grow box, close box, zoom in and zoom out region clicks.

• **EventRecord** holds the event record data. (See Inside Macintosh.)

• **Events** are present in both the DA and application but most useful to the application. Events is a list of first line handlers for the first 15 event types. These handlers do the behind the scenes houskeeping for Pocket Forth. They execute the handlers described above and do not normally need to be modified. The DA's Events list does not contain true subroutines, so should not be changed.

• **MFlag** is true if a multitasking operating system such as System 7 or Multifinder is in effect. A true mflag indicates that trap WaitNextEvent retrieves events, if false, GetNextEvent is used instead.

• **HLEvent** is a handler that executes when any event greater than number 16 is received. These high level events form the basis of the Inter Application Communication capabilities of System 7. This handler is provided to allow experimentation with these new features of the operating system. The default handler does nothing.

The final variables are file handling storage. These addresses should not be modified if text loading or pasting to the interpreter is needed. If not, the file stack handles, offsets, and ends can provide extra storage. The addresses are the same in both types.

As mentioned above, none of these names are in the dictionary, but they can be defined as constants. Here are examples of the use of these variables:

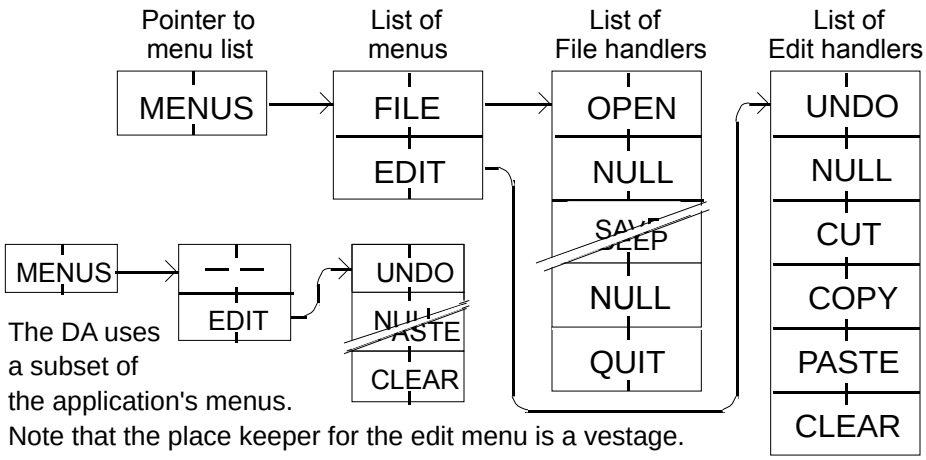
```
12 +md constant ACTIVATE
14 +md constant UPDATE
16 +md constant BUTTON
24 +md constant VERSION

: MYACT ( flag -- ) IF ." Hello again!" cr THEN ;  ( type the string )
: MYUPD ( -- ) 4 0 DO 8 emit LOOP .ok  ( back up and issue prompt )
   [ update @ compile ] ;  ( call the original update handler )
: MYBUT ( -- ) version @ execute ;  ( show version )

' myact activate !  ( set activate to address of "myact" )
' myupd update !  ( set 'update' to address of "myupd" )
' mybut button !  ( set 'button' to address of "mybut" )
```

Menus

Menus can be created and changed by your programs. The application comes with two menus already setup, and more can be added. The DA has the top of the Edit menu and an empty menu available.

Menu items have handlers, arranged in a list to correspond to their appearance in the menu. The menu handler lists themselves are kept in a list that looks like the menubar. This list of menu lists is pointed to by the Menus unnamed variable (18 +md).

The DA's menu structure is a subset of the application's. The Edit menu pointer is in the same position, but the the empty menu slot of the DA is where the application's File menu pointer is. Perhaps this picture of the menu handler structure will clear things up:



The Apple (□) menu of the application is not included because the only changeable item is About… which can be accessed via the Versions unnamed variable.

To alter an item's handler, change the value of the handler in the menu's list to point to the word you want. Change an address in the list of menus to point to a new list to change an entire menu's list. This example sets the handler of the Clear item to the word "page":

```
18 +md @  2+ @  constant EMENU  ( address of the edit menu list )
' page  emenu 10 +  !  ( set the clear item to "page" )
```

Manipulate the physical menus with the toolbox menu routines. Create new menus from resources or from 'scratch' with definition strings.

Menu ID numbers apply to all of the application's menus and the DA's extra menu if one has been created. The application menus are numbered 2 for the File menu and 3 for the Edit menu. Give new menus number 4 and higher if any are added. Handles for the existing application menus are in the unnamed variables as shown in the table.

The DA's additional menu is ID number -1. Create a double variable to hold its handle. The Edit menu is not numbered by the DA because selections from it are handled as discrete messages. There is no Edit menu handle since it is owned by the host application rather than the DA.

Resources

Both types of Pocket Forth, the application and the DA, are composed entirely of resources. Here is a list of all of the resources in each file:

	Application	
actb	257,258	alert color tables
ALRT	257,258	alert boxes
BNDL	128	(finder icons)
CODE	0,1	launching code
DICT	128	dictionary code
DITL	257,258	dialog items list
FREF	128	(finder icons)
hdlg*	257	about dialog help balloon
hfdr*	-5696	finder icon help balloon
hmnu*	1,2,3	menu help balloons
hrct*	128	window help balloon
hwin*	128	window help balloon
icl4*	128	16 color icon
icl8*	128	256 color icon
ICN#	128	black and white icon
ics#*	128	small black and white icon
ics4*	128	small 16 color icon
ics8*	128	small 256 color icon
MENU	1,2,3	what it says
P4TH	128	signature
PICT	128,129	for about dialog
SIZE	1	for multitasking
STR#*	1	for help balloons
WIND	128	main window

	Desk Accessory	
BNDL*	-15552	(finder icons)
dast*	-15552	about box string
DICT	-15552	dictionary code
DRVR	26	driver code
FREF*	-15552	(finder icons)
fwst*	-16564	suitcase use only
icl4*	-15552	4 bit icon
icl8*	-15552	8 bit icon
ICN#*	-15552	1 bit icon
ics#*	-15552	small icon
ics4*	-15552	small 4b icon
ics8*	-15552	small 256 color icon
movp*	0	suitcase use only
P4th	-15552	signature
WIND	-15552	main window

* used by System 7 only.

You can modify many of these resources as well as add new resources of your own. With system software prior to version 7, desk accessories are installed into the system or suitcase files. During the installation process the numbers of resources may change. Because of this, desk accessories own certain resource ID numbers so that installed DAs can find their resources. Each DA is allowed to own up to16 resources of each type. The numbers owned are derived from a formula: OwnedID = -16384 + (DRVR ID * 32) + index where the index is 0 to 15.

Pocket Forth programs need not calculate owned resources this way. An unnamed variable, called MyID (30 +md), has the ID of the first owned resource. Get additional resource IDs by adding an index of 1 through 15 to the owned ID.

To use resources in a program, create and install them into a copy of Pocket Forth. In desk accessories the first resource of each type should be numbered the same as the DICT resource in the file. Additional resources of the same type are that number plus one, plus two, etc. Use ResEdit or an equivalent program to do the resource surgery. Then, if using an older system, install the DA with the Font/DA Mover. Your resources will be moved along with Pocket Forth's original ones. Resources marked for use with System 7 only will not be moved by the Font/DA Mover.

The application (and the DA under System 7) can also access resources. Application resources can be any number from 128 to 32767. Desk accessory resources should be numbered as above for maximum compatibility. Put the resources into the application's file or use a separate file to hold them. These resources will not ever be renumbered, so call them with the number you gave them.

This example gets a 'P4th' resource (from the desk accessory only) and prints it:

```
hex 5034 7468 2constant "P4th"  decimal  ( the resource type )
30 +md @ constant MYID  ( the resource ID  ***DA only*** )
: .P4th ( -- ) ( print the P4th resource data )
  0 0 2>r  "p4th" 2>r  myid >r  ,$ A9A0  ( _GetResource )
  2r>  2dup dl@ 2>r ,$ A884  ( _DrawString )
  2>r ,$ A9A3 ;  ( _ReleaseResource )
```

And this example displays the picture from the application's about box:

```
: .PICT ( -- ) ( show the PICT from "About Pocket Forth..." )
  0 0 2>r  129 >r  ,$ A9BC  ( _GetPicture: PICT id 129)
  2r> 2dup 2>r 4 +md a>r ,$ A8F6  ( _DrawPicture )
  2>r ,$ A8F5 ;  ( _KillPicture )
```

There are of course many more things that can be done with resources. The DICT resource of Pocket Forth is an example of a new type of resource that has its own internal structure. A resource was used for the dictionary instead of a data block because of the ease in saving and modifying the data within it. This does not come without a compromise however. Resources are limited to 32K in size, thus the dictionary size cannot reach the full 64K allowed by a 16 bit relative address. Be sure not to "grow" the dictionary beyond 32768 bytes.

Turnkey

In the Forth world, "turnkey" means to produce a stand alone program from the interpreter. Because most commercial Forth interpreters are proprietary, a side effect of turnkeying is that the interpreter is sealed off, and no longer available to the program's user. Pocket Forth can turnkey applications and desk accessories with or without this side effect.

The essence of the turnkey process is the assignment of new values to Pocket Forth's event and message handlers. Define a word that does what you want, then place the address of the word in the unnamed variable used by the particular event or message. Use "save" to make the changes a permanent part of the program once it is debugged.

The specifics of some event and message handlers are done differently by the application and DA because of their operating environment differences.

DA Turnkey

Handlers for button down, key down, update and activate events and open, close, idle, cursor, Edit and other menu messages exist. Button,activate and update events, idle, cursor and menu messages should be set as described in the events and menu sections.

Key down handlers are unique to the DA version. The key data is found on the stack when the handler is run. The low byte of the key data is the ASCII value of the key pressed and the high byte is the key code. Typically, the key down handler begins with "255 AND" to mask out the key code, leaving the ASCII code on the stack. Replace the normal key down handler to seal off the interpreter. This is desirable if access to Forth isn't needed.

It is possible to write a key down handler that passes key data through to the interpreter after you are done with it. This example implements a tab key that emits four spaces, but runs the interpreter normally otherwise. Get the address of the interpreter's key handle, it's in the handler variable at startup:

```
32 +MD @ CONSTANT IKEY  ( interpreter's key routine address )
```

Next, define a new word that JMP's to the interpreter routine if the key isn't the tab key:

```
: W/TABS ( n -- ) ( the interpret routine with tabs )
   255 AND  ( get ASCII )
   DUP 9 = IF DROP SPACE SPACE SPACE SPACE  ( drop code, emit spaces )
   ELSE  ,$ 4EEB [ ikey , ]  THEN ;  ( jmp ikey[bp] )
```

Finally install this word into the key handler variable:

```
' w/tabs 32 +md !
```

Auto key events are handled by the key down handler. See below for how to mask out unwanted events if you do not want to handle auto key events as key down events.

Open and close handlers work in tandem to create and destroy transient data structures such as handles, menus, windows, etc. The open message happens at the beginning, right after the DA is initialized. Open message handlers should be written with some limitations in mind. Particularly, the driver's control routine has not been entered yet, meaning that events and messages cannot be received nor is the parameter block of the DA accessible. Because of this, the open handler should not try to use events.

Memory structures on the heap, created in the open routine or the body of the program, should eventually be destroyed or they will accumulate in the heap. The usual way to remove heap items is to call the appropriate ROM routine during the close handler. For example, if a handle is created in the open routine, dispose of it in the close routine.

The following example produces a turnkey DA. Don't actually save it because it doesn't do much.

```
( Handler demo DA )
: MYACT ( flag -- ) IF ." Activated" ELSE ." Deactivated" THEN CR ;
: MYUP ( -- ) ." Updated" CR ;
: MYBUTT ( -- ) ." Mouse button pressed" CR ;
: MYKEY ( n -- ) 255 AND EMIT SPACE ." Key pressed" CR ;
: MYIDLE ( -- ) ." Idle" CR ;
: MYCURS ( -- ) ." Cursor" CR ;
: MYCLOSE ( -- ) BEEP ;  ( DON'T output to the window, it ain't there!)
: MYOPEN ( -- ) ." Hi There!" CR ;
: START ( -- ) ( set it all up )
   [ ' MYOPEN LITERAL ]  26 +MD !  ( set the open handler )
   [ ' MYCLOSE LITERAL ] 22 +MD !  ( set the close handler )
   [ ' MYCURS LITERAL ]  34 +MD !  ( set the cursor handler )
   [ ' MYIDLE LITERAL ]  20 +MD !  ( set the idle handler )
   [ ' MYKEY LITERAL ]   32 +MD !  ( set the key down handler )
   [ ' MYBUTT LITERAL ]  16 +MD !  ( set the button handler )
   [ ' MYUP LITERAL ]    14 +MD !  ( set the update handler )
   [ ' MYACT LITERAL ]   12 +MD ! ;  ( set the activate handler )
START ( SAVE )  ( DON'T save unless you really want this )
```

As you can see by running this demo, idle and cursor events overwhelm the other events, so that your DA will be called often enough. If idle messages or other events are not required, they can be ignored by modifying the Pocket Forth DRVR with ResEdit.

The rate the idle handler is called is initially set to run 30 times a second. This can be changed with ResEdit. The value in the drvrDelay field represents the time between idle messages in sixtieths of a second. A value of zero lets the idle routine run as often as possible. The drvrFlags field of the DRVR's header is normally $6400. A value of $4400 causes idle messages to be ignored.

The drvrEMask field contains the event mask for the DA (see Inside Macintosh, the Event Manager, for details on constructing an event mask.) Only events that are normally enabled are handled by Pocket Forth, even if the event mask is changed to enable other events. If the event mask is changed to ignore some normally handled events, those events will be ignored.

NOTE: The first four fields of the DRVR's header are the only part of the DRVR that should ever be changed. Modifying the routine addresses will cause an error.

Application Turnkey

Button, activate and update events are handled by the application just like to their DA counterparts. Idle, open and close messages work differently and key and cursor messages don't exist.

Idle messages occur as often as possible. If a slower idle routine is needed, check the time each time through and act only if enough time has passed. Handle cursor changes in the idle routine.

The open handler is executed after everything is setup and just before the interpreter loop is entered. Unlike the DA, events are enabled at the time the open handler runs. Seal off the interpreter by entering an endless event loop. Keep the text interpreter active by returning from the open handler.

The close message is sent when the close box of the window is clicked or when 'Quit' is selected from the File menu. The default close handler calls the ROM routine _ExitToShell which quits the application; the handler causes the application to quit. This is a major difference from the DA where the close handler reacts to an already quitting program.

Key down events are not ignored, rather the application is centered around getting and handling key presses. The main loop of the program is the key down handler.

The words "key" and "?terminal" react to all events. Event loops look like:

```
        ... BEGIN  KEY .. handle key data .. AGAIN ...  ( endless event loop )
or      ... BEGIN ... ?terminal UNTIL ...  ( a one shot event loop )
```

Run non-key events in their handler routines as in the DA. Key events are handled in the event loop code. Here is the turnkey example, coded for the application:

```
( Handler demo application )
variable TLAST  0 tlast !
: TICKS ( -- n ) 364 0 l@ ;  ( low word of 'ticks' system variable )
: ?1SEC ( -- flag ) ( true if >1 second has elapsed since last time )
   ticks tlast @ - ABS 60 > ;  ( is time difference > 1 sec? )

: MYACT ( flag -- ) IF ." Activated" ELSE ." Deactivated" THEN CR ;
: MYUP ( -- ) ." Updated" CR ;
: MYBUTT ( -- ) ." Mouse button pressed" CR ;
: MYIDLE ( -- ) ?1sec IF ticks tlast !  ( update tlast )
     ." Idle" CR  THEN ;  ( wait 1 second between idle messages )
: MYKEY ( c -- ) EMIT SPACE ." key pressed" CR ;
: MYCLOSE ( -- ) ." Closed" CR ,$ A9F4 ;  ( _ExitToShell )
: MYOPEN ( -- ) ." Hi There!" CR
   BEGIN KEY  ( "key" waits for a key press and runs event handlers )
     MYKEY  AGAIN ;  ( Loop through the key handler indefinitely )
: START ( -- ) ( set the handlers )
   [ ' myopen LITERAL ] 26 +MD !  ( set the open handler )
   [ ' myclose LITERAL ] 22 +MD !  ( set the close handler )
   [ ' myidle LITERAL ] 20 +MD !  ( set the idle handler )
   [ ' mybutt LITERAL ] 16 +MD !  ( set the button handler same as DA )
   [ ' myup LITERAL ] 14 +MD !  ( set the update handler same as DA )
   [ ' myact LITERAL ] 12 +MD ! ;  ( set activate handler same as DA )
START ( SAVE ) ( DON'T save unless you really want this program )
MYOPEN  ( simulate opening as if this had been saved )
```

Once a turnkey program is finished, debugged, and all variables are initialized, type "save" to replace the current dictionary stored on the disk with the new one. Then use ResEdit to change the resources such as the window title and the application's icon (as well as BNDL, FREF and signature resource) and About… item. Be sure to "save" only copies of Pocket Forth since the changes are permanent.

The desk accessory "Bezier" (available separately) and the application "Read Me" are examples of full scale turnkey programs.

Versions

The Pocket Forth version number is a two part number written as: T.V 'T' is the program type and 'V' is the version within that type. Type 0 is the application with a sixteen bit stack and subroutine threading. Type 1 has the attributes of type 0 but is a desk accessory. The current versions are 0.5 for the application and 1.5 for the DA.

The current versions of types 0 and 1 are nominally compatible. That is, most programs written for one will run on the other. This extends to the words in the dictionary, and the behavior of most of those words. The offset of most unnamed variables are the same in each type. The actual addresses of all of the words differ from type 0 to 1, so using numeric addresses will causes problems. Other differences are described in the appropriate section of this document.

Write to the author, Chris Heilman, on CompuServe at [70566,1474], Bitnet (heilman@pc) or through the US Mail to my personal zip code: 85066-8345.

Bibliographic Notes

(1) Barnhart, Joe, "FORTH and the Motorola 68000", Dr. Dobbs Journal no. 83, September 1983, Peoples Computer Company.

(2) Greene, Ronald, "Faster Forth", Byte v.9 no.6, June 1984, McGraw Hill Publishing Co.

(3) Loeliger, R. G. Threaded Interpretive Languages, Byte Books, 1981.

(4) Fletcher, G. Yates, "A Mini Forth for the 68000", Dr. Dobbs Journal no.123, January 1987, M&T Publishing, Inc. This article presents Flint (Forth-like interpreter), a concisely implemented Forth interpreter in 68000 assembly language. Pocket Forth owes much of it's structure and philosophy to Flint, and I thank Professor Fletcher and Dr. Dobbs Journal for publishing Flint .

(5) Ragsdale, William F., *et al.* fig-FORTH Installation Manual, release 1, Nov. 1980, Forth Interest Group. This includes a model of Forthin Forth, hex dumps and source code for figFORTH for the 68000.

(6) Derick, Mitch and Linda Baker FORTH Encyclopedia, Second Edition, Mountain View Press, Inc. 1982. The FORTH Encyclopedia shows the definitions of fig-FORTH words.

(7) Brodie, Leo Starting FORTH, FORTH, Inc. / Prentice Hall, 1981. The FORTH instruction manual and fun to read. You can't learn Forth without this book.

(8) Chavez, Lori, "A Fast Forth for the 68000", Dr. Dobbs Journal no. 132, October 1987, M&T Publishing, Inc. This article describes how to implement macros.

(9) Knaster, Scott How to write Macintosh Software, Hayden Book Co., 1986

(10) Apple Computer Inc. Inside Macintosh, volume I through volume VI, Addison Wesley, 1985-1991