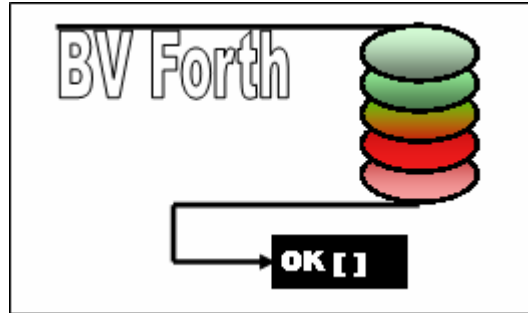


BV Forth (ARM) Language Guide



ByVac

©ByVac 2007

www.byvac.co.uk

Revision 1.0

Copyright in this work is vested in ByVac and the document is issued in confidence for the purpose only for which it is supplied. It must not be introduced in whole or in part or used for tendering or manufacturing purposes except under an agreement or with the consent in writing of ByVac and then only on condition that this notice is included in any such reproduction.

© Copyright ByVac 2007

No warranty

THE WORK IS PROVIDED "AS IS," AND COMES WITH ABSOLUTELY NO WARRANTY, EXPRESS OR IMPLIED, TO THE EXTENT PERMITTED BY APPLICABLE LAW, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Disclaimer of liability

IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS WORK, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

1. Preface

The operating system used on the BV511 microcontroller is Forth, invented in 1968 by Charles H Moore <http://www.colorforth.com/bio.html>, The language has had a strange career, full of promises but quickly overtaken by other languages such as BASIC and C. There still, may be books available about the language but for all intents and purposes it is not used that much for developing applications.

So why bother with a language that is not used very much anymore? Well, for this application, learning and using hardware with a microcontroller, there is no other language that comes close to the ease of use and flexibility. Sure once an application has been developed and the idiosyncrasies of the particular hardware / software interaction have been understood, then use another language. Particularly if this is for mass production. This is the key point about this language. it is so easy to interact with hardware that by experiment the particular hardware can be fully understood. This would take much longer using conventional compiled languages such as C.

Forth is not ideal for large collaborative programs, that is not to say that it has not been used for such projects. The Fig Forth site <http://www.forth.org/> has some fine examples of successful projects that have used the language.

If the above sounds pessimistic its not, just realistic, using the language makes a big difference to the approach of a project and brings with it a very bottom up design philosophy where each part of the system is constructed, tested and brought together to eventually form a whole. Using a wholly bottom up approach is probably as good an idea as using a wholly top down approach. The ideal is a mix of both and there is no reason why forth cannot be used this way.

There are 4 major advantages to using Forth on the BV511. 1. understanding the component parts of a system and the interaction between the software and hardware is essential Forth provides a very easy way of doing this. 2. The system is fully developed on board and so therefore does not need any other tools to use or learn, it can be used wherever a PC with a USB is available. 3. Once developed the system as-is can be implemented. 4. In-system upgrades can easily be carried out.

2. Acknowledgements

The 20 Minute Micro (BV511), so called because of the claim that it only takes 20 minutes from opening the box to getting a stand alone application running, uses a custom designed Forth, built from scratch. A great deal of insight was gleaned reading the work by Bradford J. Rodriguez in his paper Moving Forth: a series on writing Forth kernels. The full text of this can be found on his web site <http://www.zetetics.com/bj/papers/>. That particular text is about 8 bit Kernels but the techniques on building Forth still apply. Another source of information is the work done by Frank Sergeant, <http://pygmy.utoh.org/> There is an ARM Forth on this web site that is well documented and is a valuable source of information.

Although nothing to do with Forth Peter Cockerell <http://www.peter-cockerell.net/> has a very good introduction to the ARM processor, biased towards the old BBC computer but the assembly code was a great help when starting with this Forth.

** At the time of writing the web addresses above were valid, but as with most web sites they tend to come and go.

3. Introduction

The Forth described here has been built from scratch especially for the ARM microcontroller and the Philips LPC21xx in particular. It is written wholly in ARM assembly and high level Forth. The underlying philosophy has been to keep the system as simple as possible. This is in stark contrast to the C offerings with the requirements of a complete tool set in order to get anything working.

A library of Forth code is available for driving particular aspects of the LPC21xx, these are both a learning resource and a source for implementing hardware quickly. As an example of this there is an LCD.FTH file. If a standard LCD display module is connected to the ARM it is a simple matter of loading the file and typing in text to see it appear on the display. There is no need to learn how the display works in order to get it going, this can come later when using the display.

One of the main overriding advantages of this system is the ability to 'experiment' with hardware without the need for the compile, assemble, link, load cycle. To set a particular register to a value you simply type the command. For example to set the U1DLL register that holds the baud rate divisor for UART1 to 5 simply type 5 U1DLL !. The effects will be immediate, looking at register values is just as simple.

The aim of this guide is to concentrate on this Forth system and how it works. This is a reference rather than a tutorial. The other documentations in this are:

- BV511 hardware Guile
- BV Core Glossary
- BV ARM Microcontroller Foundation

The features of this system are:

- A self contained operating system
- No specialist tools needed, just a terminal emulator
- The system can be easily extended by user input

- Can create stand alone systems very quickly
- Updates can be downloaded and flashed to the system using free tools

It is described as a system because it is not just the language that offers the features but the language combined with the LPC21xx series of microcontrollers.

4. Start Up

From reset UART0 is initialised to 115200 Baud and UART1 to 9600 Baud, these can be altered later by changing the register values. The system variables are then copied from Flash to RAM and the Forth word COLD is run. The prompt is then displayed ready for user input.

4.1. Last Word

One of the actions of COLD is to establish the last word in the system. When a new word is created the Forth system needs to know the previously defined word so that it can link back to it. On a system that has just started (at reset) no new words have been defined so LASTW is used as the latest created word and the first new word can link to this.

On a fresh Forth system there are certain variables that are pre-defined, in particular the variable pointer, the free area of Flash to write to and the word that will run at start up (COLD in a new system). If the system is not to be extended then these can be defined as constants in assembler and compiled into the code. However on this system there is an opportunity to save a new system so where is the best place to put the variables in Flash? For those new to the Flash memory system on the Philips processor may not realise that it must be written to, a block at a time and before writing it must be erased. One scheme could be to reserve a block of Flash specifically for this purpose but even on this system where simplicity has been compromised for efficiency an 8k block of Flash for 3 variables is going a bit too far.

The scheme used is to keep the three variables tagged on to the back of last word. COLD which is executed on reset will search for LASTW, pick up the variables and store them in the system area in RAM. The three variables that are kept track of are:

Last word counter: This keeps track of how many last words there are, the system does not currently use this for anything.

VP: This is a copy of VP and maintains the variable offset, without this it would not be possible to maintain a variable definition from one save to the next.

FFS: Free Flash start, this holds the start of the next available free Flash so that when the save command is used, the new section of dictionary is stored at this location.

Autorun: There is an option with the SAVE command to save a word along with it. This word will be run when the system is initialised. If the value is 0 autorun is disabled, in other words the normal default start up is used.

Part of the start up prompt contains this information:

Ltst Wrd	Vble off	LatestID	FFS
00005C78	00000000	00000004	00005E00

All of the values are in hex. The first value is the location, in Flash of the last word, this will increase each time SAVE is used. It will give an indication of how full the flash is. On the LPC2132 the maximum value will be &FFFF. The next value is the variable offset value 00000000 indicates that no variables have been defined. The next value currently set to 00000004 means that the system has been saved 3 times, there will be in fact 4 x LASTW in the system. The last value is the Free Flash Start, this is the address to where the new system will be saved.

5. The Stack Prompt

The normal forth prompt `'ok'` in this system has been extended to show the stack contents within the square brackets `'[]'`. This is a great help for new and experienced programmers alike. It is also a valuable learning tool for getting used to how the stack operates.

6. Loading Forth Files

Source code? Because Forth is interactive, interpreted and compiled on the machine it is possible to write forth words without any files, simply type directly into the system.

It cannot be stressed how useful this is for examining memory contents or registers and generally experimenting with hardware, but for writing words it becomes very tedious and worst of all it is not possible to go back and see what has been done (write only). There is a word DECOMPILE (needs to be loaded from decompile.fth) that shows the contents of a single word but that is really intended for debugging single words, not whole programs.

Forth files (source code) in this system have a FTH extension, these are simply

standard text files that can be created with Notepad or a program editor (PSPad recommended). To load a Forth file into the system it is simply a matter of sending a text file, this emulates a very fast typist. There is however one problem with this and that is at the end of each line Forth attempts to compile the line into the dictionary, this takes a small amount of time. If the file is sent too quickly then errors will occur because the BV511 has not had time to complete the compilation. The problem can be overcome by adding a delay to the end of each line. The delay of course must cater for the worst case, but there is a better way.

BV Forth has a word called LOADF that uses a very simple protocol for handshaking with the device that is sending the text file. Handshaking here means telling the device when to send and when to pause.

The protocol works like this:

1. Terminal sends a line of text and then waits for ACK (ASCII 6)**
2. Forth receives line of text and at CR (carriage return) compiles the line into the dictionary, when finished it sends ACK

This very simple protocol works extremely well and does not require any special attention by the terminal emulator, other than to wait for the character 6. Some terminal emulators are capable of doing this, the old Microsoft Terminal supported it. BV Terminal of course supports it and in addition will also send the initiating word, LOADF.

** ACK

By default the value of the ACK is 6 but it can be changed to any value (0-255) by changing the system variable ACK, for example 65 ACK ! will change the ACK to 65. To see the current value use ACK @.

There are two words that enable this protocol: LOADF and ;S. LOADF simply tells Forth to send ACK at the end of each line and ;S turns it off. BV Terminal will send LOADF automatically, with other terminals LOADF will need to be typed in manually before sending the file. It follows then that all .FTH files need to end with ;S.

The file loading process also keeps track of the line numbers and any errors that may occur during the transfer:

```

ByVac Terminal -- www.byvac.com
File Settings CPU About
10|
COM3 115200 2 8
■[&2A ] drop
■[&2B ] ;
ok [] ■[&2C ]
ok [] ■[&2D ] ;s
**** There was an error loading this file at
line 11
Free ram space 14436
ok [] |
H:\Zed\ASCII-Me\ARM\Forth-Code\x.fth Complete

```

Figure 1 Error during transfer

If the file transfer has been unsuccessful then it will be shown at the end of the transfer. This will be due to a malformed word or a reference to a word that does not exist rather than an error in the actual transfer. The error mechanism can only hold one error so only the first error will be seen, if there are several errors in the file then each will need to be fixed in turn.

Another message that may be seen is "At least one duplicate word found at line nn". This is not an error but just a warning that you have used that word before and any previous word of the same name will be overwritten.

Duplicate Words

Words are not overwritten when a duplicate name is used although the effect is the same. As the dictionary is searched backwards and once a word is found the search stops, it follows that the most recent word will be found first and so it will be this 'new' word that is used, effectively hiding the older word.

As a file is loaded the stack is shown in the first [square] brackets and the line number is shown in the second (round) brackets.

7. Save & New

Unique to this system is the ability to write Forth code in RAM and then to save these words to Flash, thus extending the system. There are just two words that provide this mechanism and they are SAVE and NEW described below.

7.1. NEW

New, resets the system to its base state, it does this by erasing everything in Flash above the core Forth system. It also fills RAM with 00, the RAM is erased from the end of RAM to the Variables stack. The variable stack being currently the top of the dictionary.

7.2. SAVE

This is a complex word that has to take care of many things. The action of the word is to save a newly created system into Flash for subsequent use. All of the Flash is erased with the word NEW and so no erase process needs to take place. This means that SAVE can be used many times until the Flash is full and when full it can then be erased with NEW.

An option of save is to follow the word save with another word, e.g. SAVE GO. The word GO will be executed at start up after a reset or cold. The code field address (CFA) for this is stored as part of LASTW. A word of warning here, if GO is a word that causes the system to crash then the only recovery is to re-program the Flash with a new Forth system. This is easily carried out with the free Phillips utility, see the relevant section of the hardware guide.

7.3. Table Constants using CREATE

There are implication when saving tables and variables, for more information on this see section 16.5.

8. System Architecture

BV Forth (ARM) is designed to run on the ARM microcontroller. This is a particularly good microcontroller for this application. The main reason is that Forth is really designed to run and compile in RAM. Although it will quite happily run in Flash, it is not possible to extend the system without RAM. The ARM has Flash and RAM on a single chip making a low cost, efficient Forth system possible. To add to the usability the Flash is programmable in-system, this means that any code

©ByVac

developed in RAM can be transferred to Flash without the need of any external tools. By doing this the operating system is extended for individual purposes and optionally, incremental.

The ARM is also 32bit and this Forth takes advantage of that. What this means is that it makes integer (fast) arithmetic useful with numbers in the 0 to 4,000,000,000 plus range. This is a 'real word' number range unlike the 16 bit equivalent: 0 to 65,000 plus. The 16 bit integer quite often did not have enough digits for real applications, this meant using double words which in turn would lead to slowing down the application. Floating point is an option but the same slowing down applies.

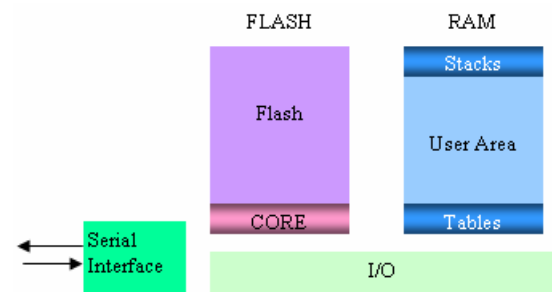


Figure 2 System Architecture

The Forth core sits at the bottom of Flash and takes up typically less than 20k. This is the operating system and takes care of all of the basic functions. The serial interface is essential to the operation of Forth although once a system has been created its use is optional. The serial interface is also used to re-program the flash with a new core system if required.

The RAM has essentially 3 areas, at the start of RAM there are a few tables that take care of interrupts and tasking, at the end of RAM are the stacks. This also contains any variables that may have been created during the course of programming. At start up the system operates from Flash, any words already defined or in the core, when invoked will run from Flash.

When a new word is created this is created in RAM and linked to Flash, the user builds up words in RAM. Words can be downloaded from a text file and at some point in time those words may need to be 'permanently' stored in Flash. This is carried out at any time using the SAVE command (WORD). Save moves all of the words in RAM to Flash and restarts the system. Save can be used over and over again until the Flash is full at which point the NEW word can be used which will erase the Flash, excluding the CORE.

9. Memory Layout

The most complex part of the memory map are the various stacks used for Forth, for this system a stack is implemented to grow downwards from high memory to low and buffers are implemented to go from low memory to high. The dictionary grows from LASTW upwards so this would be described as a buffer.

The following RAM memory map is used in this system:

Top of RAM		
< 128 bytes reserved for In App Programming >		
< sys >	buffer	buffers grow up
< tib >	buffer	
< pad >	buffer	
< return stack >	stack	stacks grow down
< parameter stack >	stack	
< leave stack >	stack	
< variables >	stack	approximately 0x40003b28 (LPC2132) [1]
< dictionary >	buffer	normally 0x40000088 [2]
< interrupts >	n/a	
Bottom of Ram		

As mentioned previously the buffers grow up and stacks grow down. On a freshly reset system [2] can be determined by using the word HERE and [1] by using the word RAMSIZE. Interrupts, consist of a list of addresses that are reserved for the next release.

Sys

This is an important area of ram that is initially loaded from Flash. It is a table that holds all of the system variables. The area is defined at the beginning of the Forth source code and copied up to RAM at start up. Some of the values are changed by COLD after the copy process. Forth uses the variable HERE for start up and to keep track of changes, the Dictionary Pointer for example is in here, this updates each time a new word is created. See section 11 for more details on the system variables.

Tib

Text Input Buffer (TIB) is an area of ram for buffering a line of text entered at the console. It follows from this description that a line is input and then interpreted (compiled), this is an important aspect of how the system works. The tib size is normally 80 characters and when it is not used for text interpretation it can be used for any other purpose.

Pad

This is another buffer that has many miscellaneous uses in various words. It is a useful buffer area for user words.

Return Stack

The return stack is the 'machine' or system stack. In the ARM architecture this stack is not used as much as other machine stacks because of its unusual subroutine calling method, preferring a link register rather than a push / pop instruction (in common with other RISC machines). On this system it is however used for the threading mechanism.

Parameter Stack

This is THE Forth stack that will hold temporary variables and values for passing from one word to another. When typing say 12 at the console this value is stored on the parameter stack.

10. The Dictionary Structure

Forth systems are differentiated by how they store and retrieve words but most Forth's have a dictionary structure that is a linked list of words.

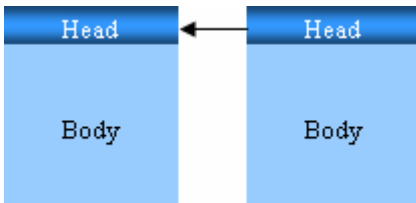


Figure 3 Linked List

The basic unit of Forth is the WORD and each word consists of a head and a body, in order for the words to be found they are organised in a one way linked list. The list begins at the latest (most recently defined) word and each word has a link back to the previous word. The first words link value is 0, indicating that the start of the list (end of the search) has been reached.

This link information is contained in the word head along with other system information.

10.1. The Word Head

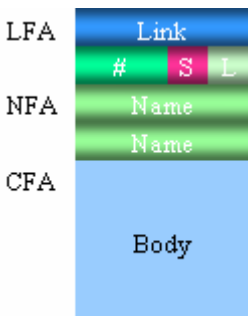


Figure 4 The Word Head

The head is comprised of four fields 32bits in length, this is a total of 16 bytes.

Link Field (32bits)

This is a 32bit word that points back to the next forth word in the dictionary. It is an absolute address that can exist in either RAM or FLASH space. The first word has a link address of 0, this indicates the end of search.

Hash # (16bits)

When a word is created only the first 7 characters are stored as part of the head but also a hash is calculated based on the RS algorithm, the lower 16 bits of this hash are stored here. When a search is performed it is this hash that is used to

determine the word value. This has the advantage that a search is much quicker and easier and if necessary the word itself does not need to be stored. The disadvantage is that the hash needs to be calculated for each new word or search. To avoid a very remote possibility of a 'collision' this is where two words may have the same hash value, the name of the word is also checked by FIND before a match is accepted.

S (8bits)

This is the system byte which is a set of flags or 1 bit values that have the following meaning:

Bit 7: Immediate flag, this word is designated an immediate word when set to 1

Bit 6: Hidden flag, when set to 1 the word will not be found in a search

Bit 5: Type flag 0=Forth word, 1=Code word

Bits 3-0 : always set to 0x9, indicating that this is a system word

L (8bits)

Length: this is the length of the original word, it is not currently used for any system functions but has been included for possible future use.

Name (8 bytes)

This is the name of the word. When a word is created the hash is calculated on the actual word but only the first 7 bytes are stored, the last byte (8th) is always a 0.

From Figure 4 it can be seen that there are three associated addresses as part of the head, these are the Link Field Address (LFA), Name Field Address (NFA) and Code Field Address (CFA). As the head has a fixed structure the addresses can be calculated from each other.

10.2. The Word Body

There are two kinds of words that could exist below the head, a Forth word and a code word. A code word is a set of assembly instructions that will execute ARM instructions directly from the CFA. A Forth word is somewhat different.

A Forth word consists of a list of addresses, an address may point to another Forth word or to a code word. The CFA of a forth word, points to an instruction that jumps to the next address, this may seem a bit bizarre but that is how it works. Although a Forth word can call a code word, a code word cannot call a Forth word, this is an important distinction and it is one of the reasons that the type flag exists.

11. System Variables

There are several variables that are kept by the system, the initial values are loaded from the Flash as constants in to the RAM and then COLD optionally overwrites some of them to reflect the new system state. The system variables can be accessed and set with SYS@ and SYS!. Some are actual values and others are addresses which point to values. System variables are referenced by number, thus 12 SYS@ will fetch the RAM start address.

Ref#	Description
0	Reserved
1	Reserved
2	Reserved
3	Start up word [1]
4	Reserved
5	Parameter stack [1]
6	Return stack [1]
7	Dictionary pointer [1]
8	Pad [1]
9	FFS (see start up)
10	Latest ID [2]
11	Line count [3]
12	RAM start [1]
13	Return stack size [2]
14	Parameter stack size [2]
15	Base (number base, radix)
16	Hold
17	Error Line number [3]

18	State
19	Latest
20	>IN
21	Reserved
22	ACK [2]
23	Seed [2]
24	Variable pointer [2]
25	Dictionary Start in RAM [1]
26	End of Flash [1]
27	Reserved

[1] Addresses

[2] Values

[3] This is used for error checking when loading Forth files, it can be used for any other purpose when not loading files.

Some of the system variables are standard Forth variables but others are unique to this system.

12. Errors and Error Checking

One of the main advantages of using Forth is that it is fast, almost as fast as assembler and not unlike C it will let you do anything you like. The penalty for the speed advantage is that error checking is very minimal.

As an example you can use IF without the corresponding ENDIF and this will be accepted, the only clue is that there will be a value in the stack after compiling the word which would not be expected. To put in checking code would mean a substantial reduction in performance.

13. Variables

In most Forth systems all of the code runs in RAM and therefore a variable can be defined as just another word, because the word is in RAM changing the value does not pose any problems. Words defined in this system are also defined in RAM, however an important aspect of this Forth is that it can be moved to Flash and if this occurs the conventional method of creating a variable will not work as Flash cannot be read and written to in the same way that RAM can. The method used is to create a word conventionally but store the variable part in RAM.

The name is defined using CREATE in the dictionary that can at a later date be saved to Flash. This definition points to an area of ram that is used to store the actual value. V0 points to the start of the variables stack and VP @ holds a value that is used to calculate an offset down the stack for the next free stack location. So, if for example there were 10 integers defined, the contents of VP (VP @) would be 40, as each integer is 4 bytes.

	return values	VP @
V0 --- say	0x40003728	0
int1	0x40003724	4
int2	0x40003720	8
int3	0x4000371C	16

As an integer is defined the value of VP @ is incremented so that the next variable points to the correct place. When a SAVE takes place this value (VP @) is stored in the new LASTW so that subsequent variables will not overwrite the older ones. COLD initialises all of the variables that have been defined to 0.

Note, although this is implemented as a stack (grows down), the variables are created with the intention of growing upwards, in other words the VP @ offset is taken away from V0 BEFORE the creation and it is this LOWER value that the variable returns when executed.

A variable therefore obtains its memory from a stack (grows down) but is used as a buffer (grows up), one or more buffers created form a stack. For more information on buffers see section 16.3 and 16.4.

14. Local Variables & Integer

Traditionally the address is always referenced in Forth variables and the words fetch (@) and store (!) are used for retrieving and updating the values. This is extremely efficient but possibly not as intuitive in a modern language. For this reason the integer variable type and the local variable type have been included in the firmware.

14.1. Integer

The INTEGER word defines a global variable just as the defining word VARIABLE does, it is, however used very differently.

Integer a

This will define an integer 'a' in the variable space as normal. When used on its own it will return its value.

a

the above will return 0. To give 'a' a value the following is used:

120 => a

This will set 'a' to 120.

-20 +> a

This will reduce 'a' by 20 and

%@ a

Will retrieve the address of 'a' so that if necessary it can be used as a standard Forth variable. The three modifiers to the integer are:

=> ` to variable'

+> ` update variable'

%@ ` address of variable'

14.2. Local Variables

Local variables can make life much easier and really should be part of the standard. They elevate mind bending stack juggling and make the code much more readable.

The down side is that they are much less efficient than using the stack, however as processors become faster this is less of a problem.

A local variable needs to be defined and this can be done in one of the following ways:

1. In line with the word using curly brackets thus : **foo { source dest }**
2. In the body of the word using **INT:**

3. In the body of a word using **INT#:**

4. In the body of the word using **V\$:**

The first method will take items from the stack and assign them to the variable names, in the above example `dest` will be assigned the topmost item and `source` will be assigned the next item on the stack.

```
: foo { led colour }
... etc.
```

Dashes can be used following the variables `{ led colour --- 0 |-1 }` to denote the stack after the word has been executed. Anything following the `-` up to the `}` will be ignored.

The second method (`INT:`) allows creation of one or more variables in a list, the variables will be assigned an initial value of 0, thus:

```
: foo
INT: var1 var2
... etc.
```

If anything follows the definitions such as a comment or another word then it must be preceded by a `}`, for example:

```
INT: var1 var2 } ( set var 1 and 2 )
```

Although the `//` type comment does not need the terminating `}`, some examples

```
INT: a b // creates a and b *okay*
INT: a b ( creates a and b ) *error*
INT: a b } ( creates a and b ) *okay*
INT: a b 0 = if 1 else 0 then *error*
INT: a b } 0 = if 1 else 0 then *okay*
```

The third method of creating a variable is to use `INT#:`, only one variable at a time can be created but its initial value can be set. In the example below `'exitval'` is created that has an initial value of -1.

```
: foo
INT#: exitval -1
... etc.
```

The terminating `}` can be used but there is no need as the word only expects two items in the input buffer. The number following the variable must be specified as a number,

an expression such as `12 4 +` will cause an error.

The forth type is for creating local buffer space and assigning a name to that space.

```
: foo
V$: buf 18
... etc.
```

This will create a buffer on the local stack at least 18 bytes big, in fact it is rounded up to the nearest 4 bytes so the space allocated will be 20 bytes in this case. The number following variable must be a number. The current available local space is set to 400 bytes.

The local variables created have a scope only within the word they are used and thus the same name can be used within different words without causing a clash.

They are used differently from global variables in that they return a VALUE and not an address, the modifiers in section 14.1 can be used to store and modify the contents.

```
: perim { x1 y1 x2 y2 --- len}
  x2 x1 - 2 * // top and bottom len
  y2 y1 - 2 * // left and right len
  + // add together
;
```

This example shows how to get the perimeter of a box given two co-ordinates. This would take a lot of stack manipulation to achieve the same results.

```
: dump { addr --- }
INT: basehold
base @ => basehold hex
dumpHead cr
16 for
  addr <# # # # # # # # # > stype space
  addr hex16. addr space ascii16.
  16 +> addr cr
next
basehold base !
addr 256 +
;
```

Here is another example where the base needs temporarily changing to 16 (hex). A variable `'basehold'` is created to hold the old base value. Without the local variables this would have to be held on the stack or in a global variable.

NOTES for Locals

The search order when compiling a word is globals, locals and then numbers. This means that if there is already a word defined it can be used again as a local without confusion.

```
: foo { dump --}
    ." the value on the stack was " dump .
;
```

In this example there is already a word called DUMP that displays the contents of memory but the output from the above will display whatever was on the stack.

The stack usage is only checked at compile time so care must be exercised when deep nesting is used. The stack will only grow when a word using locals calls another word using locals but the space is retrieved when the word returns.

15. Numbers

<# # #	BINARY	S>D	.(dot)
#>	HEX		PD.
#S	DECIMAL		PH.
?NUMBER	BASE		PB.
STYPE			U.
>DIGIT			
HOLD			

For a full list see the core glossary.

The number convention used does not quite follow the Forth standard (ANSI). It has been developed for practical reasons and in particular for use with a microcontroller system.

15.1. Number Storage (type)

All numbers are stored as a 32bit word and are signed, signed means that if bit 31 is a 1 it is a negative number. This means that the greatest negative number is -2,147,483,648 (0x80000000) and the greatest positive number is 2,147,483,647 (0x7FFFFFFF). The numbers can also be treated as unsigned that will effectively double the range available.

Some words require a double (64bit) number, M* for example that will multiply

two 64 bit numbers and return a 64 bit result. Sixty-four bit numbers are stored on the stack as two 32bit numbers with the lower number first on the stack (last off); 27 represented as a double number would therefore look like this [27 0] on the stack.

S>D This word will convert a single 32bit number into a 64bit number.

There are also characters (1 byte) used as storage that is worth mentioning here but is not dealt with in this section, see the section on strings.

15.2. Input

In conventional Forth system the word BASE controls how numbers are handled, BASE is in fact a system variable that hold the so called radix. Setting BASE to 10 will handle numbers as decimal, setting it to 16 will handle numbers as hex and so on. BASE can be set to any value but in practice only three number systems are supported, they are Binary (base 2), Decimal (base 10) and Hex (base 16).

The number 12 in each of the defined types are as follows:

Decimal	Hex	Binary
12	&C	B1100

The Forth words BINARY, DECIMAL and HEX conveniently set the base to the appropriate value. As a further convenience two number prefixes are provided: **&** for hex and **B** for binary, no prefix defaults to decimal.

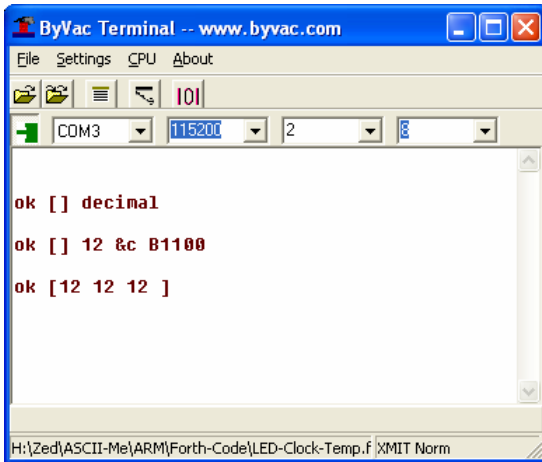


Figure 5 Number Input

By prefixing with **&** or **B** we produce the same results, as can be seen there are three values now on the stack with the same decimal value.

Important: numbers input to the system will ALWAYS be interpreted as decimal unless prefixed with either **&** or **B**, **regardless of what the base is set to.** This is not the same as ANSI or traditional Forth. In a microcontroller environment hex and binary are just (maybe more) valid then decimal. By implementing this system it makes it clear what the user intention is.

15.2.1. ?NUMBER

This word will convert text characters into numbers, it is rarely needed for user defined words but used extensively in the interpreting process.

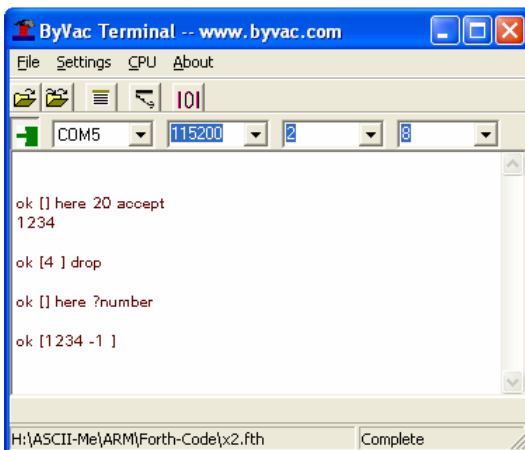


Figure 6 ?number example

The ?NUMBER word requires an address of a string, in the example (Figure 6), HERE is used, being at the end of the dictionary, as a spare section of RAM for which to store the text characters "1234". ACCEPT takes whatever is typed into the console and places the text into the address, in this case HERE (end of dictionary).

?NUMBER will now parse this address and convert the characters into a number one thousand two hundred and thirty four. The -1 following the number indicates that it was a successful conversion.

15.3. Output

The output display is determined by the BASE (words HEX, DECIMAL or BINARY set the BASE to the appropriate value). The important thing to remember is that the above words and BASE value effects only the output: what is printed to the screen.

15.3.1. U. and .

The main word for outputting numbers is **.'** dot or period, this will output a signed value, the value on the stack. This is an important distinction when working with larger numbers. A number with bit 31 set, &FFFFFFF0 for example will be displayed as -16 when using dot but when using U. it will be displayed as the unsigned value 4294967280.

15.3.2. PH. PD. And PB.

Will print out the unsigned values in the respective base, regardless of the current base setting.

15.3.3. Formatting

The following words **<#**, **#**, **#S**, **#>** and **STYPE** have complex descriptions in the Forth glossary and could possibly be used in unusual ways but really they should be considered for use with each other, just as **DO** would not be used without **LOOP**. The words are used to format numeric output and some examples will explain how they are used.

```

: TEST
  <# # # #> STYPE

```

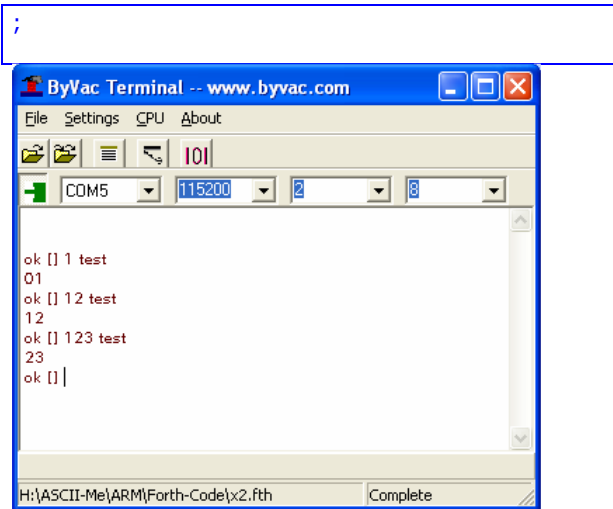


Figure 7 Number Format 1

The word TEST has been set up to output numbers to 2 places, the # word determines the number of places required, so 4 places would be <# # # # # #>. Figure 7 shows how leading zeroes are added should the number be less than the number of places specified and the right hand value of the number is displayed should the number be too large to fit into the number of places specified.

A variation on displaying numbers to two places is to use the #S:

```

: TEST
  <# # #S #> STYPE
;
    
```

The effect of this is to display numbers to **at least** 2 places but if it is a larger number then display that as well rather than chop it off.

Stack	Output
1	01
12	12
123	123

This may be more useful in certain circumstances, the effect of #S is to convert the next and all remaining numbers.

```

: mV
  <# # # # [char] . HOLD # #> STYPE
;
    
```

A lesser known option is to use HOLD to insert a character in the middle of a number format, the word [CHAR] simply converts what follows to its ASCII value so [CHAR] . is equivalent to 46 (the ASCII code for .). The above word **mV** will now display a number on the stack representing millivolts in Volts, thus:

1234 on the stack would display as 1.234

15.3.4. >DIGIT

This is pronounced "to digit" and it converts a digit into a printable character, thus 0 would be converted to 48 which is the ASCII code for 0. It is used extensively in the conversion words # and #S but it has limited use outside of this.

16. Strings Tables and Variables

."	VSPACE\$,	VSPACE\$
S"	CREATE	C,	VARIABLE
"	ALLOT	@	INTEGER
'	ALIGN	C@	CONSTANT
		!	
		C!	

For a full list see the core glossary.

This section deals with the words that are required for the adequate use of the RAM within the system. In Forth the user allocates and uses the RAM space to suit the environment and so predefined words in the core are made as flexible as possible. The technique is very different from A\$="fred" but once mastered it is much more flexible and arguably better suited to a microcontroller environment.

16.1. Variables

The simplest way of allocating RAM for use is to use this word:

VARIABLE TEST

The above creates a variable called TEST, when TEST is subsequently executed it will return the ADDRESS of the variable, to get the contents @ (fetch) is used and ! (store) is used to store values to the address.

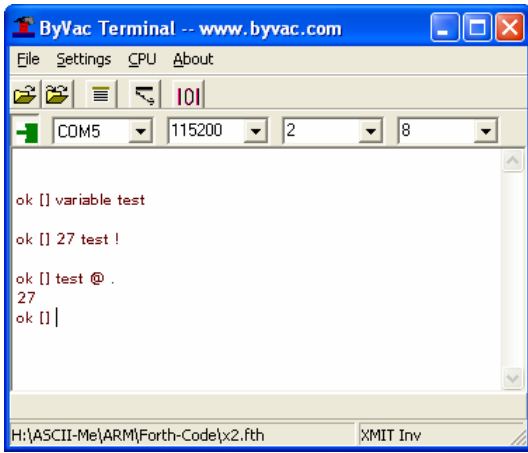


Figure 8 Using variables

This is illustrated in Figure 8. A variable allocates 4 bytes (32bits). The word INTEGER does exactly the same, VARIABLE is included for ANSI compatibility.

16.2. Constants

Variable and constants are similar but there are some subtle differences. A constant is defined thus:

```
55 CONSTANT TEST
```

A constant will return the **value** not the address, in this case 55. It is also stored as part of the word itself this means that when it is saved to Flash it cannot be changed.

16.3. Strings & Buffers

There are no specific words in the core for creating strings or tables but they can be constructed using the core words. The simplest form of string is to incorporate this in a word for example:

```
: HW ." Hello world" ;
```

The above incorporates the sting within the body of the word. This works fine for messages but quite often a table is required, a good example of this is names of the days of the week, Mon, Tue etc. that need to be equated to a number, 0 for Mon, 1 for Tue etc.

CREATE can be used for this: to explain, create will create a word and then when

executed it will return the next available address in RAM:

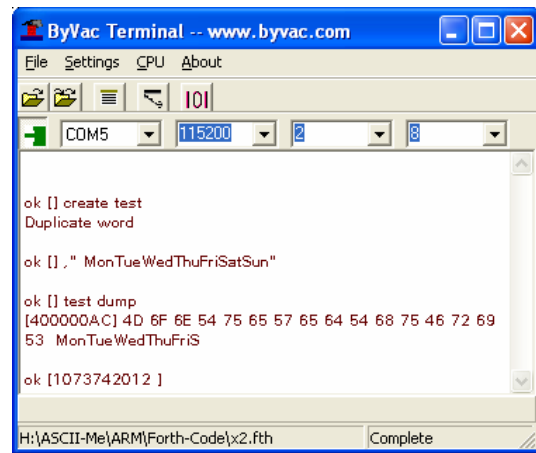


Figure 9 Creating a table

In the example the word CREATE has been used to create a dictionary entry, the word created is TEST and when executed will return the address of the next available place in the dictionary after the word was created.

In fact, in the above example this will be &400000AC. The word ,," (comma quote) that has been used after CREATE compiles text into the dictionary, to verify this TEST DUMP has been used that shows the contents of the addresses starting at &400000AC.

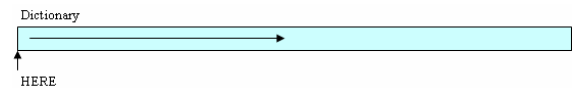


Figure 10 Dictionary before creating TEST

Before creating TEST the system looks like Figure 10, HERE points to the next available free space in ram.



Figure 11 After creating TEST

When TEST is created the header gets compiled into the dictionary so it can be found at a later date, HERE is moved up to the next available free space in RAM.

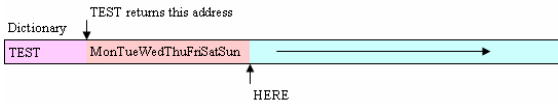


Figure 12 Inserting text into dictionary

Text is inserted into the dictionary by using `,` `HERE` once again moves as the text is put into the dictionary.

Because `TEST` places the address of the start of the new table on the stack, it is now a simple matter to calculate an offset from `TEST` to access a particular bit of the compiled string. The `W` of `'Wed'` for example has an offset of 6.

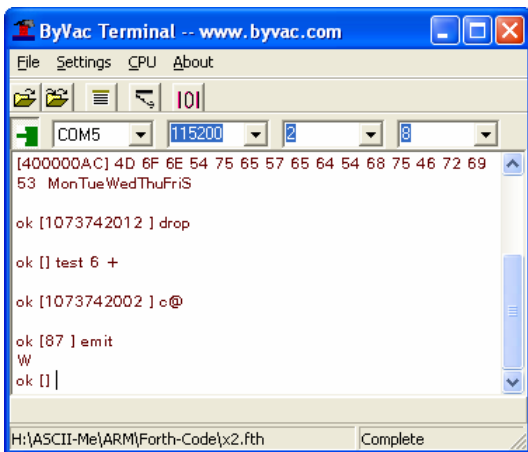


Figure 13 Accessing a table

As can be seen from Figure 13 adding 6 to the address placed on the stack by `TEST` points to the `W` of `'Wed'`, this can be fetched by using `C@` that will fetch a character from the supplied address. To verify that this is the `W`, `EMIT` has been used to print it out to the console.

To recap, the technique used is to create a word that will return an address of our table. This address is the current free space in the dictionary, this space was filled by using the `,` `HERE` word. This word (`,` `HERE`) is fine for text. To place numbers for example in the dictionary `C,` or `,` can be used. The former places a byte in the dictionary, the latter a word (4 bytes). To allocate space in the dictionary `ALLOT` is used.

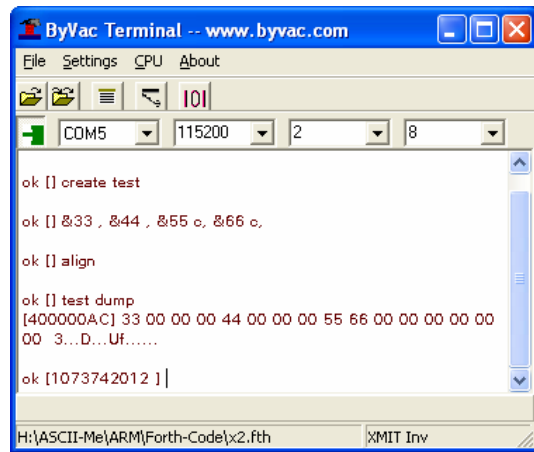


Figure 14 Placing numbers in dictionary

The example given in Figure 14 shows how numbers are placed in the dictionary for later use. A new word is created called `TEST`, as before when this word is executed it will return the address of the dictionary to a point of just after it was created.

Comma `,` will compile a 4 byte word into the dictionary, hence `&33` and `&44` are shown as `33 00 00 00` and `44 00 00 00`. Note that they are stored little endian meaning that the lowest significant bit is stored first. Using C-Comma `'C,'` only stores one byte. The dump of the memory from `TEST` onwards shows this. Looking at Figure 14 closely shows that **ALIGN** has been used, this is **IMPORTANT**, without this the system would probably crash as all ARM instructions must be aligned on a 4 byte boundary.

Most words do this automatically, in fact `,` will always finish on a 4 byte boundary regardless of the text size, any short fall is padded with zeroes. When using `C,` it is up to the user to take care of alignment.

16.4. Buffers

Often it is necessary to allocate some space in RAM for temporary storage that does not fit into the 4 byte space allocated by `VARIABLE` or `INTEGER`, this is where the word `VSPACE$` comes in. It will allocate space and assign the address of that space to a word you choose. For example:

```
18 VSPACE$ TEST
```

The above will in fact allocate 20 bytes as VSPACE\$ uses ALIGN to pad out to the nearest 4 byte boundary. The new word TEST will point to the start of the space allocated. As before @, C@, ! and C! can be used to access the contents of this space.

This word will actually allocate variable space, rather than dictionary space. This has implications when saving the code to ram, see the next section.

16.5. Buffers and Save

A word of warning. This section has used RAM and if the code is not saved to Flash then all will work as expected, when saving the code to Flash there are some considerations. The first thing to determine is if the table is a constant, as the day of week table would be or if it is a buffer or variable that is required. To take constants first:

The table as illustrated in Figure 9 will not work when saved to Flash. To create a table constant for use with both RAM and Flash use S" instead of ,". The best way to do this is to create a word rather than use create:

```
: TEST S" MonTueWedThuFriSatSun" ;
```

This will still return an address pointing to the start of the table but it will also work in Flash. Save will relocate the whole word to Flash memory and thus a table of constants will be created.

The second scenario is when a variable buffer is required, the above technique will not work because the table has become part of the Flash that cannot be simply written to, byte by byte. For this use VSPACE\$ as shown in section 16.4. This word will allocate RAM in an area at the top of the dictionary reserved for variables, when saved to Flash it will still point to that RAM.

17. Looping and Branching

For Next	Begin While Repeat	Do Loop +Loop
----------	--------------------	---------------

	Until	I, J
Exit, Escape, Unloop		

For a full list of words relating to looping and branching see the core glossary

Looping and branching follows the ANSI standard.

17.1. Do Loop

The DO LOOP structure is the main looping mechanism used in Forth and this particular implementation can have a positive or negative index, a signed 32 bit number. This will give values between -2,147,483,648 (0x80000000) to +2147483647 (0x7fffffff). To prematurely exit a DO loop the word LEAVE is used. DO-LOOP can be nested to any depth but only two levels of index can be observed by the words I and J:

```
: TEST
  2 0
  DO
    3 0
    DO
      J . I . CR
    LOOP
  LOOP
;
Output
0 0
0 1
0 2
1 0
1 1
1 2
```

There are some things to observe about this output and they are: 1. The first item on the stack is the limit value (the value the loop stops at) minus 1 and the last item on the stack is the start and index value, this value will increment up to the limit value. The implication of this is that the first number should always be greater than the second index number, also the output never reaches the limit value. 2. The index increments from its start value to the limit value minus one. 3. I and J are used as the indexes, although loops can be nested further only the top two indexes can be observed.

```
: TEST 10 -10
  DO
    I .
```

```

        2 +LOOP
;
Output
-10 -8 -6 -4 -2 0 2 4 6 8
    
```

+LOOP is used to modify the index increment from the default of 1, a +ve or -ve increment value can be used.

```

: TEST 10 0
  DO
    I DUP . 5 = IF LEAVE THEN
  LOOP
  ." Hello"
;
Output
0 1 2 3 4 5 Hello
    
```

LEAVE is used to leave a loop before the limit is reached. Note that as illustrated this will leave just the loop and carry on to just after the 'LOOP'. **ESCAPE** or **EXIT** can also be used, BUT within a loop structure the word UNLOOP must be used:

```

: TEST 10 0
  DO
    I DUP . 5 = IF UNLOOP ESCAPE THEN
  LOOP
  ." Hello"
;
Output
0 1 2 3 4 5
    
```

Using ESCAPE instead of LEAVE has now completely exited the word. It is important that **UNLOOP** is used; normally LOOP would do the job of making sure the leave stack is evenly balanced but the use of ESCAPE bypasses this mechanism, hence the need for UNLOOP. See the section on ESCAPE to understand the difference between EXIT and ESCAPE.

17.2. Return Stack

The return stack must not be used across a Do - Loop for example:

```

: TEST >r
  5 0 DO
  DO
    r@ INNER
  LOOP
;
    
```

The reason for this is that the loop structure uses the return stack to store the

nested loops. Unpredictable results will occur if this is used.

17.3. For Next

As an alternative to the Do loop structure, FOR NEXT has been provided, and will only handle a positive index.

FOR is equivalent to 0 DO and NEXT is the same as LOOP:-

```

: x
  2
  FOR
  3
  FOR
  J . I . CR
  NEXT
NEXT
;
Output
0 0
0 1
0 2
1 0
1 1
1 2
    
```

This loop does not need a starting value as it ALWAYS starts from 0 and counts up to the index given minus 1. A loop count specified as 5 will in fact iterate 5 times (0 to 5-1).

17.4. Branch Looping

All of the branch looping options start with the word BEGIN. The variance comes from how the loop is terminated.

BEGIN - AGAIN This is a loop forever constructed as:

```

BEGIN
  ... SOME OTHER WORDS
AGAIN
    
```

It is possible to get out of this loop by using ESCAPE or EXIT.

UNTIL This word will test the stack and branch back to BEGIN if the word on the stack is 0.

```

: TEST
  BEGIN
  KEY?
    
```

```
UNTIL
;
```

The above example will continue to loop until a key is pressed at the keyboard. The reason for this is that KEY? returns 0 (puts 0 on the stack) when no key is available, UNTIL will loop back to BEGIN when 0 is on the stack.

WHILE – REPEAT This is slightly more complex than the BEGIN – UNTIL loop but can be useful in certain circumstances as the test is carried out at the start of the loop rather than the end.

```
BEGIN
  <TEST VALUE ON STACK>
WHILE
  <DO THESE WORDS>
REPEAT
```

The basic construct is shown above, repeat will always loop back to begin, an example may help to explain.

```
: TEST
  0
  BEGIN
    1+ DUP 5 <
  WHILE
    DUP .
  REPEAT
  DROP
;
Output
1 2 3 4
```

This word begins with 0 on the stack. 1+ increments this to 1 and 'DUP 5 <' will leave true on the stack if the value on the stack is less than 5. If a true is on the stack the words between WHILE and REPEAT are executed. In this case the value on the stack is output to the console.

When the stack reaches a value of 5 'DUP 5 <' will no longer be true and WHILE will exit the construct (after REPEAT), DROP will drop the value on the stack which will of course be the value 5.

17.5. Escape

It is possible to leave any loop or branch structure with the word EXIT, this has the same effect as the EXIT word used in many

20

other languages. It will abort the current WORD and continue with the next one. Just as in other languages the current subroutine is ended and program execution continues with the next subroutine.

This will work while ever the program is running from RAM, however if the program is saved, as a result of the save operation the EXIT will be interpreted as the end of the word and thus when run from Flash will return unpredictable results. For this reason the **ESCAPE** word should be used **instead** of EXIT when inside a looping definition.

Example:

```
: FOO
  BEGIN
    KEY? IF ESCAPE THEN
  AGAIN
;
```

The above will compile correctly to Flash. See also ABORT and ABORT”.

18. Console I / O Access

Emit	Emit1	Uart
Key	Key1	
Key?	Key1?	

For a full list of words relating to looping and branching see the core glossary

The main access to the consol is via EMIT for output and KEY for input. EMIT will send a character that is on the stack to UART0 and KEY will receive a character from UART0 and place it on the stack.

The ARM provides a method of checking to see if a character has been received and KEY will remain in a loop until a character has been received so care must be exercised when using this word otherwise the system can 'hang' waiting for a character this is why KEY? exists. This word will return with either -1 or 0 depending on if a key is waiting in the key buffer. Zero means there is no key in the buffer.

UART = Universal Asynchronous Receiver and Transmitter

There is a 16 byte hardware buffer for both UARTS but no checking is carried out to see if the buffers have overflowed.

By default all I/O goes through UART0, words that produce output such as "." and " " etc. will send their output to UART0. The word UART is provided to enable this to be switched to UART1, the sequence **1 UART** will switch all the I/O to UART1 and **0 UART** will switch all the I/O to UART0. It is best to use this sequence within a word, say for temporarily switching as if used interactively, unless another terminal is connected to UAR1, control will be lost.

To access UART1 directly the words EMIT1, KEY1 and KEY?1 have been provided as the counterparts to the words used for UART0.

19. Flash Access & IAP

There are two methods of accessing the Flash memory for writing and that is with In System Programming (ISP) and In Application Programming (IAP).

ISP is used when a new core is required or there is a badly behaved user program that stops the system working. This is carried out with the Phillips LPC2000 flash utility. See the Hardware Guide for the procedure.

Described in this section is IAP that can be done from within the body of the program, the IAP can be used for erasing and writing to Flash and other miscellaneous functions.

Table 210: IAP Command Summary

IAP Command	Command Code	Described in
Prepare sector(s) for write operation	50 ₁₀	Table 211
Copy RAM to Flash	51 ₁₀	Table 212
Erase sector(s)	52 ₁₀	Table 213
Blank check sector(s)	53 ₁₀	Table 214
Read Part ID	54 ₁₀	Table 215
Read Boot code version	55 ₁₀	Table 216
Compare	56 ₁₀	Table 217
Reinvoke ISP	57 ₁₀	Table 218

Figure 15 IAP Command Table from Datasheet

As can be seen from the command summary there are 8 IAP commands, the tables refer to the tables in the datasheet. This description does not go into all of the commands but shows an overview of how to use the IAP word that is the interface to

the IAP. For in depth use of the commands a copy of the datasheet will be needed, this is on the CD-ROM that came with the BV processor.

The commands each have a command code (from 50 to 57) and can be optionally followed by parameters stored in memory. The main Forth word for executing In Application programming is **IAP**. The following gives an example of how to use command 53, to check for blank check sectors(s).

```
Command      53
Parameter1   Start sector Number
Parameter2   End sector number
```

IAP expects an address that will point to this structure. The structure consists of 32bit words not bytes, hence the offset of 4.

```
1. ok [] 20 vspace$ buf
2. ok [] 53 buf !
3. ok [] 0 buf 4 + !
4. ok [] 1 buf 8 + !
5. ok [] buf iap
6. ok [] buf @ .
7. 8
8. ok []
```

In the interactive code segment (blue above) a buffer has been created 20 bytes big using VSPACE\$. Note that the line numbers have been added just to aid explanation.

Address Offset	0	4	8
	Command	Start Sector	End Sector
Contents	53	0	1

The table shows how the words have been stored in BUF at lines 2 - 4, an offset of 4 has been used each time because a word consists of 4 bytes.

IAP is called with the address of BUF on the stack (line 5). When the command is completed a return code is stored back to the first word of the address, line 6 fetches this code, in this case 8.

Looking at the datasheet a return code of 8 means that the sector is not blank which is what is expected as the core Forth system is occupying this particular sector. A word can of course be created to do this without the need for typing and PAD is an ideal temporary buffer to use.

```
// ( start end --- flag )
: BLANK?
  53 PAD !           // command
  swap PAD 4 + !    // start
  PAD 8 + !         // end
  PAD dup IAP       // do command
  @                 // return value
;
```

The code above uses PAD for a temporary buffer but has the same effect of the interactive code. BLANK? will return 0 if the specified sectors are blank and 8 if they are not. Note that there is already a BLANKCHECK word in the glossary the above is similar to how it works.

Using this technique any of the IAP functions can be accessed, copying RAM to Flash for instance.

20. Revisions B3

Doc.	Code	Date	Change
1.0	1.01	May 2007	Pre-release
1.0	1.11	May 2007	Local Variables
1.1	1.15	Jun 2007	Release

21. Appendix B Resources

Here are some useful web resources:

www.byvac.co.uk Sells the BV511 ARM board and IASI devices

www.pin1.org Support for BV Forth and projects

www.forth.org The source for Forth code and documentation

<http://pygmy.utoh.org/forth.html> Frank Sergeant's Forth page

<http://www.taygeta.com/forthlit.html> Comprehensive Forth resources including tutorials and books.

<http://www.scansidk.com/Oem/Manuals>

Resource for MMC and SD cards