FIGnition User Manual

Contents

Introduction

FIGnition contains around 200 built-in commands in its version of Forth; divided into 17 categories. The Quick Reference lists the commands that are available; the Command Reference provides usage details for all the commands. The Programming Topics,Hardware reference and Cookbook are yet to be added.

Command Reference

Arithmetic / Logic

FIGnition has an extensive set of integer arithmetic operations. They normally work with 16-bit numbers, which are stored in the Data Stack in internal AVR memory, but some operations handle 32-bit numbers; stored as a pair of 16-bit numbers with the most significant 16-bits upper most in the stack. The top value of the stack is held in an AVR register (not on the data stack) to increase performance. Most operations either operate on the top item in the stack or the top two items in the stack, returning a single value.

Most Forth Arithmetic operations are fairly conventional; there's normal arithmetic, including unary negation and logical operations. Some operations are included because they're fast, special cases. 1+ and 1- are faster than 1 + and 1 - .

There are a large number of division and modulus permutations of instructions in FIGnition. There are two main reasons for this.

1. Standard low-level division algorithms naturally return both the modulus and division result and surprisingly often you have a use for both. Therefore it makes sense to include the combined operation.

2. Variants of division and modulus are used extensively when converting numbers to text.

Some operation operate on two or three items on the stack, returning one or two values.

Stack Inputs (: Return Stack Inputs)	**Command**	**Post-Command inputs**	Stack Effect	**Action**
a b	and		a & b	Returns the bitwise **and** of the top two items on the stack.
a b	or		a l b	Returns the bitwise **or** of the top two items on the stack.
a b	xor		a ^ b	Returns the bitwise **exclusive or** of the top two items on the stack.
a b	<<		a << b	Returns a * 2^b; or a shifted left b times.
a b	>>		a >> b	Returns (unsigned)a / 2^b; or a shifted right b times.
a b	+		a+b	Returns a+b.
a b	-		a-b	Returns a-b.
aL aH bL bH	d+		Lo(a+b) Hi(a+b)	The 32-bit number a=(aH*65536+aL) is added to the 32-bit value b=(bH*65536+bL) and the result is stored on the top two stack items.
a	neg		-a	The top item is negated.
aL aH	dneg		Lo(-a) Hi(-a)	The 32-bit number a=(aH*65536+aL) is negated and the result is stored on the top two stack items.

Stack Inputs (: Return Stack Inputs)	Command	Post-Command inputs	Stack Effect	Action
a b	u*		Lo(a*b) Hi(a*b)	The 32-bit unsigned result of unsigned a* unsigned b is calculated and stored in the top two stack items.
aL aH b	u/		(a Mod b) (a Div b)	The 32-bit unsigned number a=(aH*65536+aL) is calculated. (a Mod b) and (a DIV b) are stored on the stack; b is treated as unsigned.
a	1+		a+1	Increments a (it is 3x faster and shorter than 1 +)
a	1-		a-1	Decrements a (it is 3x faster and shoter than 1 -)
a b	*		a*b	Multiplies the signed values of a and b, returning the result on the stack.
a	2+		a+2	Adds 2 to a (it is faster and shorter than 2 +)
a	2*		2a	Multiplies a by 2 (it is 4 times faster and shorter than 2 * and 3 times faster and shorter than 1 <<).
	-1		-1	Adds -1 to the stack.
a b	+-		-a (if b<0) else a	Negates a if b<0
aL aH b	d+-		Lo(-a) Hi(-a) if b<0 else aL aH.	If b<0, negates the (signed)32-bit number aH*65536 + aL. Otherwise does nothing.
a	abs		abs(a)	Negates a if a had been -ve.
aL aH	dabs		Lo(abs(a)), Hi(abs(a))	Negates the 32-bit signed number a=(aH*65536+aL) if a<0
a b	min		min(a,b)	Returns the minimum of a and b.
a b	max		max(a,b)	Returns the maximum of a and b.
a b	m*		Lo(a*b) Hi(a*b)	The 32-bit signed result of signed a* signed b is calculated and stored in the top two stack items.
a b c	*/mod		(a*b Mod c) (a*b Div c)	Calculates signed a * signed b giving a signed 32-bit result, then returns signed a*b MOD signed c and signed a*b Div signed c
a b c	*/		(a*b Div c)	Calculates signed a * signed b giving a signed 32-bit result, then returns signed a*b Div signed c
aL aH b	m/		a/b	Calculates signed a=(aH*65536+aL) then returns a divided by signed b.
a b	/mod		(a Mod b) (a Div b)	Calculates signed a Mod signed b and signed a Div signed b .
a b	/		a Div b	Calculates signed a Div signed b.
a b	mod		a Mod b	Calculates signed a Mod signed b.

Stack Inputs (: Return Stack Inputs)	Command	Post-Command inputs	Stack Effect	Action
aL aH b	m/mod		(a Mod b) Lo(a Div b) Hi(a Div b)	The 32-bit unsigned number a=(aH*65536+aL) is calculated. (a Mod b) and the 32-bit unsigned result of (a DIV b) are stored on the stack; b is treated as unsigned.

Parameter Stack Operations

FIGnition includes a limited set of stack operations; and importantly doesn't include **pick** and **roll** . The normal practice with FIGnition is to use >r r and r>.

Stack Inputs (: Return Stack Inputs)	Command	Post-Command inputs	Stack Effect	Action
a b	over		a b a	Duplicates the second item on the stack to the top of the stack.
a	drop		--	Removes the top item from the stack.
a b	swap		b a	Swaps the top two items on the stack.
a	dup		a a	Duplicates the top item on the stack.
a	?dup		if a<>0, a a else a	Duplicates the top item of the stack if it's not 0.
a b c	rot		b c a	Moves the third item on the stack to the top item.
a b	2dup		a b a b	Duplicates the top two items on the stack.
a	s->d		a sign(a)	Sign extends a to 32-bits.

Control Flow

FIgnition's Forth contains a standard set of Forth commands for loops and conditional execution.

do .. Loops are loops where you know beforehand how many times you need to repeat it.

do ... loop always loops to one less than the limit. This means that it treats loop ranges as unsigned and countdown loops aren't possible.

do .. *n* +loop always loops until the counter crosses the boundary of the loop's limit. Because +loop is defined as crossing a boundary, it's possible to do create backwards loops, e.g. -10 1 do i . -1 +loop .

begin loops are loops where you want to keep looping round until a criteria is met.

begin *commands condition* **until** loops always execute at least once, because the *condition* occurs at the very end of the loop, after all the *commands* have been executed.

begin *condition* **while** *commands* **repeat** loops test the condition before the commands are executed, and thus the commands may be executed 0 or more times.

Stack Inputs (: Return Stack Inputs)	Command	Post-Command inputs	Stack Effect	Action
limit from	do		--	Starts a loop beginning at from and ending at limit, by pushing i and i' onto the return stack and then setting i to from and i' to limit.
	loop			Increments i and causes the program to loop round to its matching do command if the new value of i isn't the limit.
n	+loop		--	Adds n to i and causes the program to loop round to its matching do command if the new value of i didn't cross the limit.
	leave			Sets the loop counter i to the limit.
	i		i	The current value of the inner most loop counter.
	i'		i'	The current limit for the inner loop.
	begin			Starts a begin.. until loop or a begin .. while .. repeat loop; when the loop loops, it jumps back to here.
n	until		--	Ends a begin .. until loop. If n is 0, the program loops back to the matching begin, otherwise it continues.
n	while		--	Handles the condition for a begin .. while commands repeat loop. If n is 0 the program continues by executing *commands* all the way to the matching **repeat** .
	repeat			Ends a begin ... while .. repeat loop by looping back to its matching **begin**.
n	if		--	Tests the condition for an if commands then or if commands else alternativeCommands then command. If the condition is true (i.e. not 0) then the program continues executing the commands.
	else			The optional middle part of if ... else ... then . If the condition was false, the if ... else .. then continues executing the alternativeCommands between else and its matching then. If the condition had been true, the action of else is to jump past the alternativeCommands to the then part.
	then			The final part of an if ... then or if ... else .. then command; where execution resumes as normal.
cfa	exec		--	Executes the commands whose cfa (Code Field Address) is on the top of the stack. @See Headers.
	;s			Returns from the current command, by popping the return address from the top of the return stack.

Memory And I/O:

FIGnition uses both external Memory (Serial Ram) and internal memory belonging to the AVR Microcontroller. Therefore it supports commands which read and write to both types of memory.

Byte Order: @ and ! access 16-bit values as big-endian numbers: the first byte of a **var** variable is the most significant byte. External RAM is accessed for addresses > 0x8000 and Internal Flash is accessed for addresses <0x8000. With internal RAM, the byte order is little-endian: the first byte of an internal 16-bit number is the least significant byte. This is true for both the return stack and the data stack and matches the natural order for an AVR. Most of the time, FIGnition's conversion from external big-endian 16-bit values and internal little-endian 16-bit values is done transparently, though sometimes you need to check.

Block Memory Addressing: When using cmove and fill, the memory map is different. Addresses in the range 0 to 0x0FFF are treated as internal RAM addresses; addresses in the range 0x1000 to 0x7FFF are Flash memory and 0x8000 to 0xFFFF are treated as external RAM addresses. This means that it's possible to copy between all types of memory using cmove and fill; however, the bottom 4Kb of Flash isn't accessible.

Stack Inputs (: Return Stack Inputs)	Command	Post-Command inputs	Stack Effect	Action
addr	@		Mem[addr]	Fetches the 16-bit big-endian value at addr in external RAM (or Internal Flash)
addr	c@		Mem[addr]	Fetches the 8-bit value at addr in external RAM (or Internal Flash)
value addr	!		--	Stores the 16-bit big-endian value at addr in external RAM
value addr	c!		--	Stores the 8-bit big-endian value at addr in external RAM
src dst len	cmove		--	Copies len bytes of memory from src to dst. If src is <= dst; the memory is copied from the beginning to the end; otherwise it's copied backwards from the end to the beginning (so that writes don't overwrite values in src that haven't been copied yet).
src len value	fill		--	Copies len copies of the 8-bit value value to src.
addr	i@		Internal-Mem[addr]	Fetches the 16-bit big-endian value at addr in internal RAM
addr	ic@		Internal-Mem[addr]	Fetches the 8-bit value at addr in internal RAM
value addr	i!		--	Stores the 16-bit big-endian value at addr in internal RAM
value addr	ic!		--	Stores the 8-bit big-endian value at addr in internal RAM
orVal andVal addr	>port>		Internal-Mem[addr]	Performs an atomic read-write-modify operation on a byte in internal memory. The byte is read, then anded with andVal, then or'd with orVal and finally written back (all with interrupts turned off). It can be used to modify particular bits in ports without other tasks in the system affecting the result. >port> returns the original value of the internal Memory at addr, and thus can be used to implement rudimentary semaphore operations.
	spi			A core SPI driver. @TODO.
n addr	+!		--	Adds n to the contents of the external 16-bit value at addr, storing the results in addr.

Data

FIGnition contains a number of words for creating items of data in external RAM. In addition, one dimensional array handling is supported.

[#1].

```
10 arr stats
1000 5 stats ! ( stores 1000 at element 5 of the array stats)
5 stats @ . ( displays element 5 of stats i.e. 1000)
```

[#2].

```
10 bytes bins
1000 5 bins c! ( stores the low byte of 1000 at element 5 of the array bins)
5 bins c@ . ( displays element 5 of bins i.e. 232)
```

Stack Inputs (: Return Stack Inputs)	Command	Post-Command inputs	Stack Effect	Action
n	var	name		var creates a 16-bit variable called name and sets its value to n.
n	const	name		const creates a 16-bit constant called name and sets its value to n.
n	allot			Allocates n bytes to the end of the current program in external RAM. Before allot is executed, here returns the address of the first byte to be allocated.
n	,			Allocates 2 bytes to the end of the current program and copies n to external RAM at the original value of here.
n	c,			Allocates 1 byte to the end of the current program and copies n to external RAM at the original value of here.
n	arr	name		Creates an array called name in external RAM with n x 16-bit values allocated to it.[#1]
n	bytes	name		Creates a byte array called name in external RAM with n x 8-bit values allocated to it.[#2].

Comparison.

Fignition contains a number of simple commands used to compare values on the stack.

Stack Inputs (: Return Stack Inputs)	Command	Post-Command inputs	Stack Effect	Action
n	0=		-1 or 0	Returns -1 if n=0 or 0 otherwise.
n	0<		-1 or 0	Returns -1 if n<0 or 0 otherwise.
a b	u<		-1 or 0	Returns -1 if unsigned a < unsigned b; or 0 otherwise.
a b	<		-1 or 0	Returns -1 if signed a < signed b; or 0 otherwise.
a b	>		-1 or 0	Returns -1 if signed a > signed b; or 0 otherwise.

Stack Inputs (: Return Stack Inputs)	Command	Post-Command inputs	Stack Effect	Action
a b	=		-1 or 0	Returns -1 if a = b, or 0 otherwise.

Number Conversion

Standard Forth has extensive number conversion commands so that numbers may be displayed in a variety of formats and converted from a number of formats.

Setting the current number base. Forth provides two words for defining preset number bases: hex and decimal. By defining them as whole words, it avoids the problem that converting back to different number bases depends on the current base. For example, `10 base !` would normal set the base to base 10. However, if the computer was set to base 16, then `10 base !` would merely set the base back to 16 (because when you write 10 in base 16, it means 16).

[#1]. **<#**, **sign**, **hold**, **#**, **#s** and **#>** are used to generate formatted numbers.The process always involves starting with **<#** to set up number conversion in Forth. To actually convert numbers we need to provide a double-number on the stack (aL:aH) and numbers are always converted right-to-left; starting with the least significant digit. **#** converts the least significant single digit in aL:aH returning the new aL:aH . Thus, **#** can be used for fixed precision conversion. **#s** finishes off all the remaining digits. **#>** finishes the number conversion returning the address and length of the string (though the string can also be simply displayed using **".**). Hold is used to insert specific characters into the string. Finally **sign** is for signed number conversion and expects `sign , aL , aH` to be on the stack.

For example, if n is an unsigned 16-bit value in 10ths of a second we could convert it to text with:

```
0 ( to make it 32-bit) <# # ( convert 10ths) asc . hold (insert decimal point) #s ( finish digits) #> type
```

Or if we just wanted to display a 3 digit rev counter (and we know we never need more than 3 digits):

```
0 <# # # # ( generate 3 digits) #> type
```

Would work (we don't need the **#s** here).

Stack Inputs (: Return Stack Inputs)	Command	Post-Command inputs	Stack Effect	Action
	base		addr	Returns the address of the current base used for number conversion.
	hld		addr	When converting a number to a string, hld returns the address of the beginning of that string.
	tib		addr	Returns the address of the Terminal Input Buffer used for command line interpretation.
	in		addr	Returns the address of the offset from the Terminal Input Buffer for where interpretation is currently at.
	hex			Sets the current number base to 16 for hexadecimal numbers.
	decimal			Sets the current number base to 10 for decimal numbers.
ch base	digit		digit (-1 or 0)	Returns the unsigned digit's value in the current base and -1 if ch is in the the correct range for the base: a digit from '0' to 'base-1' or if base>9, the last acceptable character will be a letter 'A' to 'A+base-10'

Stack Inputs (: Return Stack Inputs)	Command	Post-Command inputs	Stack Effect	Action
text^	number		Lo(n) Hi(n) ntype	Expects a pointer to a string as input and returns the double-number converted from the text in the current base followed by its type: 0 for not a number, 1 for a 16-bit integer, 2 for a 32-bit integer.
ch	hold		--	Prepends ch to the hld string.
	pad		addr	The address for the end of the hld string, it's here + 0x43.
	<#			Begins number conversion by setting hld to point to the pad and terminating the hld string.
aL aH	#>		addr len	Exits number conversion, by dropping the number being built up and returning the address of the string and its length.
sgn aL aH	sign		aL aH	If sgn<0, prepends '-' to the hld string, it doesn't change aL or aH.
aL aH	#		Lo(a') Hi(a')	Extracts the least significant digit of a=(aH*65536+aL) in the current base; converts it to a character and **hold**s it. Returns a new a, divided by the current base.
aL aH	#s		0 0	Extracts all the remaining digits of a=(aH*65536+aL) in the current base, until a'=0.

User Interface:

FIGnition provides a number of commands for outputting text and numbers; and inputting them.

Emit Notes: Two characters are treated differently. 0 emit outputs nothing and doesn't move the cursor. 13 emit outputs a carriage return. Other characters in the range 1..15 output UDG characters 1..15. It's possible to override this behaviour by adding 256 to the character. Thus 256 emit outputs UDG 0 and 269 emit outputs UDG 13.

At Notes: In Text mode, at sets the character coordinate. In graphics mode it sets the pixel coordinate for displaying characters and setting the blitting position, thus *8 16 at* in graphics mode is equivalent to *1 2 at* in text mode. In addition, in hi-res mode at can be used twice to set the secondary pen coordinates and the primary pen coordinates for 2blt.

.hex Notes: .hex is a kernel routine which doesn't depend on the Forth system; it's much faster than . , but less flexible. It's also useful for debugging the Forth system itself.

Pause Notes: If n<0, Pause will wait for -n frames, but will exit early if the user presses a key (it doesn't return the keypress, you'll need to use inkey to read it). A pause of 32767 pauses for almost 11 minutes (PAL) or just over 9 minutes (NTSC).

at> Notes: at> is useful in text mode games for finding out what's on the screen at any given location, thus it's handy for collision detection.

Stack Inputs (: Return Stack Inputs)	Command	Post-Command inputs	Stack Effect	Action
ch	emit		--	Outputs the character ch on the screen and moves the cursor onto the next position. If the cursor moves off the end of the screen, the screen is scrolled (in text mode), see Emit Notes.

Stack Inputs (: Return Stack Inputs)	Command	Post-Command inputs	Stack Effect	Action
x y	at		--	Sets the cursor to x y . See At Notes.
n	.hex		--	Outputs n as an unsigned hexadecimal number preceded by '$'. See .hex notes.
	key		ch	Waits for the user to press a key and returns its character code.
	inkey		ch or 0	Returns the character code of the most recent key pressed, or 0 if no key has been pressed.
	cls			Clears the screen (in text and graphics mode)
	cr			Moves the print position to the start of the next line.
addr len	type		--	Displays len characters starting at addr.
	space			Displays a space character (equivalent to 32 emit).
n	spaces		--	Displays n space characters.
str	".		--	Displays the string at address str.
	."	*a message*"		Displays the literal string *a message*, terminated by a " character.
n	pause		--	Waits for n frames on the TV to pass and then continues. See Pause notes.
aL aH n	d.r		--	Displays a (which is aH*65536+aL) in a field of no less than n digits, in the current base.
aL aH	d.		--	Displays a (which is aH*65536+aL), in the current base.
a n	.r		--	Displays a in a field of no less than n digits, in the current base.
a	.		--	Displays a in the current base.
addr	?		--	Displays the 16-bit value at Mem[addr] in the current base.
	more		--	If the display position is on the bottom row, more waits for a keypress and then clears the screen.
x y	at>		internalAddr	Converts (x, y) to the internal Ram address in video memory. See at> notes.
x y w h	clBox		--	Clears an area on the screen w characters wide and h characters high from (x,y).

Text Processing

FIGnition Forth supports capable string handling, which is uniform across the system. Strings are a sequence of characters terminated by a 0 byte as in standard 'C' strings. It is possible to compile in literal strings; display strings, copy, cut, index and join strings; input strings from the user and convert between numbers and strings. Examples follow.

```
[#1] : stringEx1 " Hello World!" ". ; ( compile in a string, returning a pointer to it,
then display the string)
```

```
[#2] asc Hello . 72 ( 72 is the character code for 'H')

create days
" Sun" " Mon" " Tue"
" Wed" " Thu" " Fri"
" Sat"
```

Creates 7 constant strings holding the days of the week, which can be accessed as: *n* 4 * days +, e.g:

```
: .days 7 0 do i 4 * days + ". loop ;
```

Stack Inputs (: Return Stack Inputs)	Command	Post-Command inputs	Stack Effect	Action
	"	some_text″		In interpret mode, compiles the string *some_text* into the dictionary. In compile mode inserts code to push the address of the string to the stack and then compiles the string into the dictionary [#1].
	bl		32	Returns the character code for the 'blank' character, i.e. 32.
	tib		addr	Returns the address of the beginning of the Terminal Input Buffer
dir string^ ch	cIn"		string'	Given *dir*, a search direction: 1 or -1 (post-incremented if 1 or predecremented if -1); *string^* which points to a string and *ch* the character to search for; searches *string* in the specified *dir*ection for *ch* and returns *string'* the address where the character is found or where *string* terminated.
string^	"len		string'	Searches forward through *string* for the end of the string, returning it's address.
string^	"skipBl		string'	Searches forward through *string* for the first non-**bl**ank character or the end of the string, returning its address.
buff maxLen w h	boxed		exitCode	Interactively edits the text in buff in a screen region at the current at coordinate of dimensions (w,h). Editing finishes when <exe> or <cmd> are pressed and that key code is returned on exit.
ch	word		--	Text in the tib is searched until the matching character ch is found (or the end of text or end of line is found). The text is then copied to **here**.
string1 string2	"<>		difference	Compares string1 with string2 returning <0 if the string1 < string2; 0 if the strings match; >0 if string1 > string2. If string1 only matches string2 up to the end of string1, only the lower 8 bits will be 0.
	asc	aWord	asciiCode	Reads the following text word and returns the ascii code of the first character in that word. [#2].
srcStr dstStr	"!			Copies *srcStr* to *dstStr*, *srcStr* can be a string from RAM, internal RAM or Flash. *dstStr* should have enough space to store *srcStr* and the terminating byte.
srcStr dstStr	"+			Concatenates *srcStr* to *dstStr*. *dstStr* should have enough space to store the original string in *dstStr* and *srcStr*.

Stack Inputs (: Return Stack Inputs)	Command	Post-Command inputs	Stack Effect	Action
str n	"cut			Terminates *str* at offset *n*, thus making it n bytes long. This is similar to left$ in Basic Note: it doesn't check the length of str.
str n	"from		str'	Returns the string *str* starting at offset *n* or the end of the string if *n* > **str "len** .
buff	query			

System

FIGnition provides access to a number of low-level system features.

[#1] Kern Notes: FIGnition Forth contains a number of kernel vectors, which are pointers to useful system routines or addresses. If n is >0 a big-endian kernel vector is returned; otherwise a little-Endian kernel vector is returned. The following vectors are defined in the section on Kernel vectors.

[#1] SysVars Notes: FIGnition Forth contains a number of system variables in internal RAM. These are:

```
typedef struct {
     byte *gCur;     // In Text mode, a pointer to the print position.
                     // In Hires Mode, Y coord (byte) , clipTop (byte).
     byte gCurX;     // The Current X coordinate.
     byte buff[8];   // A Temporary buffer used for cmove.
     byte gKScan;    // The Current raw key scan.
     byte *stackFrame; // The loc Frame pointer.
     byte clipLeft;  // The Blitter's left clip coordinate.
     byte clipRight; // The Blitter's right clip coordinate.
     byte clipBot; // The Blitter's bottom clip coordinate.
     byte savedX;    // The previous x coordinate from an at command.
     byte savedY;    // The previous y coordinate from an at command.
} tSysVars;
```

[#1] sp0 Notes: The starting address of the data stack can be used to reset the data stack, by executing `sp0 sp i!` .

Stack Inputs (: Return Stack Inputs)	Command	Post-Command inputs	Stack Effect	Action
n	kern		vec	Returns kernel vector n. See [#1].
	vram		internalAddr	Returns the address of text mode video ram.
	clock		internalAddr	Returns the address of the frame counter, a 50Hz clock for PAL systems and a 60Hz clock for NTSC systems.
	sysvars		internalAddr	Returns the address of the kernel system variables in internal RAM. See [#2].
	sf		internalAddr	The address of the loc stack frame.
	rp		internalAddr	The address of the return pointer in internal RAM; this is the same as the AVR's stack pointer.
	sp		internalAddr	The address of the data stack pointer in internal RAM; this is the same as the location of the AVR's xh:xl rregister pair.

Stack Inputs (: Return Stack Inputs)	Command	Post-Command inputs	Stack Effect	Action
	sp0		internalAddr	The starting address of the data stack [#3].
	warning		addr	A Forth system variable. If it's set to >=0; an error will cause execution to return to the command line interpreter, otherwise, an error will cause execution to restart using **abort**.
	dp		addr	Returns the address of **here**, i.e. the address where the next byte will be allocated.
	current		addr	Returns the address of **latest**, i.e. the address of the most recently defined command.
	abort			Aborts the currently running command, resets the data stack and then restarts the command line interpreter by executing **quit**.
	quit			The interpretation loop: repeatedly reads a command line and executes it.
	cold			Restarts the FIGnition Forth interpreter from scratch, displaying the FIGnition logo and then initializing Forth variables.

Compiler

The working of FIGnition's Forth compiler is exposed to the user so that you can override the compiler's normal operation; define commands that define commands; explore the structure of compiled commands and even extend the compiler itself.

[compile] Example: `: compIf [compile] if ;`. Normally if expects to be part of an if ... [else ...] then statement and the if part is followed by a branch to the following then (or else). Here we compile in just the token for if.

literal Example: `: exLiteral [ 100 ] literal ;` This is equivalent to `: exLiteral 100 ;`. The purpose of literal is so that you can perform complex literal calculations within a definition, and then just compile the result into definition rather than having the definition having to calculate it every time it's run.

[#3]: `?pairs` is used to parse constructs such as `begin ... until` or `if ... else .. then`. These constructs are always immediate words which get executed even in compile mode; because their compilation process is more complex than merely appending their execution address. The first part of a construct (when being compiled) will push an identifier. When the next part of the construct is compiled, the compilation process checks the identifier matches the expected identifier using ?pairs and generates a "Mismatched" error otherwise.

[#4]: `compile` is used within an immediate colon definition to compile the following word whenever that colon definition is used. For example: `: testComp compile ?dup ; immediate` would cause `?dup` to be compiled into the dictionary whenever `testComp` was used in another definition. So, for example: `: test2 testComp ;` would in fact generate `: test2 ?dup ;`.

[#5]: `<builds ... does>` defining words, one of Forth's most powerful constructs. These are mostly used for type definitions, called 'definers' in Forth. They are used in the form:

```
: definerCommand <builds CompileTimeExecution does> RunTimeExecution ;
```

and the `definerCommand` is then used to create further definitions with a pattern and behaviour defined by `definerCommand`. For example:

```
: array1D <builds allot does> + ;
```

creates a definer called '`array1D`' which can then be used in the form:

```
10 array1D x and 100 array1D y
```

to create a 1 dimensional array called `x` with 10 elements and another 1 dimensional array called `y` with 100 elements. The `<builds allot` part is executed whenever `array1D` is run to create an actual array of data: `<builds` creates the header and the `allot` picks up the size of the array and allots the size number of bytes. Finally `does>` makes the array's cfa point to the code following `does>` . Later, when the named array (e.g. `x` or `y` in our case) is executed, in the form: `offset arrayName` (e.g. `5 x`) the address of the data for x is put on the stack and added to the offset giving the address of the element.

Stack Inputs (: Return Stack Inputs)	Command	Post-Command inputs	Stack Effect	Action
	[compile]	*command*		An immediate command that reads the following word and forces it to be compiled (thus you can use it to compile a command even if it would be interpreted).
[runtime] n	literal		--	An immediate command that compiles the top value on the stack into the program at the current point.
	state		addr	The address of the current compilation state. Can be used by immediate commands to determine if the computer is currently in compile mode (state @ isn't 0) or interpreting (state @ is 0) and thus behave differently.
	here		addr	The address of the first free location after the dictionary.
lfa	lfa>cfa		cfa	Converts a Link Field Address into a Code Field Address.
lfa	lfa>ffa		ffa	Converts a Link Field Address into a Flag Field Address.
lfa	lfa>nfa		nfa	Converts a Link Field Address into a Name Field Address (which is a conventional string).
	latest		addr	The address of the most recently defined command.
	?comp			Generates "Wrong State" error through ?error if Forth isn't in compile mode. It's used within the definition of words that can only be compiled, but not interpreted.
a b	?pairs			If a ≠ b, then a "Mismatched" error is generated using ?error [#3].
	:	*name*		Creates a new definition called 'name' and enters compilation mode.
err? errText^	?error			If err? ≠ 0, generates an error: it executes abort if warning < 0; otherwise if Forth is compiling it first displays the current word being compiled; then in all cases it displays the errText^ error message followed by the most recently read word which is at here.

Stack Inputs (: Return Stack Inputs)	Command	Post-Command inputs	Stack Effect	Action
	immediate			Is used at the end of a compiled definition (just after ;) to modify its header so that it becomes an immediate command.
cfa	x,			Compiles a cfa. If the code is <256; a single byte is compiled, otherwise a 16-bit word is compiled.
	compile			Compiles the next word in the instruction stream into the dictionary [#4].
	;			Terminates a definition and returns to immediate mode.
	[			Enters immediate mode.
	]			Enters compile mode.
	smudge			Toggles the invisibility flag for the `latest` definition so a previous definition of the same definition can be seen.
	create	*name*		Creates a new definition which returns its own pfa when executed.
	<builds			Used as part of a defining word to create the defining word's header. [#5].
	does>			Used as part of a defining word to create the defining word's run-time behaviour [#5].
	(	*text)*		Skips subsequent text all the way until the next ')'.

Locals

FIGnition Forth provides full support for persistent stack frames. The purpose of stack frames is to provide a means of defining local variables which can be accessed quickly and compactly; and can be allocated and deallocated dynamically. FIGnition Forth stack frames have persistent scope in that called functions have access to the stack frame of the calling function. Here's a short example:

```
: sfDisp
  l> 0 dup . ( display the stack frame item at offset 0)
  1+ l> 0 ( increment the item at offset 0)
  12 >l 2 ( store 12 at stack frame offset 2)
;

: sfExample
  4 locs ( allocate a stack frame with 4 bytes)
  17 >l 0 ( store 17 at offset 0)
  sfDisp
  l> 2 . ( display the stack frame item at offset 2)
loc; ( deallocate and return)
```

FIGnition stack frames are quick to access, because they use internal RAM and are compact, because accessing them only requires 2 bytes (the (>l) or (l>) primitive and a one byte offset). You can combine stack frames and parameter stack access.

Stack Inputs (: Return Stack Inputs)	Command	Post-Command inputs	Stack Effect	Action
n	locs			Allocates n bytes on the stack frame.
: returnAddr	loc;			Deallocates the current stack frame and returns from the current procedure.
	l>	value	sf[value]	Fetches the item on the stack frame offset by *value* bytes.
n	>l	value		Stores *n* in the item on the stack frame offset by *value* bytes.

Dictionary

FIGnition Forth provides a number of (immediate) words for managing the dictionary: finding commands within the dictionary; listing commands and forgetting commands.

[#1]: `vlist` examples. For example, `vlist cl` would list all the commands beginning with '`cl`' and by default this will be: `clBox`, `clock`, `cls`, `clip` . `vlist` by itself lists all the words in the dictionary.

Stack Inputs (: Return Stack Inputs)	Command	Post-Command inputs	Stack Effect	Action
	find	name	lfa or 0	Searches backwards for the command name in the dictionary starting with the most recently defined command, returning its link field address (lfa). If the word cannot be found, 0 is returned.
	vlist	prefixName		Lists all the commands in the dictionary that begin with prefixName [#1] .
	forget	name		Finds name in the dictionary and deletes that command and *every subsequently defined command* from the dictionary (if the comand is in RAM).

Interpreter

Forth provides a number of key words used for interpreting the command line.

[#1]. "run and interpret can be used to make Forth programmatically execute text. If the tib is made to point to the text in question, "run will execute the whole line and interpret a single command.

Stack Inputs (: Return Stack Inputs)	Command	Post-Command inputs	Stack Effect	Action
text^ delim	enclose			Given a text pointer and a delimiter, searches through the text until the delimiter is found, returning the pointer to the end of the text of the delimiter. Enclose also stops if the end of the text is found (a 0 character) or any character code below 14 (so <cr> counts as an automatic delimiter too).
	"run			Repeats interpret until the entire input text in the tib has been interpreted. [#1]

Stack Inputs (: Return Stack Inputs)	Command	Post-Command inputs	Stack Effect	Action
	interpret			Interprets a single command in the tib. The next word in the tib is looked up in the dictionary. If it can be found it is executed (in immediate mode) or compiled (in compile mode). If it can't be found, it is tested to see if it's a number and if so, pushed onto the stack (in immediate mode) or compiled (in compile mode). If it can't be found at all, an "Whats" (syntax) error is executed and Forth returns to the command line. [#2]

Return Stack Operations

FIGnition Forth has a number of simple words that operate on the Return stack.

FIGnition Forth programs use the return stack extensively, because it (deliberately) lacks commands to access more than the top 3 elements of the data stack (those these can be implemented). The general principle is that if the data stack contains, for example [*a b c d e*] and access to *a b c* is required we would first execute `>r >r` so that the stacks are: [*a b c* : *d e*] and when finished execute `r> r>` to shift them back. The advantage is that *a*, *b*, and *c* retain the same relative positions, which makes stack manipulation less error-prone and four elements *a* .. *d* are easily accessible.

[#1]: `r` note: `r` can be used to access the outer loop value in a pair of nested `do .. loop`s (in other Forths this would be done using the command `j`). This is because FIGnition Forth stores the inner counter and limit of a `do .. loop` in fast registers, but not on the return stack; instead the previous counter and limit are pushed on the return stack; thus the previous `do ... loop`'s counter will be the top item and can be accessed via `r`.

Stack Inputs (: Return Stack Inputs)	Command	Post-Command inputs	Stack Effect	Action
: n	r		n : n	Copies the number on the top of the return stack to the data stack. [#1]
n	>r		: n	Moves the value on the top of the data stack to the return stack.
: n	r>		n :	Moves the value on the top of the return stack to the data stack.

Graphics

FIGnition Forth provides core graphics commands to plot lores and hires images in a variety of plotting modes and blit bitmaps to the hires screen.

[#1], 2Blt Note: **2blt** expects **at** to be used twice, first to provide the *tile#* coordinates and secondly for the *tile2#* coordinates. For example, `10 10 at 30 30 at 0 $1010 4 $1010 2blt` would move a 16x16 pixel sprite from (10,10) to (30,30); using the bitmap at tile 0 for the first image and the bitmap at tile 4 for the second image.

[#2]. Pen. The Default plot mode is 1.

Stack Inputs (: Return Stack Inputs)	Command	Post-Command inputs	Stack Effect	Action
x y	plot			Plots a point in the current pen mode in either Lo-Res or HiRes pixels.

Stack Inputs (: Return Stack Inputs)	Command	Post-Command inputs	Stack Effect	Action
bitmap dim tile#	tile			This copies the *bitmap* whose *dim* is height*256+width pixels to the tile buffer starting at tile *tile#*.
tile# dim	blt			Blits the bitmap starting at tile tile# with dimensions dim [height*256+width pixels] to the frame buffer at the current **at** location, in xor mode. Afterwards, the graphics pen increments width pixels.
tile# dim tile2# dim2	2blt			Blits the bitmap starting at tile tile# with dimensions dim [height*256+width pixels] to the frame buffer at the secondary **at** location, in xor mode; then blits the bitmap starting at tile tile2# with dimensions dim2 [height2*256+width2 pixels] to the frame buffer at the primary **at** location. Afterwards, the graphics pen's primary **at** location is incremented by width pixels.See Note [#1].
tile# dim xrep yrep	blts			Blits the bitmap starting at tile tile# with dimensions dim [height*256+width pixels] to the frame buffer at the current at location, in copy mode. The blit is repeated xrep times along the x-axis and yrep times along the y-axis. Afterwards, the graphics pen is left at the previous coordinates + (width*xrep,height*(yrep-1)).
dx dy	clip			Defines the clip rectangle for which blitting has an effect. This clip rectangle starts at the current at coordinates and has dimensions (dx,dy) pixels.
mode	pen			Defines one of 4 plotting mode for plot operations: 0 = Move (no plotting is done), 1= Plot (white pixels are plotted); 2=Erase (black pixels are plotted); 3 = Xor (black pixels become white and vice-versa).[#2]
mode	vmode			Defines the graphics mode: 0 for 25x24 Text mode (with lo-res pixels; UDGs, but no blitting) 1 for 160x160 Graphics mode (with hi-res pixels; blitting, but no UDGs).

Storage

FIGnition Forth provides commands to read, write, copy and load (and interpret) blocks of flash storage.

[#1] blk> Notes: Block numbers in the range 16384 to 32767 actually read 256b from the physical page block#-16384. In addition, negative block numbers read a block from EEPROM, but treat it as though the block number was the negative of the given number.

[#2] loading Notes: load (and loads) are recursive; blocks loaded by load or loads can load other blocks and loading resumes at the correct place when the subordinate blocks have been loaded.

Stack Inputs (: Return Stack Inputs)	Command	Post-Command inputs	Stack Effect	Action
block#	blk>		physPage	Reads external flash block block# into external RAM at address -512. Returns physPage, the physical page in external Flash that would be used if >blk was used to write the block back.[#1]
physPage block#	>blk			Writes 512 bytes of external RAM from address -512 into external flash block block, whose physical page is physBlock (this should be the physical page returned by blk>).
	blk#		n	Returns the most recent block number read by blk>
n	load			Loads block n into external RAM and interprets it.[#2]
n len	loads			Loads len blocks starting at block n into external RAM, interpreting each block as it is loaded.[#2].
src dst	cp			Copies one block from block src to block dst. If src is negative, then the block is read from EEPROM.
n	edit			Loads block n then enters the editor. Editing block n if n<0 loads from EEPROM then treats it as editing block -N.A.M.