Revision 0.9.9: Changes and Improvements

Introduction
Revision 0.9.9 of the FIGnition firmware has a number of essential changes and significant improvements. Briefly, these are:

- The editor can now edit as much text as there is RAM available.
- **load** will now load text in contiguous blocks until there's a block <512 characters long.
- The Flash Disk driver has been re-written in Forth. It's now about half the length, but crucially makes Flash access in Forth much more convenient. The `blk>` and `>blk` commands no longer need the user to provide a physical block address (which will reduce programming errors).
- Stack underflow and overflow checking are now implemented with zero performance overhead!
- Break key checking (FIGgypad only) and Multiple, re-entrant, User-interrupts are supported with zero performance overhead.
- Memory management has been improved. FIGnition now checks the RAM size on startup and serial ram can be allocated from the top down (which means it's finally possible to use the Flash disk without it colliding with the video memory). The number of free bytes is now displayed on power-up.
- A no-video mode (which speeds FIGnition up by about 1.9 x).
- The ability to use the Software Serial Out routine.
- The ~ , _ keys are available from the keypad (and from FIGkeys).
- A few bugs have been corrected in the block editor, the blitter and the **quit**, **u/** and **locs** commands.
- A number of commands have been added to the system to support the above.
- With the exception of the Flash Driver, the system is now 10% to 20% faster than before according to the PCW Benchmarks.

The Editor
The editor can now edit as much text as there is in RAM. In addition some commands have been fixed and others have been added. Editor text will write and read multiple, contiguous blocks up to a block <512 characters long. It is possible to mark text from a different area to the editor itself.

Usage: **n edit** to edit text from block n. The editor will attempt to use all the remaining RAM for editing (with a space of about 700 bytes between **here** and the start of editor

text). The bottom line shows the first block being edited and the number of characters in the text **/** the maximum buffer space.

Features

Key	FIGkeys	Function
displayable character	displayable character	Inserts text at the current cursor location
Cursor	Cursor	Moves the cursor within the text area.
<mark>	Alt+m	Marks the current location for copying text.
<copy>	Alt+m	Inserts the character under the current mark position at the current text location.
<cmd+enter>	<ctrl>, <return>	Updates the editor status.
<cmd+z>	<ctrl>, z	Deletes the text.
<cmd+w>	<ctrl>, w	Writes the text to Flash starting at block n.
<cmd+$>	<ctrl>, $	Enters the shell, the command **bye** exits back to the editor.

Using The Shell
The Shell can be used for a wide variety of tasks. For example:

You can type in Forth commands in the shell, e.g. perform some calculations to the get results and then return to the editor.

You can load another block into FIGnition's RAM. For example, `30 blk> <exe>` would load block 30 into RAM and store the text at blk. So, then typing `blk* mark <exe>` would set the marker to the beginning of the block; so that when you `<copy>` text, text will be copied from that location to the editor's buffer.

You can generate a string and use it to mark text. For example, find `vlist lfa>nfa mark <exe>` would set the marker to the beginning of the text for vlist.

When you want to exit the shell you should type `bye <exe>` . The data stack *must* be in the same state as when you entered the shell.

Erasing The Disk
The Editor used to provide an 'E' command for erasing the Flash. The command **fdisk** now does this (**fdisk** can also be used from within the editor shell).

Command Completion
In the FIGnition Forth editor, pressing <shift>, <space> activates command completion. Whenever you start to type a new word, the list of available words that match the text you've typed appear in the bottom 4 lines. Pressing <shift>, <space> again will insert the first word on that list.

Keypad Characters
Two new characters are available from the Figgypad (and also via FIGkeys):

Key Combination	Character
<shift> , <SW1+SW7>	_
<shift> , <SW1+SW8>	~

The Forth Flash Disk Driver
The Forth Flash Disk Driver has been rewritten in Forth, but otherwise has the same functionality and supports the same disk format. Additional Forth byte codes have been added to the core system to provide access to the core 'C' functions for accessing the Amic Flash to read and write pages; read the Amic Flash ID, test the status of the Amic Flash and erase Flash sectors.

Stack Empty and Full Checking.
FIGnition Forth now checks for stack empty and stack full conditions at all times. More precisely, it checks whenever you execute **loop** , **until** , **while** , **repeat** , **if** (and it branches) , **else** , calling a command in external RAM , **;** , **c@** , **@** , **c!** and **!** commands.

If the stack pointer is out of range FIGnition displays the error message `Stack Empty in` *`aCommand`* or `Stack Full in` *`aCommand`* and then exits your code, returning to **quit**.

A stack empty condition happens when **sp** drops below **sp0** . A stack full condition happens when **sp** is within 40 bytes of **rp** (this is because system interrupts may use another 40-odd bytes).

Break Key Checking.
FIGnition Forth now checks for the break key being pressed at the same time as checking for Stack Empty or Full conditions. The break key combination is SW5+SW2+SW3+SW8. This combination was chosen because it's not easily pressed, can't be ghosted by another keypress and won't be a key combination needed by a game (because it's an unnatural position for your fingers to be in).

Break can be suppressed by setting GPIOR0 bit 7. The program code to do this is:

```
: breakoff ( dis/en -- ) 7 << 127 $3E >port> drop ;
```

Executing `1 breakoff` disables break, and executing `0 breakoff` enables break.

User Interrupts.
Multiple, re-entrant user interrupts are now supported in FIGnition Forth. Any ATMega168/328 interrupt can be trapped (apart from `vector_3`, used for Bitmap Prefetching; `vector_11` which is used for the video state machine and `vector_14` which is the software Uart).

To support them, 4 bytes were added to the **sysvars**, at offset 21 there's the interrupt vector, which is the `cfa` of a normal FIGnition Forth routine defined by the programmer. At offset 23 there are 3 bytes which contain activated interrupt flags. The firmware has every spare interrupt vector mapped to a single routine which sets the flag bit corresponding to the interrupt vector in an interrupt flag byte.

When FIGnition checks for Stack faults and the break key, it also looks for any set user-interrupt flag bits and if any are, it suspends the current Forth program and then makes forth call the user interrupt vector's `cfa` in **sysvars** +21. It also pushes the flag bits as two words on the data stack and clears them (so the same interrupts aren't immediately retriggered). The Forth user-interrupt routine is just an ordinary **:** -defined Forth word which finds the interrupt flags on the top two items of the stack and simply returns as normal.

It's fairly complex, but it's only as complex as it needs to be to support multiple interrupts simultaneously and re-entrantly. However, the following timer0 example shows it can be fairly simple to use at a basic level.

Code	Comments

`$35 const TIFR0`
`$6E const TIMSK0`
`$45 const TCCR0B`
`$46 const TCNT0`
	These are the addresses of some of the AVR registers for Timer0.We need its interrupt flags; its interrupt mask; control register B for starting and stopping the timer and the timer counter itself.

`: OVF ( flags23to16 flags15to0)`
`  drop drop`
`  16 $26 ic!`
`  clock i@ vram i!`
`; ( --)`
	OVF is the interrupt routine itself. Since only one interrupt is handled, the flags don't matter, so they are dropped. Otherwise, successive interrupts would cause the stack to overflow and a stack error would be generated.
	
	The interrupt routine just inverts the LED (`16 $26 ic!`) and then outputs the current **clock** value on the top left-hand part of the screen. It doesn't do anything clever with the interrupts itself, it just lets `timer0` carry on and generate another overflow interrupt in 256 `timer0` ticks.

`: setupInts`
`  [ find OVF lfa>cfa ] literal`
`  sysvars 21 + i!`
`  0 TCNT0 ic!`
`  5 TCCR0B ic! ( clk/1K)`
`;`
	`setupInts` sets up the interrupt. It finds the `cfa` of `OVF` and stores it in the FIGnition interrupt vector at **sysvars**+21. Then it clears the `timer0` counter and then starts it running at just under 20KHz (20MHz/1024). This will cause an overflow every 13ms (about 76Hz). However, it doesn't actually activate the overflow interrupt quite yet.

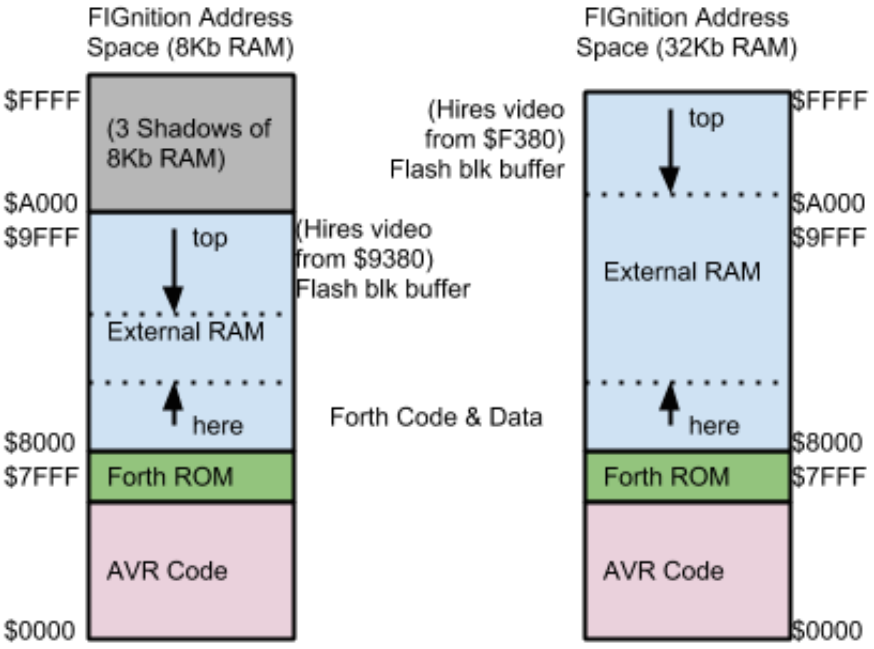
`: testInts`
`  setupInts`
`  1 TIMSK0 ic!`
`  begin`
`  inkey dup emit`
`  asc ! = until`
`  0 TIMSK0 ic!`
`;`
	`testInts` first calls `setupInts` to set the interrupt information up and then finally executes `1 TIMSK0 ic!` to activate the `timer0` overflow interrupt. Then it sits in a loop emitting anything you type until you press `!`. At the end it disables the `Timer0` interrupt. If it didn't do that, the interrupt would continue to run while the user continues to interact with FIGnition, e.g. while typing in new commands.

To use it, type `testInts <exe>` . The clock value is poked into video memory and the LED inverts every 13ms. While this is happening, you can type and text starts to appear from the current cursor position - hey presto, interrupt, driven code.

Practical interrupt code involves a number of subtle complexities. For example in this case `OVF` can't be allowed to run for more than 13ms or `OVF` itself would get interrupted! The routine would have to disable the `TIMSK0` interrupt at the beginning of `OVF`; clear any `OVF` and **sysvar** flags and then at the end re-enable the interrupt. In addition, if it handled more than one interrupt it would all have to go through the one routine which would then need to jump to the intended user interrupt code. This would add to user-interrupt latency, which is limited owing to the nature of the FIGnition Forth implementation and is significantly worse than a typical 8-bit computer. Interrupt latency is estimated to be typically in the region of 20μs to 200μs.

Nevertheless, it's possible to demonstrate interrupts in a fairly flexible way, and the mechanism more usable than many early 80s computers.

Memory Management.
Prior to Firmware 0.9.9, RAM inside FIGnition could only be allocated upwards, with **here** pointing at the last byte used by the system. Other areas of memory that were needed on demand, such as the Flash block buffer or the video memory were assigned to fixed addresses.



With Firmware 0.9.9 it's possible to dynamically allocate RAM from the top of RAM using **claim** and **reclaim**. The user variable **top** points to the last byte of memory allocated. This provides programmers with a means of, for example, allocating dynamic heaps independently of memory used by programs. Because the top of RAM depends on whether FIGnition has a standard 8Kb of RAM or is expanded to 32Kb of RAM; the size of RAM now

needs to be calculated. This is done by the command **top!** (which resets **top** to the top of RAM).

Details on all these commands are provided in the reference section.

No-Video (Fast) Mode
User Interrupts make FIGnition more suitable for embedded / realtime applications. However, to provide better interrupt latency (and lower power consumption) in some circumstances it would be helpful to be able to turn off the display. This also has the advantage of improving FIGnition's performance by a factor of 1.94.

The following commands allow the user to turn off and on the video display:

```
: fast 131 67 ic! ;
: slow 0 67 ic! ;
```

A short example follows:

```
: waitLoop
  0 do
    10000 do
    loop
  loop
;

: fastTest fast 20 waitLoop slow ;
```

Typing in `fastTest <exe>` will take the system into Fast mode, execute 200,000 empty loops and then re-enter slow mode.

There are limitations with the use of `fast`, namely that the keyboard isn't scanned in fast mode.

Software Serial Out
FIGnition's firmware has always had an internal interrupt-driven software serial out routine, which was developed to provide debugging information while the video display was being developed and debugged[1]. However, it hadn't been maintained for the past 3 years and would now conflict with normal system behaviour, in addition it used global variables which weren't accessible to Forth. In Firmware 0.9.9 the software serial out routine has been updated to make it usable again.

The software serial port always outputs via PORTD 6 (pin 12) at 9600 baud with 8-bits, no parity and 1 stop bit. It can operate correctly in text mode, but not in hires bitmapped mode. The following example code uses the Software Serial Port:

---

[1] It's part of the bootstrapping process. An LED blink routine was developed first and then it was used to debug a software serial out routine and this in turn was used to debug the video out code.

Code	Comments

`$2B const PORTD`
`$2A const DDRD`
`$44 const TCCR0A`
`$45 const TCCR0B`
`$46 const TCNT0`
`$47 const OCR0A`
`$6E const TIMSK0`
`$35 const TIFR0`
 	
`sysvars 19 +`
`  const swUartCh`
`sysvars 20 +`
`  const swUartState`
	These are the addresses of some of the AVR registers for accessing the software uart. We need to modify PORTD.6; both of the control registers for Timer 0, the Output Control 0A register and the timer 0 interrupt mask. In addition we need access to the system variables for the uart.

`: swUartInit`
`  $40 $FF 2dup PORTD`
`  >port> drop`
`  DDRD >port> drop`
`  0 TCCR0B ic!`
`  0 TIFR0 ic!`
`  2 $FF TIMSK0`
`  >port> drop`
`;`
	Initializes the software serial out. Sets PORTD.6 to output 1. Stops Timer0 and enables the OC0A interrupt.

`: swUartEmit ( ch --)`
`  begin`
`  TCCR0B ic@ 0= until`
`  gSwUartCh ic!`
`  11 gSwUartState ic!`
`  32 OCR0A ic!`
`  $82 TCCR0A ic!`
`  0 TCNT0 ic!`
`  3 TCCR0B ic!`
`;`
	Waits until the timer has stopped (it's stopped by the swUart interrupt as well as init). Sets up the Uart character and resets the uart state to 11. Sets the match period to 32 (which determines the 9600 baud). Sets the Timer mode to CTC mode that ends and Clears 0C0A on compare match, Resets the Timer0 counter and finally starts Timer0, running at clk/64, which is 312.5KHz.

The software serial out is a bit-bashing interrupt driven routine, but is accurate even though its interrupt routine may be called dozens of microseconds after its timer match occurs. This is because it uses the output compare pin itself to automatically generate the bit transition at the right time. So as long as the interrupt takes place before the next bit needs to be output, then they will be all transmitted at exactly the right time.

Bug Fixes
FIGnition contains a number of bug-fixes to the system:

1. The Editor's z command (to clear the edit text area) was very buggy. This has been fixed.

2. The **quit** command failed to reset the data and return stacks.

3. The blitter had a bug which meant that bitmaps had to be an exact multiple of 8-pixels wide.

4. The **locs** command (for allocating persistent stack frames) didn't which didn't take into account the way the avr return stack is post-decremented (odd, but that's the way Atmel handles it!). This meant that executing **>l 0** would overwrite the StackFrame restore value which meant that nested **locs** invocations would incorrectly restore the persistent stack frame.

5. The **u/** command didn't properly check for overflow, which occurs if the denominator is >=65536 * the divisor. It now returns a modulus of 0xffff if overflow occurred and a by-product of fixing the bug lead to an implementation over twice as fast and a 20% improvement for a number of benchmarks.

The following example code tests for the blitter bug:

```
hex create bm
  0103 , 070F ,
  1F3F , 7FFF ,
  80C0 , E0F0 ,
  F8FC , FEFF ,

: tb ( dim x y)
  1 vmode cls
  >r >r bm over 0 tile
  r> r>
  8 0 do
    2dup at >r >r
    0 over blt
    0 over blt
    1-
    r> r> 8 +
  loop
  key 0 vmode
;
```

Type `decimal $0810 10 10 tb <exe>` to test the bug. The diagonal images should be progressively clipped from the right, but the bug made them clip on the left hand side and sometimes align the bitmap output to the next tile. With the bug fix, the bitmaps do get progressively clipped from the right-hand side.

Benchmarks
The new benchmarks are 10% to 20% faster than before. This is partly due to the new horizontal pixel synchronisation code, which allows a little more time for user code to execute in text mode. It's also partly because the new stack/break key / interrupt checking mechanism involves a quicker calling mechanism (and reschedules a useful instruction into what would otherwise by a nop). But mostly it's because the **u/** algorithm is about twice as fast as its predecessor.

Here are the new results:

Benchmark	Time(s)	Ins's/loop	KIPS	vs Jupiter-Ace
BM1	0.0116	1	86.2	13.7
BM2	0.046	9	195.6	11.7
BM3	0.218	69	316.5	35.1
BM4	0.228	69	302.6	28.3
BM5	0.252	71	281.7	25.8
BM6	0.320	79	246.8	23
BM7	0.660	119	180.3	19.6
BM3L	0.034	10	294.1	29.4
BM1G	0.0960	9.844	102.5	31.4
			Avg (w/o BM1G)	23.3
			Avg (w/o BM3L and BM1G)	22.4

[Note: BM1G is described as having 9.844 instructions per loop. It really has 10 instructions, but the value is rescaled to compensate for the difference in pixels on a Jupiter Ace screen]

Note: There is an exception to the general performance improvement, the Forth Flash Disk driver is much most of the time owing to a slow algorithm for searching for Flash blocks. This will be addressed in Firmware 1.0.0.

Command Reference

There have been a number of changes and additions to the version of FIGnition Forth:

Stack Inputs (: Parameter Stack Inputs)	Command	Post-Command inputs	Stack Effect	Action
EEProm Access				
addr	ec@		u8	Returns the EEProm contents at address addr.
value addr	ec!			Stores value to EEProm memory at address addr.
src dst len	emove>			Copies len bytes of EEProm memory from EEProm address src to serial RAM at address dst.
Dictionary Access				
addr	>lfa		lfa	Searches for the Forth command that occupies the memory area that includes addr, returning it's lfa.

Stack Inputs (: Parameter Stack Inputs)	Command	Post-Command inputs	Stack Effect	Action
String Commands. enclose has changed between 0.9.8 and 0.9.9 and no longer properly conforms to normal Forth standards in this ROM. The reason for this is that enclose now reads from a fragment of a text buffer and uses a hidden command --> to obtain the next fragment of the text buffer if the end of the fragment is found. This is needed to support multiple contiguous blocks.				
textStart de-limiter	enclose		textStart textEnd	Searches for the character delim starting at text-Start. Returns a possibly modified textStart and textEnd in order to include the delimited text.
System Commands				
	bye			Returns from the current command shell (calling cold if this is the top-level command shell)
Memory Management				
	top!			Tests the size of external RAM, storing the last address ($9FFF or $FFFF) in top.
size	claim		addr	Claims a block of memory size bytes from external RAM at top, returning the first byte available. Top is then placed just below the claimed memory block.
addr	reclaim			Releases previously claimed external RAM from (and including) the block starting at addr. Top is updated to point after the block that started at addr.
	top		addr	Returns the address of top. Use top @ to return the contents of top.

Flash Disk Commands
 	
Loads and loads have been modified to handle multiple contiguous Forth blocks; whilst being upwardly compatible with the previous definition of load and loads. This is possible because in Version 0.9.8, the maximum text size for a block was 511 bytes and in 0.9.9 (and onwards) a block text length of <=511 characters will be still interpreted as the last block.
 	
blk* has now been defined because previously the block buffer was always set to point to the last 513 bytes of RAM. From firmware 0.9.9 it can be allocated dynamically.
 	
blk> and >blk no longer require a physical block parameter. They claim a block if needed (but never reclaim it. The user must reclaim the block manually and store 0 in blk* to free the block buffer).
 	
Low-level Flash Access via Forth byte codes is now possible. To gain access to them you need to provide access to the byte codes themselves:
 	
`: :inline create 128 + c, latest lfa>ffa dup c@ 128 xor swap c! ;`
`52 :inline SerFlRd`
`53 :inline SerFlWr`
`67 :inline SerFlEr`
`68 :inline SerFlId`
 	
Flash IDs for Amic chips are: A25L40 = $2013 , A25L040 = $3013 , A25L080 = $3014.				

Stack Inputs (: Parameter Stack Inputs)	Command	Post-Command inputs	Stack Effect	Action
n	load			Loads a set of contiguous Forth blocks starting at block n until a block containing <512 characters is found.
n count	loads			Loads *count* sets of contiguous Forth blocks starting at block *n* where each set of blocks is terminated by a block containing <512 characters.
	blk*		addr	Returns the reference to address of the current block buffer pointer. Use blk* @ to obtain the address of the buffer itself.
	blks		addr	Unused.
	blk#		n	The current block number being loaded.
	fdisk			Formats the External Flash Disk.
	blksize		n	Returns the size of a block (currently the constant 512).
virt	blk>			Claims a 512 byte buffer if needed and reads virtual block virt from Flash (or EEProm if a negative number is used) into it.
virt	>blk			Writes the 512 byte buffer at blk* to virtual block virt in Flash.
virt dst maxBlks	blks>		Over-flow?	Reads up to maxBlks contiguous blocks from Flash (or EEProm) starting at block virt; storing the text at dst. Returns -1 if > maxBlks could have been read or 0 otherwise.
virt src count	>blks			Writes count*blockSize bytes of external RAM starting at src to count contiguous Flash blocks starting at virt.
	SerFlId		id	Returns the Flash ID of the chip.
n dst^	SerFlRd			Reads Flash page n to internal Ram at address dst^
n src^	SerFlWr			Writes contents of internal Ram from address src to Flash page n.
n	SerFlEr			Erases the sector that starts at page n.
Editor Shell Commands Note: fdisk is the substitute for the E command in the editor.				
addr	mark			Sets the text editor's mark/copy address to addr.

kern supports the following vectors:

Vector	Stack Inputs (: Parameter Stack Inputs)	Command	Post-Command inputs	Stack Effect	Action
0		kChrSet		addr	Returns the address in Flash for the character set.
1		FigVer		version	Returns the version number of this FIGnition firmware.
2		(.")	String		(Compile mode only). Expects a string to follow (.") and displays the string.
3	value addr	toggle			xors the word in external RAM at addr with value.
4	lo hi str ntype	digits		lo hi str' ntype	Processes an unsigned number in the current base starting at str, leaving str pointing at the terminating character. ntype is decremented if it was <0 to begin with.
5	lo hi str ntype	intnum		lo hi str' ntype	Processes a signed number in the current base using digits.
6	str^	(vlist)			Lists the commands beginning with the text in *str*.
7	refresh?	ed		cmd	The vector for the core editing code. See the section on the editor (@TODO).
8	: returnAddr^	(“)		str^ : ret+2	The core routine for processing an inline string. It returns execution to just after the String leaving the pointer to the string on the stack.
9	: n oldRp oldSf .. ret^	(loc;)			Deallocates a persistent stack frame, restores the Return Stack pointer to just after it; and finally returns to the command that originally called the routine that set up this stack frame.
10		(ec!)			Performs the EEProm command that writes the EEPRom value in EEDR to EEProm address EEAR.

sysvars have been changed and are now:

Offset	Name	Type	Purpose
0	gCur	byte*	The address of the video RAM at the beginning of the current display line in text mode.
0	plotY	byte	The y coordinate of the pen on the screen in bitmap mode.
1	clipTop	byte	The y coordinate of the top of the clipping rectangle in bitmap mode.
2	gCurX	byte	The x coordinate of the print position in text mode; or the x coordinate of the pen in bitmap mode.
3	buff	byte[8]	An 8 byte buffer used for cmove.
11	gKScan	byte	The debounced keypad state (where bit 0=SW1 and bit7=SW8)