Revision 1.0.0: Changes and Improvements

Contents

1: Introduction
Revision 1.0.0 of the FIGnition firmware fulfills the long-term objectives for FIGnition including major improvements and a significant connectivity change. Briefly these are:

- Audio data transfer. Firmware 1.0.0 directly supports audio data transfer to and from RAM and indirectly between external flash and audio as well as providing the ability to upgrade via audio. Providing lossless audio formats are used, any loading function can be achieved via any audio source, including mp3 players, computers, CD players (and theoretically, obsolete cassette tapes).

- A java app called FIGgyTape exists for transferring program, data and firmware upgrade files between FIGnition and a host computer.

- AtMega168 FIGnitions require a replacement bootloader from version 1.0.0. The ability to upgrade and transfer data via USB will be lost, but the same functionality and more is provided via audio. A special migration firmware image is required to upgrade to Firmware 1.0.0.

- Floating-point arithmetic support. FIGnition now provides support for basic floating-point arithmetic and conversions to and from text and to and from 32-bit integers.

- Additional 32-bit comparison and stack handling commands.

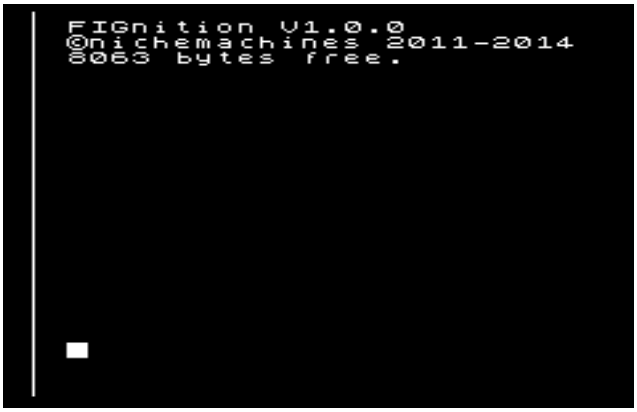
- A flkr video mode for audio data transfer.

- The Software Serial Out routine has been moved to Port D5.

- External Flash disk searching is as fast as earlier versions of FIGnition.

- A number of kern vectors have been added.

2: Upgrading FIGnition to firmware 1.0.0 on an AtMega328 FIGnition
This is done the same way as for previous upgrades, using USB. Use avrdude to download the firmware image called FirmwareRev1_0_0PAL32.hex (or FirmwareRev1_0_0NTSC32.hex for American users). The new audio features will be available to AtMega328 owners and the USB bootloader will still be available and will be the means for upgrading to future firmware revisions.



You have now migrated to Firmware 1.0.0 and can use all the new features :-)

3: Upgrading FIGnition to firmware 1.0.0 on an AtMega168 FIGnition

Upgrading to 1.0.0 (or later) on an AtMega168 FIGnition with earlier firmware is probably the most challenging upgrade task you'll have to face. The procedure should be realiable, but it is possible for you to end up bricking your FIGnition, so take care!

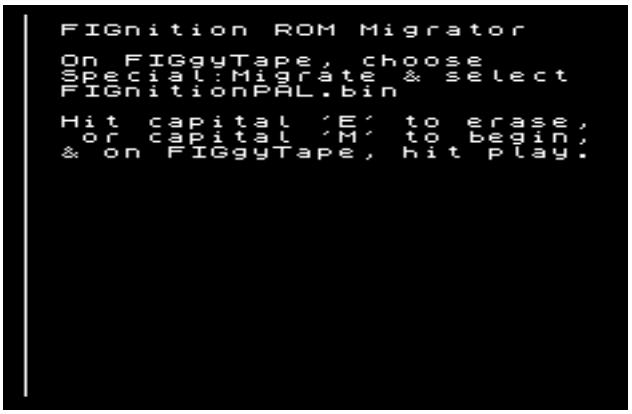
3.1: Requirements:

- A FIGnition running any firmware from 0.5.1 onwards.
- The FIGnition *must* be either a FIGnition RevC or RevD with the audio add-on or a FIGnition RevE (a.k.a FIGnition FUZE or FIGnition inFUZE from RS components) or a FIGnition FLINT with the audio add-on.
- A working external Flash chip (the Amic flash chip supplied with your FIGnition).
- A host computer with an audio jack output (can be Mac, PC or even a Sparc-based computer with Java).
- The host computer must have the Java SE 1.5 or later run-time installed. Macintosh computers are usually supplied with Java installed. Windows PCs can obtain the Java run-time from: http://www.oracle.com/technetwork/java/javase/downloads/jre8-downloads-2133155.html . It is possible to obtain versions of Java for Linux from openJava.net: http://openjdk.java.net
- A stereo to mono splitter. On my development Mac mini I required a stereo to dual mono phono audio splitter in order to prevent periodic glitches in audio output. A mono jack to phono converter would not work.

3.2 Install the Migration Firmware

From the Firmware 1.0.0 package, download `FIGgyMigrationPAL.hex` (or `FIGgyMigrationNTSC.hex` for American users) via Avrdude[1]. This is a custom firmware image and temporarily replaces your existing FIGnition firmware.

When the Migration firmware is installed, FIGnition reboots with the title message:



The first time you run the Migration firmware, you should **`Erase`** the external Flash hidden region, which is a reserved area of at least 24Kb on the external Flash chip. Press 'E' and FIGnition will ask for confirmation:

---

[1] Guidelines on using avrdude can be found at: https://sites.google.com/site/libby8dev/fignition/documentation/use-it#upgrading

```
Hit '!' to confirm
external Flash ROM area
erase.
```

You won't erase any existing blocks you've used for saving data, it just clears the reserved area. Press '**!**'. You should see the message:

Erasing: $20 (the page number will update quickly)

And after a short period:

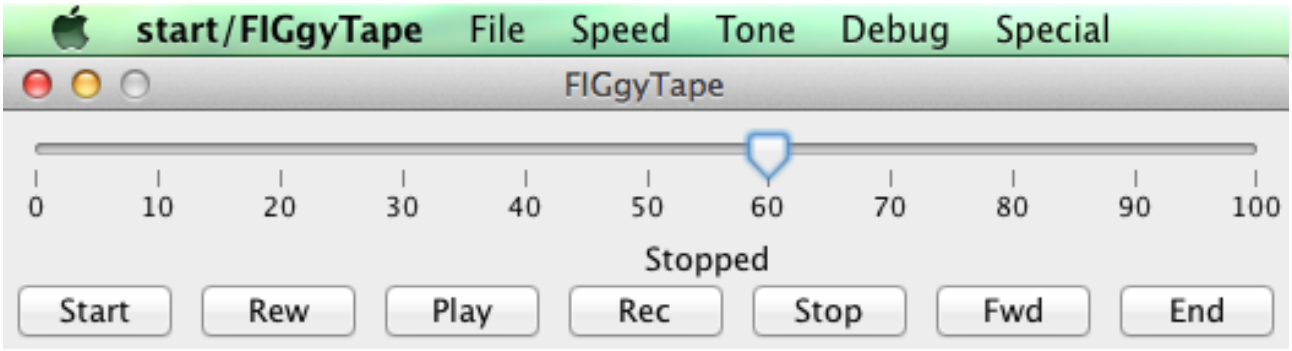
```
Erasing: $70
Done, hit key.
```

Then FIGnition will return to the initial screen, but without providing the 'E' option:

```
FIGnition ROM Migrator

On FIGgyTape, choose
Special:Migrate & select
FIGnitionPAL.bin

Hit capital 'M' to begin
& on FIGgyTape, hit play.
```

At this point you should run the java program: `FIGgyTape.jar`, which can be executed from the desktop or using the command line: **`java -jar FIGgyTape.jar`** . You will find that on the latest versions of Mac OS X (and Windows?) the program cannot be run directly. On a Mac you will need to hold the Ctrl key and choose open from the pop-up menu and then confirm that you want to run the program.

When the program runs you'll find it has a fairly crude interface:



cli

You will need to quit or exit all programs on your computer that could generate sound. If you're reading these instructions from your web-browser; close all the tabs apart from this one.

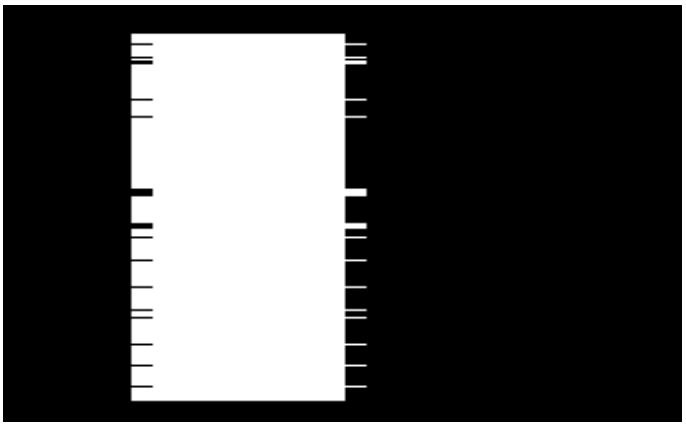
3.3: Setting the Volume Level

On FIGnition, press 'M' to begin Migration. The FIGnition flkr video mode will begin:

The relatively steady image shows that FIGnition isn't yet receiving any audio data.

Despite the instructions, the first time you run Migrator you will need to set the correct volume level on the host computer. On FIGgyTape choose: Tone:Leader and a few seconds later, press the Play button. A 2.7KHz tone should begin. Plug the phono plug into the phono socket on FIGnition and the other end into the audio-out on your PC. You should hear the sound stop when audio goes through to the FIGnition.

Adjust the volume level on FIGgyTape so that FIGnition reliably shows this kind of pattern:



On my Mac mini, setting the computer's volume to maximum and the FIGgytape volume to 60 works well (this is the default setting).

When the volume level looks good, press the Stop button and the FIGnition video display should look pretty much like the initial flkr video mode.

You are now ready to migrate to the new FIGnition firmware.

3.4: Performing the Migration

As per the original Migration program instructions, choose **`Special:Migrate..`** option and then choose the file **`FirmwareRev1_0_0PAL.bin`** (or **`FirmwareRev1_0_0NTSC.bin`** for American users). After a few seconds FIGgyTape should have processed the data in the file and it will be ready to play.

Press 

. FIGnition's flkr video should start to read the leader tone and then start to look like this kind of pattern:

You can see how the 0s and 1s affect it as FIGnition loads all the data in the new firmware Rev1_0_0 image. After another 30s or so, the download should be complete. Press 

on FIGgyTape at the end.

If any audio had been downloaded incorrectly, you will see a message like:

```
$0a/$40 pages downloaded
wrongly.
Press any key to retry.
```

Press any key and the Migrator will start again with the message:

```
On FIGgyTape, choose
Special:Migrate & select
FIGnitionPAL.bin

Hit capital 'M' to begin
& on FIGgyTape, hit play.
```

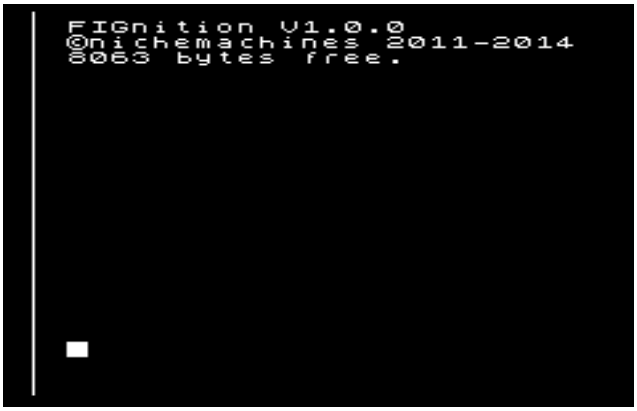
Whenever you need to upgrade to a new version of the FIGnition ROM you should start by erasing the external Flash ROM area (the upgrader also gives you this option) and then make repeated attempts to download the firmware. Sometimes waiting for a couple of seconds at the beginning of playback and then hitting pause on FIGgyTape before resuming can help (this I think is a glitch with FIGgyTape).

If the firmware was downloaded correctly, FIGnition should display the message:

```
FIGnition 1.0.0 firmware
download OK!
To complete the migration
the bootloader must be
overwritten. Press
Capital 'B' to overwrite
the bootloader or 'q' to
quit.
```

It should now be safe to complete the migration. If you press ‘B’ FIGnition will then ask you to confirm by pressing ‘!’ .

Press ‘!’. The screen will turn off; the FIGnition LED will flicker or just display fairly faintly for a few seconds and then FIGnition will automatically reboot with the message:



You have now safely migrated to Firmware 1.0.0 and can use all the new features :-)

3.5: Help! I’ve Bricked My FIGnition!

It’s pretty unlikely you will brick your FIGnition: the migration application checks that all pages have been downloaded correctly as they are downloaded and then checks again that all the new firmware on the external flash has CRC checking codes that match the set of CRC checking codes calculated by FIGgyTape and specially downloaded after the firmware image itself.

So, there are two levels of checking: you should not brick your FIGnition! Nevertheless, it may be possible.

There are four obvious solutions to this. Firstly, check the FIGnition Google group and search for the topic “Migration, Bricked FIGnition”. You might find that you haven’t really Bricked your FIGnition or that another suitable solution can be found there.

Alternatively, if you have an arduino or a spare FIGnition you can use it as an In-circuit Serial Programmer to reprogram your FIGnition’s AVR chip from scratch. This is covered in the FIGnition google group topic:

https://groups.google.com/forum/#!topic/fignition/9hi__dZIj0w

The third solution is to send the chip back to me along with a Paypal payment of £10 to cover the cost of postage and my labour :-)

The fourth solution is to see if another FIGnition owner will do the same job (perhaps even more cheaply :-)).

4: Audio Data Transfer

Here we cover the new audio data transfer commands `ear` and `mic` and how to use them with the java application `FIGgyTape`. Firmware 1.0.0 can transfer binary programs directly to and from RAM; can load and compile source code directly into RAM; can load and save source code to and from external Flash storage and future upgrades can be performed over audio too. In addition, you can execute command lines directly from FIGgyTape.

Finally, this section covers technical details on the audio format used by Firmware V1.0.0 (and later) as well as how to write your own audio data transfer code using the audio `kern` vectors.

4.1: FIGgyTape

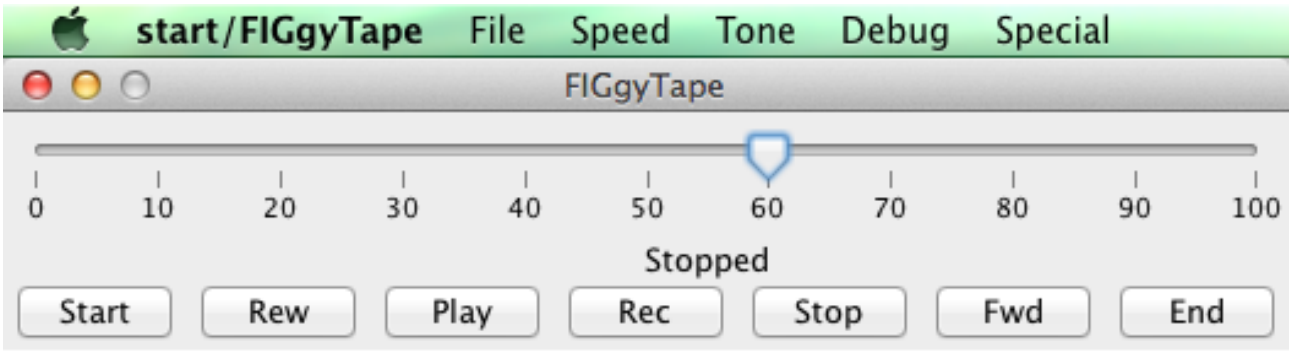
FIGgyTape is a small, cross-platform Java application that converts text or binary files into .wav audio output in the FIGgyAudio format and can also convert FIGgyAudio format audio input back into text or binary files. It’s available as part of the Firmware 1.0.0 distribution.

To use it you’ll need:

- A host computer with an audio jack output (can be Mac, PC or even a Sparc-based computer with Java).
- The host computer must have the Java SE 1.5 or later run-time installed. Macintosh computers are usually supplied with Java installed. Windows PCs can obtain the Java run-time from: http://www.oracle.com/technetwork/java/javase/downloads/jre8-downloads-2133155.html . It is possible to obtain versions of Java for Linux from openJava.net: http://openjdk.java.net
- A stereo to mono splitter. On my development Mac mini I required a stereo to dual mono phono audio splitter in order to prevent periodic glitches in audio output. A mono jack to phono converter would not work.

The application itself: `FIGgyTape.jar`, can be executed from the desktop or using the command line: **`java -jar FIGgyTape.jar`** . You will find that on the latest versions of Mac OS X (and Windows?) the program cannot be run directly. You will need to hold the Ctrl key and choose open from the pop-up menu and then confirm that you want to run the program.

When the program runs you'll find it has a fairly crude interface:



When using FIGgyTape it is wise to quit or exit all programs on your computer that could generate sound. If you're reading these instructions from your web-browser; close all the tabs apart from this one.

FIGgyTape supports the following UI features:

1. A slider for setting the audio level. I normally set the computer's audio level to maximum and then the FIGgyTape audio slider provides the maximum range it can.
2. Some buttons for a tape transport: the Start, Rew, Fwd and End buttons currently do nothing.
3. The play button. After a file (or a command line) has been processed into a FIGgyAudio format; pressing play (or the space key) will output the audio and turn the play button into Pause. Pressing pause will pause the audio, which is needed to transfer to external Flash or perform an upgrade. Pressing the button again will resume playback.
4. The stop button. It's possible to end playback or recordings by pressing the stop button. Pressing play afterwards currently will do nothing (you'll need to choose another file).
5. The Rec button. When transferring audio data from a FIGnition you'll need to make sure the audio connection goes from the FIGnition audio phono to the audio input on the computer and then press the Rec button. FIGnition will begin to record audio at 16bits per sample at 44.1KHz.
6. A command line text box. You can type any command in here up to 80 characters and FIGnition will convert it to FIGgyAudio format when you choose Debug:Cli.

FIGgyTape supports the following menu options:

1. **`File:New`**. Currently unimplemented.
2. **`File:Open`**.. This is used for processing .bin files. Choose a suitable file from the diolog box and FIGgyTape will process it. Press the Play button for FIGgyTape to start playing it.

3. **File:Run..** This is used for downloading source code directly into RAM and compiling it. Choose a suitable file from the dialog box and FIGgyTape will process it, generating a section of audio for the command line which then loads in the source code. Press the Play button for FIGgyTape to start playing it.
4. **File:Flash..** This is used for downloading source code or binary flash blocks into external Flash. Choose a suitable file from the dialog box and FIGgyTape will process it, generating a section of audio for the command line; another section for the Flash loader itself and finally the data you want to download. Press the Play button for FIGgyTape to start playing it and pause when FIGnition tells you to pause the audio.
5. **File:Upgrade..** This is used for upgrading to new firmware on an AtMega168 FIGnition. Choose a suitable file from the dialog box and FIGgyTape will process it, generating a section of audio for the command line; another section for the upgrader program itself and finally the new firmware image. Press the Play button for FIGgyTape to start playing it and pause when FIGnition tells you to pause the audio.
6. **File:Close.** Currently unimplemented.
7. **File:Save.** This is used for saving binary or text files after you've recorded them on a FIGnition. Choose a suitable name for the file from within the dialog box and save the data.
8. **Speed.** These are sample-rate settings for data transfer. The default and only currently tested rate is 11KHz.
9. **Tone:Leader.** This option is used for testing the correct volume level. Press type `6 ear <exe>` on your FIGnition and then choose Tone:Leader. The flkr video image will appear and you can adjust the volume as described in section 3.3 until the video image looks like it should.
10. **Tone:LeaderStart/LeaderByte** and **Saw**. These were menu options used in development and have no current purpose.
11. **Tone:Cli**. This converts a command line in the text box into FIGgyAudio format audio. Press the Play button for FIGgyTape to start playing it.
12. **Tone:Rec (pure Wav)**. This was a menu option used in development and has no current purpose.
13. **Debug:Log**. This causes lots of extra processing information to be dumped in a file called DebugLog.txt. It is for development purposes.
14. **Special:Migrate**. This is covered in sections 3.4.

4.2: Loading Programs and Data Into FIGnition

This section covers loading into RAM and also External Flash.

Forth Programs Directly To RAM: To load source code programs directly into RAM and compile them: on your FIGnition type the command:

```
6 ear <exe>
```

Connect the audio phono cable to FIGnition's audio socket and (via an audio splitter if needed) to the computer's audio out jack socket. Run the FIGgyTape java app.

If you haven't done so already, you will need to make sure the volume level on FIGgyTape is correct. See section 3.3.

When this is done, choose **File:Run..** from FIGgyTape and then choose the `.fth` program you wish to run. For most FIGnitions, this must be a `.fth` file less than 6Kb (and it might need to be slightly different from the Flash version so that it doesn't try to load and

compile itself from Flash). FIGgyTape will convert the file into the FIGgyAudio format. Press play on FIGgyTape. FIGgyTape will transfer a command line first; the screen will flash; and then after a few more seconds the Forth program will load and compile. If everything loads correctly you'll see quite a bit of gibberish on the FIGnition screen and after a pause (while it compiles) you should see an "OK" message. Otherwise as soon as the audio finishes you'll see the error message:

```
Tape :-(
```

When the audio stops, press the Stop button on FIGgyTape.

Forth Programs To External Flash: To load source code programs into Flash requires about 2Kb of free RAM on your FIGnition and is at least a two-step process. Files can be downloaded onto FIGnition with multiple attempts. On your FIGnition type the command:

```
block 6 ear <exe> ( e.g. 20 6 ear will load to block 20 onwards)
```

Connect the audio phono cable to FIGnition's audio socket and (via an audio splitter if needed) to the computer's audio out jack socket. Run the FIGgyTape java app.

If you haven't done so already, you will need to make sure the volume level on FIGgyTape is correct. See section 3.3.

When this is done, choose **File:Flash..** from FIGgyTape and then choose the file you wish to transfer. FIGgyTape will convert the file into the FIGgyAudio format. Press play on FIGgyTape and get ready to press pause. After a few seconds, FIGgytape will transfer a command line; the screen will flash; and then after a few more seconds a Flash loader program will load. If it loads OK, you'll see the message of the form:

```
Pause tape to prep
10 blocks.
```

Appear on the screen and when the flash loader is ready you'll see the message.

```
Hit key & start audio..
```

Press play on FIGgyTape to resume playback and after a few more seconds it will transfer the file to external flash memory.

If everything loads correctly you'll see quite a bit of gibberish on the FIGnition screen and after a pause (while it compiles) you should see an "**OK**" message. If any error occurs during the initial loading process you'll see the error message:

```
Tape :-(
```

When the audio stops, press the Stop button on FIGgyTape. If a loading error occurs you should re-attempt to download the file as described earlier.

At the end, the flashloader will have been deleted from RAM. If you downloaded a source program you will be able to compile it by typing:

```
block load <exe>
```

And run it according to the instructions for the program.

Executing Forth command lines from FIGgytape: On your FIGnition type the command:

```
6 ear <exe>
```

Connect the audio phono cable to FIGnition's audio socket and (via an audio splitter if needed) to the computer's audio out jack socket. Run the FIGgyTape java app.

When this is done, type the command line you want to execute into the edit box on FIGgyTape and then choose Tone:Cli . The Command line will be transferred to FIGnition and executed immediately (it can also overwrite the beginning of your program area too!).

4.3: Saving Source code and Data from a FIGnition

This section covers saving binary images directly from FIGnition and blocks of Forth code from Flash.

Saving Binary Images From external RAM: To save a binary image from RAM type:

```
start end 6 mic fileName.bin" <exe>
```

For example, if you're running a program called **GiniSim** to save it as binary, type:

```
$8000 here 6 mic GiniSim.bin" <exe>
```

FIGnition will display the message:

```
Start Rec, hit key.
```

Now make sure the audio cable connects the FIGnition Audio socket to the mic input on your computer (with the mic level set to something fairly sensible). Press the Rec button on FIGgyTape. After about 4 seconds finally press any key on FIGnition to start saving the program.

The display should change to a flkr video image and you'll see the data being saved on the screen. The program should save pretty quickly (FIGnition Forth is quite compact).

When it's finished FIGnition should display "**OK**". Press the Stop button on FIGgytape and after waiting for a little while, choose **File:Save..** and save the program. If the file was transferred correctly it should save OK with the same name as you gave it at the beginning.

Saving Blocks of data From external Flash: To save blocks of data from external flash you will first need to load in the program `ExtFlashSave.fth` into Flash as described in section 4.2. Once this is done; compile the external Flash Save program into FIGnition by typing:

```
extFlashSaveBlockNumber load <exe>
```

When this is done the **save"** command is now available. Type:

```
blockStart blockCount 6 save" fileName.fth" <exe>
```

For example, to save a source code program called Luna that's 7 blocks long starting at block 113, type:

```
113 7 6 save" Luna.fth" <exe>
```

FIGnition will display the message:

```
Start Rec, hit key.
```

Now make sure the audio cable connects the FIGnition Audio socket to the mic input on your computer (with the mic level set to something fairly sensible). Press the Rec button on FIGgyTape. After about 4 seconds finally press any key on FIGnition to start saving the program.

The display should change to a flkr video image and you'll see the data being saved on the screen. The program will take somewhat longer to save as it's saving source code.

When it's finished FIGnition should display "OK". Press the Stop button on FIGgytape and after waiting for a little while, choose **File:Save..** and save the program. If the file was transferred correctly it should save OK with the same name as you gave it at the beginning.

4.4: Upgrading the firmware on an AtMega168 FIGnition via Audio

This section covers upgrading the firmware using Audio itself. It works in a similar way to the migration program. It requires about 2.5Kb of free RAM on your FIGnition and is at least a two-step process. Files can be downloaded onto FIGnition with multiple attempts. On your FIGnition type the command:

```
6 ear <exe>
```

Connect the audio phono cable to FIGnition's audio socket and (via an audio splitter if needed) to the computer's audio out jack socket. Run the FIGgyTape java app.

If you haven't done so already, you will need to make sure the volume level on FIGgyTape is correct. See section 3.3.

When this is done, choose **File:Upgrade..** from FIGgyTape and then choose the new firmare you wish to transfer - it will be a binary file and ***must*** be a firmware version later than 1.0.0.

FIGgyTape will convert the file into the FIGgyAudio format. Press play on FIGgyTape and get ready to press pause. After a few seconds, FIGgytape will transfer a command line; the screen will flash; and then after a few more seconds the upgrader program will load. If it loads OK, you'll see the message:

```
Pause tape.
```

Appear on the screen and when the upgrader is ready you'll see the message:

```
Hit 'E' to erase, any
other key to upgrade.
```

As in section 3.2, you should press ‘E’ is you’re downloading a new version for the first time and you’ll need to confirm by pressing ‘!’ (or ‘c’ to cancel). After an erase you’ll see the message:

```
Done. Hit key to upgrade.
```

Press any key and then press play on FIGgyTape to resume playback. After a few more seconds it will transfer the new firmware image to a special region on external Flash memory.

When the audio stops, press the Stop button on FIGgyTape, FIGnition will check that the external Flash pages match the supplied check codes and then you’ll either see the success message:

```
0 bad pages, done!
Reboot with SW1+SW3
```

Or an error message.

```
x bad pages, retry
```

You can re-attempt to download *just* the firmware again, by choosing **`Special:Migrate..`**from the FIGgyTape program; pressing any key apart from ‘q’ on FIGnition and then pressing the play button on FIGgyTape. The firmware upgrader will reload blocks that failed to load the first time. This process can be repeated until the firmware fully loads correctly and an **`OK`** message is displayed.

To complete the upgrade. Unplug your FIGnition; hold down SW1 and plug the USB cable back in; press SW3, the LED should flicker and the firmware will be upgraded after a couple of seconds.

4.5: Command Line And Kern reference.

Stack Inputs (: Parameter Stack Inputs)	Command	Post-Command inputs	Stack Effect	Action
Loading and Saving using audio Note: dataRate governs the data transfer rate, theoretically they are as follows, but only rate 6 is currently supported. **6 is clk/64, 1Period=28 => 11.21KHz** 5 is clk/64, 1Period=56 => 5.605KHz. 4 is clk/64, 1Period=113 => 2.808KHz. 3 is clk/8, 1Period=14 => 89.69KHz 2 is clk/8, 1Period=28 => 44.843KHz 1 is clk/8, 1Period=56 => 22.421KHz 0 is clk/8, 1Period=113 => 11.21KHz.				
dataRate	ear			FIGnition loads the next block of data containing a RAM header at the given dataRate.

Stack Inputs (: Parameter Stack Inputs)	Command	Post-Command inputs	Stack Effect	Action
start end dataRate	mic			FIGnition saves bytes start .. to end in external RAM at the given dataRate.

Kern Vectors

The conventional way to define a kern command is:

```
: inline
  create -1 allot
  dup 256 < if c, else , then
  $80 latest lfa>ffa c!
; immediate
```

following kern vectors are supported:

```
11 kern inline VDskFind
12 kern inline crc
13 kern inline audioOutHeader
14 kern inline >tape
15 kern inline +tape>
16 kern inline -tape
17 kern inline tape>
18 kern inline />tape
19 kern inline packet
```

buff^ blk#	VDskFind		phys	Searches for the Physical block corresponding to logical block blk# and returns its value. The last block on the Flash disk is returned if it cannot be found. buff^ is the area in internal RAM to be used as a buffer (usually vram is used).
data crc	crc		crc'	Calculates the next CCITT 16-bit CRC code given the byte value data.
start len type dataRate	audioOutHeader	fileName"	bias packet	Used when saving data. Starts the leader tone and displays the message "Start tape & press key". Fills in a header packet with the start and len entries. Names the header packet fileName. Resets the bias to 0 and returns the packet pointer.
bias src len	>tape		bias'	Sends len complete audio frames from internal memory at src with an initial audio bias and returns the final audio bias.
dataRate	+tape>		frame^	Takes FIGnition into flkr mode; sets up the packet to work at dataRate; waits for a header packet to be received and then returns a pointer to its audio frame.
expectedPkts	-tape			Turns off flkr video and disables tape interrupts. If expectedPkts doesn't match the downloaded packets, generates a "Tape :-(" error. Otherwise continues.

Stack Inputs (: Parameter Stack Inputs)	Command	Post-Command inputs	Stack Effect	Action
blk	tape>		frame^	Reads in the next packet, update the loadmap for the given blk and returns a pointer to its audio frame or 0 if the packet's crc didn't match its data.
timerClk	/>tape			Stops mic output interrupts and sets the timer 0 clk rate.
	packet		packet^	Returns the internal RAM pointer to the tAudioPkt used to manage Audio data transfer (see section 4.6).

4.6: Technical: FIGgyAudio format

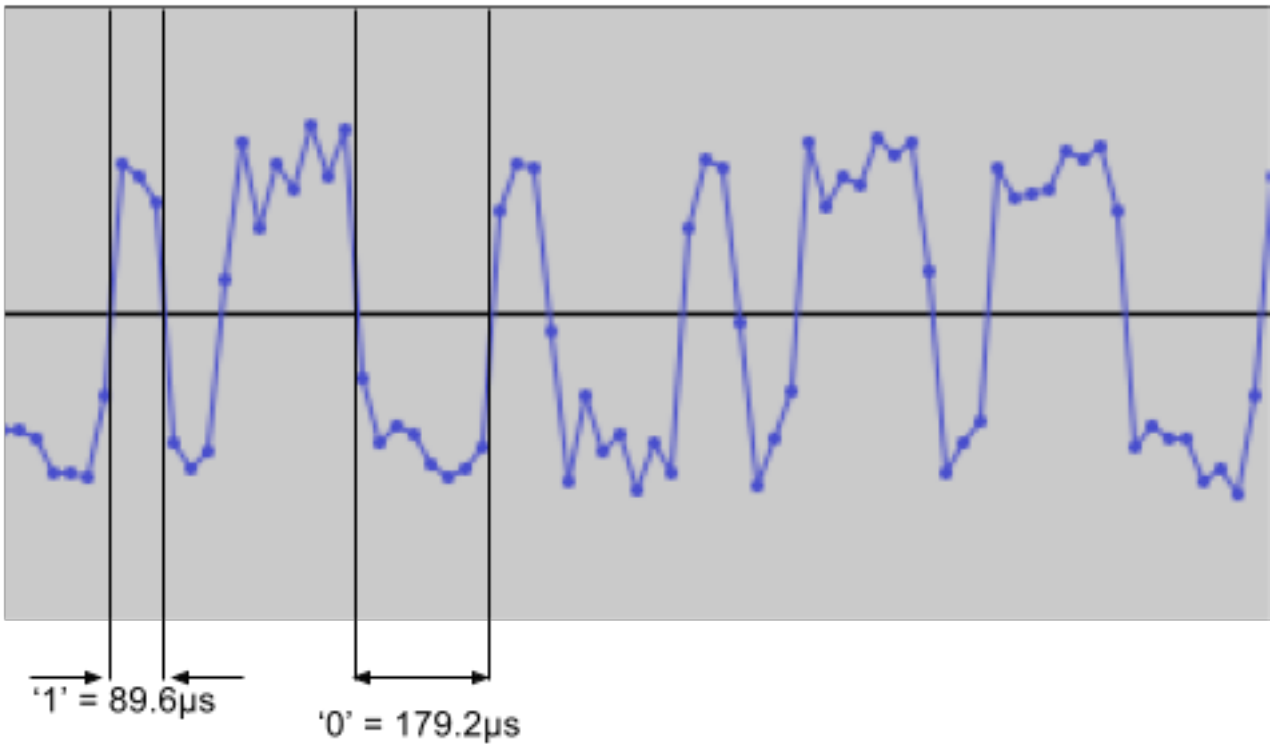
Audio data transfer (historically cassette tape data transfer) is notoriously unreliable. The FIGgyAudio format takes a number of steps to increase the reliablity:

- FIGgyAudio uses a zero-crossing technique which was found to be a relatively reliable and amazingly simple method during the 1980s. Here the value of each bit of data is conveyed by the time period between +1v to -1v voltage transitions on the audio line.
- FIGgyAudio uses a bit-biasing technique to keep the DC bias of data transitions to a minimum.
- FIGgyAudio transmits data in 64b packets with CCITT 16-bit CRC codes at the end of each packet. CRCs are an extremely reliable error checking technique compared with simple checksums or one's complement checking.
- FIGgyAudio can support scatter-loading so that load operations can be repeated and erroneous packets can be added if they are reloaded correctly.

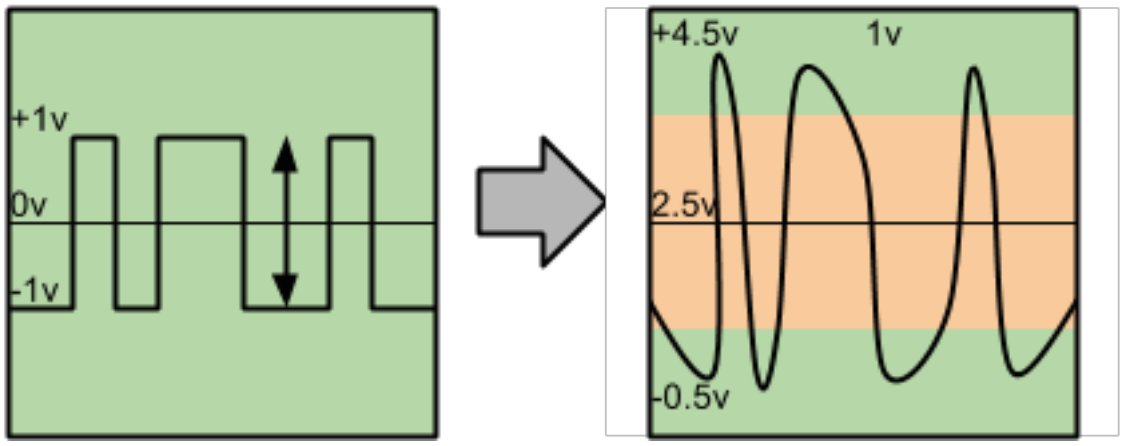
As standard FIGgyAudio can run at between 5.6KBits/s and 11KBits/s making it substantially faster than cassette transfer on home computers in the 80s and comparable with USB transfer using FIGnition's previous V-USB bootloader.

Transmitting Bits

In FIGgyAudio, all bits are transmitted as a time period between a transition from +1v on the audio line to -1v on the line. At its standard data rate, a '1' value is an 89.6μs period between a transition and a '0' value is a 179.2μs period between a transition.

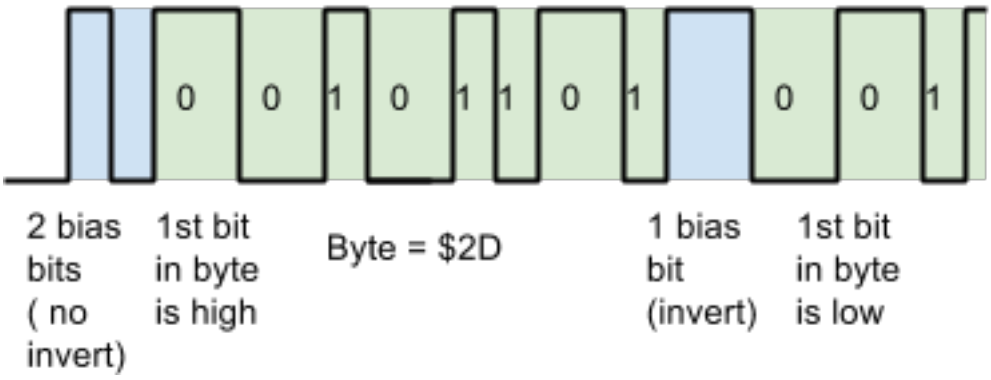


The FIGnition audio hardware consists of a capacitor and pair of resistors. On the input side, these act to roughly double the voltage transition so that a swing of +1v/-1v on the audio cable can become as large as +2v/-2v, which makes it possible to correctly read the transitions digitally.



Transmitting Bytes

A whole byte is transmitted as one or two 'bias' bits followed by 8 transitions representing bits 7 to 0 of the byte.



Bias bits are used to keep the DC bias of the signal as close as possible to 0v. The capacitor and resistors will naturally dissipate the energy from transitions over a short period of time, but will keep some of the energy in the transitions for frequencies over 20Hz or so - this is why the circuit would reproduce audio over the normal audible range.

Transmitted bytes on average should contain the same number of 0s and 1s, but over shorter periods the transitions will generate effective frequencies much lower than the bit rate. This will translate into a short-term (millisecond) bias on the input and reduce transmission reliability (as the average voltage is pulled up or down).

However, by preceding a data byte with a single bias bit we will cause all the transitions to be inverted, thus pulling down the bias voltage for the following byte and therefore increasingly reliability. However, what if we don't need to change the bias?

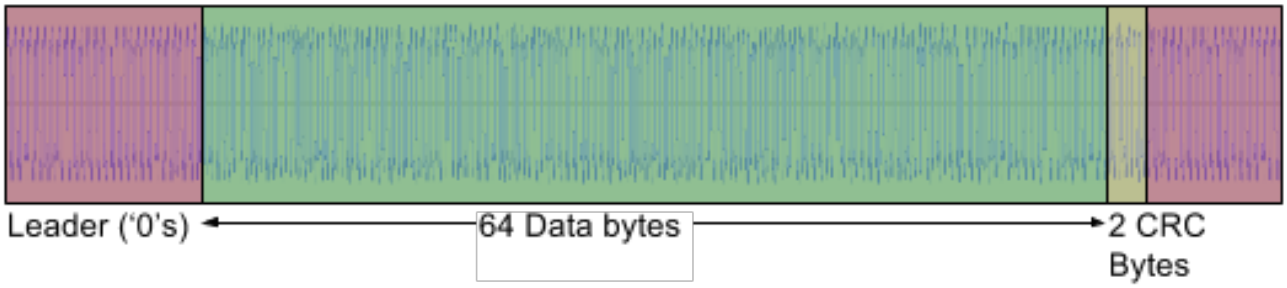
FIGgyAudio resolves this by either sending a pair of '1's (which don't change the bias for the following byte because they invert the signal twice) or a single '0' (which does change the bias). The pair of '1's take the same amount of time as a '0', thus the bias transitions at the beginning of each byte always take the same amount of time.

Note: on the decoding side, FIGnition doesn't need to consider whether transitions have been inverted by bias bits, because the bias bits themselves do the job of the inversion. It simply treats them as bits as bit 8 or bits 8 and 9 of the byte (i.e. they end up being ignored).

FIGgyAudio doesn't transmit parity bits, error correction is at the frame level.

Transmitting Frames

FIGgyAudio transmits files as a set of audio frames. Each frame always consists of a leader tone followed by 64 data bytes where the first data byte's bias will always be a pair of '1's. Finally the frame is followed by a CCITT 16-bit CRC.



The leader tone is always at least 18 '0' transitions. This is because it is impossible for this sequence to occur as part of a the data part of the frame. Here's why: a sequence of 9 '0' bits means that the bias needed to be inverted (a '0' bias) and then 8 '0' bit values were transmitted. Thus it is impossible for the next byte to contain a bias of 0 and 8 '0' data bits, because the 8 '0' data bits don't change the bias, and therefore would require a pair of '1' bias bits. In practice over 24 '0' transitions are used between frames.

A pair of '1's therefore will unambiguously denote the beginning of a data packet.

The CCITT 16-bit CRC algorithm is a fast byte-parallel algorithm everyone should use.

```
crc  = (unsigned char)(crc >> 8) | (crc << 8);
crc ^= byteValue;
crc ^= (unsigned char)(crc & 0xff) >> 4;
crc ^= (crc << 8) << 4;
crc ^= ((crc & 0xff) << 4) << 1;
```

It computes a CRC in 16c and 13 instructions on an AtMega AVR. The CCITT algorithm used here was first documented as a BSI standard algorithm for the 6800 microprocessor in the 1970s; fell into disuse in the 90s and 2000s, but has recently become more popular. The one used in FIGnition firmware 1.0.0 and later was adapted from:

http://embdev.net/articles/CRC-16-CCITT_in_AVR_Assembler

Transmitting Entire Files

FIGgyAudio transmits each file as a long leader tone of at least 3 seconds followed by a header frame; followed immediately by all the data frames in the file.

The format of a header packet is:

```
typedef struct {
     byte fileType;
     ushort len;
     ushort start;
     byte fileName[32];
     byte unused[64-37];
} tFIGgyAudioHeader;
```

The currently defined fileTypes are:

```
typedef enum {
     kFIGgyAudioNone=0,
     kFIGGyAudioUpgrade=1,
```

```
    kFIGgyAudioRAM=2,
    kFIGgyAudioExtFlash=3
} tFIGgyAudioFileTypes;
```

The `len` field is the number of data frames in the file for RAM files (providing up to 4Mb), but the number of flash blocks in the file for flash files (providing up to 32Mb).

The `start` field is the start address for RAM files (or 0 if the file should be loaded just under **top**) and is meaningless for Flash files and Upgrade files (it is set to 0 by convention).

The `fileName` should be zero-terminated and limited to 32 characters though currently there are no checks on this and the remaining bytes are undefined.

FIGnition Support For Non-RAM files

The current FIGnition firmware has no direct support for anything other than loading binary files directly into RAM. This is due to a lack of firmware space. However, because the command line can be overwritten and executed with this method it is possible to place executable Forth scripts into the command line which then 'bootstrap' other loading programs.

So, to load .fth source files directly into RAM we provide a command line which loads the following file and then sets the Forth compiler to start executing it (which compiles it).

To load files into Flash we provide a command line which loads a program into RAM which provides the means to load files into Flash.

Thus the process in either case is relatively simple for the user.

4.7: Technical: Writing Your Own Audio Data Transfer Code

Using the firmware kern vectors it is possible to write custom audio data transfer code. It is possible to record audio data directly to a computer and be processed by FIGgyTape or record directly onto an audio device for later translation or playback to a FIGnition. Custom audio data transfer code is used to support saving and loading to external flash memory as well as upgrading to new firmware images.

At the time of writing only data transfer to FIGgyTape has been tested. A FIGgyTape recording is currently limited to 128Kb of transition data (i.e. a raw series of '0' or '1' transitions).

Defining Kern Vectors. To use the kern vectors properly you'll need to obtain access to them. The complete set of kern vectors are defined in section 4.5, you don't need all of them.

Saving Data. Custom code for saving audio data should follow this kind of pattern:

```
start len type audioOutHeader ( setup header & driver info & start leader tone)
1 >tape ( send the header frame)
len 0 do
    ( prepare x frames of data and store in internal RAM at addr)
    addr x >tape
```

```
loop
( drop temporary data from data stack)
0 />tape
```

A simple example can be seen in the code for saving external Flash:

```
: save" ( blk len div , name --)
  >r 2dup kExtFlash r> audioOutHeader ( blk len /blk len type div/ amp iSrc )
  1 >tape ( blk len amp )
  swap 0 do
    over i + blk> ( read block from flash, will copy to vram)
    vram blkPages >tape ( write 8 frames)
  loop
  drop drop 0 />tape
;
```

The tape saving system in `>tape` is a blocking routine that uses about 50% of CPU and runs for between 6.1ms and 12.2ms. If your code has hard real time constraints for generating data that exceed these parameters then you'll need to use the user-interrupt facility to provide data in the background. Note: if you do this, your user-interrupt must run for no more than about 400μs, which is between 80 and 320 Forth byte codes in flkr or video off mode).

However, the tape saving system doesn't force hard real-time constraints on you. At the end of `>tape` when the last byte has been sent (which will be the crc followed by 3 leader bytes), the leader tone will continue to be sent automatically at roughly 2.7KHz (or every 5.7 video scans). This means that it should be possible to turn the video back on and perform a number of other operations even with user interaction before saving the next block.

Even though the first command to `audioOutHeader` sets up the information for a header block this doesn't mean you're forced to know in advance how much you want to save and save everything as a single file (though it's simpler that way). You could, for example, set up a means of sending everything in short files of a known number of frames and at the end of each set of frames, send a longer lead tone; construct your own packet header with another known set of frames and simply send it using `>tape` to begin the next 'file'. In this way, you won't force the user to pause and restart playing on the audio device (which `audioOutHeader` does). The header frame structure is given in section 4.6 / Transmitting Entire Files.

Loading Data. Custom code for loading audio data always follows this kind of pattern:

```
rate +tape> ( starts flicker mode at dataRate=rate and returns next header addr)
dup ic@ kExtFlash = if
     dup 1+ i@ ( len )
     swap 3 + i@ ( start) swap 0 do
          i tape> ( return 0 for a bad frame, addr otherwise)
          ?dup if
               ( process the frame)
          then
     len
then
```

```
totalPacketsReceived -tape ( stop tape, report any error)
```

`tape>` blocks execution until a complete frame has been received (and then checks its crc and updates its loadMap bit). However, the frame receive code is interrupt-driven and can receive an entire packet + crc in the background. This means that although `tape>` does impose real-time constraints, your code has approximately 6.1ms to process the previous packet of data. In flkr and video off mode this represents between 80 and 320 Forth byte codes per byte of data you need to process.

The external Flash loader is a good example of a worst-case scenario. The **>blk** routine to save external RAM to flash is not real-time at all, because searching for blocks can take a wide span of time and saving a block may involve purging the external Flash of dirty blocks. Therefore the external Flash loader prepares all the saving operations in the background *before* the actual data is loaded. The FIGgyTape application provides the external Flash loader with information about the number of blocks needed (and the starting block is provided by the user as part of the command line).

Depending on the application, it may be that you need to modify FIGgyTape or produce a custom `.wav` generator to be able to send data in the right way for your FIGnition code to load it.

Flkr Mode. Flicker mode is a video mode designed to provide information to the user - you don't have to use it to save or load data. You could for example turn the video off completely using the fast command:

```
: fast 129 $43 ic! ;
```

And then when you need to go back into a proper video mode execute `0 vmode` or `1 vmode`. Note: you can't have video on when loading data, you must either turn it off or use `flkr` mode.

Crc Generator. The crc generator is available at **kern** vector 12. The firmware code uses it to check crcs (in `tape>`) or generate crcs (in `>tape`) for individual audio frames, but it can work for any block of data. Using the crc code follows the pattern:

```
-1 ( prime the crc with the value -1)
endAddr startAddr do
     i ic@ ( or c@, or perhaps it's just computed data)
     swap crc
loop
```

The crc is left on the top of the stack. FIGnition's crc algorithm as described earlier is very fast - much faster than bit-oriented algorithms, at the equivalent of 1cycle per bit (ignoring call/return).

5: FIGnition Floating-Point Arithmetic

Firmware 1.0.0 provides a small, but useful set of single-precision floating point arithmetic commands. You can:

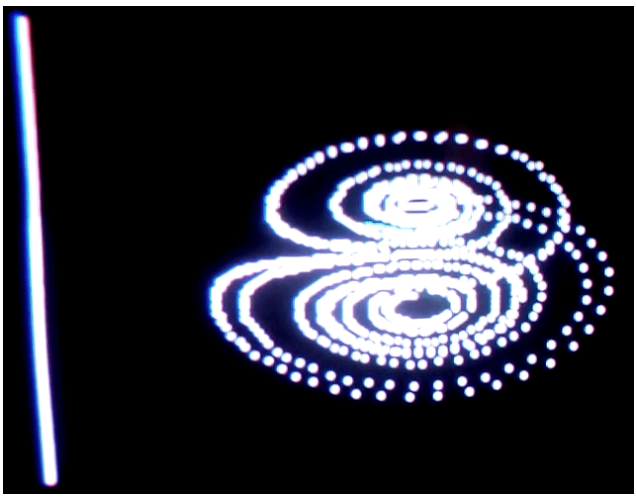
- Enter floating point numbers directly on the command line and FIGnition will parse them correctly (you can also convert floating point strings to values using **number**).

- Convert floating point values back into strings using **#f** or display them using **f.** (floating-point values are always displayed in scientific notation).
- Perform basic floating point arithmetic using **f+** , **f-**, f*, **f/** and **fneg**.
- Convert between 32-bit integer and floating point values using **fint** and **float**.
- Compare floating point values (and 32-bit integers) using **d0<** and **f<** .
- Manage floating point values on the data and return stacks using **2dup**, **2drop**, **2over**, **2swap**, **d>r** and **dr>** (you can use these commands to manage the top two items on the stack even if they don't contain floating-point numbers).
- Test for overflow (the value will be 1d)

Some examples follow:

```
: dconst
  create -1 allot
  71 c, d,
; immediate

3.1415927 dconst pi

: sphere ( r )
     2dup 2dup f* f*
     1.3333333 f* pi f*
; ( r^3*4/3*pi )

: lozEqDt ( dt x y z )
 [ -8.0 3.0 f/ ] dliteral
 2over f* 2swap d>r ( dxy[-bz]:z)
 d>r 2over 2over f* ( dxy[xy]:[-bz]z)
 dr> f+ dr> 2swap d>r ( dxyz:Z)
 fneg 28.0 f+ d>r ( dxy:[r-z]Z)
 2over dr> f* 2over f- ( dxy[[x*[r-z]]-y]:Z)
 d>r f- -10.0 f* ( d[-s[x-y]]:YZ)
 2over f* 2swap ( [dX]d:YZ)
 dr> 2over f* 2swap ( d x' y' d:Z)
 dr> f*
; ( Dt x' y' z' )

: lozSum ( dxXYyZz)
  f+ d>r f+ d>r f+ ( dx':y'z')
  dr> dr>
;

2048 bytes history
0 var his

: mplot ( xy)
  2dup 1 pen plot
```

```
  his @ >r
  r history ( xyh^:h)
  dup c@ over 1+ c@
  2 pen plot ( xyh^)
  swap over 1+ c! c!
  r> 2+ 2047 and his !
;

: .loz ( Dt x0 y0 z0 --)
    0 history 2048 0 fill
     1 vmode cls
     begin
          2over 2.2 f* ( dxyz[y*2.2])
          fint drop 80 + >r ( dxyz:py)
          2dup 2.5 f* ( dxyz[z*2.5]:py)
          fint drop 5 + r> ( dxyz/[pz][py]/plot )
          ( cr swap . swap . sp i@ . )
          mplot
          d>r d>r 2over ( dxd:yz)
          2over dr> dr>  ( dxdxyz)
          2over 2over d>r d>r ( dxdxyz:yz)
          lozEqDt ( dxXYZ:yz)
          dr> 2swap dr> ( dxXYyZz)
          lozSum ( dx'y'z)
     inkey 32 = until
     2drop 2drop 2drop 2drop
     0 vmode
;

: bigFib
     1.0 1.0
     begin
          2dup f.
          2swap 2over f+
     2dup 3.0e38 f< 0= until
     2drop 2drop
;

: fpDemo
  pi f. ( 3.141592e+00 )
  key drop cr

  6371.0 sphere f. ( 1.083206e+12 Km3)
  key drop cr

  0.01 1.0 0.0 0.0 .loz

  bigFib
;
```



(Lorenz Attractor demo)

Handling Overflow. Overflow occurs when floating-point calculations exceed **±3.402822e+38** . FIGnition doesn't trap overflow by throwing an error, but instead by returning a result of **1d** and any floating point arithmetic operations applied to inputs that have overflowed always return the overflow value. For example: `1d 0.0 f*` is still overflow. The following command sequence returns **-1** if overflows has occurred:

```
1d d- or 0=
```

This means that it's possible to perform a whole sequence of operations and only have to test for overflow at the end:

```
1.7e38 sphere ( returns overflow)
1.0e26 f/ 9.5 f+ 16 float f- fneg ( still returns overflow)
2dup 1d d- or 0= ( tests for overflow)

: fabs $7FFF and ;
-1.345 fabs f.
```

5.1: Command Reference.

Stack Inputs (: Parameter Stack Inputs)	Command	Post-Command inputs	Stack Effect	Action
Floating-Point number conversion #f works in conjunction with <# , sign and #> to convert floating point values into scientific notation, though it doesn't really have any flexibility. f. merely makes the input absolute whilst storing the sign on the stack in the manner expected by sign. #> returns the address and length as for normal number conversion. However, this does mean that it can be used for simply converting to strings and then manipulating the result as a string.				
fa fb	f+		fa+fb	Adds the two floating-point inputs fa and fb.
fa fb	f-		fa-fb	Subtracts the floating-point input fb from fa.
fa fb	f*		fa*fb	Multiplies the two floating-point inputs fa and fb

Stack Inputs (: Parameter Stack Inputs)	Command	Post-Command inputs	Stack Effect	Action
fa fb	f/		fa/fb	Divides the floating-point input fa by fb.
fa	fneg		-fa	Negates the floating-point input fa.
fa	fint		al aH	Converts the floating-point input fa into a signed 32-bit integer in the range -2^31 to 2^31-1. Values are rounded to 0. Numbers outside the correct range return incorrect values.
al aH	float		f=	Converts the double number a into a floating-point value fa. Only the most significant 24 bits of the input are converted accurately.
fa fb	f<		fa<fb?	Returns a single value -1 if fa<fb or 0 otherwise.
fA addr^	#fd		fA' addr'	Given fA in the range 0.0 to 9.99999 converts the integer portion of fA into a digit character stored in addr (which is incremented); returns the fractional addr of fA scaled by 10.0.
fA	#f		fA'	Converts the absolute floating-point value in fA into its text representation in the form d.dddddd±eEE into the pad. Returns the remainder.
fa	f.			Displays fa as a signed floating-point value in scientific notation.
text^	number		fa -1	If text^ contains a floating-point string, returns the floating-point value and -1 to denote floating-point.

Supporting Double-Number Commands.

A number of double-number commands (which also work with floating-point values) have been added to support floating-point arithmetic operations.

In addition, byte code 71 performs a floating-point constant fetch operation and can be used to create a dconst command:

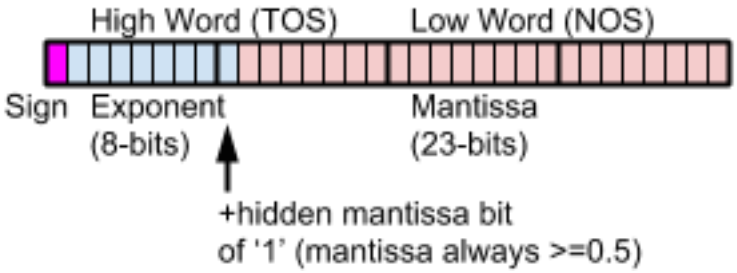
```
: dconst create -1 allot 71 c, d, ; immediate

3.1415927 dconst pi
pi f. ( displays 3.141593e+00 OK )
```

al aH	2drop			Drops a double or floating point number from the stack.
al aH bl bH	2over		al aH bl bH al aH	Copies the previous double or floating point number to the top of the stack.
al aH bl bH	2swap		bl bH al aH	Swaps the top two double or floating point numbers on the stack.
al aH	d>r		: al aH	Moves the double or floating point number at the top of the stack to the return stack.

Stack Inputs (: Parameter Stack Inputs)	Command	Post-Command inputs	Stack Effect	Action
: al aH	dr>		al aH	Moves the double or floating point number on the top of the return stack to the data stack.
al aH	d,			Inserts the double or floating point number on the top of the stack into the dictionary at **here**.
addr	d@		al aH	Reads a double or floating point number from external RAM at addr.
al aH addr	d!			Stores the double or floating-point number a in external RAM at address addr.

5.2: Technical: FIGnition FP Format and Performance

FIGnition Floating-point arithmetic is based on IEE754 single-precision arithmetic. It has a mantissa sign-bit, followed by an 8-bit biased exponent, followed by a 23-bit mantissa producing a range 0.5 to 0.99999994 (in steps of 5.96×10^{-8}) .



A floating-point value is: sign*mantissa*2^(exponent-127).

FIGnition Floating-point arithmetic differs from IEE754 single-precision in the following ways:

- FIGnition Floating-point arithmetic only handles a generalized 'overflow' condition rather than ±∞ and NaNs when the floating-point range is exceeded.
- FIGnition Floating-point arithmetic doesn't handle denormalised numbers, results whose mantissas are < 5.87747e-39 are simply 0.0.

The following technical specifications are valid:

- MaxFloat = 3.402822e+38, which is $7FFFFFFFd .
- MinNormalisedFloat = 5.87747e-39 which is $800000d
- f* calculates mantissas to 48 significant binary digits before rounding.
- f/, f+ and f- calculates mantissa results to 32 significant binary digits before rounding.
- Rounding always rounds up the mantissa if the 24th most significant digit is 1.
- Floating point performance is on average 14.8KFlops.

The benchtest used to establish floating point performance is:

```
: bm1f
```

```
    10000 0 do
        10.9 9.8 f+ 7.6 f- 5.4 f* 3.2 f/
        2drop
    loop
;
```

The time taken is 3.3s of which 0.5s is occupied by pushing floating point values, 2drop and the loop operation. Thus 2.7s are spent actually calculating: the performance is 14.8KFlops.

6: System Modifications

A few changes have been made to the system. A number of **kern** functions have been added to support audio data transfer (documented in section 4.5).

The following additional changes have been made.

6.1: SysVars and User variables Changes

Internal RAM system variables 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)
12	stackFrame	byte*	The current base address for the persistent stack frame used by l> and >l (as well as locs and loc; for setting up and releasing persistent stack frames).
14	clipLeft	byte	The LHS pixel coordinate for the clip rectangle
15	clipRight	byte	The RHS pixel coordinate for the clip rectangle.
16	clipBot	byte	The bottom pixel coordinate for the clip rectangle.
17	savedX	byte	The secondary pen position x coordinate (used in 2blt).
18	savedY	byte	The secondary pen position y coordinate (used in 2blt).
19	swUartCh	byte	The Software Uart's current character.
20	swUartState	byte	The Software Uart's current state.
21	userIntVec	byte*	The address in external RAM of the user interrupt vector.
23	**userIntFlags**	**byte[4]**	**A Bit array of interrupts that have been triggered since the previous user interrupt routine was executed. In Firmware 0.9.9 it used 3 bytes, but it turned out 4 bytes were needed.**

Offset	Name	Type	Purpose
24	**key**	**byte**	**The key code returned by the keyboard driver is now a system variable.**
25	**helpCount**	**char**	**The helpCount used by the keyboard driver is available.**

A number of changes were required for the Forth User variables in order to support Command Line execution from the audio system. They are now:

Offset	Name	Type	Purpose
0	current	word^	Contains the address of the **latest** definition. Current linkage now uses single indirection rather than double-indirection.
2	warning	word	Unused. It was part of the error system, but isn't needed.
4	marker	byte^	The address of the text marker used in the editor.
6	top	byte^	The address of the top of free external RAM.
8	blk*	byte^	The address of the flash block allocated by blk> and >blk.
12	blk#	word	The value of the current block used by load.
14	fparse	word	Unused
16	dp	word	The dictionary pointer, points to the first free byte of external RAM. **dp @** is **here** .
18	state	word	The compiler state, 0 for interpret, $40 for compile mode.
20	base	word	The number base when converting between strings and numbers.
22	hld	byte^	The pointer to the current character when converting numbers to strings.
24	tib	byte^	A pointer to the terminal input buffer.
26	unused	byte[8]	Unused

6.2: Software Serial Out Change

The Software serial out has been moved to PortD5 which means the correct code for using it is now:

Code	Comments

`$2B const PORTD`
`$2A const DDRD`
`$44 const TCCR0A`
`$45 const TCCR0B`
`$46 const TCNT0`
`$48 const OCR0B`
`$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.5; 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`
`  $20 $FF 2dup PORTD`
`  >port> drop`
`  DDRD >port> drop`
`  0 TCCR0B ic!`
`  0 TIFR0 ic!`
`  4 $FF TIMSK0`
`  >port> drop`
`;`
	Initializes the software serial out. Sets PORTD.6 to output 1. Stops Timer0 and enables the OC0B interrupt.

`: swUartEmit ( ch --)`
`  begin`
`  TCCR0B ic@ 0= until`
`  gSwUartCh ic!`
`  11 gSwUartState ic!`
`  32 OCR0B ic!`
`  $20 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 free-running mode which Clears 0C0B on compare match, Resets the Timer0 counter and finally starts Timer0, running at clk/64, which is 312.5KHz.

6.3: Flash Disk Searching.
In version 1.0.0 the algorithm for mapping virtual disk blocks to physical flash pages has been redesigned and the key routine has been made much faster. This means that disk operations in 1.0.0 are back to the same speeds as for Firmware 0.9.8 and earlier.

6.4: FIGgyPad interrupts
The keyboard scanning is always called from the video interrupt code, at about 100Hz (i.e. a single row scan every 10ms).

In earlier Firmware revisions the video interrupt code had to push every caller-saved register, about 17 registers in total. This added 3.4μs to the interrupt. The reason it had to be done was because Interrupts calling conventions in Gcc on an AVR always push all of these registers if you call a sub-function in the interrupts handler. Since I didn't want this overhead every time the video interrupt was called, I shifted it to a special routine that only

does all of this when the keyboard scanning happens. It's more efficient that way, but very ugly.

In Firmware 1.0.0, I added a tool which could analyse and entire set of registers that would need to be pushed when calling a function to make it register-safe.

The concept and a critique of GCC's caller-saving register convention is covered more fully in this blog entry:

http://oneweekwonder.blogspot.co.uk/2014/03/caller-convention-calamities.html