fJACK - a FORTH jack audio connection kit client

Hanno Schwalm

June 11, 2017

Contents

0.1 Hallo, dear FORTH-audio friend!

This is a about FORTH programs doing real time audio processing. We don't write hardware drivers for the AD/DA but use standard libraries to access the hardware. Modern sound systems use callback driven models, ALSA and Pulseaudio are examples. But both are not portable and are not designed for professional audio environments.

One popular system targeting this is the 'jack-audio-connection-kit' JACK. It is available for all major desktop platforms Linux, OSX and Windows. It makes use of the hardware driver in a realtime process and can be used with hardare on the motherboard, connected via USB, firewire or even 'over the net'. There are many audio applications around using the JACK interface and Forth can now connect to **all** of them.

fJACK is a client for JACK; it's written in FORTH and uses advanced techniques like threading, access to external libraries and callbacks. So it is **not** ANS Forth. The good news: it's available for all current iForth and VFX Forth (Intel & ARM) versions.

fJACK is a pretty large and complex system – don't get frustrated. Read the example sources and feel free to send me an email, hanno@schwalm-bremen.de

Chapter 1

Installation

1.1 Installing a JACK system and get it running.

Read the HOWTO in the net and specialized audio groups to find out how you can get the most out of it. Getting JACK running with low latency and no drops/xruns is sometimes a bit difficult. Look out for hardware that is well supported for your environment.

It used to be necessary to use a special realtime-kernel, current mainline linux kernels with optimized settings work sufficiently in most cases. On my Sandybridge generation desktop i find latencies of ~10ms for my Focusrite 6i6 interface or ~20ms on the motherboard audio without any xruns using the out-of-.the box Fedora 25.

1.2 Downloading jack

The first place to look for jack is: http://jackaudio.org/download or go for precompiled packages for your system. All linux distributions have rpm or deb packages available.

1.3 What sort of JACK?

You should always use the newer jack2 system which is available for all linux distributions, windows and OSX, the current version is 1.9.10.

In 2017 most computers run a 64bit operating system so the jack server is a 64bit application too. When you want to run a 32bit jack client you need support for this **mixed mode** withing the jack server. The mixed mode had problems related to a jack2 bug related to the way futexes were used (fixed in jack2/git).

When you observe endless xruns for your 32bit client you will probably have to recompile your jack system using the git code. I use the code compiled with:

```
./waf configure -prefix=/usr -mixed -libdir=/usr/lib64 -alsa -firewire -dbus
```

You have to find out where your jack libraries are on your system and you might have to modify the source in `fjack/jacklib.frt` accordingly.

1.4 How is fJACK made running?

All newer JACK versions (since 0.118 or 1.9.4) have the ability to auto-start a jack server when a client wants to register and no running server is found. But - this is not a good way to start a jack server at all! (See comments on jackaudio.org about this.)

fJACK can autostart but it's much better to use controlling applications like QjackCtl - on linux i use **cadence** from KXStudio. Both tools make sure that other sound systems like pulseaudio are easy to integrate.

So you can choose between three options:

- Start the jack server before using fJACK with a tool like qjackctl. This allows a precise setup of jack according to your system and can be done when you log-in. I think this is the best way. Also it allows

session management, starting fJACK turnkey applications and other audio application with one click and connecting them all. Current linux distributions support starting jack using the DBUS interface.

- Let fJACK start the server automatically. This needs a correct setup of JACK and you will have to set `JACK-AUTOSTARTING?` to true in the config file.

- Write your special `KICKSTART` word before loading fJACK. This method is the least preferable and will probably be removed when JACKs Windows integration has improved.

1.5 What about the fftw3 library?

All recent fJACK versions use the fftw3 library to define the filter response of FIR filters so you should install it. `BAND-PASS BAND-STOP` and `USER-FIR` filters with a user-specified frequency response and the `EQUALIZER` will not work without it. Calculating the filter parameters is not a real performance problem. But doing signal analysis on large amount of data or with sizes <> 2^n depends on a 'calibrated' fftw3 system. Currently creating fftw3 plans uses `#FFTW_ESTIMATE` because in fjack only 2^n sizes are used and then there not much of a difference. See fftw3lib.frt and fftw3.frt

1.6 What about performance?

fJACK 3.0 can be used on three very different platforms, also the supported Forth compilers differ a lot.

On iForth 64bit system we have a lot of ram available and a **very** good floating point compiler. Also the timing critical FIR filters have optimized assembler code using SSE2 instructions. On CPUs with a small cache or slow RAM access you might want to decrease the sie of the history buffers.

In intel vfx platforms ths situation is slightly different. We also have the SSE2 filter and the Ndp387.fth is even slightly faster than the iForth generated code but is restricted to a FP stack depth of 8. The fjack modules use floating data up to a depth of 5, so beware! The Hfp387.fth module has a much worse performance as it does not optimize well, also there are severe problems with external libraries and callbacks. fJACK does **not** support it.

A - for me completely new experience - was working on ARM single board systems. I got myself a ARTIK710 system, it uses a comparably fast CPU running 8 Cortex A53 cores. The Vfx ARM Linux compiler generates good integer code but the floating point performance is pretty bad because the code is almost not optimized.

What can we do to tune performance on these small boards? The most effective tuning will be done by decreasing the audio sampling rate!

Many ARM cpus have very limited ram cache, so it will be good to decrease the history buffer size (see the config options on how to do that). FIR filters need a lot of FP maths, use IIR filters instead until we have NEON optimised code or reduce the number of filter points. Let's hope vfx works on the floating point compiler!

There is an option *\{JACK-EMBEDDED?}, it will set the defaults to values more appropriate on these boards.

1.7 What about MIDI?

fJACK supports the jack MIDI system - see the MIDI section for details.

Bruno Degazio wrote the Karplus-Pluck synthesizer for fJACK which can be used as a MIDI instrument. Another hammond-like synthesizer is also available. Both use the fjack/instrument framework. This is a new feature and the instrument API is not yet fixed, so only a few words are in the FORTH vocabulary yet. Read the source and ask via email if you want to dig into this.

1.8 What about memory?

fJACK needs quite a lot of ram allocated from the system. As this is a realtime application, all memory **must** be permanent and **MUST NOT** be swapped in/out. When using the default config parameters fJACK needs ~10MB of allocated memory. The biggest 'memory-grabber' is the history mechanism.

See the 'What can be configured in fJACK?' chapter to find out more about changing such config options.

1.9 Contact, help & bug reports

Feel free to ask for help at hanno@schwalm-bremen.de - normally i will respond in a few days.

Whenever you plan to use fJACK in commercial applications read the Licence.txt and send me an email. Using fJACK in private or educational settings is free of charge on all platforms.

1.10 Where are the fJACK sources?

They are all inside the **fjack** directory containing the client and extension module sources plus the 'loader' **fjack.frt**. The fjack/sysfiles directory holds the connections and some binary files. This subdirectory should be writable.

All iForth versions are supported. The fJACK sources are inside the **include** directory. `IN fjack` will do.

On **VFX Forth** systems the fJACK sources are in the **examples** directory. You should **cd** to this directory, then `INCLUDE fjack.frt` will do. When you get an fjack update you could replace that version or have it somewhere else.

The release history file is fjack/history.txt

1.11 Testing the client

Loading as above compiles the basic Forth interface to jack audio, defines a jack-client with some ports, connects these ports and plays some tones via the karplus instrument.

There are some demos availabe, see fjack/examples/tests.frt or fjack/examples/board.frt.

When you **don't want the whole fJACK** stuff included you have some configure options, see below.

iForth systems have a realtime capable graphics user interface, the gui/graftool system has some plugins that can be used to control filter settings and more for the analyser plugin fjack/fft.frt and the visual frontend gui/showfft.frt

As fully working example applications we have gui/studio and the radio/swlapp applications.

1.12 Making fJACK Turnkey Applications

fJACK can be used in turnkey applications, all plugin object structs and the client data are in permanent memory space to support this. fJACK inside a turnkey applications will start automatically, you will just have to put your application 'xt' in the boot chain.

```
' application AtCold
```

Stopping the client and system housekeeping is done in the `AtExit`, `AtCold` and fJACKs internal chain.

vfx produces 'complete' binaries, iForth turnkeys using `SAVE-SYSTEM` are fully supported, see gui/studio.frt as an example.

1.13 Removing the fJACK client

On all fJACK systems we have `-fjack ( - )`. It works like a `MARKER`, it stops the client and background processing, frees all allocated memory and removes the fJACK system from forth dictionary space.

Chapter 2

The fJACK Client User Manual

`VOCABULARY JACKAUDIO`

The **`JACKAUDIO`** vocabulary holds internals of the fJACK system, it's words could be changed in next releases and should be used in end-user applications with care.

2.1 A short processing overview

fJACK does real-time processing on a user defined number of audio channels (ports) **`JACK-CHANNELS`**, it controls and is controlled by a running jack server. You can connect fJACK with other jack-ready application via these ports. To allow audio data processing you can define audio plugins and put them in any of the slots. So the basic overview is like shown in **fjack/io_slots.png**

Please recognize that every processing plugin has it's own history buffer. Later you will find out, that **every** plugin can indeed read data from **any** history buffer. This sheme allows much improved sound effects and mixing.

2.2 Configuring fJACK

There are some configure options defined in **fjack/config.frt**. All these configure options are defined by a **`cCONSTANT`** definer; it's used like a **`CONSTANT`** but it looks for an existing definition having the same name. If such a definition exists, it **won't** define a new constant but will use your predefined constant instead. When using a predefined constant you will get a notice.

So - let's say you want to use fJACK on stage with a powerful computer having a 12-channel audio hardware and use echoes for ~3seconds. You will have to define **`JACK-CHANNELS`** and **`#HISTORY-SIZE`** **before** INCLUDE fjack.frt

```
16 CONSTANT JACK-CHANNELS
$20000 CONSTANT #HISTORY-SIZE
```

This way you may configure your fJACK application without changing the fJACK sources itself.

2.3 What can be configured?

`ARMLINUX? cCONSTANT JACK-EMBEDDED?`

When TRUE some defaults will be set TO appropriate values. For ARM systems this currently defaults to TRUE.

`JACK-EMBEDDED? [IF] #2 [ELSE] #4 [THEN] cCONSTANT JACK-CHANNELS`

It can be set to any number >= 1 you like. The default is 4, when you have a powerful computer and wish to use more audio channels, just set this to 8 or whatever you need. Each channel has one input and one output port defined. These ports are used for the audio processing plugins described later. There is also a programmers interface to be used by you to define your own sound processing tools. Also think about using other jack plugins available in the net to be used by your forth application. Just one hint - more channels than needed will use a lot of memory and slow down the system a bit.

`#32 cCONSTANT JACK-IO-SLOTS`

Defines the maximum number of processing plugins that can be placed in the Input-Port->Output-Port processing chain for a channel. Also see **#PLUGINS**

`#64 cCONSTANT #PLUGINS`

The maximum number of plugins that can be used at the same time in the input->output processing chain.

`FALSE cCONSTANT JACK-AUTOSTARTING?`

Should the jack server be autostarted with the client? This requires a modern jack server installed correctly.

```
:  KICKSTART ( - ) ;
```

Sometimes you might want to start the jack server in a user specified way not using any of the control applications or the autostart feature. In this case you can define a word KICKSTART before loading fjack that does so.

```
 :  KICKSTART S~ start /d "C:\xyz\Jack" /i jackd -S -d portaudio -p1024~ SYSTEM #5000 MS ;
:  JACK-LOGIN-WAVFILE ( - addr len ) S" fjack/snippets/start.wav" ;
```

This is the wavfile played when starting the fjack client. You may choose any other wav filename and define **JACK-LOGIN-WAVFILE** before loading fjack.frt

```
:  JACK-CLIENT-NAME ( - addr len ) S" fJACK" ;
```

Normally the fjack client is called fJACK. To support session management you may define another client name for your target application. To further support jack session management it is assumed here that you will have a shell script available in your iforth directory with exactly the name of the client name, this will be started when reading your session. To use another client name you'll have to define *forth{JACK-CLIENT-NAME} before loading fjack.

`TRUE cCONSTANT IO-MODULES?`

Load the plugin modules? Without these modules no filtering is possible!

`TRUE cCONSTANT WAV-SUPPORT?`

Load the wav support module?

`TRUE cCONSTANT JACK-MIDI?`

Adds JACK-MIDI ports to the client and includes basic MIDI commands.

`TRUE cCONSTANT MIDI-INSTRUMENTS?`

Include fjacks MIDI instruments

`FALSE cCONSTANT MIDI-DEMO?`

You may set this to FALSE when you don't want to play the demo when starting the client

`JACK-EMBEDDED? [IF] #512 [ELSE] #2048 [THEN] cCONSTANT #FIR_BUFF`

This defines the maximum number of taps in adaptive and fir filters; every defined FIR filter object uses 5*#FIR_BUFF DFLOATS dynamic memory. Must be a power of 2 and at least 4. This is quite a good default as normal fir filters won't need larger buffers. When using special effect plugins using the generic fir filter routines you might need larger buffers.

`#16 cCONSTANT #jack-fifos`

Gapless wav playing and writing is controlled via a FIFO with **#jack-fifos** elements. If an applications writes small chunks of wav data at very high speed you could increase this. More information is available in the wav handling chapter. Please note, it must be a power of 2

`$800 cCONSTANT #max-frame-size`

The maximum frame size suported. Only with very slow hardware you will want larger buffers as the latency gets higher. #1024 is the default for the current jack versions so it must be at least $400. Please note, it must be a power of 2

`JACK-EMBEDDED? [IF] $2000 [ELSE] $10000 [THEN] cCONSTANT #HISTORY-SIZE`

fJACK may want access to data played some time before for delaying or data analysis so data are saved in this circular buffer. This history is available in every plugin. The mixer and delay plugins both need the history size to be large enough. Please note; it must be a power of 2 and at least 4 * #max-frame-size

`FALSE cCONSTANT AUTO-HISTORY-CLEAR?`

Whenever a plugin object is put into the processing chain the history might be cleared automatically. I don't think it's worth to set this to TRUE but it's your decision.

`1 cCONSTANT DEFAULT-FIR-WINDOW`

The coefficients of FIR filters are calculated by a windowed inverse FFT, this windowing function can 0,1 or 2. See **SET-FIR-WINDOWING** for more information.

`JACK-EMBEDDED? [IF] #8 [ELSE] #16 [THEN] cCONSTANT JACK-BOARDJOBS`

The number of wavboard jobs to be used at the same time. Must be at least one

`FALSE cCONSTANT JACK-MIDI-DEBUG?`

When TRUE midi in/out info is written to the messages buffer; realtime safe

`FALSE cCONSTANT JACK-DEMOS?`

When TRUE fjack will load the demos.

`FALSE cCONSTANT JACK-FORTH-FIR?`

When TRUE the FIR filters will e forced to use the standard floating point code instead of the SSE2 optimised assembler code. This should be done when you use a cpu without SSE2 instructions.

2.4 Datatypes used in the client

Datatype implementation details might change in future releases. Currently the object-makers all return object-pointers or are defined as **CONSTANT**s. Please read the fjack/types.frt to understand how it works.

2.5 Using the Client

`0 VALUE JACK-POSITION ( - n )`

The sample position in the audio data stream shared by all clients, it's an unsigned 32bit integer. Read-Only!

`0 VALUE #FJACK-SAMPLE ( - n )`

In the input->output processing chain you can use this value as the sample number. It is derived from **JACK-POSITION**. Read-Only in processing chain!

`1E0 FVALUE PLAY-WAV-LEVEL`

The level of the WAV data played common for all channels. The default of 1.0 is ok in most situations, when you have wav files that are a) 8bit-coded b) on almost full level and c) need resampling you might set this to a smaller value like 0.9 to avoid distortions.

`1E0 FVALUE READ-WAV-LEVEL`

The level of the recorded=read WAV data common for all channels.

`1E0 FVALUE JACK-IO-LEVEL`

This level is applied to data read at the InputPort before the Input->Output processing is done, it's common for all channels. The leveler plugin uses it as an example.

`#10000 VALUE JACK-SAMPLE-RATE ( - n )`

The sample rate the JACK server is running. The fJACK client is told about it via a callback, you can - and should - only read it.

`#1024 VALUE JACK-CHKSIZE ( - n )`

Number of samples in a **process frames callback** as returned by the JACK server.

`0 VALUE JACK-XRUNS ( - n )`

Counter for the xrun callbacks

`0 VALUE fJACK-ID ( - n )`

The client id returned by the JACK library call when registering a client is kept here.

`: JACK-LOAD ( - n )`

The JACK server measures the time spent in the clients callback by using the high-resolution-timer and calculates the 'load' as it also knows the time between callbacks. The value n is the load in percent.

`: JACK-RUNNING? ( - flag )`

To be used by the user application testing both the client id and the state given by the processing callback. **TRUE** when the client is really active.

`: JACK-ZOMBIE? ( - flag )`

This flag is TRUE when the fjack client was zombiefied by the server.

`0 VALUE JACK-SESSION-BYE?`

This is a global value that can be tested by user applications. It is set to **TRUE** when a jack session manager like QjackCtl tells save&quit. As there is no global safe way to leave all apllications this is done via this flag.

`DEFER GRAPH-REORDER-HOOK ( - ) ' NOOP IS GRAPH-REORDER-HOOK`

Called by the reorder callback; you can define a function watching the jack environment, for example you could make sure about connections.

`DEFER PORT-CONNECT-HOOK ( a b flag - ) ' 3DROP IS PORT-CONNECT-HOOK`

Called be the port-connect callback; you can define a function that looks for connections. This function must take the flag and both port id's from the stack

`DEFER SAVE-SESSION-DATA ( event - ) ' DROP IS SAVE-SESSION-DATA`

The session management of the jack server allows saving of data, this is a hook for your user applications, See **DEFAULT-SESSION-SAVER** as an example.

`: SET-CHANNEL-VOLUME \ ( channel - ) (F level - )`

Besides setting the global volume you may set a volume to a specified output channel by this function.

`: GET-CHANNEL-VOLUME \ ( channel - ) (F - level )`

reads the channels output volume.

2.6 Managing the history buffers

`: OBJECT-HISTORY ( plugin-object|channel - addr )`

To use **@HISTORY-DATA**, **MOVE-HISTORY-DATA**, **@IDX-HISTORY-DATA**, **RECONNECT-PLUGIN** or **@INTERPOLATED-HISTORY-DATA** you need the history buffers base address, this calculates it for you. The parameter can be either the channel getting the inport history buffer or the plugin object, then the plugins private history buffers is returned. As the history holds sfloat data there is a very small precision loss compared to pre-0.115 versions.

There are always **JACK-CHANNELS** history buffers for the input ports and **#PLUGINS** buffers for the active plugin objects allocated. The size of each history buffers is **#HISTORY-SIZE SFLOATS**. See for the config.frt file to change the default values. Each history holds several groups of **JACK-CHKSIZE SFLOATS** segments like shown in **fjack/history.png**

: `*N-HISTORY ( history idx - addr )`
Calculates the address inthe history buffer

: `@IDX-HISTORY-DATA ( history idx - ) ( F - data )`
Reads the audio data in the history buffer at position idx.

: `@HISTORY-DATA ( history offset - ) ( F - data )`
Reads the audio data in the history buffer relative to the current position in the audio stream.

: `MOVE-HISTORY-DATA ( historybuffer startoffset size target-addr - )`
Copies size samples of audio data from the history buffer to the target buffer. The offset marks the end of data.

: `@INTERPOLATED-HISTORY-DATA \ ( historybuffer - ) (F time-offset - data )`
You may read the interpolated audio data from the history buffer by this, the **`time-offset`** is measured in seconds. a zero time offset always references to the currently processed sample in the input->output processing chain

so it is **only** valid there../waf configure –prefix=/usr –mixed –libdir=/usr/lib64 –alsa –firewire –dbus

2.7 Managing time

: `JACK-TIME? ( - d )`
The time of the JACK server is always 64bit, so for portability it's a double in Forth. The time is derived from the high resolution timer, it is measured in usec. This is a 'real-time' when you ask for it, it is the servers time.

: `JACK-PROCESS-TIME? ( - d )`
The time of the JACK server is always 64bit, so for portability it's a double in Forth. The time is derived from the high resolution timer, it is measured in usec. This time is not the 'real-time' but the real-time when the last processing callback for all connected clients started.

: `USEC>SAMPLES ( d-usec - samples )`
Converts the double number usec to samples.

: `SAMPLES>USEC ( samples - d-usec )`
Converts samples to the double number usec.

: `JACK-UNIQUE ( - d )`
This is a unique double telling about the position in JACKs processing queue. See for more details in the libjack sources or API documentation, not much use for this in most situations.

2.8 Managing blocks of 'WAV' data

`0 VALUE WAV-PLAY-MODE`
When playing wav data the audio samples are inserted in the input->output procesing stream. This can be before or after the signal processing is done. **`WAV-PLAY-MODE`** is used to switch between these modes , n=0 means before processing.

`1 VALUE WAV-READ-MODE`
When recording wav data from the input->output processing stream the samples can be fetched from before or after are signal processing. **`WAV-READ-MODE`** is used to switch between these modes , n=0 means before processing.

So wav-playing and recording uses history buffers selected by the values of **`WAV-PLAY-MODE`** and **`WAV-READ-MODE`**. In this graphics using a value of 0 uses the green buffers, red buffers otherwise. The picture **fjack/waving.png** might help you to understand.

: CALC-WAV-RATE (srate - true-sample-rate)
The sample rate conversion when playing or recording wav data uses a rather simple linear approximation algorithm, the used 'real' sample rates can be any **truerate = n * samplerate / JACK-CHKSIZE**. This ensures a small distortion and no glitches are in recorded or played data. Another point is: there are better buffersizes than others! When you want to use resampling, **CALC-WAV-RATE** tells the true sample rate calculated from the desired one.

This is only relevant for **PLAY-WAV RECORD-WAV**.

: CALC-WAV-BLOCK (srate channels mode - n)
As told at **CALC-WAV-RATE** there are 'better buffersizes than others'. **CALC-WAV-BLOCK** gives a good suggestion what the block size should be, or better multiples of this.

: WAV-MODE? (mode - flag)
Flag is true when the wav mode is supported by this system.

: PLAY-WAV (addr size rate channels mode - id ?error)
Plays WAV data available at an addr and a size. These data are not played immediately but are inserted into a FIFO 'slot' and will be played after all before playing commands have finished. When size is negative the FIFO is cleared before inserting this block into the FIFO so playing will start at once. More precise - it will start when the next **PROCESS-FRAMES** callback is taken. A negative channel number results in reversed channels when listening, playing wav files with only one channel will be played as stereo. Thew rate can be freely choosen, the mode can be any of the 8bit/16bit/32bit/32bit-float modes supported here. Named constants for these are defined in **JACKAUDIO** these are defined **STEREO MONO 8BIT-AUDIO 16BIT-AUDIO 32BIT-AUDIO SFLOAT-AUDIO**.

The 'id' can be used to test a block of wav data in the FIFO. See: **WAV-PLAYING? STOP-PLAY-WAV**, ?error tells about the status

- 0 – everything was ok and id is 'legal'
- 1 – not supported mode
- 2 – too many channels
- 3 – FIFO overflow
- 4 – JACK is not running or has stopped

: READ-WAV (addr size rate channels mode - id ?error)
READ-WAV works as **PLAY-WAV** except it reads the data from the audio stream and puts them into the buffer at addr. More info at **PLAY-WAV**

: STOP-PLAY-WAV (ms id -)
The data that had earlier been put into the play FIFO and had returned 'id' will stop playing after the specified time ms (milliseconds). When the original block had a shorter duration all data will be played.

: STOP-READ-WAV (ms id -)
The data that had earlier been put into the read FIFO and had returned 'id' will stop reading into this buffer after the specified time ms (milliseconds). When the original block had a shorter duration all data read later won't be put into the buffer.

2.8.1 Checking for blocks of wav data

: WAV-PLAYING? (id - val)
Checks the status of the played wav block 'id'. 'val' is the remaining time until this block is played completely or zero when the fifo block is unused. A typical piece of code code be

```
( id ) BEGIN DUP WAV-PLAYING? DUP MS 0= UNTIL DROP
```

: WAV-READING? (id - val)
Checks the status of the recorded wav block 'id'. 'val' is the remaining time until this block is recorded completely or zero when the fifo block is unused. See **WAV-PLAYING?**

2.8.2 Using the Wavboard

The Wavboard is a new feature in fJACK 1.10. You can now play wav data directly from memory without the other wav playing sheme.

You can play up to **JACK-BOARDJOBS** sounds at the same time - this is a config option -, you may also auto-repeat them and define a starting time delay.

: START-WAVBOARD-JOB (delay firstchannel repeats address size channels wavmode - id) (F: speedup -)

This starts playing the wavdata from memory after **delay** milliseconds, it will be repeated **repeats** times. When delay is negative the absolute value is the number of samples to be delayed. All negative values between -15 and -1 should not be used as these are reserved for extensions. Also when delay is negative the returned value of **WAVBOARD-PLAYING?** will be the number of samples. **firstchannel** is the lowest fjack channel to be used. **address** points to the data, **size** is the size of the wavdata block in bytes. **wavmode** is the same as for other wavdata shown in **PLAY-WAV**.

Playing such wav data can be done at any sample rate; conversion is done automatically from wav-rate to **JACK-SAMPLE-RATE**, an extra tuning with **speedup**. Please note that wavdata at a) the original samplerate and b) 1-4 channels have a much lower jack load so it might be worth to do conversion before. As the whole block of data is already available there is no restriction for wavrates

: GET-WAVBOARD-POSITION (id - done total)

You can read the number of samples done already and total. Both values are corrected using the used **JACK-SAMPLE-RATE**

: SET-WAWBOARD-POSITION (newpos id -)

Sets the playing position for this wavboard job. **newpos** is the new position, see also **GET-WAVBOARD-POSITION**

: SHOW-WAVBOARD (-)

Lists active wavboard jobs

: STOP-WAVBOARD-JOB (delay id -)

A wavboard job started earlier can be stopped by this function, it requires the job id and a delay time. delay can also be negative as described in **START-WAVBOARD-JOB**

: SPEEDUP-WAVBOARD-JOB (id -) (F: speedup -)

A wavboard job can be given a new samplerate, the given speedup is multiplied with the speedup used when the wavboard was started.

: WAVBOARD-PLAYING? (id - time)

Checks the status of the played wavboard 'id'. time is the remaining time until this wavboard sound is played completely or zero if it couldn't be found. You could use this to start another wavblock right after this one. At least 1 is returned as val, the maximum is $7FFFFFFF. time can be either the time in msec or in samples, this depends the delay parameter when the wavboard job had started.

: STOP-ALL-WAVBOARD (-)

You may also stop all jobs right now.

2.9 The Callbacks

These functions are **not** to be used in user applications. They are functions to be called by the callbacks defined for this client. They could be used by testing tools to measure the exact performance of the io system but so far no such tools exist.

: `XT-PROCESS-FRAMES ( frames parameter - 0 )`
The main processing callback; must return 0. When the client is initialised the jack server starts a realtime thread, this thread later calls this callback and **MUST NEVER** use any forth word that might add some sort of delay longer than a few ms. Otherwise the client might be terminated or lots of xruns are signalled.

So — No semaphores! No disk-io! No terminal-io!

The **XT-PROCESS-FRAMES** callback does **all** the signal processing defined by you in the input->output processing slots. This practically limits things you can do in the processing chain, when writing your own plugins you should keep this in mind. Data sampling to/from disk or network, signal analysis, graphics ... all must be done in a strictly asynchronous way. As you can not safely use semaphores you should use shared memory for the data and thread-local data for your 'housekeeping'.

: `XT-GRAPH-REORDER ( 0 - 0 )`
Whenever the jack servers processing graph is changed this callback is executed. This happens when ports are disconnected or a new jack application is starting. The user-defined **GRAPH-REORDER-HOOK (-)** is executed in this callback to allow you to watch the surrounding.

: `XT-PORT-CONNECT ( a b parameter flag 0 - 0 )`
Whenever a port is connected or reconnected this callback is executed. You can find the fJACK ports with **>inport (idx - id)** and **>outport (idx - id)**, both words are in **JACKAUDIO**. idx is the channel, the returned id is unique and given by the jack server. I won't give any details here how to use these id's or how to use the **PORT-CONNECT-HOOK** - go and read the jack manual, you will need deeper informations to make use of it. **PORT-CONNECT-HOOK** is deferred. So your callback code must work on these stack values

\ : `YOUR-DEFINITION ( a b flag - ) ....... ;`
The prototype for the client supplied function that is called whenever a connection is changed. a and b are the two ports to be connected or disconnected. The flag is non-zero if ports were connected and zero if ports were disconnected

: `XT-NEW-FRAMES ( fr parameter - 0 )`
When some other client changes the chunksize this is told here. So far i don't know ANY client that does so and JACK versions up to 0.109 didn't allow this to happen at all. But the jacklib API defines it and the latest JACK development goes in this direction - jackdmp derived sources. See **JACK-CHKSIZE**

: `XT-NEW-SAMPLERATE ( sr parameter - FALSE )`
The callback when other clients change the sampling rate for the jackd. See **JACK-SAMPLE-RATE**

: `XT-CLIENT-STOPPED ( parameter - 0 )`
A client that is forced by some other client to shut down tells all it's registered clients about the new situation.

: `XT-COUNT-XRUNS ( parameter - 0 )`
When the JACK server gets an interrupt by the audio master clock to do new processing and the queue of callbacks of the last interrupt has not been fully processed, this condition is noted to all clients - it's called a 'XRUN'. See **JACK-XRUNS**

: `XT-NEW-TRANSPORT ( s p dummy - f )`
A Repositioning callback, so far the position is not used.

: `XT-SERVER-ERROR ( str - str )`
The callback function to receive a jack server error message.

: `XT-SERVER-INFO ( str - str )`
The callback to receive a jack server information message.

: `XT-SESSION-EVENT ( event *args - )`
The callback serving session events

: `XT-LATENCIES ( mode *args - )`
The latency callback

2.10 Connecting the ports

There are some graphical tools around to connect ports between the different JACK clients or the playback or capture ports. You may use these nice tools to connect any ports but how can you save and restore your configuration?

: `SAVE-CONNECTIONS ( addr cnt - )`

writes a file specified by the name 'addr cnt' holding information about all ports connected to your fJACK client. This file can later be used with **`LOAD-CONNECTIONS`** and **`CLEAR-CONNECTIONS`**. Actually the file is a forth source.

Please make sure the directory fjack/sysfiles has writing enabled for you.

: `CONNECT-JACKPORTS ( - ) \ source destination`

Connects two ports, the portnames are read from the input stream.

: `UNCONNECT-JACKPORTS ( - ) \ source destination`

Disconnects two ports, the portnames are read from the input stream.

: `CLEAR-CONNECTIONS ( - )`

Unconnects all ports of the fJACK client.

: `LOAD-CONNECTIONS ( addr len - )`

Connects all connections specified in the file.

: `UNLOAD-CONNECTIONS ( addr len - )`

Disconnects all connections specified in the file.

2.11 Getting detailed information.

: `JACK-LATENCIES ( - max min )`

Returns the latencies found for the fJACK client and it's ports.

: `JACKLOG ( - )`

Shows the jack servers and fjack client error messages. This works slightly different on vfx and iForth systems; on iForth we have the thread-safe debugging device that also tells about exact timing of the messages. This might help a lot when debugging asynchronous messages from callbacks or midi interfaces. On vfx the genio buffer device is used. For most situations this is fine.

On all systems supported by fJACK you can be sure **any** debugging info will **not** interfere with realtime behaviour. If you write applications based on fJACK you might use **`<<DEBUG ." This is nice CR DEBUG>>`** sequences.

: `.JACK-VERSION ( - )`

prints the current fJACK version and mods

: `JACK-INFO ( - )`

Prints information about the fJACK client.

: `ANY-PORT-CONNECTED? ( - flag )`

Checks for output port connections done.

: `JACK-PORTS ( - )`

Prints information about all connections to the fJACK client.

2.12 Running the fJACK client

: `START-JACK-TRANSPORT ( - )`

As the name says the JACK server transport is started.

: `STOP-JACK-TRANSPORT ( - )`

As the name says the JACK server transport is stopped.

: `JACK-SERVER-RUNNING? ( - flag )`

flag is **TRUE** when a running jack server is available

: `STOP-JACK-CLIENT ( - )`

Stops the fJACK client running.

: `START-JACK-CLIENT ( - )`

Starts the fJACK client - this is done automatically when fjack is compiled and also via the **AtCold** hook so you will only need it after you have stopped the client or it has been stopped by the control application. If the clients name is not unique - there could be another fJACK client running already - the jack server will choose another name like client-01. This is totally OK as the client struct in fJACK takes care about this, and knows about the connections. There is one good reason to do so:

Different applications can all use the default client fJACK and don't have to care about other names. So you may start several Forth terminals and start a fJACK client, all these clients can share the connection files - the true client name is just hidden. You might also choose another client name, see the config option **JACK-CLIENT-NAME**

When starting the jack client you will often want to connect some ports. This can be done via a session manager although this might not be very intuitive. Another very simple way is: the forth code file **fjack/sysfiles/defaultports.frt** is always interpreted when *forth{START-JACK-CLIENT} is done.

Earlier fJACK versions tried to start the jack server somehow; this has always been somewhat tricky and needed OS dependent tricks. **All** jack versions 0.121/1.97 or later can start the jack server automatically but autostarting the jack server is **not** the default anymore, you can change this via the **JACK-AUTOSTART** config option.

BTW, you should really switch to modern jack control applications like QjackCtl as they support application sessions, and many more features for a better desktop integration.

: `AT-START-JACK-CLIENT ( xt - )`

Your application might want to do something more when the jack client is started. This adds this 'xt' into a chain.

: `AT-STOP-JACK-CLIENT ( xt - )`

Your application might want to do something more when the jack client is stopped. This adds this 'xt' into a chain.

: `AT-REMOVE-JACK-CLIENT`

Your application might want to do something more when the jack system is stopped/removed. This adds this 'xt' into a chain.

2.12.1 Sessions

fJACK 1.10 brings support for session management, this feature allows control applications like 'qjackctl' to start a bunch of jack applications and connect them all in a way you defined. fJACK should be saved in a turnkey application and have a starting bash script. This script can be called by the session manager. This feature is very new and not well tested by other users and probably only works on iforth linux systems ... please let me know what you find out.

Chapter 3

Using .wav Files

Some constants - in **JACKAUDIO** - for better readability

```
     2 CONSTANT STEREO
    1 CONSTANT MONO
    8 CONSTANT 8BIT-AUDIO
  #16 CONSTANT 16BIT-AUDIO
  #32 CONSTANT 32BIT-AUDIO
 #-32 CONSTANT SFLOAT-AUDIO
```

3.1 Playing and recording

As playing and recording of complete .wav files is done in a background thread is it sometimes conveniant to have a hook when the recording/playing is actually started or stopped in the thread.

```
 DEFER BEGIN-OF-PLAYING          ' NOOP IS BEGIN-OF-PLAYING
DEFER END-OF-PLAYING            ' NOOP IS END-OF-PLAYING
DEFER BEGIN-OF-RECORDING        ' NOOP IS BEGIN-OF-RECORDING
DEFER END-OF-RECORDING          ' NOOP IS END-OF-RECORDING
```

`: .WAV ( c-addr u - ) 1 (PLAY-WAVFILE) ;`

Plays the complete file defined by the parameters in a background thread, show some information about the file.

`: .SILENTWAV ( c-addr u - ) 0 (PLAY-WAVFILE) ;`

Plays the complete file defined by the parameters in a background thread, no information or error messages are printed.

`: .SOUND ( samplerate channels mode c-addr u - ) 2 (PLAY-WAVFILE) ;`

Play any wav file with specified parameters.

`: STOP-PLAYING ( - )`

Any file already playing can be stopped by this command.

`: REPOSITION-PLAY-FILE ( position relative-flag - )`

The played wav file may be repositioned during playing. If relative-flag is **TRUE** the position is relative to the current position, otherwise it's the absolute position. The 'position' always means a sample position, so you don't have to worry about channels and mode.

`: PLAY-FILE-POSITION ( - position|-1 )`

This tells you the current position of the played wav file, it's also a sample postion so don't care about channels and mode. If the position is negative, there is no file played at the moment.

: START-RECORDING (time samplerate channels mode name namelen -)

Starts recording a .wav file with a given file name. Also the audio parameters must be specified. time is in seconds. This word is the basis of the other file recording commands.

: STOP-RECORDING (-)

Stops .wav file recording.

: RECORD-WAV (time name len -)

Records a .wav file with the duration of time seconds and the given file name. audio parameters are **JACK-SAMPLE-RATE STEREO 16BIT-AUDIO**

: RECORD-SIMPLE (time name len -)

Records a .wav file with the duration of time seconds and the given file name. audio parameters are **16000 MONO 8BIT-AUDIO**

: RECORD-FLOATS (time name len -)

Records a .wav file with the duration of time seconds and the given file name. audio parameters are **JACK-SAMPLE-RATE STEREO SFLOAT-AUDIO**

: RECORD-STUDIO (time name len -)

Records a 4 channel .wav file with **SFLOAT-AUDIO**. Best for later processing in applications like audacity ...

: WAV-PLAYING-ON? (- flag)

flag is true when fJACK is already playing a .wav file.

: WAV-RECORDING-ON? (- flag)

flag is true when fJACK is already recording a .wav file.

3.2 Using snippets

: SNIPPET (c-addr u -) \ name

Expects a filename on stack and creates a word 'name'. The file must be a wavfile, it's read into allocated memory. The runtime behaviour of the defined word is: play the audio data immediately via the fJACK client.

Snippets are defined in a linked list; audio data are also available in turnkey applications as they are loaded in allocated memory automatically.

: WAVSOUND (c-addr u -) \ name

Expects a filename on stack and creates a word 'name'. The file must be a wavfile, it's read into allocated memory. The runtime behaviour of the defined word is:

name (delay-ms lowest-channel repeats - id) (F: speedup/-1 -)

... playing the audio data via the wavboard. See **START-WAVBOARD-JOB** for more details. If speedup is -1 the samplerate conversion is turned off.

wavsounds are defined in the snippets linked list; audio data are also available in turnkey applications as they are loaded in allocated memory automatically.

: .SNIPPETS (-)

Prints a list of all defined **SNIPPET**s and **WAVSOUND**s.

3.3 The wavboard

: .WAVBOARD (name len - 0|id)

Plays the file via the wavboard and returns the wavboard job id or 0 when playing could not be started. This is done with a back thread making sure all memory is deallocated when playing has finished

Chapter 4

The FFTW3 library interface

4.1 Calculating FFT

This library can be tuned for optimized performance, this implementation currently supports the import of system wisdom files - see the fftw3 docs how to generate such a wisdom file.

Without such wisdom the best algorithm is just estimated, you can change this to one of the other methods in fftw3lib.frt.

: `CHECK-FFTW-SIZE ( n - ok-mask ) \ sizes in fftw may be multiples of primes 2-13`

Checks the desired size of a fftw3 plan. Principally all sizes are suported but many ready-compiled libraries only support a limited number range. For all

n = 2*a * 3^b * 5^c * 7^d * 11^e * 13^f

the fftw3 library has algorithms available. The 'ok' is a bit-mask about a-f being >= 1, for probably impossible sizes %000000 is returned

```
%000001-a, %000010-b, %000100-c, %001000-d, %010000-e, %100000-f
%000001 has best algorithms
%001110 still is good
%110000 has existing algorithms in probably all compiled libraries but will be slower
```

: `WINDOWING-OFF ( struct )`

The input parameters for the signal analysis may be 'windowed', this switches windowing off.

: `HAMMING-WINDOW ( struct )`

The input parameters for the signal analysis may be 'windowed', this switches to 'hamming'

: `HANN-WINDOW ( struct )`

The input parameters for the signal analysis may be 'windowed', this switches to 'hann'

: `FFT-STRUCT ( n - struct )`

When you want to to signal analysis using the fftw3 library you must define a forth struct that defines the interface to the fftw3 library. 'n' is the number of samples to be analysed, **`FFT-STRUCT`** defines the struct and returns a pointer to it. This struct pointer will be used by the next words.

Internally there are arrays for real parameter data and complex results at place, a fftw-plan is calculated. See **`CHECK-FFTW-SIZE`** for a discussion about possible sizes.

: `INVERSE-FFT-STRUCT ( n - struct )`

When you want to to signal analysis using the fftw3 library you must define a forth struct that defines the interface to the fftw3 library. 'n' is the number of samples to be analysed, **`INVERSE-FFT-STRUCT`** defines the struct and returns a pointer to it. This struct pointer will be used by the next words.

Internally there are arrays for real parameter data and complex results at place, a fftw-plan is calculated. See **`CHECK-FFTW-SIZE`** for a discussion about possible sizes.

: `#FFT-SIZE ( struct - #elements )`

tells the number of samples for this struct.

: `>FFT-COMPLEXDATA ( struct idx - addr )`

Calculates the address in a **FFT-STRUCT** result complex datastructure or **INVERSE-FFT-STRUCT** input parameter complex datastructure.

All FFT data are of **SFLOAT** type and so must be accessed with **SF@ SF!** and friends. In complex number arrays the real-part is always at the lower address.

: `>FFT-DATA ( struct idx - addr )`

Calculates the address in a **FFT-STRUCT** input parameter datastructure or **INVERSE-FFT-STRUCT** result datastructure.

: `CALCULATE-FFT ( struct )`

After storing the input data into the **FFT-STRUCT** parameter array you can do the signal analysis with **CALCULATE-FFT**.

: `CALCULATE-INVERSE-FFT ( struct )`

After storing the input data into the **INVERSE-FFT-STRUCT** complex parameter array you can do the signal

Chapter 5

The fJACK Input->Output Processing

You know that you can put audio processing plugin objects in any one of the slots, this module takes care of it. The order of processing in the jack-callback is shown in **fjack/order.png**

In effect all plugins can assume, that data in a lower slot number have been completely processed already.

5.1 Defining the plugin objects

A plugin object is a data struct, some parts of this struct are the same in all plugin objects - see the client.frt source for details. These public parts are used by the plugin manager to take care about safety, setting parameters like frequency, quality and more.

Also the structs are 'connected' to other plugin objects or the input/output jack ports via plug_input and plug_output. basically a plugin object reads samples from the plug_input history buffer, does it's specific processing and writes data to the plug_input history buffer - in the graphics it's the red arrows. Also you may read data from other history buffers by words like **@HISTORY-DATA** and friends, see the green arrow and docs in the client manual. See picture **fjack/plugin.png**

The fjack system knows some predefined plugin object types, when you write other plugins use numbers above #256 for your private applications or let me know to include them in here.

```
 $01 CONSTANT LOW-PASS
$02 CONSTANT HIGH-PASS
$03 CONSTANT BAND-PASS
$04 CONSTANT BAND-STOP
$05 CONSTANT ADAPTIVE
$06 CONSTANT DECODER
$07 CONSTANT SIN-TONE
$08 CONSTANT DELAYER
$09 CONSTANT AUTOLEVEL
$0A CONSTANT USER-FIR
$0B CONSTANT MIXER-PLUGIN
$0C CONSTANT LESLIE-PLUGIN
$0D CONSTANT PHASER-PLUGIN
$0E CONSTANT ECHO&HALL-PLUGIN
$0F CONSTANT VIBRATO-PLUGIN
$10 CONSTANT EQUALIZER
$11 CONSTANT EFFECTS-FIR
$12 CONSTANT INSTRUMENT-PLUGIN
$13 CONSTANT GOERTZEL-PLUGIN
```

`\ CONSTANT #PRIVATE-DSP ( - n )`

All plugin objects have some data in common, the public part of the objects struct is used by the plugin handler.

The common part of a plugin object holds predefined **DSP-VALUE**s, so when you are making a plugin object definer you must leave this public part in peace. So when defining the struct you should 'start with **#PRIVATE-DSP**.

5.2 Using the plugin objects in the signal processing chain

: `LIST-PLUGINS ( - )`
prints all objects plugged into the process chain.

: `LIST-ALL-PLUGINS`
prints information about all defined plugins.

: `REMOVE-JACK-PLUGIN ( filterobject - )`
revoves the object from the process chain.

: `MAKE&ALLOCATE-PLUGIN-OBJECT ( initialize-xt dynmemsize objectsize - object )`
Whenever you define a plugin object you MUST use this function to initialize it. 'size' is the objects size in bytes, the XT is the function to initialize the object.

Dynamic memory size is allocated and plug_dynmem is set.

This is done at compile time - so the object can be used at once - also this allows turnkey applications, as when booting all defined objects are initialized.

When developing your own plugins you can and definitly should use the automatic memory allocation, this is done before the initialize-xt is called.

: `SET-PLUGIN-BYPASS ( flag filterobject - )`
You may set the objects bypass status. When the bypass flag is set - flag is **TRUE**, the plugin just passes the audio data to it's history buffer but does no processing on the data. See comments in **SET-JACK-PLUGIN**.

: `SET-JACK-PLUGIN ( filterobject slot channel - )`
Puts the object in the slot/channel. channel must be between 0 and **JACK-CHANNELS-1**, slot must be between 0 and **JACK-IO-SLOTS**. If there are wrong parameters an abort message is displayed. If an object is already in use at that slot it is removed.

Please remember two things, at the moment you may only use **#PLUGINS** at the same time, the history buffers are allocated when starting fjack. **SET-JACK-PLUGIN** and **REMOVE-JACK-PLUGIN** change the processing order and clear the history buffers used. On slow machines the clearing might take some time resulting in audio drops when changing the plugin objects in use. To overcome this problems you could keep a plugin object in the processing chain and just toggle it's bypass status.

: `RECONNECT-PLUGIN ( history-buffer object/output-port - )`
Due to the new input->output sheduler every plugin object or output port can get it's data from another history buffer. This function can be used to connect at runtime. See the example where the two dotted red lines show reconnected plugins, the green line shows a reconnected output port. You can get the address of the history buffer by **OBJECT-HISTORY**. See picture: **fjack/reconnect.png**

: `LOCATE-JACK-PLUGIN ( object - slot channel true | false )`
Gives information about the slot/channel for a specified object and a **TRUE** flag when the object was installed or **FALSE** when not found.

: `GET-PLUGIN-OBJECT ( slot channel - 0|filterobject )`
Gives the object found in slot/channel, when no object has been placed there with **SET-JACK-PLUGIN** a **FALSE** flag is returned.

: `SET-OBJ-FREQUENCY ( frequency object - )`
Many plugins have a specified tune-to-frequency function available, you can use this to do so. This sets the frequency of the specified object, the object itself known what do do, f.e. filters will be tuned to the specified frequency. A few object types have to frequencies, **BAND-PASS BAND-STOP** filters expect two integers as a special case.

: `SET-OBJ-FLOAT-FREQUENCY ( object - ) ( F: frequency - )`
In many situations the precision of **SET-OBJ-FREQUENCY** is not good enough, then you can set the frequency by this. BEWARE: ONLY ONE parameter possible as float!

: SET-OBJ-QUALITY (quality object -)

Many plugins have a quality - in notch filters this would be the sharpness in FIR and adaptive filters this will be the number of taps. This functions handles the objects quality in a general way so look for the special plugin docs. The effect of setting the quality can be viewed in the fft display plugin, shown are band filters 41 **fjack/41poles.png** and 125 poles **fjack/125poles.png**

: SET-OBJ-LEVEL \ (object -) (F level -)

All plugin objects have a specific level value this word sets it.

: SLOT/CHANNEL-HISTORY (slot channel - history)

Finds the history address of a slot/channel pair, if the slot is less than 0 it always means the input port. If a slots object is undefined the result is 0.

: CALCULATE-FILTER-DELAY (slot1 slot2 channel - n)

Calculates the phase delay in the processing chain between 2 slots for a given channel, the result is the phase shift measured in samples.

5.3 The Group of Finite Impulse Response Filters

: MAKE-FIR-FILTER (filtype size - object)

Constructs a linear-phase FIR filter object and leave it's address. 'type' can be **LOW-PASS HIGH-PASS BAND-PASS BAND-STOP USER-FIR**, the 'size' is the number of taps corrected to odd. FIR filters add a signal delay by half the number of taps.

You may select a different number of taps later via **SET-OBJ-QUALITY (taps object -)**, also you might change the frequency response of this filter by **SET-FIR-TUNING**.

BAND-PASS BAND-STOP USER-FIR filters and the **SET-FIR-TUNING SET-FIR-WINDOWING SET-FIR-COEFF** are only possible when the fftw3 library is installed.

: FIR-FILTER: (type size -) \ name

Defines a named **MAKE-FIR-FILTER** plugin object, calling 'name' returns the object address.

: MAKE-EQUALIZER (size - object)

Define an linear phase equalizer object with size taps corrected to odd, it's a special type of the generic FIR filter. It offers 256 frequency bands in the 0-JACK-SAMPLE-RATE/2 range, each band has it's own defined level. You can set these levels by **SET-EQUALIZER-BAND** and **SET-EQUALIZER-RANGE**. You may also select a different number of taps later via **SET-OBJ-QUALITY (taps object -)**.

For iForth users only: the showfft module now can directly control a bound equalizer filter, see **SET-FFT-EQUALIZER** in the gui/showfft module.

: EQUALIZER: (size -)

Defines a named **MAKE-EQUALIZER** plugin object, calling 'name' returns the object address.

: SET-EQUALIZER-BAND \ (object -) (F dBlevel frequency -)

Sets the level for the specified frequency in the equalizer object and calculates the filter coefficients.

: SET-EQUALIZER-RANGE \ (object -) (F dBlevel1 frequency1 dBlevel2 frequency2 -)

Sets a linear level curve between the level/frequency pairs in for the equalizer objects levels and calculates the filter coefficients. At least one point is set.

: SET-EQUALIZER-MODE (mode object -)

Sets the approximation mode for **SET-EQUALIZER-RANGE**, it can be either 0 for linear or 1 for sinoid mode.

: EXPORT-EQUALIZER (name len object -)

Exports the equalizer setings to a file

: IMPORT-EQUALIZER (name len object -)
Imports the equalizer setings from a file

: SET-FIR-WINDOWING (windowmode object -)
Calculating the filter coefficents is done with a IFFT algorithm using a windowing function. This windowing function can be set for each filter with **SET-FIR-WINDOWING**. See the literature for descriptions of these windowing functions. The default is set by **DEFAULT-FIR-WINDOW**.

The blackmann window **0** gives the deepest out-of-band suppression with a modest steepness.

The hamming window **1** has a steeper flank with less out of band suppression

The rectangle window **2** has the steepest flank, highest ripple and lowest out-of-band suppression.

: SET-FIR-TUNING (0|tuning-xt object -)
The **MAKE-FIR-FILTER**s frequency response curve of each filter can be tuned to your needs by adding a tuning-function-xt describing the desired freuency response. The function must have this stack effect:

```
(F level frequency - your-level )
```

It must be able to return values for frequencies from 0 to sample-rate/2.

: SET-FIR-COEFF (0|set-coeff-xt object -)
The **USER-FIR MAKE-FIR-FILTER** is a very special type of FIR filter, the dsp algorithm is the same but there are many situations where you don't want to specify a frequency response of the filter but want to add effects like hall, echo, delay, averagers and more.

To use this sort of filter you will need to think about the way the filter coefficients are calculated. Whenever the filter is tuned, resized or defined there is the default function that sets x[0] to 1.0e0, all others are zero. To define a filter according to your needs you will have to tell the coefficient-calculating-function execution token xt. The calculating function must take two integer parameters **(number-of-filtertaps index)** and must return the float **(F - coefficient)** for that tap. This automatically set the filter size, so don't use **SET-OBJ-QUALITY** after calculating. The maximum number of taps is **#FIR_BUFF** so when you need a larger range think of the generic effects plugin.

The graphics shows the signal flow, the number of taps is the number of multipliers, view **fjack/firkern.png**

5.4 The Adaptive Filter object

: MAKE-ADAPTIVE-FILTER (size - object)
Constructs an adaptive filter object and leave it's address, the 'size' is the number of taps corrected to even. Adaptive filters add a signal delay by half the number of taps. This filter uses a simple 'least mean square' algorithm.

You may select a different number of taps later via **SET-OBJ-QUALITY (taps object -)**.

: ADAPTIVE-FILTER: (n -) \ name
Defines a named adaptive plugin object, calling 'name' returns the object address.

: SET-FREEZE-ADAPTIVE (flag object -)
You can freeze the adaptive filter coefficients - this is usefull when you have 'tuned' to a noise signal and want to stay with this filter. **FALSE** will continue adapting otherwise it will be frozen. As changing the filter qaulity means other coefficients **SET-OBJ-QUALITY** for an adaptive filter always unfreezes it.

: SET-ADAPTIVE-SPEED (n object -)
Sets the speed of the adapting algorithm, 0-100 are accepted.

5.5 The Infinite Impulse Response Filter

: MAKE-IIR-FILTER (type - filter)
Constructs a 10pole IIR-FILTER (Butterworth) plugin object and leaves the address of the object. The freqency of this filter can be set as usual with **SET-OBJ-FREQUENCY** the quality (or sharpness) of the filter with **SET-OBJ-QUALITY**

: IIR-FILTER: (type -) \ name
Defines a named iir-filter plugin object, calling 'name' returns the object address.

5.6 The FFT Analyser Plugin

Quite often you will have to do a signal analysis of your audio signals somewhere in the input->output processing chain. You can define such an analyser plugin with **MAKE-FFT-PORT** and can always do an ad-hoc real time analysis of the signal.

This plugin does an fft analysis, avaraging of the spectrum, finds maxima and the average level in a defined frequency range. In the background this uses the fftw3 library.

: `FFT-PORT-SIZE ( fftport - size )`

Tells the size of the port.

: `FFT-HAMMING ( mode fftport )`

Changes the 'windowing' function of the analyser port; 0 means no windowing, 1 is hamming and 2 hann.

: `FFT-ANARANGE ( low-frequency high-frequency fftport - )`

Sets the frequency range in which the average signal level is calculated.

: `FFT-GET-ANARANGE ( fftport - low-frequency-index high-frequency-index )`

Gets the frequency range for average analysis.

: `MAKE-FFT-PORT ( size - fftport )`

Define a fft-port, this can be used as a plugin in the processing chain. For the iForth system there is a fft viewing module available **FFT-VIEWPORT:** that works perfectly with fft-ports. See **fjack/fftplug.png**

: `FFT-PORT: \ ( size - ) name`

Defines a named fft-port.

: `FFT-ANALYSIS ( fftport - )`

Does an fft analysis of the last audio data in the stream plus avaraging and maxima finding.

: `@FFT-dB-LAST \ ( fftport idx - ) (F - level-dB )`

gets the average level for the last 8 **FFT-ANALYSIS** for this port.

: `@FFT-dB-AV \ ( fftport idx - ) (F - average-dB )`

gets the average level for the last 8 **FFT-ANALYSIS** for this port.

: `FFT-MAXIMUM? \ ( fftport idx - flag )`

returns **TRUE** when a maximum is detected at this point.

: `FFT-GREATEST-MAXIMUM \ ( fftport - index ) (F - level )`

Finds the greatest maximum for the port and returns the index.

: `FIND-AVERAGE-LEVEL \ ( fftport - ) (F - avlevel )`

returns the average signal level in the specified frequency range measured in dB

: `FFT-IDX>FREQU \ ( index fftport - frequency )`

Converts index for this port to frequency.

: `FFT-FREQU>IDX \ ( frequency fftport - idx fftport )`

Converts frequency for this port to index, the port kept on the stack as this is most often used later.

: `SET-FFT-DELAY ( delay fftport - )`

The fft analyser allows a delay in the history to compensate in fft displays.

5.7 The Channel-Mixer (and processing) plugin object.

Allows mixing and processing of audio data.

: `MAKE-MIXER ( - object )`

Construct a mixer plugin object and leave it's address. The value in the processing stream is calculated by the presented value plus all other slots/channels that have been added to the mixing by **`SET-MIXER-LEVEL`**. **`SET-OBJ-LEVEL`** sets the level of the original source of this plugin.

The standard way to mix the audio data is just adding all values. There is a special feature here to do some more tricks on the data, you may replace the 'adding' function by some other word with **`SET-MIXER-ACTION`**.

: `MIXER:`

Creates a named mixer plugin.

: `SET-MIXER-INPUT \ ( slot channel mixer - ) (F level delay - )`

Sets the level and delaytime (seconds) in the mixer. A slot-number <0 means the input port. You can only mix data from smaller slots than the mixer objects slot. The picture **fjack/mixer.png** shows the mixer inputs

: `SET-MIXER-ACTION ( action-xt object - )`

This replaces the default 'adding' function of a mixer plugin object by another one.

The stack effect of the new function must be

```
: my-function ( array cnt - ) (F - value )
```

The array holds cnt sfloats of input data read from all the activated mixer channels, you can do whatever you want to do with these data but you **MUST** take the parameters and **MUST** leave the floating result. All data have already been changed to the desired level, the first position in the array holds the original input stream to this slot.

: `.MIXER-INFO ( object - )`

Show the status of the mixer object; level is absolute; delay measured in seconds.

5.8 The generic Effect Filter

Allows the user defined definitions of effects. The generic effect plugin is just a special type of FIR filter, the dsp algorithm is the same but the coefficients are not calculated from a defined frequency response. So the **`SET-OBJECT-FREQUENCY`** tool is undefined, instead it uses a callback function defining the coefficient for each tap number.

: `DEFINE-EFFECT-FILTER ( calc-xt|0 object - )`

calculates the effect filter coefficients for a given plugin object with the calculating function xt. The calculating function must take two integer parameters **`( number-of-filtertaps index - )`** and must return the float **`(F - coefficient)`** value for the index tap. When **`DEFINE-EFFECT-FILTER`** is executed, it calls the function for every tap from 0 to {#HISTORY-SIZE}, so the default setting will allow processing in the ~3 seconds range for CD sampling rate. When the xt is **`FALSE`** the effects filter is 'just-do-nothing'.

: `MAKE-EFFECT-FILTER ( - object )`

Constructs an effects plugin object leaves it's address. Calculation of the filter coefficients is done with **`DEFINE-EFFECT-FILTER`**.

This word is meant for user extensions like hall and echo effects, combs ... let's see what we will make out of it.

5.9 JACK MIDI support

fJACK supports MIDI now! There is a basic wordset available that allows control of external instruments, communication with other JACK MIDI clients and receiving MIDI-IN events for things like synthesizers. Of course this is all done in realtime or made safe by fifos.

5.9.1 Receiving MIDI messages

\ YOUR-MIDI-IN-EVENT (struct -)
The fJACK midi ports receive messages from connected midi enabled jack clients. You can define a word that is executed by the processing callback whenever a MIDI event is received. All MIDI events valid at this time are served in a bunch **before** the audio processing is done. ' YOUR-MIDI-IN-EVENT IS DO-PROCESS-MIDI-IN-EVENT defines the predefined hook to do what you desire.

Please remember again: this is all done in the processing callback! So all comments regarding realtime processing elsewhere in this manual are relevant for the midi processing too.

The struct passed holds the pure 'MIDI-command' in the first bytes. A simple MIDI interpreter/synthesizer could just interpret this information. As JACK-MIDI is designed for sample-precise audio we need an exact-to-the-frame time for the command. The struct has more information than just the command bytes. See MIDI-IN-EVENT-DATA

: MIDI-IN-EVENT-DATA (event-struct - str len time)
Reads information from the midi-event struct. This event-struct is the given to you in the DO-PROCESS-MIDI-IN-EVENT. This word will be used by a more advanced interpreter hooked into DO-PROCESS-MIDI-IN-EVENT. You will probably also use INTERPRET-MIDI-MESSAGE that extracts the information passed.

: SEND-MIDI-MESSAGE (p2? p1? p0 pars frame -)
We also can send MIDI messages via the outport to other clients like a fully featured synthesizer. This command is availble at the output port precise-to-the-frame - you can calculate the frame by JACK-POSITION plus some offset you would like. SEND-MIDI-MESSAGE is threading safe by semaphore protection. All sent messages are queued and presented at the outport in the right order with precise timing.

: MIDI-TO-MYSELF (flag -)
You can connect fJACKs MIDI output port to fJACKS MIDI input port or can disconnect it. This is just a shortcut to let the sent MIDI messages appear at fJACK MIDI-inport.

: ACTIONS-MIDI-SAMPLE (- 0 | cmd0 len0 cmd1 len1 .. cmd_n-1 len_n-1 n)
All MIDI-IN events are presented to the DO-PROCESS-MIDI-IN-EVENT hook in a bunch. In some situations you might want an io-processing plugin to know about at MIDI-IN event at the current sample.

: INTERPRET-MIDI-MESSAGE (message-string - type channel val1 val2)
A MIDI message string will be interpreted quite often, this returns the parameters to be presented to locals or what you like ...

5.9.2 Sending MIDI messages

The fJACK MIDI system allows MIDI messages to be sent/received to/from other MIDI handling clients with sample-precise timing. This means: Sending a message to start a tone will be received at exactly the sample you desired when sending the message. You can either send a message at a time-from-now or at a time-from-reference-time.

When you want to start a bunch of tones all at exactly the same time you will probably define that time by using SET-TIMEREFERENCE and later use the TIMERREFERENCE>FRAME calculation

: TIMEDIFF>FRAME (- #frame) (F: sec -)
Calculates the absolute #frame from the time (seconds-from-now)

: SET-TIMEREFERENCE (-) (F: seconds -)
Sets the current time + seconds as the time reference to be used later by TIMERREFERENCE>FRAME.

: TIMERREFERENCE>FRAME (- #frame) (F: seconds -)
Calculate the absolute #frame from the time (seconds-from-timer-reference)

: NOTE-ON (velocity note channel -) (F: sec -)
Sends a command to start a midi-note with given velocity and midi-channel. Also the time has to be given. This commands sends at time-from-now.

: NOTE-OFF (note channel -) (F: sec -)
Sends a command to stop a midi-note with given midi-channel. Also the time has to be given. This commands sends at time-from-now.

5.9.3 Instruments

An fJACKs "instrument" implements many things needed by real-time synthesizers, instrument and voices objects, allocation of dynamic memory, managing the voices of an instruments listening to midi commands, management of received MIDI commands and dispatching those commands to the available instruments depending on the midi channel.

The fJACK plugin system is fully supported, available are plugins 'listening' to the instruments voices.

This can be understood as a synthesizer middleware, currently there are two instruments coming with fjack. There is the wonderful Karplus-Pluck string-like instrument (fjack/karplus.frt). It was implemented by Bruno Degazio <lifemusic@sympatico.ca> and we worked hard to make it as performant as possible. He also implemented & inspired important parts of the instruments internals and kept me debugging.

There is also the fjack/synth.frt module; it's a much simpler system and should be read as a 'make-your-own-synth' addon.

Instruments are pretty complex, they produce signals on the fly and have to listen to MIDI commands. When developing karplus-pluck we ran into many problems, if you dig into this, i suggest to use MIDI debugging :-)

: `MAKE-INSTRUMENT-STRUCT \ name ( instrument-struct-size - )`

Instruments are structs - as you might have expected. Internals are mostly hidden and should be used with great care. This defines an instrument struct with a given name and struct size. After creating this instrument you will have to fill in some data to lets the instrument know what to do and what data it should use. Look into the karplus.frt and synth.frt sources to get a better idea. If you want to make a fjack instruments, just ask for suport.

: `MAKE-INSTRUMENT-LISTENER ( instrument - plugin )`

Make a listener plugin. This behaves like most other plugins, it adds the instruments sound to the audio stream. It is 'hard-wired' to the instrument, this can't be changed later. You can set the volume for this plugin by **SET-OBJ-LEVEL**

: `MAKE-SYNTH-INSTRUMENT \ name ( voices - )`

This defines a named synth instrument structure, you must pass the number of voices this instrument has. The new word will return the address of the structure. Code for allocating & deallocating dynamic memory is managed automatically.

: `DEFAULT-SYNTH ( midichannel - )`

This uses the existing **STANDARD-SYNTH**instrument and defines how it should sound ...

: `SIMPLE-SYNTH ( midichannel - )`

This uses the existing **STANDARD-SYNTH**instrument and defines how it should sound ...

: `MAKE-KARPLUS-INSTRUMENT \ name ( voices - )`

This defines a named karplus-pluck instrument structure, you must pass the number of voices this instrument has. The new word will return the address of the structure. Code for allocating & deallocating dynamic memory is managed automatically.

: `DEFAULT-KARPLUS ( midi-channel - ) \ you may define other pluck instruments like this`

This uses the existing **STANDARD-KARPLUS**instrument and defines how it should sound ...

: `SIMPLE-KARPLUS ( midi-channel - ) \ you may define other pluck instruments like this`

This uses the existing **STANDARD-KARPLUS**instrument and defines how it should sound ...

Chapter 6

Some tests

Some working plugins found in fjack/examples are here for you to start. All the plugins defined here and the test suite are NOT included in the default fjack module.

6.1 The Sinus Tone Oszillator Object, a detailed example for writing plugin objects

When writing more plugins read this and the **types.frt** carefully, also you can use the **prototype.frt** as a starting point.

Words that are used inside the object should be defined in the **JACKAUDIO** vocabulary.

```
 ALSO JACKAUDIO ALSO DEFINITIONS
```

Now a pointer to a counted text should be defined, later this is set to the **plug_type** object-element; it's there for a) making the plug_type defined and b) the **LIST-PLUGINS** and **LIST-ALL-PLUGINS** tools gives more detailed information.

```
 HERE ," SINUS oscillator" CONSTANT #sin-id
#PRIVATE-DSP
        DSP-FVALUE sin_phase
        DSP-FVALUE sin_step
        CONSTANT #sinstruct
```

Then the object elements are defined and the size of the whole object struct is made a **CONSTANT**

```
:  CHK-SIN #sin-id plug_type <> ABORT" Invalid SINUS-TONE object" ;
```

Check for a correct plug_type and aborts when this not ok

Each object type MUST have a specified signal processing funtion; the execution token of this function is stored in the **plug_xt** element. The funcion MUST take one **FLOAT** as a parameter and MUST leave one **FLOAT** as a result. So the depth of the floating stack must never be changed.

```
 : calc_sin_val  \ ( -- ) (F val -- val' )
        sin_step sin_phase F+ REDUCE.2PI FDUP TO sin_phase FSIN plug_level F* F+ ;
```

Before you start writing your own audio-data-processing code remember that

- the code is executed in a JACK demon callback
- the code MUST exit with a correct stack and floating stack

- the code MUST NEVER do anything that might give larger delays than a few usecs
- the code MUST NEVER do text/graphics output, file writing, use semaphores
- the JACK demon will halt the client as 'zombified' when delay is too large

Whenever you want to do things like that: FORGET ABOUT THEM in the process function. What you could and should do: Write thread-safe buffering functions!

You may watch the jack performance with two included tools, **JACK-LOAD** and **JACK-XRUNS**.

Many - as this example plugin - can be tuned to a frequency; the io_procs module implements a generic tuning word **SET-PLUGIN-FREQUENCY** which calls a function defined in the object making phase.

```
 : (tune-sine-object)    ( frequency -- )
        FDUP TO plug_frequency PI F* F2* JACK-SAMPLE-F-RATE F/ TO sin_step ;
```

Every object definer needs a function that initialises the plugin object when it is defined and also in the turnkey bootstrap word **INIT-ALL-JACK-OBJECTS**. If - in some cases - there is no such function needed **MAKE&ALLOCATE-PLUGIN-OBJECT** must be given 0.

```
 : (init-sine-object)
         ; \ nothing to do; no allocation
```

Now you can make an object-definer, as this should be public you should switch to **FORTH** before.

```
 FORTH DEFINITIONS
: MAKE-SINE-OSCILLATOR  ( -- addr )     \ Create a sinus oszillator struct
        \ Make an object and make it the 'current' object
        ['] (init-sine-object) 0 #sinstruct MAKE&ALLOCATE-PLUGIN-OBJECT
        \ Set the correct plug_type and set a plug_filtype mode
        #sin-id TO plug_type  SIN-TONE TO plug_filtype
        \ set the execution token of the signal processing function
        ['] calc_sin_val TO plug_xt
        \ set the tuning function
        ['] (tune-sine-object) TO plug_xt_tune
        \ these data can be permanent
        0e0 TO sin_step  0e0 TO sin_phase
        \ leave the 'current' object on the stack
        @DSP ;

\ A more simple definer makes the object and defines it as a constant
: SINE-OSCILLATOR:      ( -- ) \ name
        MAKE-SINE-OSCILLATOR CONSTANT ;


PREVIOUS PREVIOUS DEFINITIONS
```

6.2 A Bank of notch filters

Implements a bank of notch IIR butterworth filters, the slots 15-22 in a channel are used

`8 CONSTANT #NOTCHES`

The number of notch filters available

`:  GET-FREE-NOTCH ( - object flag )`

Try to get a notch filter object from a bunch of 8, if all notches are already in use flag is **FALSE**.

: `NOTCH-THERE? ( frequ channel - object flag )`

checks for an already used notch filter with frequency - or very close to. If such a notch is plugged in, the object and a **TRUE** flag are returned.

: `SET-NOTCH-FILTER \ ( frequ channel - )`

Sets a notch filter from a bunch of 8. If there is a notch already plugged in it is switched off.

: `CLEAR-ALL-NOTCHES ( - )`

removes all notches from the processing queue.

6.3 The Delayer Object

This can add an echo to the current signal, the level of the echo and the original signal can be set.

: `MAKE-DELAYER ( - object )`

Construct a delayer object and leave it's address.

: `DELAYER: ( - ) \ name`

Define a named delayer object

: `SET-DELAY-TIME ( useconds object - )`

Sets the dapay time for the delayer object. Time in usec.

: `SET-DELAY-LEVELS \ ( object - ) (F delayed-level now-level - )`

Sets the signal levels for a delayer object; both level must be floats.

6.4 A Vibrato effect - you can select the vibrato frequency and the strength.

: `MAKE-VIBRATO ( - addr )`

Make a vibrato effect plugin object and leave a pointer to it. When setting the frequency by **SET-OBJ-FREQUENCY** the argument is the frequency*1000 to allow more precise settings.

The vibrato strength is set as usual by the **SET-OBJ-LEVEL** method.

: `VIBRATO: ( - ) \ name`

make a named vibrato effect plugin

6.5 The Autolevel Object

This set the **JACK-IO-LEVEL** to a moderate level.

: `MAKE-AUTOLEVELER ( - object )`

Construct a autolevel object and leave it's address.

: `AUTOLEVELER: \ ( - ) name`

Define a named autolevel object

6.6 The Phaser Object

The phaser plugin emulates a phaser/flanger. You can control the timing of the shifted signal, the timing is also modulated by an oszillator.

: `MAKE-PHASER ( - object )`

Construct a phaser object and leave it's address. The default settings are a timeshift of 10msec, a strength of 0.2 and an oszillating speed of 5Hz.

: `PHASER: ( - ) \ name`

Define a named delayer object

: `SET-PHASER-PARAMETERS \ ( object - ) (F frequ strength timeshift - )`

Sets parameters for the phaser object.

The strength may be choosen from 0 to 1, this controls the oszillating amount, the default of 0.2 is already pretty strong.

Timeshift in milliseconds, the maximum depends on the history size, probably up to 100 msecs are usefull.

frequency controls the shift oszillator in Hz.

: `PLAY-SNIPPETS ( - )`

Plays some demo sound effects.

: `JACK-BENCHMARK ( - )`

A little fjack processing benchmark.

Have fun!