

**NAME**

**emc** – EMC (The Enhanced Machine Controller)

**SYNOPSIS**

**emc** [-v] [-d] [*INIFILE*]

**DESCRIPTION**

**emc** is used to start EMC (The Enhanced Machine Controller). It starts the realtime system and then initializes a number of EMC components (IO, Motion, GUI, HAL, etc). The most important parameter is *INIFILE*, which specifies the configuration name you would like to run. If *INIFILE* is not specified, the **emc** script presents a graphical wizard to let you choose one.

**OPTIONS**

- v** Be a little bit verbose. This causes the script to print information as it works.
- d** Print lots of debug information. All executed commands are echoed to the screen. This mode is useful when something is not working as it should.

**INIFILE**

The ini file is the main piece of an EMC configuration. It is not the entire configuration; there are various other files that go with it (NML files, HAL files, TBL files, VAR files). It is, however, the most important one, because it is the file that holds the configuration together. It can adjust a lot of parameters itself, but it also tells **emc** which other files to load and use.

There are several ways to specify which config to use:

Specify the absolute path to an ini, e.g.

**emc** /usr/local/emc2/configs/sim/sim.ini

Specify a relative path from the current directory, e.g.

**emc** configs/sim/sim.ini

Otherwise, in the case where the **INIFILE** is not specified, the behavior will depend on whether you configured emc with **--enable-run-in-place**. If so, the emc config chooser will search only the configs directory in your source tree. If not (or if you are using a packaged version of emc), it may search several directories. The config chooser is currently set to search the path:

**~/emc2/configs:/usr/local/etc/emc2/configs:/etc/emc2/sample-configs**

**EXAMPLES**

**emc**

**emc** configs/sim/sim.ini

**emc** /etc/emc2/sample-configs/stepper/stepper\_mm.ini

**SEE ALSO**

**halcmd(1)**

Much more information about EMC2 and HAL is available in the EMC2 and HAL User Manuals, found at /usr/share/doc/emc2/.

**HISTORY****BUGS**

None known at this time.

**AUTHOR**

This man page written by Alex Joni, as part of the Enhanced Machine Controller (EMC) project.

**REPORTING BUGS**

Report bugs to alex\_joni AT users DOT sourceforge DOT net

**COPYRIGHT**

Copyright © 2006 Alex Joni.

This is free software; see the source for copying conditions. There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

**NAME**

axis-remote – AXIS Remote Interface

**SYNOPSIS**

**axis-remote** <-ping> <-reload> <-quit> <-help>

**DESCRIPTION**

**axis-remote** is a small script to control a running AXIS GUI. Use **axis-remote -help** for further information.

**OPTIONS**

- ping** Check whether AXIS is running.
- reload** Make AXIS reload the currently loaded file.
- quit** Make AXIS quit.
- help** Display a list of valid parameters for **axis-remote**.

**SEE ALSO**

**axis(1)**

Much more information about EMC2 and HAL is available in the EMC2 and HAL User Manuals, found at [/usr/share/doc/emc2/](#).

**HISTORY****BUGS**

None known at this time.

**AUTHOR**

This man page written by Alex Joni, as part of the Enhanced Machine Controller (EMC) project.

**REPORTING BUGS**

Report bugs to alex\_joni AT users DOT sourceforge DOT net

**COPYRIGHT**

Copyright © 2007 Alex Joni.

This is free software; see the source for copying conditions. There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

## NAME

`axis` – AXIS EMC (The Enhanced Machine Controller) Graphical User Interface

## SYNOPSIS

`axis -ini INIFILE`

## DESCRIPTION

`axis` is one of the Graphical User Interfaces (GUI) for EMC (The Enhanced Machine Controller). It gets run by the runscrip usually.

## OPTIONS

### INIFILE

The ini file is the main piece of an EMC configuration. It is not the entire configuration; there are various other files that go with it (NML files, HAL files, TBL files, VAR files). It is, however, the most important one, because it is the file that holds the configuration together. It can adjust a lot of parameters itself, but it also tells **emc** which other files to load and use.

## SEE ALSO

`emc(1)`

Much more information about EMC2 and HAL is available in the EMC2 and HAL User Manuals, found at `/usr/share/doc/emc2/`.

## HISTORY

## BUGS

None known at this time.

## AUTHOR

This man page written by Alex Joni, as part of the Enhanced Machine Controller (EMC) project.

## REPORTING BUGS

Report bugs to alex\_joni AT users DOT sourceforge DOT net

## COPYRIGHT

Copyright © 2007 Alex Joni.

This is free software; see the source for copying conditions. There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

## NAME

comp – Build, compile and install EMC HAL components

## SYNOPSIS

```
comp [--compile|--preprocess|--document|--view-doc] compfile...
sudo comp [--install|--install-doc] compfile...
comp --compile --userspace cfile...
sudo comp --install --userspace cfile...
sudo comp --install --userspace pyfile...
```

## DESCRIPTION

**comp** performs many different functions:

- Compile **.comp** and **.c** files into **.so** or **.ko** HAL realtime components (the **--compile** flag)
- Compile **.comp** and **.c** files into HAL userspace components (the **--compile --userspace** flag)
- Preprocess **.comp** files into **.c** files (the **--preprocess** flag)
- Extract documentation from **.comp** files into **.9** manpage files (the **--document** flag)
- Display documentation from **.comp** files onscreen (the **--view-doc** flag)
- Compile and install **.comp** and **.c** files into the proper directory for HAL realtime components (the **--install** flag), which may require *sudo* to write to system directories.
- Install **.c** and **.py** files into the proper directory for HAL userspace components (the **--install --userspace** flag), which may require *sudo* to write to system directories.
- Extract documentation from **.comp** files into **.9** manpage files in the proper system directory (the **--install** flag), which may require *sudo* to write to system directories.
- Preprocess **.comp** files into **.c** files (the **--preprocess** flag)

## SEE ALSO

*Comp: A tool for creating HAL components* in the emc2 documentation for a full description of the **.comp** syntax, along with examples

**pydoc hal** and *Creating Userspace Python Components with the 'hal' module* for documentation on the Python interface to HAL components

**comp(9)** for documentation on the "two input comparator with hysteresis", a HAL realtime component with the same name as this program

**NAME**

`hal_input` – control HAL pins with any Linux input device, including USB HID devices

**SYNOPSIS**

`loadusr hal_input [-KRAL] inputspec ...`

**DESCRIPTION**

`hal_input` is an interface between HAL and any Linux input device, including USB HID devices. For each device named, **hal\_input** creates pins corresponding to its keys, absolute axes, and LEDs. At a fixed rate of approximately 10ms, it synchronizes the device and the HAL pins.

**INPUT SPECIFICATION**

The *inputspec* may be in one of several forms:

A string *S*

A substring or shell-style pattern match will be tested against the "name" of the device, the "phys" (which gives information about how it is connected), and the "id", which is a string of the form "Bus=... Vendor=... Product=... Version=...". You can view the name, phys, and id of attached devices by executing **less /proc/bus/input/devices**. Examples:

```
SpaceBall
"Vendor=001f Product=0001"
serio*/input0
```

A number *N*

This opens `/dev/input/eventN`. Except for devices that are always attached to the system, this number may change over reboots or when the device is removed. For this reason, using an integer is not recommended.

If the first character of the *inputspec* is a "+", then **hal\_input** requests exclusive access to the device. The first device matching an *inputspec* is used. Any number of *inputspecs* may be used.

A *subset option* may precede each *inputspec*. The subset option begins with a dash. Each letter in the subset option specifies a device feature to **include**. Features that are not specified are excluded. For instance, to export keyboard LEDs to HAL without exporting keys, use

```
hal_input -L keyboard ...
```

**DEVICE FEATURES SUPPORTED**

- EV\_KEY (buttons and keys). Subset -K
- EV\_ABS (absolute analog inputs). Subset -A
- EV\_REL (relative analog inputs). Subset -R
- EV\_LED (LED outputs). Subset -L

**HAL PINS AND PARAMETERS****For buttons**

**input.N.btn-name** bit out

**input.N.btn-name-not** bit out

Created for each button on the device.

**For keys**

**input.N.key-name**

**input.N.key-name-not**

Created for each key on the device.

**For absolute axes**

**input.N.abs-name-counts** s32 out

**input.N.abs-name-position** float out

**input.N.abs-name-scale** parameter float rw

**input.N.abs-name-offset** parameter float rw

**input.N.abs-name-fuzz** parameter s32 rw

**input.N.abs-name-flat** parameter s32 rw

**input.N.abs-name-min** parameter s32 r

**input.N.abs-name-max** parameter s32 r

Created for each absolute axis on the device. Device positions closer than **flat** to **offset** are reported as **offset** in **counts**, and **counts** does not change until the device position changes by at least **fuzz**. The position is computed as **position** = (**counts** - **offset**) / **scale**. The default value of **scale** and **offset** map the range of the axis reported by the operating system to [-1,1]. The default values of **fuzz** and **flat** are those reported by the operating system. The values of **min** and **max** are those reported by the operating system.

#### For relative axes

**input.N.rel-name-counts** s32 out

**input.N.rel-name-position** float out

**input.N.rel-name-reset** bit in

**input.N.rel-name-scale** parameter float rw

Created for each relative axis on the device. As long as **reset** is true, **counts** is reset to zero regardless of any past or current axis movement. Otherwise, **counts** increases or decreases according to the motion of the axis. **counts** is divided by position-scale to give **position**. The default value of **position** is 1.

#### For LEDs

**input.N.led-name** bit out

**input.N.led-name-invert** parameter bit rw

Created for each LED on the device.

## PERMISSIONS AND UDEV

By default, the input devices may not be accessible to regular users--**hal\_input** requires read-write access, even if the device has no outputs. To change the default permission of a device, add a new file to `/etc/udev/rules.d` to set the device's GROUP to "plugdev". You can do this for all input devices with this rule:

```
SUBSYSTEM="input", mode="0660", group="plugdev"
```

You can also make more specific rules for particular devices. For instance, a SpaceBall input device uses the 'spaceball' kernel module, so a udev entry for it would read:

```
DRIVER="spaceball", MODE="0660", GROUP="plugdev"
```

the next time the device is attached to the system, it will be accessible to the "plugdev" group.

For USB devices, the udev line would refer to the device's Vendor and Product values, such as

```
SYSFS{idProduct}=="c00e", SYSFS{idVendor}=="046d", MODE="0660", GROUP="plugdev"
```

for a particular logitech-brand mouse.

For more information on writing udev rules, see **udev(8)**.

## BUGS

The initial state of keys, buttons, and absolute axes are erroneously reported as FALSE or 0 until an event is received for that key, button, or axis.

## SEE ALSO

**hal\_joystick(1)**, **udev(8)**

**NAME**

`hal_joystick` – control HAL pins with a joystick

**SYNOPSIS**

`hal_joystick [-d device] [-p prefix]`

**DESCRIPTION**

**hal\_joystick** allows a joystick to generate HAL (Hardware Abstraction Layer) signals. Although not a hard realtime component, it is quite responsive under moderate system load. It provides analog (float) HAL pins for each joystick axis, and digital (bit) pins for each joystick button or trigger.

**OPTIONS**

**-d *device***

use *device* as the joystick device (default is `/dev/input/js0`).

**-p *prefix***

use *prefix* for the HAL pin names (default is `"joystick.0"`).

**USAGE**

**hal\_joystick** runs forever until interrupted with SIGINT or SIGTERM. Normally it would be invoked as **hal\_joystick &** to run in the background.

For each joystick axis, it exports a HAL float pin called "`<prefix>.axis.<N>`" where N is an integer, starting at zero. The value of the pin will range from -1.0 to +1.0 as the axis is moved thru its range of motion.

For each joystick button, it exports a HAL bit pin called "`<prefix>.button.<M>`" where M is also an integer starting at zero.

The mapping of axis and buttons to N and M are joystick dependent, as is the direction of motion that results in positive values of the axis pin. **hal\_joystick** uses the numbering and direction that is reported by the Linux joystick driver. For modern USB or other digital joysticks, the Linux driver figures out the number of axis and buttons automatically. For older analog joysticks, the driver may need configured by the user. See Linux documentation for more details. Once the Linux driver is properly configured, the HAL driver will configure itself to match automatically.

**SEE ALSO****HISTORY****BUGS**

Perhaps the analog axes should have a "scale" parameter that could be used to scale the -1.0 to +1.0 range to whatever the user needs. It would also allow the direction of an axis to be reversed by using a negative scale. This can already be done using a HAL scale block, but a built-in scale parameter would be more convenient.

**AUTHOR**

Written by John Kasunich, as part of the Enhanced Machine Controller (EMC) project.

**REPORTING BUGS**

Report bugs to `jmkasunich AT users DOT sourceforge DOT net`

**COPYRIGHT**

Copyright © 2003 John Kasunich.

This is free software; see the source for copying conditions. There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.



**NAME**

**halcmd** – manipulate the Enhanced Machine Controller HAL from the command line

**SYNOPSIS**

**halcmd** [*OPTIONS*] [*COMMAND*] [*ARG*]

**halrun** [*-I*] [*HALCMD OPTIONS*]

**halrun** [*-U*]

**DESCRIPTION**

**halcmd** is used to manipulate the HAL (Hardware Abstraction Layer) from the command line. **halcmd** can optionally read commands from a file, allowing complex HAL configurations to be set up with a single command.

**halrun** is a convenience script which sets up the realtime environment, executes **halcmd** with the given arguments, optionally runs an interactive **halcmd -kf** if *-I* is given, then tears down the realtime environment.

If the **readline** library is available when emc is compiled, then **halcmd** offers commandline editing and completion when running interactively. Use the up arrow to recall previous commands, and press tab to complete the names of items such as pins and signals.

**OPTIONS**

- I** Before tearing down the realtime environment, run an interactive **halcmd**. **halrun** only. **-I** must precede all other commandline arguments.
- f** [*file*] Ignore commands on command line, take input from *file* instead. If *file* is not specified, take input from *stdin*.
- i** *inifile* Use variables from *inifile* for substitutions. See **SUBSTITUTION** below.
- k** Keep going after failed command(s). The default is to stop and return failure if any command fails.
- q** display errors only (default)
- Q** display nothing, execute commands silently
- s** Script-friendly mode. In this mode, *show* will not output titles for the items shown. Also, module names will be printed instead of ID codes in pin, param, and funct listings. Threads are printed on a single line, with the thread period, FP usage and name first, followed by all of the functions in the thread, in execution order. Signals are printed on a single line, with the type, value, and signal name first, followed by a list of pins connected to the signal, showing both the direction and the pin name. No prompt will be printed if both **-s** and **-f** are specified.
- R** Release the HAL mutex. This is useful for recovering when a HAL component has crashed while holding the HAL mutex.
- U** Forcibly cause the realtime environment to exit. It releases the HAL mutex, requests that all HAL components unload, and stops the realtime system. **halrun** only. **-U** must be the only commandline argument.
- v** display results of each command
- V** display lots of debugging junk
- h** [*command*] display a help screen and exit, displays extended help on *command* if specified

**COMMANDS**

Commands tell **halcmd** what to do. Normally **halcmd** reads a single command from the command line and executes it. If the **-f** option is used to read commands from a file, **halcmd** reads each line of the file as a new command. Anything following **#** on a line is a comment.

**loadrt** *modname*

(load realtime module) Loads a realtime HAL module called *modname*. **halcmd** looks for the module in a directory specified at compile time.

In systems with realtime, **halcmd** calls the **emc\_module\_helper** to load realtime modules. **emc\_module\_helper** is a setuid program and is compiled with a whitelist of modules it is allowed to load. This is currently just a list of **EMC**-related modules. The **emc\_module\_helper** execs **insmod**, so return codes and error messages are those from **insmod**. Administrators who wish to restrict which users can load these **EMC**-related kernel modules can do this by setting the permissions and group on **emc\_module\_helper** appropriately.

In systems without realtime **halcmd** calls the **rtapi\_app** which creates the simulated realtime environment if it did not yet exist, and then loads the requested component with a call to **dlopen(3)**.

**unloadrt** *modname*

(unload realtime module) Unloads a realtime HAL module called *modname*. If *modname* is "all", it will unload all currently loaded realtime HAL modules. **unloadrt** also works by execing **emc\_module\_helper** or **rtapi\_app**, just like **loadrt**.

**loadusr** [*flags*] *unix-command*

(load Userspace component) Executes the given *unix-command*, usually to load a userspace component. [*flags*] may be one or more of:

- **-W** to wait for the component to become ready. The component is assumed to have the same name as the first argument of the command.
- **-Wn name** to wait for the component, which will have the given name.
- **-w** to wait for the program to exit
- **-i** to ignore the program return value (with -w)

**waitusr** *name*

(wait for Userspace component) Waits for user space component *name* to disconnect from HAL (usually on exit). The component must already be loaded. Usefull near the end of a HAL file to wait until the user closes some user interface component before cleaning up and exiting.

**unloadusr** *compname*

(unload Userspace component) Unloads a userspace component called *compname*. If *compname* is "all", it will unload all userspace components. **unloadusr** works by sending SIGTERM to all userspace components.

**unload** *compname*

Unloads a userspace component or realtime module. If *compname* is "all", it will unload all userspace components and realtime modules.

**newsig** *signame type*

(new signal) Creates a new HAL signal called *signame* that may later be used to connect two or more HAL component pins. *type* is the data type of the new signal, and must be one of "bit", "s32", "u32", or "float". Fails if a signal of the same name already exists.

**delsig** *signame*

(delete signal) Deletes HAL signal *signame*. Any pins currently linked to the signal will be unlinked. Fails if *signame* does not exist.

**setsig** *signame value*

(set signal) Sets the value of signal *signame* to *value*. Fails if *signame* does not exist, if it already has a writer, or if *value* is not a legal value. Legal values depend on the signals's type.

**stype** *name*

(signal type) Gets the type of signal *name*. Fails if *name* does not exist as a signal.

**gets** *signame*

(get signal) Gets the value of signal *signame*. Fails if *signame* does not exist.

**linkps** *pinname* [*arrow*] *signame*

(link pin to signal) Establishes a link between a HAL component pin *pinname* and a HAL signal *signame*. Any previous link to *pinname* will be broken. *arrow* can be "=", "<=", "<=>", or omitted. **halcmd** ignores arrows, but they can be useful in command files to document the direction of data flow. Arrows should not be used on the command line since the shell might try to interpret them. Fails if either *pinname* or *signame* does not exist, or if they are not the same type type.

**linksp** *signame* [*arrow*] *pinname*

(link signal to pin) Works like **linkps** but reverses the order of the arguments. **halcmd** treats both link commands exactly the same. Use whichever you prefer.

**linkpp** *pinname1* [*arrow*] *pinname2*

(OBSOLETE - use **net** instead) (*link pin to pin*) Shortcut for **linkps** that creates the signal (named like the first pin), then links them both to that signal. **halcmd** treats this just as if it were:

```
halcmd newsig pinname1
halcmd linksp pinname1 pinname1
halcmd linksp pinname1 pinname2
```

**net** *signame* *pinname* ...

Create *signame* to match the type of *pinname* if it does not yet exist. Then, link *signame* to each *pinname* in turn. Arrows may be used as in **linkps**.

**unlinkp** *pinname*

(unlink pin) Breaks any previous link to *pinname*. Fails if *pinname* does not exist.

**setp** *name* *value*

(set parameter or pin) Sets the value of parameter or pin *name* to *value*. Fails if *name* does not exist as a pin or parameter, if it is a parameter that is not writable, if it is a pin that is an output, if it is a pin that is already attached to a signal, or if *value* is not a legal value. Legal values depend on the type of the pin or parameter. If a pin and a parameter both exist with the given name, the parameter is acted on.

*paramname* = *value*

*pinname* = *value*

Identical to **setp**. This alternate form of the command may be more convenient and readable when used in a file.

**ptype** *name*

(parameter or pin type) Gets the type of parameter or pin *name*. Fails if *name* does not exist as a pin or parameter. If a pin and a parameter both exist with the given name, the parameter is acted on.

**getp** *name*

(get parameter or pin) Gets the value of parameter or pin *name*. Fails if *name* does not exist as a pin or parameter. If a pin and a parameter both exist with the given name, the parameter is acted on.

**addf** *funcname* *threadname*

(add function) Adds function *funcname* to realtime thread *threadname*. *funcname* will run after any functions that were previously added to the thread. Fails if either *funcname* or *threadname* does not exist, or if they are incompatible.

**delf** *funcname* *threadname*

(delete function) Removes function *funcname* from realtime thread *threadname*. Fails if either *funcname* or *threadname* does not exist, or if *funcname* is not currently part of *threadname*.

**start** Starts execution of realtime threads. Each thread periodically calls all of the functions that were added to it with the **addf** command, in the order in which they were added.

**stop** Stops execution of realtime threads. The threads will no longer call their functions.

**show** [*item*]

Prints HAL items to *stdout* in human readable format. *item* can be one of "**comp**" (components), "**pin**", "**sig**" (signals), "**param**" (parameters), "**funct**" (functions), or "**thread**". The type "**all**" can be used to show matching items of all the preceding types. If *item* is omitted, **show** will print everything.

**item** This is equivalent to **show all** [*item*].

**save** [*item*]

Prints HAL items to *stdout* in the form of HAL commands. These commands can be redirected to a file and later executed using **halcmd -f** to restore the saved configuration. *item* can be one of the following: "**comp**" generates a **loadrt** command for realtime component. "**sig**" generates a **newsig** command for each signal, and "**sigu**" generates a **newsig** command for each unlinked signal (for use with **netl** and **netla**). "**link**" and "**linka**" both generate **linkps** commands for each link. (**linka** includes arrows, while **link** does not.) "**net**" and "**neta**" both generate one **newsig** command for each signal, followed by **linkps** commands for each pin linked to that signal. (**neta** includes arrows.) "**netl**" generates one **net** command for each linked signal, and "**netla**" generates a similar command using arrows. "**param**" generates one **setp** command for each parameter. "**thread**" generates one **addf** command for each function in each realtime thread. If *item* is omitted, **save** does the equivalent of **comp**, **sigu**, **link**, **param**, and **thread**.

**source** *filename.hal*

Execute the commands from *filename.hal*.

## SUBSTITUTION

After a command is read but before it is executed, several types of variable substitution take place.

### Environment Variables

Environment variables have the following formats:

**\$ENVVAR** followed by end-of-line or whitespace

**\$(ENVVAR)**

### Infile Variables

Infile variables are available only when an infile was specified with the **halcmd -i** flag. They have the following formats:

**[SECTION]VAR** followed by end-of-line or whitespace

**[SECTION](VAR)**

## EXAMPLES

## SEE ALSO

## HISTORY

## BUGS

None known at this time.

## AUTHOR

Original version by John Kasunich, as part of the Enhanced Machine Controller (EMC) project. Now includes major contributions by several members of the project.

## REPORTING BUGS

Report bugs to the [emc bug tracker](http://sf.net/tracker/?group_id=6744&atid=106744) ([http://sf.net/tracker/?group\\_id=6744&atid=106744](http://sf.net/tracker/?group_id=6744&atid=106744)).

## COPYRIGHT

Copyright © 2003 John Kasunich.

This is free software; see the source for copying conditions. There is NO warranty; not even for

MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

**NAME**

halmeter – observe HAL pins, signals, and parameters

**SYNOPSIS**

**halmeter** [-s] [**pin**|**sig**|**param** *name*]

**DESCRIPTION**

**halmeter** is used to observe HAL (Hardware Abstraction Layer) pins, signals, or parameters. It serves the same purpose as a multimeter does when working on physical systems.

**OPTIONS**

**pin** *name*

display the HAL pin *name*.

**sig** *name*

display the HAL signal *name*.

**param** *name*

display the HAL parameter *name*.

If neither **pin**, **sig**, or **param** are specified, the window starts out blank and the user must select an item to observe.

**-s** small window. Non-interactive, must be used with **pin**, **sig**, or **param** to select the item to display. The item name is displayed in the title bar instead of the window, and there are no "Select" or "Exit" buttons. Handy when you want a lot of meters in a small space.

**USAGE**

Unless **-s** is specified, there are two buttons, "Select" and "Exit". "Select" opens a dialog box to select the item (pin, signal, or parameter) to be observed. "Exit" does what you expect.

The selection dialog has "OK", "Apply", and "Cancel" buttons. OK displays the selected item and closes the dialog. "Apply" displays the selected item but keeps the selection dialog open. "Cancel" closes the dialog without changing the displayed item.

**EXAMPLES**

**halmeter**

Opens a meter window, with nothing initially displayed. Use the "Select" button to choose an item to observe. Does not return until the window is closed.

**halmeter &**

Open a meter window, with nothing initially displayed. Use the "Select" button to choose an item. Runs in the background leaving the shell free for other commands.

**halmeter pin parport.0.pin-03-out &**

Open a meter window, initially displaying HAL pin *parport.0.pin-03-out*. The "Select" button can be used to display other items. Runs in background.

**halmeter -s pin parport.0.pin-03-out &**

Open a small meter window, displaying HAL pin *parport.0.pin-03-out*. The displayed item cannot be changed. Runs in background.

**SEE ALSO****HISTORY****BUGS****AUTHOR**

Original version by John Kasunich, as part of the Enhanced Machine Controller (EMC) project. Improvements by several other members of the EMC development team.

**REPORTING BUGS**

Report bugs to [jmkasunich](mailto:jmkasunich) AT users DOT sourceforge DOT net

**COPYRIGHT**

Copyright © 2003 John Kasunich.

This is free software; see the source for copying conditions. There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

**NAME**

**halsampler** – sample data from HAL in realtime

**SYNOPSIS**

**halsampler** [*options*]

**DESCRIPTION**

**sampler**(9) and **halsampler** are used together to sample HAL data in real time and store it in a file. **sampler** is a realtime HAL component that exports HAL pins and creates a FIFO in shared memory. It then begins sampling data from the HAL and storing it to the FIFO. **hal\_sampler** is a user space program that copies data from the FIFO to stdout, where it can be redirected to a file or piped to some other program.

**OPTIONS**

**-c** *CHAN*

instructs **halsampler** to read from FIFO *CHAN*. FIFOs are numbered from zero, and the default value is zero, so this option is not needed unless multiple FIFOs have been created.

**-n** *COUNT*

instructs **halsampler** to read *COUNT* samples from the FIFO, then exit. If **-n** is not specified, **halsampler** will read continuously until it is killed.

**-t** instructs **halsampler** to tag each line by printing the sample number in the first column.

**USAGE**

A FIFO must first be created by loading **sampler**(9) with **halcmd loadrt** or a **loadrt** command in a .hal file. Then **halsampler** can be invoked to begin printing data from the FIFO to stdout.

Data is printed one line per sample. If **-t** was specified, the sample number is printed first. The data follows, in the order that the pins were defined in the config string. For example, if the **sampler** config string was "ffbs" then a typical line of output (without **-t**) would look like:

```
123.55 33.4 0 -12
```

**halsampler** prints data as fast as possible until the FIFO is empty, then it retries at regular intervals, until it is either killed or has printed *COUNT* samples as requested by **-n**. Usually, but not always, data printed by **halsampler** will be redirected to a file or piped to some other program.

The FIFO size should be chosen to absorb samples captured during any momentary disruptions in the flow of data, such as disk seeks, terminal scrolling, or the processing limitations of subsequent program in a pipeline. If the FIFO gets full and **sampler** is forced to overwrite old data, **halsampler** will print 'overrun' on a line by itself to mark each gap in the sampled data. If **-t** was specified, gaps in the sequential sample numbers in the first column can be used to determine exactly how many samples were lost.

The data format for **halsampler** output is the same as for **halstreamer**(1) input, so 'waveforms' captured with **halsampler** can be replayed using **halstreamer**. The **-t** option should not be used in this case.

**EXIT STATUS**

If a problem is encountered during initialization, **halsampler** prints a message to stderr and returns failure.

Upon printing *COUNT* samples (if **-n** was specified) it will shut down and return success. If it is terminated before printing the specified number of samples, it returns failure. This means that when **-n** is not specified, it will always return failure when terminated.

**SEE ALSO**

**sampler**(9) **streamer**(9) **halstreamer**(1)



**HISTORY****BUGS****AUTHOR**

Original version by John Kasunich, as part of the Enhanced Machine Controller (EMC) project.  
Improvements by several other members of the EMC development team.

**REPORTING BUGS**

Report bugs to [jmkasunich](mailto:jmkasunich) AT users DOT sourceforge DOT net

**COPYRIGHT**

Copyright © 2006 John Kasunich.

This is free software; see the source for copying conditions. There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

**NAME**

**halstreamer** – stream file data into HAL in real time

**SYNOPSIS**

**halstreamer** [*options*]

**DESCRIPTION**

**streamer**(9) and **halstreamer** are used together to stream data from a file into the HAL in real time. **streamer** is a realtime HAL component that exports HAL pins and creates a FIFO in shared memory. **hal\_streamer** is a user space program that copies data from stdin into the FIFO, so that **streamer** can write it to the HAL pins.

**OPTIONS**

**-c** *CHAN*

instructs **halstreamer** to write to FIFO *CHAN*. FIFOs are numbered from zero, and the default value is zero, so this option is not needed unless multiple FIFOs have been created.

**USAGE**

A FIFO must first be created by loading **streamer**(9) with **halcmd loadrt** or a **loadrt** command in a .hal file. Then **halstreamer** can be invoked to begin writing data into the FIFO.

Data is read from stdin, and is almost always either redirected from a file or piped from some other program, since keyboard input would be unable to keep up with even slow streaming rates.

Each line of input must match the pins that are attached to the FIFO, for example, if the **streamer** config string was "ffbs" then each line of input must consist of two floats, a bit, and a signed integer, in that order and separated by whitespace. Floats must be formatted as required by **strtod**(3), signed and unsigned integers must be formatted as required by **strtol**(3) and **strtoul**(3), and bits must be either '0' or '1'.

**halstreamer** transfers data to the FIFO as fast as possible until the FIFO is full, then it retries at regular intervals, until it is either killed or reads **EOF** from stdin. Data can be redirected from a file or piped from some other program.

The FIFO size should be chosen to ride through any momentary disruptions in the flow of data, such as disk seeks. If the FIFO is big enough, **halstreamer** can be restarted with the same or a new file before the FIFO empties, resulting in a continuous stream of data.

The data format for **halstreamer** input is the same as for **halsampler**(1) output, so 'waveforms' captured with **halsampler** can be replayed using **halstreamer**.

**EXIT STATUS**

If a problem is encountered during initialization, **halstreamer** prints a message to stderr and returns failure.

If a badly formatted line is encountered while writing to the FIFO, it prints a message to stderr, skips the line, and continues (this behavior may be revised in the future).

Upon reading **EOF** from the input, it returns success. If it is terminated before the input ends, it returns failure.

**SEE ALSO**

**streamer**(9) **sampler**(9) **halsampler**(1)

**HISTORY****BUGS****AUTHOR**

Original version by John Kasunich, as part of the Enhanced Machine Controller (EMC) project. Improvements by several other members of the EMC development team.

**REPORTING BUGS**

Report bugs to jmkasunich AT users DOT sourceforge DOT net

**COPYRIGHT**

Copyright © 2006 John Kasunich.

This is free software; see the source for copying conditions. There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

**NAME**

halui – observe HAL pins and command EMC through NML

**SYNOPSIS**

**halui** [-ini <path-to-ini>]

**DESCRIPTION**

**halui** is used to build a User Interface using hardware knobs and switches. It exports a big number of pins, and acts accordingly when these change.

**OPTIONS****-ini name**

use the *name* as the configuration file. Note: halui must find the nml file specified in the ini, usually that file is in the same folder as the ini, so it makes sense to run halui from that folder.

**USAGE**

When run, **halui** will export a large number of pins. A user can connect those to his physical knobs & switches & leds, and when a change is noticed halui triggers an appropriate event.

**halui** expects the signals to be debounced, so if needed (bad knob contact) connect the physical button to a HAL debounce filter first.

**EXPORTED PINS****machine**

*halui.machine.on*

pin for setting machine On

*halui.machine.off*

pin for setting machine Off

*halui.machine.is-on*

pin for machine is On/Off

**estop**

*halui.estop.activate*

pin for setting Estop (emc internal) On

*halui.estop.reset*

pin for resetting Estop (emc internal) Off

*halui.estop.is-activated*

pin for displaying Estop state (emc internal) On/Off

**mode**

*halui.mode.manual*

pin for requesting manual mode

*halui.mode.is\_manual*

pin for manual mode is on

*halui.mode.auto*

pin for requesting auto mode

*halui.mode.is\_auto*

pin for auto mode is on

*halui.mode.mdi*

pin for requesting mdi mode

*halui.mode.is\_mdi*

pin for mdi mode is on

*halui.mode.teleop*  
pin for requesting coordinated jog mode

*halui.mode.is\_teleop*  
pin showing coordinated jog mode is on

*halui.mode.joint*  
pin for requesting joint by joint jog mode

*halui.mode.is\_joint*  
pin showing joint by joint jog mode is on

**mist, flood, lube**

*halui.mist.on*  
pin for starting mist

*halui.mist.off*  
pin for stoping mist

*halui.mist.is-on*  
pin for mist is on

*halui.flood.on*  
pin for starting flood

*halui.flood.off*  
pin for stoping flood

*halui.flood.is-on*  
pin for flood is on

*halui.lube.on*  
pin for starting lube

*halui.lube.off*  
pin for stoping lube

*halui.lube.is-on*  
pin for lube is on

**spindle**

*halui.spindle.start*  
pin for starting the spindle

*halui.spindle.stop*  
pin for stopping the spindle

*halui.spindle.forward*  
pin for making the spindle go forward

*halui.spindle.reverse*  
pin for making the spindle go reverse

*halui.spindle.increase*  
pin for making the spindle go faster

*halui.spindle.decrease*  
pin for making the spindle go slower

*halui.spindle.brake-on*  
pin for activating the spindle brake

*halui.spindle.brake-off*  
pin for deactivating the spindle brake

*halui.spindle.brake-is-on*

status pin that tells us if brake is on

### **joint**

*halui.joint.x.home*

pin for homing the specific joint (x = 0..7)

*halui.joint.x.is-homed*

status pin telling that the joint is homed (x = 0..7)

*halui.joint.selected.home*

pin for homing the selected joint

*halui.joint.selected.is-homed*

status pin telling that the selected joint is homed

*halui.joint.x.on-soft-min-limit*

status pin telling that the joint is on the negative software limit (x=0..7, selected)

*halui.joint.x.on-soft-max-limit*

status pin telling that the joint is on the positive software limit (x=0..7, selected)

*halui.joint.x.on-hard-min-limit*

status pin telling that the joint is on the negative hardware limit (x=0..7, selected)

*halui.joint.x.on-hard-max-limit*

status pin telling that the joint is on the positive hardware limit (x=0..7, selected)

*halui.joint.x.has-fault*

status pin telling that the joint has a fault (x = 0..7, selected)

*halui.joint.select*

select joint (value = 0..7)

*halui.joint.selected*

selected joint (value = 0..7)

*halui.joint.x.select*

pins for selecting a joint (x = 0..7)

*halui.joint.x.is-selected*

status pin that a joint is selected (x = 0..7)

### **jogging**

*halui.jog.speed*

pin for setting jog speed. will be used for minus/plus jogging.

*halui.jog.deadband*

pin for setting jog analog deadband (where not to move)

*halui.jog.N.minus*

pin for jogging axis N in negative direction at the *halui.jog.speed* velocity

*halui.jog.N.plus*

pin for jogging axis N in positive direction at the *halui.jog.speed* velocity

*halui.jog.N.analog*

pin for jogging the axis X using an float value (e.g. joystick)

*halui.jog.selected.minus*

pin for jogging the selected axis in negative direction at the *halui.jog.speed* velocity

*halui.jog.selected.plus*

pin for jogging the selected axis in positive direction at the *halui.jog.speed* velocity

**tool***halui.tool.number*

current selected tool

*halui.tool.length-offset*

current applied tool-length-offset

**program***halui.program.is-idle*

status pin telling that no program is running

*halui.program.is-running*

status pin telling that a program is running

*halui.program.is-paused*

status pin telling that a program is paused

*halui.program.run*

pin for running a program

*halui.program.pause*

pin for pausing a program

*halui.program.resume*

pin for resuming a program

*halui.program.step*

pin for stepping in a program

*halui.program.stop*pin for stopping a program (note: this pin does the same thing as *halui.abort*)**general***halui.abort*

pin to send an abort message (clears out most errors)

**feed-override***halui.feed-override.value*

current Feed Override value

*halui.feed-override.scale*

pin for setting the scale on changing the FO

*halui.feed-override.counts*

counts from an encoder to change FO

*halui.feed-override.increase*

pin for increasing the FO (+scale)

*halui.feed-override.decrease*

pin for decreasing the FO (-scale)

**spindle-override***halui.spindle-override.value*

current FO value

*halui.spindle-override.scale*

pin for setting the scale on changing the SO

*halui.spindle-override.counts*

counts from an encoder for example to change SO

*halui.spindle-override.increase*  
pin for increasing the SO (+=scale)  
*halui.spindle-override.decrease*  
pin for decreasing the SO (-=scale)

**SEE ALSO****HISTORY****BUGS**

none known at this time.

**AUTHOR**

Written by Alex Joni, as part of the Enhanced Machine Controller (EMC2) project.

**REPORTING BUGS**

Report bugs to alex\_joni AT users DOT sourceforge DOT net

**COPYRIGHT**

Copyright © 2006 Alex Joni.

This is free software; see the source for copying conditions. There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.



**NAME**

iocontrol – accepts NML I/O commands, interacts with HAL in userspace

**SYNOPSIS**

**loadusr io** [-ini *inifile*]

**DESCRIPTION**

These pins are created by the userspace IO controller, usually found in \$EMC2\_HOME/bin/io

The signals are turned on and off in userspace - if you have strict timing requirements or simply need more i/o, consider using the realtime synchronized i/o provided by **motion**(9) instead.

The inifile is searched for in the directory from which halcmd was run, unless an absolute path is specified.

**PINS****iocontrol.0.coolant-flood**

TRUE when flood coolant is requested

**iocontrol.0.coolant-mist**

TRUE when mist coolant is requested

**iocontrol.0.emc-enable-in**

Should be driven FALSE when an external estop condition exists.

**iocontrol.0.lube**

TRUE when lube is requested

**iocontrol.0.lube\_level**

Should be driven FALSE when lubrication tank is empty.

**iocontrol.0.tool-change**

TRUE when a tool change is requested

**iocontrol.0.tool-changed**

Should be driven TRUE when a tool change is completed.

**iocontrol.0.tool-prep-number**

The number of the next tool, from the RS274NGC T-word

**iocontrol.0.tool-prepare**

TRUE when a *Tn* tool prepare is requested

**iocontrol.0.tool-prepared**

Should be driven TRUE when a tool prepare is completed.

**iocontrol.0.user-enable-out**

FALSE when an internal estop condition exists

**iocontrol.0.user-request-enable**

TRUE when the user has requested that estop be cleared

**SEE ALSO**

**motion(9)**

**NAME**

pyvcp – Virtual Control Panel for EMC2

**SYNOPSIS**

**pyvcp** [-c *component-name*] *myfile.xml*

**OPTIONS**

**-c** *component-name*

Use *component-name* as the HAL component name. If the component name is not specified, the basename of the xml file is used.

**SEE ALSO**

*Virtual Control Panels* in the emc2 documentation for a description of the xml syntax, along with examples

## NAME

hal – Introduction to the HAL API

## DESCRIPTION

HAL stands for Hardware Abstraction Layer, and is used by EMC to transfer realtime data to and from I/O devices and other low-level modules.

**hal.h** defines the API and data structures used by the HAL. This file is included in both realtime and non-realtime HAL components. HAL uses the RTPAI real time interface, and the #define symbols RTAPI and ULAPI are used to distinguish between realtime and non-realtime code. The API defined in this file is implemented in `hal_lib.c` and can be compiled for linking to either realtime or user space HAL components.

The HAL is a very modular approach to the low level parts of a motion control system. The goal of the HAL is to allow a systems integrator to connect a group of software components together to meet whatever I/O requirements he (or she) needs. This includes realtime and non-realtime I/O, as well as basic motor control up to and including a PID position loop. What these functions have in common is that they all process signals. In general, a signal is a data item that is updated at regular intervals. For example, a PID loop gets position command and feedback signals, and produces a velocity command signal.

HAL is based on the approach used to design electronic circuits. In electronics, off-the-shelf components like integrated circuits are placed on a circuit board and their pins are interconnected to build whatever overall function is needed. The individual components may be as simple as an op-amp, or as complex as a digital signal processor. Each component can be individually tested, to make sure it works as designed. After the components are placed in a larger circuit, the signals connecting them can still be monitored for testing and troubleshooting.

Like electronic components, HAL components have pins, and the pins can be interconnected by signals.

In the HAL, a *signal* contains the actual data value that passes from one pin to another. When a signal is created, space is allocated for the data value. A *pin* on the other hand, is a pointer, not a data value. When a pin is connected to a signal, the pin's pointer is set to point at the signal's data value. This allows the component to access the signal with very little run-time overhead. (If a pin is not linked to any signal, the pointer points to a dummy location, so the realtime code doesn't have to deal with null pointers or treat unlinked variables as a special case in any way.)

There are three approaches to writing a HAL component. Those that do not require hard realtime performance can be written as a single user mode process. Components that need hard realtime performance but have simple configuration and init requirements can be done as a single kernel module, using either pre-defined init info, or insmod-time parameters. Finally, complex components may use both a kernel module for the realtime part, and a user space process to handle ini file access, user interface (possibly including GUI features), and other details.

HAL uses the RTAPI/ULAPI interface. If RTAPI is #defined `hal_lib.c` would generate a kernel module `hal_lib.o` that is insmoded and provides the functions for all kernel module based components. The same source file compiled with the ULAPI #define would make a user space `hal_lib.o` that is statically linked to user space code to make user space executables. The variable lists and link information are stored in a block of shared memory and protected with mutexes, so that kernel modules and any of several user mode programs can access the data.

## HAL STATUS CODES

HAL\_SUCCESS  
call successful

HAL\_UNSUP  
function not supported

HAL\_BADVAR  
duplicate or not-found variable name

HAL\_INVALID  
invalid argument

HAL\_NOMEM  
not enough memory

HAL\_LIMIT  
resource limit reached

HAL\_PERM  
permission denied

HAL\_BUSY  
resource is busy or locked

HAL\_NOTFND  
object not found

HAL\_FAIL  
operation failed

**SEE ALSO**

**intro(3rtapi)**

**NAME**

hal\_add\_funct\_to\_thread – one-line description of hal\_add\_funct\_to\_thread

**SYNTAX**

```
int hal_add_funct_to_thread(char *funct_name, char *thread_name,
                           int position)
```

```
int hal_del_funct_from_thread(char *funct_name, char *thread_name)
```

**ARGUMENTS**

*funct\_name*

The name of the function

*thread\_name*

The name of the thread

*position*

The desired location within the thread. This determines when the function will run, in relation to other functions in the thread. A positive number indicates the desired location as measured from the beginning of the thread, and a negative is measured from the end. So +1 means this function will become the first one to run, +5 means it will be the fifth one to run, -2 means it will be next to last, and -1 means it will be last. Zero is illegal.

**DESCRIPTION**

**hal\_add\_funct\_to\_thread** adds a function exported by a realtime HAL component to a realtime thread. This determines how often and in what order functions are executed.

**hal\_del\_funct\_from\_thread** removes a function from a thread.

**RETURN VALUE**

Returns a HAL status code.

**REALTIME CONSIDERATIONS**

Call only from realtime init code, not from user space or realtime code.

**SEE ALSO**

**hal\_thread\_new(3hal), hal\_export\_funct(3hal)**

**NAME**

hal\_create\_thread – Create a HAL thread

**SYNTAX**

```
int hal_create_thread(char *name, unsigned long period, int uses_fp)
```

```
int hal_thread_delete(char *name)
```

**ARGUMENTS**

*name*     The name of the thread

*period*   The interval, in nanoseconds, between iterations of the thread

*uses\_fp*   Must be nonzero if a function which uses floating-point will be attached to this thread.

**DESCRIPTION**

**hal\_create\_thread** establishes a realtime thread that will execute one or more HAL functions periodically.

All thread periods are rounded to integer multiples of the hardware timer period, and the timer period is based on the first thread created. Threads must be created in order, from the fastest to the slowest. HAL assigns decreasing priorities to threads that are created later, so creating them from fastest to slowest results in rate monotonic priority scheduling.

**hal\_delete\_thread** deletes a previously created thread.

**REALTIME CONSIDERATIONS**

Call only from realtime init code, not from user space or realtime code.

**RETURN VALUE**

Returns a HAL status code.

**SEE ALSO**

**hal\_export\_funct(3hal)**

**NAME**

hal\_exit – Shut down HAL

**SYNTAX**

int hal\_exit(int *comp\_id*)

**ARGUMENTS**

*comp\_id*

A HAL component identifier returned by an earlier call to **hal\_init**.

**DESCRIPTION**

**hal\_exit** shuts down and cleans up HAL and RTAPI. It must be called prior to exit by any module that called **hal\_init**.

**REALTIME CONSIDERATIONS**

Call only from within user or init/cleanup code, not from realtime tasks.

**RETURN VALUE**

Returns a HAL status code.



**NAME**

hal\_export\_funct – create a realtime function callable from a thread

**SYNTAX**

```
typedef void(*hal_func_t)(void * arg, long period)  
int hal_export_funct(char *name, hal_func_t func, void *arg, int uses_fp, int reentrant, int comp_id)
```

**ARGUMENTS**

*name*     The name of the function.

*func*     The pointer to the function

*arg*       The argument to be passed as the first parameter of *func*

*uses\_fp*   Nonzero if the function uses floating-point operations, including assignment of floating point values with "=".

*reentrant*

If *reentrant* is non-zero, the function may be preempted and called again before the first call completes. Otherwise, it may only be added to one thread.

*comp\_id*

A HAL component identifier returned by an earlier call to **hal\_init**.

**DESCRIPTION**

**hal\_export\_funct** makes a realtime function provided by a component available to the system. A subsequent call to **hal\_add\_func\_to\_thread** can be used to schedule the execution of the function as needed by the system.

When this function is placed on a HAL thread, and HAL threads are started, *func* is called repeatedly with two arguments: *void \*arg* is the same value that was given to **hal\_export\_funct**, and *long period* is the interval between calls in nanoseconds.

Each call to the function should do a small amount of work and return.

**RETURN VALUE**

Returns a HAL status code.

**SEE ALSO**

**hal\_create\_thread(3hal)**, **hal\_add\_func\_to\_thread(3hal)**

**NAME**

hal\_init – Sets up HAL and RTAPI

**SYNTAX**

```
int hal_init(char *modname)
```

**ARGUMENTS**

*modname*

The name of this hal module

**DESCRIPTION**

**hal\_init** sets up HAL and RTAPI. It must be called by any module that intends to use the API, before any other RTAPI calls.

*modname* can optionally point to a string that identifies the module. The string will be truncated at **RTAPI\_NAME\_LEN** characters. If *modname* is **NULL**, the system will assign a name.

**REALTIME CONSIDERATIONS**

Call only from within user or init/cleanup code, not from realtime tasks.

**RETURN VALUE**

On success, returns a positive integer module ID, which is used for subsequent calls to hal and rtapi APIs.  
On failure, returns a HAL error code.

**NAME**

hal\_malloc – Allocate space in the HAL shared memory area

**SYNTAX**

```
void *hal_malloc(long int size)
```

**ARGUMENTS**

*size*      Gives the size, in bytes, of the block

**DESCRIPTION**

**hal\_malloc** allocates a block of memory from the main HAL shared memory area. It should be used by all components to allocate memory for HAL pins and parameters. It allocates ‘size’ bytes, and returns a pointer to the allocated space, or NULL (0) on error. The returned pointer will be properly aligned for any type HAL supports. A component should allocate during initialization all the memory it needs.

The allocator is very simple, and there is no ‘free’. The entire HAL shared memory area is freed when the last component calls **hal\_exit**. This means that if you continuously install and remove one component while other components are present, you eventually will fill up the shared memory and an install will fail. Removing all components completely clears memory and you start fresh.

**RETURN VALUE**

A pointer to the allocated space, which is properly aligned for any variable HAL supports. Returns NULL on error.

**NAME**

hal\_param\_new – Create a HAL parameter

**SYNTAX**

```
int hal_param_bit_new(char *name, hal_param_dir_t dir, hal_bit_t * data_addr, int comp_id)

int hal_param_float_new(char *name, hal_param_dir_t dir, hal_float_t * data_addr, int comp_id)

int hal_param_u32_new(char *name, hal_param_dir_t dir, hal_u32_t * data_addr, int comp_id)

int hal_param_s32_new(char *name, hal_param_dir_t dir, hal_s32_t * data_addr, int comp_id)

int hal_param_bit_newf(hal_param_dir_t dir, hal_bit_t * data_addr, int comp_id)

int hal_param_float_newf(hal_param_dir_t dir, hal_float_t * data_addr, int comp_id)

int hal_param_u32_newf(hal_param_dir_t dir, hal_u32_t * data_addr, int comp_id, char *fmt, ...)

int hal_param_s32_newf(hal_param_dir_t dir, hal_s32_t * data_addr, int comp_id, char *fmt, ...)

int hal_param_new(char *name, hal_type_t type, hal_in_dir_t dir, void *data_addr, int comp_id)
```

**ARGUMENTS**

*name*     The name to give to the created parameter

*dir*       The direction of the parameter, from the viewpoint of the component. It may be one of **HAL\_RO**, or **HAL\_RW**. A component may assign a value to any parameter, but other programs (such as hal-cmd) may only assign a value to a parameter that is **HAL\_RW**.

*data\_addr*  
            The address of the data, which must lie within memory allocated by **hal\_malloc**.

*comp\_id*  
            A HAL component identifier returned by an earlier call to **hal\_init**.

*fmt, ...*   A printf-style format string and arguments

*type*      The type of the parameter, as specified in **hal\_type\_t(3hal)**.

**DESCRIPTION**

The **hal\_param\_new** family of functions create a new *param* object.

There are functions for each of the data types that the HAL supports. Pins may only be linked to signals of the same type.

**RETURN VALUE**

Returns a HAL status code.

**SEE ALSO**

**hal\_type\_t(3hal)**

**NAME**

hal\_pin\_new – Create a HAL pin

**SYNTAX**

```
int hal_pin_bit_new(char *name, hal_pin_dir_t dir, hal_bit_t ** data_ptr_addr, int comp_id)

int hal_pin_float_new(char *name, hal_pin_dir_t dir, hal_float_t ** data_ptr_addr, int comp_id)

int hal_pin_u32_new(char *name, hal_pin_dir_t dir, hal_u32_t ** data_ptr_addr, int comp_id)

int hal_pin_s32_new(char *name, hal_pin_dir_t dir, hal_s32_t ** data_ptr_addr, int comp_id)

int hal_pin_bit_newf(hal_pin_dir_t dir, hal_bit_t ** data_ptr_addr, int comp_id)

int hal_pin_float_newf(hal_pin_dir_t dir, hal_float_t ** data_ptr_addr, int comp_id)

int hal_pin_u32_newf(hal_pin_dir_t dir, hal_u32_t ** data_ptr_addr, int comp_id, char *fmt, ...)

int hal_pin_s32_newf(hal_pin_dir_t dir, hal_s32_t ** data_ptr_addr, int comp_id, char *fmt, ...)

int hal_pin_new(char *name, hal_type_t type, hal_in_dir_t dir, void **data_ptr_addr, int comp_id)
```

**ARGUMENTS**

*name*     The name of the pin

*dir*        The direction of the pin, from the viewpoint of the component. It may be one of **HAL\_IN**, **HAL\_OUT**, or **HAL\_IO**. Any number of **HAL\_IN** or **HAL\_IO** pins may be connected to the same signal, but at most one **HAL\_OUT** pin is permitted. A component may assign a value to a pin that is **HAL\_OUT** or **HAL\_IO**, but may not assign a value to a pin that is **HAL\_IN**.

*data\_ptr\_addr*

The address of the pointer-to-data, which must lie within memory allocated by **hal\_malloc**.

*comp\_id*

A HAL component identifier returned by an earlier call to **hal\_init**.

*fmt,*

A printf-style format string and arguments

*type*

The type of the param, as specified in **hal\_type\_t(3hal)**.

**DESCRIPTION**

The **hal\_pin\_new** family of functions create a new *pin* object. Once a pin has been created, it can be linked to a signal object using **hal\_link**. A pin contains a pointer, and the component that owns the pin can dereference the pointer to access whatever signal is linked to the pin. (If no signal is linked, it points to a dummy signal.)

There are functions for each of the data types that the HAL supports. Pins may only be linked to signals of

hal\_pin\_new(3hal)

HAL

hal\_pin\_new(3hal)

the same type.

#### **RETURN VALUE**

Returns a HAL status code.

#### **SEE ALSO**

**hal\_type\_t(3hal), hal\_link(3hal)**

**NAME**

`hal_ready` – indicates that this component is ready

**SYNTAX**

`hal_ready(int comp_id)`

**ARGUMENTS**

*comp\_id*

A HAL component identifier returned by an earlier call to **hal\_init**.

**DESCRIPTION**

**hal\_ready** indicates that this component is ready (has created all its pins, parameters, and functions). This must be called in any realtime HAL component before its **rtapi\_app\_init** exits, and in any userspace component before it enters its main loop.

**RETURN VALUE**

Returns a HAL status code.

**NAME**

hal\_set\_constructor – Set the constructor function for this component

**SYNTAX**

```
typedef int (*hal_constructor_t)(char *prefix, char *arg); int hal_set_constructor(int comp_id, hal_constructor_t constructor)
```

**ARGUMENTS**

*comp\_id* A HAL component identifier returned by an earlier call to **hal\_init**.

*prefix* The prefix to be given to the pins, parameters, and functions in the new instance

*arg* An argument that may be used by the component to customize this instance.

**DESCRIPTION**

As an experimental feature in HAL 2.1, components may be *constructable*. Such a component may create pins and parameters not only at the time the module is loaded, but it may create additional pins and parameters, and functions on demand.

**RETURN VALUE**

Returns a HAL status code.

**SEE ALSO**

**halcmd(1)**



**NAME**

hal\_set\_lock, hal\_get\_lock – Set or get the HAL lock level

**SYNTAX**

```
int hal_set_lock(unsigned char lock_type)
```

```
int hal_get_lock()
```

**ARGUMENTS**

*lock\_type*

The desired lock type, which may be a bitwise combination of: **HAL\_LOCK\_LOAD**, **HAL\_LOCK\_CONFIG**, **HAL\_LOCK\_PARAMS**, or **HAL\_LOCK\_PARAMS**. **HAL\_LOCK\_NONE** or 0 locks nothing, and **HAL\_LOCK\_ALL** locks everything.

**DESCRIPTION****RETURN VALUE**

**hal\_set\_lock** Returns a HAL status code. **hal\_get\_lock** returns the current HAL lock level or a HAL status code.

**NAME**

hal\_signal\_new, hal\_signal\_delete, hal\_link, hal\_unlink – Manipulate HAL signals

**SYNTAX**

```
int hal_signal_new(char *signal_name, hal_type_t type)
```

```
int hal_signal_delete(char *signal_name)
```

```
int hal_link(char *pin_name, char *signal_name)
```

```
int hal_unlink(char *pin_name)
```

**ARGUMENTS**

*signal\_name*

The name of the signal

*pin\_name*

The name of the pin

*type*      The type of the signal, as specified in **hal\_type\_t(3hal)**.

**DESCRIPTION**

**hal\_signal\_new** creates a new signal object. Once a signal has been created, pins can be linked to it with **hal\_link**. The signal object contains the actual storage for the signal data. Pin objects linked to the signal have pointers that point to the data. 'name' is the name of the new signal. If longer than HAL\_NAME\_LEN it will be truncated. If there is already a signal with the same name the call will fail.

**hal\_link** links a pin to a signal. If the pin is already linked to the desired signal, the command succeeds. If the pin is already linked to some other signal, it is an error. In either case, the existing connection is not modified. (Use 'hal\_unlink' to break an existing connection.) If the signal already has other pins linked to it, they are unaffected - one signal can be linked to many pins, but a pin can be linked to only one signal.

**hal\_unlink** unlinks any signal from the specified pin.

**hal\_signal\_delete** deletes a signal object. Any pins linked to the object are unlinked.

**RETURN VALUE**

Returns a HAL status code.

**SEE ALSO**

**hal\_type\_t(3hal)**

**NAME**

hal\_start\_threads – Allow HAL threads to begin executing

**SYNTAX**

```
int hal_start_threads()
```

```
int hal_stop_threads()
```

**ARGUMENTS****DESCRIPTION**

**hal\_start\_threads** starts all threads that have been created. This is the point at which realtime functions start being called.

**hal\_stop\_threads** stops all threads that were previously started by **hal\_start\_threads**. It should be called before any component that is part of a system exits.

**RETURN VALUE**

Returns a HAL status code.

**SEE ALSO**

**hal\_export\_funct(3hal), hal\_create\_thread(3hal), hal\_add\_funct\_to\_thread(3hal)**

**NAME**

hal\_type\_t – typedefs for HAL datatypes

**DESCRIPTION**

typedef ... **hal\_bit\_t**;

A type which may have a value of 0 or nonzero.

typedef ... **hal\_s32\_t**;

A type which may have a value from -2147483648 to 2147483647.

typedef ... **hal\_u32\_t**;

A type which may have a value from 0 to 4294967295.

typedef ... **hal\_float\_t**;

A floating-point type, which typically has the same precision and range as the C type **float**.

typedef enum **hal\_type\_t**;

**HAL\_BIT**

Corresponds to the type **hal\_bit\_t**.

**HAL\_FLOAT**

Corresponds to the type **hal\_float\_t**.

**HAL\_S32**

Corresponds to the type **hal\_s32\_t**.

**HAL\_U32**

Corresponds to the type **hal\_u32\_t**.

**NOTES**

**hal\_bit\_t** is typically a typedef to an integer type whose range is larger than just 0 and 1. When testing the value of a **hal\_bit\_t**, never compare it to 1. Prefer one of the following:

- if(b)
- if(b != 0)

**SEE ALSO**

**hal\_pin\_new(3hal)**, **hal\_param\_new(3hal)**

**NAME**

undocumented – undocumented functions in HAL

**SEE ALSO**

The header file *hal.h*. Most hal functions have documentation in that file.

**NAME**

rtapi – Introduction to the RTAPI API

**DESCRIPTION**

RTAPI is a library providing a uniform API for several real time operating systems. As of ver 2.1, RTLinux, RTAI, and a pure userspace simulator are supported.

The file **rtapi.h** defines the RTAPI for both realtime and non-realtime code. This is a change from Rev 2, where the non-realtime (user space) API was defined in `ulapi.h` and used different function names. The symbols RTAPI and ULAPI are used to determine which mode is being compiled, RTAPI for realtime and ULAPI for non-realtime.

**RTAPI STATUS CODES**

RTAPI\_SUCCESS

call successfull

RTAPI\_UNSUP

function not supported

RTAPI\_BADID

bad task, shmem, sem, or fifo ID

RTAPI\_INVALID

invalid argument

RTAPI\_NOMEM

not enough memory

RTAPI\_LIMIT

resource limit reached

RTAPI\_PERM

permission denied

RTAPI\_BUSY

resource is busy or locked

RTAPI\_NOTFND

object not found

RTAPI\_FAIL

operation failed

**NAME**

rtapi\_clock\_set\_period – set the basic time interval for realtime tasks

**SYNTAX**

rtapi\_clock\_set\_period(long int *nsec*)

**ARGUMENTS**

*nsec*     The desired basic time interval for realtime tasks.

**DESCRIPTION**

**rtapi\_clock\_set\_period** sets the basic time interval for realtime tasks. All periodic tasks will run at an integer multiple of this period. The first call to **rtapi\_clock\_set\_period** with *nsec* greater than zero will start the clock, using *nsec* as the clock period in nano-seconds. Due to hardware and RTOS limitations, the actual period may not be exactly what was requested. On success, the function will return the actual clock period if it is available, otherwise it returns the requested period. If the requested period is outside the limits imposed by the hardware or RTOS, it returns **RTAPI\_INVALID** and does not start the clock. Once the clock is started, subsequent calls with non-zero *nsec* return **RTAPI\_INVALID** and have no effect. Calling **rtapi\_clock\_set\_period** with *nsec* set to zero queries the clock, returning the current clock period, or zero if the clock has not yet been started.

**REALTIME CONSIDERATIONS**

Call only from within init/cleanup code, not from realtime tasks. This function is not available from user (non-realtime) code.

**RETURN VALUE**

The actual period provided by the RTOS, which may be different than the requested period, or a RTAPI status code.

**NAME**

rtapi\_delay – one-line description of rtapi\_delay

**SYNTAX**

void rtapi\_delay(long int *nsec*)

void rtapi\_delay\_max()

**ARGUMENTS**

*nsec*     The desired delay length in nanoseconds

**DESCRIPTION**

**rtapi\_delay** is a simple delay. It is intended only for short delays, since it simply loops, wasting CPU cycles.

**rtapi\_delay\_max** returns the max delay permitted (usually approximately 1/4 of the clock period). Any call to **rtapi\_delay** requesting a delay longer than the max will delay for the max time only.

**rtapi\_delay\_max** should be called before using **rtapi\_delay** to make sure the required delays can be achieved. The actual resolution of the delay may be as good as one nano-second, or as bad as a several microseconds.

**REALTIME CONSIDERATIONS**

May be called from init/cleanup code, and from within realtime tasks.

**RETURN VALUE**

**rtapi\_delay\_max** returns the maximum delay permitted.

**SEE ALSO**

**rtapi\_clock\_set\_period(3rtapi)**



**NAME**

rtapi\_exit – Shut down RTAPI

**SYNTAX**

int rtapi\_exit(int *module\_id*)

**ARGUMENTS**

*module\_id*

An rtapi module identifier returned by an earlier call to **rtapi\_init**.

**DESCRIPTION**

**rtapi\_exit** shuts down and cleans up the RTAPI. It must be called prior to exit by any module that called **rtapi\_init**.

**REALTIME CONSIDERATIONS**

Call only from within user or init/cleanup code, not from realtime tasks.

**RETURN VALUE**

Returns a RTAPI status code.

**NAME**

rtapi\_get\_time – get the current time

**SYNTAX**

long long rtapi\_get\_time()

long long rtapi\_get\_clocks()

**DESCRIPTION**

**rtapi\_get\_time** returns the current time in nanoseconds. Depending on the RTOS, this may be time since boot, or time since the clock period was set, or some other time. Its absolute value means nothing, but it is monotonically increasing and can be used to schedule future events, or to time the duration of some activity. Returns a 64 bit value. The resolution of the returned value may be as good as one nano-second, or as poor as several microseconds. May be called from init/cleanup code, and from within realtime tasks.

Experience has shown that the implementation of this function in some RTOS/Kernel combinations is horrible. It can take up to several microseconds, which is at least 100 times longer than it should, and perhaps a thousand times longer. Use it only if you **MUST** have results in seconds instead of clocks, and use it sparingly. In most cases, **rtapi\_get\_clocks** should be used instead.

**rtapi\_get\_clocks** returns the current time in CPU clocks. It is fast, since it just reads the TSC in the CPU instead of calling a kernel or RTOS function. Of course, times measured in CPU clocks are not as convenient, but for relative measurements this works fine. Its absolute value means nothing, but it is monotonically increasing and can be used to schedule future events, or to time the duration of some activity. (on SMP machines, the two TSC's may get out of sync, so if a task reads the TSC, gets swapped to the other CPU, and reads again, the value may decrease. RTAPI tries to force all RT tasks to run on one CPU.) Returns a 64 bit value. The resolution of the returned value is one CPU clock, which is usually a few nanoseconds to a fraction of a nanosecond.

Note that *long long* math may be poorly supported on some platforms, especially in kernel space. Also note that `rtapi_print()` will NOT print *long longs*. Most time measurements are relative, and should be done like this:

```
deltat = (long int)(end_time - start_time);
```

where `end_time` and `start_time` are `longlong` values returned from `rtapi_get_time`, and `deltat` is an ordinary `long int` (32 bits). This will work for times up to a second or so, depending on the CPU clock frequency. It is best used for millisecond and microsecond scale measurements though.

**RETURN VALUE**

Returns the current time in nanoseconds or CPU clocks.

**NOTES**

Certain versions of the Linux kernel provide a global variable **cpu\_khz**. Computing

```
deltat = (end_clocks - start_clocks) / cpu_khz;
```

gives the duration measured in milliseconds. Computing

```
deltat = (end_clocks - start_clocks) * 1000000 / cpu_khz;
```

gives the duration measured in nanoseconds for deltas less than about 9 trillion clocks (e.g., 3000 seconds at 3GHz).

**NAME**

rtapi\_init – Sets up RTAPI

**SYNTAX**

```
int rtapi_init(char *modname)
```

**ARGUMENTS**

*modname*

The name of this rtapi module

**DESCRIPTION**

**rtapi\_init** sets up the RTAPI. It must be called by any module that intends to use the API, before any other RTAPI calls.

*modname* can optionally point to a string that identifies the module. The string will be truncated at **RTAPI\_NAME\_LEN** characters. If *modname* is **NULL**, the system will assign a name.

**REALTIME CONSIDERATIONS**

Call only from within user or init/cleanup code, not from realtime tasks.

**RETURN VALUE**

On success, returns a positive integer module ID, which is used for subsequent calls to `rtapi_xxx_new`, `rtapi_xxx_delete`, and `rtapi_exit`. On failure, returns an RTAPI error code.

**NAME**

rtapi\_module\_param – Specifying module parameters

**SYNTAX**

RTAPI\_MP\_INT(*var*, *description*)

RTAPI\_MP\_LONG(*var*, *description*)

RTAPI\_MP\_STRING(*var*, *description*)

RTAPI\_MP\_ARRAY\_INT(*var*, *num*, *description*)

RTAPI\_MP\_ARRAY\_LONG(*var*, *num*, *description*)

RTAPI\_MP\_ARRAY\_STRING(*var*, *num*, *description*)

MODULE\_LICENSE(*license*)

MODULE\_AUTHOR(*author*)

MODULE\_DESCRIPTION(*description*)

EXPORT\_FUNCTION(*function*)

**ARGUMENTS**

*var*      The variable where the parameter should be stored

*description*  
A short description of the parameter or module

*num*      The maximum number of values for an array parameter

*license*   The license of the module, for instance "GPL"

*author*   The author of the module

*function*  
The pointer to the function to be exported

**DESCRIPTION**

These macros are portable ways to declare kernel module parameters. They must be used in the global scope, and are not followed by a terminating semicolon. They must be used after the associated variable or function has been defined.

**NAME**

rtapi\_mutex – Mutex-related functions

**SYNTAX**

int rtapi\_mutex\_try(unsigned long *\*mutex*)

void rtapi\_mutex\_get(unsigned long *\*mutex*)

void rtapi\_mutex\_give(unsigned long *\*mutex*)

**ARGUMENTS**

*mutex*    A pointer to the mutex.

**DESCRIPTION**

**rtapi\_mutex\_try** makes a non-blocking attempt to get the mutex. If the mutex is available, it returns 0, and the mutex is no longer available. Otherwise, it returns a nonzero value.

**rtapi\_mutex\_get** blocks until the mutex is available.

**rtapi\_mutex\_give** releases a mutex acquired by **rtapi\_mutex\_try** or **rtapi\_mutex\_get**.

**REALTIME CONSIDERATIONS**

**rtapi\_mutex\_give** and **rtapi\_mutex\_try** may be used from user, init/cleanup, and realtime code.

**rtapi\_mutex\_get** may not be used from realtime code.

**RETURN VALUE**

**rtapi\_mutex\_try** returns 0 for if the mutex was claimed, and nonzero otherwise.

**rtapi\_mutex\_get** and **rtapi\_mutex\_gif** have no return value.

**NAME**

rtapi\_outb, rtapi\_inb – Perform hardware I/O

**SYNTAX**

void rtapi\_outb(unsigned char *byte*, unsigned int *port*)

**ARGUMENTS**

*port*      The address of the I/O port

*byte*      The byte to be written to the port

**DESCRIPTION**

**rtapi\_outb** writes a byte to a hardware I/O port. **rtapi\_inb** reads a byte from a hardware I/O port.

**REALTIME CONSIDERATIONS**

May be called from init/cleanup code and from within realtime tasks.

**RETURN VALUE**

**rtapi\_inb** returns the byte read from the given I/O port

**NOTES**

The I/O address should be within a region previously allocated by **rtapi\_request\_region**. Otherwise, another real-time module or the Linux kernel might attempt to access the I/O region at the same time.

**SEE ALSO**

**rtapi\_region(3rtapi)**

**NAME**

rtapi\_print, rtapi\_print\_msg – print diagnostic messages

**SYNTAX**

```
void rtapi_print(const char *fmt, ...)
```

```
void rtapi_print_msg(int level, const char *fmt, ...)
```

```
typedef void(*rtapi_msg_handler_t)(msg_level_t level, char *msg);
```

```
void rtapi_set_msg_handler(rtapi_msg_handler_t handler);
```

```
rtapi_msg_handler_t rtapi_set_msg_handler(void);
```

**ARGUMENTS**

*level* A message level: One of **RTAPI\_MSG\_ERR**, **RTAPI\_MSG\_WARN**, **RTAPI\_MSG\_INFO**, or **RTAPI\_MSG\_DBG**.

*handler* A function to call from **rtapi\_print** or **rtapi\_print\_msg** to actually output the message.

*fmt*  
... Other arguments are as for *printf(3)*.

**DESCRIPTION**

**rtapi\_print** and **rtapi\_print\_msg** work like the standard C *printf* functions, except that a reduced set of formatting operations are supported.

Depending on the RTOS, the default may be to print the message to stdout, stderr, a kernel log, etc. In RTAPI code, the action may be changed by a call to **rtapi\_set\_msg\_handler**. A **NULL** argument to **rtapi\_set\_msg\_handler** restores the default handler. **rtapi\_msg\_get\_handler** returns the current handler. When the message came from **rtapi\_print**, *level* is **RTAPI\_MSG\_ALL**.

**rtapi\_print\_msg** works like **rtapi\_print** but only prints if *level* is less than or equal to the current message level.

**REALTIME CONSIDERATIONS**

**rtapi\_print** and **rtapi\_print\_msg** May be called from user, init/cleanup, and realtime code. **rtapi\_get\_msg\_handler** and **rtapi\_set\_msg\_handler** may be called from realtime init/cleanup code. A message handler passed to **rtapi\_set\_msg\_handler** may only call functions that can be called from real-time code.

**RETURN VALUE**

None.

**SEE ALSO**

**rtapi\_set\_msg\_level(3rtapi)**, **rtapi\_get\_msg\_level(3rtapi)**, **printf(3)**

**NAME**

rtapi\_prio – thread priority functions

**SYNTAX**

```
int rtapi_prio_highest()
int rtapi_prio_lowest()
int rtapi_prio_next_higher(int prio)
int rtapi_prio_next_lower(int prio)
```

**ARGUMENTS**

*prio*     A value returned by a prior **rtapi\_prio\_xxx** call

**DESCRIPTION**

The **rtapi\_prio\_xxxx** functions provide a portable way to set task priority. The mapping of actual priority to priority number depends on the RTOS. Priorities range from **rtapi\_prio\_lowest** to **rtapi\_prio\_highest**, inclusive. To use this API, use one of two methods:

- 1) Set your lowest priority task to **rtapi\_prio\_lowest**, and for each task of the next lowest priority, set their priorities to **rtapi\_prio\_next\_higher(previous)**.
- 2) Set your highest priority task to **rtapi\_prio\_highest**, and for each task of the next highest priority, set their priorities to **rtapi\_prio\_next\_lower(previous)**.

N.B. A high priority task will pre-empt or interrupt a lower priority task. Linux is always the lowest priority!

**REALTIME CONSIDERATIONS**

Call these functions only from within init/cleanup code, not from realtime tasks.

**RETURN VALUE**

Returns an opaque real-time priority number.

**SEE ALSO**

**rtapi\_task\_new(3rtapi)**



**NAME**

rtapi\_region – functions to manage I/O memory regions

**SYNTAX**

```
void *rtapi_request_region(unsigned long base, unsigned long int size, const char *name)
```

```
void rtapi_release_region(unsigned long base, unsigned long int size)
```

**ARGUMENTS**

*base*     The base address of the I/O region

*size*     The size of the I/O region

*name*     The name to be shown in /proc/ioprocs

**DESCRIPTION**

**rtapi\_request\_region** reserves I/O memory starting at *base* and going for *size* bytes.

**REALTIME CONSIDERATIONS**

May be called from realtime init/cleanup code only.

**BUGS**

On kernels before 2.4.0, **rtapi\_request\_region** always succeeds.

**RETURN VALUE**

**rtapi\_request\_region** returns NULL if the allocation fails, and a non-NULL value otherwise.

**rtapi\_release\_region** has no return value.

**NAME**

rtapi\_get\_msg\_level, rtapi\_set\_msg\_level – Get or set the logging level

**SYNTAX**

int rtapi\_set\_msg\_level(int *level*)

int rtapi\_get\_msg\_level()

**ARGUMENTS**

*level*     The desired logging level

**DESCRIPTION**

Get or set the RTAPI message level used by **rtapi\_print\_msg**. Depending on the RTOS, this level may apply to a single RTAPI module, or it may apply to a group of modules.

**REALTIME CONSIDERATIONS**

May be called from user, init/cleanup, and realtime code.

**RETURN VALUE**

**rtapi\_set\_msg\_level** returns a status code, and **rtapi\_get\_msg\_level** returns the current level.

**SEE ALSO**

**rtapi\_print\_msg(3rtapi)**

## NAME

rtapi\_shmem – Functions for managing shared memory blocks

## SYNTAX

```
int rtapi_shmem_new(int key, int module_id, unsigned long int size)
```

```
int rtapi_shmem_delete(int shmem_id, int module_id)
```

```
int rtapi_shmem_getptr(int shmem_id, void ** ptr)
```

## ARGUMENTS

*key*        Identifies the memory block. Key must be nonzero. All modules wishing to use the same memory must use the same key.

*module\_id*  
            Module identifier returned by a prior call to **rtapi\_init**.

*size*        The desired size of the shared memory block, in bytes

*ptr*        The pointer to the shared memory block. Note that the block may be mapped at a different address for different modules.

## DESCRIPTION

**rtapi\_shmem\_new** allocates a block of shared memory. *key* identifies the memory block, and must be non-zero. All modules wishing to access the same memory must use the same key. *module\_id* is the ID of the module that is making the call (see **rtapi\_init**). The block will be at least *size* bytes, and may be rounded up. Allocating many small blocks may be very wasteful. When a particular block is allocated for the first time, the first 4 bytes are zeroed. Subsequent allocations of the same block by other modules or processes will not touch the contents of the block. Applications can use those bytes to see if they need to initialize the block, or if another module already did so. On success, it returns a positive integer ID, which is used for all subsequent calls dealing with the block. On failure it returns a negative error code.

**rtapi\_shmem\_delete** frees the shared memory block associated with *shmem\_id*. *module\_id* is the ID of the calling module. Returns a status code.

**rtapi\_shmem\_getptr** sets *\*ptr* to point to shared memory block associated with *shmem\_id*.

## REALTIME CONSIDERATIONS

**rtapi\_shmem\_getptr** may be called from user code, init/cleanup code, or realtime tasks.

**rtapi\_shmem\_new** and **rtapi\_shmem\_dete** may not be called from realtime tasks.

## RETURN VALUE

**NAME**

rtapi\_snprintf, rtapi\_vsnprintf – Perform snprintf-like string formatting

**SYNTAX**

int rtapi\_snprintf(char *\*buf*, unsigned long int *size*, const char *\*fmt*, ...)

int rtapi\_vsnprintf(char *\*buf*, unsigned long int *size*, const char *\*fmt*, va\_list *apfB*)

**ARGUMENTS**

As for *snprintf(3)* or *vsnprintf(3)*.

**DESCRIPTION**

These functions work like the standard C printf functions, except that a reduced set of formatting operations are supported.

**REALTIME CONSIDERATIONS**

May be called from user, init/cleanup, and realtime code.

**RETURN VALUE**

The number of characters written to *buf*.

**SEE ALSO**

**printf(3)**

**NAME**

rtapi\_task\_new – create a realtime task

**SYNTAX**

```
int rtapi_task_new(void (*taskcode)(void*), void *arg,          int prio, unsigned long stacksize, int
                   uses_fp)
int rtapi_task_delete(int task_id)
```

**ARGUMENTS**

*taskcode*

A pointer to the function to be called when the task is started

*arg*

An argument to be passed to the *taskcode* function when the task is started

*prio*

A task priority value returned by **rtapi\_prio\_xxxx**

*uses\_fp*

A flag that tells the OS whether the task uses floating point or not.

*task\_id*

A task ID returned by a previous call to **rtapi\_task\_new**

**DESCRIPTION**

**rtapi\_task\_new** creates but does not start a realtime task. The task is created in the "paused" state. To start it, call either **rtapi\_task\_start** for periodic tasks, or **rtapi\_task\_resume** for free-running tasks.

**REALTIME CONSIDERATIONS**

Call only from within init/cleanup code, not from realtime tasks.

**RETURN VALUE**

On success, returns a positive integer task ID. This ID is used for all subsequent calls that need to act on the task. On failure, returns an RTAPI status code.

**SEE ALSO**

**rtapi\_prio(3rtapi)**, **rtapi\_task\_start(3rtapi)**, **rtapi\_task\_wait(3rtapi)**, **rtapi\_task\_resume(3rtapi)**

**NAME**

rtapi\_task\_pause, rtapi\_task\_resume – pause and resume real-time tasks

**SYNTAX**

void rtapi\_task\_pause(int *task\_id*)

void rtapi\_task\_resume(int *task\_id*)

**ARGUMENTS**

*task\_id* An RTAPI task identifier returned by an earlier call to **rtapi\_task\_new**.

**DESCRIPTION**

**rtapi\_task\_resume** starts a task in free-running mode. The task must be in the "paused" state.

A free running task runs continuously until either:

- 1) It is preempted by a higher priority task. It will resume as soon as the higher priority task releases the CPU.
- 2) It calls a blocking function, like **rtapi\_sem\_take**. It will resume when the function unblocks.
- 3) It is returned to the "paused" state by **rtapi\_task\_pause**. May be called from init/cleanup code, and from within realtime tasks.

**rtapi\_task\_pause** causes a task to stop execution and change to the "paused" state. The task can be free-running or periodic. Note that **rtapi\_task\_pause** may be called from any task, or from init or cleanup code, not just from the task that is to be paused. The task will resume execution when either **rtapi\_task\_resume** or **rtapi\_task\_start** (depending on whether this is a free-running or periodic task) is called.

**REALTIME CONSIDERATIONS**

May be called from init/cleanup code, and from within realtime tasks.

**RETURN VALUE**

An RTAPI status code.

**SEE ALSO**

**rtapi\_task\_new(3rtapi)**, **rtapi\_task\_start(3rtapi)**

**NAME**

rtapi\_task\_start – start a realtime task in periodic mode

**SYNTAX**

int rtapi\_task\_start(int *task\_id*, unsigned long *period\_nsec*)

**ARGUMENTS**

*task\_id* A task ID returned by a previous call to **rtapi\_task\_new**

*period\_nsec*

The clock period in nanoseconds between iterations of a periodic task

**DESCRIPTION**

**rtapi\_task\_start** starts a task in periodic mode. The task must be in the *paused* state.

**REALTIME CONSIDERATIONS**

Call only from within init/cleanup code, not from realtime tasks.

**RETURN VALUE**

Returns an RTAPI status code.

**SEE ALSO**

**rtapi\_task\_new(3rtapi)**, **rtapi\_task\_pause(3rtapi)**, **rtapi\_task\_resume(3rtapi)**

**NAME**

rtapi\_task\_wait – suspend execution of this periodic task

**SYNTAX**

void rtapi\_task\_wait()

**DESCRIPTION**

**rtapi\_task\_wait** suspends execution of the current task until the next period. The task must be periodic. If not, the result is undefined.

**REALTIME CONSIDERATIONS**

Call only from within a periodic realtime task

**RETURN VALUE**

None

**SEE ALSO**

**rtapi\_task\_start(3rtapi)**, **rtapi\_task\_pause(3rtapi)**



**NAME**

undocumented – undocumented functions in RTAPI

**SEE ALSO**

The header file *rtapi.h*. Most rtapi functions have documentation in that file.

**NAME**

abs – Compute the absolute value and sign of the input signal

**SYNOPSIS**

**loadrt abs [count=N]**

**FUNCTIONS**

**abs.N** (uses floating-point)

**PINS**

**abs.N.in** float in

Analog input value

**abs.N.out** float out

Analog output value, always positive

**abs.N.sign** bit out

Sign of input, false for positive, true for negative

**LICENSE**

GPL

**NAME**

and2 – Two-input AND gate

**SYNOPSIS**

**loadrt and2 [count=*N*]**

**FUNCTIONS**

**and2.*N***

**PINS**

**and2.*N*.in0** bit in

**and2.*N*.in1** bit in

**and2.*N*.out** bit out

**out** is computed from the value of **in0** and **in1** according to the following rule:

**in0=TRUE in1=TRUE**

**out=TRUE**

Otherwise,

**out=FALSE**

**LICENSE**

GPL

**NAME**

`at_pid` – proportional/integral/derivative controller with auto tuning

**SYNOPSIS**

`loadrt at_pid num_chan=num [debug=dbg]`

**DESCRIPTION**

**at\_pid** is a classic Proportional/Integral/Derivative controller, used to control position or speed feedback loops for servo motors and other closed-loop applications.

**at\_pid** supports a maximum of sixteen controllers. The number that are actually loaded is set by the **num\_chan** argument when the module is loaded. If **numchan** is not specified, the default value is three. If **debug** is set to 1 (the default is 0), some additional HAL parameters will be exported, which might be useful for tuning, but are otherwise unnecessary.

**at\_pid** has a built in auto tune mode. It works by setting up a limit cycle to characterize the process. From this, **Pgain/Igain/Dgain** or **Pgain/Igain/FF1** can be determined using Ziegler-Nichols. When using **FF1**, scaling must be set so that **output** is in user units per second.

During auto tuning, the **command** input should not change. The limit cycle is setup around the commanded position. No initial tuning values are required to start auto tuning. Only **tune-cycles**, **tune-effort** and **tune-mode** need be set before starting auto tuning. When auto tuning completes, the tuning parameters will be set. If running from EMC, the FERROR setting for the axis being tuned may need to be loosened up as it must be larger than the limit cycle amplitude in order to avoid a following error.

To perform auto tuning, take the following steps. Move the axis to be tuned, to somewhere near the center of it's travel. Set **tune-cycles** (the default value should be fine in most cases) and **tune-mode**. Set **tune-effort** to a small value. Set **enable** to true. Set **tune-mode** to true. Set **tune-start** to true. If no oscillation occurs, or the oscillation is too small, slowly increase **tune-effort**. Auto tuning can be aborted at any time by setting **enable** or **tune-mode** to false.

**FUNCTIONS**

**pid.N.do-pid-calcs** (uses floating-point)

Does the PID calculations for control loop *N*.

**PINS**

**pid.N.command** float in

The desired (commanded) value for the control loop.

**pid.N.feedback** float in

The actual (feedback) value, from some sensor such as an encoder.

**pid.N.error** float out

The difference between command and feedback.

**pid.N.output** float out

The output of the PID loop, which goes to some actuator such as a motor.

**pid.N.enable** bit in

When true, enables the PID calculations. When false, **output** is zero, and all internal integrators, etc, are reset.

**pid.N.tune-mode** bit in

When true, enables auto tune mode. When false, normal PID calculations are performed.

**pid.N.tune-start** bit io

When set to true, starts auto tuning. Cleared when the auto tuning completes.

## PARAMETERS

**pid.N.Pgain** float rw

Proportional gain. Results in a contribution to the output that is the error multiplied by **Pgain**.

**pid.N.Igain** float rw

Integral gain. Results in a contribution to the output that is the integral of the error multiplied by **Igain**. For example an error of 0.02 that lasted 10 seconds would result in an integrated error (**errorI**) of 0.2, and if **Igain** is 20, the integral term would add 4.0 to the output.

**pid.N.Dgain** float rw

Derivative gain. Results in a contribution to the output that is the rate of change (derivative) of the error multiplied by **Dgain**. For example an error that changed from 0.02 to 0.03 over 0.2 seconds would result in an error derivative (**errorD**) of 0.05, and if **Dgain** is 5, the derivative term would add 0.25 to the output.

**pid.N.bias** float rw

**bias** is a constant amount that is added to the output. In most cases it should be left at zero. However, it can sometimes be useful to compensate for offsets in servo amplifiers, or to balance the weight of an object that moves vertically. **bias** is turned off when the PID loop is disabled, just like all other components of the output. If a non-zero output is needed even when the PID loop is disabled, it should be added with an external HAL sum2 block.

**pid.N.FF0** float rw

Zero order feed-forward term. Produces a contribution to the output that is **FF0** multiplied by the commanded value. For position loops, it should usually be left at zero. For velocity loops, **FF0** can compensate for friction or motor counter-EMF and may permit better tuning if used properly.

**pid.N.FF1** float rw

First order feed-forward term. Produces a contribution to the output that **FF1** multiplied by the derivative of the commanded value. For position loops, the contribution is proportional to speed, and can be used to compensate for friction or motor CEMF. For velocity loops, it is proportional to acceleration and can compensate for inertia. In both cases, it can result in better tuning if used properly.

**pid.N.FF2** float rw

Second order feed-forward term. Produces a contribution to the output that is **FF2** multiplied by the second derivative of the commanded value. For position loops, the contribution is proportional to acceleration, and can be used to compensate for inertia. For velocity loops, it should usually be left at zero.

**pid.N.deadband** float rw

Defines a range of "acceptable" error. If the absolute value of **error** is less than **deadband**, it will be treated as if the error is zero. When using feedback devices such as encoders that are inherently quantized, the deadband should be set slightly more than one-half count, to prevent the control loop from hunting back and forth if the command is between two adjacent encoder values. When the absolute value of the error is greater than the deadband, the deadband value is subtracted from the error before performing the loop calculations, to prevent a step in the transfer function at the edge of the deadband. (See **BUGS**.)

**pid.N.maxoutput** float rw

Output limit. The absolute value of the output will not be permitted to exceed **maxoutput**, unless **maxoutput** is zero. When the output is limited, the error integrator will hold instead of integrating, to prevent windup and overshoot.

**pid.N.maxerror** float rw

Limit on the internal error variable used for P, I, and D. Can be used to prevent high **Pgain** values from generating large outputs under conditions when the error is large (for example, when the command makes a step change). Not normally needed, but can be useful when tuning non-linear systems.

**pid.N.maxerrorD** float rw

Limit on the error derivative. The rate of change of error used by the **Dgain** term will be limited to this value, unless the value is zero. Can be used to limit the effect of **Dgain** and prevent large output spikes due to steps on the command and/or feedback. Not normally needed.

**pid.N.maxerrorI** float rw

Limit on error integrator. The error integrator used by the **Igain** term will be limited to this value, unless it is zero. Can be used to prevent integrator windup and the resulting overshoot during/after sustained errors. Not normally needed.

**pid.N.maxcmdD** float rw

Limit on command derivative. The command derivative used by **FF1** will be limited to this value, unless the value is zero. Can be used to prevent **FF1** from producing large output spikes if there is a step change on the command. Not normally needed.

**pid.N.maxcmdDD** float rw

Limit on command second derivative. The command second derivative used by **FF2** will be limited to this value, unless the value is zero. Can be used to prevent **FF2** from producing large output spikes if there is a step change on the command. Not normally needed.

**pid.N.tune-type** u32 rw

When set to 0, **Pgain/Igain/Dgain** are calculated. When set to 1, **Pgain/Igain/FF1** are calculated.

**pid.N.tune-cycles** u32 rw

Determines the number of cycles to run to characterize the process. **tune-cycles** actually sets the number of half cycles. More cycles results in a more accurate characterization as the average of all cycles is used.

**pid.N.tune-effort** float rw

Determines the effort used in setting up the limit cycle in the process. **tune-effort** should be set to a positive value less than **maxoutput**. Start with something small and work up to a value that results in a good portion of the maximum motor current being used. The smaller the value, the smaller the amplitude of the limit cycle.

**pid.N.errorI** float ro (only if debug=1)

Integral of error. This is the value that is multiplied by **Igain** to produce the Integral term of the output.

**pid.N.errorD** float ro (only if debug=1)

Derivative of error. This is the value that is multiplied by **Dgain** to produce the Derivative term of the output.

**pid.N.commandD** float ro (only if debug=1)

Derivative of command. This is the value that is multiplied by **FF1** to produce the first order feed-forward term of the output.

**pid.N.commandDD** float ro (only if debug=1)

Second derivative of command. This is the value that is multiplied by **FF2** to produce the second order feed-forward term of the output.

**pid.N.ultimate-gain** float ro (only if debug=1)

Determined from process characterization. **ultimate-gain** is the ratio of **tune-effort** to the limit cycle amplitude multiplied by 4.0 divided by Pi. **pid.N.ultimate-period** float ro (only if debug=1)

Determined from process characterization. **ultimate-period** is the period of the limit cycle.

## BUGS

Some people would argue that deadband should be implemented such that error is treated as zero if it is within the deadband, and be unmodified if it is outside the deadband. This was not done because it would cause a step in the transfer function equal to the size of the deadband. People who prefer that behavior are welcome to add a parameter that will change the behavior, or to write their own version of **at\_pid**. However, the default behavior should not be changed.

**NAME**

biquad – Biquad IIR filter

**SYNOPSIS**

**loadrt biquad [count=N]**

**DESCRIPTION**

Biquad IIR filter. Implements the following transfer function:  $H(z) = (n_0 + n_1z^{-1} + n_2z^{-2}) / (1 + d_1z^{-1} + d_2z^{-2})$

**FUNCTIONS**

**biquad.N** (uses floating-point)

**PINS**

**biquad.N.in** float in  
Filter input.

**biquad.N.out** float out  
Filter output.

**biquad.N.enable** bit in (default: 0)  
Filter enable. When false, the in is passed to out without any filtering. A transition from false to true causes filter coefficients to be calculated according to parameters

**biquad.N.valid** bit out (default: 0)  
When false, indicates an error occurred when calculating filter coefficients.

**PARAMETERS**

**biquad.N.type** u32 rw (default: 0)  
Filter type determines the type of filter coefficients calculated. When 0, coefficients must be loaded directly. When 1, a low pass filter is created. When 2, a notch filter is created.

**biquad.N.f0** float rw (default: 250.0)  
The corner frequency of the filter.

**biquad.N.Q** float rw (default: 0.7071)  
The Q of the filter.

**biquad.N.d1** float rw (default: 0.0)  
1st-delayed denominator coef

**biquad.N.d2** float rw (default: 0.0)  
2nd-delayed denominator coef

**biquad.N.n0** float rw (default: 1.0)  
non-delayed numerator coef

**biquad.N.n1** float rw (default: 0.0)  
1st-delayed numerator coef

**biquad.N.n2** float rw (default: 0.0)  
2nd-delayed numerator coef

**biquad.N.s1** float rw (default: 0.0)

**biquad.N.s2** float rw (default: 0.0)

**LICENSE**

GPL

**NAME**

blend – Perform linear interpolation between two values

**SYNOPSIS**

**loadrt blend [count=*N*]**

**FUNCTIONS**

**blend.*N*** (uses floating-point)

**PINS**

**blend.*N*.in1** float in

First input. If select is equal to 0.0, the output is equal to in1

**blend.*N*.in2** float in

Second input. If select is equal to 1.0, the output is equal to in2

**blend.*N*.select** float in

Select input. For values between 0.0 and 1.0, the output changes linearly from in1 to in2

**blend.*N*.out** float out

Output value.

**PARAMETERS**

**blend.*N*.open** bit rw

If true, select values outside the range 0.0 to 1.0 give values outside the range in1 to in2. If false, outputs are clamped to the the range in1 to in2

**LICENSE**

GPL



**NAME**

blocks – Old style HAL blocks (deprecated)

**SYNOPSIS**

**loadrt blocks** [*blockname=N*]

**DESCRIPTION**

Most of the items available in **blocks** are the same as in the individual components, named below. **blocks** is deprecated and should not be used in new HAL configurations. **blocks** may be removed from emc2 as early as version 2.2.0.

**SEE ALSO**

**constant(9)**, **wcomp(9)**, **comp(9)**, **sum2(9)**, **mult2(9)**, **hypot(9)**, **mux2(9)**, **mux4(9)**, **integ(9)**, **ddt(9)**, **limit1(9)**, **limit2(9)**, **limit3(9)**, **estop\_latch(9)** (called "estop" in blocks), **not(9)**, **and2(9)**, **or2(9)**, **scale(9)**, **lowpass(9)**, **match8(9)**, **minmax(9)**

**NAME**

charge\_pump – Create a square-wave for the 'charge pump' input of some controller boards

**SYNOPSIS**

**loadrt charge\_pump**

**FUNCTIONS**

**charge-pump**

Toggle the output bit (if enabled)

**PINS**

**charge-pump.out** bit out

Square wave if 'enable' is TRUE or unconnected, low if 'enable' is FALSE

**charge-pump.enable** bit in (default: *TRUE*)

If FALSE, forces 'out' to be low

**LICENSE**

GPL

**NAME**

clarke2 – Two input version of Clarke transform

**SYNOPSIS**

**loadrt clarke2 [count=*N*]**

**DESCRIPTION**

The Clarke transform can be used to translate a vector quantity from a three phase system (three components 120 degrees apart) to a two phase Cartesian system.

**clarke2** implements a special case of the Clarke transform, which only needs two of the three input phases. In a three wire three phase system, the sum of the three phase currents or voltages must always be zero. As a result only two of the three are needed to completely define the current or voltage. **clarke2** assumes that the sum is zero, so it only uses phases A and B of the input. Since the H (homopolar) output will always be zero in this case, it is not generated.

**FUNCTIONS**

**clarke2.*N*** (uses floating-point)

**PINS**

**clarke2.*N.a*** float in

**clarke2.*N.b*** float in

first two phases of three phase input

**clarke2.*N.x*** float out

**clarke2.*N.y*** float out

cartesian components of output

**SEE ALSO**

**clarke3** for the general case, **clarkeinv** for the inverse transform.

**LICENSE**

GPL

**NAME**

clarke3 – Clarke (3 phase to cartesian) transform

**SYNOPSIS**

**loadrt clarke3 [count=*N*]**

**DESCRIPTION**

The Clarke transform can be used to translate a vector quantity from a three phase system (three components 120 degrees apart) to a two phase Cartesian system (plus a homopolar component if the three phases don't sum to zero).

**clarke3** implements the general case of the transform, using all three phases. If the three phases are known to sum to zero, see **clarke2** for a simpler version.

**FUNCTIONS**

**clarke3.*N*** (uses floating-point)

**PINS**

**clarke3.*N.a*** float in

**clarke3.*N.b*** float in

**clarke3.*N.c*** float in

three phase input vector

**clarke3.*N.x*** float out

**clarke3.*N.y*** float out

cartesian components of output

**clarke3.*N.h*** float out

homopolar component of output

**SEE ALSO**

**clarke2** for the 'a+b+c=0' case, **clarkeinv** for the inverse transform.

**LICENSE**

GPL

**NAME**

clarkeinv – Inverse Clarke transform

**SYNOPSIS**

**loadrt clarkeinv [count=*N*]**

**DESCRIPTION**

The inverse Clarke transform can be used to translate a vector quantity from Cartesian coordinate system to a three phase system (three components 120 degrees apart).

**FUNCTIONS**

**clarkeinv.*N*** (uses floating-point)

**PINS**

**clarkeinv.*N.x*** float in

**clarkeinv.*N.y*** float in

cartesian components of input

**clarkeinv.*N.h*** float in

homopolar component of input (usually zero)

**clarkeinv.*N.a*** float out

**clarkeinv.*N.b*** float out

**clarkeinv.*N.c*** float out

three phase output vector

**SEE ALSO**

**clarke2** and **clarke3** for the forward transform.

**LICENSE**

GPL

**NAME**

classicladder – realtime software plc based on ladder logic

**SYNOPSIS**

**loadrt classicladder\_rt [numRungs=*N*] [numBits=*N*] [numWords=*N*] [numTimers=*N*] [numMonostables=*N*] [numCounters=*N*] [numPhysInputs=*N*] [numPhysOutputs=*N*] [numArithmExpr=*N*] [numSections=*N*] [numSymbols=*N*] [numS32in=*N*] [numS32out=*N*]**

**DESCRIPTION**

These pins and parameters are created by the realtime **classicladder\_rt** module. Each period, classicladder reads the inputs, evaluates the ladder logic defined in the GUI, and then writes the outputs.

**PINS**

**classicladder.0.in-*N*** IN bit

Connect a hal bit signal to this pin to use **B*NN*** in classicladder

**classicladder.0.out-*N*** OUT bit

Output from classicladder

**classicladder.0.in-*N*** IN s32

Connect a hal s32 signal to this pin to use **W*NN*** in classicladder

**classicladder.0.out-*N*** OUT s32

Integer output from classicladder

**PARAMETERS**

**classicladder.0.refresh.time** RO s32

not sure what this does anymore

**classicladder.0.refresh.tmax** RW s32

ditto

**classicladder.0.ladder-state** RO s32

**FUNCTIONS**

**classicladder.0.refresh** FP

The rung update rate. Add this to the servo thread.

**BUGS**

The classicladder\_rt module does not unload correctly.

**SEE ALSO**

until we get some real docs put together:

<http://wiki.linuxcnc.org/cgi-bin/emcinfo.pl?ClassicLadder>

**NAME**

comp – Two input comparator with hysteresis

**SYNOPSIS**

**loadrt comp [count=*N*]**

**FUNCTIONS**

**comp.*N*** (uses floating-point)

Update the comparator

**PINS**

**comp.*N*.in0** float in

Inverting input to the comparator

**comp.*N*.in1** float in

Non-inverting input to the comparator

**comp.*N*.out** bit out

Normal output. True when **in1** > **in0** (see parameter **hyst** for details)

**comp.*N*.equal** bit out

Match output. True when difference between **in1** and **in0** is less than **hyst/2**

**PARAMETERS**

**comp.*N*.hyst** float rw (default: 0.0)

Hysteresis of the comparator (default 0.0)

With zero hysteresis, the output is true when **in1** > **in0**. With nonzero hysteresis, the output switches on and off at two different values, separated by distance **hyst** around the point where **in1** = **in0**. Keep in mind that floating point calculations are never absolute and it is wise to always set **hyst** if you intend to use equal

**LICENSE**

GPL

**NAME**

constant – Use a parameter to set the value of a pin

**SYNOPSIS**

**loadrt constant [count=*N*]**

**FUNCTIONS**

**constant.*N*** (uses floating-point)

**PINS**

**constant.*N*.out** float out

**PARAMETERS**

**constant.*N*.value** float rw

**LICENSE**

GPL



**NAME**

conv\_bit\_s32 – Convert a value from bit to s32

**SYNOPSIS**

**loadrt conv\_bit\_s32 [count=*N*]**

**FUNCTIONS**

**conv-bit-s32.*N***

Update 'out' based on 'in'

**PINS**

**conv-bit-s32.*N.in*** bit in

**conv-bit-s32.*N.out*** s32 out

**LICENSE**

GPL

**NAME**

conv\_bit\_u32 – Convert a value from bit to u32

**SYNOPSIS**

**loadrt conv\_bit\_u32 [count=*N*]**

**FUNCTIONS**

**conv-bit-u32.*N***

Update 'out' based on 'in'

**PINS**

**conv-bit-u32.*N*.in** bit in

**conv-bit-u32.*N*.out** u32 out

**LICENSE**

GPL

**NAME**

conv\_float\_s32 – Convert a value from float to s32

**SYNOPSIS**

**loadrt conv\_float\_s32 [count=N]**

**FUNCTIONS**

**conv-float-s32.N**

Update 'out' based on 'in'

**PINS**

**conv-float-s32.N.in** float in

**conv-float-s32.N.out** s32 out

**conv-float-s32.N.out-of-range** bit out

TRUE when 'in' is not in the range of s32

**PARAMETERS**

**conv-float-s32.N.clamp** bit rw

If TRUE, then clamp to the range of s32. If FALSE, then allow the value to "wrap around".

**LICENSE**

GPL

**NAME**

conv\_float\_u32 – Convert a value from float to u32

**SYNOPSIS**

**loadrt conv\_float\_u32 [count=*N*]**

**FUNCTIONS**

**conv-float-u32.*N***

Update 'out' based on 'in'

**PINS**

**conv-float-u32.*N.in*** float in

**conv-float-u32.*N.out*** u32 out

**conv-float-u32.*N.out-of-range*** bit out

TRUE when 'in' is not in the range of u32

**PARAMETERS**

**conv-float-u32.*N.clamp*** bit rw

If TRUE, then clamp to the range of u32. If FALSE, then allow the value to "wrap around".

**LICENSE**

GPL

**NAME**

conv\_s32\_bit – Convert a value from s32 to bit

**SYNOPSIS**

**loadrt conv\_s32\_bit [count=*N*]**

**FUNCTIONS**

**conv-s32-bit.*N***

Update 'out' based on 'in'

**PINS**

**conv-s32-bit.*N.in*** s32 in

**conv-s32-bit.*N.out*** bit out

**conv-s32-bit.*N.out-of-range*** bit out

TRUE when 'in' is not in the range of bit

**PARAMETERS**

**conv-s32-bit.*N.clamp*** bit rw

If TRUE, then clamp to the range of bit. If FALSE, then allow the value to "wrap around".

**LICENSE**

GPL

**NAME**

conv\_s32\_float – Convert a value from s32 to float

**SYNOPSIS**

**loadrt conv\_s32\_float [count=*N*]**

**FUNCTIONS**

**conv-s32-float.*N***

Update 'out' based on 'in'

**PINS**

**conv-s32-float.*N.in*** s32 in

**conv-s32-float.*N.out*** float out

**LICENSE**

GPL

**NAME**

conv\_s32\_u32 – Convert a value from s32 to u32

**SYNOPSIS**

**loadrt conv\_s32\_u32 [count=*N*]**

**FUNCTIONS**

**conv-s32-u32.*N***

Update 'out' based on 'in'

**PINS**

**conv-s32-u32.*N*.in** s32 in

**conv-s32-u32.*N*.out** u32 out

**conv-s32-u32.*N*.out-of-range** bit out

TRUE when 'in' is not in the range of u32

**PARAMETERS**

**conv-s32-u32.*N*.clamp** bit rw

If TRUE, then clamp to the range of u32. If FALSE, then allow the value to "wrap around".

**LICENSE**

GPL

**NAME**

conv\_u32\_bit – Convert a value from u32 to bit

**SYNOPSIS**

**loadrt conv\_u32\_bit [count=*N*]**

**FUNCTIONS**

**conv-u32-bit.*N***

Update 'out' based on 'in'

**PINS**

**conv-u32-bit.*N*.in** u32 in

**conv-u32-bit.*N*.out** bit out

**conv-u32-bit.*N*.out-of-range** bit out

TRUE when 'in' is not in the range of bit

**PARAMETERS**

**conv-u32-bit.*N*.clamp** bit rw

If TRUE, then clamp to the range of bit. If FALSE, then allow the value to "wrap around".

**LICENSE**

GPL



**NAME**

conv\_u32\_float – Convert a value from u32 to float

**SYNOPSIS**

**loadrt conv\_u32\_float [count=*N*]**

**FUNCTIONS**

**conv-u32-float.*N***

Update 'out' based on 'in'

**PINS**

**conv-u32-float.*N*.in** u32 in

**conv-u32-float.*N*.out** float out

**LICENSE**

GPL

**NAME**

conv\_u32\_s32 – Convert a value from u32 to s32

**SYNOPSIS**

**loadrt conv\_u32\_s32 [count=*N*]**

**FUNCTIONS**

**conv-u32-s32.*N***

Update 'out' based on 'in'

**PINS**

**conv-u32-s32.*N*.in** u32 in

**conv-u32-s32.*N*.out** s32 out

**conv-u32-s32.*N*.out-of-range** bit out

TRUE when 'in' is not in the range of s32

**PARAMETERS**

**conv-u32-s32.*N*.clamp** bit rw

If TRUE, then clamp to the range of s32. If FALSE, then allow the value to "wrap around".

**LICENSE**

GPL

**NAME**

counter – counts input pulses (**DEPRECATED**)

**SYNOPSIS**

**loadrt counter [num\_chan=N]**

**DESCRIPTION**

**counter** is a deprecated HAL component and will be removed in a future release. Use the **encoder** component with encoder.X.counter-mode set to TRUE.

**counter** is a HAL component that provides software- based counting that is useful for spindle position sensing and maybe other things. Instead of using a real encoder that outputs quadrature, some lathes have a sensor that generates a simple pulse stream as the spindle turns and an index pulse once per revolution. This component simply counts up when a "count" pulse (phase-A) is received, and if reset is enabled, resets when the "index" (phase-Z) pulse is received.

This is of course only useful for a unidirectional spindle, as it is not possible to sense the direction of rotation.

**counter** conforms to the "canonical encoder" interface described in the HAL manual.

**FUNCTIONS**

**counter.capture-position** (uses floating-point)

Updates the counts, position and velocity outputs based on internal counters.

**counter.update-counters**

Samples the phase-A and phase-Z inputs and updates internal counters.

**PINS**

**counter.N.phase-A** bit in

The primary input signal. The internal counter is incremented on each rising edge.

**counter.N.phase-Z** bit in

The index input signal. When the **index-enable** pin is TRUE and a rising edge on **phase-Z** is seen, **index-enable** is set to FALSE and the internal counter is reset to zero.

**counter.N.index-enable** bit io

**counter.N.reset** bit io

**counter.N.counts** signed out

**counter.N.position** float out

**counter.N.velocity** float out

These pins function according to the canonical digital encoder interface.

**PARAMETERS**

**counter.N.position-scale** float rw

This parameter functions according to the canonical digital encoder interface.

**counter.N.rawcounts** signed ro

The internal counts value, updated from **update-counters** and reflected in the output pins at the next call to **capture-position**.

**SEE ALSO**

**encoder(9)**. The HAL User Manual.

**NAME**

ddt – Compute the derivative of the input function

**SYNOPSIS**

**loadrt ddt [count=*N*]**

**FUNCTIONS**

**ddt.*N*** (uses floating-point)

**PINS**

**ddt.*N.in*** float in

**ddt.*N.out*** float out

**LICENSE**

GPL

**NAME**

deadzone

**SYNOPSIS****loadrt deadzone [count=*N*]****FUNCTIONS****deadzone.*N*** (uses floating-point)Update **out** based on **in** and the parameters.**PINS****deadzone.*N.in*** float in**deadzone.*N.out*** float out**PARAMETERS****deadzone.*N.center*** float rw (default: *0.0*)

The center of the dead zone

**deadzone.*N.threshold*** float rw (default: *1.0*)The dead zone is **center**  $\pm$  (**threshold**/2)**LICENSE**

GPL

**NAME**

debounce – filter noisy digital inputs

**SYNOPSIS**

**loadrt debounce [cfg=size[,size,...]]**

Creates filter groups each with the given number of filters (*size*). Each filter group has the same sample rate and delay.

**DESCRIPTION**

The debounce filter works by incrementing a counter whenever the input is true, and decrementing the counter when it is false. If the counter decrements to zero, the output is set false and the counter ignores further decrements. If the counter increments up to a threshold, the output is set true and the counter ignores further increments. If the counter is between zero and the threshold, the output retains its previous state. The threshold determines the amount of filtering: a threshold of 1 does no filtering at all, and a threshold of N requires a signal to be present for N samples before the output changes state.

**FUNCTIONS**

**debounce.G**

Sample all the input pins in group G and update the output pins.

**PINS**

**debounce.G.F.in** bit in

The F'th input pin in group G.

**debounce.G.F.out** bit out

The F'th output pin in group G. Reflects the last "stable" input seen on the corresponding input pin.

**PARAMETERS**

**debounce.G.delay** signed rw

Sets the amount of filtering for all pins in group G.

**NAME**

edge – Edge detector

**SYNOPSIS**

**loadrt edge [count=*N*]**

**FUNCTIONS**

**edge.*N*** Produce output pulses from input edges

**PINS**

**edge.*N*.in** bit in

**edge.*N*.out** bit out

Goes high when the desired edge is seen on 'in'

**edge.*N*.out-invert** bit out

Goes low when the desired edge is seen on 'in'

**PARAMETERS**

**edge.*N*.in-edge** bit rw (default: *TRUE*)

Selects the desired edge: TRUE means falling, FALSE means rising

**edge.*N*.out-width-ns** s32 rw (default: *0*)

Time in nanoseconds of the output pulse

**edge.*N*.time-left-ns** s32 r

Time left in this output pulse

**edge.*N*.last-in** bit r

Previous input value

**LICENSE**

GPL

**NAME**

encoder – software counting of quadrature encoder signals

**SYNOPSIS**

**loadrt encoder num\_chan=num**

**DESCRIPTION**

**encoder** is used to measure position by counting the pulses generated by a quadrature encoder. As a software-based implementation it is much less expensive than hardware, but has a limited maximum count rate. The limit is in the range of 10KHz to 50KHz, depending on the computer speed and other factors. If better performance is needed, a hardware encoder counter is a better choice. Some hardware-based systems can count at MHz rates.

**encoder** supports a maximum of eight channels. The number of channels actually loaded is set by the **num\_chan** argument when the module is loaded. If **numchan** is not specified, the default value is three.

**encoder** has a one-phase, unidirectional mode called *counter*. In this mode, the **phase-B** input is ignored; the counts increase on each rising edge of **phase-A**. This mode may be useful for counting a unidirectional spindle with a single input line, though the noise-resistant characteristics of quadrature are lost.

**FUNCTIONS**

**encoder.update-counters** (no floating-point)

Does the actual counting, by sampling the encoder signals and decoding the quadrature waveforms. Must be called as frequently as possible, preferably twice as fast as the maximum desired count rate. Operates on all channels at once.

**encoder.capture-position** (uses floating point)

Captures the raw counts from **update-counters** and performs scaling and other necessary conversion, handles counter rollover, etc. Can (and should) be called less frequently than **update-counters**. Operates on all channels at once.

**PINS**

**encoder.N.phase-A** bit in

Quadrature input for encoder channel *N*.

**encoder.N.phase-B** bit in

Quadrature input.

**encoder.N.phase-Z** bit in

Index pulse input.

**encoder.N.reset** bit in

When true, **counts** and **position** are reset to zero immediately.

**encoder.N.index-enable** bit i/o

When true, **counts** and **position** are reset to zero on the next rising edge of **Phase-Z**. At the same time, **index-enable** is reset to zero to indicate that the rising edge has occurred.

**encoder.N.counts** s32 out

Position in encoder counts.

**encoder.N.position** float out

Position in scaled units (see **position-scale**)

**encoder.N.velocity** float out

Velocity in scaled units per second. **encoder** uses an algorithm that greatly reduces quantization noise as compared to simply differentiating the **position** output.



## PARAMETERS

**encoder.N.position-scale** float rw

Scale factor, in counts per length unit. For example, if **position-scale** is 500, then 1000 counts of the encoder will be reported as a position of 2.0 units.

**encoder.N.counter-mode** bit rw

Enables counter mode. When true, the counter counts each rising edge of the phase-A input, ignoring the value on phase-B. This is useful for counting the output of a single channel (non-quadrature) sensor. When false (the default), it counts in quadrature mode.

**encoder.N.x4-mode** bit rw

Enables times-4 mode. When true (the default), the counter counts each edge of the quadrature waveform (four counts per full cycle). When false, it only counts once per full cycle. In **counter-mode**, this parameter is ignored.

**encoder.N.rawcounts** s32 ro

The raw count, as determined by **update-counters**. This value is updated more frequently than **counts** and **position**. It is also unaffected by **reset** or the index pulse.

## SEE ALSO

**counter(9)**

**NAME**

encoder\_ratio – an electronic gear to synchronize two axes

**SYNOPSIS**

**loadrt encoder\_ratio [num\_chan=N]**

**DESCRIPTION**

**encoder\_ratio** can be used to synchronize two axes (like an "electronic gear"). It counts encoder pulses from both axes in software, and produces an error value that can be used with a PID loop to make the slave encoder track the master encoder with a specific ratio.

This module supports up to eight axis pairs. The number of pairs is set by the module parameter **num\_chan**.

**FUNCTIONS****encoder\_ratio.sample**

Read all input pins. Must be called at twice the maximum desired count rate.

**encoder\_ratio.update (uses floating-point)**

Updates all output pins. May be called from a slower thread.

**PINS**

**encoder\_ratio.N.master-A** bit in

**encoder\_ratio.N.master-B** bit in

**encoder\_ratio.N.slave-A** bit in

**encoder\_ratio.N.slave-B** bit in

The encoder channels of the master and slave axes

**encoder\_ratio.N.enable** bit in

When the enable pin is FALSE, the error pin simply reports the slave axis position, in revolutions. As such, it would normally be connected to the feedback pin of a PID block for closed loop control of the slave axis. Normally the command input of the PID block is left unconnected (zero), so the slave axis simply sits still. However when the enable input goes TRUE, the error pin becomes the slave position minus the scaled master position. The scale factor is the ratio of master teeth to slave teeth. As the master moves, error becomes non-zero, and the PID loop will drive the slave axis to track the master.

**encoder\_ratio.N.error** float out

The error in the position of the slave (in revolutions)

**PARAMETERS**

**encoder\_ratio.N.master-ppr** unsigned rw

**encoder\_ratio.N.slave-ppr** unsigned rw

The number of pulses per revolution of the master and slave axes

**encoder\_ratio.N.master-teeth** unsigned rw

**encoder\_ratio.N.slave-teeth** unsigned rw

The number of "teeth" on the master and slave gears.

**SEE ALSO**

**encoder(9)**

**NAME**

`estop_latch` – ESTOP latch which sets ok-out true and fault-out false only if ok-in is true, fault-in is false, and a rising edge is seen on reset. While ok-out is true, watchdog toggles, and can be used for chargepumps or similar needs.

**SYNOPSIS**

`loadrt estop_latch [count=N]`

**FUNCTIONS**

`estop-latch.N`

**PINS**

`estop-latch.N.ok-in` bit in  
`estop-latch.N.fault-in` bit in  
`estop-latch.N.reset` bit in  
`estop-latch.N.ok-out` bit out  
`estop-latch.N.fault-out` bit out  
`estop-latch.N.watchdog` bit out

**LICENSE**

GPL

**NAME**

flipflop – D type flip-flop

**SYNOPSIS**

**loadrt flipflop [count=*N*]**

**FUNCTIONS**

**flipflop.*N***

**PINS**

**flipflop.*N*.data** bit in  
data input

**flipflop.*N*.clk** bit in  
clock, rising edge writes data to out

**flipflop.*N*.set** bit in  
when true, force out true

**flipflop.*N*.reset** bit in  
when true, force out false; overrides set

**flipflop.*N*.out** bit io  
output

**LICENSE**

GPL

**NAME**

freqgen – software step pulse generation

**OBsolete** - see **stepgen**'s 'ctrl\_type=v' option.

**SYNOPSIS**

**loadrt freqgen step\_type=type0[,type1...]**

**DESCRIPTION**

**freqgen** is used to control stepper motors. The maximum step rate depends on the CPU and other factors, and is usually in the range of 10KHz to 50KHz. If higher rates are needed, a hardware step generator is a better choice.

**freqgen** runs the motor at a commanded velocity, subject to acceleration and velocity limits. It does not directly control position.

**freqgen** can control a maximum of eight motors. The number of motors/channels actually loaded depends on the number of *type* values given. The value of each *type* determines the outputs for that channel. **freqgen** supports 15 possible step types.

By far the most common step type is '0', standard step and direction. Others include up/down, quadrature, and a wide variety of three, four, and five phase patterns that can be used to directly control some types of motor windings. (When used with appropriate buffers of course.)

Some of the stepping types are described below, but for more details (including timing diagrams) see the **stepgen** section of the HAL reference manual.

type 0: step/dir

Two pins, one for step and one for direction. **make-pulses** must run at least twice for each step (once to set the step pin true, once to clear it). This limits the maximum step rate to half (or less) of the rate that can be reached by types 2-14. The parameters **steplen** and **stepspace** can further lower the maximum step rate. Parameters **dirsetup** and **dirhold** also apply to this step type.

type 1: up/down

Two pins, one for 'step up' and one for 'step down'. Like type 0, **make-pulses** must run twice per step, which limits the maximum speed.

type 2: quadrature

Two pins, phase-A and phase-B. For forward motion, A leads B. Can advance by one step every time **make-pulses** runs.

type 3: three phase, full step

Three pins, phase-A, phase-B, and phase-C. Three steps per full cycle, then repeats. Only one phase is high at a time - for forward motion the pattern is A, then B, then C, then A again.

type 4: three phase, half step

Three pins, phases A through C. Six steps per full cycle. First A is high alone, then A and B together, then B alone, then B and C together, etc.

types 5 through 8: four phase, full step

Four pins, phases A through D. Four steps per full cycle. Types 5 and 6 are suitable for use with unipolar steppers, where power is applied to the center tap of each winding, and four open-collector transistors drive the ends. Types 7 and 8 are suitable for bipolar steppers, driven by two H-bridges.

types 9 and 10: four phase, half step

Four pins, phases A through D. Eight steps per full cycle. Type 9 is suitable for unipolar drive, and type 10 for bipolar drive.

types 11 and 12: five phase, full step

Five pins, phases A through E. Five steps per full cycle. See HAL reference manual for the patterns.

types 13 and 14: five phase, half step

Five pins, phases A through E. Ten steps per full cycle. See HAL reference manual for the patterns.

## FUNCTIONS

**freqgen.make-pulses** (no floating-point)

Generates the step pulses, using information computed by **update-freq**. Must be called as frequently as possible, to maximize the attainable step rate and minimize jitter. Operates on all channels at once.

**freqgen.capture-position** (uses floating point)

Captures position feedback value from the high speed code and makes it available on a pin for use elsewhere in the system. Operates on all channels at once.

**freqgen.update-freq** (uses floating point)

Accepts a velocity command and converts it into a form usable by **make-pulses** for step generation. Operates on all channels at once.

## PINS

**freqgen.N.counts** s32 out

The current position, in counts, for channel *N*. Updated by **capture-position**.

**freqgen.N.position-fb** float out

The current position, in length units (see parameter **position-scale**). Updated by **capture-position**.

**freqgen.N.velocity** float in (**freqgen** only)

Commanded velocity, in length units per second (see parameter **velocity-scale**).

**freqgen.N.step** bit out (step type 0 only)

Step pulse output.

**freqgen.N.dir** bit out (step type 0 only)

Direction output: low for forward, high for reverse.

**freqgen.N.up** bit out (step type 1 only)

Count up output, pulses for forward steps.

**freqgen.N.down** bit out (step type 1 only)

Count down output, pulses for reverse steps.

**freqgen.N.phase-A** thru **phase-E** bit out (step types 2-14 only)

Output bits. **phase-A** and **phase-B** are present for step types 2-14, **phase-C** for types 3-14, **phase-D** for types 5-14, and **phase-E** for types 11-14. Behavior depends on selected stepping type.

## PARAMETERS

**freqgen.N.frequency** float ro

The current step rate, in steps per second, for channel *N*.

**freqgen.N.maxaccel** float rw

The acceleration/deceleration limit, in steps per second squared.

**freqgen.N.maxfreq** float rw (**freqgen** only)

The maximum allowable velocity, in steps per second. If the requested maximum velocity cannot be reached with the current **make-pulses** thread period, it will be reset to the highest attainable value.

**freqgen.N.position-scale** float rw

The scaling for position feedback, in steps per length unit.

**freqgen.N.velocity-scale** float rw

The scaling for the velocity command, in steps per length unit.

**freqgen.N.rawcounts** s32 ro

The position in counts, as updated by **make-pulses**. (Note: this is updated more frequently than the **counts** pin.)

**freqgen.N.steplen** u32 rw (step type 0 only)

The length of the step pulses, in **make-pulses** periods. Measured from rising edge to falling edge.

**freqgen.N.stepspace** u32 rw (step type 0 only)

The minimum space between step pulses, in **make-pulses** periods. Measured from falling edge to rising edge. The actual time depends on the step rate and can be much longer.

**freqgen.N.dirsetup** u32 rw (step type 0 only)

The minimum setup time from direction to step, in **make-pulses** periods. Measured from change of direction to rising edge of step.

**freqgen.N.dirhold** u32 rw (step type 0 only)

The minimum hold time of direction after step, in **make-pulses** periods. Measured from falling edge of step to change of direction.

## BUGS

**freqgen** should have an **enable** pin.

**freqgen**'s command pin should be called **velocity-cmd**, not **velocity**, for clarity and consistency with **stepgen**.

**freqgen** should use **maxvel**, not **maxfreq**. (In other words, the velocity limit should be scaled in length units per second, not steps per second. The scale parameter can be set to 1.0 if it is desired to work in steps instead of length units.)

**freqgen**'s maxaccel parameter should be in length units per second squared, not steps per second squared, for consistency with **stepgen**.

**freqgen** should use **position-scale** for scaling both command and feedback, **velocity-scale** is redundant and should be eliminated.

Step type 1 (up/down) should respect the **steplen** and **stepspace** limits.

Timing parameters **steplen**, **stepspace**, **dirsetup**, and **dirhold** should be in nano-seconds, not **make-pulses** periods. That would allow the period to be changed without requiring the parameters to be recalculated.

All of these bugs have been fixed in **stepgen**. Only **stepgen** will continue to be maintained, since **freqgen** contains large amounts of code that duplicates code in **stepgen**. Since **stepgen** can provide the same functionality, there is no reason to maintain the duplicate code. **freqgen** may be eliminated at any time, and almost certainly **will** be eliminated for the version 2.2 release of EMC.

## SEE ALSO

**stepgen(9)**

**NAME**

hypot – Three-input hypotenuse (Euclidean distance) calculator

**SYNOPSIS**

**loadrt hypot [count=*N*]**

**FUNCTIONS**

**hypot.*N*** (uses floating-point)

**PINS**

**hypot.*N.in0*** float in

**hypot.*N.in1*** float in

**hypot.*N.in2*** float in

**hypot.*N.out*** float out

$\text{out} = \text{sqrt}(\text{in0}^2 + \text{in1}^2 + \text{in2}^2)$

**LICENSE**

GPL



**NAME**

integ – Integrator

**SYNOPSIS**

**loadrt integ [count=*N*]**

**FUNCTIONS**

**integ.*N*** (uses floating-point)

**PINS**

**integ.*N*.in** float in

**integ.*N*.out** float out

The discrete integral of 'in' since 'reset' was deasserted

**integ.*N*.reset** bit in

When asserted, set out to 0

**LICENSE**

GPL

**NAME**

kinematics definitions for emc2

**SYNOPSIS**

**loadrt trivkins**

**loadrt rotatekins**

**loadrt tripodkins**

**loadrt genhexkins**

**DESCRIPTION**

Rather than exporting HAL pins and functions, these components provide the forward and inverse kinematics definitions for emc2.

**trivkins – Trivial Kinematics**

There is a 1:1 correspondence between joints and axes. Most standard milling machines and lathes use the trivial kinematics module.

**rotatekins – Rotated Kinematics**

The X and Y axes are rotated 45 degrees compared to the joints 0 and 1.

**tripodkins – Tripod Kinematics**

The joints represent the distance of the controlled point from three predefined locations (the motors), giving three degrees of freedom in position (XYZ)

**tripodkins.Bx**

**tripodkins.Cx**

**tripodkins.Cy**

The location of the three motors is (0,0), (Bx,0), and (Cx,Cy)

**genhexkins – Hexapod Kinematics**

Gives six degrees of freedom in position and orientation (XYZABC). The location of the motors is defined at compile time.

**SEE ALSO**

The Kinematics section of the EMC2 Developer Manual

**NAME**

knob2float – Convert counts (probably from an encoder) to a float value

**SYNOPSIS**

**loadrt knob2float [count=*N*]**

**FUNCTIONS**

**knob2float.*N*** (uses floating-point)

**PINS**

**knob2float.*N*.counts** s32 in  
Counts

**knob2float.*N*.enable** bit in  
When TRUE, output is controlled by count, when FALSE, output is fixed

**knob2float.*N*.scale** float in  
Amount of output change per count

**knob2float.*N*.out** float out  
Output value

**PARAMETERS**

**knob2float.*N*.max-out** float rw (default: *1.0*)  
Maximum output value, further increases in count will be ignored

**knob2float.*N*.min-out** float rw (default: *0.0*)  
Minimum output value, further decreases in count will be ignored

**LICENSE**

GPL

**NAME**

limit1 – Limit the output signal to fall between min and max

**SYNOPSIS**

**loadrt limit1 [count=*N*]**

**FUNCTIONS**

**limit1.*N*** (uses floating-point)

**PINS**

**limit1.*N.in*** float in

**limit1.*N.out*** float out

**PARAMETERS**

**limit1.*N.min*** float rw (default: *-1e20*)

**limit1.*N.max*** float rw (default: *1e20*)

**LICENSE**

GPL

**NAME**

limit2 – Limit the output signal to fall between min and max and limit its slew rate to less than maxv per second. When the signal is a position, this means that position and velocity are limited.

**SYNOPSIS**

**loadrt limit2 [count=*N*]**

**FUNCTIONS**

**limit2.*N*** (uses floating-point)

**PINS**

**limit2.*N.in*** float in

**limit2.*N.out*** float out

**PARAMETERS**

**limit2.*N.min*** float rw (default: *-1e20*)

**limit2.*N.max*** float rw (default: *1e20*)

**limit2.*N.maxv*** float rw (default: *1e20*)

**LICENSE**

GPL

**NAME**

limit3 – Limit the output signal to fall between min and max, limit its slew rate to less than maxv per second, and limit its second derivative to less than maxa per second squared. When the signal is a position, this means that the position, velocity, and acceleration are limited.

**SYNOPSIS**

**loadrt limit3 [count=*N*]**

**FUNCTIONS**

**limit3.*N*** (uses floating-point)

**PINS**

**limit3.*N.in*** float in

**limit3.*N.out*** float out

**PARAMETERS**

**limit3.*N.min*** float rw (default: *-1e20*)

**limit3.*N.max*** float rw (default: *1e20*)

**limit3.*N.maxv*** float rw (default: *1e20*)

**limit3.*N.maxa*** float rw (default: *1e20*)

**LICENSE**

GPL

**NAME**

logic

**SYNOPSIS**

**loadrt logic [count=*N*] [personality=*P,P,...*]**

**DESCRIPTION**

Experimental general ‘logic function’ component. Can perform ‘and’, ‘or’ and ‘xor’ of up to 16 inputs. Determine the proper value for ‘personality’ by adding:

- The number of input pins, usually from 2 to 16
- 256 (0x100) if the ‘and’ output is desired
- 512 (0x200) if the ‘or’ output is desired
- 1024 (0x400) if the ‘xor’ (exclusive or) output is desired

**FUNCTIONS**

**logic.*N***

**PINS**

**logic.*N.in-MM*** bit in (MM=00..personality & 0xff)

**logic.*N.and*** bit out [if personality & 0x100]

**logic.*N.or*** bit out [if personality & 0x200]

**logic.*N.xor*** bit out [if personality & 0x400]

**LICENSE**

GPL

**NAME**

lowpass – Low-pass filter

**SYNOPSIS**

**loadrt lowpass [count=*N*]**

**FUNCTIONS**

**lowpass.*N*** (uses floating-point)

**PINS**

**lowpass.*N.in*** float in

**lowpass.*N.out*** float out

out += (in - out) \* gain

**PARAMETERS**

**lowpass.*N.gain*** float rw

**LICENSE**

GPL



**NAME**

lut5 – Arbitrary 5-input logic function based on a look-up table

**SYNOPSIS**

**loadrt lut5 [count=*N*]**

**DESCRIPTION**

**lut5** constructs an arbitrary logic function with up to 5 inputs using a **look-up table**. The function is specified by the HAL pin **function**. The necessary value for **function** can be determined by writing the truth table, and computing the sum of the **weights** for which the output value should be TRUE.

**Example Functions**

A 5-input *and* function is TRUE only when all the inputs are true, so the correct value for **function** is **0x80000000**.

A 5-input *or* function is TRUE whenever any of the inputs are true, so the correct value for **function** is **0xffffffff**.

A 2-input *xor* function is TRUE whenever exactly one of the inputs is true, so the correct value for **function** is **0x6**. Only **in-0** and **in-1** should be connected to signals, because if any other bit is **TRUE** then the output will be **FALSE**.

Weights for each line of truth table					
Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Weight
0	0	0	0	0	0x1
0	0	0	0	1	0x2
0	0	0	1	0	0x4
0	0	0	1	1	0x8
0	0	1	0	0	0x10
0	0	1	0	1	0x20
0	0	1	1	0	0x40
0	0	1	1	1	0x80
0	1	0	0	0	0x100
0	1	0	0	1	0x200
0	1	0	1	0	0x400
0	1	0	1	1	0x800
0	1	1	0	0	0x1000
0	1	1	0	1	0x2000
0	1	1	1	0	0x4000
0	1	1	1	1	0x8000
1	0	0	0	0	0x10000
1	0	0	0	1	0x20000
1	0	0	1	0	0x40000
1	0	0	1	1	0x80000
1	0	1	0	0	0x100000
1	0	1	0	1	0x200000
1	0	1	1	0	0x400000
1	0	1	1	1	0x800000
1	1	0	0	0	0x1000000
1	1	0	0	1	0x2000000
1	1	0	1	0	0x4000000
1	1	0	1	1	0x8000000
1	1	1	0	0	0x10000000
1	1	1	0	1	0x20000000
1	1	1	1	0	0x40000000
1	1	1	1	1	0x80000000

**FUNCTIONS****lut5.*N*****PINS****lut5.*N*.in-0** bit in**lut5.*N*.in-1** bit in**lut5.*N*.in-2** bit in**lut5.*N*.in-3** bit in**lut5.*N*.in-4** bit in**lut5.*N*.out** bit out**PARAMETERS****lut5.*N*.function** u32 rw**LICENSE**

GPL

**NAME**

maj3 – Compute the majority of 3 inputs

**SYNOPSIS**

**loadrt maj3 [count=*N*]**

**FUNCTIONS**

**maj3.*N***

**PINS**

**maj3.*N*.in1** bit in

**maj3.*N*.in2** bit in

**maj3.*N*.in3** bit in

**maj3.*N*.out** bit out

**PARAMETERS**

**maj3.*N*.invert** bit rw

**LICENSE**

GPL

**NAME**

match8 – 8-bit binary match detector

**SYNOPSIS**

**loadrt match8 [count=*N*]**

**FUNCTIONS**

**match8.*N***

**PINS**

**match8.*N*.in** bit in (default: *TRUE*)

cascade input - if false, output is false regardless of other inputs

**match8.*N*.a0** bit in

**match8.*N*.a1** bit in

**match8.*N*.a2** bit in

**match8.*N*.a3** bit in

**match8.*N*.a4** bit in

**match8.*N*.a5** bit in

**match8.*N*.a6** bit in

**match8.*N*.a7** bit in

**match8.*N*.b0** bit in

**match8.*N*.b1** bit in

**match8.*N*.b2** bit in

**match8.*N*.b3** bit in

**match8.*N*.b4** bit in

**match8.*N*.b5** bit in

**match8.*N*.b6** bit in

**match8.*N*.b7** bit in

**match8.*N*.out** bit out

true only if in is true and a[m] matches b[m] for m = 0 thru 7

**LICENSE**

GPL

**NAME**

minmax – Track the minimum and maximum values of the input to the outputs

**SYNOPSIS**

**loadrt minmax [count=*N*]**

**FUNCTIONS**

**minmax.*N*** (uses floating-point)

**PINS**

**minmax.*N.in*** float in

**minmax.*N.reset*** bit in

When reset is asserted, 'in' is copied to the outputs

**minmax.*N.max*** float out

**minmax.*N.min*** float out

**LICENSE**

GPL

**NAME**

motion – accepts NML motion commands, interacts with HAL in realtime

**SYNOPSIS**

```
loadrt motmod [base_period_nsec=period] [servo_period_nsec=period] [traj_period_nsec=period]
[key=SHMEM_KEY] [num_joints=[0-9]]
```

**DESCRIPTION**

These pins and parameters are created by the realtime **motmod** module. This module provides a HAL interface for EMC's motion planner. Basically **motmod** takes in a list of waypoints and generates a nice blended and constraint-limited stream of joint positions to be fed to the motor drives.

Pin names starting with "**axis**" are actually joint values, but the pins and parameters are still called "**axis.N**". They are read and updated by the motion-controller function.

**PINS**

**axis.N.amp-enable-out** OUT bit

TRUE if the amplifier for this joint should be enabled

**axis.N.amp-fault-in** IN bit

Should be driven TRUE if an external fault is detected with the amplifier for this joint

**axis.N.home-sw-in** IN bit

Should be driven TRUE if the home switch for this joint is closed

**axis.N.homing** OUT bit

TRUE if the joint is currently homing

**axis.N.index-enable** IO BIT

Should be attached to the index-enable pin of the joint's encoder to enable homing to index pulse

**axis.N.jog-counts** IN s32

Connect to the "counts" pin of an external encoder to use a physical jog wheel.

**axis.N.jog-enable** IN bit

When TRUE (and in manual mode), any change to "jog-counts" will result in motion. When false, "jog-counts" is ignored.

**axis.N.jog-scale** IN float

Sets the distance moved for each count on "jog-counts", in machine units.

**axis.N.jog-vel-mode** IN bit

When FALSE (the default), the jogwheel operates in position mode. The axis will move exactly jog-scale units for each count, regardless of how long that might take. When TRUE, the wheel operates in velocity mode - motion stops when the wheel stops, even if that means the commanded motion is not completed.

**axis.N.joint-pos-cmd** OUT float

The joint (as opposed to motor) commanded position. There may be several offsets between the joint and motor coordinates: backlash compensation, screw error compensation, and home offsets.

**axis.N.joint-pos-fb** OUT float

The joint feedback position. This value is computed from the actual motor position minus joint offsets. Useful for machine visualization.

**axis.N.motor-pos-cmd** OUT float

The commanded position for this joint.

**axis.N.motor-pos-fb** IN float

The actual position for this joint.

**axis.N.neg-lim-sw-in** IN bit

Should be driven TRUE if the negative limit switch for this joint is tripped.

**axis.N.pos-lim-sw-in** IN bit

Should be driven TRUE if the positive limit switch for this joint is tripped.

**motion.adaptive-feed** IN float

When adaptive feed is enabled with M52 P1, the commanded velocity is multiplied by this value. This effect is multiplicative with the NML-level feed override value and motion.feed-hold.

(not yet implemented) **motion.analog-in-NN** IN float

These pins are used by M66 Enn wait-for-input mode.

**motion.digital-in-NN** IN bit

These pins are used by M66 Pnn wait-for-input mode.

**motion.digital-out-NN** OUT bit

These pins are controlled by the M62 through M65 words.

**motion.enable** IN bit

If this bit is driven FALSE, motion stops, the machine is placed in the "machine off" state, and a message is displayed for the operator. For normal motion, drive this bit TRUE.

**motion.feed-hold** IN bit

When Feed Stop Control is enabled with M53 P1 (See section, and this bit is TRUE, the feed rate is set to 0.

**motion.motion-inpos** OUT bit

TRUE if the machine is in position.

**motion.probe-input** IN bit

G38.2 uses the value on this pin to determine when the probe has made contact. TRUE for probe contact closed (touching), FALSE for probe contact open.

**motion.spindle-brake** OUT bit

TRUE when the spindle brake should be applied

**motion.spindle-forward** OUT bit

TRUE when the spindle should rotate forward

**motion.spindle-index-enable** I/O bit

For correct operation of spindle synchronized moves, this signal must be hooked to the index-enable pin of the spindle encoder.

**motion.spindle-on** OUT bit

TRUE when spindle should rotate

**motion.spindle-reverse** OUT bit

TRUE when the spindle should rotate backward

**motion.spindle-revs** IN float

For correct operation of spindle synchronized moves, this signal must be hooked to the position pin of the spindle encoder.

**motion.spindle-speed-in** IN float

Actual spindle speed feedback; used for G96 feed-per-revolution and constant surface speed modes.

**motion.spindle-speed-out** OUT float

Desired spindle speed in rotations per minute

**PARAMETERS**

Many of these parameters serve as debugging aids, and are subject to change or removal at any time.

**axis.N.active**

TRUE when this joint is active

**axis.N.backlash-corr**

Backlash or screw compensation raw value

**axis.N.backlash-filt**

Backlash or screw compensation filtered value (respecting motion limits)

**axis.N.backlash-vel**

Backlash or screw compensation velocity

**axis.N.coarse-pos-cmd****axis.N.error**

TRUE when this joint has encountered an error, such as a limit switch closing

**axis.N.f-error**

The actual following error



**axis.N.f-error-lim**

The following error limit

**axis.N.f-errored**

TRUE when this joint has exceeded the following error limit

**axis.N.faulted****axis.N.free-pos-cmd**

The "free planner" commanded position for this joint.

**axis.N.free-tp-enable**

TRUE when the "free planner" is enabled for this joint

**axis.N.free-vel-lim**

The velocity limit for the free planner

**axis.N.home-state**

Reflects the step of homing currently taking place

**axis.N.homed**

TRUE if the joint has been homed

**axis.N.in-position**

TRUE if the joint is using the "free planner" and has come to a stop

**axis.N.joint-vel-cmd**

The joint's commanded velocity

**axis.N.kb-jog-active****axis.N.neg-hard-limit**

The negative hard limit for the joint

(removed) axis.N.neg-soft-limit

The negative soft limit for the joint

**axis.N.pos-hard-limit**

The positive hard limit for the joint

(removed) axis.N.pos-soft-limit

The positive soft limit for the joint

**axis.N.wheel-jog-active**

**motion-command-handler.time**

**motion-command-handler.tmax**

**motion-controller.time**

**motion-controller.tmax**

**motion.coord-error**

TRUE when motion has encountered an error, such as exceeding a soft limit

**motion.coord-mode**

TRUE when motion is in "coordinated mode", as opposed to "teleop mode"

**motion.current-vel**

**motion.debug-\***

These values are used for debugging purposes.

**motion.in-position**

Same as the pin motion.motion-inpos

**motion.motion-enabled**

TRUE when motion is enabled

**motion.on-soft-limit**

**motion.program-line**

**motion.servo.last-period**

The number of CPU cycles between invocations of the servo thread. Typically, this number divided by the CPU speed gives the time in seconds, and can be used to determine whether the realtime motion controller is meeting its timing constraints

**motion.servo.overruns**

By noting large differences between successive values of motion.servo.last-period, the motion controller can determine that there has probably been a failure to meet its timing constraints. Each time such a failure is detected, this value is incremented.

**motion.teleop-mode**

TRUE when motion is in "teleop mode", as opposed to "coordinated mode"

## FUNCTIONS

Generally, these functions are both added to the servo-thread in the order shown.

### **motion-command-handler**

Processes motion commands coming from user space

### **motion-controller**

Runs the emc motion controller

## BUGS

This manual page is horribly incomplete.

## SEE ALSO

iocontrol(1)

**NAME**

mult2 – Product of two inputs

**SYNOPSIS**

**loadrt mult2 [count=*N*]**

**FUNCTIONS**

**mult2.*N*** (uses floating-point)

**PINS**

**mult2.*N*.in0** float in

**mult2.*N*.in1** float in

**mult2.*N*.out** float out

out = in0 \* in1

**LICENSE**

GPL

**NAME**

mux2 – Select from one of two input values

**SYNOPSIS**

**loadrt mux2 [count=*N*]**

**FUNCTIONS**

**mux2.*N*** (uses floating-point)

**PINS**

**mux2.*N*.sel** bit in

**mux2.*N*.out** float out

Follows the value of in0 if sel is FALSE, or in1 if sel is TRUE

**mux2.*N*.in1** float in

**mux2.*N*.in0** float in

**LICENSE**

GPL

**NAME**

mux4 – Select from one of four input values

**SYNOPSIS**

**loadrt mux4 [count=*N*]**

**FUNCTIONS**

**mux4.*N*** (uses floating-point)

**PINS**

**mux4.*N*.sel0** bit in

**mux4.*N*.sel1** bit in

Together, these determine which **in** value is copied to **out**.

**mux4.*N*.out** float out

Follows the value of one of the **in***N* values according to the two **sel** values

**sel1=FALSE, sel0=FALSE**

**out** follows **in0**

**sel1=FALSE, sel0=TRUE**

**out** follows **in1**

**sel1=TRUE, sel0=FALSE**

**out** follows **in2**

**sel1=TRUE, sel0=TRUE**

**out** follows **in3**

**mux4.*N*.in0** float in

**mux4.*N*.in1** float in

**mux4.*N*.in2** float in

**mux4.*N*.in3** float in

**LICENSE**

GPL

**NAME**

not – Inverter

**SYNOPSIS**

**loadrt not [count=*N*]**

**FUNCTIONS**

**not.*N***

**PINS**

**not.*N*.in** bit in

**not.*N*.out** bit out

**LICENSE**

GPL

**NAME**

offset – Adds an offset to an input, and subtracts it from the feedback value

**SYNOPSIS**

**loadrt offset [count=*N*]**

**FUNCTIONS**

**offset.*N*.update-output** (uses floating-point)

Updated the output value by adding the offset to the input

**offset.*N*.update-feedback** (uses floating-point)

Update the feedback value by subtracting the offset from the feedback

**PINS**

**offset.*N*.offset** float in

The offset value

**offset.*N*.in** float in

The input value

**offset.*N*.out** float out

The output value

**offset.*N*.fb-in** float in

The feedback input value

**offset.*N*.fb-out** float out

The feedback output value

**LICENSE**

GPL



**NAME**

oneshot – one-shot pulse generator

**SYNOPSIS**

**loadrt oneshot [count=*N*]**

**FUNCTIONS**

**oneshot.*N*** (uses floating-point)

Produce output pulses from input edges

**PINS**

**oneshot.*N.in*** bit in

Trigger input

**oneshot.*N.out*** bit out

Active high pulse

**oneshot.*N.out-not*** bit out

Active low pulse

**oneshot.*N.width*** float in (default: *0*)

Pulse width in seconds

**oneshot.*N.time-left*** float out

Time left in current output pulse

**PARAMETERS**

**oneshot.*N.retriggerable*** bit rw (default: *TRUE*)

Allow additional edges to extend pulse

**oneshot.*N.rising*** bit rw (default: *TRUE*)

Trigger on rising edge

**oneshot.*N.falling*** bit rw (default: *FALSE*)

Trigger on falling edge

**LICENSE**

GPL

**NAME**

or2 – Two-input OR gate

**SYNOPSIS**

**loadrt or2 [count=*N*]**

**FUNCTIONS**

**or2.*N***

**PINS**

**or2.*N*.in0** bit in

**or2.*N*.in1** bit in

**or2.*N*.out** bit out

**out** is computed from the value of **in0** and **in1** according to the following rule:

**in0=FALSE in1=FALSE**

**out=FALSE**

Otherwise,

**out=TRUE**

**LICENSE**

GPL

**NAME**

`pid` – proportional/integral/derivative controller

**SYNOPSIS**

```
loadrt pid num_chan=num [debug=dbg]
```

**DESCRIPTION**

**pid** is a classic Proportional/Integral/Derivative controller, used to control position or speed feedback loops for servo motors and other closed-loop applications.

**pid** supports a maximum of sixteen controllers. The number that are actually loaded is set by the **num\_chan** argument when the module is loaded. If **numchan** is not specified, the default value is three. If **debug** is set to 1 (the default is 0), some additional HAL parameters will be exported, which might be useful for tuning, but are otherwise unnecessary.

**FUNCTIONS**

**pid.N.do-pid-calcs** (uses floating-point)

Does the PID calculations for control loop *N*.

**PINS**

**pid.N.command** float in

The desired (commanded) value for the control loop.

**pid.N.feedback** float in

The actual (feedback) value, from some sensor such as an encoder.

**pid.N.error** float out

The difference between command and feedback.

**pid.N.output** float out

The output of the PID loop, which goes to some actuator such as a motor.

**pid.N.enable** bit in

When true, enables the PID calculations. When false, **output** is zero, and all internal integrators, etc, are reset.

**PARAMETERS**

**pid.N.Pgain** float rw

Proportional gain. Results in a contribution to the output that is the error multiplied by **Pgain**.

**pid.N.Igain** float rw

Integral gain. Results in a contribution to the output that is the integral of the error multiplied by **Igain**. For example an error of 0.02 that lasted 10 seconds would result in an integrated error (**errorI**) of 0.2, and if **Igain** is 20, the integral term would add 4.0 to the output.

**pid.N.Dgain** float rw

Derivative gain. Results in a contribution to the output that is the rate of change (derivative) of the error multiplied by **Dgain**. For example an error that changed from 0.02 to 0.03 over 0.2 seconds would result in an error derivative (**errorD**) of 0.05, and if **Dgain** is 5, the derivative term would add 0.25 to the output.

**pid.N.bias** float rw

**bias** is a constant amount that is added to the output. In most cases it should be left at zero. However, it can sometimes be useful to compensate for offsets in servo amplifiers, or to balance the weight of an object that moves vertically. **bias** is turned off when the PID loop is disabled, just like all other components of the output. If a non-zero output is needed even when the PID loop is disabled, it should be added with an external HAL sum2 block.

**pid.N.FF0** float rw

Zero order feed-forward term. Produces a contribution to the output that is **FF0** multiplied by the commanded value. For position loops, it should usually be left at zero. For velocity loops, **FF0** can compensate for friction or motor counter-EMF and may permit better tuning if used properly.

**pid.N.FF1** float rw

First order feed-forward term. Produces a contribution to the output that **FF1** multiplied by the derivative of the commanded value. For position loops, the contribution is proportional to speed, and can be used to compensate for friction or motor CEMF. For velocity loops, it is proportional to acceleration and can compensate for inertia. In both cases, it can result in better tuning if used properly.

**pid.N.FF2** float rw

Second order feed-forward term. Produces a contribution to the output that is **FF2** multiplied by the second derivative of the commanded value. For position loops, the contribution is proportional to acceleration, and can be used to compensate for inertia. For velocity loops, it should usually be left at zero.

**pid.N.deadband** float rw

Defines a range of "acceptable" error. If the absolute value of **error** is less than **deadband**, it will be treated as if the error is zero. When using feedback devices such as encoders that are inherently quantized, the deadband should be set slightly more than one-half count, to prevent the control loop from hunting back and forth if the command is between two adjacent encoder values. When the absolute value of the error is greater than the deadband, the deadband value is subtracted from the error before performing the loop calculations, to prevent a step in the transfer function at the edge of the deadband. (See **BUGS**.)

**pid.N.maxoutput** float rw

Output limit. The absolute value of the output will not be permitted to exceed **maxoutput**, unless **maxoutput** is zero. When the output is limited, the error integrator will hold instead of integrating, to prevent windup and overshoot.

**pid.N.maxerror** float rw

Limit on the internal error variable used for P, I, and D. Can be used to prevent high **Pgain** values from generating large outputs under conditions when the error is large (for example, when the command makes a step change). Not normally needed, but can be useful when tuning non-linear systems.

**pid.N.maxerrorD** float rw

Limit on the error derivative. The rate of change of error used by the **Dgain** term will be limited to this value, unless the value is zero. Can be used to limit the effect of **Dgain** and prevent large output spikes due to steps on the command and/or feedback. Not normally needed.

**pid.N.maxerrorI** float rw

Limit on error integrator. The error integrator used by the **Igain** term will be limited to this value, unless it is zero. Can be used to prevent integrator windup and the resulting overshoot during/after sustained errors. Not normally needed.

**pid.N.maxcmdD** float rw

Limit on command derivative. The command derivative used by **FF1** will be limited to this value, unless the value is zero. Can be used to prevent **FF1** from producing large output spikes if there is a step change on the command. Not normally needed.

**pid.N.maxcmdDD** float rw

Limit on command second derivative. The command second derivative used by **FF2** will be limited to this value, unless the value is zero. Can be used to prevent **FF2** from producing large output spikes if there is a step change on the command. Not normally needed.

**pid.N.errorI** float ro (only if debug=1)

Integral of error. This is the value that is multiplied by **Igain** to produce the Integral term of the output.

**pid.N.errorD** float ro (only if debug=1)

Derivative of error. This is the value that is multiplied by **Dgain** to produce the Derivative term of the output.

**pid.N.commandD** float ro (only if debug=1)

Derivative of command. This is the value that is multiplied by **FF1** to produce the first order feed-forward term of the output.

**pid.N.commandDD** float ro (only if debug=1)

Second derivative of command. This is the value that is multiplied by **FF2** to produce the second order feed-forward term of the output.

## BUGS

Some people would argue that deadband should be implemented such that error is treated as zero if it is within the deadband, and be unmodified if it is outside the deadband. This was not done because it would cause a step in the transfer function equal to the size of the deadband. People who prefer that behavior are welcome to add a parameter that will change the behavior, or to write their own version of **pid**. However, the default behavior should not be changed.

**NAME**

`pluto_servo` – Hardware driver and firmware for the Pluto-P parallel-port FPGA, for use with servo machines.

**SYNOPSIS**

**loadrt** `pluto_servo` [`ioaddr=N`] [`ioaddr_hi=N`] [`epp_wide=N`] [`watchdog=N`] [`test_encoder=N`]

**ioaddr** [default: 0x378]

The base address of the parallel port.

**ioaddr\_hi** [default: 0]

The secondary address of the parallel port, used to set EPP mode. 0 means to use `ioaddr + 0x400`. -1 means there is no secondary address. The secondary address is used to set the port to EPP mode.

**epp\_wide** [default: 1]

Set to zero to disable the "wide EPP mode". "Wide" mode allows a 16- and 32-bit EPP transfers, which can reduce the time spent in the read and write functions. However, this may not work on all EPP parallel ports.

**watchdog** [default: 1]

Set to zero to disable the "hardware watchdog". "Watchdog" will tristate all outputs approximately 6ms after the last execution of **`pluto_servo.write`**, which adds some protection in the case of emc crashes.

**test\_encoder** [default: 0]

Internally connect `dout0..2` to `QA0`, `QB0`, `QZ0` to test quadrature counting

**DESCRIPTION**

`Pluto_servo` is an emc2 software driver and associated firmware that allow the Pluto-P board to be used to control a servo-based CNC machine.

The driver has 4 PWM channels, 4 quadrature channels with index pulse, 18 digital outputs (8 shared with PWM), and 20 digital inputs (12 shared with quadrature).

**Encoders**

The encoder pins and parameters conform to the 'canonical encoder' interface described in the HAL manual. It operates in 'x4 mode'.

The sample rate of the encoder is 40MHz. The maximum number quadrature rate is 8191 counts per emc2 servo cycle. For correct handling of the index pulse, the number of encoder counts per revolution must be less than 8191.

**PWM**

The PWM pins and parameters conform to the 'canonical analog output' interface described in the HAL manual. The output pins are 'up/down' or 'pwm/dir' pins as described in the documentation of the 'pwm-gen' component.

Internally the PWM generator is based on a 12-bit, 40MHz counter, giving 4095 duty cycles from -100% to +100% and a frequency of approximately 19.5kHz. In PDM mode, the duty periods are approximately 100ns long.

**Digital I/O**

The digital output pins conform to the 'canonical digital output' interface described in the HAL manual.

The digital input pins conform to the 'canonical digital input' interface described in the HAL manual.

## FUNCTIONS

- pluto-servo.read** (uses floating-point)  
Read all the inputs from the pluto-servo board
- pluto-servo.write** (uses floating-point)  
Write all the outputs on the pluto-servo board

## PINS

- pluto-servo.encoder.M.count** s32 out (M=0..3)  
**pluto-servo.encoder.M.position** float out (M=0..3)  
**pluto-servo.encoder.M.velocity** float out (M=0..3)  
**pluto-servo.encoder.M.reset** bit in (M=0..3)  
**pluto-servo.encoder.M.index-enable** bit io (M=0..3)  
encoder.M corresponds to the pins labeled QAM, QBM, and QZM on the pinout diagram
- pluto-servo.pwm.M.value** float in (M=0..3)  
**pluto-servo.pwm.M.enable** bit in (M=0..3)  
pwm.M corresponds to the pins labeled UPM and DNM on the pinout diagram
- pluto-servo.dout.MM** bit in (MM=00..19)  
dout.0M corresponds to the pin labeled OUTM on the pinout diagram. Other pins are shared with the PWM function, as follows:

Pin	Shared with	Pin	Shared with
dout.10	UP0	dout.11	DOWN0
dout.12	UP1	dout.13	DOWN1
dout.14	UP2	dout.15	DOWN2
dout.18	UP3	dout.19	DOWN3

- pluto-servo.din.MM** bit out (MM=00..19)  
**pluto-servo.din.MM-not** bit out (MM=00..19)  
For M=0 through 7, din.0M corresponds to the pin labeled INM on the pinout diagram. Other pins are shared with the encoder function, as follows:

Pin	Shared with	Pin	Shared with
din.8	QZ0	din.9	QZ1
din.10	QZ2	din.11	QZ3
din.12	QB0	din.13	QB1
din.14	QB2	din.15	QB3
din.16	QA0	din.17	QA1
din.18	QA2	din.19	QA3

## PARAMETERS

- pluto-servo.encoder.M.scale** float rw (M=0..3) (default: 1)  
**pluto-servo.encoder.z-polarity** bit rw  
Set to TRUE if the index pulse is active low, FALSE if it is active high. Affects all encoders.
- pluto-servo.pwm.M.offset** float rw (M=0..3)  
**pluto-servo.pwm.M.scale** float rw (M=0..3) (default: 1)  
**pluto-servo.pwm.M.max-dc** float rw (M=0..3) (default: 1)  
**pluto-servo.pwm.M.min-dc** float rw (M=0..3) (default: 0)  
**pluto-servo.pwm.M.pwmdir** bit rw (M=0..3) (default: 0)  
Set to TRUE use PWM+direction mode. Set to FALSE to use Up/Down mode.
- pluto-servo.pwm.is-pdm** bit rw  
Set to TRUE to use PDM (also called interleaved PWM) mode. Set to FALSE to use traditional PWM mode. Affects all PWM outputs.

**pluto-servo.dout.MM-invert** bit rw (MM=00..19)

If TRUE, the output on the corresponding **dout.MM** is inverted.

**pluto-servo.communication-error** u32 rw

Incremented each time pluto-servo.read detects an error code in the EPP status register. While this register is nonzero, new values are not being written to the Pluto-P board, and the status of digital outputs and the PWM duty cycle of the PWM outputs will remain unchanged. If the watchdog is enabled, it will activate soon after the communication error is detected. To continue after a communication error, set this parameter back to zero.

**pluto-servo.debug-0** s32 rw

**pluto-servo.debug-1** s32 rw

These parameters can display values which are useful to developers or for debugging the driver and firmware. They are not useful for integrators or users.

## SEE ALSO

The *pluto\_servo* section in the HAL User Manual, which shows the location of each physical pin on the pluto board.

## LICENSE

GPL



**NAME**

`pluto_step` – Hardware driver and firmware for the Pluto-P parallel-port FPGA, for use with stepper machines.

**SYNOPSIS**

**Note:** In this release of emc2, this driver is **alpha-quality** and not suitable for use on production machines.

**loadrt** `pluto_step` **ioaddr**=*addr* **ioaddr\_hi**=*addr* **epp\_wide**=[0/1]

**ioaddr** [default: 0x378]

The base address of the parallel port.

**ioaddr\_hi** [default: 0]

The secondary address of the parallel port, used to set EPP mode. 0 means to use `ioaddr + 0x400`. -1 means there is no secondary address.

**epp\_wide** [default: 1]

Set to zero to disable "wide EPP mode". "Wide" mode allows 16- and 32-bit EPP transfers, which can reduce the time spent in the read and write functions. However, this mode may not work on all EPP parallel ports.

**watchdog** [default: 1]

Set to zero to disable the "hardware watchdog". "Watchdog" will tristate all outputs approximately 6ms after the last execution of **pluto\_step.write**, which adds some protection in the case of emc crashes.

**speedrange** [default: 0]

Selects one of four speed ranges:

- 0: Top speed 312.5kHz; minimum speed 610Hz
- 1: Top speed 156.25kHz; minimum speed 305Hz
- 2: Top speed 78.125kHz; minimum speed 153Hz
- 3: Top speed 39.06kHz; minimum speed 76Hz

Choosing the smallest maximum speed that is above the maximum for any one axis may give improved step regularity at low step speeds.

**DESCRIPTION**

`Pluto_step` is an emc2 software driver and associated firmware that allow the Pluto-P board to be used to control a stepper-based CNC machine.

The driver has 4 step+direction channels, 14 dedicated digital outputs, and 16 dedicated digital inputs.

**Step generators**

The step generator takes a position input and output.

The step waveform includes step length/space and direction hold/setup time. Step length and direction setup/hold time is enforced in the FPGA. Step space is enforced by a velocity cap in the driver.

*(all the following numbers are subject to change)* In `speedrange=0`, the maximum step rate is 312.5kHz. For position feedback to be accurate, the maximum step rate is 512 pulses per servo cycle (so a 1kHz servo cycle does not impose any additional limitation). The maximum step rate may be lowered by the step length and space parameters, which are rounded up to the nearest multiple of 1600ns.

In successive speedranges the maximum step rate is divided in half, as is the maximum steps per servo

cycle, and the minimum nonzero step rate.

### Digital I/O

The digital output pins conform to the ‘canonical digital output’ interface described in the HAL manual.

The digital input pins conform to the ‘canonical digital input’ interface described in the HAL manual.

### FUNCTIONS

**pluto-step.read** (uses floating-point)

Read all the inputs from the pluto-step board

**pluto-step.write** (uses floating-point)

Write all the outputs on the pluto-step board

### PINS

**pluto-step.stepgen.M.position-cmd** float in (M=0..3)

**pluto-step.stepgen.M.velocity-fb** float out (M=0..3)

**pluto-step.stepgen.M.position-fb** float out (M=0..3)

**pluto-step.stepgen.M.counts** s32 out (M=0..3)

**pluto-step.stepgen.M.enable** bit in (M=0..3)

**pluto-step.stepgen.M.reset** bit in (M=0..3)

When TRUE, reset position-fb to 0

**pluto-step.dout.MM** bit in (MM=00..13)

dout.MM corresponds to the pin labeled OUTM on the pinout diagram.

**pluto-step.din.MM** bit out (MM=00..15)

**pluto-step.din.MM-not** bit out (MM=00..15)

din.MM corresponds to the pin labeled INM on the pinout diagram.

### PARAMETERS

**pluto-step.stepgen.M.scale** float rw (M=0..3) (default: 1.0)

**pluto-step.stepgen.M.maxvel** float rw (M=0..3) (default: 0)

**pluto-step.stepgen.step-polarity** bit rw

**pluto-step.stepgen.steplen** u32 rw

Step length in ns.

**pluto-step.stepgen.stepspace** u32 rw

Step space in ns

**pluto-step.stepgen.dirttime** u32 rw

Dir hold/setup in ns. Refer to the pdf documentation for a diagram of what these timings mean.

**pluto-step.dout.MM-invert** bit rw (MM=00..13)

If TRUE, the output on the corresponding **dout.MM** is inverted.

**pluto-step.communication-error** u32 rw

Incremented each time pluto-step.read detects an error code in the EPP status register. While this register is nonzero, new values are not being written to the Pluto-P board, and the status of digital outputs and the PWM duty cycle of the PWM outputs will remain unchanged. If the hardware watchdog is enabled, it will activate shortly after the communication error is detected by emc. To continue after a communication error, set this parameter back to zero.

**pluto-step.debug-0** s32 rw

**pluto-step.debug-1** s32 rw

**pluto-step.debug-2** float rw (default: .5)

**pluto-step.debug-3** float rw (default: 2.0)

Registers that hold debugging information of interest to developers

**SEE ALSO**

The *pluto\_step* section in the HAL User Manual, which shows the location of each physical pin on the pluto board.

**LICENSE**

GPL

**NAME**

**pwmgen** – software PWM/PDM generation

**SYNOPSIS**

**loadrt pwmgen output\_type=type0[,type1...]**

**DESCRIPTION**

**pwmgen** is used to generate PWM (pulse width modulation) or PDM (pulse density modulation) signals. The maximum PWM frequency and the resolution is quite limited compared to hardware-based approaches, but in many cases software PWM can be very useful. If better performance is needed, a hardware PWM generator is a better choice.

**pwmgen** supports a maximum of eight channels. The number of channels actually loaded depends on the number of *type* values given. The value of each *type* determines the outputs for that channel.

type 0: single output

A single output pin, **pwm**, whose duty cycle is determined by the input value for positive inputs, and which is off (or at **min-dc**) for negative inputs. Suitable for single ended circuits.

type 1: pwm/direction

Two output pins, **pwm** and **dir**. The duty cycle on **pwm** varies as a function of the input value. **dir** is low for positive inputs and high for negative inputs.

type 2: up/down

Two output pins, **up** and **down**. For positive inputs, the PWM/PDM waveform appears on **up**, while **down** is low. For negative inputs, the waveform appears on **down**, while **up** is low. Suitable for driving the two sides of an H-bridge to generate a bipolar output.

**FUNCTIONS**

**pwmgen.make-pulses** (no floating-point)

Generates the actual PWM waveforms, using information computed by **update**. Must be called as frequently as possible, to maximize the attainable PWM frequency and resolution, and minimize jitter. Operates on all channels at once.

**pwmgen.update** (uses floating point)

Accepts an input value, performs scaling and limit checks, and converts it into a form usable by **make-pulses** for PWM/PDM generation. Can (and should) be called less frequently than **make-pulses**. Operates on all channels at once.

**PINS**

**pwmgen.N.enable** bit in

Enables PWM generator *N* - when false, all **pwmgen.N** output pins are low.

**pwmgen.N.value** float in

Commanded value. When **value** = 0.0, duty cycle is 0%, and when **value** = +/-**scale**, duty cycle is +/- 100%. (Subject to **min-dc** and **max-dc** limitations.)

**pwmgen.N.pwm** bit out (output types 0 and 1 only)

PWM/PDM waveform.

**pwmgen.N.dir** bit out (output type 1 only)

Direction output: low for forward, high for reverse.

**pwmgen.N.up** bit out (output type 2 only)

PWM/PDM waveform for positive input values, low for negative inputs.

**pwmgen.N.down** bit out (output type 2 only)

PWM/PDM waveform for negative input values, low for positive inputs.

## PARAMETERS

**pwmgen.N.curr-dc** float ro

The current duty cycle, after all scaling and limits have been applied. Range is from -1.0 to +1.0.

**pwmgen.N.max-dc** float rw

The maximum duty cycle. A value of 1.0 corresponds to 100%. This can be useful when using transistor drivers with bootstrapped power supplies, since the supply requires some low time to recharge.

**pwmgen.N.min-dc** float rw

The minimum duty cycle. A value of 1.0 corresponds to 100%. Note that when the pwm generator is disabled, the outputs are constantly low, regardless of the setting of **min-dc**.

**pwmgen.N.scale** float rw

**pwmgen.N.offset** float rw

These parameters provide a scale and offset from the **value** pin to the actual duty cycle. The duty cycle is calculated according to  $dc = (value/scale) + offset$ , with 1.0 meaning 100%.

**pwmgen.N.pwm-freq** float rw

PWM frequency in Hz. The upper limit is half of the frequency at which **make-pulses** is invoked, and values above that limit will be changed to the limit. If **dither-pwm** is false, the value will be changed to the nearest integer submultiple of the **make-pulses** frequency. A value of zero produces Pulse Density Modulation instead of Pulse Width Modulation.

**pwmgen.N.dither-pwm** bit rw

Because software-generated PWM uses a fairly slow timebase (several to many microseconds), it has limited resolution. For example, if **make-pulses** is called at a 20KHz rate, and **pwm-freq** is 2KHz, there are only 10 possible duty cycles. If **dither-pwm** is false, the commanded duty cycle will be rounded to the nearest of those values. Assuming **value** remains constant, the same output will repeat every PWM cycle. If **dither-pwm** is true, the output duty cycle will be dithered between the two closest values, so that the long-term average is closer to the desired level. **dither-pwm** has no effect if **pwm-freq** is zero (PDM mode), since PDM is an inherently dithered process.

**NAME**

sample\_hold – Sample and Hold

**SYNOPSIS**

**loadrt sample\_hold [count=*N*]**

**FUNCTIONS**

**sample-hold.*N***

**PINS**

**sample-hold.*N*.in** s32 in

**sample-hold.*N*.hold** bit in

**sample-hold.*N*.out** s32 out

**LICENSE**

GPL

**NAME**

sampler – sample data from HAL in real time

**SYNOPSIS**

**loadrt sampler depth=depth1[,depth2...] cfg=string1[,string2...]**

**DESCRIPTION**

**sampler** and **halsampler**(1) are used together to sample HAL data in real time and store it in a file. **sampler** is a realtime HAL component that exports HAL pins and creates a FIFO in shared memory. It then begins sampling data from the HAL and storing it to the FIFO. **hal\_sampler** is a user space program that copies data from the FIFO to stdout, where it can be redirected to a file or piped to some other program.

**OPTIONS**

**depth=depth1[,depth2...]**

sets the depth of the realtime->user FIFO that **sampler** creates to buffer the realtime data. Multiple values of *depth* (separated by commas) can be specified if you need more than one FIFO (for example if you want to sample data from two different realtime threads).

**cfg=string1[,string2...]**

defines the set of HAL pins that **sampler** exports and later samples data from. One *string* must be supplied for each FIFO, separated by commas. **sampler** exports one pin for each character in *string*. Legal characters are:

**F, f** (float pin)

**B, b** (bit pin)

**S, s** (s32 pin)

**U, u** (u32 pin)

**FUNCTIONS**

**sampler.N**

One function is created per FIFO, numbered from zero.

**PINS**

**sampler.N.pin.M** input

Pin for the data that will wind up in column *M* of FIFO *N* (and in column *M* of the output file). The pin type depends on the config string.

**sampler.N.curr-depth** s32 output

Current number of samples in the FIFO. When this reaches *depth* new data will begin overwriting old data, and some samples will be lost.

**sampler.N.full** bit output

TRUE when the FIFO *N* is full, FALSE when there is room for another sample.

**sampler.N.enable** bit input

When TRUE, samples are captured and placed in FIFO *N*, when FALSE, no samples are acquired. Defaults to TRUE.

**PARAMETERS**

**sampler.N.overruns** s32 read/write

The number of times that **sampler** has tried to write data to the HAL pins but found no room in the FIFO. It increments whenever **full** is true, and can be reset by the **setp** command.

**sampler.N.sample-num** s32 read/write

A number that identifies the sample. It is automatically incremented for each sample, and can be reset using the **setp** command. The sample number can optionally be printed in the first column of the output from **halsampler**, using the **-t** option. (see **man 1 halsampler**)

## SEE ALSO

**halsampler(1)** **streamer(9)** **halstreamer(1)**

## HISTORY

## BUGS

Should an **enable** HAL pin be added, to allow sampling to be turned on and off?

## AUTHOR

Original version by John Kasunich, as part of the Enhanced Machine Controller (EMC) project. Improvements by several other members of the EMC development team.

## REPORTING BUGS

Report bugs to [jmkasunich AT users DOT sourceforge DOT net](mailto:jmkasunich@users.sourceforge.net)

## COPYRIGHT

Copyright © 2006 John Kasunich.

This is free software; see the source for copying conditions. There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.



**NAME**

scale

**SYNOPSIS****loadrt scale [count=*N*]****FUNCTIONS****scale.*N*** (uses floating-point)**PINS****scale.*N*.in** float in**scale.*N*.gain** float in**scale.*N*.offset** float in**scale.*N*.out** float out

out = in \* gain + offset

**LICENSE**

GPL

**NAME**

select8 – 8-bit binary match detector

**SYNOPSIS**

**loadrt select8 [count=*N*]**

**FUNCTIONS**

**select8.*N***

**PINS**

**select8.*N*.sel** s32 in

The number of the output to set TRUE. All other outputs will be set FALSE

**select8.*N*.out** *M* bit out (*M*=0..7)

Output bits. If enable is set and the sel input is between 0 and 7, then the corresponding output bit will be set true

**PARAMETERS**

**select8.*N*.enable** bit rw (default: *TRUE*)

Set enable to FALSE to cause all outputs to be set FALSE

**LICENSE**

GPL

**NAME**

serport – Hardware driver for the digital I/O bits of the 8250 and 16550 serial port.

**SYNOPSIS**

**loadrt serport io=addr[,addr...]**

The pin numbers refer to the 9-pin serial pinout. Keep in mind that these ports generally use rs232 voltages, not 0/5V signals.

Specify the I/O address of the serial ports using the module parameter **io=addr[,addr...]**. These ports must not be in use by the kernel. To free up the I/O ports after bootup, install setserial and execute a command like:

```
sudo setserial /dev/ttyS0 none
```

but it is best to ensure that the serial port is never used or configured by the Linux kernel by setting a kernel commandline parameter or not loading the serial kernel module if it is a modularized driver.

**FUNCTIONS**

**serport.N.read**

**serport.N.write**

**PINS**

**serport.N.pin-1-in** bit out

Also called DCD (data carrier detect); pin 8 on the 25-pin serial pinout

**serport.N.pin-6-in** bit out

Also called DSR (data set ready); pin 6 on the 25-pin serial pinout

**serport.N.pin-8-in** bit out

Also called CTS (clear to send); pin 5 on the 25-pin serial pinout

**serport.N.pin-9-in** bit out

Also called RI (ring indicator); pin 22 on the 25-pin serial pinout

**serport.N.pin-1-in-not** bit out

Inverted version of pin-1-in

**serport.N.pin-6-in-not** bit out

Inverted version of pin-6-in

**serport.N.pin-8-in-not** bit out

Inverted version of pin-8-in

**serport.N.pin-9-in-not** bit out

Inverted version of pin-9-in

**serport.N.pin-3-out** bit in

Also called TX (transmit data); pin 2 on the 25-pin serial pinout

**serport.N.pin-4-out** bit in

Also called DTR (data terminal ready); pin 20 on the 25-pin serial pinout

**serport.N.pin-7-out** bit in

Also called RTS (request to send); pin 4 on the 25-pin serial pinout

**PARAMETERS**

**serport.N.pin-3-out-invert** bit rw

**serport.N.pin-4-out-invert** bit rw

**serport.N.pin-7-out-invert** bit rw

**serport.N.ioaddr** u32 r

**NAME**

`siggen` – signal generator

**SYNOPSIS**

`loadrt siggen num_chan=num`

**DESCRIPTION**

**siggen** is a signal generator that can be used for testing and other applications that need simple waveforms. It produces sine, cosine, triangle, sawtooth, and square waves of variable frequency, amplitude, and offset, which can be used as inputs to other HAL components.

**siggen** supports a maximum of sixteen channels. The number of channels actually loaded is set by the **num\_chan** argument when the module is loaded. If **numchan** is not specified, the default value is one.

**FUNCTIONS**

**siggen.N.update** (uses floating-point)

Updates output pins for signal generator *N*. Each time it is called it calculates a new sample. It should be called many times faster than the desired signal frequency, to avoid distortion and aliasing.

**PINS**

**siggen.N.frequency** float in

The output frequency for signal generator *N*, in Hertz. The default value is 1.0 Hertz.

**siggen.N.amplitude** float in

The output amplitude for signal generator *N*. If **offset** is zero, the outputs will swing from **-amplitude** to **+amplitude**. The default value is 1.00.

**siggen.N.offset** float in

The output offset for signal generator *N*. This value is added directly to the output signal. The default value is zero.

**siggen.N.square** float out

The square wave output. Positive while **triangle** and **cosine** are ramping upwards, and while **sine** is negative.

**siggen.N.sine** float out

The sine output. Lags **cosine** by 90 degrees.

**siggen.N.cosine** float out

The cosine output. Leads **sine** by 90 degrees.

**siggen.N.triangle** float out

The triangle wave output. Ramps up while **square** is positive, and down while **square** is negative. Reaches its positive and negative peaks at the same time as **cosine**.

**siggen.N.sawtooth** float out

The sawtooth output. Ramps upwards to its positive peak, then instantly drops to its negative peak and starts ramping again. The drop occurs when **triangle** and **cosine** are at their positive peaks, and coincides with the falling edge of **square**.

**PARAMETERS**

None

**NAME**

`sim_encoder` – simulated quadrature encoder

**SYNOPSIS**

`loadrt sim_encoder num_chan=num`

**DESCRIPTION**

**sim\_encoder** can generate quadrature signals as if from an encoder. It also generates an index pulse once per revolution. It is mostly used for testing and simulation, to replace hardware that may not be available. It has a limited maximum frequency, as do all software based pulse generators.

**sim\_encoder** supports a maximum of eight channels. The number of channels actually loaded is set by the **num\_chan** argument when the module is loaded. If **numchan** is not specified, the default value is one.

**FUNCTIONS**

**sim\_encoder.make-pulses** (no floating-point)

Generates the actual quadrature and index pulses. Must be called as frequently as possible, to maximize the count rate and minimize jitter. Operates on all channels at once.

**sim\_encoder.update-speed** (uses floating-point)

Reads the **speed** command and other parameters and converts the data into a form that can be used by **make-pulses**. Changes take effect only when **update-speed** runs. Can (and should) be called less frequently than **make-pulses**. Operates on all channels at once.

**PINS**

**sim\_encoder.N.phase-A** bit out

One of the quadrature outputs.

**sim\_encoder.N.phase-B** bit out

The other quadrature output.

**sim\_encoder.N.phase-Z** bit out

The index pulse.

**sim\_encoder.N.speed** float in

The desired speed of the encoder, in user units per per second. This is divided by **scale**, and the result is used as the encoder speed in revolutions per second.

**PARAMETERS**

**sim\_encoder.N.ppr** u32 rw

The pulses per revolution of the simulated encoder. Note that this is pulses, not counts, per revolution. Each pulse or cycle from the encoder results in four counts, because every edge is counted. Default value is 100 ppr, or 400 counts per revolution.

**sim\_encoder.N.scale** float rw

Scale factor for the **speed** input. The **speed** value is divided by **scale** to get the actual encoder speed in revolutions per second. For example, if **scale** is set to 60, then **speed** is in revolutions per minute (RPM) instead of revolutions per second. The default value is 1.00.

**NAME**

stepgen – software step pulse generation

**SYNOPSIS**

```
loadrt stepgen step_type=type0[,type1...] [ctrl_type=type0[,type1...]]
```

**DESCRIPTION**

**stepgen** is used to control stepper motors. The maximum step rate depends on the CPU and other factors, and is usually in the range of 5KHz to 25KHz. If higher rates are needed, a hardware step generator is a better choice.

**stepgen** has two control modes, which can be selected on a channel by channel basis using **ctrl\_type**. Possible values are "p" for position control, and "v" for velocity control. The default is position control, which drives the motor to a commanded position, subject to acceleration and velocity limits. Velocity control drives the motor at a commanded speed, again subject to accel and velocity limits. Usually, position mode is used for machine axes. Velocity mode is reserved for unusual applications where continuous movement at some speed is desired, instead of movement to a specific position. (Note that velocity mode replaces the former component **freqgen**.)

**stepgen** can control a maximum of eight motors. The number of motors/channels actually loaded depends on the number of *type* values given. The value of each *type* determines the outputs for that channel. Position or velocity mode can be individually selected for each channel. Both control modes support the same 15 possible step types.

By far the most common step type is '0', standard step and direction. Others include up/down, quadrature, and a wide variety of three, four, and five phase patterns that can be used to directly control some types of motor windings. (When used with appropriate buffers of course.)

Some of the stepping types are described below, but for more details (including timing diagrams) see the **stepgen** section of the HAL reference manual.

## type 0: step/dir

Two pins, one for step and one for direction. **make-pulses** must run at least twice for each step (once to set the step pin true, once to clear it). This limits the maximum step rate to half (or less) of the rate that can be reached by types 2-14. The parameters **steplen** and **stepspace** can further lower the maximum step rate. Parameters **dirsetup** and **dirhold** also apply to this step type.

## type 1: up/down

Two pins, one for 'step up' and one for 'step down'. Like type 0, **make-pulses** must run twice per step, which limits the maximum speed.

## type 2: quadrature

Two pins, phase-A and phase-B. For forward motion, A leads B. Can advance by one step every time **make-pulses** runs.

## type 3: three phase, full step

Three pins, phase-A, phase-B, and phase-C. Three steps per full cycle, then repeats. Only one phase is high at a time - for forward motion the pattern is A, then B, then C, then A again.

## type 4: three phase, half step

Three pins, phases A through C. Six steps per full cycle. First A is high alone, then A and B together, then B alone, then B and C together, etc.

## types 5 through 8: four phase, full step

Four pins, phases A through D. Four steps per full cycle. Types 5 and 6 are suitable for use with unipolar steppers, where power is applied to the center tap of each winding, and four open-collector transistors drive the ends. Types 7 and 8 are suitable for bipolar steppers, driven by two H-bridges.

types 9 and 10: four phase, half step

Four pins, phases A through D. Eight steps per full cycle. Type 9 is suitable for unipolar drive, and type 10 for bipolar drive.

types 11 and 12: five phase, full step

Five pins, phases A through E. Five steps per full cycle. See HAL reference manual for the patterns.

types 13 and 14: five phase, half step

Five pins, phases A through E. Ten steps per full cycle. See HAL reference manual for the patterns.

## FUNCTIONS

**stepgen.make-pulses** (no floating-point)

Generates the step pulses, using information computed by **update-freq**. Must be called as frequently as possible, to maximize the attainable step rate and minimize jitter. Operates on all channels at once.

**stepgen.capture-position** (uses floating point)

Captures position feedback value from the high speed code and makes it available on a pin for use elsewhere in the system. Operates on all channels at once.

**stepgen.update-freq** (uses floating point)

Accepts a velocity or position command and converts it into a form usable by **make-pulses** for step generation. Operates on all channels at once.

## PINS

**stepgen.N.counts** s32 out

The current position, in counts, for channel *N*. Updated by **capture-position**.

**stepgen.N.position-fb** float out

The current position, in length units (see parameter **position-scale**). Updated by **capture-position**. The resolution of **position-fb** is much finer than a single step. If you need to see individual steps, use **counts**.

**stepgen.N.enable** bit in

Enables output steps - when false, no steps are generated.

**stepgen.N.velocity-cmd** float in (velocity mode only)

Commanded velocity, in length units per second (see parameter **position-scale**).

**stepgen.N.position-cmd** float in (position mode only)

Commanded position, in length units (see parameter **position-scale**).

**stepgen.N.step** bit out (step type 0 only)

Step pulse output.

**stepgen.N.dir** bit out (step type 0 only)

Direction output: low for forward, high for reverse.

**stepgen.N.up** bit out (step type 1 only)

Count up output, pulses for forward steps.

**stepgen.N.down** bit out (step type 1 only)

Count down output, pulses for reverse steps.

**stepgen.N.phase-A** thru **phase-E** bit out (step types 2-14 only)

Output bits. **phase-A** and **phase-B** are present for step types 2-14, **phase-C** for types 3-14, **phase-D** for types 5-14, and **phase-E** for types 11-14. Behavior depends on selected stepping type.



## PARAMETERS

**stepgen.N.frequency** float ro

The current step rate, in steps per second, for channel *N*.

**stepgen.N.maxaccel** float rw

The acceleration/deceleration limit, in length units per second squared.

**stepgen.N.maxvel** float rw

The maximum allowable velocity, in length units per second. If the requested maximum velocity cannot be reached with the current combination of scaling and **make-pulses** thread period, it will be reset to the highest attainable value.

**stepgen.N.position-scale** float rw

The scaling for position feedback, position command, and velocity command, in steps per length unit.

**stepgen.N.rawcounts** s32 ro

The position in counts, as updated by **make-pulses**. (Note: this is updated more frequently than the **counts** pin.)

**stepgen.N.steplen** u32 rw

The length of the step pulses, in nanoseconds. Measured from rising edge to falling edge.

**stepgen.N.stepspace** u32 rw (step types 0 and 1 only) The minimum

space between step pulses, in nanoseconds. Measured from falling edge to rising edge. The actual time depends on the step rate and can be much longer. If **stepspace** is 0, then **step** can be asserted every period. This can be used in conjunction with **hal\_parport**'s auto-resetting pins to output one step pulse per period. In this mode, **steplen** must be set for one period or less.

**stepgen.N.dirsetup** u32 rw (step type 0 only)

The minimum setup time from direction to step, in nanoseconds periods. Measured from change of direction to rising edge of step.

**stepgen.N.dirhold** u32 rw (step type 0 only)

The minimum hold time of direction after step, in nanoseconds. Measured from falling edge of step to change of direction.

**stepgen.N.dirdelay** u32 rw (step types 1 and higher only)

The minimum time between a forward step and a reverse step, in nanoseconds.

## TIMING

There are five timing parameters which control the output waveform. No step type uses all five, and only those which will be used are exported to HAL. The values of these parameters are in nano-seconds, so no recalculation is needed when changing thread periods. In the timing diagrams that follow, they are identified by the following numbers:

(1) **stepgen.n.steplen**

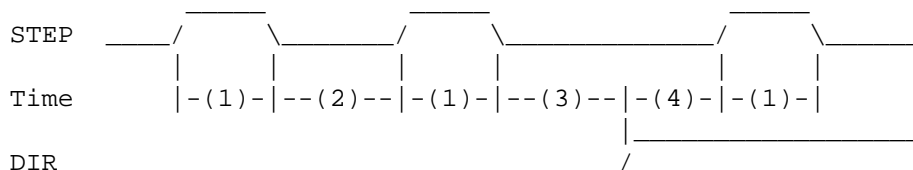
(2) **stepgen.n.stepspace**

(3) **stepgen.n.dirhold**

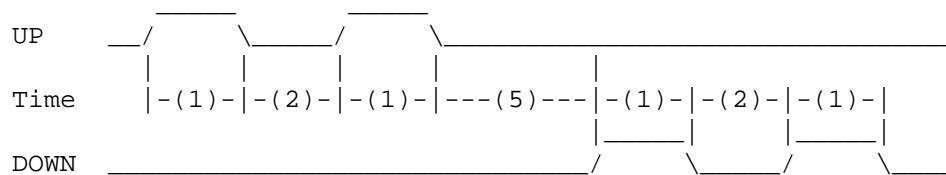
(4) **stepgen.n.dirsetup**

(5) **stepgen.n.dirdelay**

For step type 0, timing parameters 1 thru 4 are used. The following timing diagram shows the output waveforms, and what each parameter adjusts.



For step type 1, timing parameters 1, 2, and 5 are used. The following timing diagram shows the output waveforms, and what each parameter adjusts.



For step types 2 and higher, the exact pattern of the outputs depends on the step type (see the HAL manual for a full listing). The outputs change from one state to another at a minimum interval of **steplen**. When a direction change occurs, the minimum time between the last step in one direction and the first in the other direction is the sum of **steplen** and **dirdelay**.

### SEE ALSO

The HAL User Manual.

**NAME**

steptest – Used by Stepconf to allow testing of acceleration and velocity values for an axis.

**SYNOPSIS**

**loadrt steptest [count=N]**

**FUNCTIONS**

**steptest.N** (uses floating-point)

**PINS**

**steptest.N.jog-minus** bit in

Drive TRUE to jog the axis in its minus direction

**steptest.N.jog-plus** bit in

Drive TRUE to jog the axis in its positive direction

**steptest.N.run** bit in

Drive TRUE to run the axis near its current position\_fb with a trapezoidal velocity profile

**steptest.N.maxvel** float in

Maximum velocity

**steptest.N.maxaccel** float in

Permitted Acceleration

**steptest.N.amplitude** float in

Approximate amplitude of positions to command during 'run'

**steptest.N.dir** s32 in

Direction from central point to test: 0 = both, 1 = positive, 2 = negative

**steptest.N.position-cmd** float out

**steptest.N.position-fb** float in

**steptest.N.running** bit out

**steptest.N.run-target** float out

**steptest.N.run-start** float out

**steptest.N.run-low** float out

**steptest.N.run-high** float out

**PARAMETERS**

**steptest.N.epsilon** float rw (default: .001)

**LICENSE**

GPL

**NAME**

streamer – stream file data into HAL in real time

**SYNOPSIS**

**loadrt streamer depth=depth1[,depth2...] cfg=string1[,string2...]**

**DESCRIPTION**

**streamer** and **halstreamer**(1) are used together to stream data from a file into the HAL in real time. **streamer** is a realtime HAL component that exports HAL pins and creates a FIFO in shared memory. **hal\_streamer** is a user space program that copies data from stdin into the FIFO, so that **streamer** can write it to the HAL pins.

**OPTIONS**

**depth=depth1[,depth2...]**

sets the depth of the user->realtime FIFO that **streamer** creates to receive data from **halstreamer**. Multiple values of *depth* (separated by commas) can be specified if you need more than one FIFO (for example if you want to stream data from two different realtime threads).

**cfg=string1[,string2...]**

defines the set of HAL pins that **streamer** exports and later writes data to. One *string* must be supplied for each FIFO, separated by commas. **streamer** exports one pin for each character in *string*. Legal characters are:

**F, f** (float pin)

**B, b** (bit pin)

**S, s** (s32 pin)

**U, u** (u32 pin)

**FUNCTIONS**

**streamer.N**

One function is created per FIFO, numbered from zero.

**PINS**

**streamer.N.pin.M** output

Data from column *M* of the data in FIFO *N* appears on this pin. The pin type depends on the config string.

**streamer.N.curr-depth** s32 output

Current number of samples in the FIFO. When this reaches zero, new data will no longer be written to the pins.

**streamer.N.empty** bit output

TRUE when the FIFO *N* is empty, FALSE when valid data is available.

**streamer.N.enable** bit input

When TRUE, data from FIFO *N* is written to the HAL pins. When false, no data is transferred. Defaults to TRUE.

**PARAMETERS**

**streamer.N.underruns** s32 read/write

The number of times that **sampler** has tried to write data to the HAL pins but found no fresh data in the FIFO. It increments whenever **empty** is true, and can be reset by the **setp** command.

**SEE ALSO**

**halstreamer(1) sampler(9) halsampler(1)**

**HISTORY****BUGS**

Should an **enable** HAL pin be added, to allow streaming to be turned on and off?

**AUTHOR**

Original version by John Kasunich, as part of the Enhanced Machine Controller (EMC) project.  
Improvements by several other members of the EMC development team.

**REPORTING BUGS**

Report bugs to [jmkasunich](mailto:jmkasunich) AT users DOT sourceforge DOT net

**COPYRIGHT**

Copyright © 2006 John Kasunich.

This is free software; see the source for copying conditions. There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

**NAME**

sum2 – Sum of two inputs (each with a gain) and an offset

**SYNOPSIS**

**loadrt sum2 [count=*N*]**

**FUNCTIONS**

**sum2.*N*** (uses floating-point)

**PINS**

**sum2.*N.in0*** float in

**sum2.*N.in1*** float in

**sum2.*N.out*** float out

$\text{out} = \text{in0} * \text{gain0} + \text{in1} * \text{gain1} + \text{offset}$

**PARAMETERS**

**sum2.*N.gain0*** float rw (default: *1.0*)

**sum2.*N.gain1*** float rw (default: *1.0*)

**sum2.*N.offset*** float rw

**LICENSE**

GPL

**NAME**

supply – set output pins with values from parameters (obsolete)

**SYNOPSIS**

**loadrt supply num\_chan=*num***

**DESCRIPTION**

**supply** was used to allow the inputs of other HAL components to be manipulated for testing purposes. When it was written, the only way to set the value of an input pin was to connect it to a signal and connect that signal to an output pin of some other component, and then let that component write the pin value. **supply** was written to be that "other component". It reads values from parameters (set with the HAL command **setp**) and writes them to output pins.

Since **supply** was written, the **setp** command has been modified to allow it to set unconnected pins as well as parameters. In addition, the **sets** command was added, which can directly set HAL signals, as long as there are no output pins connected to them. Therefore, **supply** is obsolete.

**supply** supports a maximum of eight channels. The number of channels actually loaded is set by the **num\_chan** argument when the module is loaded. If **numchan** is not specified, the default value is one.

**FUNCTIONS**

**supply.N.update** (uses floating-point)  
Updates output pins for channel *N*.

**PINS**

**supply.N.q** bit out  
Output bit, copied from parameter **supply.N.d**.

**supply.N.\_q** bit out  
Output bit, inverted copy of parameter **supply.N.d**.

**supply.N.variable** float out  
Analog output, copied from parameter **supply.N.value**.

**supply.N.\_variable** float out  
Analog output, equal to -1.0 times parameter **supply.N.value**.

**PARAMETERS**

**supply.N.d** bit rw  
Data source for **q** and **\_q** output pins.

**supply.N.value** bit rw  
Data source for **variable** and **\_variable** output pins.

**NAME**

**threads** – creates hard realtime HAL threads

**SYNOPSIS**

**loadrt threads name1=*name* period1=*period* [fp1=<0|1>] [<thread-2-info>] [<thread-3-info>]**

**DESCRIPTION**

**threads** is used to create hard realtime threads which can execute HAL functions at specific intervals. It is not a true HAL component, in that it does not export any functions, pins, or parameters of its own. Once it has created one or more threads, the threads stand alone, and the **threads** component can be unloaded without affecting them. In fact, it can be unloaded and then reloaded to create additional threads, as many times as needed.

**threads** can create up to three realtime threads. Threads must be created in order, from fastest to slowest. Each thread is specified by three arguments. **name1** is used to specify the name of the first thread (thread 1). **period1** is used to specify the period of thread 1 in nanoseconds. Both *name* and *period* are required. The third argument, **fp1** is optional, and is used to specify if thread 1 will be used to execute floating point code. If not specified, it defaults to **1**, which means that the thread will support floating point. Specify **0** to disable floating point support, which saves a small amount of execution time by not saving the FPU context. For additional threads, **name2**, **period2**, **fp2**, **name3**, **period3**, and **fp3** work exactly the same. If more than three threads are needed, unload threads, then reload it to create more threads.

**FUNCTIONS**

None

**PINS**

None

**PARAMETERS**

None

**BUGS**

The existence of **threads** might be considered a bug. Ideally, creation and deletion of threads would be done directly with **halcmd** commands, such as "**newthread name period**", "**delthread name**", or similar. However, limitations in the current HAL implementation require thread creation to take place in kernel space, and loading a component is the most straightforward way to do that.



**NAME**

threadtest

**SYNOPSIS****loadrt threadtest [count=*N*]****FUNCTIONS****threadtest.*N*.increment****threadtest.*N*.reset****PINS****threadtest.*N*.count** u32 out**LICENSE**

GPL

**NAME**

timedelta

**SYNOPSIS**

**loadrt** timedelta [count=*N*]

**FUNCTIONS**

**timedelta.N**

**PINS**

**timedelta.N.out** s32 out

**timedelta.N.err** s32 out (default: 0)

**timedelta.N.min** s32 out (default: 0)

**timedelta.N.max** s32 out (default: 0)

**timedelta.N.jitter** s32 out (default: 0)

**timedelta.N.avg-err** float out (default: 0)

**timedelta.N.reset** bit in

**LICENSE**

GPL

**NAME**

toggle – 'push-on, push-off' from momentary pushbuttons

**SYNOPSIS**

**loadrt toggle [count=*N*]**

**FUNCTIONS**

**toggle.*N***

**PINS**

**toggle.*N.in*** bit in  
button input

**toggle.*N.out*** bit io  
on/off output

**PARAMETERS**

**toggle.*N.debounce*** u32 rw (default: 2)  
debounce delay in periods

**LICENSE**

GPL

**NAME**

tristate\_bit – Place a signal on an I/O pin only when enabled, similar to a tristate buffer in electronics

**SYNOPSIS**

**loadrt tristate\_bit [count=*N*]**

**FUNCTIONS**

**tristate-bit.*N***

If **enable** is TRUE, copy **in** to **out**.

**PINS**

**tristate-bit.*N.in*** bit in

Input value

**tristate-bit.*N.out*** bit io

Output value

**tristate-bit.*N.enable*** bit in

When TRUE, copy in to out

**LICENSE**

GPL

**NAME**

tristate\_float – Place a signal on an I/O pin only when enabled, similar to a tristate buffer in electronics

**SYNOPSIS**

**loadrt** tristate\_float [**count**=*N*]

**FUNCTIONS**

**tristate-float.N** (uses floating-point)

If **enable** is TRUE, copy **in** to **out**.

**PINS**

**tristate-float.N.in** float in

Input value

**tristate-float.N.out** float io

Output value

**tristate-float.N.enable** bit in

When TRUE, copy in to out

**LICENSE**

GPL

**NAME**

updown – Counts up or down, with optional limits and wraparound behavior

**SYNOPSIS**

**loadrt updown [count=*N*]**

**FUNCTIONS**

**updown.*N***

Process inputs and update count if necessary

**PINS**

**updown.*N*.countup** bit in

Increment count when this pin goes from 0 to 1

**updown.*N*.countdown** bit in

Decrement count when this pin goes from 1 to 0

**updown.*N*.count** s32 out

The current count

**PARAMETERS**

**updown.*N*.clamp** bit rw

If TRUE, then clamp the output to the min and max parameters.

**updown.*N*.wrap** bit rw

If TRUE, then wrap around when the count goes above or below the min and max parameters.

Note that wrap implies (and overrides) clamp.

**updown.*N*.max** s32 rw (default: *0x7FFFFFFF*)

If clamp or wrap is set, count will never exceed this number

**updown.*N*.min** s32 rw

If clamp or wrap is set, count will never be less than this number

**LICENSE**

GPL

**NAME**

wcomp – Window comparator

**SYNOPSIS**

**loadrt wcomp [count=*N*]**

**FUNCTIONS**

**wcomp.*N*** (uses floating-point)

**PINS**

**wcomp.*N.out*** bit out

True if in is between min and max

**wcomp.*N.in*** float in

**PARAMETERS**

**wcomp.*N.min*** float rw

**wcomp.*N.max*** float rw

**LICENSE**

GPL

**NAME**

weighted\_sum – convert a group of bits to an integer

**SYNOPSIS**

**loadrt weighted\_sum wsum\_sizes=size[,size,...]**

Creates weighted sum groups each with the given number of input bits (*size*).

**DESCRIPTION**

This component is a "weighted summer": Its output is the offset plus the sum of the weight of each TRUE input bit. The default value for each weight is  $2^n$  where  $n$  is the bit number. This results in a binary to unsigned conversion.

There is a limit of 8 weighted summers and each may have up to 16 input bits.

**FUNCTIONS****process\_wsums**

Read all input values and update all output values.

**PINS**

**wsum.N.bit.M.in** bit in

The  $m$ 'th input of weighted summer  $n$ .

**wsum.N.hold** bit in

When TRUE, the *sum* output does not change. When FALSE, the *sum* output tracks the *bit* inputs according to the weights and offset.

**wsum.N.sum** signed out

The output of the weighted summer

**PARAMETERS**

**wsum.N.bit.M.weight** signed rw

The weight of the  $m$ 'th input of weighted summer  $n$ . The default value is  $2^m$ .

**wsum.N.offset** signed rw

The offset is added to the weights corresponding to all TRUE inputs to give the final sum.



**NAME**

xor2 – Two-input XOR (exclusive OR) gate

**SYNOPSIS**

**loadrt xor2 [count=*N*]**

**FUNCTIONS**

**xor2.*N***

**PINS**

**xor2.*N*.in0** bit in

**xor2.*N*.in1** bit in

**xor2.*N*.out** bit out

**out** is computed from the value of **in0** and **in1** according to the following rule:

**in0=TRUE in1=FALSE**

**in0=FALSE in1=TRUE**

**out=TRUE**

Otherwise,

**out=FALSE**

**LICENSE**

GPL