HAL Manual V2.5, 2016-05-12

Contents

The LinuxCNC Team

This handbook is a work in progress. If you are able to help with writing, editing, or graphic preparation please contact any member of the writing team or join and send an email to emc-users@lists.sourceforge.net.

Part I

Hardware Abstract Layer

Chapter 1

HAL Introduction

HAL stands for Hardware Abstraction Layer. At the highest level, it is simply a way to allow a number of *building blocks* to be loaded and interconnected to assemble a complex system. The *Hardware* part is because HAL was originally designed to make it easier to configure LinuxCNC for a wide variety of hardware devices. Many of the building blocks are drivers for hardware devices. However, HAL can do more than just configure hardware drivers.

1.1 HAL is based on traditional system design techniques

HAL is based on the same principles that are used to design hardware circuits and systems, so it is useful to examine those principles first.

Any system (including a CNC machine), consists of interconnected components. For the CNC machine, those components might be the main controller, servo amps or stepper drives, motors, encoders, limit switches, pushbutton pendants, perhaps a VFD for the spindle drive, a PLC to run a toolchanger, etc. The machine builder must select, mount and wire these pieces together to make a complete system.

1.1.1 Part Selection

The machine builder does not need to worry about how each individual part works. He treats them as black boxes. During the design stage, he decides which parts he is going to use - steppers or servos, which brand of servo amp, what kind of limit switches and how many, etc. The integrator's decisions about which specific components to use is based on what that component does and the specifications supplied by the manufacturer of the device. The size of a motor and the load it must drive will affect the choice of amplifier needed to run it. The choice of amplifier may affect the kinds of feedback needed by the amp and the velocity or position signals that must be sent to the amp from a control.

In the HAL world, the integrator must decide what HAL components are needed. Usually every interface card will require a driver. Additional components may be needed for software generation of step pulses, PLC functionality, and a wide variety of other tasks.

1.1.2 Interconnection Design

The designer of a hardware system not only selects the parts, he also decides how those parts will be interconnected. Each black box has terminals, perhaps only two for a simple switch, or dozens for a servo drive or PLC. They need to be wired together. The motors connect to the servo amps, the limit switches connect to the controller, and so on. As the machine builder works on the design, he creates a large wiring diagram that shows how all the parts should be interconnected.

When using HAL, components are interconnected by signals. The designer must decide which signals are needed, and what they should connect.

1.1.3 Implementation

Once the wiring diagram is complete it is time to build the machine. The pieces need to be acquired and mounted, and then they are interconnected according to the wiring diagram. In a physical system, each interconnection is a piece of wire that needs to be cut and connected to the appropriate terminals.

HAL provides a number of tools to help *build* a HAL system. Some of the tools allow you to *connect* (or disconnect) a single *wire*. Other tools allow you to save a complete list of all the parts, wires, and other information about the system, so that it can be *rebuilt* with a single command.

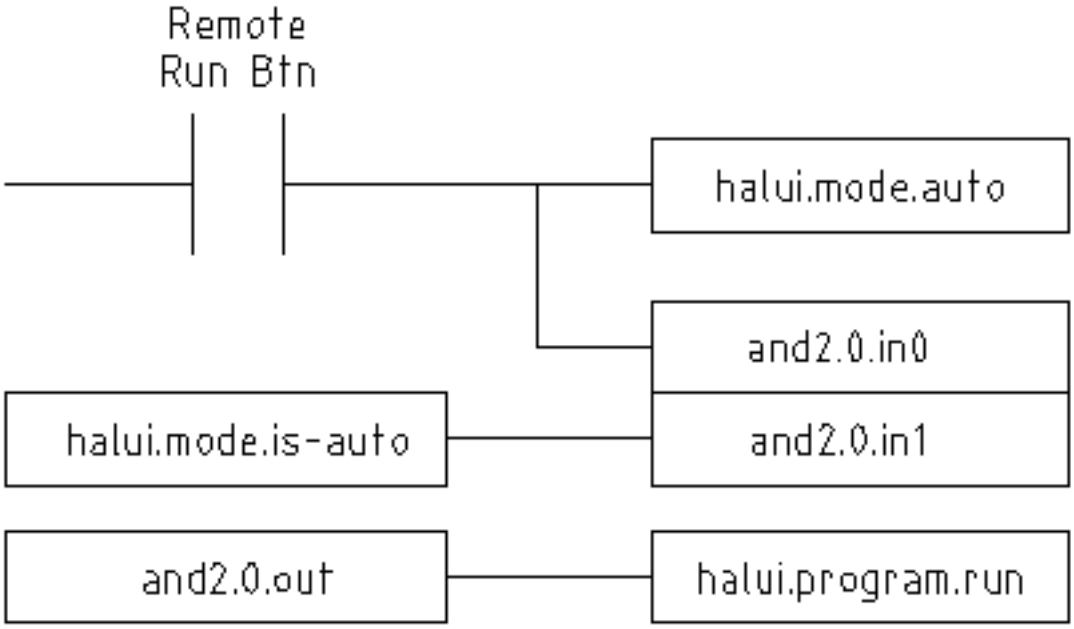
1.1.4 Testing

Very few machines work right the first time. While testing, the builder may use a meter to see whether a limit switch is working or to measure the DC voltage going to a servo motor. He may hook up an oscilloscope to check the tuning of a drive, or to look for electrical noise. He may find a problem that requires the wiring diagram to be changed; perhaps a part needs to be connected differently or replaced with something completely different.

HAL provides the software equivalents of a voltmeter, oscilloscope, signal generator, and other tools needed for testing and tuning a system. The same commands used to build the system can be used to make changes as needed.

1.1.5 Summary

This document is aimed at people who already know how to do this kind of hardware system integration, but who do not know how to connect the hardware to LinuxCNC. See the Remote Start Example section in the HAL UI Examples documentation.



The traditional hardware design as described above ends at the edge of the main control. Outside the control are a bunch of relatively simple boxes, connected together to do whatever is needed. Inside, the control is a big mystery — one huge black box that we hope works.

HAL extends this traditional hardware design method to the inside of the big black box. It makes device drivers and even some internal parts of the controller into smaller black boxes that can be interconnected and even replaced just like the external hardware. It allows the *system wiring diagram* to show part of the internal controller, rather than just a big black box. And most importantly, it allows the integrator to test and modify the controller using the same methods he would use on the rest of the hardware.

Terms like motors, amps, and encoders are familiar to most machine integrators. When we talk about using extra flexible eight conductor shielded cable to connect an encoder to the servo input board in the computer, the reader immediately understands what it is and is led to the question, *what kinds of connectors will I need to make up each end.* The same sort of thinking is essential for the HAL but the specific train of thought may take a bit to get on track. Using HAL words may seem a bit strange at first, but the concept of working from one connection to the next is the same.

This idea of extending the wiring diagram to the inside of the controller is what HAL is all about. If you are comfortable with the idea of interconnecting hardware black boxes, you will probably have little trouble using HAL to interconnect software black boxes.

1.2 HAL Concepts

This section is a glossary that defines key HAL terms but it is a bit different than a traditional glossary because these terms are not arranged in alphabetical order. They are arranged by their relationship or flow in the HAL way of things.

Component
When we talked about hardware design, we referred to the individual pieces as *parts*, *building blocks*, *black boxes*, etc. The HAL equivalent is a *component* or *HAL component*. (This document uses *HAL component* when there is likely to be confusion with other kinds of components, but normally just uses *component*.) A HAL component is a piece of software with well-defined inputs, outputs, and behavior, that can be installed and interconnected as needed.

Parameter
Many hardware components have adjustments that are not connected to any other components but still need to be accessed. For example, servo amps often have trim pots to allow for tuning adjustments, and test points where a meter or scope can be attached to view the tuning results. HAL components also can have such items, which are referred to as *parameters*. There are two types of parameters: Input parameters are equivalent to trim pots - they are values that can be adjusted by the user, and remain fixed once they are set. Output parameters cannot be adjusted by the user - they are equivalent to test points that allow internal signals to be monitored.

Pin
Hardware components have terminals which are used to interconnect them. The HAL equivalent is a *pin* or *HAL pin*. (*HAL pin* is used when needed to avoid confusion.) All HAL pins are named, and the pin names are used when interconnecting them. HAL pins are software entities that exist only inside the computer.

Physical_Pin
Many I/O devices have real physical pins or terminals that connect to external hardware, for example the pins of a parallel port connector. To avoid confusion, these are referred to as *physical pins*. These are the things that *stick out* into the real world.

Signal
In a physical machine, the terminals of real hardware components are interconnected by wires. The HAL equivalent of a wire is a *signal* or *HAL signal*. HAL signals connect HAL pins together as required by the machine builder. HAL signals can be disconnected and reconnected at will (even while the machine is running).

Type
When using real hardware, you would not connect a 24 volt relay output to the +/-10V analog input of a servo amp. HAL pins have the same restrictions, which are based upon their type. Both pins and signals have types, and signals can only be connected to pins of the same type. Currently there are 4 types, as follows:

- bit - a single TRUE/FALSE or ON/OFF value
- float - a 64 bit floating point value, with approximately 53 bits of resolution and over 1000 bits of dynamic range.
- u32 - a 32 bit unsigned integer, legal values are 0 to 4,294,967,295
- s32 - a 32 bit signed integer, legal values are -2,147,483,647 to +2,147,483,647

Function
Real hardware components tend to act immediately on their inputs. For example, if the input voltage to a servo amp changes, the output also changes automatically. However software components cannot act *automatically*. Each component

has specific code that must be executed to do whatever that component is supposed to do. In some cases, that code simply runs as part of the component. However in most cases, especially in realtime components, the code must run in a specific sequence and at specific intervals. For example, inputs should be read before calculations are performed on the input data, and outputs should not be written until the calculations are done. In these cases, the code is made available to the system in the form of one or more *functions*. Each function is a block of code that performs a specific action. The system integrator can use *threads* to schedule a series of functions to be executed in a particular order and at specific time intervals.

Thread
A *thread* is a list of functions that runs at specific intervals as part of a realtime task. When a thread is first created, it has a specific time interval (period), but no functions. Functions can be added to the thread, and will be executed in order every time the thread runs.

As an example, suppose we have a parport component named hal_parport. That component defines one or more HAL pins for each physical pin. The pins are described in that component's doc section: their names, how each pin relates to the physical pin, are they inverted, can you change polarity, etc. But that alone doesn't get the data from the HAL pins to the physical pins. It takes code to do that, and that is where functions come into the picture. The parport component needs at least two functions: one to read the physical input pins and update the HAL pins, the other to take data from the HAL pins and write it to the physical output pins. Both of these functions are part of the parport driver.

1.3 HAL components

Each HAL component is a piece of software with well-defined inputs, outputs, and behavior, that can be installed and interconnected as needed. This section lists some of the available components and a brief description of what each does. Complete details for each component are available later in this document.

1.3.1 External Programs with HAL hooks

motion
A realtime module that accepts NML [1] motion commands and interacts with HAL

iocontrol
A user space module that accepts NML I/O commands and interacts with HAL

classicladder
A PLC using HAL for all I/O

halui
A user space program that interacts with HAL and sends NML commands, it is intended to work as a full User Interface using external knobs & switches

1.3.2 Internal Components

stepgen
Software step pulse generator with position loop. See section Section 9.1

encoder
Software based encoder counter. See section Section 9.3

pid
Proportional/Integral/Derivative control loops. See section Section 9.4

siggen
A sine/cosine/triangle/square wave generator for testing. See section [sec:Siggen]

[1] Neutral Message Language provides a mechanism for handling multiple types of messages in the same buffer as well as simplifying the interface for encoding and decoding buffers in neutral format and the configuration mechanism.

supply
a simple source for testing

blocks
assorted useful components (mux, demux, or, and, integ, ddt, limit, wcomp, etc.)

1.3.3 Hardware Drivers

hal_ax5214h
A driver for the Axiom Measurement & Control AX5241H digital I/O board

hal_m5i20
Mesa Electronics 5i20 board

hal_motenc
Vital Systems MOTENC-100 board

hal_parport
PC parallel port.

hal_ppmc
Pico Systems family of controllers (PPMC, USC and UPC)

hal_stg
Servo To Go card (version 1 & 2)

hal_vti
Vigilant Technologies PCI ENCDAC-4 controller

1.3.4 Tools and Utilities

halcmd
Command line tool for configuration and tuning. See section [sec:Halcmd]

halgui
GUI tool for configuration and tuning (not implemented yet).

halmeter
A handy multimeter for HAL signals. See section [sec:Halmeter].

halscope
A full featured digital storage oscilloscope for HAL signals. See section [sec:Halscope].

Each of these building blocks is described in detail in later chapters.

1.4 Timing Issues In HAL

Unlike the physical wiring models between black boxes that we have said that HAL is based upon, simply connecting two pins with a hal-signal falls far short of the action of the physical case.

True relay logic consists of relays connected together, and when a contact opens or closes, current flows (or stops) immediately. Other coils may change state, etc, and it all just *happens*. But in PLC style ladder logic, it doesn't work that way. Usually in a single pass through the ladder, each rung is evaluated in the order in which it appears, and only once per pass. A perfect example is a single rung ladder, with a NC contact in series with a coil. The contact and coil belong to the same relay.

If this were a conventional relay, as soon as the coil is energized, the contacts begin to open and de-energize it. That means the contacts close again, etc, etc. The relay becomes a buzzer.

With a PLC, if the coil is OFF and the contact is closed when the PLC begins to evaluate the rung, then when it finishes that pass, the coil is ON. The fact that turning on the coil opens the contact feeding it is ignored until the next pass. On the next pass, the PLC sees that the contact is open, and de-energizes the coil. So the relay still switches rapidly between on and off, but at a rate determined by how often the PLC evaluates the rung.

In HAL, the function is the code that evaluates the rung(s). In fact, the HAL-aware realtime version of ClassicLadder exports a function to do exactly that. Meanwhile, a thread is the thing that runs the function at specific time intervals. Just like you can choose to have a PLC evaluate all its rungs every 10 ms, or every second, you can define HAL threads with different periods.

What distinguishes one thread from another is *not* what the thread does - that is determined by which functions are connected to it. The real distinction is simply how often a thread runs.

In LinuxCNC you might have a 50 us thread and a 1 ms thread. These would be created based on BASE_PERIOD and SERVO_PERIOD, the actual times depend on the values in your ini file.

The next step is to decide what each thread needs to do. Some of those decisions are the same in (nearly) any LinuxCNC system—For instance, motion-command-handler is always added to servo-thread.

Other connections would be made by the integrator. These might include hooking the STG driver's encoder read and DAC write functions to the servo thread, or hooking stepgen's function to the base-thread, along with the parport function(s) to write the steps to the port.

Chapter 2

Advanced HAL Tutorial

2.1 Introduction

Configuration moves from theory to device — HAL device that is. For those who have had just a bit of computer programming, this section is the *Hello World* of the HAL. Halrun can be used to create a working system. It is a command line or text file tool for configuration and tuning. The following examples illustrate its setup and operation.

2.1.1 Notation

Terminal commands are shown without the system prompt unless you are running *HAL*. The terminal window is in *Applications/Accessories* from the main Ubuntu menu bar.

Terminal Command Example

```
me@computer:~linuxcnc$ halrun
(will be shown like the following line)
halrun

(the halcmd: prompt will be shown when running HAL)
halcmd: loadrt debounce
halcmd: show pin
```

2.1.2 Tab-completion

Your version of halcmd may include tab-completion. Instead of completing file names as a shell does, it completes commands with HAL identifiers. You will have to type enough letters for a unique match. Try pressing tab after starting a HAL command:

Tab Completion

```
halcmd: loa<TAB>
halcmd: load
halcmd: loadrt
halcmd: loadrt deb<TAB>
halcmd: loadrt debounce
```

2.1.3 The RTAPI environment

RTAPI stands for Real Time Application Programming Interface. Many HAL components work in realtime, and all HAL components store data in shared memory so realtime components can access it. Normal Linux does not support realtime programming

or the type of shared memory that HAL needs. Fortunately there are realtime operating systems (RTOS's) that provide the necessary extensions to Linux. Unfortunately, each RTOS does things a little differently.

To address these differences, the LinuxCNC team came up with RTAPI, which provides a consistent way for programs to talk to the RTOS. If you are a programmer who wants to work on the internals of LinuxCNC, you may want to study *linuxcnc/src/rtapi/rtapi.h* to understand the API. But if you are a normal person all you need to know about RTAPI is that it (and the RTOS) needs to be loaded into the memory of your computer before you do anything with HAL.

2.2 A Simple Example

2.2.1 Loading a component

For this tutorial, we are going to assume that you have successfully installed the Live CD and, if using a RIP [1], invoked the *rip-environment* script to prepare your shell. In that case, all you need to do is load the required RTOS and RTAPI modules into memory. Just run the following command from a terminal window:

Loading HAL

```
cd linuxcnc
halrun
halcmd:
```

With the realtime OS and RTAPI loaded, we can move into the first example. Notice that the prompt is now shown as *halcmd:*. This is because subsequent commands will be interpreted as HAL commands, not shell commands.

For the first example, we will use a HAL component called *siggen*, which is a simple signal generator. A complete description of the *siggen* component can be found in the Siggen section of this Manual. It is a realtime component, implemented as a Linux kernel module. To load *siggen* use the HAL command *loadrt*.

Loading siggen

```
halcmd: loadrt siggen
```

2.2.2 Examining the HAL

Now that the module is loaded, it is time to introduce *halcmd* , the command line tool used to configure the HAL. This tutorial will introduce some halcmd features, for a more complete description try *man halcmd*, or see the reference in Hal Commands section of this document. The first halcmd feature is the *show* command. This command displays information about the current state of the HAL. To show all installed components:

Show Components

```
halcmd: show comp

Loaded HAL Components:
ID      Type  Name                                PID   State
    3   RT    siggen                                    ready
    2   User  halcmd2177                          2177  ready
```

Since *halcmd* itself is a HAL component, it will always show up in the list. The number after halcmd in the component list is the process ID. It is possible to run more than one copy of halcmd at the same time (in different windows for example), so the PID is added to the end of the name to make it unique. The list also shows the *siggen* component that we installed in the previous step. The *RT* under *Type* indicates that *siggen* is a realtime component. The *User* under *Type* indicates it is a user space component.

Next, let's see what pins *siggen* makes available:

Show Pins

[1] Run In Place, when the source files have been downloaded to a user directory.

```
halcmd: show pin

Component Pins:
Owner   Type   Dir         Value  Name
     3  float  IN              1  siggen.0.amplitude
     3  bit    OUT         FALSE  siggen.0.clock
     3  float  OUT             0  siggen.0.cosine
     3  float  IN              1  siggen.0.frequency
     3  float  IN              0  siggen.0.offset
     3  float  OUT             0  siggen.0.sawtooth
     3  float  OUT             0  siggen.0.sine
     3  float  OUT             0  siggen.0.square
     3  float  OUT             0  siggen.0.triangle
```

This command displays all of the pins in the current HAL. A complex system could have dozens or hundreds of pins. But right now there are only nine pins. All eight of these pins are floating point, and carry data out of the *siggen* component. Since we have not yet executed the code contained within the component, some the pins have a value of zero.

The next step is to look at parameters:

Show Parameters

```
halcmd: show param

Parameters:
Owner   Type  Dir         Value   Name
     3  s32   RO              0   siggen.0.update.time
     3  s32   RW              0   siggen.0.update.tmax
```

The *show param* command shows all the parameters in the HAL. Right now each parameter has the default value it was given when the component was loaded. Note the column labeled *Dir*. The parameters labeled *-W* are writable ones that are never changed by the component itself, instead they are meant to be changed by the user to control the component. We will see how to do this later. Parameters labeled *R-* are read only parameters. They can be changed only by the component. Finally, parameter labeled *RW* are read-write parameters. That means that they are changed by the component, but can also be changed by the user. Note: the parameters *siggen.0.update.time* and *siggen.0.update.tmax* are for debugging purposes, and won't be covered in this section.

Most realtime components export one or more functions to actually run the realtime code they contain. Let's see what function(s) *siggen* exported:

Show Functions

```
halcmd: show funct

Exported Functions:
Owner   CodeAddr  Arg       FP   Users  Name
 00003  f801b000  fae820b8  YES      0   siggen.0.update
```

The siggen component exported a single function. It requires floating point. It is not currently linked to any threads, so *users* is zero.

2.2.3 Making realtime code run

To actually run the code contained in the function *siggen.0.update*, we need a realtime thread. The component called *threads* that is used to create a new thread. Lets create a thread called *test-thread* with a period of 1 ms (1,000 us or 1,000,000 ns):

```
halcmd: loadrt threads name1=test-thread period1=1000000
```

Let's see if that worked:

Show Threads

```
halcmd: show thread

Realtime Threads:
     Period  FP     Name               (     Time, Max-Time )
     999855  YES           test-thread (        0,        0 )
```

It did. The period is not exactly 1,000,000 ns because of hardware limitations, but we have a thread that runs at approximately the correct rate, and which can handle floating point functions. The next step is to connect the function to the thread:

Add Function

```
halcmd: addf siggen.0.update test-thread
```

Up till now, we've been using *halcmd* only to look at the HAL. However, this time we used the *addf* (add function) command to actually change something in the HAL. We told *halcmd* to add the function *siggen.0.update* to the thread *test-thread*, and if we look at the thread list again, we see that it succeeded:

```
halcmd: show thread

Realtime Threads:
     Period  FP     Name               (     Time, Max-Time )
     999855  YES           test-thread (        0,        0 )
                  1 siggen.0.update
```

There is one more step needed before the *siggen* component starts generating signals. When the HAL is first started, the thread(s) are not actually running. This is to allow you to completely configure the system before the realtime code starts. Once you are happy with the configuration, you can start the realtime code like this:

```
halcmd: start
```

Now the signal generator is running. Let's look at its output pins:

```
halcmd: show pin

Component Pins:
Owner   Type  Dir         Value  Name
     3  float IN              1  siggen.0.amplitude
     3  bit   OUT         FALSE  siggen.0.clock
     3  float OUT    -0.1640929  siggen.0.cosine
     3  float IN              1  siggen.0.frequency
     3  float IN              0  siggen.0.offset
     3  float OUT    -0.4475303  siggen.0.sawtooth
     3  float OUT     0.9864449  siggen.0.sine
     3  float OUT            -1  siggen.0.square
     3  float OUT    -0.1049393  siggen.0.triangle
```

And let's look again:

```
halcmd: show pin

Component Pins:
Owner   Type  Dir         Value  Name
     3  float IN              1  siggen.0.amplitude
     3  bit   OUT         FALSE  siggen.0.clock
     3  float OUT     0.0507619  siggen.0.cosine
     3  float IN              1  siggen.0.frequency
     3  float IN              0  siggen.0.offset
     3  float OUT     -0.516165  siggen.0.sawtooth
     3  float OUT     0.9987108  siggen.0.sine
     3  float OUT            -1  siggen.0.square
     3  float OUT    0.03232994  siggen.0.triangle
```

We did two *show pin* commands in quick succession, and you can see that the outputs are no longer zero. The sine, cosine, sawtooth, and triangle outputs are changing constantly. The square output is also working, however it simply switches from +1.0 to -1.0 every cycle.

2.2.4 Changing Parameters

The real power of HAL is that you can change things. For example, we can use the *setp* command to set the value of a parameter. Let's change the amplitude of the signal generator from 1.0 to 5.0:

Set Pin

```
halcmd: setp siggen.0.amplitude 5
```

Check the parameters and pins again

```
halcmd: show param

Parameters:
Owner   Type  Dir         Value  Name
     3  s32   RO           1754  siggen.0.update.time
     3  s32   RW          16997  siggen.0.update.tmax

halcmd: show pin

Component Pins:
Owner   Type  Dir         Value  Name
     3  float IN              5  siggen.0.amplitude
     3  bit   OUT         FALSE  siggen.0.clock
     3  float OUT     0.8515425  siggen.0.cosine
     3  float IN              1  siggen.0.frequency
     3  float IN              0  siggen.0.offset
     3  float OUT      2.772382  siggen.0.sawtooth
     3  float OUT     -4.926954  siggen.0.sine
     3  float OUT             5  siggen.0.square
     3  float OUT      0.544764  siggen.0.triangle
```

Note that the value of parameter *siggen.0.amplitude* has changed to 5, and that the pins now have larger values.

2.2.5 Saving the HAL configuration

Most of what we have done with *halcmd* so far has simply been viewing things with the *show* command. However two of the commands actually changed things. As we design more complex systems with HAL, we will use many commands to configure things just the way we want them. HAL has the memory of an elephant, and will retain that configuration until we shut it down. But what about next time? We don't want to manually enter a bunch of commands every time we want to use the system. We can save the configuration of the entire HAL with a single command:

Save

```
halcmd: save
# components
loadrt threads name1=test-thread period1=1000000
loadrt siggen
# pin aliases
# signals
# nets
# parameter values
setp siggen.0.update.tmax 14687
# realtime thread/function links
addf siggen.0.update test-thread
```

The output of the *save* command is a sequence of HAL commands. If you start with an *empty* HAL and run all these commands, you will get the configuration that existed when the *save* command was issued. To save these commands for later use, we simply redirect the output to a file:

Save to a file

```
halcmd: save all saved.hal
```

2.2.6 Exiting halrun

When you're finished with your HAL session type *exit* at the *halcmd:* prompt. This will return you to the system prompt and close down the HAL session. Do not simply close the terminal window without shutting down the HAL session.

Exit HAL

```
halcmd: exit
```

2.2.7 Restoring the HAL configuration

To restore the HAL configuration stored in *saved.hal*, we need to execute all of those HAL commands. To do that, we use *-f <file name>* which reads commands from a file, and *-I* (upper case i) which shows the halcmd prompt after executing the commands:

Run a Saved File

```
halrun -I -f saved.hal
```

Notice that there is not a *start* command in saved.hal. It's necessary to issue it again (or edit saved.hal to add it there).

2.2.8 Removing HAL from memory

If an unexpected shut down of a HAL session occurs you might have to unload HAL before another session can begin. To do this type the following command in a terminal window.

Removing HAL

```
halrun -U
```

2.3 Halmeter

You can build very complex HAL systems without ever using a graphical interface. However there is something satisfying about seeing the result of your work. The first and simplest GUI tool for the HAL is halmeter. It is a very simple program that is the HAL equivalent of the handy Fluke multimeter (or Simpson analog meter for the old timers).

We will use the siggen component again to check out halmeter. If you just finished the previous example, then you can load siggen using the saved file. If not, we can load it just like we did before:

```
halrun
halcmd: loadrt siggen
halcmd: loadrt threads name1=test-thread period1=1000000
halcmd: addf siggen.0.update test-thread
halcmd: start
halcmd: setp siggen.0.amplitude 5
```

At this point we have the siggen component loaded and running. It's time to start halmeter.

Starting Halmeter

```
halcmd: loadusr halmeter
```

The first window you will see is the *Select Item to Probe* window.

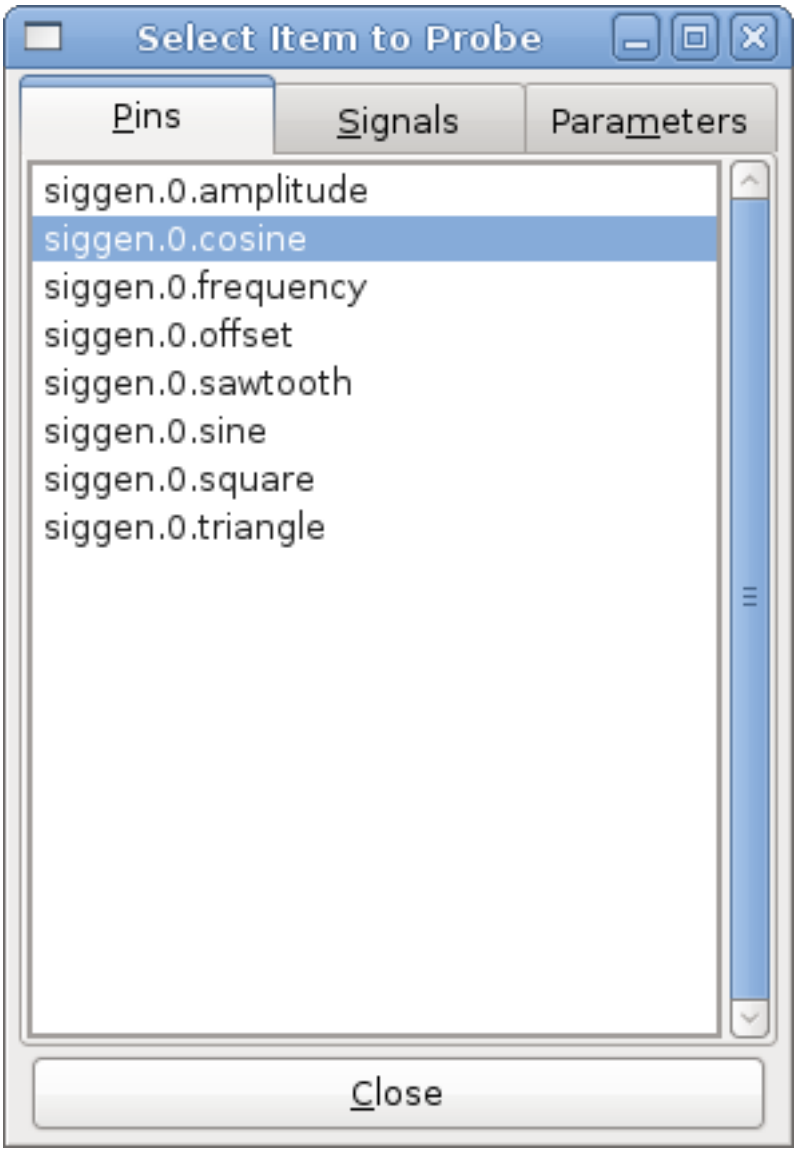


Figure 2.1: Halmeter Select Window

This dialog has three tabs. The first tab displays all of the HAL pins in the system. The second one displays all the signals, and the third displays all the parameters. We would like to look at the pin *siggen.0.cosine* first, so click on it then click the *Close* button. The probe selection dialog will close, and the meter looks something like the following figure.

Figure 2.2: Halmeter

To change what the meter displays press the *Select* button which brings back the *Select Item to Probe* window.

You should see the value changing as siggen generates its cosine wave. Halmeter refreshes its display about 5 times per second.

To shut down halmeter, just click the exit button.

If you want to look at more than one pin, signal, or parameter at a time, you can just start more halmeters. The halmeter window was intentionally made very small so you could have a lot of them on the screen at once.

2.4 Stepgen Example

Up till now we have only loaded one HAL component. But the whole idea behind the HAL is to allow you to load and connect a number of simple components to make up a complex system. The next example will use two components.

Before we can begin building this new example, we want to start with a clean slate. If you just finished one of the previous examples, we need to remove the all components and reload the RTAPI and HAL libraries.

```
halcmd: exit
```

2.4.1 Installing the components

Now we are going to load the step pulse generator component. For a detailed description of this component refer to the stepgen section of the Integrator Manual. In this example we will use the *velocity* control type of stepgen. For now, we can skip the details, and just run the following commands.

```
halrun
halcmd: loadrt stepgen step_type=0,0 ctrl_type=v,v
halcmd: loadrt siggen
halcmd: loadrt threads name1=fast fp1=0 period1=50000 name2=slow period2=1000000
```

The first command loads two step generators, both configured to generate stepping type 0. The second command loads our old friend siggen, and the third one creates two threads, a fast one with a period of 50 microseconds and a slow one with a period of 1 millisecond. The fast thread doesn't support floating point functions.

As before, we can use *halcmd show* to take a look at the HAL. This time we have a lot more pins and parameters than before:

```
halcmd: show pin

Component Pins:
Owner   Type  Dir         Value  Name
     4  float IN              1  siggen.0.amplitude
     4  bit   OUT         FALSE  siggen.0.clock
     4  float OUT             0  siggen.0.cosine
     4  float IN              1  siggen.0.frequency
     4  float IN              0  siggen.0.offset
```

```
     4  float OUT             0  siggen.0.sawtooth
     4  float OUT             0  siggen.0.sine
     4  float OUT             0  siggen.0.square
     4  float OUT             0  siggen.0.triangle
     3  s32   OUT             0  stepgen.0.counts
     3  bit   OUT         FALSE  stepgen.0.dir
     3  bit   IN          FALSE  stepgen.0.enable
     3  float OUT             0  stepgen.0.position-fb
     3  bit   OUT         FALSE  stepgen.0.step
     3  float IN              0  stepgen.0.velocity-cmd
     3  s32   OUT             0  stepgen.1.counts
     3  bit   OUT         FALSE  stepgen.1.dir
     3  bit   IN          FALSE  stepgen.1.enable
     3  float OUT             0  stepgen.1.position-fb
     3  bit   OUT         FALSE  stepgen.1.step
     3  float IN              0  stepgen.1.velocity-cmd

halcmd: show param

Parameters:
Owner   Type  Dir         Value  Name
     4  s32   RO              0  siggen.0.update.time
     4  s32   RW              0  siggen.0.update.tmax
     3  u32   RW     0x00000001  stepgen.0.dirhold
     3  u32   RW     0x00000001  stepgen.0.dirsetup
     3  float RO              0  stepgen.0.frequency
     3  float RW              0  stepgen.0.maxaccel
     3  float RW              0  stepgen.0.maxvel
     3  float RW              1  stepgen.0.position-scale
     3  s32   RO              0  stepgen.0.rawcounts
     3  u32   RW     0x00000001  stepgen.0.steplen
     3  u32   RW     0x00000001  stepgen.0.stepspace
     3  u32   RW     0x00000001  stepgen.1.dirhold
     3  u32   RW     0x00000001  stepgen.1.dirsetup
     3  float RO              0  stepgen.1.frequency
     3  float RW              0  stepgen.1.maxaccel
     3  float RW              0  stepgen.1.maxvel
     3  float RW              1  stepgen.1.position-scale
     3  s32   RO              0  stepgen.1.rawcounts
     3  u32   RW     0x00000001  stepgen.1.steplen
     3  u32   RW     0x00000001  stepgen.1.stepspace
     3  s32   RO              0  stepgen.capture-position.time
     3  s32   RW              0  stepgen.capture-position.tmax
     3  s32   RO              0  stepgen.make-pulses.time
     3  s32   RW              0  stepgen.make-pulses.tmax
     3  s32   RO              0  stepgen.update-freq.time
     3  s32   RW              0  stepgen.update-freq.tmax
```

2.4.2 Connecting pins with signals

What we have is two step pulse generators, and a signal generator. Now it is time to create some HAL signals to connect the two components. We are going to pretend that the two step pulse generators are driving the X and Y axis of a machine. We want to move the table in circles. To do this, we will send a cosine signal to the X axis, and a sine signal to the Y axis. The siggen module creates the sine and cosine, but we need *wires* to connect the modules together. In the HAL, *wires* are called signals. We need to create two of them. We can call them anything we want, for this example they will be *X-vel* and *Y-vel*. The signal *X-vel* is intended to run from the cosine output of the signal generator to the velocity input of the first step pulse generator. The first step is to connect the signal to the signal generator output. To connect a signal to a pin we use the net command.

net command

```
halcmd: net X-vel <= siggen.0.cosine
```

To see the effect of the *net* command, we show the signals again.

```
halcmd: show sig

Signals:
Type          Value  Name     (linked to)
float             0  X-vel <== siggen.0.cosine
```

When a signal is connected to one or more pins, the show command lists the pins immediately following the signal name. The *arrow* shows the direction of data flow - in this case, data flows from pin *siggen.0.cosine* to signal *X-vel*. Now let's connect the *X-vel* to the velocity input of a step pulse generator.

```
halcmd: net X-vel => stepgen.0.velocity-cmd
```

We can also connect up the Y axis signal *Y-vel*. It is intended to run from the sine output of the signal generator to the input of the second step pulse generator. The following command accomplishes in one line what two *net* commands accomplished for *X-vel*.

```
halcmd: net Y-vel siggen.0.sine => stepgen.1.velocity-cmd
```

Now let's take a final look at the signals and the pins connected to them.

```
halcmd: show sig

Signals:
Type          Value  Name     (linked to)
float             0  X-vel <== siggen.0.cosine
                            ==> stepgen.0.velocity-cmd
float             0  Y-vel <== siggen.0.sine
                            ==> stepgen.1.velocity-cmd
```

The *show sig* command makes it clear exactly how data flows through the HAL. For example, the *X-vel* signal comes from pin *siggen.0.cosine*, and goes to pin *stepgen.0.velocity-cmd*.

2.4.3 Setting up realtime execution - threads and functions

Thinking about data flowing through *wires* makes pins and signals fairly easy to understand. Threads and functions are a little more difficult. Functions contain the computer instructions that actually get things done. Thread are the method used to make those instructions run when they are needed. First let's look at the functions available to us.

```
halcmd: show funct

Exported Functions:
Owner   CodeAddr  Arg       FP   Users  Name
 00004  f9992000  fc731278  YES      0   siggen.0.update
 00003  f998b20f  fc7310b8  YES      0   stepgen.capture-position
 00003  f998b000  fc7310b8  NO       0   stepgen.make-pulses
 00003  f998b307  fc7310b8  YES      0   stepgen.update-freq
```

In general, you will have to refer to the documentation for each component to see what its functions do. In this case, the function *siggen.0.update* is used to update the outputs of the signal generator. Every time it is executed, it calculates the values of the sine, cosine, triangle, and square outputs. To make smooth signals, it needs to run at specific intervals.

The other three functions are related to the step pulse generators.

The first one, *stepgen.capture_position*, is used for position feedback. It captures the value of an internal counter that counts the step pulses as they are generated. Assuming no missed steps, this counter indicates the position of the motor.

The main function for the step pulse generator is *stepgen.make_pulses*. Every time *make_pulses* runs it decides if it is time to take a step, and if so sets the outputs accordingly. For smooth step pulses, it should run as frequently as possible. Because it needs to run so fast, *make_pulses* is highly optimized and performs only a few calculations. Unlike the others, it does not need floating point math.

The last function, *stepgen.update-freq*, is responsible for doing scaling and some other calculations that need to be performed only when the frequency command changes.

What this means for our example is that we want to run *siggen.0.update* at a moderate rate to calculate the sine and cosine values. Immediately after we run *siggen.0.update*, we want to run *stepgen.update_freq* to load the new values into the step pulse generator. Finally we need to run *stepgen.make_pulses* as fast as possible for smooth pulses. Because we don't use position feedback, we don't need to run *stepgen.capture_position* at all.

We run functions by adding them to threads. Each thread runs at a specific rate. Let's see what threads we have available.

```
halcmd: show thread

Realtime Threads:
     Period  FP     Name               (     Time, Max-Time )
     996980  YES                slow (        0,        0 )
      49849  NO                 fast (        0,        0 )
```

The two threads were created when we loaded *threads*. The first one, *slow*, runs every millisecond, and is capable of running floating point functions. We will use it for *siggen.0.update* and *stepgen.update_freq*. The second thread is *fast*, which runs every 50 microseconds, and does not support floating point. We will use it for *stepgen.make_pulses*. To connect the functions to the proper thread, we use the *addf* command. We specify the function first, followed by the thread.

```
halcmd: addf siggen.0.update slow
halcmd: addf stepgen.update-freq slow
halcmd: addf stepgen.make-pulses fast
```

After we give these commands, we can run the *show thread* command again to see what happened.

```
halcmd: show thread

Realtime Threads:
     Period  FP     Name               (     Time, Max-Time )
     996980  YES                slow (        0,        0 )
                  1 siggen.0.update
                  2 stepgen.update-freq
      49849  NO                 fast (        0,        0 )
                  1 stepgen.make-pulses
```

Now each thread is followed by the names of the functions, in the order in which the functions will run.

2.4.4 Setting parameters

We are almost ready to start our HAL system. However we still need to adjust a few parameters. By default, the siggen component generates signals that swing from +1 to -1. For our example that is fine, we want the table speed to vary from +1 to -1 inches per second. However the scaling of the step pulse generator isn't quite right. By default, it generates an output frequency of 1 step per second with an input of 1.000. It is unlikely that one step per second will give us one inch per second of table movement. Let's assume instead that we have a 5 turn per inch leadscrew, connected to a 200 step per rev stepper with 10x microstepping. So it takes 2000 steps for one revolution of the screw, and 5 revolutions to travel one inch. that means the overall scaling is 10000 steps per inch. We need to multiply the velocity input to the step pulse generator by 10000 to get the proper output. That is exactly what the parameter *stepgen.n.velocity-scale* is for. In this case, both the X and Y axis have the same scaling, so we set the scaling parameters for both to 10000.

```
halcmd: setp stepgen.0.position-scale 10000
halcmd: setp stepgen.1.position-scale 10000
halcmd: setp stepgen.0.enable 1
halcmd: setp stepgen.1.enable 1
```

This velocity scaling means that when the pin *stepgen.0.velocity-cmd* is 1.000, the step generator will generate 10000 pulses per second (10KHz). With the motor and leadscrew described above, that will result in the axis moving at exactly 1.000 inches per second. This illustrates a key HAL concept - things like scaling are done at the lowest possible level, in this case in the step pulse generator. The internal signal *X-vel* is the velocity of the table in inches per second, and other components such as *siggen* don't know (or care) about the scaling at all. If we changed the leadscrew, or motor, we would change only the scaling parameter of the step pulse generator.

2.4.5 Run it!

We now have everything configured and are ready to start it up. Just like in the first example, we use the *start* command.

```
halcmd: start
```

Although nothing appears to happen, inside the computer the step pulse generator is cranking out step pulses, varying from 10KHz forward to 10KHz reverse and back again every second. Later in this tutorial we'll see how to bring those internal signals out to run motors in the real world, but first we want to look at them and see what is happening.

2.5 Halscope

The previous example generates some very interesting signals. But much of what happens is far too fast to see with halmeter. To take a closer look at what is going on inside the HAL, we want an oscilloscope. Fortunately HAL has one, called halscope.

Halscope has two parts - a realtime part that is loaded as a kernel module, and a user part that supplies the GUI and display. However, you don't need to worry about this, because the userspace portion will automatically request that the realtime part be loaded. Also notice the first time you run halscope in a directory it gives a benign message that the file *autosave.halscope* could not be opened.

Starting Halscope

```
halcmd: loadusr halscope
halcmd: halscope: config file 'autosave.halscope' could not be opened
```

The scope GUI window will open, immediately followed by a *Realtime function not linked* dialog that looks like the following figure.

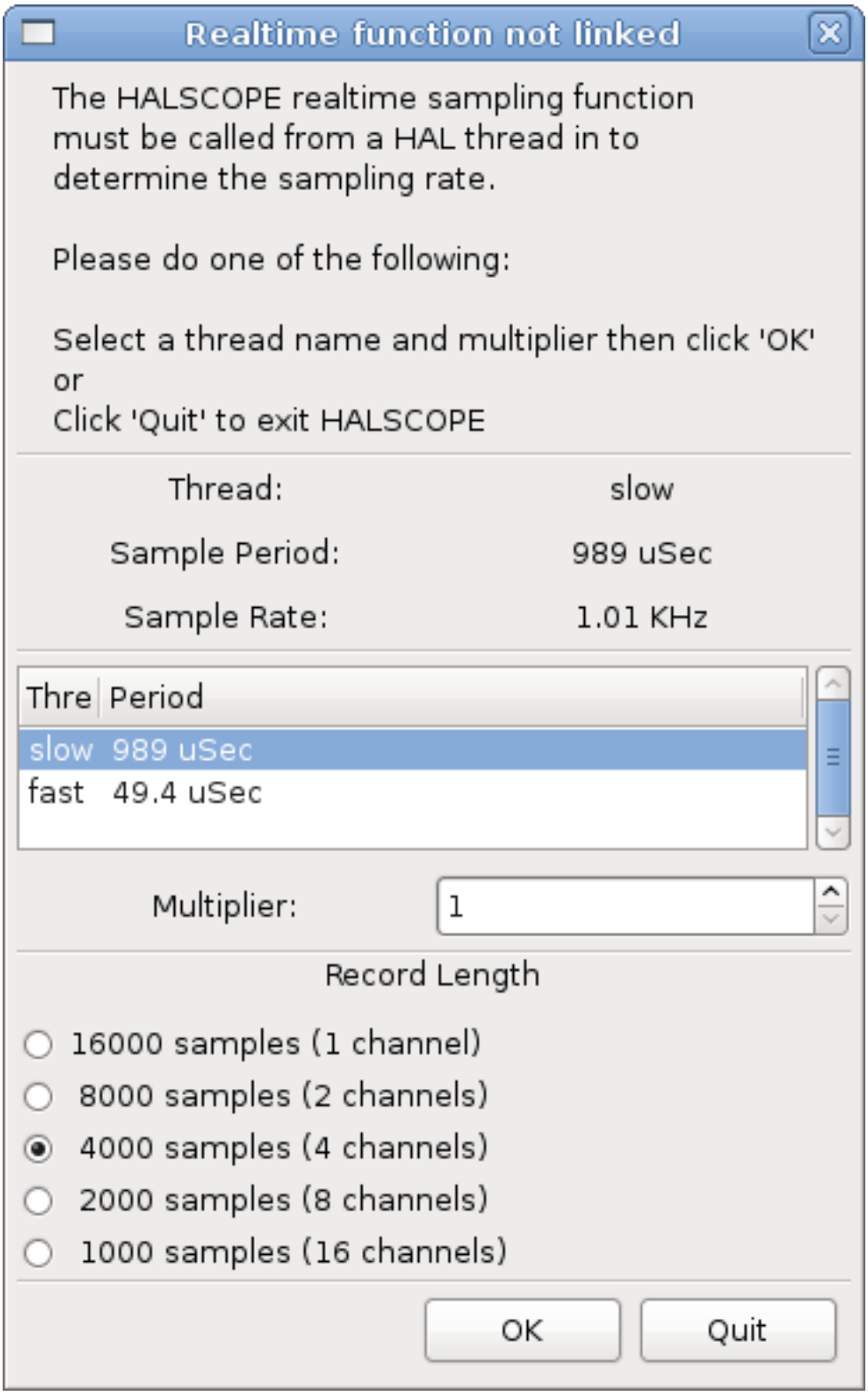


Figure 2.3: Realtime function not linked dialog

This dialog is where you set the sampling rate for the oscilloscope. For now we want to sample once per millisecond, so click on the 989 us thread *slow* and leave the multiplier at 1. We will also leave the record length at 4000 samples, so that we can use up to four channels at one time. When you select a thread and then click *OK*, the dialog disappears, and the scope window looks

something like the following figure.

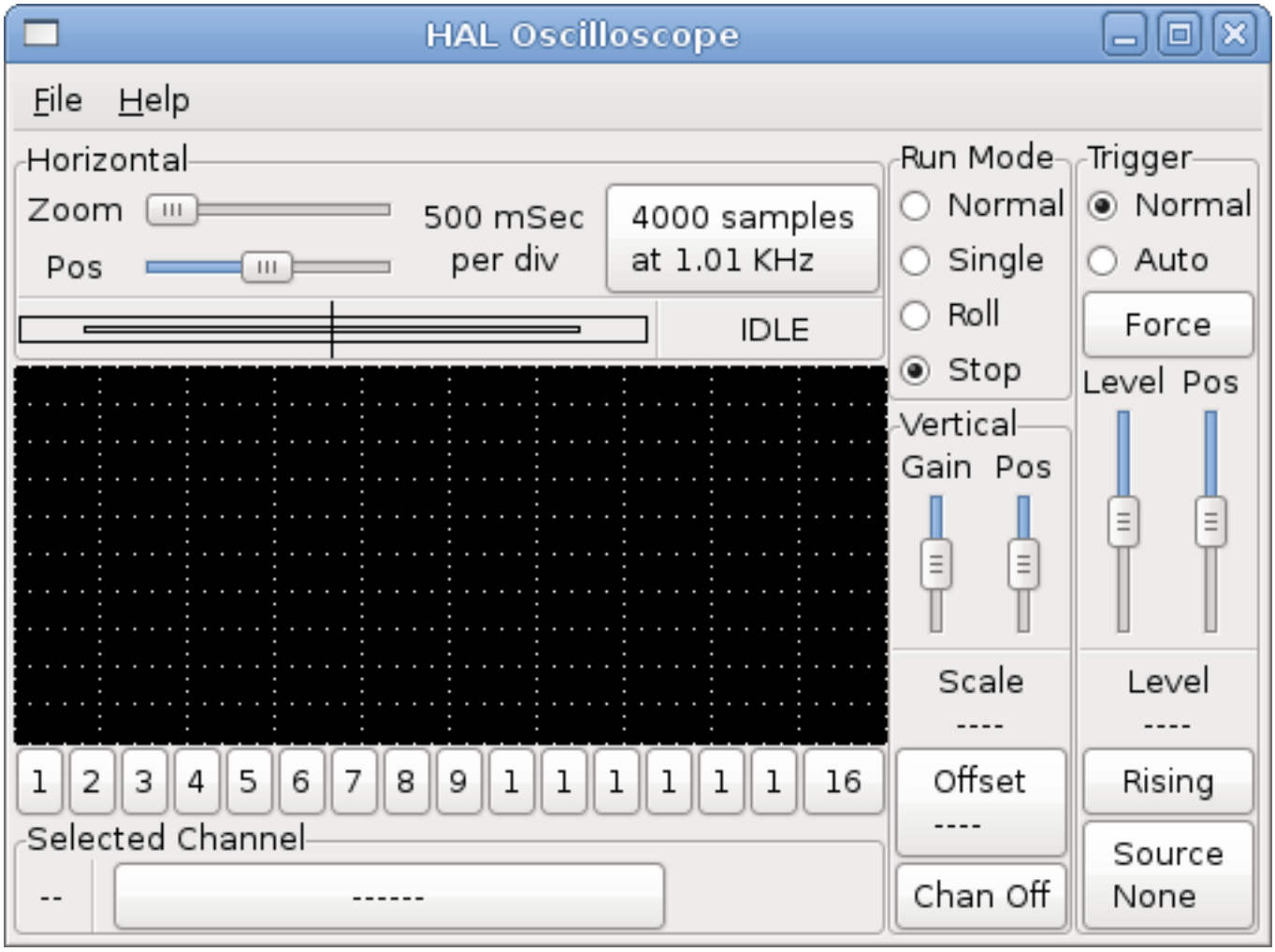


Figure 2.4: Initial scope window

2.5.1 Hooking up the scope probes

At this point, Halscope is ready to use. We have already selected a sample rate and record length, so the next step is to decide what to look at. This is equivalent to hooking *virtual scope probes* to the HAL. Halscope has 16 channels, but the number you can use at any one time depends on the record length - more channels means shorter records, since the memory available for the record is fixed at approximately 16,000 samples.

The channel buttons run across the bottom of the halscope screen. Click button *1*, and you will see the *Select Channel Source* dialog as shown in the following figure. This dialog is very similar to the one used by Halmeter. We would like to look at the signals we defined earlier, so we click on the *Signals* tab, and the dialog displays all of the signals in the HAL (only two for this example).

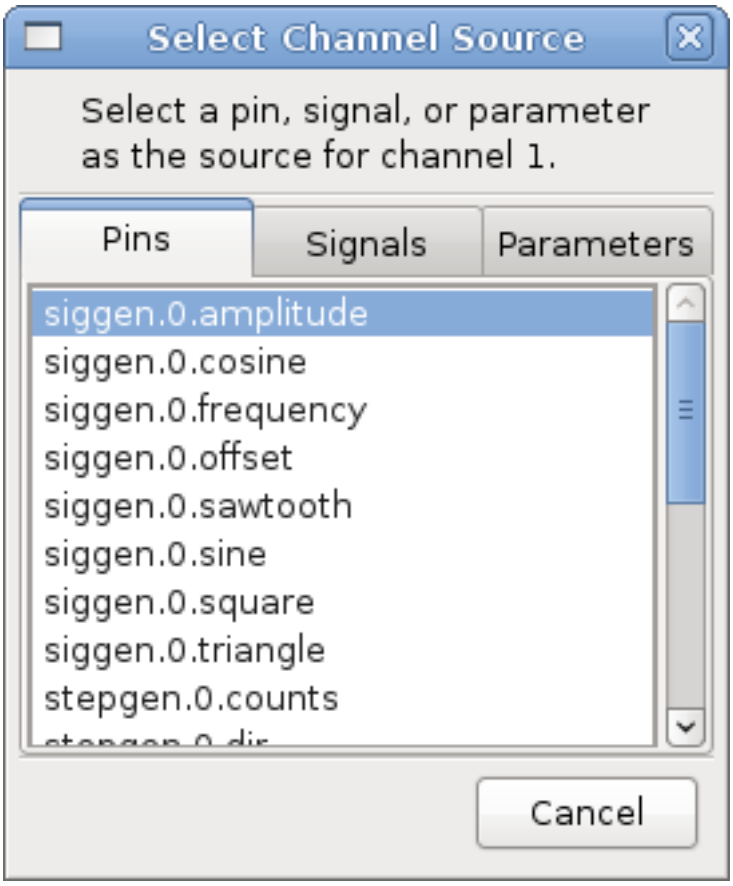


Figure 2.5: Select Channel Source

To choose a signal, just click on it. In this case, we want channel 1 to display the signal *X-vel*. Click on the Signals tab then click on *X-vel* and the dialog closes and the channel is now selected.

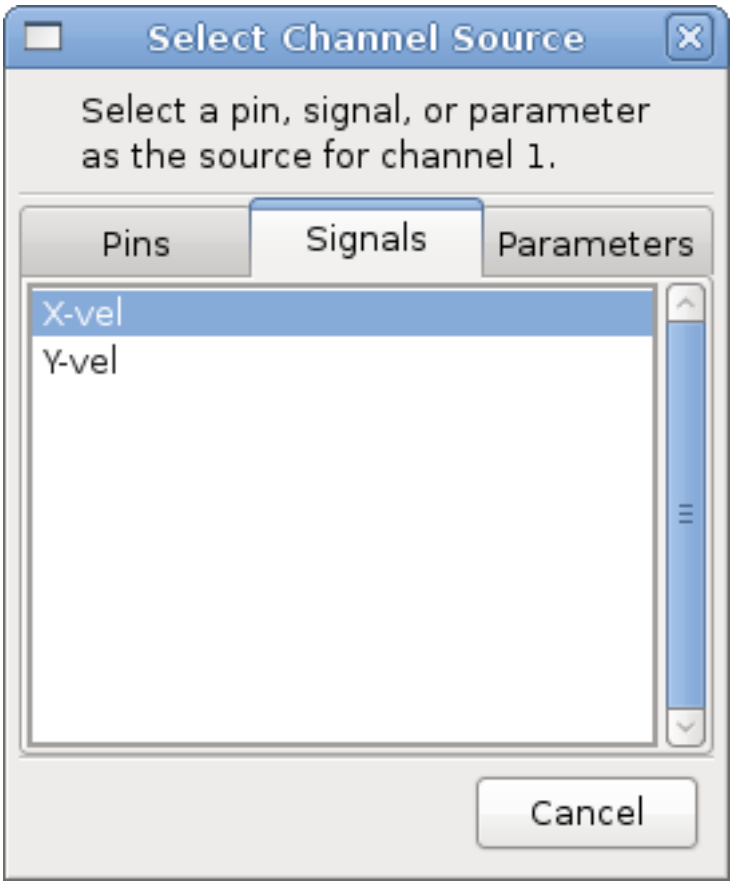


Figure 2.6: Select Signal

The channel 1 button is pressed in, and channel number 1 and the name *X-vel* appear below the row of buttons. That display always indicates the selected channel - you can have many channels on the screen, but the selected one is highlighted, and the various controls like vertical position and scale always work on the selected one.

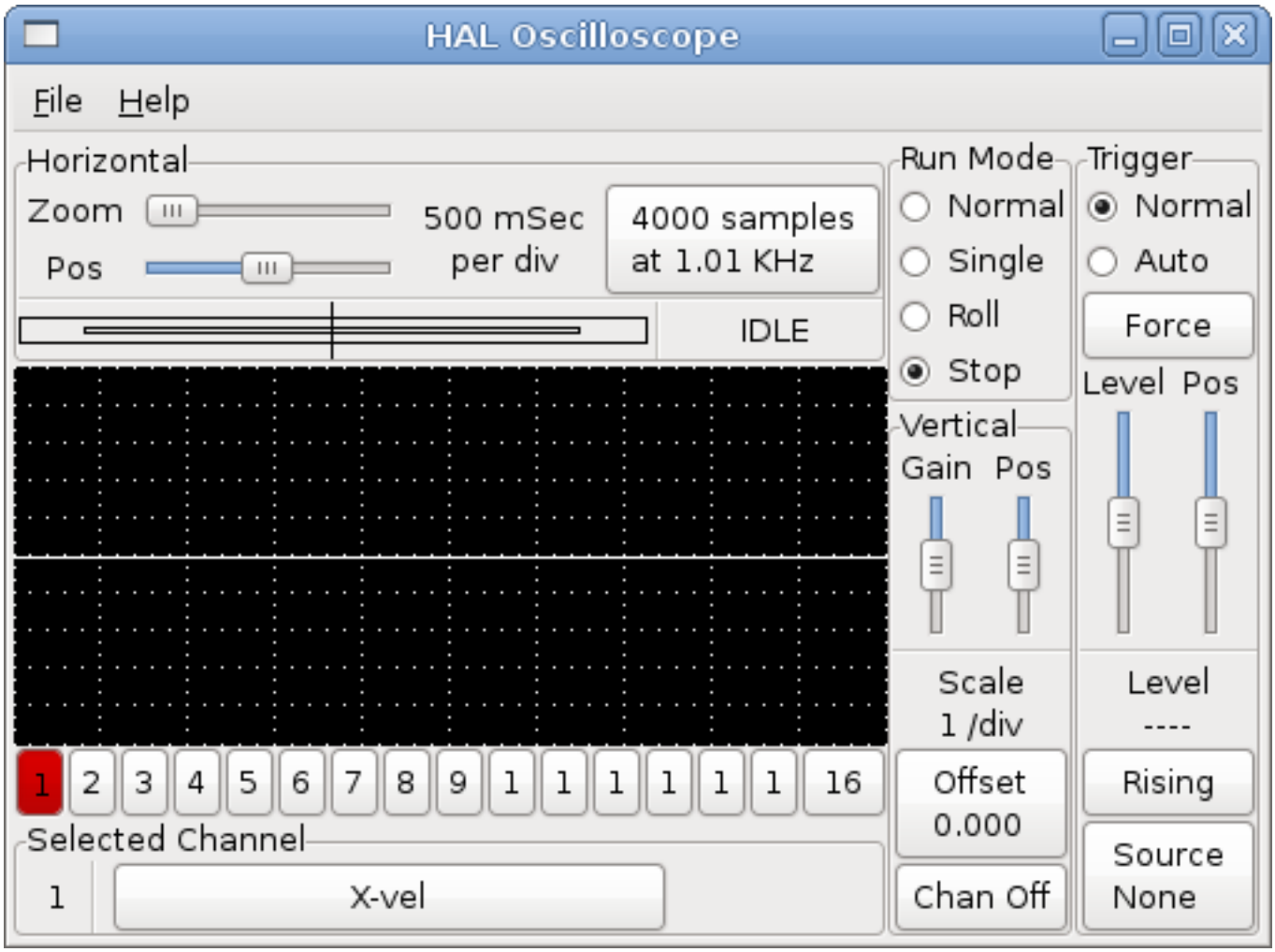


Figure 2.7: Halscope

To add a signal to channel 2, click the *2* button. When the dialog pops up, click the *Signals* tab, then click on *Y-vel*. We also want to look at the square and triangle wave outputs. There are no signals connected to those pins, so we use the *Pins* tab instead. For channel 3, select *siggen.0.triangle* and for channel 4, select *siggen.0.square*.

2.5.2 Capturing our first waveforms

Now that we have several probes hooked to the HAL, it's time to capture some waveforms. To start the scope, click the *Normal* button in the *Run Mode* section of the screen (upper right). Since we have a 4000 sample record length, and are acquiring 1000 samples per second, it will take halscope about 2 seconds to fill half of its buffer. During that time a progress bar just above the main screen will show the buffer filling. Once the buffer is half full, the scope waits for a trigger. Since we haven't configured one yet, it will wait forever. To manually trigger it, click the *Force* button in the *Trigger* section at the top right. You should see the remainder of the buffer fill, then the screen will display the captured waveforms. The result will look something like the following figure.

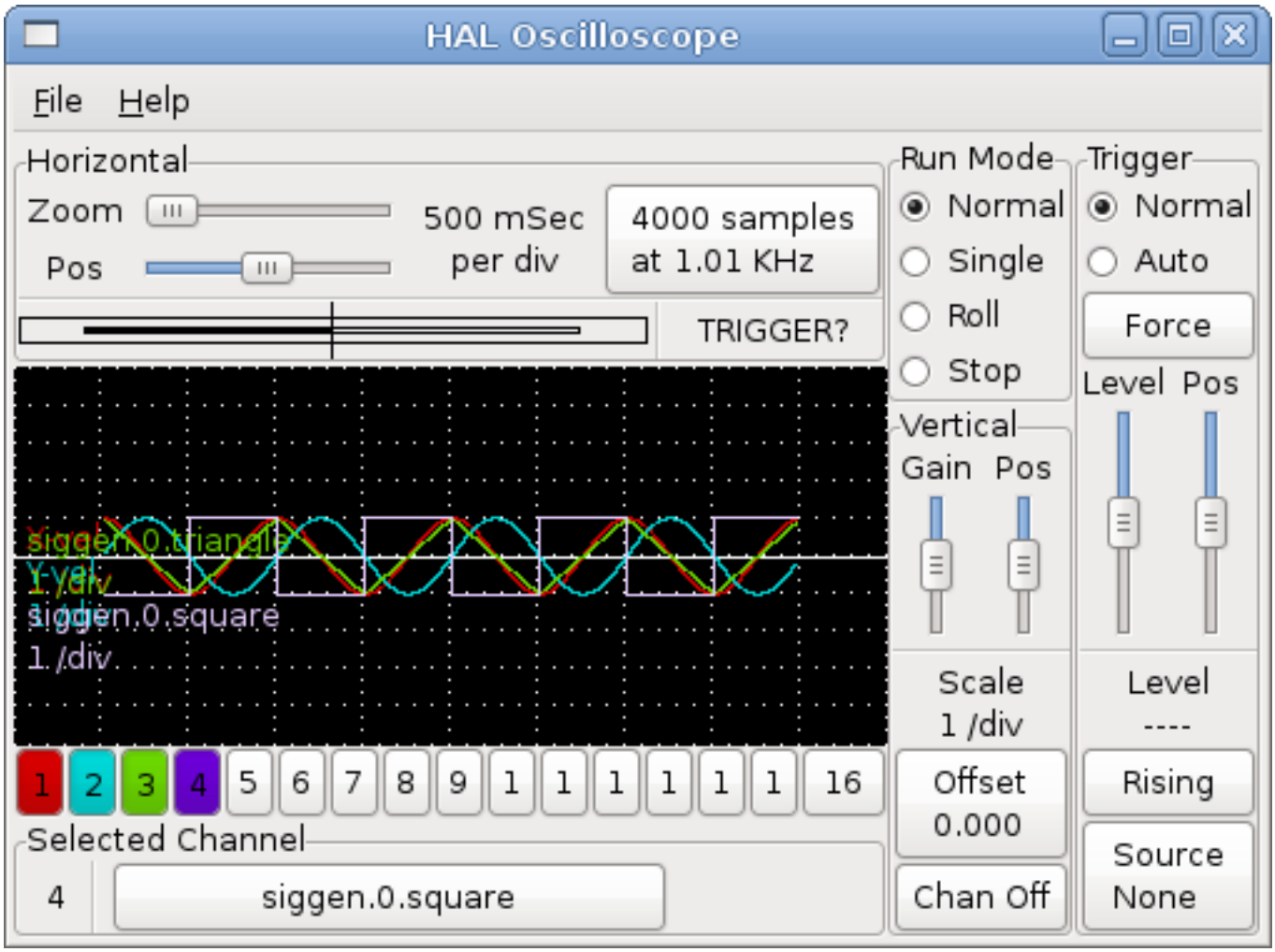


Figure 2.8: Captured Waveforms

The *Selected Channel* box at the bottom tells you that the purple trace is the currently selected one, channel 4, which is displaying the value of the pin *siggen.0.square*. Try clicking channel buttons 1 through 3 to highlight the other three traces.

2.5.3 Vertical Adjustments

The traces are rather hard to distinguish since all four are on top of each other. To fix this, we use the *Vertical* controls in the box to the right of the screen. These controls act on the currently selected channel. When adjusting the gain, notice that it covers a huge range - unlike a real scope, this one can display signals ranging from very tiny (pico-units) to very large (Tera-units). The position control moves the displayed trace up and down over the height of the screen only. For larger adjustments the offset button should be used.

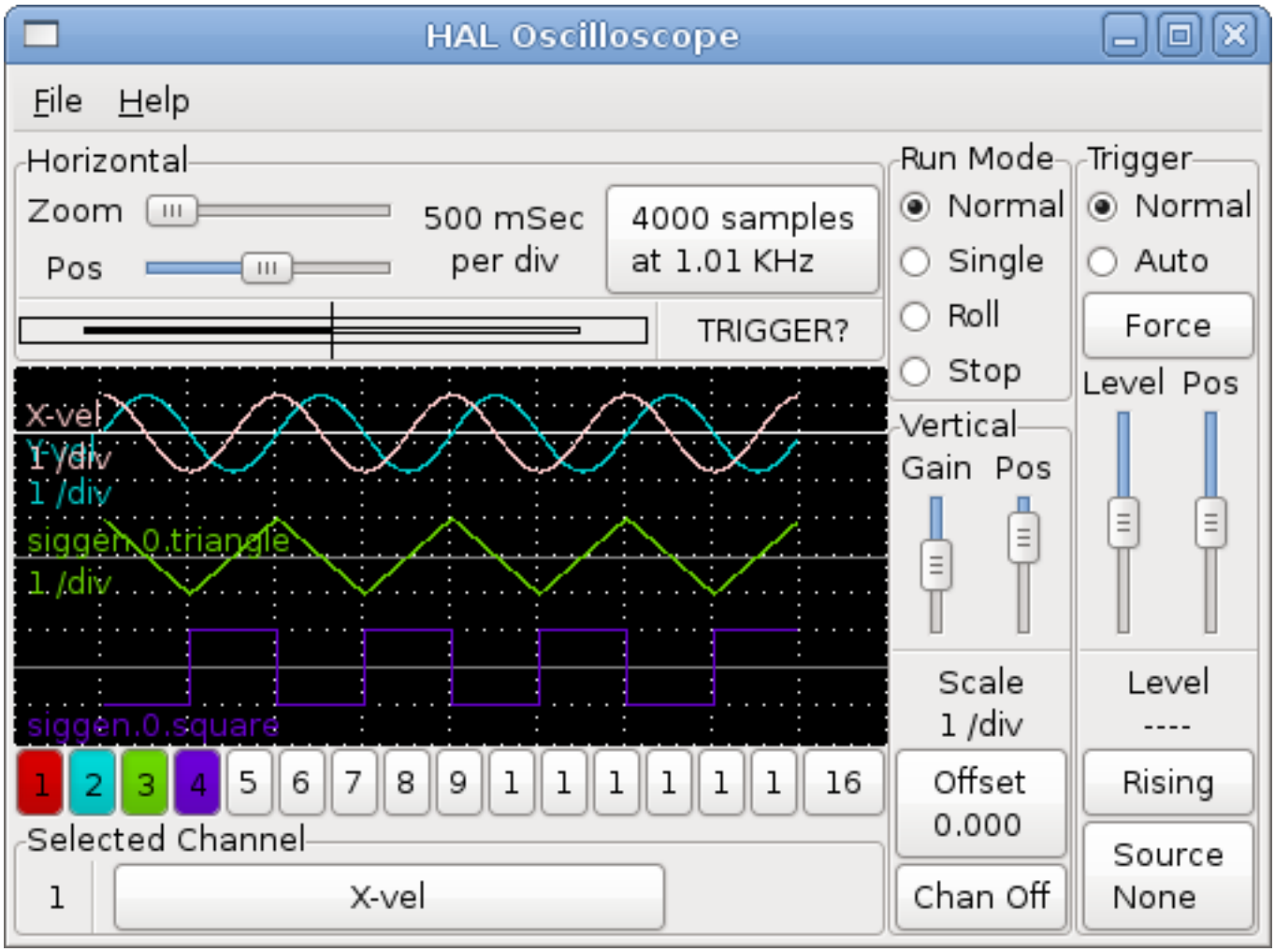


Figure 2.9: Vertical Adjustment

2.5.4 Triggering

Using the *Force* button is a rather unsatisfying way to trigger the scope. To set up real triggering, click on the *Source* button at the bottom right. It will pop up the *Trigger Source* dialog, which is simply a list of all the probes that are currently connected. Select a probe to use for triggering by clicking on it. For this example we will use channel 3, the triangle wave as shown in the following figure.

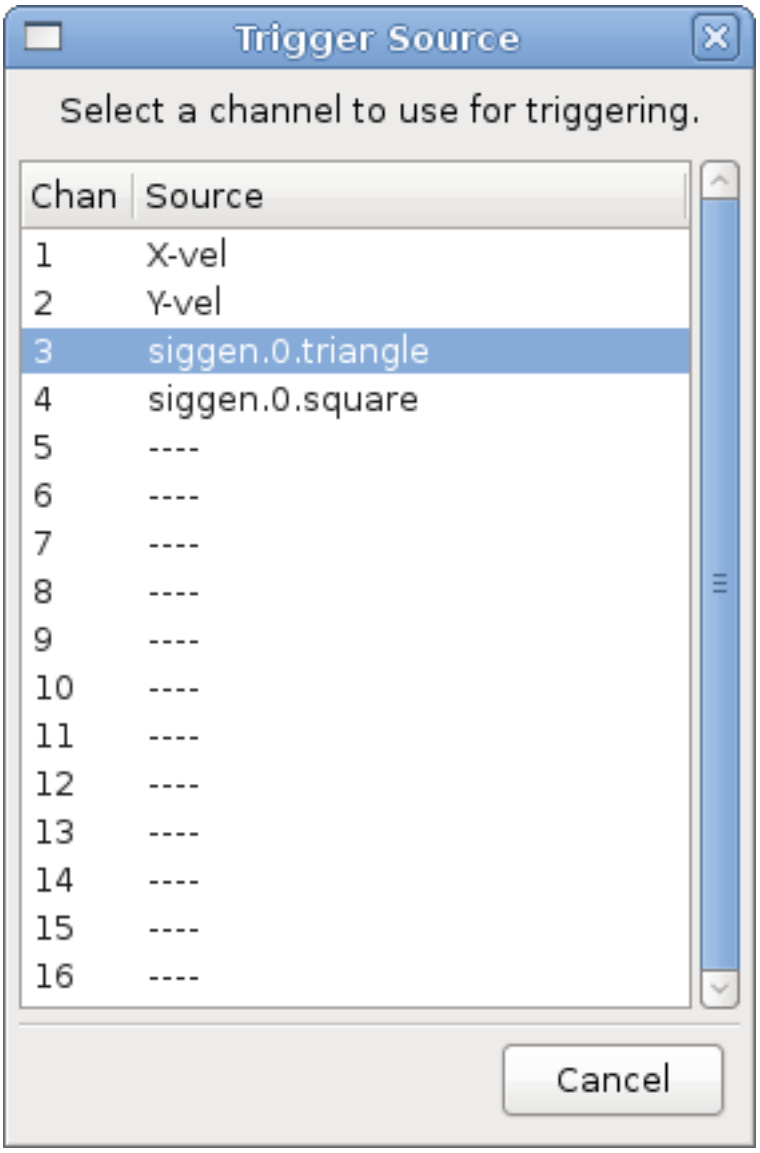


Figure 2.10: Trigger Source Dialog

After setting the trigger source, you can adjust the trigger level and trigger position using the sliders in the *Trigger* box along the right edge. The level can be adjusted from the top to the bottom of the screen, and is displayed below the sliders. The position is the location of the trigger point within the overall record. With the slider all the way down, the trigger point is at the end of the record, and halscope displays what happened before the trigger point. When the slider is all the way up, the trigger point is at the beginning of the record, displaying what happened after it was triggered. The trigger point is visible as a vertical line in the progress box above the screen. The trigger polarity can be changed by clicking the button just below the trigger level display.

Now that we have adjusted the vertical controls and triggering, the scope display looks something like the following figure.

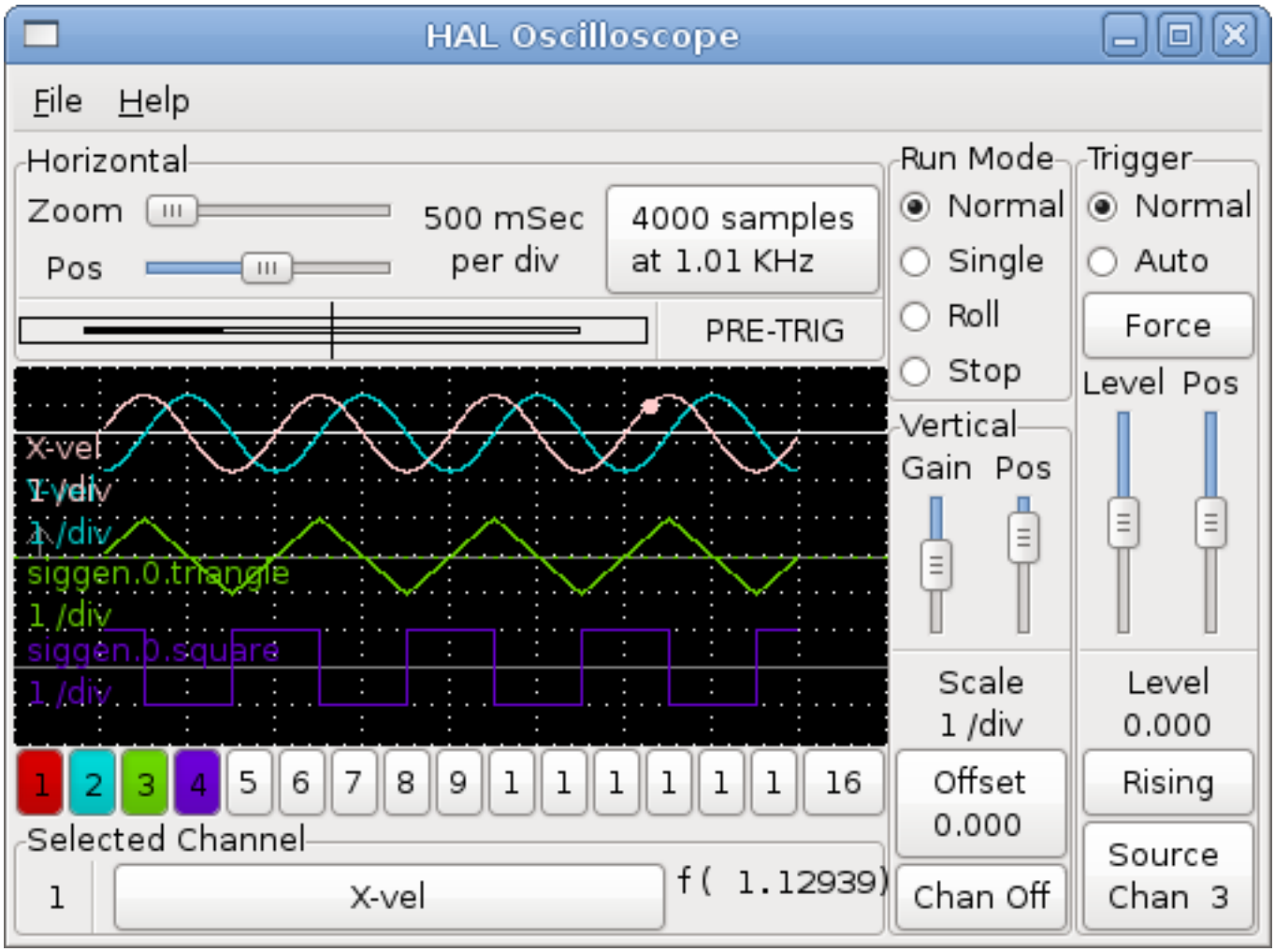


Figure 2.11: Waveforms with Triggering

2.5.5 Horizontal Adjustments

To look closely at part of a waveform, you can use the zoom slider at the top of the screen to expand the waveforms horizontally, and the position slider to determine which part of the zoomed waveform is visible. However, sometimes simply expanding the waveforms isn't enough and you need to increase the sampling rate. For example, we would like to look at the actual step pulses that are being generated in our example. Since the step pulses may be only 50 us long, sampling at 1KHz isn't fast enough. To change the sample rate, click on the button that displays the number of samples and sample rate to bring up the *Select Sample Rate* dialog, figure . For this example, we will click on the 50 us thread, *fast*, which gives us a sample rate of about 20KHz. Now instead of displaying about 4 seconds worth of data, one record is 4000 samples at 20KHz, or about 0.20 seconds.

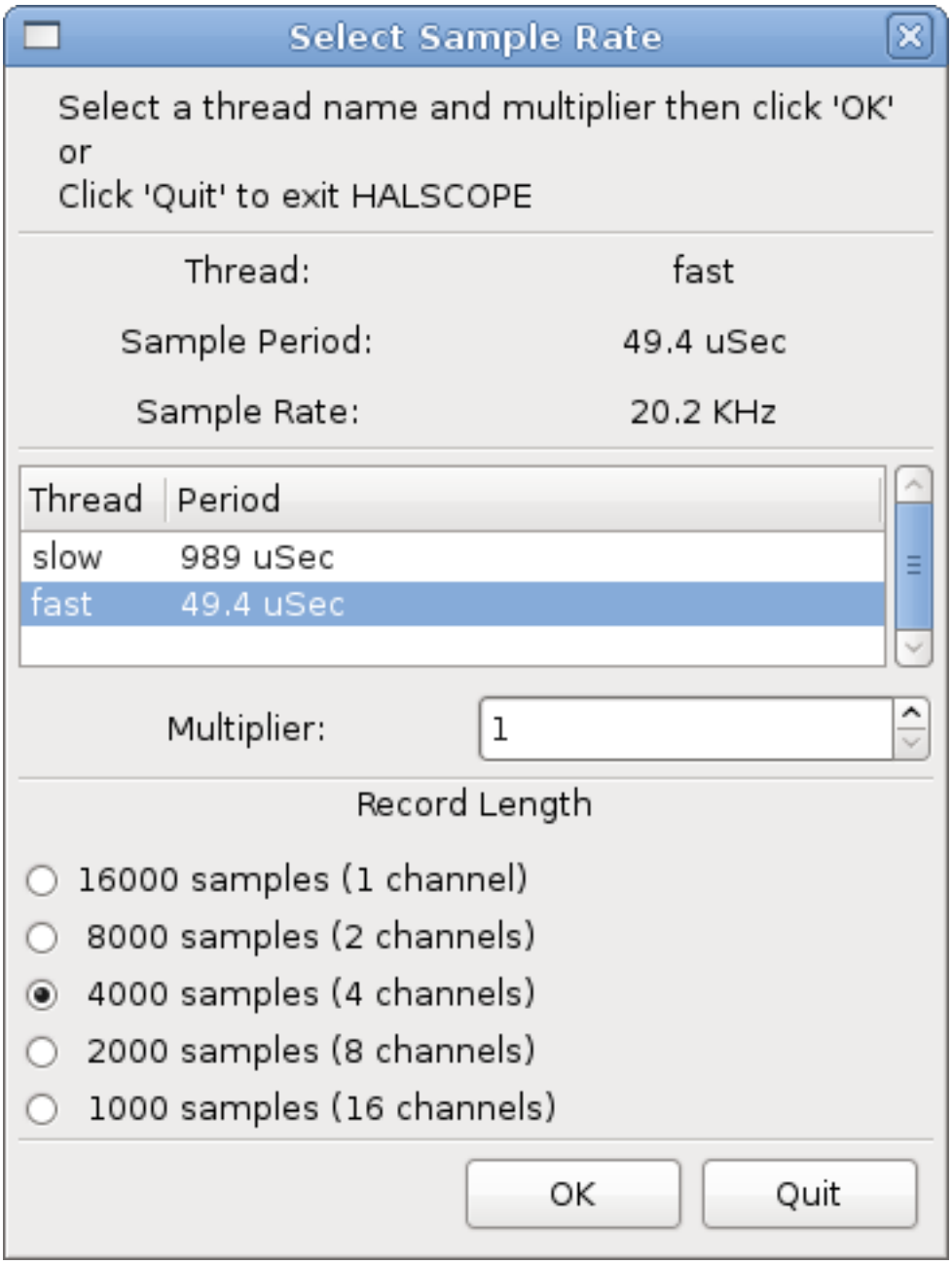


Figure 2.12: Sample Rate Dialog

2.5.6 More Channels

Now let's look at the step pulses. Halscope has 16 channels, but for this example we are using only 4 at a time. Before we select any more channels, we need to turn off a couple. Click on the channel 2 button, then click the *Chan Off* button at the bottom of the *Vertical* box. Then click on channel 3, turn if off, and do the same for channel 4. Even though the channels are turned off, they still remember what they are connected to, and in fact we will continue to use channel 3 as the trigger source. To add new channels, select channel 5, and choose pin *stepgen.0.dir*, then channel 6, and select *stepgen.0.step*. Then click run mode *Normal* to start the scope, and adjust the horizontal zoom to 5 ms per division. You should see the step pulses slow down as the velocity command (channel 1) approaches zero, then the direction pin changes state and the step pulses speed up again. You might want to increase the gain on channel 1 to about 20 milli per division to better see the change in the velocity command. The result should look like the following figure.

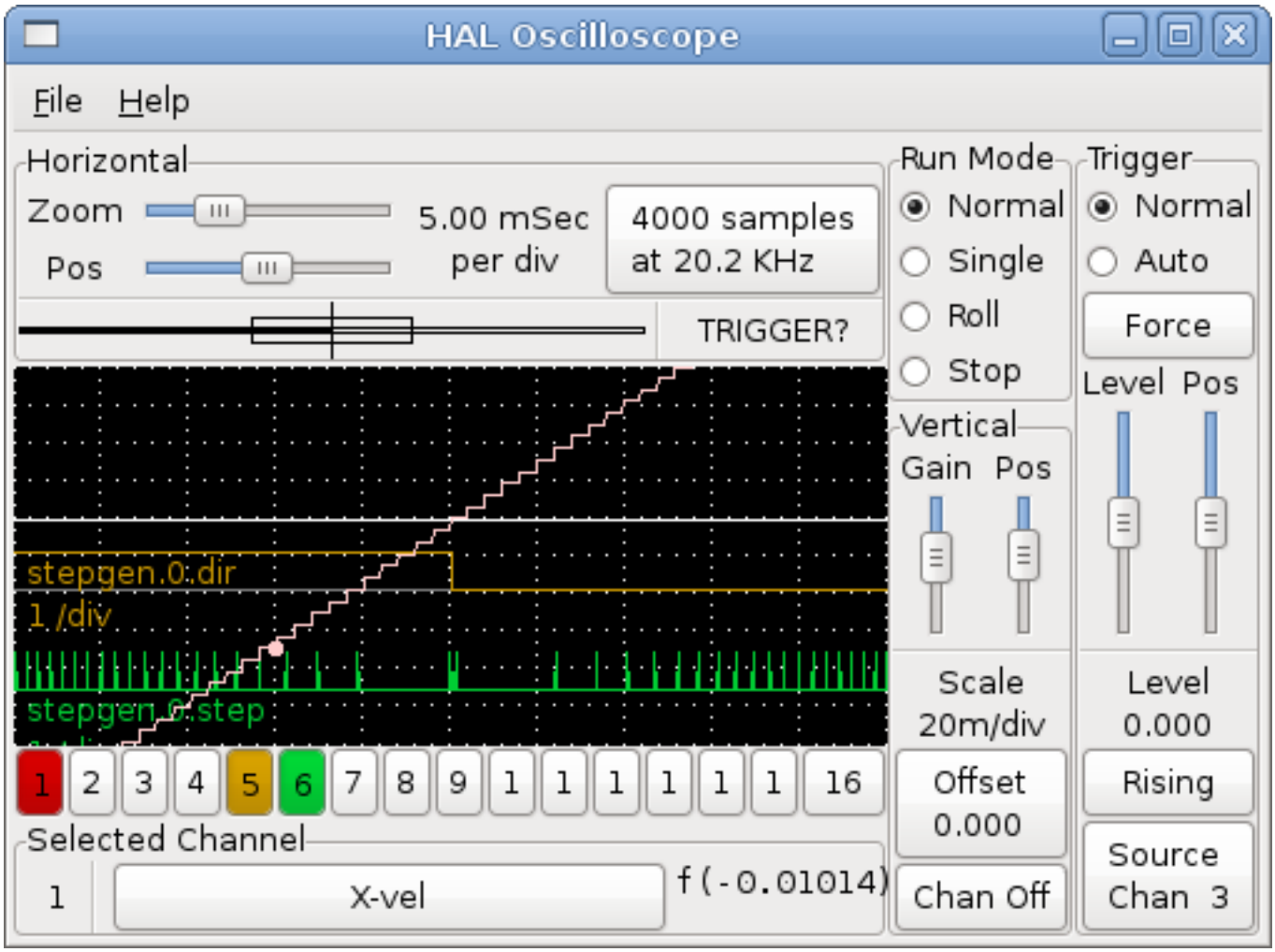


Figure 2.13: Step Pulses

2.5.7 More samples

If you want to record more samples at once, restart realtime and load halscope with a numeric argument which indicates the number of samples you want to capture.

```
halcmd: loadusr halscope 80000
```

If the *scope_rt* component was not already loaded, halscope will load it and request 80000 total samples, so that when sampling 4 channels at a time there will be 20000 samples per channel. (If *scope_rt* was already loaded, the numeric argument to halscope will have no effect).

Chapter 3

General Reference

3.1 General Naming Conventions

Consistent naming conventions would make HAL much easier to use. For example, if every encoder driver provided the same set of pins and named them the same way it would be easy to change from one type of encoder driver to another. Unfortunately, like many open-source projects, HAL is a combination of things that were designed, and things that simply evolved. As a result, there are many inconsistencies. This section attempts to address that problem by defining some conventions, but it will probably be a while before all the modules are converted to follow them.

Halcmd and other low-level HAL utilities treat HAL names as single entities, with no internal structure. However, most modules do have some implicit structure. For example, a board provides several functional blocks, each block might have several channels, and each channel has one or more pins. This results in a structure that resembles a directory tree. Even though halcmd doesn't recognize the tree structure, proper choice of naming conventions will let it group related items together (since it sorts the names). In addition, higher level tools can be designed to recognize such structure, if the names provide the necessary information. To do that, all HAL components should follow these rules:

- Dots (".") separate levels of the hirearchy. This is analogous to the slash ("/") in a filename.
- Hyphens ("-") separate words or fields in the same level of the hirearchy.
- HAL components should not use underscores or "MixedCase".
- Use only lowercase letters and numbers in names.

3.2 Hardware Driver Naming Conventions

3.2.1 Pin/Parameter names

Hardware drivers should use five fields (on three levels) to make up a pin or parameter name, as follows:

<device-name>.<device-num>.<io-type>.<chan-num>.<specific-name>

The individual fields are:

<device-name>
The device that the driver is intended to work with. This is most often an interface board of some type, but there are other possibilities.

<device-num>
It is possible to install more than one servo board, parallel port, or other hardware device in a computer. The device number identifies a specific device. Device numbers start at 0 and increment.

<io-type>
Most devices provide more than one type of I/O. Even the simple parallel port has both digital inputs and digital outputs. More complex boards can have digital inputs and outputs, encoder counters, pwm or step pulse generators, analog-to-digital converters, digital-to-analog converters, or other unique capabilities. The I/O type is used to identify the kind of I/O that a pin or parameter is associated with. Ideally, drivers that implement the same I/O type, even if for very different devices, should provide a consistent set of pins and parameters and identical behavior. For example, all digital inputs should behave the same when seen from inside the HAL, regardless of the device.

<chan-num>
Virtually every I/O device has multiple channels, and the channel number identifies one of them. Like device numbers, channel numbers start at zero and increment.[1] If more than one device is installed, the channel numbers on additional devices start over at zero. If it is possible to have a channel number greater than 9, then channel numbers should be two digits, with a leading zero on numbers less than 10 to preserve sort ordering. Some modules have pins and/or parameters that affect more than one channel. For example a PWM generator might have four channels with four independent "duty-cycle" inputs, but one "frequency" parameter that controls all four channels (due to hardware limitations). The frequency parameter should use "0-3" as the channel number.

<specific-name>
An individual I/O channel might have just a single HAL pin associated with it, but most have more than one. For example, a digital input has two pins, one is the state of the physical pin, the other is the same thing inverted. That allows the configurator to choose between active high and active low inputs. For most io-types, there is a standard set of pins and parameters, (referred to as the "canonical interface") that the driver should implement. The canonical interfaces are described in the Canonical Device Interfaces chapter.

EXAMPLES

motenc.0.encoder.2.position
— the position output of the third encoder channel on the first Motenc board.

stg.0.din.03.in
— the state of the fourth digital input on the first Servo-to-Go board.

ppmc.0.pwm.00-03.frequency
— the carrier frequency used for PWM channels 0 through 3 on the first Pico Systems ppmc board.

3.2.2 Function Names

Hardware drivers usually only have two kinds of HAL functions, ones that read the hardware and update HAL pins, and ones that write to the hardware using data from HAL pins. They should be named as follows:

`<device-name>-<device-num>.<io-type>-<chan-num-range>.read|write`

<device-name>
The same as used for pins and parameters.

<device-num>
The specific device that the function will access.

<io-type>
Optional. A function may access all of the I/O on a board, or it may access only a certain type. For example, there may be independent functions for reading encoder counters and reading digital I/O. If such independent functions exist, the <io-type> field identifies the type of I/O they access. If a single function reads all I/O provided by the board, <io-type> is not used. [2]

[1] One exception to the "channel numbers start at zero" rule is the parallel port. Its HAL pins are numbered with the corresponding pin number on the DB-25 connector. This is convenient for wiring, but inconsistent with other drivers. There is some debate over whether this is a bug or a feature.

[2] Note to driver programmers: do NOT implement separate functions for different I/O types unless they are interruptible and can work in independent threads. If interrupting an encoder read, reading digital inputs, and then resuming the encoder read will cause problems, then implement a single function that does everything.

<chan-num-range>
Optional. Used only if the <io-type> I/O is broken into groups and accessed by different functions.

read|write
Indicates whether the function reads the hardware or writes to it.

EXAMPLES

motenc.0.encoder.read
— reads all encoders on the first motenc board.

generic8255.0.din.09-15.read
— reads the second 8 bit port on the first generic 8255 based digital I/O board.

ppmc.0.write
— writes all outputs (step generators, pwm, DACs, and digital) on the first Pico Systems ppmc board.

Chapter 4

Canonical Device Interfaces

Note

By version 2.1, the HAL drivers should have all been updated to match these specs. Send an email if you spot any problems.

4.1 Introduction

The following sections show the pins, parameters, and functions that are supplied by "canonical devices". All HAL device drivers should supply the same pins and parameters, and implement the same behavior.

Note that the only the `<io-type>` and `<specific-name>` fields are defined for a canonical device. The `<device-name>`, `<device-num>`, and `<chan-num>` fields are set based on the characteristics of the real device.

4.2 Digital Input

The canonical digital input (I/O type field: `digin`) is quite simple.

4.2.1 Pins

- (bit) **in** — State of the hardware input.
- (bit) **in-not** — Inverted state of the input.

4.2.2 Parameters

- None

4.2.3 Functions

- (funct) **read** — Read hardware and set `in` and `in-not` HAL pins.

4.3 Digital Output

The canonical digital output (I/O type field: `digout`) is also very simple.

4.3.1 Pins

- (bit) **out** — Value to be written (possibly inverted) to the hardware output.

4.3.2 Parameters

- (bit) **invert** — If TRUE, **out** is inverted before writing to the hardware.

4.3.3 Functions

- (funct) **write** — Read **out** and **invert**, and set hardware output accordingly.

4.4 Analog Input

The canonical analog input (I/O type: `adcin`). This is expected to be used for analog to digital converters, which convert e.g. voltage to a continuous range of values.

4.4.1 Pins

- (float) **value** — The hardware reading, scaled according to the **scale** and **offset** parameters. **Value** = ((input reading, in hardware-dependent units) * **scale**) - **offset**

4.4.2 Parameters

- (float) **scale** — The input voltage (or current) will be multiplied by **scale** before being output to **value**.
- (float) **offset** — This will be subtracted from the hardware input voltage (or current) after the scale multiplier has been applied.
- (float) **bit_weight** — The value of one least significant bit (LSB). This is effectively the granularity of the input reading.
- (float) **hw_offset** — The value present on the input when 0 volts is applied to the input pin(s).

4.4.3 Functions

- (funct) **read** — Read the values of this analog input channel. This may be used for individual channel reads, or it may cause all channels to be read

4.5 Analog Output

The canonical analog output (I/O Type: **adcout**). This is intended for any kind of hardware that can output a more-or-less continuous range of values. Examples are digital to analog converters or PWM generators.

4.5.1 Pins

- (float) **value** — The value to be written. The actual value output to the hardware will depend on the scale and offset parameters.
- (bit) **enable** — If false, then output 0 to the hardware, regardless of the **value** pin.

4.5.2 Parameters

- (float) **offset** — This will be added to the **value** before the hardware is updated
- (float) **scale** — This should be set so that an input of 1 on the **value** pin will cause the analog output pin to read 1 volt.
- (float) **high_limit** (optional) — When calculating the value to output to the hardware, if **value + offset** is greater than **high_limit**, then **high_limit** will be used instead.
- (float) **low_limit** (optional) — When calculating the value to output to the hardware, if **value + offset** is less than **low_limit**, then **low_limit** will be used instead.
- (float) **bit_weight** (optional) — The value of one least significant bit (LSB), in volts (or mA, for current outputs)
- (float) **hw_offset** (optional) — The actual voltage (or current) that will be output if 0 is written to the hardware.

4.5.3 Functions

(funct) **write** — This causes the calculated value to be output to the hardware. If enable is false, then the output will be 0, regardless of **value**, **scale**, and **offset**. The meaning of "0" is dependent on the hardware. For example, a bipolar 12-bit A/D may need to write 0x1FF (mid scale) to the D/A get 0 volts from the hardware pin. If enable is true, read scale, offset and value and output to the adc (**scale** * **value**) + **offset**. If enable is false, then output 0.

Chapter 5

HAL Tools

5.1 Halcmd

Halcmd is a command line tool for manipulating the HAL. There is a rather complete man page for halcmd, which will be installed if you have installed LinuxCNC from either source or a package. If you have compiled LinuxCNC for "run-in-place", the man page is not installed, but it is accessible. From the main LinuxCNC directory, do:

```
man -M docs/man halcmd
```

The HAL Tutorial has a number of examples of halcmd usage, and is a good tutorial for halcmd.

5.2 Halmeter

Halmeter is a *voltmeter* for the HAL. It lets you look at a pin, signal, or parameter, and displays the current value of that item. It is pretty simple to use. Start it by typing **halmeter** in an X windows shell. Halmeter is a GUI application. It will pop up a small window, with two buttons labeled *Select* and *Exit*. Exit is easy - it shuts down the program. Select pops up a larger window, with three tabs. One tab lists all the pins currently defined in the HAL. The next lists all the signals, and the last tab lists all the parameters. Click on a tab, then click on a pin/signal/parameter. Then click on *OK*. The lists will disappear, and the small window will display the name and value of the selected item. The display is updated approximately 10 times per second. If you click *Accept* instead of *OK*, the small window will display the name and value of the selected item, but the large window will remain on the screen. This is convenient if you want to look at a number of different items quickly.

You can have many halmeters running at the same time, if you want to monitor several items. If you want to launch a halmeter without tying up a shell window, type *halmeter &* to run it in the background. You can also make halmeter start displaying a specific item immediately, by adding *pin|sig|par[am] <name>* to the command line. It will display the pin, signal, or parameter <name> as soon as it starts. (If there is no such item, it will simply start normally.) And finally, if you specify an item to display, you can add *-s* before the pin|sig|param to tell halmeter to use a small window. The item name will be displayed in the title bar instead of under the value, and there will be no buttons. Useful when you want a lot of meters in a small amount of screen space.

Refer to Halmeter Tutorial section for more information.

Halmeter can be loaded from a terminal or from Axis. Halmeter is faster than Halshow at displaying values. Halmeter has two windows, one to pick the pin, signal, or parameter to monitor and one that displays the value. Multiple Halmeters can be open at the same time. If you use a script to open multiple Halmeters you can set the position of each one with -g X Y relative to the upper left corner of your screen. For example:

```
loadusr halmeter pin hm2.0.stepgen.00.velocity-fb -g 0 500
```

See the man page for more options. See section Halmeter.

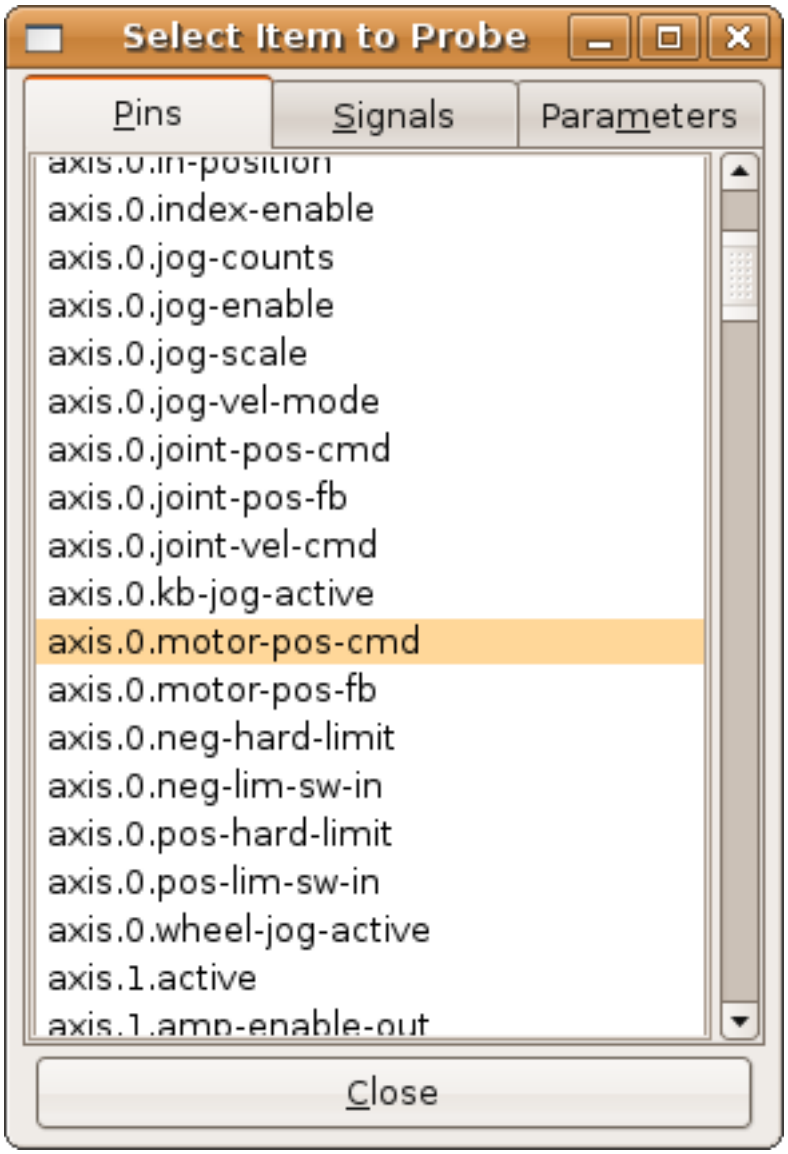
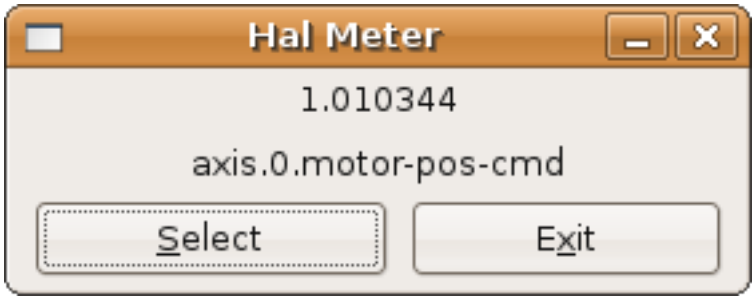


Figure 5.1: Halmeter



5.3 Halscope

Halscope is an *oscilloscope* for the HAL. It lets you capture the value of pins, signals, and parameters as a function of time. Complete operating instructions should be located here eventually. For now, refer to section [sec:Tutorial-Halscope] in the tutorial chapter, which explains the basics.

Chapter 6

Basic HAL Tutorial

6.1 HAL Commands

More detailed information can be found in the man page for halcmd *man halcmd* in a terminal window. To see the HAL configuration and check the status of pins and parameters use the HAL Configuration window on the Machine menu in AXIS. To watch a pin status open the Watch tab and click on each pin you wish to watch and it will be added to the watch window.

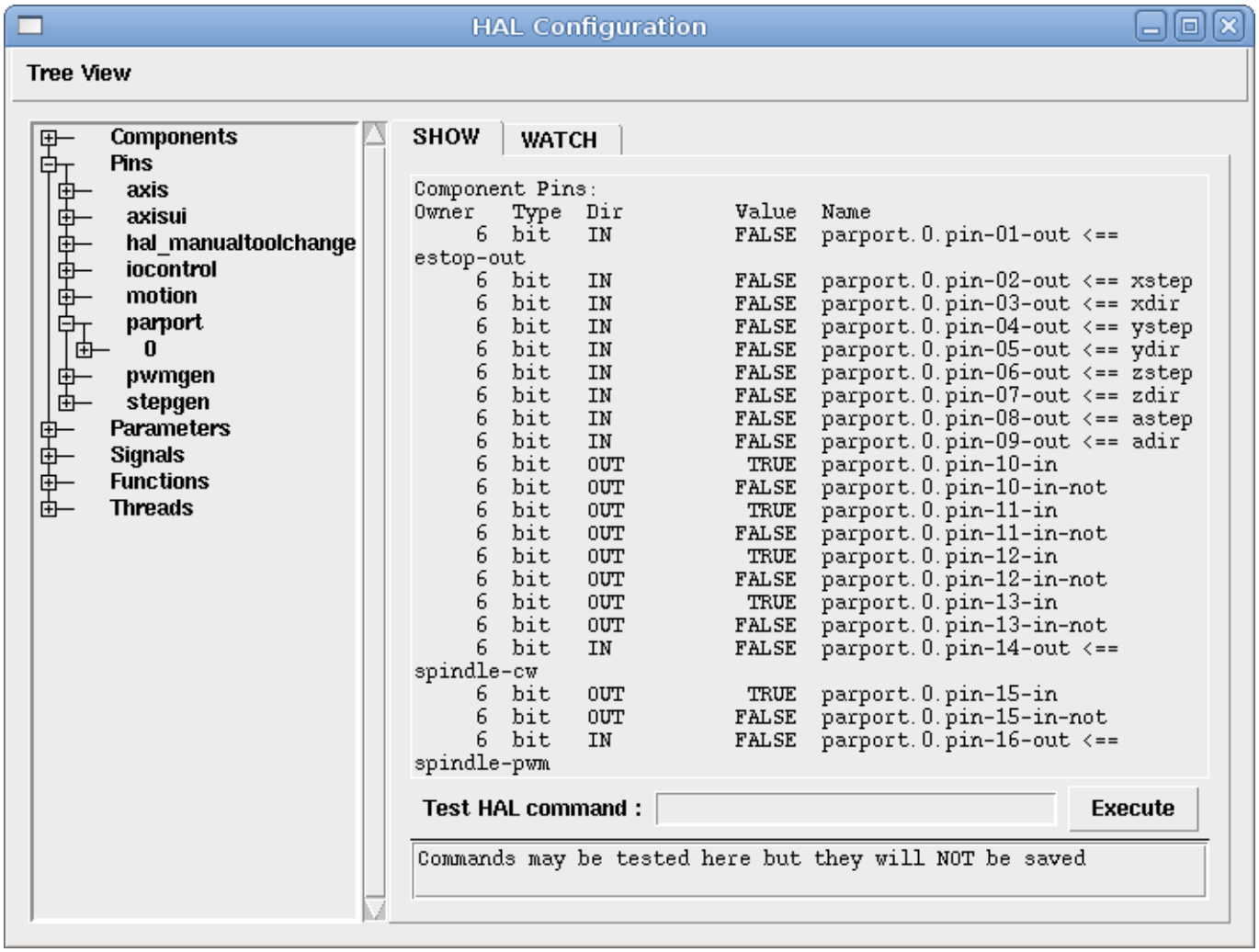


Figure 6.1: HAL Configuration Window

6.1.1 loadrt

The command *loadrt* loads a real time HAL component. Real time component functions need to be added to a thread to be updated at the rate of the thread. You cannot load a user space component into the real time space.

The syntax and an example:

```
loadrt <component> <options>

loadrt mux4 count=1
```

6.1.2 addf

The command *addf* adds a real time component function to a thread. You have to add a function from a HAL real time component to a thread to get the function to update at the rate of the thread.

If you used the Stepper Config Wizard to generate your config you will have two threads.

- base-thread (the high-speed thread): this thread handles items that need a fast response, like making step pulses, and reading and writing the parallel port.

- servo-thread (the slow-speed thread): this thread handles items that can tolerate a slower response, like the motion controller, ClassicLadder, and the motion command handler.

The syntax and an example:

```
addf <component> <thread>

addf mux4 servo-thread
```

6.1.3 loadusr

The command *loadusr* loads a user space HAL component. User space programs are their own separate processes, which optionally talk to other HAL components via pins and parameters. You cannot load real time components into user space.

Flags may be one or more of the following:

-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>. This only applies if the component has a name option.
-w	to wait for the program to exit
-i	to ignore the program return value (with -w)
-n	name a component when it is a valid option for that component.

The syntax and examples:

```
loadusr <component> <options>

loadusr halui

loadusr -Wn spindle gs2_vfd -n spindle
```

In English it means *loadusr wait for name spindle component gs2_vfd name spindle*.

6.1.4 net

The command *net* creates a *connection* between a signal and one or more pins. If the signal does not exist net creates the new signal. This replaces the need to use the command newsig. The optional direction arrows <=, => and <=> make it easier to follow the logic when reading a *net* command line and are not used by the net command. The direction arrows must be separated by a space from the pin names.

Syntax and Example:

```
net signal-name pin-name <optional arrow> <optional second pin-name>

net home-x axis.0.home-sw-in <= parport.0.pin-11-in
```

In the above example *home-x* is the signal name, *axis.0.home-sw-in* is a *Direction IN* pin, <= is the optional direction arrow, and *parport.0.pin-11-in* is a *Direction OUT* pin. This may seem confusing but the in and out labels for a parallel port pin indicates the physical way the pin works not how it is handled in HAL.

A pin can be connected to a signal if it obeys the following rules:

- An IN pin can always be connected to a signal
- An IO pin can be connected unless there's an OUT pin on the signal
- An OUT pin can be connected only if there are no other OUT or IO pins on the signal

The same *signal-name* can be used in multiple net commands to connect additional pins, as long as the rules above are obeyed.

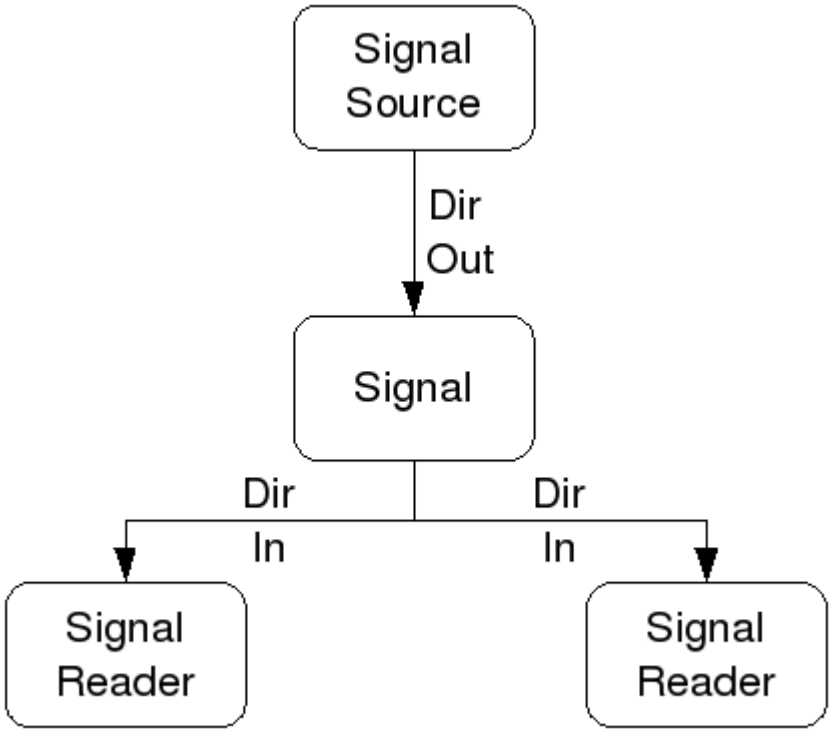


Figure 6.2: Signal Direction

This example shows the signal xStep with the source being stepgen.0.out and with two readers, parport.0.pin-02-out and parport.0.pin-08-out. Basically the value of stepgen.0.out is sent to the signal xStep and that value is then sent to parport.0.pin-02-out and parport.0.pin-08-out.

```
#   signal    source            destination          destination
net xStep stepgen.0.out => parport.0.pin-02-out parport.0.pin-08-out
```

Since the signal xStep contains the value of stepgen.0.out (the source) you can use the same signal again to send the value to another reader. To do this just use the signal with the readers on another line.

```
net xStep => parport.0.pin-02-out
```

I/O pins An I/O pin like encoder.N.index-enable can be read or set as allowed by the component.

6.1.5 setp

The command *setp* sets the value of a pin or parameter. The valid values will depend on the type of the pin or parameter. It is an error if the data types do not match.

Some components have parameters that need to be set before use. Parameters can be set before use or while running as needed. You cannot use setp on a pin that is connected to a signal.

The syntax and an example:

```
setp <pin/parameter-name> <value>

setp parport.0.pin-08-out TRUE
```

6.1.6 sets

The command *sets* sets the value of a signal.

The syntax and an example:

```
sets <signal-name> <value>

net mysignal and2.0.in0 pyvcp.my-led

sets mysignal 1
```

It is an error if:

- The signal-name does not exist
- If the signal already has a writer
- If value is not the correct type for the signal

6.1.7 unlinkp

The command *unlinkp* unlinks a pin from the connected signal. If no signal was connected to the pin prior running the command, nothing happens. The *unlinkp* command is useful for trouble shooting.

The syntax and an example:

```
unlinkp <pin-name>

unlinkp parport.0.pin-02-out
```

6.1.8 Obsolete Commands

The following commands are depreciated and may be removed from future versions. Any new configuration should use the net command. These commands are included so older configurations will still work.

6.1.8.1 linksp

The command *linksp* creates a *connection* between a signal and one pin.

The syntax and an example:

```
linksp <signal-name> <pin-name>
linksp X-step parport.0.pin-02-out
```

The *linksp* command has been superseded by the *net* command.

6.1.8.2 linkps

The command *linkps* creates a *connection* between one pin and one signal. It is the same as linksp but the arguments are reversed.

The syntax and an example:

```
linkps <pin-name> <signal-name>

linkps parport.0.pin-02-out X-Step
```

The *linkps* command has been superseded by the *net* command.

6.1.8.3 newsig

the command *newsig* creates a new HAL signal by the name <signame> and the data type of <type>. Type must be *bit*, *s32*, *u32* or *float*. Error if <signame> all ready exists.

The syntax and an example:

```
newsig <signame> <type>

newsig Xstep bit
```

More information can be found in the HAL manual or the man pages for halrun.

6.2 HAL Data

6.2.1 Bit

A bit value is an on or off.

- bit values = true or 1 and false or 0 (True, TRUE, true are all valid)

6.2.2 Float

A *float* is a floating point number. In other words the decimal point can move as needed.

- float values = a 64 bit floating point value, with approximately 53 bits of resolution and over 1000 bits of dynamic range.

For more information on floating point numbers see:

http://en.wikipedia.org/wiki/Floating_point

6.2.3 s32

An *s32* number is a whole number that can have a negative or positive value.

- s32 values = integer numbers -2147483648 to 2147483647

6.2.4 u32

A *u32* number is a whole number that is positive only.

- u32 values = integer numbers 0 to 4294967295

6.3 HAL Files

If you used the Stepper Config Wizard to generate your config you will have up to three HAL files in your config directory.

- my-mill.hal (if your config is named *my-mill*) This file is loaded first and should not be changed if you used the Stepper Config Wizard.
- custom.hal This file is loaded next and before the GUI loads. This is where you put your custom HAL commands that you want loaded before the GUI is loaded.
- custom_postgui.hal This file is loaded after the GUI loads. This is where you put your custom HAL commands that you want loaded after the GUI is loaded. Any HAL commands that use pyVCP widgets need to be placed here.

6.4 HAL Components

Two parameters are automatically added to each HAL component when it is created. These parameters allow you to scope the execution time of a component.

`.time`

`.tmax`

Time is the number of CPU cycles it took to execute the function.

Tmax is the maximum number of CPU cycles it took to execute the function. Tmax is a read/write parameter so the user can set it to 0 to get rid of the first time initialization on the function's execution time.

6.5 Logic Components

HAL contains several real time logic components. Logic components follow a *Truth Table* that states what the output is for any given input. Typically these are bit manipulators and follow electrical logic gate truth tables.

6.5.1 and2

The *and2* component is a two input *and* gate. The truth table below shows the output based on each combination of input.

Syntax

```
and2 [count=N] | [names=name1[,name2...]]
```

Functions

and2.n

Pins

```
and2.N.in0 (bit, in)
and2.N.in1 (bit, in)
and2.N.out (bit, out)
```

Truth Table

in0	**in1**	**out**
False	False	False
True	False	False
False	True	False
True	True	True

6.5.2 not

The *not* component is a bit inverter.

Syntax

```
not [count=n] | [names=name1[,name2...]]
```

Functions

```
not.all
not.n
```

Pins

```
not.n.in (bit, in)
not.n.out (bit, out)
```

Truth Table

in	out
True	False
False	True

6.5.3 or2

The *or2* component is a two input OR gate.

Syntax

```
or2[count=n] | [names=name1[,name2...]]
```

Functions

```
or2.n
```

Pins

```
or2.n.in0 (bit, in)
or2.n.in1 (bit, in)
or2.n.out (bit, out)
```

Truth Table

in0	in1	out
True	False	True
True	True	True
False	True	True
False	False	False

6.5.4 xor2

The *xor2* component is a two input XOR (exclusive OR)gate.

Syntax

```
xor2[count=n] | [names=name1[,name2...]]
```

Functions

```
xor2.n
```

Pins

```
xor2.n.in0 (bit, in)
xor2.n.in1 (bit, in)
xor2.n.out (bit, out)
```

Truth Table

in0	in1	out
True	False	True
True	True	False
False	True	True
False	False	False

6.5.5 Logic Examples

An *and2* example connecting two inputs to one output.

```
loadrt and2 count=1

addf and2.0 servo-thread

net my-sigin1 and2.0.in0 <= parport.0.pin-11-in

net my-sigin2 and2.0.in1 <= parport.0.pin-12-in

net both-on parport.0.pin-14-out <= and2.0.out
```

In the above example one copy of and2 is loaded into real time space and added to the servo thread. Next pin 11 of the parallel port is connected to the in0 bit of the and gate. Next pin 12 is connected to the in1 bit of the and gate. Last we connect the and2 out bit to the parallel port pin 14. So following the truth table for and2 if pin 11 and pin 12 are on then the output pin 14 will be on.

6.6 Conversion Components

6.6.1 weighted_sum

The weighted_sum converts a group of bits to an integer. The conversion is the sum of the *weights* of the bits that are on plus any offset. The weight of the m-th bit is 2^m. This is similar to a binary coded decimal but with more options. The *hold* bit stops processing the input changes so the *sum* will not change.

The following syntax is used to load the weighted_sum component.

```
loadrt weighted_sum wsum_sizes=size[,size,...]
```

Creates weighted sum groups each with the given number of input bits (size).

To update the weighted_sum you need to attach process_wsums to a thread.

```
addf process_wsums servo-thread
```

This updates the weighted_sum component.

In the following example clipped from the HAL Configuration window in Axis the bits *0* and *2* are true and there is no offset. The *weight* of 0 is 1 and the *weight* of 2 is 4 so the sum is 5.

weighted_sum

```
Component Pins:
Owner   Type  Dir         Value  Name
    10  bit   In           TRUE  wsum.0.bit.0.in
    10  s32   I/O             1  wsum.0.bit.0.weight
    10  bit   In          FALSE  wsum.0.bit.1.in
    10  s32   I/O             2  wsum.0.bit.1.weight
    10  bit   In           TRUE  wsum.0.bit.2.in
    10  s32   I/O             4  wsum.0.bit.2.weight
    10  bit   In          FALSE  wsum.0.bit.3.in
    10  s32   I/O             8  wsum.0.bit.3.weight
    10  bit   In          FALSE  wsum.0.hold
    10  s32   I/O             0  wsum.0.offset
    10  s32   Out             5  wsum.0.sum
```

Chapter 7

Halshow

The script halshow can help you find your way around a running HAL. This is a very specialized system and it must connect to a working HAL. It cannot run standalone because it relies on the ability of HAL to report what it knows of itself through the halcmd interface library. It is discovery based. Each time halshow runs with a different LinuxCNC configuration it will be different.

As we will soon see, this ability of HAL to document itself is one key to making an effective CNC system.

7.1 Starting Halshow

Halshow is in the AXIS menu under Machine/Show HAL Configuration.

Halshow is in the TkLinuxCNC menu under Scripts/HAL Show.

7.2 HAL Tree Area

At the left of its display as shown in figure [cap:Halshow-Layout] is a tree view, somewhat like you might see with some file browsers. At the right is a tabbed notebook with tabs for show and watch.

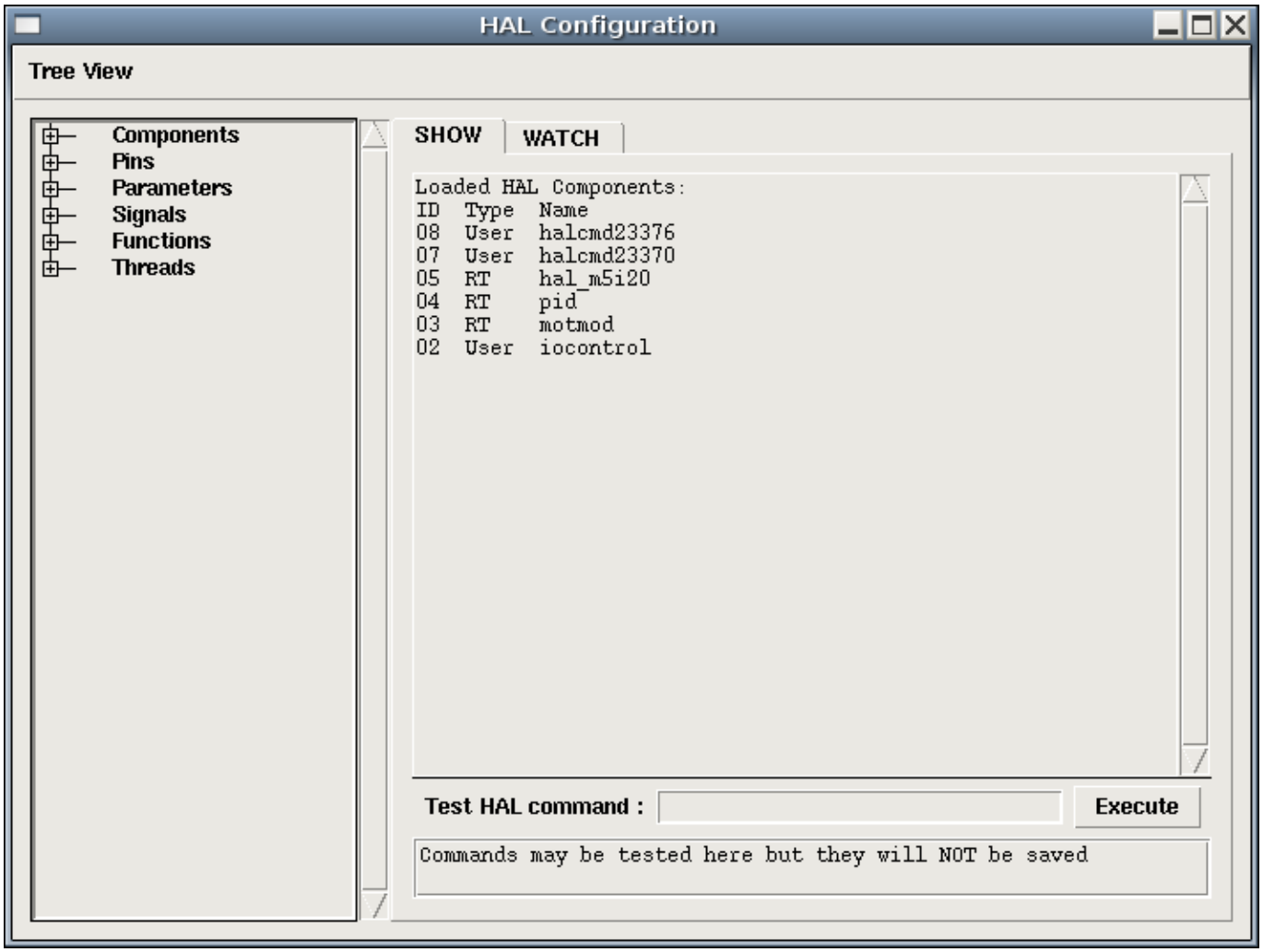


Figure 7.1: Halshow Layout

The tree shows all of the major parts of a HAL. In front of each is a small plus (+) or minus (-) sign in a box. Clicking the plus will expand that tree node to display what is under it. If that box shows a minus sign, clicking it will collapse that section of the tree.

You can also expand or collapse the tree display using the Tree View menu at the upper left edge of the display. Under Tree View you will find: Expand Tree, Collapse Tree; Expand Pins, Expand Parameters, Expand Signals; and Erase Watch. (Note that Erase Watch erases *all* previously set watches, you cannot erase just one watch.)

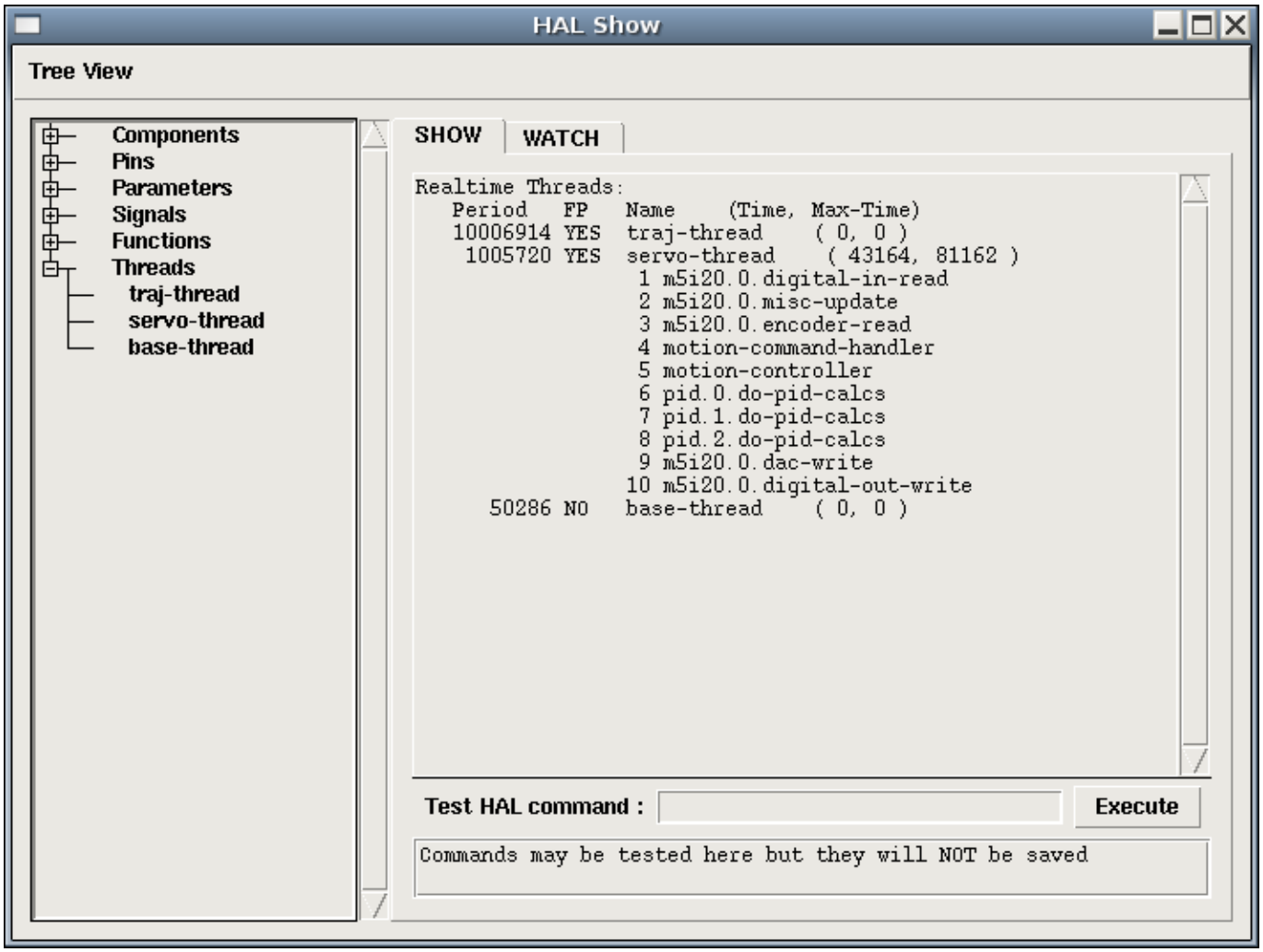


Figure 7.2: Show Menu

7.3 HAL Show Area

Clicking on the node name, the word "Components" for example, will show you (under the "Show" tab) all that HAL knows about the contents of that node. Figure [cap:Halshow-Layout] shows a list exactly like you will see if you click the "Components" name while you are running a standard m5i20 servo card. The information display is exactly like those shown in traditional text based HAL analysis tools. The advantage here is that we have mouse click access, access that can be as broad or as focused as you need.

If we take a closer look at the tree display we can see that the six major parts of a HAL can all be expanded at least one level. As these levels are expanded you can get more focused with the reply when you click on the rightmost tree node. You will find that there are some HAL pins and parameters that show more than one reply. This is due to the nature of the search routines in halcmd itself. If you search one pin you may get two, like this:

```
Component Pins:
Owner  Type  Dir  Value  Name
06     bit    -W   TRUE  parport.0.pin-10-in
06     bit    -W  FALSE  parport.0.pin-10-in-not
```

The second pin's name contains the complete name of the first.

Below the show area on the right is a set of widgets that will allow you to play with the running HAL. The commands you enter here and the effect that they have on the running HAL are not saved. They will persist as long as LinuxCNC remains up but are gone as soon as LinuxCNC is.

The entry box labeled "Test HAL Command:" will accept any of the commands listed for halcmd. These include:

- loadrt, unloadrt (load/unload real-time module)
- loadusr, unloadusr (load/unload user-space component)
- addf, delf (add/delete a function to/from a real-time thread)
- net (create a connection between two or more items)
- setp (set parameter (or pin) to a value)

This little editor will enter a command any time you press <enter> or push the execute button. An error message from halcmd will show below this entry widget when these commands are not properly formed. If you are not certain how to set up a proper command you'll need to read again the documentation on halcmd and the specific modules that you are working with.

Let's use this editor to add a differential module to a HAL and connect it to axis position so that we could see the rate of change in position, i.e., acceleration. We first need to load a HAL component named blocks, add it to the servo thread, then connect it to the position pin of an axis. Once that is done we can find the output of the differentiator in halscope. So let's go. (Yes, I looked this one up.)

```
loadrt blocks ddt=1
```

Now look at the components node and you should see blocks in there someplace.

```
Loaded HAL Components:
ID Type        Name
10 User halcmd29800
09 User halcmd29374
08   RT      blocks
06   RT hal_parport
05   RT    scope_rt
04   RT     stepgen
03   RT      motmod
02 User   iocontrol
```

Sure enough there it is. Notice that its ID is 08. Next we need to find out what functions are available with it so we look at functions:

```
Exported Functions:
Owner  CodeAddr      Arg  FP Users Name
  08   E0B97630 E0DC7674 YES     0 ddt.0
  03   E0DEF83C 00000000 YES     1 motion-command-handler
  03   E0DF0BF3 00000000 YES     1 motion-controller
  06   E0B541FE E0DC75B8  NO     1 parport.0.read
  06   E0B54270 E0DC75B8  NO     1 parport.0.write
  06   E0B54309 E0DC75B8  NO     0 parport.read-all
  06   E0B5433A E0DC75B8  NO     0 parport.write-all
  05   E0AD712D 00000000  NO     0 scope.sample
  04   E0B618C1 E0DC7448 YES     1 stepgen.capture-position
  04   E0B612F5 E0DC7448  NO     1 stepgen.make-pulses
  04   E0B614AD E0DC7448 YES     1 stepgen.update-freq
```

Here we look for owner #08 and see that blocks has exported a function named ddt.0. We should be able to add ddt.0 to the servo thread and it will do its math each time the servo thread is updated. Once again we look up the addf command and find that it uses three arguments like this:

```
addf <functname> <threadname> [<position>]
```

We already know the functname=ddt.0 so let's get the thread name right by expanding the thread node in the tree. Here we see two threads, servo-thread and base-thread. The position of ddt.0 in the thread is not critical. So we add the function ddt.0 to the servo-thread:

```
addf ddt.0 servo-thread
```

This is just for viewing, so we leave position blank and get the last position in the thread. Figure [cap:Addf-Command] shows the state of halshow after this command has been issued.

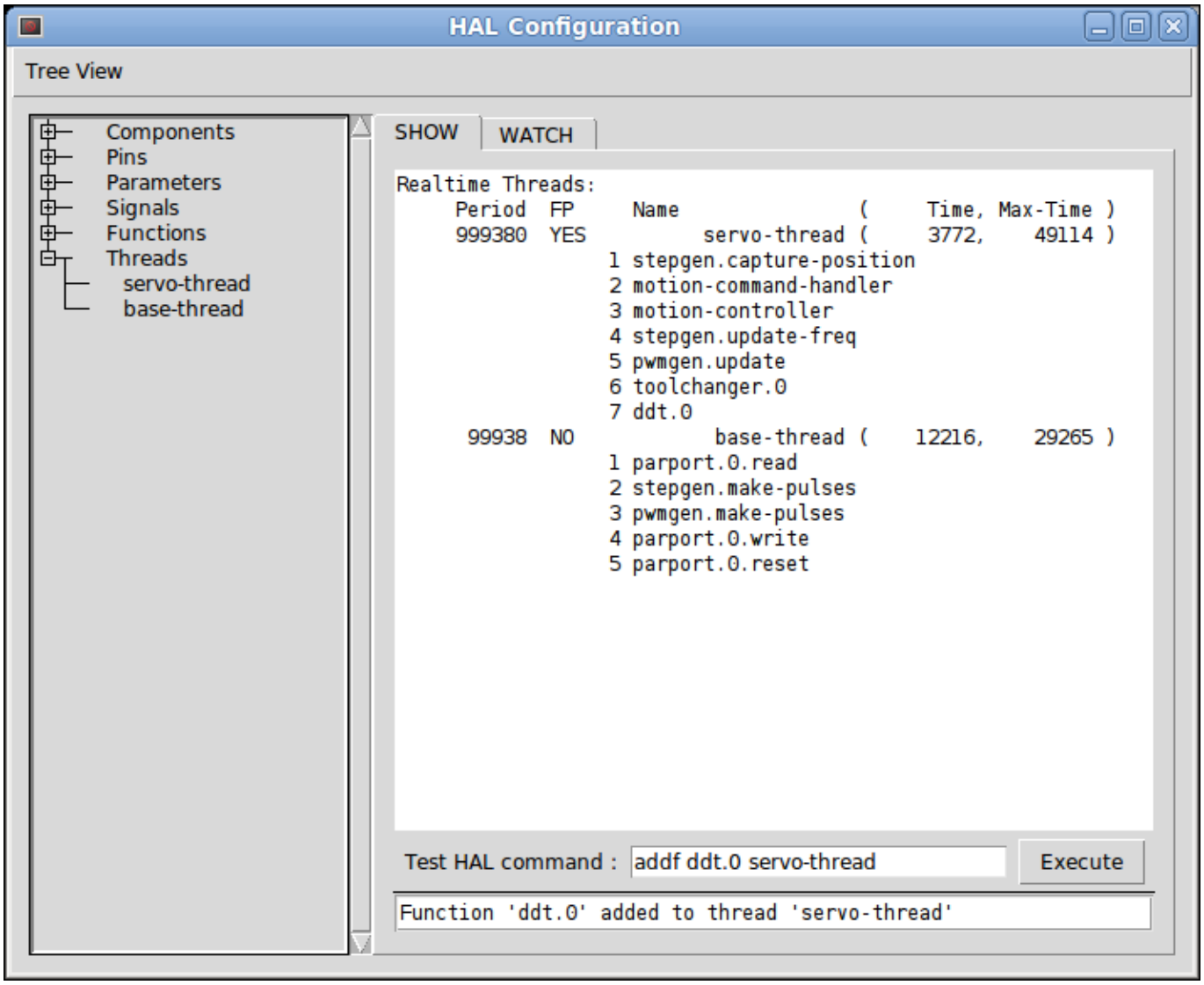


Figure 7.3: Addf Command

Next we need to connect this block to something. But how do we know what pins are available? The answer is to look under pins. There we find ddt and see this:

```
Component Pins:
Owner Type  Dir Value       Name
08    float R-  0.00000e+00 ddt.0.in
08    float -W  0.00000e+00 ddt.0.out
```

That looks easy enough to understand, but what signal or pin do we want to connect to it? It could be an axis pin, a stepgen pin, or a signal. We see this when we look at axis.0:

```
Component Pins:
Owner Type  Dir Value       Name
03    float -W  0.00000e+00 axis.0.motor-pos-cmd ==> Xpos-cmd
```

So it looks like Xpos-cmd should be a good signal to use. Back to the editor where we enter the following command:

```
linksp Xpos-cmd ddt.0.in
```

Now if we look at the Xpos-cmd signal using the tree node we'll see what we've done:

```
Signals:
Type Value Name
float 0.00000e+00 Xpos-cmd
<== axis.0.motor-pos-cmd
==> ddt.0.in
==> stepgen.0.position-cmd
```

We see that this signal comes from axis.o.motor-pos-cmd and goes to both ddt.0.in and stepgen.0.position-cmd. By connecting our block to the signal we have avoided any complications with the normal flow of this motion command.

The HAL Show Area uses halcmd to discover what is happening in a running HAL. It gives you complete information about what it has discovered. It also updates as you issue commands from the little editor panel to modify that HAL. There are times when you want a different set of things displayed without all of the information available in this area. That is where the HAL Watch Area is of value.

7.4 HAL Watch Area

Clicking the watch tab produces a blank canvas. You can add signals and pins to this canvas and watch their values.[1] You can add signals or pins when the watch tab is displayed by clicking on the name of it. Figure [cap:Watch-Display] shows this canvas with several "bit" type signals. These signals include enable-out for the first three axes and two of the three iocontrol "estop" signals. Notice that the axes are not enabled even though the estop signals say that LinuxCNC is not in estop. A quick look at TkLinuxCNC shows that the condition of LinuxCNC is ESTOP RESET. The amp enables do not turn true until the machine has been turned on.

[1] The refresh rate of the watch display is much lower than Halmeter or Halscope. If you need good resolution of the timing of signals those tools are much more effective.

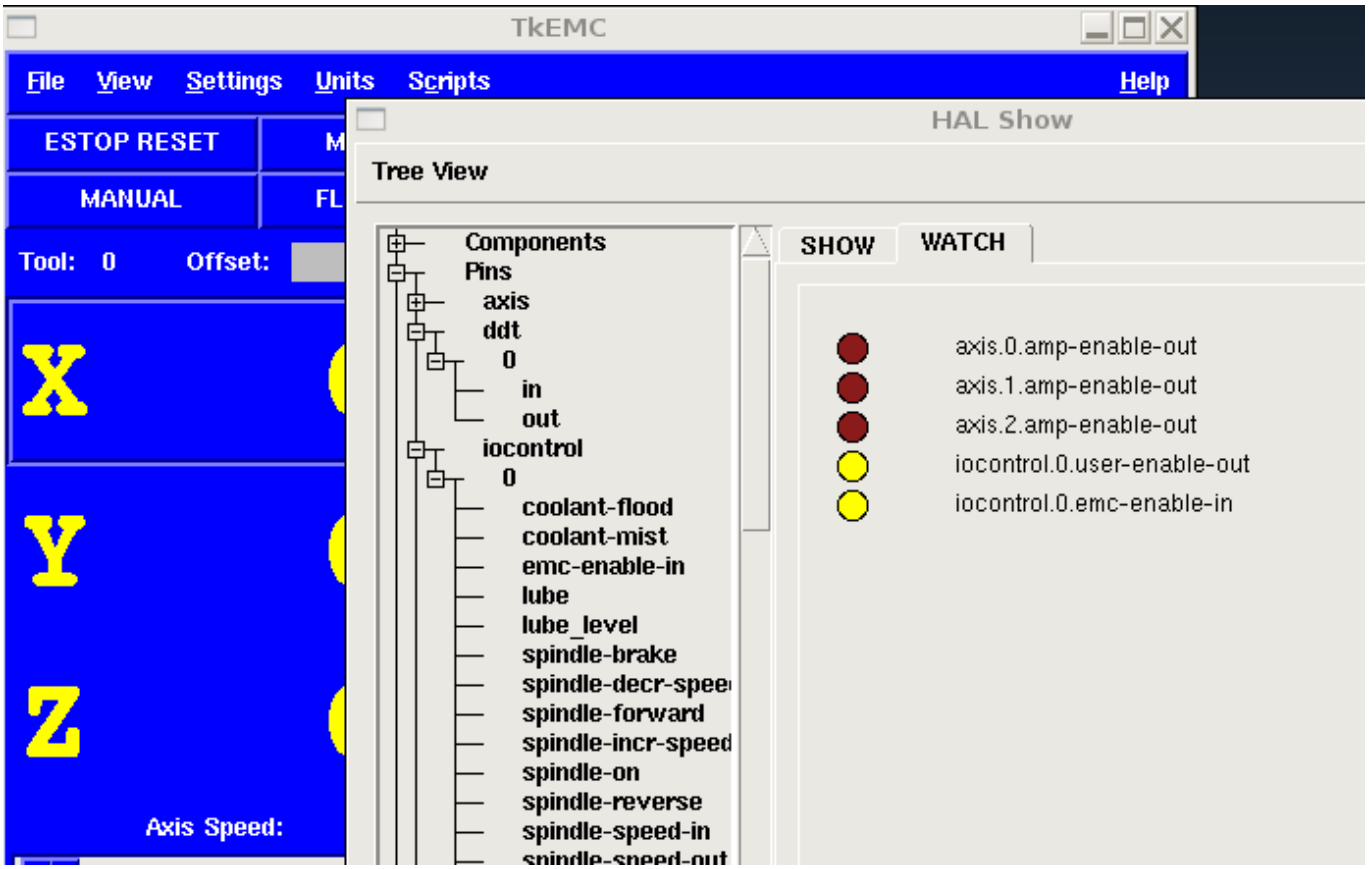


Figure 7.4: Watch Display

Watch displays bit type (binary) values using colored circles representing LEDs. They show as dark brown when a bit signal or pin is false, and as light yellow whenever that signal is true. If you select a pin or signal that is not a bit type (binary) signal, watch will show it as a numerical value.

Watch will quickly allow you to test switches or see the effect of changes that you make to LinuxCNC while using the graphical interface. Watch's refresh rate is a bit slow to see stepper pulses, but you can use it for these if you move an axis very slowly or in very small increments of distance. If you've used IO_Show in LinuxCNC, the watch page in halshow can be set up to watch a parport much as IO_Show did.

Chapter 8

HAL Components

8.1 Commands and Userspace Components

All of the commands in the following list have man pages. Some will have expanded descriptions, some will have limited descriptions. Also, all of the components listed below have man pages. From these two lists you know what components exist, and you can use *man n name* to get additional information. To view the information in the man page, in a terminal window type:

```
man axis (or perhaps 'man 1 axis' if your system requires it.)
```

axis
AXIS LinuxCNC (The Enhanced Machine Controller) Graphical User Interface.

axis-remote
AXIS Remote Interface.

comp
Build, compile and install LinuxCNC HAL components.

emc
LinuxCNC (The Enhanced Machine Controller).

gladevcp
Virtual Control Panel for LinuxCNC based on Glade, Gtk and HAL widgets.

gs2
HAL userspace component for Automation Direct GS2 VFD's.

halcmd
Manipulate the Enhanced Machine Controller HAL from the command line.

hal_input
Control HAL pins with any Linux input device, including USB HID devices.

halmeter
Observe HAL pins, signals, and parameters.

halrun
Manipulate the Enhanced Machine Controller HAL from the command line.

halsampler
Sample data from HAL in realtime.

halstreamer
Stream file data into HAL in real time.

halui
Observe HAL pins and command LinuxCNC through NML.

io
Accepts NML I/O commands, interacts with HAL in userspace.

iocontrol
Accepts NML I/O commands, interacts with HAL in userspace.

pyvcp
Virtual Control Panel for LinuxCNC.

shuttlexpress
control HAL pins with the ShuttleXpress device made by Contour Design.

8.2 Realtime Components List

Some of these will have expanded descriptions from the man pages. Some will have limited descriptions. All of the components have man pages. From this list you know what components exist and can use *man n name* to get additional information in a terminal window.

Note
If the component requires a floating point thread that is usually the slower servo-thread.

8.2.1 Core LinuxCNC components

motion
Accepts NML motion commands, interacts with HAL in realtime.

axis
Accepts NML motion commands, interacts with HAL in realtime.

classicladder
Realtime software PLC based on ladder logic. See Classic Ladder manual for more information.

gladevcp
Displays Virtual Control Panels built with GTK/Glade.

threads
Creates hard realtime HAL threads.

8.2.2 Logic and bitwise components

and2
Two-input AND gate. For out to be true both inputs must be true.

not
Inverter.

or2
Two-input OR gate.

xor2
Two-input XOR (exclusive OR) gate.

debounce
Filter noisy digital inputs. Description

edge
Edge detector.

flipflop
D type flip-flop.

oneshot
One-shot pulse generator.

logic
General logic function component.

lut5
A 5-input logic function based on a look-up table. Description

match8
8-bit binary match detector.

select8
8-bit binary match detector.

8.2.3 Arithmetic and float-components

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

blend
Perform linear interpolation between two values.

comp
Two input comparator with hysteresis.

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

sum2
Sum of two inputs (each with a gain) and an offset.

counter
Counts input pulses (deprecated). Use the encoder component.

updown
Counts up or down, with optional limits and wraparound behavior.

ddt
Compute the derivative of the input function.

deadzone
Return the center if within the threshold.

hypot
Three-input hypotenuse (Euclidean distance) calculator.

mult2
Product of two inputs.

mux16
Select from one of sixteen input values.

mux2
Select from one of two input values.

mux4
Select from one of four input values.

mux8
Select from one of eight input values.

near
Determine whether two values are roughly equal.

offset
Adds an offset to an input, and subtracts it from the feedback value.

integ
Integrator.

invert
Compute the inverse of the input signal.

wcomp
Window comparator.

weighted_sum
Convert a group of bits to an integer.

biquad
Biquad IIR filter

lowpass
Low-pass filter

limit1
Limit the output signal to fall between min and max. [1]

limit2
Limit the output signal to fall between min and max. Limit its slew rate to less than maxv per second. [2]

limit3
Limit the output signal to fall between min and max. Limit its slew rate to less than maxv per second. Limit its second derivative to less than MaxA per second squared. [3]

maj3
Compute the majority of 3 inputs.

scale
Applies a scale and offset to its input.

8.2.4 Type conversion

conv_bit_s32
Convert a value from bit to s32.

conv_bit_u32
Convert a value from bit to u32.

conv_float_s32
Convert a value from float to s32.

[1] When the input is a position, this means that the *position* is limited.
[2] When the input is a position, this means that *position* and *velocity* are limited.
[3] When the input is a position, this means that the *position*, *velocity*, and *acceleration* are limited.

conv_float_u32
Convert a value from float to u32.

conv_s32_bit
Convert a value from s32 to bit.

conv_s32_float
Convert a value from s32 to float.

conv_s32_u32
Convert a value from s32 to u32.

conv_u32_bit
Convert a value from u32 to bit.

conv_u32_float
Convert a value from u32 to float.

conv_u32_s32
Convert a value from u32 to s32.

8.2.5 Hardware drivers

hm2_7i43
HAL driver for the Mesa Electronics 7i43 EPP Anything IO board with HostMot2.

hm2_pci
HAL driver for the Mesa Electronics 5i20, 5i22, 5i23, 4i65, and 4i68 Anything I/O boards, with HostMot2 firmware.

hostmot2
HAL driver for the Mesa Electronics HostMot2 firmware.

mesa_7i65
Support for the Mesa 7i65 eight-axis servo card.

pluto_servo
Hardware driver and firmware for the Pluto-P parallel-port FPGA, for use with servos.

pluto_step
Hardware driver and firmware for the Pluto-P parallel-port FPGA, for use with steppers.

thc
Torch Height Control using a Mesa THC card.

serport
Hardware driver for the digital I/O bits of the 8250 and 16550 serial port.

8.2.6 Kinematics

kins
kinematics definitions for LinuxCNC.

gantrykins
A kinematics module that maps one axis to multiple joints.

genhexkins
Gives six degrees of freedom in position and orientation (XYZABC). The location of the motors is defined at compile time.

genserkins
Kinematics that can model a general serial-link manipulator with up to 6 angular joints.

maxkins
Kinematics for a tabletop 5 axis mill named *max* with tilting head (B axis) and horizontal rotary mounted to the table (C axis). Provides UVW motion in the rotated coordinate system. The source file, maxkins.c, may be a useful starting point for other 5-axis systems.

tripodkins
The joints represent the distance of the controlled point from three predefined locations (the motors), giving three degrees of freedom in position (XYZ).

trivkins
There is a 1:1 correspondence between joints and axes. Most standard milling machines and lathes use the trivial kinematics module.

pumakins
Kinematics for PUMA-style robots.

rotatekins
The X and Y axes are rotated 45 degrees compared to the joints 0 and 1.

scarakins
Kinematics for SCARA-type robots.

8.2.7 Motor control

at_pid
Proportional/integral/derivative controller with auto tuning.

pid
Proportional/integral/derivative controller. Description

pwmgen
Software PWM/PDM generation. Description

encoder
Software counting of quadrature encoder signals. Description.

stepgen
Software step pulse generation. Description.

freqgen
Software step pulse generation.

8.2.8 BLDC and 3-phase motor control

bldc_hall3
3-wire Bipolar trapezoidal commutation BLDC motor driver using Hall sensors.

clarke2
Two input version of Clarke transform.

clarke3
Clarke (3 phase to cartesian) transform.

clarkeinv
Inverse Clarke transform.

8.2.9 Other

charge_pump
Creates a square-wave for the *charge pump* input of some controller boards. The *Charge Pump* should be added to the base thread function. When enabled the output is on for one period and off for one period. To calculate the frequency of the output 1/(period time in seconds x 2) = hz. For example if you have a base period of 100,000ns that is 0.0001 seconds and the formula would be 1/(0.0001 x 2) = 5,000 hz or 5 Khz.

encoder_ratio
An electronic gear to synchronize two axes.

estop_latch
ESTOP latch.

feedcomp
Multiply the input by the ratio of current velocity to the feed rate.

gearchange
Select from one of two speed ranges.

ilowpass
While it may find other applications, this component was written to create smoother motion while jogging with an MPG.

In a machine with high acceleration, a short jog can behave almost like a step function. By putting the ilowpass component between the MPG encoder counts output and the axis jog-counts input, this can be smoothed.

Choose scale conservatively so that during a single session there will never be more than about 2e9/scale pulses seen on the MPG. Choose gain according to the smoothing level desired. Divide the axis.N.jog-scale values by scale.

joyhandle
Sets nonlinear joypad movements, deadbands and scales.

knob2float
Convert counts (probably from an encoder) to a float value.

minmax
Track the minimum and maximum values of the input to the outputs.

sample_hold
Sample and Hold.

sampler
Sample data from HAL in real time.

siggen
Signal generator. Description.

sim_encoder
Simulated quadrature encoder. Description.

sphereprobe
Probe a pretend hemisphere.

steptest
Used by Stepconf to allow testing of acceleration and velocity values for an axis.

streamer
Stream file data into HAL in real time.

supply
Set output pins with values from parameters (deprecated).

threadtest
Component for testing thread behavior.

time
Accumulated run-time timer counts HH:MM:SS of *active* input.

timedelay
The equivalent of a time-delay relay.

timedelta
Component that measures thread scheduling timing behavior.

toggle2nist
Toggle button to nist logic.

toggle
Push-on, push-off from momentary pushbuttons.

tristate_bit
Place a signal on an I/O pin only when enabled, similar to a tristate buffer in electronics.

tristate_float
Place a signal on an I/O pin only when enabled, similar to a tristate buffer in electronics.

watchdog
Monitor one to thirty-two inputs for a *heartbeat*.

8.3 HAL API calls

```
hal_add_funct_to_thread.3hal
hal_bit_t.3hal
hal_create_thread.3hal
hal_del_funct_from_thread.3hal
hal_exit.3hal
hal_export_funct.3hal
hal_float_t.3hal
hal_get_lock.3hal
hal_init.3hal
hal_link.3hal
hal_malloc.3hal
hal_param_bit_new.3hal
hal_param_bit_newf.3hal
hal_param_float_new.3hal
hal_param_float_newf.3hal
hal_param_new.3hal
hal_param_s32_new.3hal
hal_param_s32_newf.3hal
hal_param_u32_new.3hal
hal_param_u32_newf.3hal
hal_parport.3hal
hal_pin_bit_new.3hal
hal_pin_bit_newf.3hal
hal_pin_float_new.3hal
hal_pin_float_newf.3hal
hal_pin_new.3hal
hal_pin_s32_new.3hal
hal_pin_s32_newf.3hal
hal_pin_u32_new.3hal
hal_pin_u32_newf.3hal
hal_ready.3hal
hal_s32_t.3hal
```

hal_set_constructor.3hal
hal_set_lock.3hal
hal_signal_delete.3hal
hal_signal_new.3hal
hal_start_threads.3hal
hal_type_t.3hal
hal_u32_t.3hal
hal_unlink.3hal
intro.3hal
undocumented.3hal

8.4 RTAPI calls

EXPORT_FUNCTION.3rtapi
MODULE_AUTHOR.3rtapi
MODULE_DESCRIPTION.3rtapi
MODULE_LICENSE.3rtapi
RTAPI_MP_ARRAY_INT.3rtapi
RTAPI_MP_ARRAY_LONG.3rtapi
RTAPI_MP_ARRAY_STRING.3rtapi
RTAPI_MP_INT.3rtapi
RTAPI_MP_LONG.3rtapi
RTAPI_MP_STRING.3rtapi
intro.3rtapi
rtapi_app_exit.3rtapi
rtapi_app_main.3rtapi
rtapi_clock_set_period.3rtapi
rtapi_delay.3rtapi
rtapi_delay_max.3rtapi
rtapi_exit.3rtapi
rtapi_get_clocks.3rtapi
rtapi_get_msg_level.3rtapi
rtapi_get_time.3rtapi
rtapi_inb.3rtapi
rtapi_init.3rtapi
rtapi_module_param.3rtapi
RTAPI_MP_ARRAY_INT.3rtapi
RTAPI_MP_ARRAY_LONG.3rtapi
RTAPI_MP_ARRAY_STRING.3rtapi
RTAPI_MP_INT.3rtapi
RTAPI_MP_LONG.3rtapi
RTAPI_MP_STRING.3rtapi
rtapi_mutex.3rtapi
rtapi_outb.3rtapi
rtapi_print.3rtap
rtapi_prio.3rtapi
rtapi_prio_highest.3rtapi
rtapi_prio_lowest.3rtapi
rtapi_prio_next_higher.3rtapi
rtapi_prio_next_lower.3rtapi
rtapi_region.3rtapi
rtapi_release_region.3rtapi
rtapi_request_region.3rtapi
rtapi_set_msg_level.3rtapi
rtapi_shmem.3rtapi

```
rtapi_shmem_delete.3rtapi
rtapi_shmem_getptr.3rtapi
rtapi_shmem_new.3rtapi
rtapi_snprintf.3rtapi
rtapi_task_delete.3rtpi
rtapi_task_new.3rtapi
rtapi_task_pause.3rtapi
rtapi_task_resume.3rtapi
rtapi_task_start.3rtapi
rtapi_task_wait.3rtapi
```

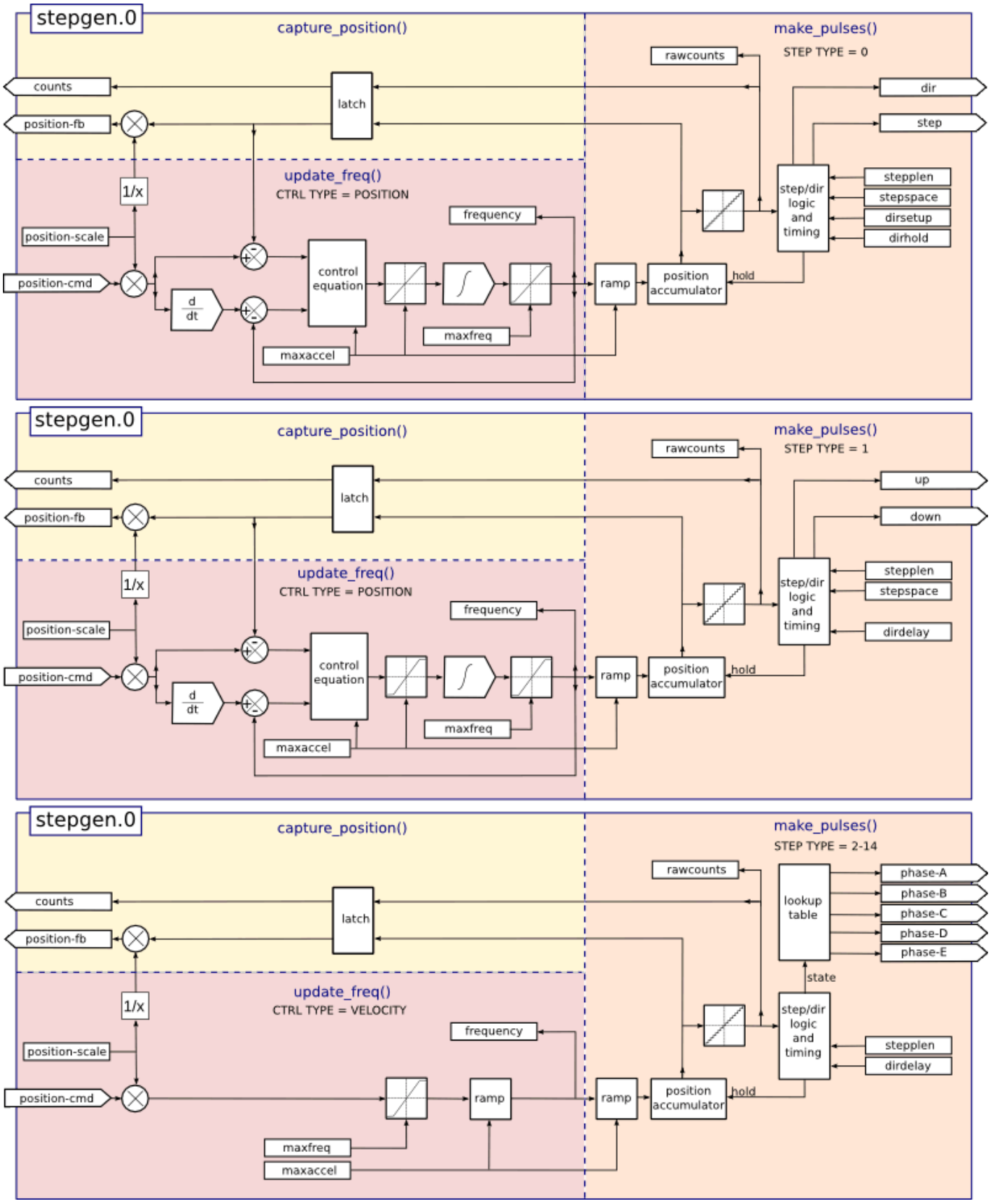
Chapter 9

HAL Component Descriptions

9.1 Stepgen

This component provides software based generation of step pulses in response to position or velocity commands. In position mode, it has a built in pre-tuned position loop, so PID tuning is not required. In velocity mode, it drives a motor at the commanded speed, while obeying velocity and acceleration limits. It is a realtime component only, and depending on CPU speed, etc, is capable of maximum step rates of 10kHz to perhaps 50kHz. Figure Step Pulse Generator Block Diagram shows three block diagrams, each is a single step pulse generator. The first diagram is for step type *0*, (step and direction). The second is for step type *1* (up/down, or pseudo-PWM), and the third is for step types 2 through 14 (various stepping patterns). The first two diagrams show position mode control, and the third one shows velocity mode. Control mode and step type are set independently, and any combination can be selected.

Step Pulse Generator Block Diagram position mode



Installing

```
halcmd: loadrt stepgen step_type=<type-array> [ctrl_type=<ctrl_array>]
```

<*type-array*> is a series of comma separated decimal integers. Each number causes a single step pulse generator to be loaded, the value of the number determines the stepping type. <*ctrl_array*> is a comma separated series of *p* or *v* characters, to specify position or velocity mode. *ctrl_type* is optional, if ommitted, all of the step generators will be position mode.

For example:

```
halcmd: loadrt stepgen step_type=0,0,2 ctrl_type=p,p,v
```

will install three step generators. The first two use step type *0* (step and direction) and run in position mode. The last one uses step type *2* (quadrature) and runs in velocity mode. The default value for <*config-array*> is *0,0,0* which will install three type *0* (step/dir) generators. The maximum number of step generators is 8 (as defined by MAX_CHAN in stepgen.c). Each generator is independent, but all are updated by the same function(s) at the same time. In the following descriptions, <*chan*> is the number of a specific generator. The first generator is number 0. .Removing

```
halcmd: unloadrt stepgen
```

Pins Each step pulse generator will have only some of these pins, depending on the step type and control type selected.

- *(float) stepgen.<chan>.position-cmd* - Desired motor position, in position units (position mode only).
- *(float) stepgen.<chan>.velocity-cmd* - Desired motor velocity, in position units per second (velocity mode only).
- *(s32) stepgen.<chan>.counts* - Feedback position in counts, updated by *capture_position()*.
- *(float) stepgen.<chan>.position-fb* - Feedback position in position units, updated by *capture_position()*.
- *(bit) stepgen.<chan>.enable* - Enables output steps - when false, no steps are generated.
- *(bit) stepgen.<chan>.step* - Step pulse output (step type 0 only).
- *(bit) stepgen.<chan>.dir* - Direction output (step type 0 only).
- *(bit) stepgen.<chan>.up* - UP pseudo-PWM output (step type 1 only).
- *(bit) stepgen.<chan>.down* - DOWN pseudo-PWM output (step type 1 only).
- *(bit) stepgen.<chan>.phase-A* - Phase A output (step types 2-14 only).
- *(bit) stepgen.<chan>.phase-B* - Phase B output (step types 2-14 only).
- *(bit) stepgen.<chan>.phase-C* - Phase C output (step types 3-14 only).
- *(bit) stepgen.<chan>.phase-D* - Phase D output (step types 5-14 only).
- *(bit) stepgen.<chan>.phase-E* - Phase E output (step types 11-14 only).

PARAMETERS

- *(float) stepgen.<chan>.position-scale* - Steps per position unit. This parameter is used for both output and feedback.
- *(float) stepgen.<chan>.maxvel* - Maximum velocity, in position units per second. If 0.0, has no effect.
- *(float) stepgen.<chan>.maxaccel* - Maximum accel/decel rate, in positions units per second squared. If 0.0, has no effect.
- *(float) stepgen.<chan>.frequency* - The current step rate, in steps per second.
- *(float) stepgen.<chan>.steplen* - Length of a step pulse (step type 0 and 1) or minimum time in a given state (step types 2-14), in nano-seconds.
- *(float) stepgen.<chan>.stepspace* - Minimum spacing between two step pulses (step types 0 and 1 only), in nano-seconds. Set to 0 to enable the stepgen *doublefreq* function. To use *doublefreq* the parport reset function must be enabled.
- *(float) stepgen.<chan>.dirsetup* - Minimum time from a direction change to the beginning of the next step pulse (step type 0 only), in nanoseconds.

- *(float) stepgen.<chan>.dirhold* - Minmum time from the end of a step pulse to a direction change (step type 0 only), in nanoseconds.
- *(float) stepgen.<chan>.dirdelay* - Minmum time any step to a step in the opposite direction (step types 1-14 only), in nanoseconds.
- *(s32) stepgen.<chan>.rawcounts* - The raw feedback count, updated by *make_pulses()*.

In position mode, the values of maxvel and maxaccel are used by the internal position loop to avoid generating step pulse trains that the motor cannot follow. When set to values that are appropriate for the motor, even a large instantaneous change in commanded position will result in a smooth trapezoidal move to the new location. The algorithm works by measuring both position error and velocity error, and calculating an acceleration that attempts to reduce both to zero at the same time. For more details, including the contents of the *control equation* box, consult the code.

In velocity mode, maxvel is a simple limit that is applied to the commanded velocity, and maxaccel is used to ramp the actual frequency if the commanded velocity changes abruptly. As in position mode, proper values for these parameters ensure that the motor can follow the generated pulse train.

Step Type 0

Step type 0 is the standard step and direction type. When configured for step type 0, there are four extra parameters that determine the exact timing of the step and direction signals. In the following figure the meaning of these parameters is shown. The parameters are in nanoseconds, but will be rounded up to an integer multiple of the thread period for the threaed that calls *make_pulses()*. For example, if *make_pulses()* is called every 16 us, and steplen is 20000, then the step pulses will be 2 x 16 = 32 us long. The default value for all four of the parameters is 1ns, but the automatic rounding takes effect the first time the code runs. Since one step requires *steplen* ns high and *stepspace* ns low, the maximum frequency is 1,000,000,000 divided by *(steplen+stepspace)*. If *maxfreq* is set higher than that limit, it will be lowered automatically. If maxfreq is zero, it will remain zero, but the output frequency will still be limited.

When using the parallel port driver the step frequency can be doubled using the parport reset function together with stepgen's *doublefreq* setting.

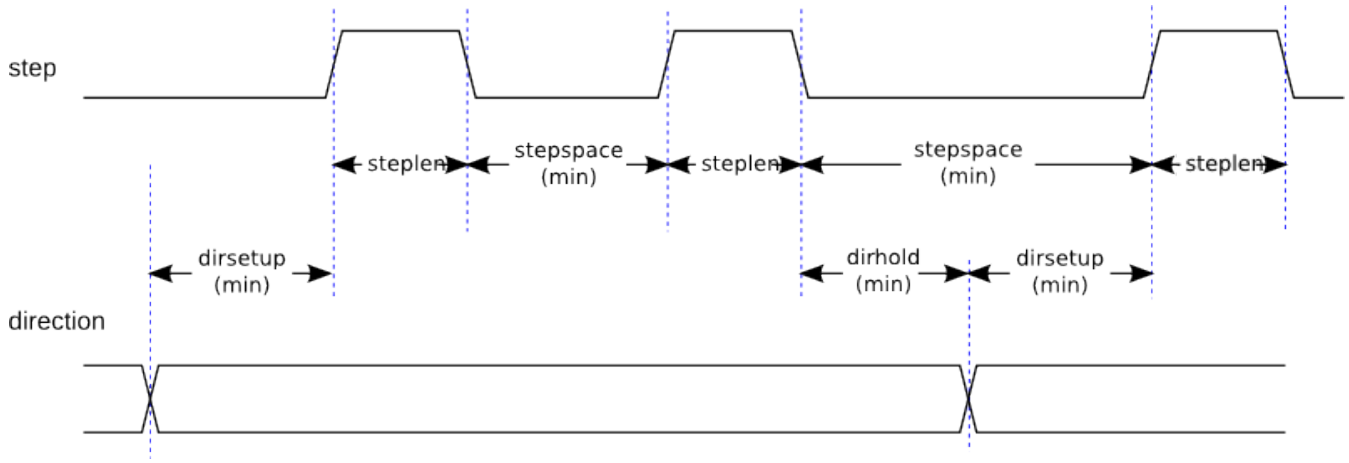


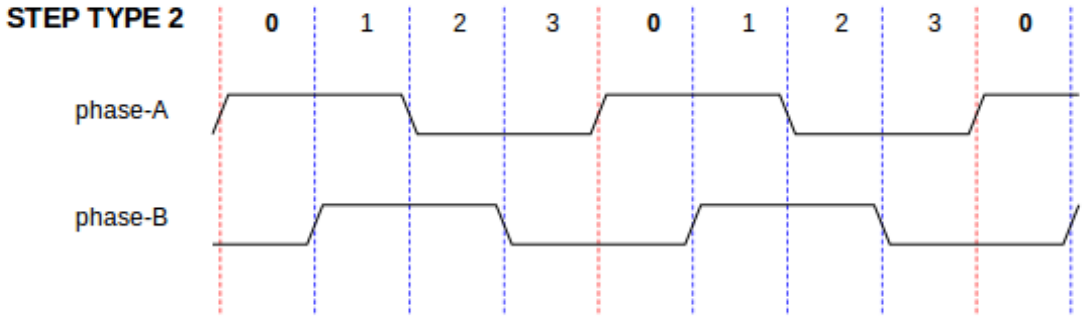
Figure 9.1: Step and Direction Timing

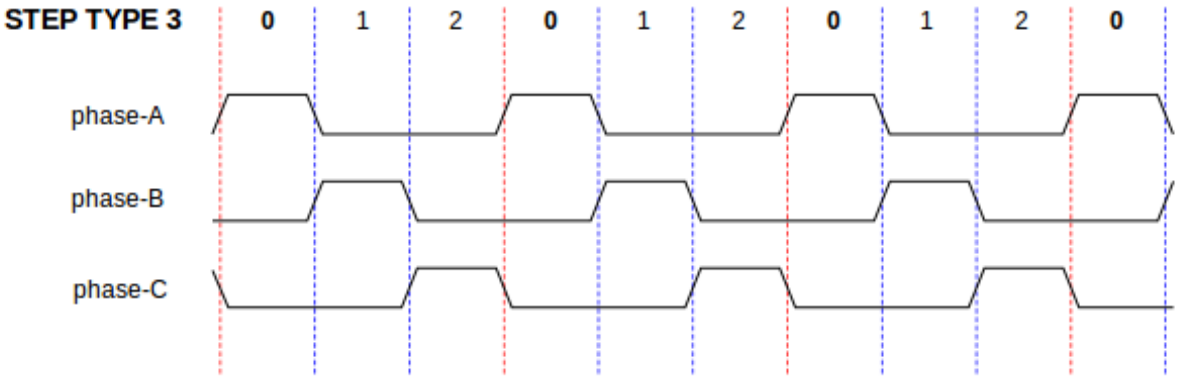
Step Type 1 Step type 1 has two outputs, up and down. Pulses appear on one or the other, depending on the direction of travel. Each pulse is *steplen* ns long, and the pulses are separated by at least *stepspace* ns. The maximum frequency is the same as for step type 0. If *maxfreq* is set higher than the limit it will be lowered. If *maxfreq* is zero, it will remain zero but the output frequency will still be limited.

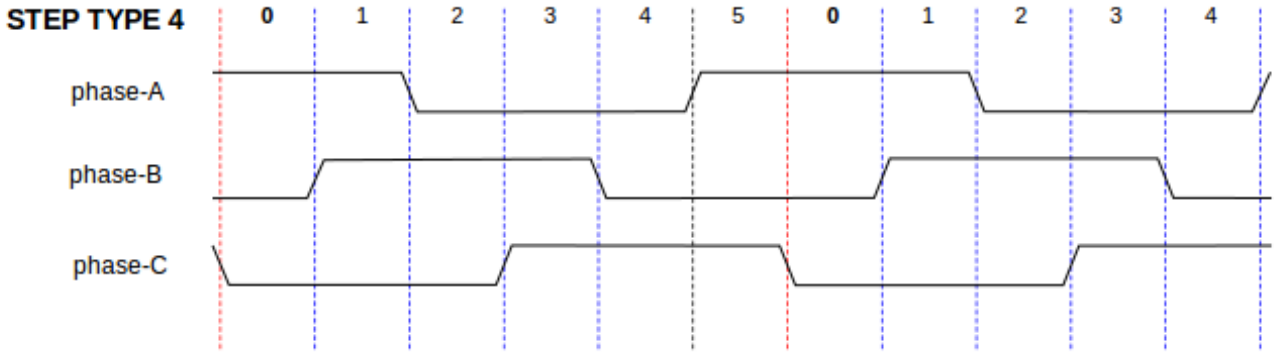
Warning
Do not use the parport reset function with step types 2 - 14. Unexpected results can happen.

Step Type 2 - 14 Step types 2 through 14 are state based, and have from two to five outputs. On each step, a state counter is incremented or decremented. Figures Two-and-Three-Phase, Four-Phase, and Five-Phase show the output patterns as a function of the state counter. The maximum frequency is 1,000,000,000 divided by *steplen*, and as in the other modes, *maxfreq* will be lowered if it is above the limit.

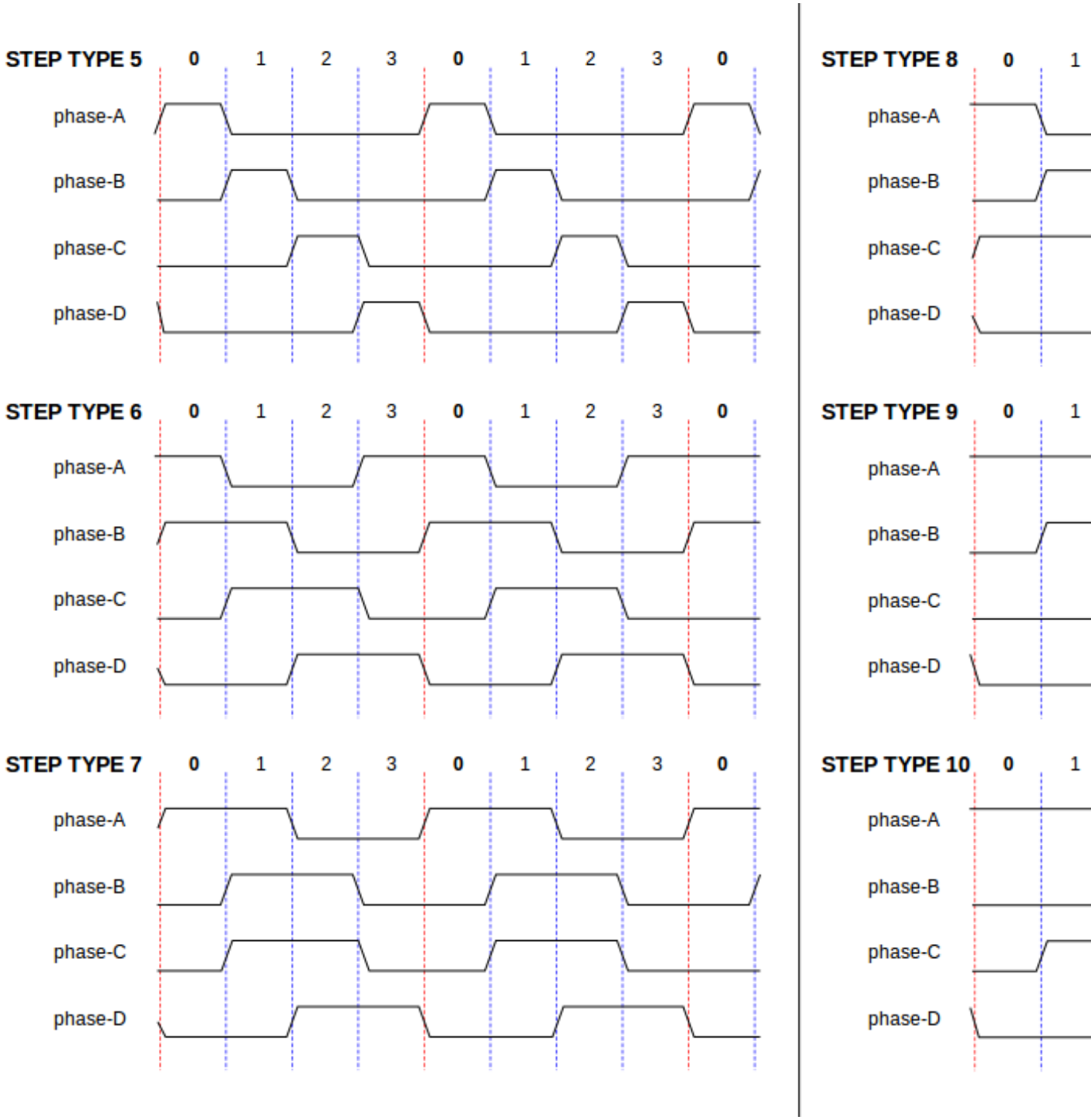
Two-and-Three-Phase Step Types



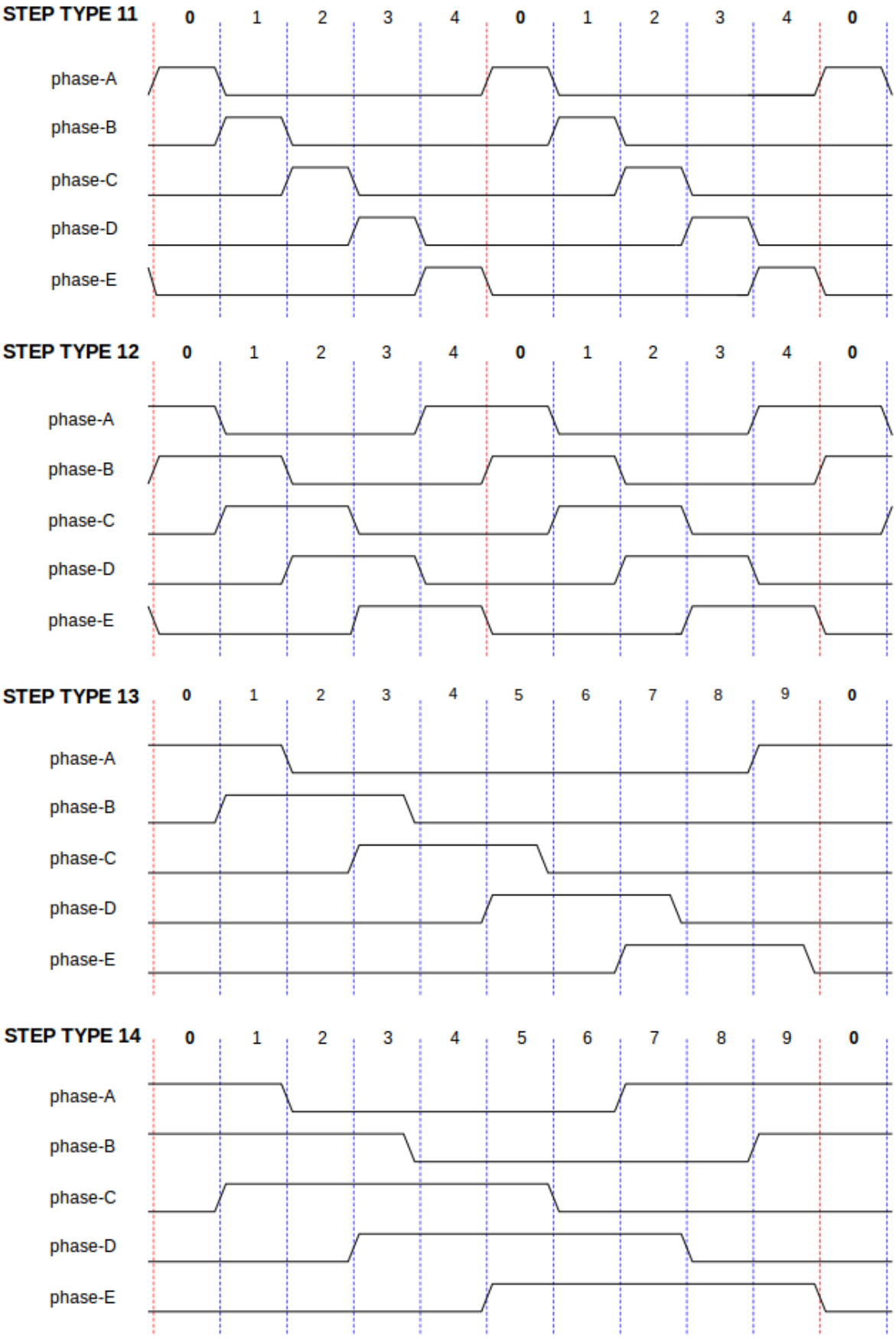




Four-Phase Step Types



Five-Phase Step Types


STEP TYPE 11
0 1 2 3 4 0 1 2 3 4 0
phase-A
phase-B
phase-C
phase-D
phase-E
STEP TYPE 12
0 1 2 3 4 0 1 2 3 4 0
phase-A
phase-B
phase-C
phase-D
phase-E
STEP TYPE 13
0 1 2 3 4 5 6 7 8 9 0
phase-A
phase-B
phase-C
phase-D
phase-E
STEP TYPE 14
0 1 2 3 4 5 6 7 8 9 0
phase-A
phase-B
phase-C
phase-D
phase-E

Functions The component exports three functions. Each function acts on all of the step pulse generators - running different generators in different threads is not supported.

- *(funct) stepgen.make-pulses* - High speed function to generate and count pulses (no floating point).
- *(funct) stepgen.update-freq* - Low speed function does position to velocity conversion, scaling and limiting.
- *(funct) stepgen.capture-position* - Low speed function for feedback, updates latches and scales position.

The high speed function *stepgen.make-pulses* should be run in a very fast thread, from 10 to 50 us depending on the capabilities of the computer. That thread's period determines the maximum step frequency, since *steplen*, *stepspace*, *dirsetup*, *dirhold*, and *dirdelay* are all rounded up to a integer multiple of the thread periond in nanoseconds. The other two functions can be called at a much lower rate.

9.2 PWMgen

This component provides software based generation of PWM (Pulse Width Modulation) and PDM (Pulse Density Modulation) waveforms. It is a realtime component only, and depending on CPU speed, etc, is capable of PWM frequencies from a few hundred Hertz at pretty good resolution, to perhaps 10KHz with limited resolution.

Installing

```
loadrt pwmgen output_type=<config-array>
```

The <*config-array*> is a series of comma separated decimal integers. Each number causes a single PWM generator to be loaded, the value of the number determines the output type. The following example will install three PWM generators. There is no default value, if <*config-array*> is not specified, no PWM generators will be installed. The maximum number of frequency generators is 8 (as defined by MAX_CHAN in pwmgen.c). Each generator is independent, but all are updated by the same function(s) at the same time. In the following descriptions, <*chan*> is the number of a specific generator. The first generator is number 0.

Example

```
loadrt pwmgen output_type=0,1,2
```

Removing

```
unloadrt pwmgen
```

Output Types The PWM generator supports three different *output types*.

- *Output type 0* - PWM output pin only. Only positive commands are accepted, negative values are treated as zero (and will be affected by the parameter *min-dc* if it is non-zero).
- *Output type 1* - PWM/PDM and direction pins. Positive and negative inputs will be output as positive and negative PWM. The direction pin is false for positive commands, and true for negative commands. If your control needs positive PWM for both CW and CCW use the abs component to convert your PWM signal to positive value when a negative input is input.
- *Output type 2* - UP and DOWN pins. For positive commands, the PWM signal appears on the up output, and the down output remains false. For negative commands, the PWM signal appears on the down output, and the up output remains false. Output type 2 is suitable for driving most H-bridges.

Pins Each PWM generator will have the following pins:

- *(float) pwmgen.<chan>.value* - Command value, in arbitrary units. Will be scaled by the *scale* parameter (see below).
- *(bit) pwmgen.<chan>.enable* - Enables or disables the PWM generator outputs.

Each PWM generator will also have some of these pins, depending on the output type selected:

- *(bit) pwmgen.<chan>.pwm* - PWM (or PDM) output, (output types 0 and 1 only).
- *(bit) pwmgen.<chan>.dir* - Direction output (output type 1 only).
- *(bit) pwmgen.<chan>.up* - PWM/PDM output for positive input value (output type 2 only).
- *(bit) pwmgen.<chan>.down* - PWM/PDM output for negative input value (output type 2 only).

PARAMETERS

- *(float) pwmgen.<chan>.scale* - Scaling factor to convert *value* from arbitrary units to duty cycle.
- *(float) pwmgen.<chan>.pwm-freq* - Desired PWM frequency, in Hz. If 0.0, generates PDM instead of PWM. If set higher than internal limits, next call of *update_freq()* will set it to the internal limit. If non-zero, and *dither* is false, next call of *update_freq()* will set it to the nearest integer multiple of the *make_pulses()* function period.
- *(bit) pwmgen.<chan>.dither-pwm* - If true, enables dithering to achieve average PWM frequencies or duty cycles that are unobtainable with pure PWM. If false, both the PWM frequency and the duty cycle will be rounded to values that can be achieved exactly.
- *(float) pwmgen.<chan>.min-dc* - Minimum duty cycle, between 0.0 and 1.0 (duty cycle will go to zero when disabled, regardless of this setting).
- *(float) pwmgen.<chan>.max-dc* - Maximum duty cycle, between 0.0 and 1.0.
- *(float) pwmgen.<chan>.curr-dc* - Current duty cycle - after all limiting and rounding (read only).

Functions The component exports two functions. Each function acts on all of the PWM generators - running different generators in different threads is not supported.

- *(funct) pwmgen.make-pulses* - High speed function to generate PWM waveforms (no floating point).
- *(funct) pwmgen.update* - Low speed function to scale and limit value and handle other parameters.

The high speed function *pwmgen.make-pulses* should be run in a very fast thread, from 10 to 50 us depending on the capabilities of the computer. That thread's period determines the maximum PWM carrier frequency, as well as the resolution of the PWM or PDM signals. The other function can be called at a much lower rate.

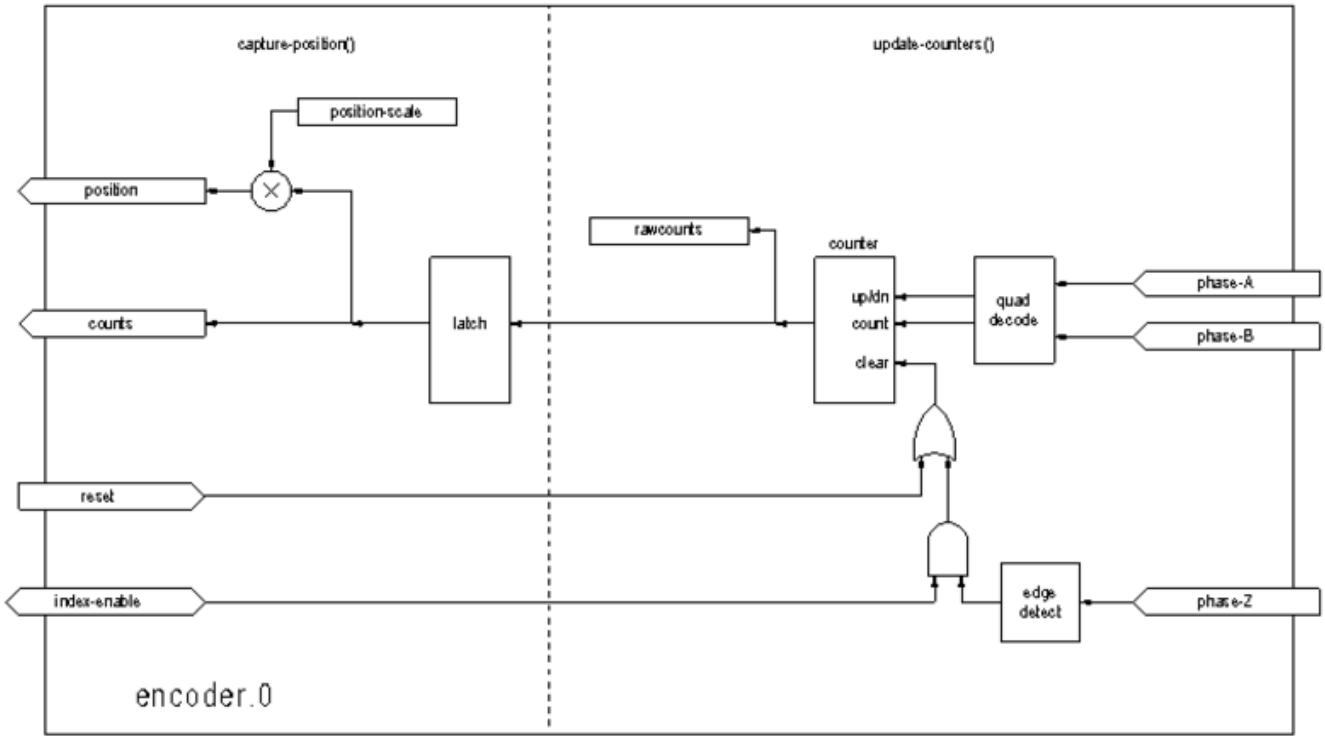
9.3 Encoder

This component provides software based counting of signals from quadrature encoders. It is a realtime component only, and depending on CPU speed, latency, etc, is capable of maximum count rates of 10kHz to perhaps up to 50kHz.

The base thread should be 1/2 count speed to allow for noise and timing variation. For example if you have a 100 pulse per revolution encoder on the spindle and your maximnum RPM is 3000 the maximum base thread should be 25 us. A 100 pulse per revolution encoder will have 400 counts. The spindle speed of 3000 RPM = 50 RPS (revolutions per second). 400 * 50 = 20,000 counts per second or 50 us between counts.

Figure Encoder Counter Block Diagram is a block diagram of one channel of encoder counter.

Encoder Counter Block Diagram



Installing

```
halcmd: loadrt encoder [num_chan=<counters>]
```

<*counters*> is the number of encoder counters that you want to install. If *numchan* is not specified, three counters will be installed. The maximum number of counters is 8 (as defined by MAX_CHAN in encoder.c). Each counter is independent, but all are updated by the same function(s) at the same time. In the following descriptions, <*chan*> is the number of a specific counter. The first counter is number 0.

Removing

```
halcmd: unloadrt encoder
```

PINS

- *encoder.*<*chan*>*.counter-mode* (bit, I/O) (default: FALSE) - 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, it counts in quadrature mode.
- *encoder.*<*chan*>*.counts* (s32, Out) - Position in encoder counts.
- *encoder.*<*chan*>*.counts-latched* (s32, Out) - Not used at this time.
- *encoder.*<*chan*>*.index-enable* (bit, I/O) - When True, *counts* and *position are* reset to zero on next rising edge of Phase Z. At the same time, *index-enable* is reset to zero to indicate that the rising edge has occoured. The *index-enable* pin is bi-directional. If *index-enable* is False, the Phase Z channel of the encoder will be ignored, and the counter will count normally. The encoder driver will never set *index-enable* True. However, some other component may do so.

- *encoder.<chan>.latch-falling* (bit, In) (default: TRUE) - Not used at this time.
- *encoder.<chan>.latch-input* (bit, In) (default: TRUE) - Not used at this time.
- *encoder.<chan>.latch-rising* (bit, In) - Not used at this time.
- *encoder.<chan>.min-speed-estimate* (float, in) - Determine the minimum true velocity magnitude at which velocity will be estimated as nonzero and postition-interpolated will be interpolated. The units of *min-speed-estimate* are the same as the units of *velocity* . Scale factor, in counts per length unit. Setting this parameter too low will cause it to take a long time for velocity to go to 0 after encoder pulses have stopped arriving.
- *encoder.<chan>.phase-A* (bit, In) - Phase A of the quadrature encoder signal.
- *encoder.<chan>.phase-B* (bit, In) - Phase B of the quadrature encoder signal.
- *encoder.<chan>.phase-Z* (bit, In) - Phase Z (index pulse) of the quadrature encoder signal.
- *encoder.<chan>.position* (float, Out) - Position in scaled units (see *position-scale*).
- *encoder.<chan>.position-interpolated* (float, Out) - Position in scaled units, interpolated between encoder counts. The *position-interpolated* attempts to interpolate between encoder counts, based on the most recently measured velocity. Only valid when velocity is approximately constant and above *min-speed-estimate*. Do not use for position control, since its value is incorrect at low speeds, during direction reversals, and during speed changes. However, it allows a low ppr encoder (including a one pulse per revolution *encoder*) to be used for lathe threading, and may have other uses as well.
- *encoder.<chan>.position-latched (float, Out)* - Not used at this time.
- *encoder.<chan>.position-scale (float, I/O)* - 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.<chan>.rawcounts (s32, In)* - 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.
- *encoder.<chan>.reset* (bit, In) - When True, force *counts* and *position* to zero immediately.
- *encoder.<chan>.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. When the magnitude of the true velocity is below min-velocity-estimate, the velocity output is 0.
- *encoder.<chan>.x4-mode (bit, I/O) (default: TRUE)* - Enables times-4 mode. When true, 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. The 1x mode is useful for some jogwheels.

PARAMETERS

- *encoder.<chan>.capture-position.time (s32, RO)*
- *encoder.<chan>.capture-position.tmax (s32, RW)*
- *encoder.<chan>.update-counters.time (s32, RO)*
- *encoder.<chan>.update-counter.tmax (s32, RW)*

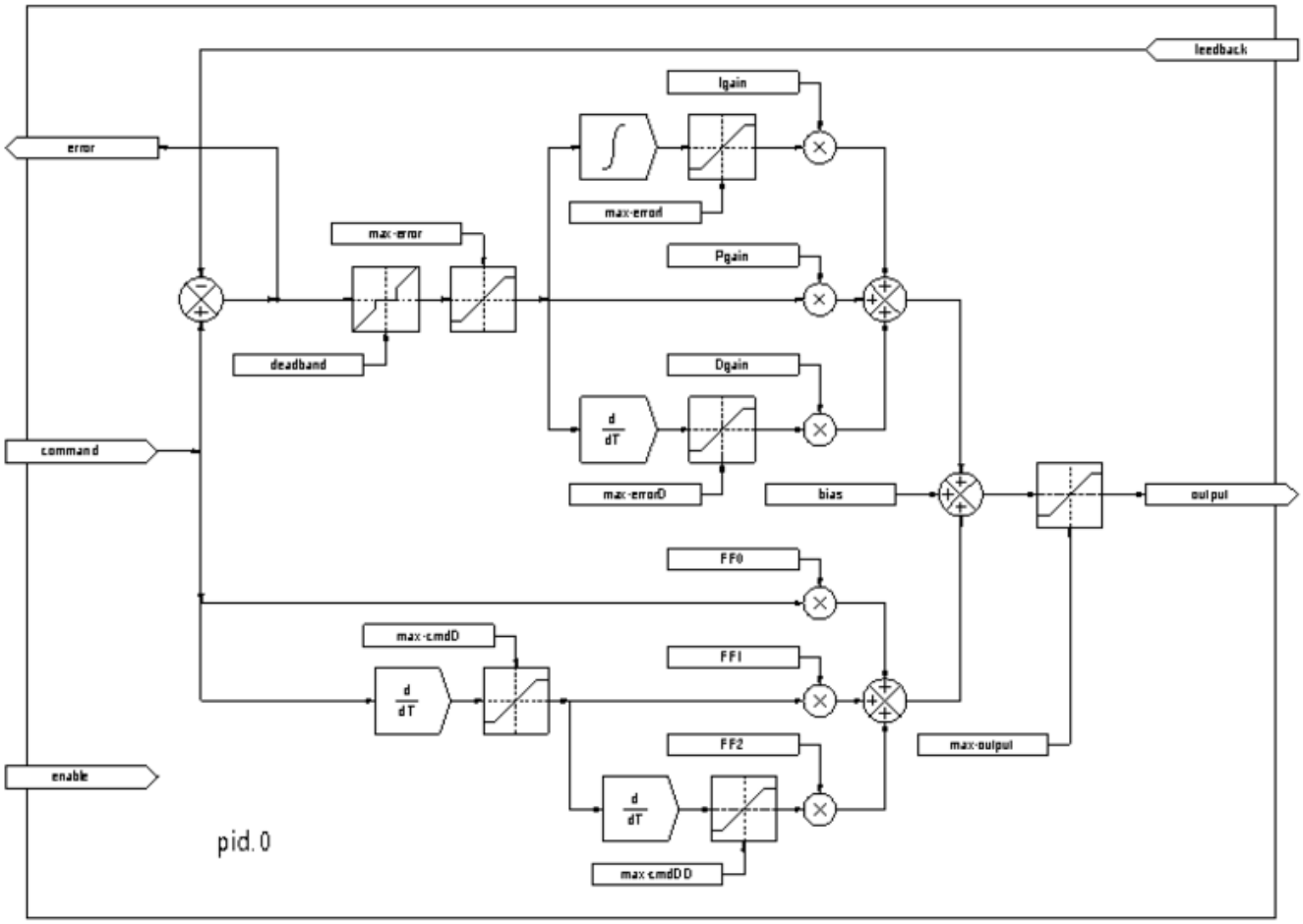
Functions The component exports two functions. Each function acts on all of the encoder counters - running different counters in different threads is not supported.

- *(funct) encoder.update-counters* - High speed function to count pulses (no floating point).
- *(funct) encoder.capture-position* - Low speed function to update latches and scale position.

9.4 PID

This component provides Proportional/Integral/Derivative control loops. It is a realtime component only. For simplicity, this discussion assumes that we are talking about position loops, however this component can be used to implement other feedback loops such as speed, torch height, temperature, etc. Figure PID Loop Block Diagram is a block diagram of a single PID loop.

PID Loop Block Diagram



Installing

```
halcmd: loadrt pid [num_chan=<loops>] [debug=1]
```

<*loops*> is the number of PID loops that you want to install. If *numchan* is not specified, one loop will be installed. The maximum number of loops is 16 (as defined by MAX_CHAN in pid.c). Each loop is completely independent. In the following descriptions, <*loopnum*> is the loop number of a specific loop. The first loop is number 0.

If *debug=1* is specified, the component will export a few extra parameters that may be useful during debugging and tuning. By default, the extra parameters are not exported, to save shared memory space and avoid cluttering the parameter list.

Removing

```
halcmd: unloadrt pid
```

Pins The three most important pins are

- *(float) pid.<loopnum>.command* - The desired position, as commanded by another system component.
- *(float) pid.<loopnum>.feedback* - The present position, as measured by a feedback device such as an encoder.
- *(float) pid.<loopnum>.output* - A velocity command that attempts to move from the present position to the desired position.

For a position loop, *command* and *feedback* are in position units. For a linear axis, this could be inches, mm, meters, or whatever is relevant. Likewise, for an angular axis, it could be degrees, radians, etc. The units of the *output* pin represent the change needed to make the feedback match the command. As such, for a position loop *Output* is a velocity, in inches/sec, mm/sec, degrees/sec, etc. Time units are always seconds, and the velocity units match the position units. If command and feedback are in meters, then output is in meters per second.

Each loop has two pins which are used to monitor or control the general operation of the component.

- *(float) pid.<loopnum>.error* - Equals *.command* minus *.feedback*.
- *(bit) pid.<loopnum>.enable* - A bit that enables the loop. If *.enable* is false, all integrators are reset, and the output is forced to zero. If *.enable* is true, the loop operates normally.

Pins used to report saturation. Saturation occurs when the output of the PID block is at its maximum or minimum limit.

- *(bit) pid.<loopnum>.saturated* - True when output is saturated.
- *(float) pid.<loopnum>.saturated_s* - The time the output has been saturated.
- *(s32) pid.<loopnum>.saturated_count* - The time the output has been saturated.

Parameters The PID gains, limits, and other *tunable* features of the loop are implemented as parameters.

- *(float) pid.<loopnum>.Pgain* - Proportional gain
- *(float) pid.<loopnum>.Igain* - Integral gain
- *(float) pid.<loopnum>.Dgain* - Derivative gain
- *(float) pid.<loopnum>.bias* - Constant offset on output
- *(float) pid.<loopnum>.FF0* - Zeroth order feedforward - output proportional to command (position).
- *(float) pid.<loopnum>.FF1* - First order feedforward - output proportional to derivative of command (velocity).
- *(float) pid.<loopnum>.FF2* - Second order feedforward - output proportional to 2nd derivative of command (acceleration)[1].
- *(float) pid.<loopnum>.deadband* - Amount of error that will be ignored
- *(float) pid.<loopnum>.maxerror* - Limit on error
- *(float) pid.<loopnum>.maxerrorI* - Limit on error integrator
- *(float) pid.<loopnum>.maxerrorD* - Limit on error derivative
- *(float) pid.<loopnum>.maxcmdD* - Limit on command derivative
- *(float) pid.<loopnum>.maxcmdDD* - Limit on command 2nd derivative
- *(float) pid.<loopnum>.maxoutput* - Limit on output value

All of the *max* limits are implemented such that if the parameter value is zero, there is no limit.

If *debug=1* was specified when the component was installed, four additional parameters will be exported:

- *(float) pid.<loopnum>.errorI* - Integral of error.

[1] FF2 is not currently implemented, but it will be added. Consider this note a “FIXME” for the code

- *(float) pid.<loopnum>.errorD* - Derivative of error.
- *(float) pid.<loopnum>.commandD* - Derivative of the command.
- *(float) pid.<loopnum>.commandDD* - 2nd derivative of the command.

Functions The component exports one function for each PID loop. This function performs all the calculations needed for the loop. Since each loop has its own function, individual loops can be included in different threads and execute at different rates.

- *(funct) pid.<loopnum>.do_pid_calcs* - Performs all calculations for a single PID loop.

If you want to understand the exact algorithm used to compute the output of the PID loop, refer to figure PID Loop Block Diagram, the comments at the beginning of *emc2/src/hal/components/pid.c* , and of course to the code itself. The loop calculations are in the C function *calc_pid()*.

9.5 Simulated Encoder

The simulated encoder is exactly that. It produces quadrature pulses with an index pulse, at a speed controlled by a HAL pin. Mostly useful for testing.

Installing

```
halcmd: loadrt sim-encoder num_chan=<number>
```

<number> is the number of encoders that you want to simulate. If not specified, one encoder will be installed. The maximum number is 8 (as defined by MAX_CHAN in sim_encoder.c).

Removing

```
halcmd: unloadrt sim-encoder
```

PINS

- *(float) sim-encoder.<chan-num>.speed* - The speed command for the simulated shaft.
- *(bit) sim-encoder.<chan-num>.phase-A* - Quadrature output.
- *(bit) sim-encoder.<chan-num>.phase-B* - Quadrature output.
- *(bit) sim-encoder.<chan-num>.phase-Z* - Index pulse output.

When *.speed* is positive, *.phase-A* leads *.phase-B*.

PARAMETERS

- *(u32) sim-encoder.<chan-num>.ppr* - Pulses Per Revolution.
- *(float) sim-encoder.<chan-num>.scale* - Scale Factor for *speed*. The default is 1.0, which means that *speed* is in revolutions per second. Change to 60 for RPM, to 360 for degrees per second, 6.283185 for radians per seconed, etc.

Note that pulses per revolution is not the same as counts per revolution. A pulse is a complete quadrature cycle. Most encoder counters will count four times during one complete cycle.

Functions The component exports two functions. Each function affects all simulated encoders.

- *(funct) sim-encoder.make-pulses* - High speed function to generate quadrature pulses (no floating point).
- *(funct) sim-encoder.update-speed* - Low speed function to read *speed*, do scaling, and set up *make-pulses*.

9.6 Debounce

Debounce is a realtime component that can filter the glitches created by mechanical switch contacts. It may also be useful in other applications where short pulses are to be rejected.

Installing

```
halcmd: loadrt debounce cfg=<config-string>
```

<*config-string*> is a series of comma separated decimal integers. Each number installs a group of identical debounce filters, the number determines how many filters are in the group.

For example:

```
halcmd: loadrt debounce cfg=1,4,2
```

will install three groups of filters. Group 0 contains one filter, group 1 contains four, and group 2 contains two filters. The default value for <*config-string*> is *"1"* which will install a single group containing a single filter. The maximum number of groups 8 (as defined by MAX_GROUPS in debounce.c). The maximum number of filters in a group is limited only by shared memory space. Each group is completely independent. All filters in a single group are identical, and they are all updated by the same function at the same time. In the following descriptions, <*G*> is the group number and <*F*> is the filter number within the group. The first filter is group 0, filter 0.

Removing

```
halcmd: unloadrt debounce
```

Pins Each individual filter has two pins.

- *(bit) debounce.*<*G*>.<*F*>*.in* - Input of filter <*F*> in group <*G*>.
- *(bit) debounce.*<*G*>.<*F*>*.out* - Output of filter <*F*> in group <*G*>.

Parameters Each group of filters has one parameter[2].

- *(s32) debounce.*<*G*>*.delay* - Filter delay for all filters in group <*G*>.

The filter delay is in units of thread periods. The minimum delay is zero. The output of a zero delay filter exactly follows its input - it doesn't filter anything. As *delay* increases, longer and longer glitches are rejected. If *delay* is 4, all glitches less than or equal to four thread periods will be rejected.

Functions Each group of filters has one function, which updates all the filters in that group *simultaneously*. Different groups of filters can be updated from different threads at different periods.

- *(funct) debounce.*<*G*> - Updates all filters in group <*G*>.

9.7 Siggen

Siggen is a realtime component that generates square, triangle, and sine waves. It is primarily used for testing.

Installing

```
halcmd: loadrt siggen [num_chan=<chans>]
```

[2] Each individual filter also has an internal state variable. There is a compile time switch that can export that variable as a parameter. This is intended for testing, and simply wastes shared memory under normal circumstances.

<*chans*> is the number of signal generators that you want to install. If *numchan* is not specified, one signal generator will be installed. The maximum number of generators is 16 (as defined by MAX_CHAN in siggen.c). Each generator is completely independent. In the following descriptions, <*chan*> is the number of a specific signal generator (the numbers start at 0).

Removing

```
halcmd: unloadrt siggen
```

Pins Each generator has five output pins.

- *(float) siggen.*<*chan*>*.sine* - Sine wave output.
- *(float) siggen.*<*chan*>*.cosine* - Cosine output.
- *(float) siggen.*<*chan*>*.sawtooth* - Sawtooth output.
- *(float) siggen.*<*chan*>*.triangle* - Triangle wave output.
- *(float) siggen.*<*chan*>*.square* - Square wave output.

All five outputs have the same frequency, amplitude, and offset.

In addition to the output pins, there are three control pins:

- *(float) siggen.*<*chan*>*.frequency* - Sets the frequency in Hertz, default value is 1 Hz.
- *(float) siggen.*<*chan*>*.amplitude* - Sets the peak amplitude of the output waveforms, default is 1.
- *(float) siggen.*<*chan*>*.offset* - Sets DC offset of the output waveforms, default is 0.

For example, if *siggen.0.amplitude* is 1.0 and *siggen.0.offset* is 0.0, the outputs will swing from -1.0 to +1.0. If *siggen.0.amplitude* is 2.5 and *siggen.0.offset* is 10.0, then the outputs will swing from 7.5 to 12.5.

Parameters None. [3]

FUNCTIONS

- *(funct) siggen.*<*chan*>*.update* - Calculates new values for all five outputs.

9.8 lut5

The lut5 component is a 5 input logic component based on a look up table.

- *lut5* does not require a floating point thread.

Installing

```
loadrt lut5 [count=N|names=name1[,name2...]]
addf lut5.N servo-thread | base-thread
setp lut5.N.function 0xN
```

Computing Function To compute the hexadecimal number for the function starting from the top put a 1 or 0 to indicate if that row would be true or false. Next write down every number in the output column starting from the top and writing them from right to left. This will be the binary number. Using a calculator with a program view like the one in Ubuntu enter the binary number and then convert it to hexadecimal and that will be the value for function.

[3] Prior to version 2.1, frequency, amplitude, and offset were parameters. They were changed to pins to allow control by other components.

Table 9.1: Look Up Table

Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Output
0	0	0	0	0	
0	0	0	0	1	
0	0	0	1	0	
0	0	0	1	1	
0	0	1	0	0	
0	0	1	0	1	
0	0	1	1	0	
0	0	1	1	1	
0	1	0	0	0	
0	1	0	0	1	
0	1	0	1	0	
0	1	0	1	1	
0	1	1	0	0	
0	1	1	0	1	
0	1	1	1	0	
0	1	1	1	1	
1	0	0	0	0	
1	0	0	0	1	
1	0	0	1	0	
1	0	0	1	1	
1	0	1	0	0	
1	0	1	0	1	
1	0	1	1	0	
1	0	1	1	1	
1	1	0	0	0	
1	1	0	0	1	
1	1	0	1	0	
1	1	0	1	1	
1	1	1	0	0	
1	1	1	0	1	
1	1	1	1	0	
1	1	1	1	1	

Two Input Example In the following table we have selected the output state for each line that we wish to be true.

Table 9.2: Look Up Table

Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Output
0	0	0	0	0	0
0	0	0	0	1	1
0	0	0	1	0	0
0	0	0	1	1	1

Looking at the output column of our example we want the output to be on when Bit 0 or Bit 0 and Bit1 is on and nothing else. The binary number is *b1010* (rotate the output 90 degrees CW). Enter this number into the calculator then change the display to hexadecimal and the number needed for function is *0xa*. The hexadecimal prefix is *0x*.

Chapter 10

Parallel Port Driver

10.1 Parport

Parport is a driver for the traditional PC parallel port. The port has a total of 17 physical pins. The original parallel port divided those pins into three groups: data, control, and status. The data group consists of 8 output pins, the control group consists of 4 pins, and the status group consists of 5 input pins.

In the early 1990's, the bidirectional parallel port was introduced, which allows the data group to be used for output or input. The HAL driver supports the bidirectional port, and allows the user to set the data group as either input or output. If configured as output, a port provides a total of 12 outputs and 5 inputs. If configured as input, it provides 4 outputs and 13 inputs.

In some parallel ports, the control group pins are open collectors, which may also be driven low by an external gate. On a board with open collector control pins, the *x* mode allows a more flexible mode with 8 outputs, and 9 inputs. In other parallel ports, the control group has push-pull drivers and cannot be used as an input.

HAL and Open Collectors
HAL cannot automatically determine if the *x* mode bidirectional pins are actually open collectors (OC). If they are not, they cannot be used as inputs, and attempting to drive them LOW from an external source can damage the hardware.
To determine whether your port has *open collector* pins, load hal_parport in *x* mode. With no device attached, HAL should read the pin as TRUE. Next, insert a 470 ohm resistor from one of the control pins to GND. If the resulting voltage on the control pin is close to 0V, and HAL now reads the pin as FALSE, then you have an OC port. If the resulting voltage is far from 0V, or HAL does not read the pin as FALSE, then your port cannot be used in *x* mode.
The external hardware that drives the control pins should also use open collector gates (e.g., 74LS05).
On some machines, BIOS settings may affect whether *x* mode can be used. *SPP* mode is most likely to work.

No other combinations are supported, and a port cannot be changed from input to output once the driver is installed. The Parport Block Diagram shows two block diagrams, one showing the driver when the data group is configured for output, and one showing it configured for input. For *x* mode, refer to the pin listing of *halcmd show pin* for pin direction assignment.

The parport driver can control up to 8 ports (defined by MAX_PORTS in hal_parport.c). The ports are numbered starting at zero.

10.1.1 Installing

```
loadrt hal_parport cfg="<config-string>"
```

Using the Port Index I/O addresses below 16 are treated as port indexes. This is the simplest way to install the parport driver and cooperates with the Linux parport_pc driver if it is loaded. This will use the address Linux has detected for parport 0.

```
loadrt hal_parport cfg="0"
```

Using the Port Address The configure string consists of a hex port address, followed by an optional direction, repeated for each port. The direction is *in*, *out*, or *x* and determines the direction of the physical pins 2 through 9, and whether to create input HAL pins for the physical control pins. If the direction is not specified, the data group defaults to output. For example:

```
loadrt hal_parport cfg="0x278 0x378 in 0x20A0 out"
```

This example installs drivers for one port at 0x0278, with pins 2-9 as outputs (by default, since neither *in* nor *out* was specified), one at 0x0378, with pins 2-9 as inputs, and one at 0x20A0, with pins 2-9 explicitly specified as outputs. Note that you must know the base address of the parallel port to properly configure the driver. For ISA bus ports, this is usually not a problem, since the port is almost always at a *well known* address, like 0278 or 0378 which is typically configured in the system BIOS. The address for a PCI card is usually shown in *lspci -v* in an *I/O ports* line, or in the kernel message log after executing *sudo modprobe -a parport_pc*. There is no default address; if <*config-string*> does not contain at least one address, it is an error.

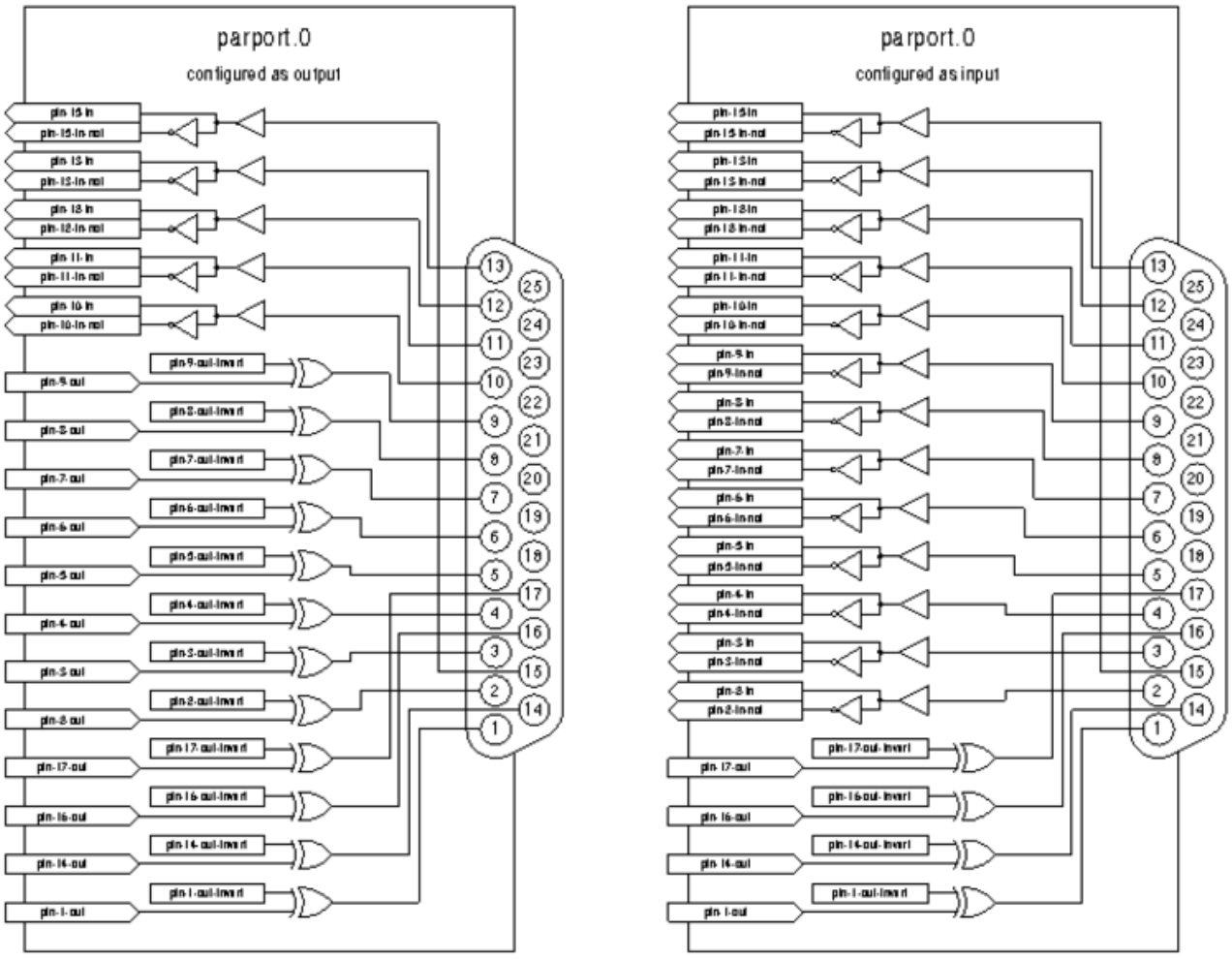


Figure 10.1: Parport Block Diagram

10.1.2 Pins

- *parport.*<*p*>*.pin-*<*n*>*-out* (bit) Drives a physical output pin.
- *parport.*<*p*>*.pin-*<*n*>*-in* (bit) Tracks a physical input pin.
- *parport.*<*p*>*.pin-*<*n*>*-in-not* (bit) Tracks a physical input pin, but inverted.

For each pin, <*p*> is the port number, and <*n*> is the physical pin number in the 25 pin D-shell connector.

For each physical output pin, the driver creates a single HAL pin, for example: *parport.0.pin-14-out*.

Pins 2 through 9 are part of the data group and are output pins if the port is defined as an output port. (Output is the default.) Pins 1, 14, 16, and 17 are outputs in all modes. These HAL pins control the state of the corresponding physical pins.

For each physical input pin, the driver creates two HAL pins, for example: *parport.0.pin-12-in* and *parport.0.pin-12-in-not*.

Pins 10, 11, 12, 13, and 15 are always input pins. Pins 2 through 9 are input pins only if the port is defined as an input port. The *-in* HAL pin is TRUE if the physical pin is high, and FALSE if the physical pin is low. The *-in-not* HAL pin is inverted — it is FALSE if the physical pin is high. By connecting a signal to one or the other, the user can determine the state of the input. In *x* mode, pins 1, 14, 16, and 17 are also input pins.

10.1.3 Parameters

- *parport.*<*p*>*.pin-*<*n*>*-out-invert* (bit) Inverts an output pin.
- *parport.*<*p*>*.pin-*<*n*>*-out-reset* (bit) (only for *out* pins) TRUE if this pin should be reset when the *-reset* function is executed.
- parport.<p>.reset-time' (U32) The time (in nanoseconds) between a pin is set by *write* and reset by the *reset* function if it is enabled.

The *-invert* parameter determines whether an output pin is active high or active low. If *-invert* is FALSE, setting the HAL *-out* pin TRUE drives the physical pin high, and FALSE drives it low. If *-invert* is TRUE, then setting the HAL *-out* pin TRUE will drive the physical pin low.

10.1.4 Functions

- *parport.*<*p*>*.read* (funct) Reads physical input pins of port <*portnum*> and updates HAL *-in* and *-in-not* pins.
- *parport.read-all* (funct) Reads physical input pins of all ports and updates HAL *-in* and *-in-not* pins.
- *parport.*<*p*>*.write* (funct) Reads HAL *-out* pins of port <*p*> and updates that port's physical output pins.
- *parport.write-all* (funct) Reads HAL *-out* pins of all ports and updates all physical output pins.
- *parport.*<*p*>*.reset* (funct) Waits until *reset-time* has elapsed since the associated *write*, then resets pins to values indicated by *-out-invert* and *-out-invert* settings. *reset* must be later in the same thread as *write. 'If '-reset* is TRUE, then the *reset* function will set the pin to the value of *-out-invert*. This can be used in conjunction with stepgen's *doublefreq* to produce one step per period. The stepgen stepspace for that pin must be set to 0 to enable doublefreq.

The individual functions are provided for situations where one port needs to be updated in a very fast thread, but other ports can be updated in a slower thread to save CPU time. It is probably not a good idea to use both an *-all* function and an individual function at the same time.

10.1.5 Common problems

If loading the module reports

```
insmod: error inserting '/home/jepler/emc2/rtlib/hal_parport.ko':
-1 Device or resource busy
```

then ensure that the standard kernel module *parport_pc* is not loaded[1] and that no other device in the system has claimed the I/O ports.

If the module loads but does not appear to function, then the port address is incorrect or the *probe_parport* module is required.

[1] In the LinuxCNC packages for Ubuntu, the file /etc/modprobe.d/emc2 generally prevents *parport_pc* from being automatically loaded.

10.1.6 Using DoubleStep

To setup DoubleStep on the parallel port you must add the function parport.n.reset after parport.n.write and configure stepspace to 0 and the reset time wanted. So that step can be asserted on every period in HAL and then toggled off by parport after being asserted for time specificed by parport.n.reset-time.

For example:

```
loadrt hal_parport cfg="0x378 out"
setp parport.0.reset-time 5000
loadrt stepgen step_type=0,0,0
addf parport.0.read base-thread
addf stepgen.make-pulses base-thread
addf parport.0.write base-thread
addf parport.0.reset base-thread
addf stepgen.capture-position servo-thread
...
setp stepgen.0.steplen 1
setp stepgen.0.stepspace 0
```

More information on DoubleStep can be found on the wiki.

10.2 probe_parport

In modern PCs, the parallel port may require plug and play (PNP) configuration before it can be used. The *probe_parport* module performs configuration of any PNP ports present, and should be loaded before *hal_parport*. On machines without PNP ports, it may be loaded but has no effect.

10.2.1 Installing

```
loadrt probe_parport

loadrt hal_parport ...
```

If the Linux kernel prints a message similar to

```
parport: PnPBIOS parport detected.
```

when the parport_pc module is loaded (*sudo modprobe -a parport_pc; sudo rmmod parport_pc)* then use of this module is probably required.

Chapter 11

HAL Examples

All of these examples assume you are starting with a stepconf based configuration and have two threads base-thread and servo-thread. The stepconf wizard will create an empty custom.hal and a custom_postgui.hal file. The custom.hal file will be loaded after the configuration HAL file and the custom_postgui.hal file is loaded after the GUI has been loaded.

11.1 Manual Toolchange

In this example it is assumed that you're *rolling your own* configuration and wish to add the HAL Manual Toolchange window. The HAL Manual Toolchange is primarily useful if you have presettable tools and you store the offsets in the tool table. If you need to touch off for each tool change then it is best just to split up your g code. To use the HAL Manual Toolchange window you basically have to load the hal_manualtoolchange component then send the iocontrol *tool change* to the hal_manualtoolchange *change* and send the hal_manualtoolchange *changed* back to the iocontrol *tool changed*.

This is an example of manual toolchange *with* the HAL Manual Toolchange component:

```
loadusr -W hal_manualtoolchange
net tool-change iocontrol.0.tool-change => hal_manualtoolchange.change
net tool-changed iocontrol.0.tool-changed <= hal_manualtoolchange.changed
net tool-number iocontrol.0.tool-prep-number => hal_manualtoolchange.number
net tool-prepare-loopback iocontrol.0.tool-prepare => iocontrol.0.tool-prepared
```

This is an example of manual toolchange *without* the HAL Manual Toolchange component:

```
net tool-number <= iocontrol.0.tool-prep-number
net tool-change-loopback iocontrol.0.tool.-change => iocontrol.0.tool-changed
net tool-prepare-loopback iocontrol.0.tool-prepare => iocontrol.0.tool-prepared
```

11.2 Compute Velocity

This example uses *ddt*, *mult2* and *abs* to compute the velocity of a single axis. For more information on the real time components see the man pages or the Realtime Components section ([sec:Realtime-Components]).

The first thing is to check your configuration to make sure you are not using any of the real time components all ready. You can do this by opening up the HAL Configuration window and look for the components in the pin section. If you are then find the .hal file that they are being loaded in and increase the counts and adjust the instance to the correct value. Add the following to your custom.hal file.

Load the realtime components.

```
loadrt ddt count=1
loadrt mult2 count=1
loadrt abs count=1
```

Add the functions to a thread so it will get updated.

```
addf ddt.0 servo-thread
addf mult2.0 servo-thread
addf abs.0 servo-thread
```

Make the connections.

```
setp mult2.in1 60
net xpos-cmd ddt.0.in
net X-IPS mult2.0.in0 <= ddt.0.out
net X-ABS abs.0.in <= mult2.0.out
net X-IPM abs.0.out
```

In this last section we are setting the mult2.0.in1 to 60 to convert the inch per second to inch per minute that we get from the ddt.0.out.

The xpos-cmd sends the commanded position to the ddt.0.in. The ddt computes the derivative of the change of the input.

The ddt2.0.out is multiplied by 60 to give IPM.

The mult2.0.out is sent to the abs to get the absolute value.

The following figure shows the result when the X axis is moving at 15 IPM in the minus direction. Notice that we can get the absolute value from either the abs.0.out pin or the X-IPM signal.

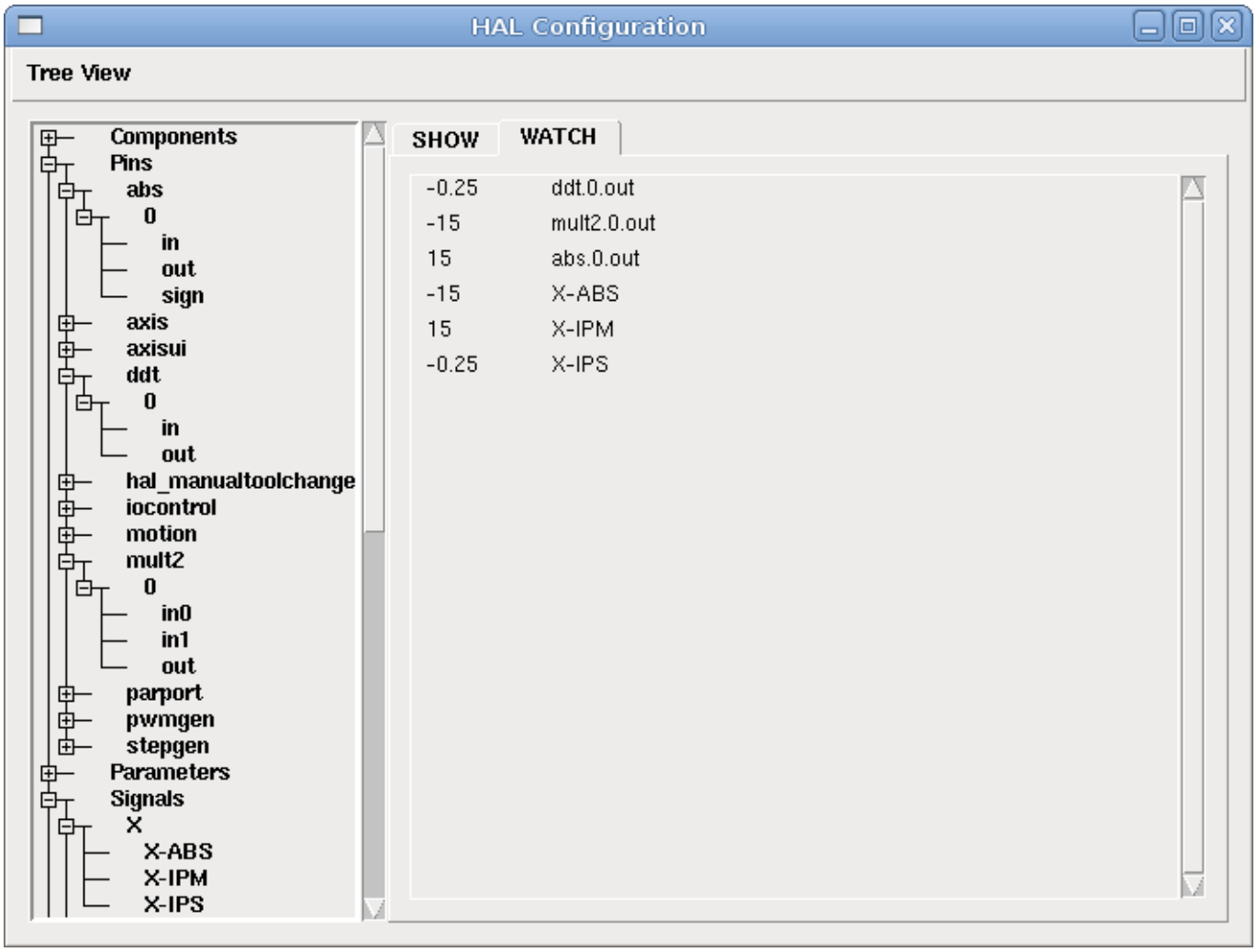


Figure 11.1: Velocity Example

11.3 Soft Start

This example shows how the HAL components *lowpass*, *limit2* or *limit3* can be used to limit how fast a signal changes.

In this example we have a servo motor driving a lathe spindle. If we just used the commanded spindle speeds on the servo it will try to go from present speed to commanded speed as fast as it can. This could cause a problem or damage the drive. To slow the rate of change we can send the motion.spindle-speed-out through a limiter before the PID, so that the PID command value changes to new settings more slowly.

Three built-in components that limit a signal are:

- *limit2* limits the range and first derivative of a signal.
- *limit3* limits the range, first and second derivatives of a signal.
- *lowpass* uses an exponentially-weighted moving average to track an input signal.

To find more information on these HAL components check the man pages.

Place the following in a text file called softstart.hal. If you're not familiar with Linux place the file in your home directory.

```
loadrt threads period1=1000000 name1=thread
loadrt siggen
loadrt lowpass
loadrt limit2
loadrt limit3
net square siggen.0.square => lowpass.0.in limit2.0.in limit3.0.in
net lowpass <= lowpass.0.out
net limit2 <= limit2.0.out
net limit3 <= limit3.0.out
setp siggen.0.frequency .1
setp lowpass.0.gain .01
setp limit2.0.maxv 2
setp limit3.0.maxv 2
setp limit3.0.maxa 10
addf siggen.0.update thread
addf lowpass.0 thread
addf limit2.0 thread
addf limit3.0 thread
start
loadusr halscope
```

Open a terminal window and run the file with the following command.

```
halrun -I softstart.hal
```

When the HAL Oscilloscope first starts up click *OK* to accept the default thread.

Next you have to add the signals to the channels. Click on channel 1 then select *square* from the Signals tab. Repeat for channels 2-4 and add lowpass, limit2, and limit3.

Next to set up a trigger signal click on the Source None button and select square. The button will change to Source Chan 1.

Next click on Single in the Run Mode radio buttons box. This will start a run and when it finishes you will see your traces.

To separate the signals so you can see them better click on a channel then use the Pos slider in the Vertical box to set the positions.

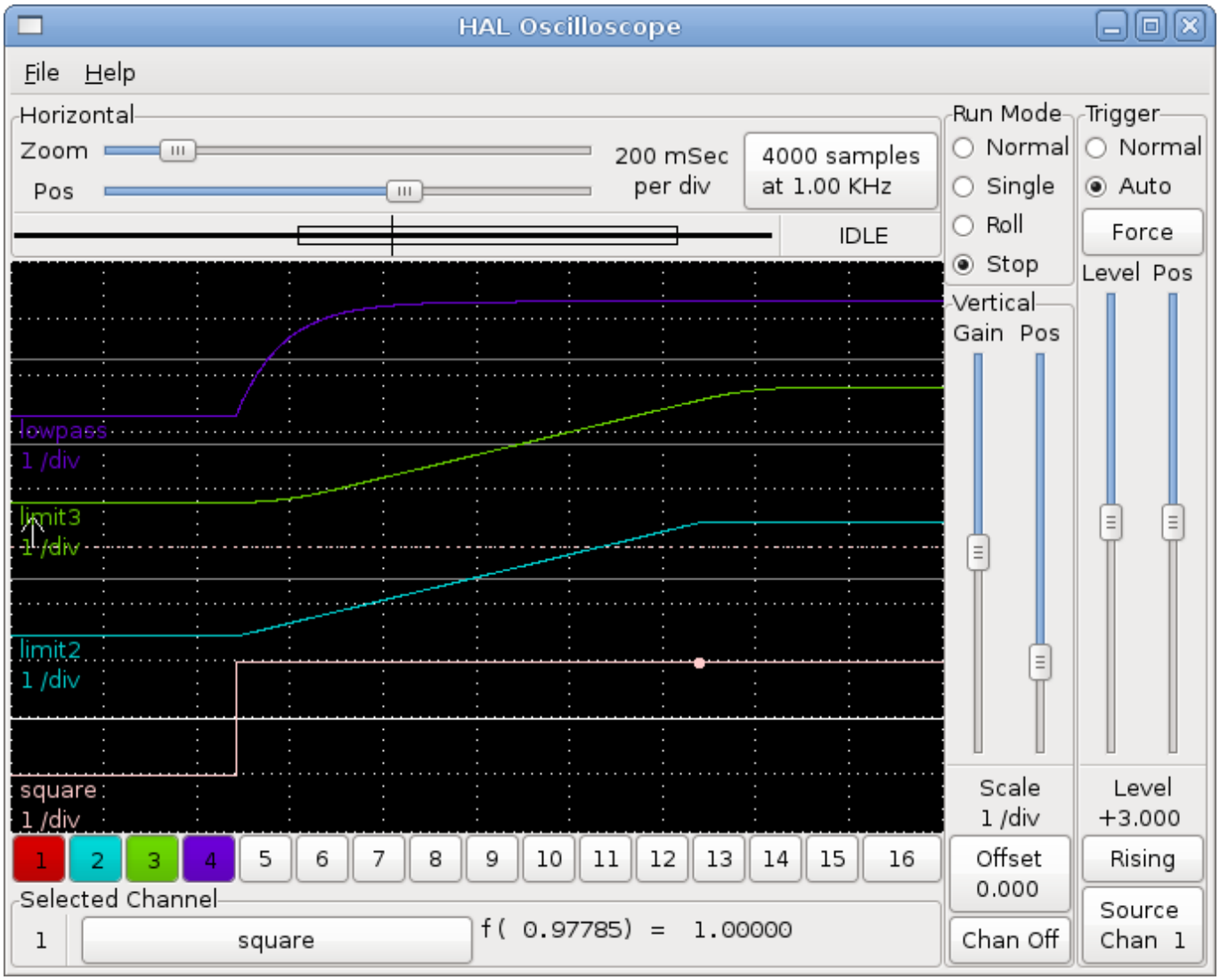


Figure 11.2: Softstart

To see the effect of changing the set point values of any of the components you can change them in the terminal window. To see what different gain settings do for lowpass just type the following in the terminal window and try different settings.

```
setp lowpass.0.gain *.01
```

After changing a setting run the oscilloscope again to see the change.

When you're finished type *exit* in the terminal window to shut down halrun and close the halscope. Don't close the terminal window with halrun running as it might leave some things in memory that could prevent EMC from loading.

For more information on Halscope see the HAL manual.

11.4 Stand Alone HAL

In some cases you might want to run a GladeVCP screen with just HAL. For example say you had a stepper driven device that all you need is to run a stepper motor. A simple *Start/Stop* interface is all you need for your application so no need to load up and configure a full blown CNC application.

In the following example we have created a simple GladeVCP panel with one

Basic Syntax

```
# load the winder.glade GUI and name it winder
loadusr -Wn winder gladevcp -c winder -u handler.py winder.glade

# load realtime components
loadrt threads name1=fast period1=50000 fp1=0 name2=slow period2=1000000
loadrt stepgen step_type=0 ctrl_type=v
loadrt hal_parport cfg="0x378 out"

# add functions to threads
addf stepgen.make-pulses fast
addf stepgen.update-freq slow
addf stepgen.capture-position slow
addf parport.0.read fast
addf parport.0.write fast

# make hal connections
net winder-step parport.0.pin-02-out <= stepgen.0.step
net winder-dir parport.0.pin-03-out <= stepgen.0.dir
net run-stepgen stepgen.0.enable <= winder.start_button



# start the threads
start

# comment out the following lines while testing and use the interactive
# option halrun -I -f start.hal to be able to show pins etc.

# wait until the gladevcp GUI named winder terminates
waitusr winder

# stop HAL threads
stop

# unload HAL all components before exiting
unloadrt all
```

Chapter 12

Comp HAL Component Generator

12.1 Introduction

Writing a HAL component can be a tedious process, most of it in setup calls to *rtapi_* and *hal_* functions and associated error checking. *comp* will write all this code for you, automatically.

Compiling a HAL component is also much easier when using *comp*, whether the component is part of the LinuxCNC source tree, or outside it.

For instance, when coded in C, a simple component such as "ddt" is around 80 lines of code. The equivalent component is very short when written using the *comp* preprocessor:

Simple Comp Example

```
component ddt "Compute the derivative of the input function";
pin in float in;
pin out float out;
variable float old;
function _;
license "GPL"; // indicates GPL v2 or later
;;
float tmp = in;
out = (tmp - old) / fperiod;
old = tmp;
```

12.2 Installing

If you're working with an installed version of LinuxCNC you will need to install the development packages.

One method is to say following line in a terminal.

Installing Dev

```
sudo apt-get install linuxcnc-dev
```

Another method is to use the Synaptic Package Manager from the main Ubuntu menu to install linuxcnc-dev.

12.3 Definitions

- *component* - A component is a single real-time module, which is loaded with *halcmd loadrt*. One *.comp* file specifies one component. The component name and file name must match.

- *instance* - A component can have zero or more instances. Each instance of a component is created equal (they all have the same pins, parameters, functions, and data) but behave independently when their pins, parameters, and data have different values.
- *singleton* - It is possible for a component to be a "singleton", in which case exactly one instance is created. It seldom makes sense to write a *singleton* component, unless there can literally only be a single object of that kind in the system (for instance, a component whose purpose is to provide a pin with the current UNIX time, or a hardware driver for the internal PC speaker)

12.4 Instance creation

For a singleton, the one instance is created when the component is loaded.

For a non-singleton, the *count* module parameter determines how many numbered instances are created. If not specified, the *name* module parameter determines how many named instances are created. Otherwise, a single numbered instance is created.

12.5 Implicit Parameters

Functions are implicitly passed the *period* parameter which is the time in nanoseconds of the last period to execute the comp. Functions which use floating-point can also refer to *fperiod* which is the floating-point time in seconds, or (period*1e-9). This can be useful in comps that need the timing information.

12.6 Syntax

A *.comp* file consists of a number of declarations, followed by *;;* on a line of its own, followed by C code implementing the module's functions.

Declarations include:

- *component HALNAME (DOC);*
- *pin PINDIRECTION TYPE HALNAME ([SIZE]\|[MAXSIZE: CONDSIZE]) (if CONDITION) (= STARTVALUE) (DOC) ;*
- *param PARAMDIRECTION TYPE HALNAME ([SIZE]\|[MAXSIZE: CONDSIZE]) (if CONDITION) (= STARTVALUE) (DOC) ;*
- *function HALNAME (fp | nofp) (DOC);*
- *option OPT (VALUE);*
- *variable CTYPE STARREDNAME ([SIZE]);*
- *description DOC;*
- *see_also DOC;*
- *license LICENSE;*
- *author AUTHOR;*

Parentheses indicate optional items. A vertical bar indicates alternatives. Words in *CAPITALS* indicate variable text, as follows:

- *NAME* - A standard C identifier
- *STARREDNAME* - A C identifier with zero or more * before it. This syntax can be used to declare instance variables that are pointers. Note that because of the grammar, there may not be whitespace between the * and the variable name.

- *HALNAME* - An extended identifier. When used to create a HAL identifier, any underscores are replaced with dashes, and any trailing dash or period is removed, so that "this_name_" will be turned into "this-name", and if the name is "_", then a trailing period is removed as well, so that "function _" gives a HAL function name like "component.<num>" instead of "component.<num>."

  If present, the prefix *hal_* is removed from the beginning of the component name when creating pins, parameters and functions.

  In the HAL identifier for a pin or parameter, # denotes an array item, and must be used in conjunction with a *[SIZE]* declaration. The hash marks are replaced with a 0-padded number with the same length as the number of # characters.

  When used to create a C identifier, the following changes are applied to the HALNAME:

  1. Any "#" characters, and any ".", "_" or "-" characters immediately before them, are removed.
  2. Any remaining "." and "-" characters are replaced with "_".
  3. Repeated "_" characters are changed to a single "_" character.

  A trailing "_" is retained, so that HAL identifiers which would otherwise collide with reserved names or keywords (e.g., *min*) can be used.

HALNAME	C Identifier	HAL Identifier
x_y_z	x_y_z	x-y-z
x-y.z	x_y_z	x-y.z
x_y_z_	x_y_z_	x-y-z
x.##.y	x_y(MM)	x.MM.z
x.##	x(MM)	x.MM

- *if CONDITION* - An expression involving the variable *personality* which is nonzero when the pin or parameter should be created
- *SIZE* - A number that gives the size of an array. The array items are numbered from 0 to *SIZE*-1.
- *MAXSIZE : CONDSIZE* - A number that gives the maximum size of the array followed by an expression involving the variable *personality* and which always evaluates to less than *MAXSIZE*. When the array is created its size will be *CONDSIZE*.
- *DOC* - A string that documents the item. String can be a C-style "double quoted" string, like:

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

  or a Python-style "triple quoted" string, which may include embedded newlines and quote characters, such as:

```
"""The effect of this parameter, also known as "the orb of zot",
will require at least two paragraphs to explain.

Hopefully these paragraphs have allowed you to understand "zot"
better."""
```

  The documentation string is in "groff -man" format. For more information on this markup format, see *groff_man(7)*. Remember that comp interprets backslash escapes in strings, so for instance to set the italic font for the word *example*, write:

```
"\\fIexample\\fB"
```

- *TYPE* - One of the HAL types: *bit*, *signed*, *unsigned*, or *float*. The old names *s32* and *u32* may also be used, but *signed* and *unsigned* are preferred.
- *PINDIRECTION* - One of the following: *in*, *out*, or *io*. A component sets a value for an *out* pin, it reads a value from an *in* pin, and it may read or set the value of an *io* pin.
- *PARAMDIRECTION* - One of the following: *r* or *rw*. A component sets a value for a *r* parameter, and it may read or set the value of a *rw* parameter.
- *STARTVALUE* - Specifies the initial value of a pin or parameter. If it is not specified, then the default is *0* or *FALSE*, depending on the type of the item.

12.6.1 HAL functions

- *fp* - Indicates that the function performs floating-point calculations.
- *nofp* - Indicates that it only performs integer calculations. If neither is specified, *fp* is assumed. Neither comp nor gcc can detect the use of floating-point calculations in functions that are tagged *nofp*, but use of such operations results in undefined behavior.

12.6.2 Options

The currently defined options are:

- *option singleton yes* - (default: no) Do not create a *count* module parameter, and always create a single instance. With *singleton*, items are named *component-name.item-name* and without *singleton*, items for numbered instances are named *component-name.<num>.item-name*.
- *option default_count number* - (default: 1) Normally, the module parameter *count* defaults to 1. If specified, the *count* will default to this value instead.
- *option count_function yes* - (default: no) Normally, the number of instances to create is specified in the module parameter *count*; if *count_function* is specified, the value returned by the function *int get_count(void)* is used instead, and the *count* module parameter is not defined.
- *option rtapi_app no* - (default: yes) Normally, the functions *rtapi_app_main* and *rtapi_app_exit* are automatically defined. With *option rtapi_app no*, they are not, and must be provided in the C code. When implementing your own *rtapi_app_main*, call the function *int export(char *prefix, long extra_arg)* to register the pins, parameters, and functions for *prefix*.
- *option data TYPE* - (default: none) **deprecated** If specified, each instance of the component will have an associated data block of type *TYPE* (which can be a simple type like *float* or the name of a type created with *typedef*). In new components, *variable* should be used instead.
- *option extra_setup yes* - (default: no) If specified, call the function defined by *EXTRA_SETUP* for each instance. If using the automatically defined *rtapi_app_main*, *extra_arg* is the number of this instance.
- *option extra_cleanup yes* - (default: no) If specified, call the function defined by *EXTRA_CLEANUP* from the automatically defined *rtapi_app_exit*, or if an error is detected in the automatically defined *rtapi_app_main*.
- *option userspace yes* - (default: no) If specified, this file describes a userspace component, rather than a real one. A userspace component may not have functions defined by the *function* directive. Instead, after all the instances are constructed, the C function *user_mainloop()* is called. When this function returns, the component exits. Typically, *user_mainloop()* will use *FOR_ALL_INSTS()* to perform the update action for each instance, then sleep for a short time. Another common action in *user_mainloop()* may be to call the event handler loop of a GUI toolkit.
- *option userinit yes* - (default: no) This option is ignored if the option *userspace* (see above) is set to *no*. If *userinit* is specified, the function *userinit(argc,argv)* is called before *rtapi_app_main()* (and thus before the call to *hal_init()*). This function may process the commandline arguments or take other actions. Its return type is *void*; it may call *exit()* if it wishes to terminate rather than create a HAL component (for instance, because the commandline arguments were invalid).

If an option's VALUE is not specified, then it is equivalent to specifying *option ... yes*. The result of assigning an inappropriate value to an option is undefined. The result of using any other option is undefined.

12.6.3 License and Authorship

- *LICENSE* - Specify the license of the module for the documentation and for the MODULE_LICENSE() module declaration. For example, to specify that the module's license is GPL v2 or later,

```
license "GPL"; // indicates GPL v2 or later
```

For additional information on the meaning of MODULE_LICENSE() and additional license identifiers, see <*linux/module.h*>. or the manual page *rtapi_module_param(3)*

This declaration is required.

- *AUTHOR* - Specify the author of the module for the documentation.

12.6.4 Per-instance data storage

- *variable CTYPE STARREDNAME;*
- *variable CTYPE STARREDNAME[SIZE];*
- *variable CTYPE STARREDNAME = DEFAULT;*
- *variable CTYPE STARREDNAME[SIZE] = DEFAULT;*

Declare a per-instance variable *STARREDNAME* of type *CTYPE*, optionally as an array of *SIZE* items, and optionally with a default value *DEFAULT*. Items with no *DEFAULT* are initialized to all-bits-zero. *CTYPE* is a simple one-word C type, such as *float*, *u32*, *s32*, int, etc. Access to array variables uses square brackets.

If a variable is to be of a pointer type, there may not be any space between the "*" and the variable name. Therefore, the following is acceptable:

```
variable int *example;
```

but the following are not:

```
variable int* badexample;
variable int * badexample;
```

12.6.5 Comments

C++-style one-line comments (//. . .) and

C-style multi-line comments (/* . . . */) are both supported in the declaration section.

12.7 Restrictions

Though HAL permits a pin, a parameter, and a function to have the same name, comp does not.

Variable and function names that can not be used or are likely to cause problems include:

- Anything beginning with _comp.
- *comp_id*
- *fperiod*
- *rtapi_app_main*
- *rtapi_app_exit*
- *extra_setup*
- *extra_cleanup*

12.8 Convenience Macros

Based on the items in the declaration section, *comp* creates a C structure called `struct __comp_state`. However, instead of referring to the members of this structure (e.g., `*(inst->name)`), they will generally be referred to using the macros below. The details of `struct __comp_state` and these macros may change from one version of *comp* to the next.

- *FUNCTION(name)* - Use this macro to begin the definition of a realtime function which was previously declared with *function NAME*. The function includes a parameter *period* which is the integer number of nanoseconds between calls to the function.
- *EXTRA_SETUP()* - Use this macro to begin the definition of the function called to perform extra setup of this instance. Return a negative Unix *errno* value to indicate failure (e.g., *return -EBUSY* on failure to reserve an I/O port), or 0 to indicate success.
- *EXTRA_CLEANUP()* - Use this macro to begin the definition of the function called to perform extra cleanup of the component. Note that this function must clean up all instances of the component, not just one. The "pin_name", "parameter_name", and "data" macros may not be used here.
- *pin_name* or *parameter_name* - For each pin *pin_name* or param *parameter_name* there is a macro which allows the name to be used on its own to refer to the pin or parameter. When *pin_name* or *parameter_name* is an array, the macro is of the form *pin_name(idx)* or *param_name(idx)* where *idx* is the index into the pin array. When the array is a variable-sized array, it is only legal to refer to items up to its *condsize*.

  When the item is a conditional item, it is only legal to refer to it when its *condition* evaluated to a nonzero value.
- *variable_name* - For each variable *variable_name* there is a macro which allows the name to be used on its own to refer to the variable. When *variable_name* is an array, the normal C-style subscript is used: *variable_name[idx]*
- *data* - If "option data" is specified, this macro allows access to the instance data.
- *fperiod* - The floating-point number of seconds between calls to this realtime function.
- *FOR_ALL_INSTS() {...}* - For userspace components. This macro iterates over all the defined instances. Inside the body of the loop, the *pin_name*, *parameter_name*, and *data* macros work as they do in realtime functions.

12.9 Components with one function

If a component has only one function and the string "FUNCTION" does not appear anywhere after *;;*, then the portion after *;;* is all taken to be the body of the component's single function. See the Simple Comp for and example of this.

12.10 Component Personality

If a component has any pins or parameters with an "if condition" or "[maxsize : condsize]", it is called a component with *personality*. The *personality* of each instance is specified when the module is loaded. *Personality* can be used to create pins only when needed. For instance, personality is used in the *logic* component, to allow for a variable number of input pins to each logic gate and to allow for a selection of any of the basic boolean logic functions *and*, *or*, and *xor*.

12.11 Compiling

Place the *.comp* file in the source directory *linuxcnc/src/hal/components* and re-run *make*. *Comp* files are automatically detected by the build system.

If a *.comp* file is a driver for hardware, it may be placed in *linuxcnc/src/hal/components* and will be built unless LinuxCNC is configured as a userspace simulator.

12.12 Compiling realtime components outside the source tree

comp can process, compile, and install a realtime component in a single step, placing *rtexample.ko* in the LinuxCNC realtime module directory:

```
comp --install rtexample.comp
```

Or, it can process and compile in one step, leaving *example.ko* (or *example.so* for the simulator) in the current directory:

```
comp --compile rtexample.comp
```

Or it can simply process, leaving *example.c* in the current directory:

```
comp rtexample.comp
```

comp can also compile and install a component written in C, using the *--install* and *--compile* options shown above:

```
comp --install rtexample2.c
```

man-format documentation can also be created from the information in the declaration section:

```
comp --document rtexample.comp
```

The resulting manpage, *example.9* can be viewed with

```
man ./example.9
```

or copied to a standard location for manual pages.

12.13 Compiling userspace components outside the source tree

comp can process, compile, install, and document userspace components:

```
comp usrexample.comp
comp --compile usrexample.comp
comp --install usrexample.comp
comp --document usrexample.comp
```

This only works for *.comp* files, not for *.c* files.

12.14 Examples

12.14.1 constant

Note that the declaration "function _" creates functions named "constant.0" , etc. The file name must match the component name.

```
component constant;
pin out float out;
param r float value = 1.0;
function _;
license "GPL"; // indicates GPL v2 or later
;;
FUNCTION(_) { out = value; }
```

12.14.2 sincos

This component computes the sine and cosine of an input angle in radians. It has different capabilities than the "sine" and "cosine" outputs of siggen, because the input is an angle, rather than running freely based on a "frequency" parameter.

The pins are declared with the names *sin_* and *cos_* in the source code so that they do not interfere with the functions *sin()* and *cos()*. The HAL pins are still called *sincos.<num>.sin*.

```
component sincos;
pin out float sin_;
pin out float cos_;
pin in float theta;
function _;
license "GPL"; // indicates GPL v2 or later
;;
#include <rtapi_math.h>
FUNCTION(_) { sin_ = sin(theta); cos_ = cos(theta); }
```

12.14.3 out8

This component is a driver for a *fictional* card called "out8", which has 8 pins of digital output which are treated as a single 8-bit value. There can be a varying number of such cards in the system, and they can be at various addresses. The pin is called *out_* because *out* is an identifier used in *<asm/io.h>*. It illustrates the use of *EXTRA_SETUP* and *EXTRA_CLEANUP* to request an I/O region and then free it in case of error or when the module is unloaded.

```
component out8;
pin out unsigned out_ "Output value; only low 8 bits are used";
param r unsigned ioaddr;

function _;

option count_function;
option extra_setup;
option extra_cleanup;
option constructable no;

license "GPL"; // indicates GPL v2 or later
;;
#include <asm/io.h>

#define MAX 8
int io[MAX] = {0,};
RTAPI_MP_ARRAY_INT(io, MAX, "I/O addresses of out8 boards");

int get_count(void) {
    int i = 0;
    for(i=0; i<MAX && io[i]; i++) { /* Nothing */ }
    return i;
}

EXTRA_SETUP() {
    if(!rtapi_request_region(io[extra_arg], 1, "out8")) {
        // set this I/O port to 0 so that EXTRA_CLEANUP does not release the IO
        // ports that were never requested.
        io[extra_arg] = 0;
        return -EBUSY;
    }
    ioaddr = io[extra_arg];
    return 0; }
```

```
EXTRA_CLEANUP() {
    int i;
    for(i=0; i < MAX && io[i]; i++) {
        rtapi_release_region(io[i], 1);
    }
}

FUNCTION(_) { outb(out_, ioaddr); }
```

12.14.4 hal_loop

```
component hal_loop;
pin out float example;
```

This fragment of a component illustrates the use of the *hal_* prefix in a component name. *loop* is the name of a standard Linux kernel module, so a *loop* component might not successfully load if the Linux *loop* module was also present on the system.

When loaded, *halcmd show comp* will show a component called *hal_loop*. However, the pin shown by *halcmd show pin* will be *loop.0.example*, not *hal-loop.0.example*.

12.14.5 arraydemo

This realtime component illustrates use of fixed-size arrays:

```
component arraydemo "4-bit Shift register";
pin in bit in;
pin out bit out-# [4];
function _ nofp;
license "GPL"; // indicates GPL v2 or later
;;
int i;
for(i=3; i>0; i--) out(i) = out(i-1);
out(0) = in;
```

12.14.6 rand

This userspace component changes the value on its output pin to a new random value in the range (0,1) about once every 1ms.

```
component rand;
option userspace;

pin out float out;
license "GPL"; // indicates GPL v2 or later
;;
#include <unistd.h>

void user_mainloop(void) {
    while(1) {
        usleep(1000);
        FOR_ALL_INSTS() out = drand48();
    }
}
```

12.14.7 logic

This realtime component shows how to use "personality" to create variable-size arrays and optional pins.

```
component logic "LinuxCNC HAL component providing experimental logic functions";
pin in bit in-##[16 : personality & 0xff];
pin out bit and if personality & 0x100;
pin out bit or if personality & 0x200;
pin out bit xor if personality & 0x400;
function _ nofp;
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:
.IP \\(bu 4
The number of input pins, usually from 2 to 16
.IP \\(bu
256 (0x100)  if the 'and' output is desired
.IP \\(bu
512 (0x200)  if the 'or' output is desired
.IP \\(bu
1024 (0x400)  if the 'xor' (exclusive or) output is desired""";
license "GPL"; // indicates GPL v2 or later
;;
FUNCTION(_) {
    int i, a=1, o=0, x=0;
    for(i=0; i < (personality & 0xff); i++) {
        if(in(i)) { o = 1; x = !x; }
        else { a = 0; }
    }
    if(personality & 0x100) and = a;
    if(personality & 0x200) or = o;
    if(personality & 0x400) xor = x;
}
```

A typical load line for this component might be

```
loadrt logic count=3 personality=0x102,0x305,0x503
```

which creates the following pins:

- A 2-input AND gate: logic.0.and, logic.0.in-00, logic.0.in-01
- 5-input AND and OR gates: logic.1.and, logic.1.or, logic.1.in-00, logic.1.in-01, logic.1.in-02, logic.1.in-03, logic.1.in-04,
- 3-input AND and XOR gates: logic.2.and, logic.2.xor, logic.2.in-00, logic.2.in-01, logic.2.in-02

Chapter 13

Creating Userspace Python Components

13.1 Basic usage

A userspace component begins by creating its pins and parameters, then enters a loop which will periodically drive all the outputs from the inputs. The following component copies the value seen on its input pin (*passthrough.in*) to its output pin (*passthrough.out*) approximately once per second.

```
#!/usr/bin/python
import hal, time
h = hal.component("passthrough")
h.newpin("in", hal.HAL_FLOAT, hal.HAL_IN)
h.newpin("out", hal.HAL_FLOAT, hal.HAL_OUT)
h.ready()
try:
    while 1:
        time.sleep(1)
        h['out'] = h['in']
except KeyboardInterrupt:
    raise SystemExit
```

Copy the above listing into a file named "passthrough", make it executable (*chmod +x*), and place it on your *$PATH*. Then try it out:

```
halrun

halcmd: loadusr passthrough

halcmd: show pin

    Component Pins:
    Owner Type  Dir     Value  Name
     03   float IN          0  passthrough.in
     03   float OUT         0  passthrough.out

halcmd: setp passthrough.in 3.14

halcmd: show pin

    Component Pins:
    Owner Type  Dir     Value  Name
     03   float IN       3.14  passthrough.in
     03   float OUT      3.14  passthrough.out
```

13.2 Userspace components and delays

If you typed “show pin” quickly, you may see that *passthrough.out* still had its old value of 0. This is because of the call to *time.sleep(1)*, which makes the assignment to the output pin occur at most once per second. Because this is a userspace component, the actual delay between assignments can be much longer if the memory used by the passthrough component is swapped to disk, the assignment could be delayed until that memory is swapped back in.

Thus, userspace components are suitable for user-interactive elements such as control panels (delays in the range of milliseconds are not noticed, and longer delays are acceptable), but not for sending step pulses to a stepper driver board (delays must always be in the range of microseconds, no matter what).

13.3 Create pins and parameters

```
h = hal.component("passthrough")
```

The component itself is created by a call to the constructor *hal.component*. The arguments are the HAL component name and (optionally) the prefix used for pin and parameter names. If the prefix is not specified, the component name is used.

```
h.newpin("in", hal.HAL_FLOAT, hal.HAL_IN)
```

Then pins are created by calls to methods on the component object. The arguments are: pin name suffix, pin type, and pin direction. For parameters, the arguments are: parameter name suffix, parameter type, and parameter direction.

Table 13.1: HAL Option Names

Pin and Parameter Types:	HAL_BIT	HAL_FLOAT	HAL_S32	HAL_U32
Pin Directions:	HAL_IN	HAL_OUT	HAL_IO	
Parameter Directions:	HAL_RO	HAL_RW		

The full pin or parameter name is formed by joining the prefix and the suffix with a ".", so in the example the pin created is called *passthrough.in*.

```
h.ready()
```

Once all the pins and parameters have been created, call the *.ready()* method.

13.3.1 Changing the prefix

The prefix can be changed by calling the *.setprefix()* method. The current prefix can be retrieved by calling the *.getprefix()* method.

13.4 Reading and writing pins and parameters

For pins and parameters which are also proper Python identifiers, the value may be accessed or set using the attribute syntax:

```
h.out = h.in
```

For all pins, whether or not they are also proper Python identifiers, the value may be accessed or set using the subscript syntax:

```
h['out'] = h['in']
```

13.4.1 Driving output (HAL_OUT) pins

Periodically, usually in response to a timer, all HAL_OUT pins should be "driven" by assigning them a new value. This should be done whether or not the value is different than the last one assigned. When a pin is connected to a signal, its old output value is not copied into the signal, so the proper value will only appear on the signal once the component assigns a new value.

13.4.2 Driving bidirectional (HAL_IO) pins

The above rule does not apply to bidirectional pins. Instead, a bidirectional pin should only be driven by the component when the component wishes to change the value. For instance, in the canonical encoder interface, the encoder component only sets the *index-enable* pin to **FALSE** (when an index pulse is seen and the old value is **TRUE**), but never sets it to **TRUE**. Repeatedly driving the pin **FALSE** might cause the other connected component to act as though another index pulse had been seen.

13.5 Exiting

A *halcmd unload* request for the component is delivered as a *KeyboardInterrupt* exception. When an unload request arrives, the process should either exit in a short time, or call the *.exit()* method on the component if substantial work (such as reading or writing files) must be done to complete the shutdown process.

13.6 Project ideas

- Create an external control panel with buttons, switches, and indicators. Connect everything to a microcontroller, and connect the microcontroller to the PC using a serial interface. Python has a very capable serial interface module called pyserial (Ubuntu package name “python-serial”, in the universe repository)
- Attach a LCDProc-compatible LCD module and use it to display a digital readout with information of your choice (Ubuntu package name “lcdproc”, in the universe repository)
- Create a virtual control panel using any GUI library supported by Python (gtk, qt, wxwindows, etc)

Chapter 14

Index

X