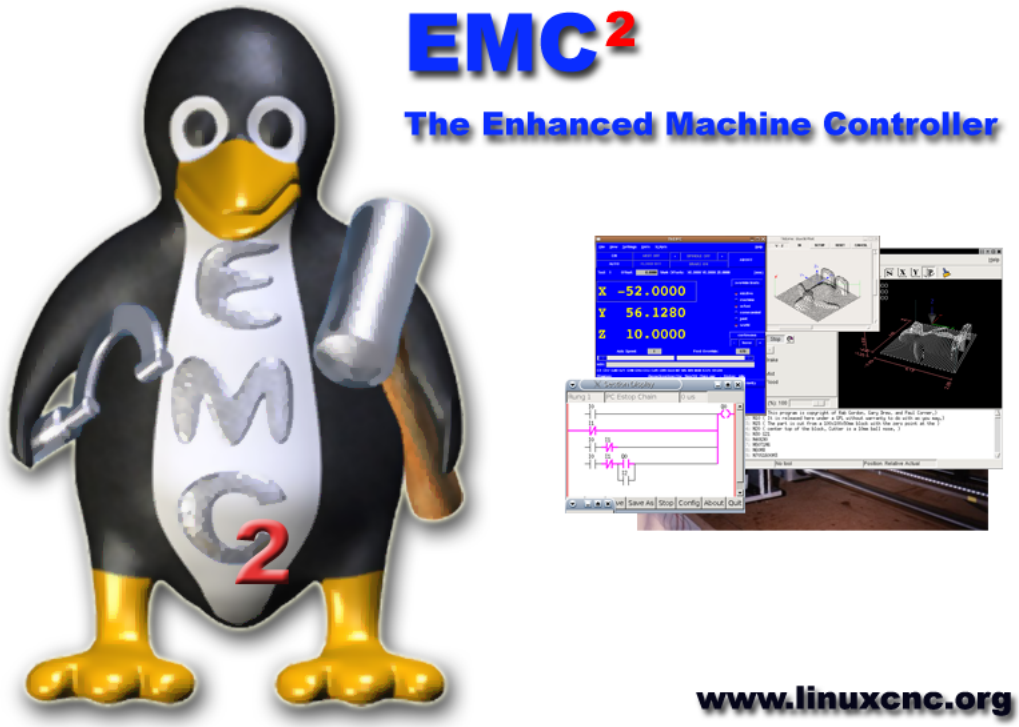
HAL Manual 2.3

The EMC Team

May 16, 2009



This manual 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.

Contents

Chapter 1

Introduction

This manual is for the person who wants to know more about HAL than is needed to just set up an EMC configuration file. HAL can run without EMC so this manual focuses on the stand alone HAL. For information on EMC related HAL see the Integrators manual.

1.1 What is HAL?

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 EMC 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.

1.1.1 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.

1.1.1.1 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.

1.1.1.2 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.

1.1.1.3 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.

1.1.1.4 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.

1.1.2 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 EMC.

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 +4294967295
- S32 - a 32 bit signed integer, legal values are -2147483648 to +2147483647

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 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 **??**

encoder Software based encoder counter. See section **??**

pid Proportional/Integral/Derivative control loops. See section **??**

siggen A sine/cosine/triangle/square wave generator for testing. See section **??**

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. See section **??**

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 5.1

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

halmeter A handy multimeter for HAL signals. See section 5.2

halscope A full featured digital storage oscilloscope for HAL signals. See section 5.3

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

1.4 Tinkertoys, Erector Sets, Legos and the HAL

A first introduction to HAL concepts can be mind boggling. Building anything with blocks can be a challenge but some of the toys that we played with as kids can be an aid to building things with the HAL.

1.4.1 Tower

> I'm watching as my son and his six year old daughter build a tower from a box full of random sized blocks, rods, jar lids and such. The aim is to see how tall they can make the tower. The narrower the base the more blocks left to stack on top. But the narrower the base, the less stable the tower. I see them studying both the next block and the shelf where they want to place it to see how it will balance out with the rest of the tower.

The notion of stacking cards to see how tall you can make a tower is a very old and honored way of spending spare time. At first read, the integrator may have gotten the impression that building a HAL was a bit like that. It can be but with proper planning an integrator can build a stable system as complex as the machine at hand requires.

1.4.2 Erector Sets[1]

What was great about the sets was the building blocks, metal struts and angles and plates, all with regularly spaced holes. You could design things and hold them together with the little screws and nuts.

> I got my first erector set for my fourth birthday. I know the box suggested a much older age than I was. Perhaps my father was really giving himself a present. I had a hard time with the little screws and nuts. I really needed four arms, one each for the screwdriver, screw, parts to be bolted together, and nut. Perseverance, along with father's eventual boredom, got me to where I had built every project in the booklet. Soon I was lusting after the bigger sets that were also printed on that paper. Working with those regular sized pieces opened up a world of construction for me and soon I moved well beyond the illustrated projects.

Hal components are not all the same size and shape but they allow for grouping into larger units that will do useful work.In this sense they are like the parts of an Erector set. Some components are long and thin. They essentially connect high level commands to specific physical pins. Other components are more like the rectangular platforms upon which whole machines could be built. An integrator will quickly get beyond the brief examples and begin to bolt together components in ways that are unique to them.

[1] The Erector Set was an invention of AC Gilbert

1.4.3 Tinkertoys[2]

> Wooden Tinker toys had a more humane feel that the cold steel of Erector Sets. The heart of construction with Tinker Toys was a round connector with eight holes equally spaced around the circumference. It also had a hole in the center that was perpendicular to all the holes around the hub.
>
> Hubs were connected with rods of several different lengths. Builders would make large wheels by using these rods as spokes sticking out from the center hub.
>
> My favorite project was a rotating space station. Short spokes radiated from all the holes in the center hub and connected with hubs on the ends of each spoke. These outer hubs were connected to each other with longer spokes. I'd spend hours dreaming of living in such a device, walking from hub to hub around the outside as it slowly rotated producing near gravity in weightless space. Supplies traveled through the spokes in elevators that transferred them to an from rockets docked at the center hub while they transferred their precious cargoes.

The idea of one pin or component being the hub for many connections is also an easy concept within the HAL. Examples two and four (see section 2) connect the meter and scope to signals that are intended to go elsewhere. Less easy is the notion of a hub for several incoming signals but that is also possible with proper use of functions within that hub component that handle those signals as they arrive from other components.

Another thought that comes forward from this toy is a mechanical representation of HAL threads. A thread might look a bit like a centipede, caterpillar, or earwig. A backbone of hubs, HAL components, strung together with rods, HAL signals. Each component takes in it own parameters and input pins and passes on output pins and parameters to the next component. Signals travel along the backbone from end to end and are added to or modified by each component in turn.

Threads are all about timing and doing a set of tasks from end to end. A mechanical representation is available with Tinkertoys also when we think of the length of the toy as a measure of the time taken to get from one end to the other. A very different thread or backbone is created by connecting the same set of hubs with different length rods. The total length of the backbone can be changed by the length of rods used to connect the hubs. The order of operations is the same but the time to get from beginning to end is very different.

1.4.4 A Lego Example[3]

When Lego blocks first arrived in our stores they were pretty much all the same size and shape. Sure there were half sized one and a few quarter sized as well but that rectangular one did most of the work. Lego blocks interconnected by snapping the holes in the underside of one onto the pins that stuck up on another. By overlapping layers, the joints between could be made very strong, even around corners or tees.

> I watched my children and grandchildren build with legos – the same legos. There are a few thousand of them in an old ratty but heavy duty cardboard box that sits in a corner of the recreation room. It stays there in the open because it was too much trouble to put the box away and then get it back out for every visit and it is always used during a visit. There must be Lego parts in there from a couple dozen different sets. The little booklets that came with them are long gone but the magic of building with interlocking pieces all the same size is something to watch.

1.5 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.

[2] Tinkertoy is now a registered trademark of the Hasbro company.
[3] The Lego name is a trademark of the Lego company.

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 10mS, 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 EMC you might have a 50μs thread and a 1ms thread. These would be created based on BASE_PERIOD and SERVO_PERIOD–the actual times depend on the ini.

The next step is to decide what each thread needs to do. Some of those decisions are the same in (nearly) any EMC 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

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.

Notation

Command line examples are presented in **`bold typewriter`** font. Responses from the computer will be in `typewriter` font. Text inside square brackets `[like-this]` is optional. Text inside angle brackets `<like-this>` represents a field that can take on different values, and the adjacent paragraph will explain the appropriate values. Text items separated by a vertical bar "|"means that one or the other, but not both, should be present. All command line examples assume that you are in the `emc2/` directory, and paths will be shown accordingly when needed.

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:

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

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 EMC 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 EMC, you may want to study `emc2/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

Loading a realtime component

For this tutorial, we are going to assume that you have successfully installed the Live CD or compiled the emc2/ source tree and, if necessary, invoked the `emc-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:

~$ **cd emc2**
~/emc2$ **halrun**
halcmd:

With the realtime OS and RTAPI loaded, we can move into the first example. Notice that the prompt has changed from the shell's "$" to "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 Siggen section of the Integrators Manual. It is a realtime component, implemented as a Linux kernel module. To load `siggen` use the `halcmd loadrt` command:

halcmd: **loadrt siggen**

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 `halcmd` reference in section 5.1 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:

halcmd: **show comp**
Loaded HAL Components:
ID Type Name PID State
3 RT siggen ready
2 User halcmd10190 10190 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.

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

halcmd: **show pin**
Component Pins:
Owner Type Dir Value Name
3 float IN 1 siggen.0.amplitude
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 HAL - a complex system could have dozens or hundreds of pins. But right now there are only eight 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:

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:

halcmd: **show funct**
Exported Functions:
Owner CodeAddr Arg FP Users Name
00003 b7f74ac5 b7d0c0b4 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[1].

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 1mS (1000000nS):

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

Let's see if that worked:

halcmd: **show thread**
Realtime Threads:
Period FP Name (Time, Max-Time)
999849 YES test-thread (0, 0)

It did. The period is not exactly 1000000nS 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:

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:

[1] The codeaddr and arg fields were used in development, and should probably be removed from the halcmd listing.

halcmd: **show thread**
Realtime Threads:
Period FP Name (Time, Max-Time)
999849 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 float OUT -0.9406941 siggen.0.cosine
3 float IN 1 siggen.0.frequency
3 float IN 0 siggen.0.offset
3 float OUT -0.1164055 siggen.0.sawtooth
3 float OUT 0.379820 siggen.0.sine
3 float OUT -1 siggen.0.square
3 float OUT -0.7728110 siggen.0.triangle
halcmd: **show pin**
Component Pins:
Owner Type Dir Value Name
3 float IN 1 siggen.0.amplitude
3 float OUT 0.9958036 siggen.0.cosine
3 float IN 1 siggen.0.frequency
3 float IN 0 siggen.0.offset
3 float OUT 0.9708287 siggen.0.sawtooth
3 float OUT -0.09151597 siggen.0.sine
3 float OUT 1 siggen.0.square
3 float OUT 0.9416574 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.

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:

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

Check the parameters and pins again:

halcmd: **show param**
Parameters:
Owner Type Dir Value Name
3 s32 RO 397 siggen.0.update.time
3 s32 RW 109100 siggen.0.update.tmax

halcmd: **show pin**
Component Pins:
Owner Type Dir Value Name
3 float IN 5 siggen.0.amplitude
3 float OUT -4.179375 siggen.0.cosine
3 float IN 1 siggen.0.frequency
3 float IN 0 siggen.0.offset
3 float OUT 0.9248036 siggen.0.sawtooth
3 float OUT -2.744599 siggen.0.sine
3 float OUT 5 siggen.0.square
3 float OUT -3.150393 siggen.0.triangle

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

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:

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:

halcmd: **save all saved.hal**

Exiting HalRun

When your finished with your HAL session type "exit" at the halcmd: prompt. Do not simply close the terminal window without shutting down the HAL session.

halcmd: **exit**
~/emc2$

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:

~/emc2$ **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):

halcmd: **start**
halcmd: **exit**
~/emc2$

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:

~/emc2$ **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:

~/emc2$ **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**

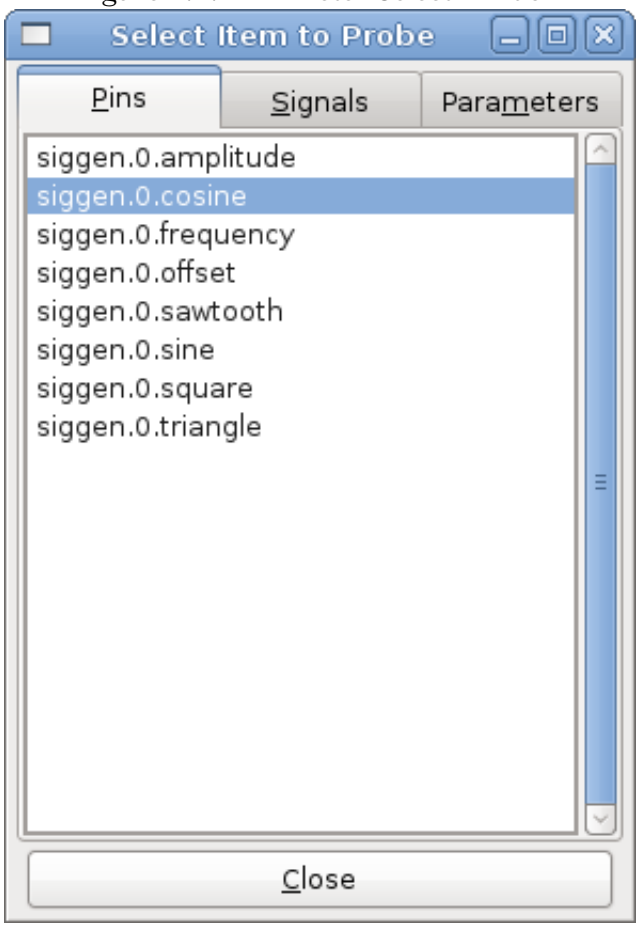
Starting halmeter

At this point we have the siggen component loaded and running. It's time to start halmeter. Since halmeter is a GUI app, X must be running.

halcmd: **loadusr halmeter**

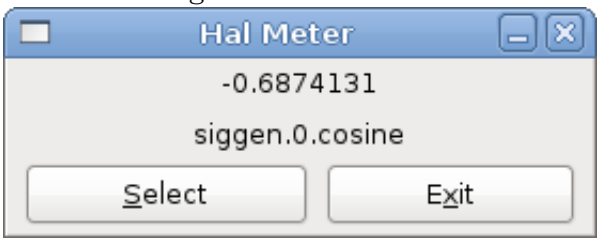
The first window you will see is the "Select Item to Probe" window.

Figure 2.1: HAL Meter 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**
~/emc2$ **halrun**

Installing the components

Now we are going to load the step pulse generator component. For a detailed description of this component refer to Stepgen section of the Integrators Manual. For now, we can skip the details, and just run the following commands:[2]

In this example we will use the "velocity" control type of stepgen.

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 micro-seconds and a slow one with a period of 1mS. 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
3 float IN 1 siggen.0.amplitude
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
3 float OUT 0 stepgen.0.counts
2 bit OUT FALSE stepgen.0.dir
2 bit IN FALSE stepgen.0.enable
2 float IN 0 stepgen.0.position-fb
2 float OUT 0 stepgen.0.step
2 bit OUT FALSE stepgen.0.velocity-cmd
2 s32 OUT 0 stepgen.1.counts
2 bit OUT FALSE stepgen.1.dir
2 bit IN FALSE stepgen.1.enable
2 float IN 0 stepgen.1.position-fb
2 float OUT 0 stepgen.1.step
2 bit OUT FALSE stepgen.1.velocity-cmd

[2] The "\" at the end of a long line indicates line wrapping (needed for formatting this document). When entering the commands at the command line, simply skip the "\" (do not hit enter) and keep typing from the following line.

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
2 u32 RW 00000001 stepgen.0.dirhold
2 u32 RW 00000001 stepgen.0.dirsetup
2 float RO 0 stepgen.0.frequency
2 float RW 0 stepgen.0.maxaccel
2 float RW 0 stepgen.0.maxvel
2 float RW 1 stepgen.0.position-scale
2 s32 RO 0 stepgen.0.rawcounts
2 u32 RW 00000001 stepgen.0.steplen
2 u32 RW 00000001 stepgen.0.stepspace
2 u32 RW 00000001 stepgen.1.dirhold
2 u32 RW 00000001 stepgen.1.dirsetup
2 float RO 0 stepgen.1.frequency
2 float RW 0 stepgen.1.maxaccel
2 float RW 0 stepgen.1.maxvel
2 float RW 1 stepgen.1.position-scale
2 s32 RO 0 stepgen.1.rawcounts
2 u32 RW 00000001 stepgen.1.steplen
2 u32 RW 00000001 stepgen.1.stepspace
2 s32 RO 0 stepgen.capture-position.time
2 s32 RW 0 stepgen.capture-position.tmax
2 s32 RO 0 stepgen.make-pulses.time
2 s32 RW 0 stepgen.make-pulses.tmax
2 s32 RO 0 stepgen.update-freq.time
2 s32 RW 0 stepgen.update-freq.tmax

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.

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
float 0 Y-vel
<== siggen.0.sine
==> stepgen.1.velocity

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.

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 d8a3a120 d8bd322c YES 0 siggen.0.update
00003 d8bf45b0 d8bd30b4 YES 0 stepgen.capture-position
00003 d8bf42c0 d8bd30b4 NO 0 stepgen.make-pulses
00003 d8bf46b0 d8bd30b4 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)
988960 YES slow (0, 0)
49448 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 s`iggen.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)
988960 YES slow (0, 0)
1 siggen.0.update
2 stepgen.update-freq
49448 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.

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.

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.

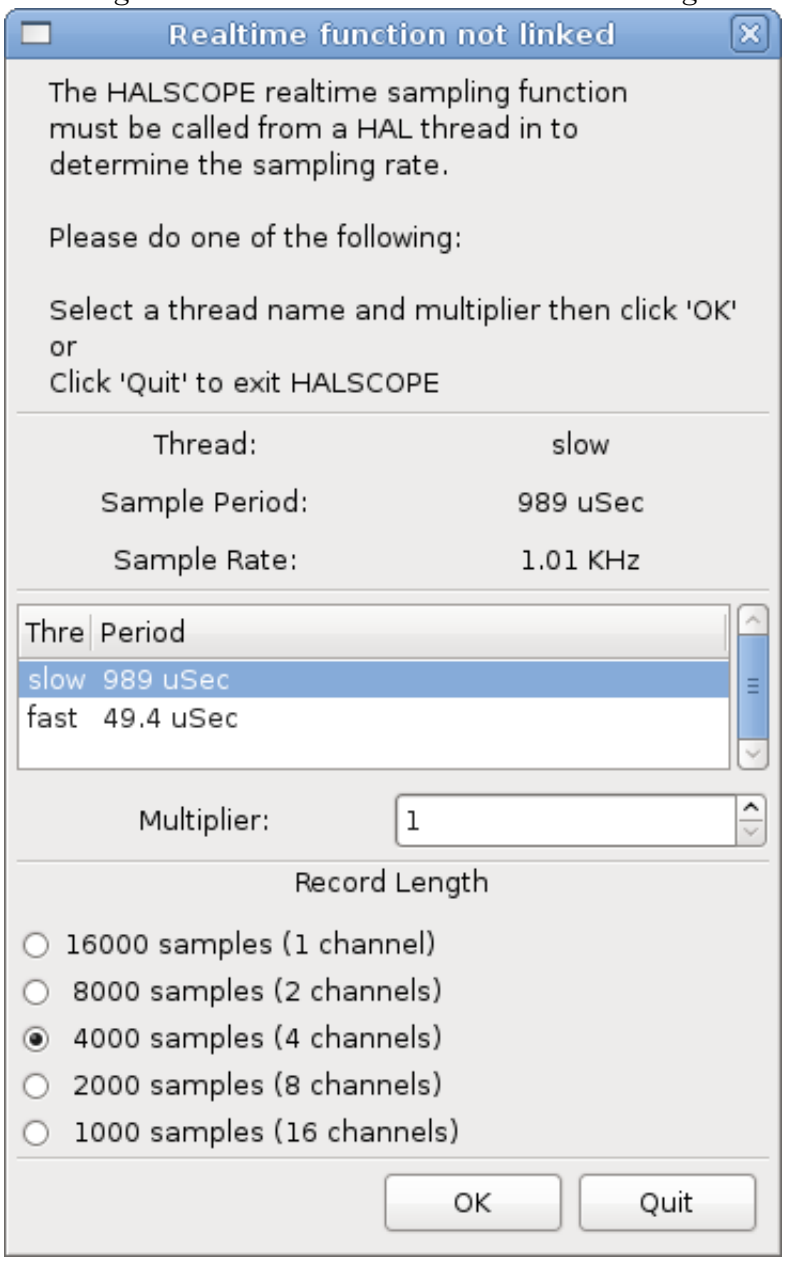
Starting Halscope

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.

halcmd: **loadusr halscope**

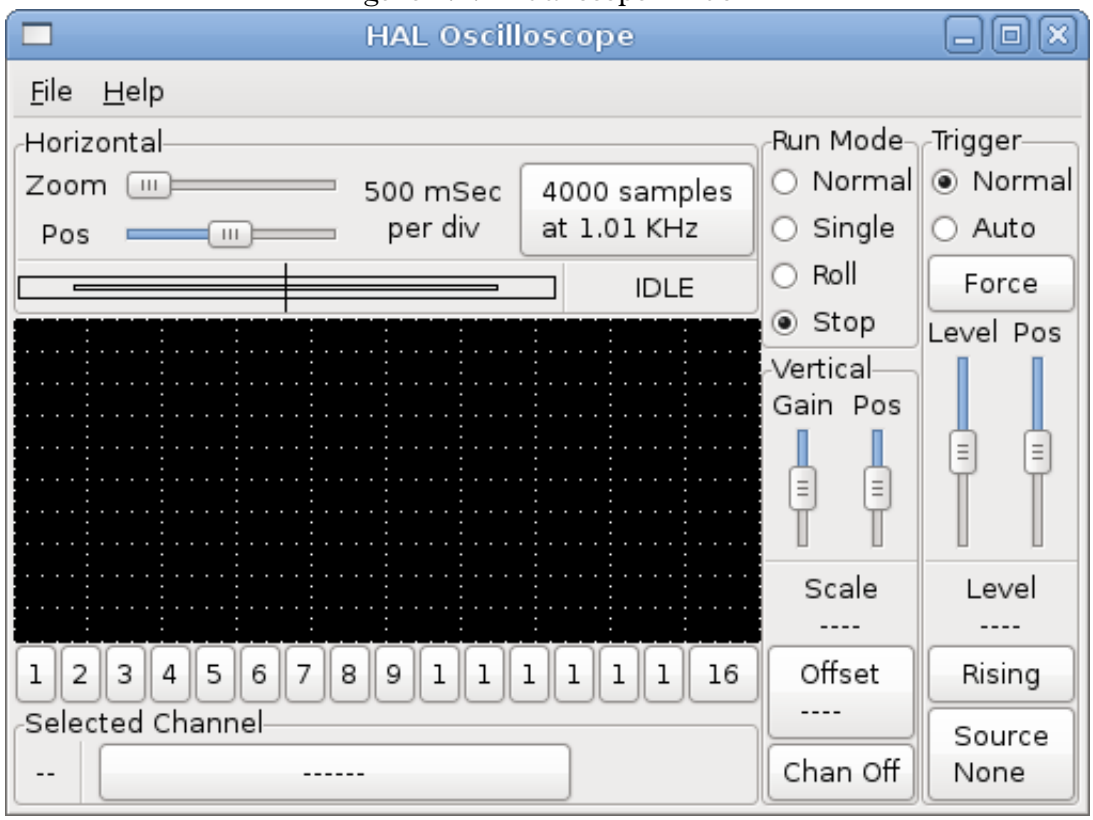
The scope GUI window will open, immediately followed by a "Realtime function not linked" dialog that looks like the following figure .

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 989uS 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

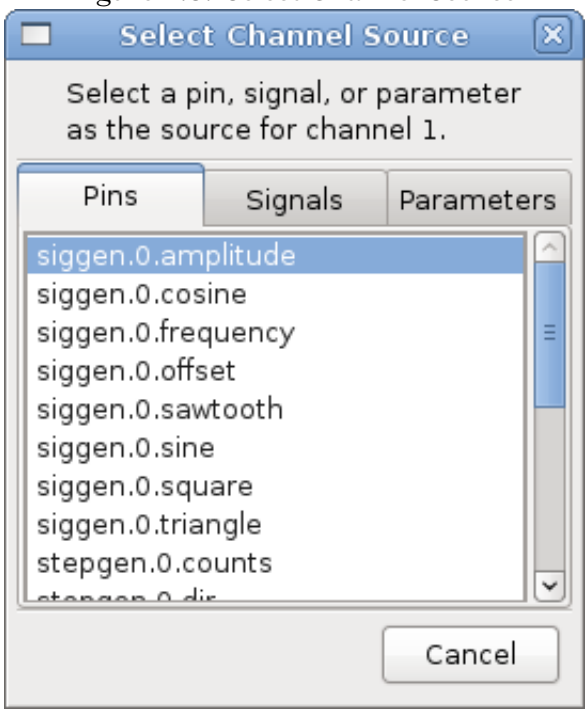


Hooking up the scope probes

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.

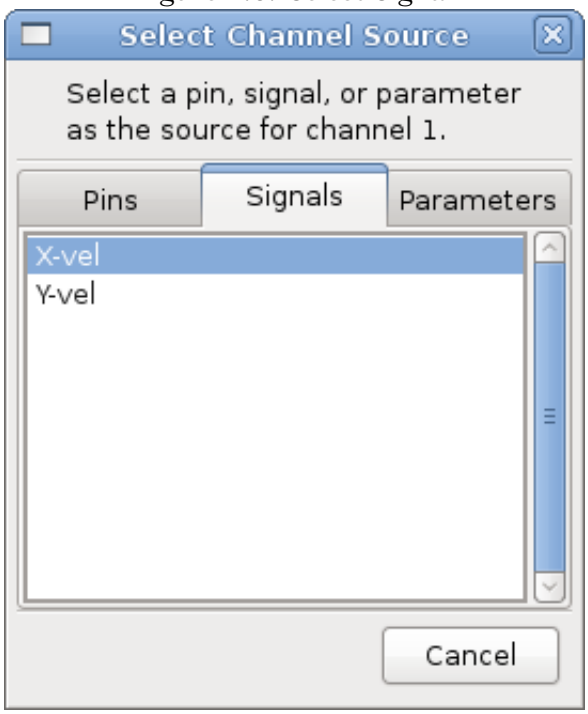
The channel buttons run across the bottom of the halscope screen. Click button "1", and you will see the "Select Channel Source" dialog as shown in the following figure. This dialog is very similar to the one used by Halmeter. We would like to look at the signals we defined earlier, so we click on the "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.

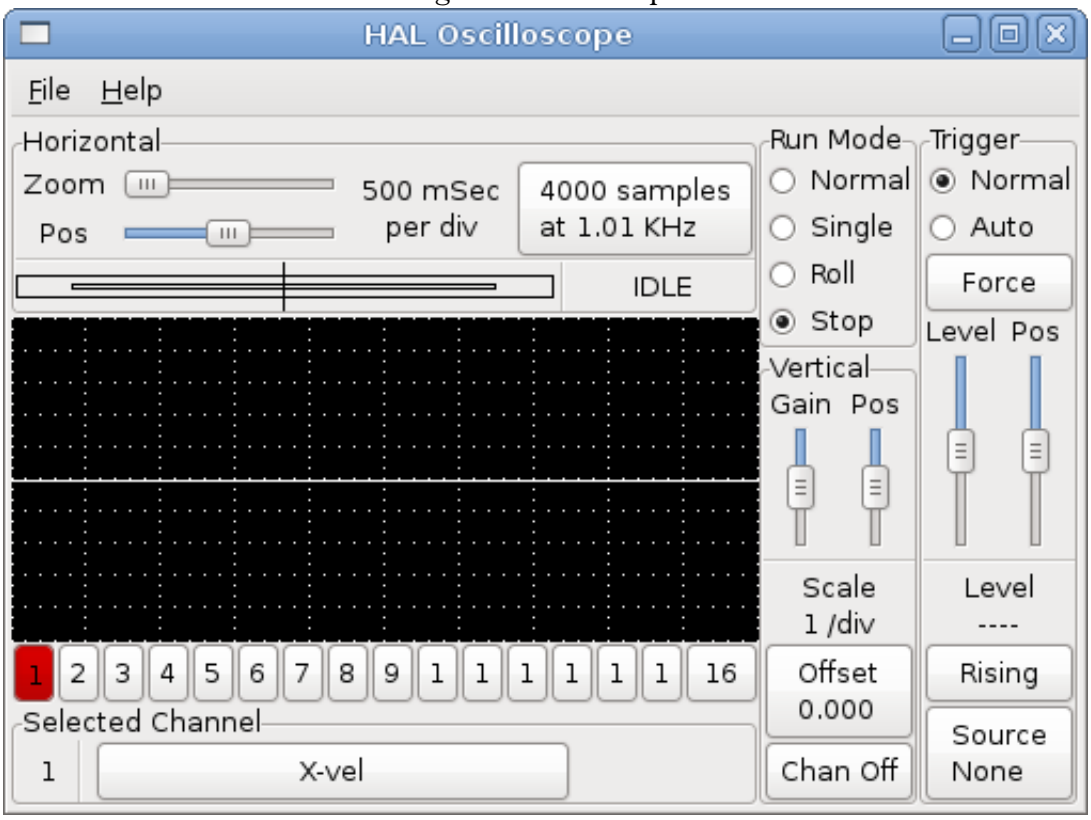
Figure 2.6: Select Signal



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.

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".

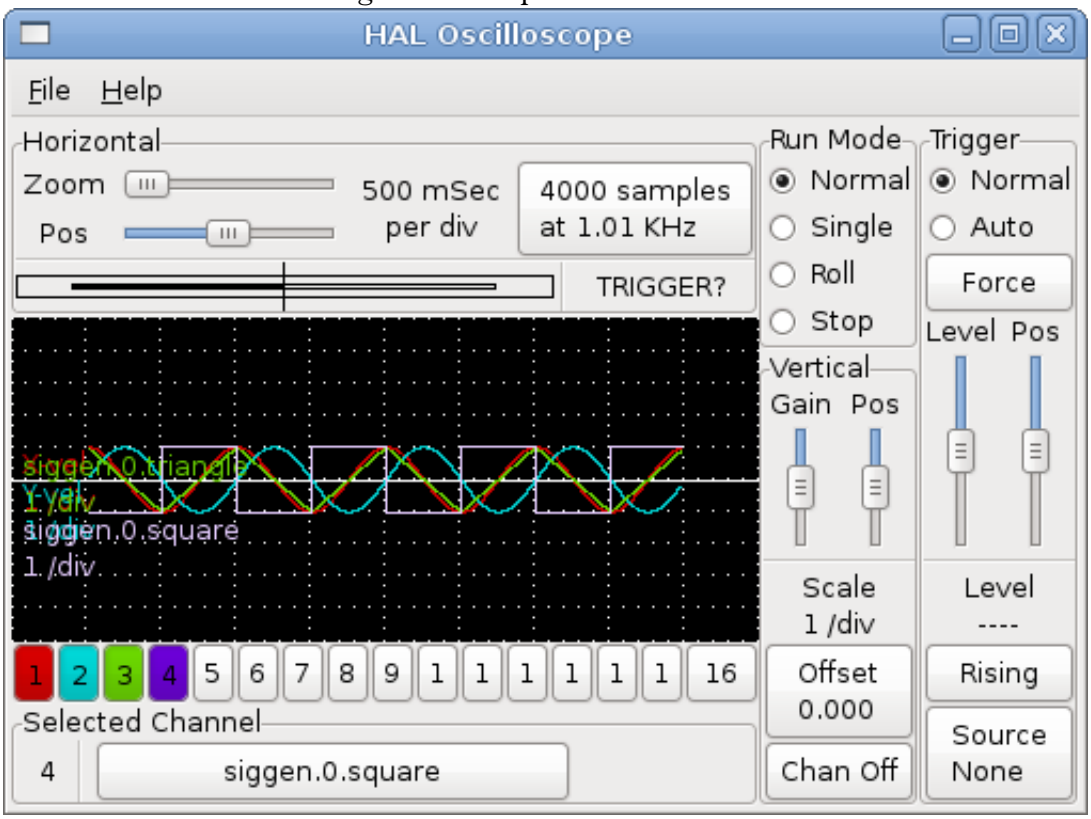
Figure 2.7: Halscope



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

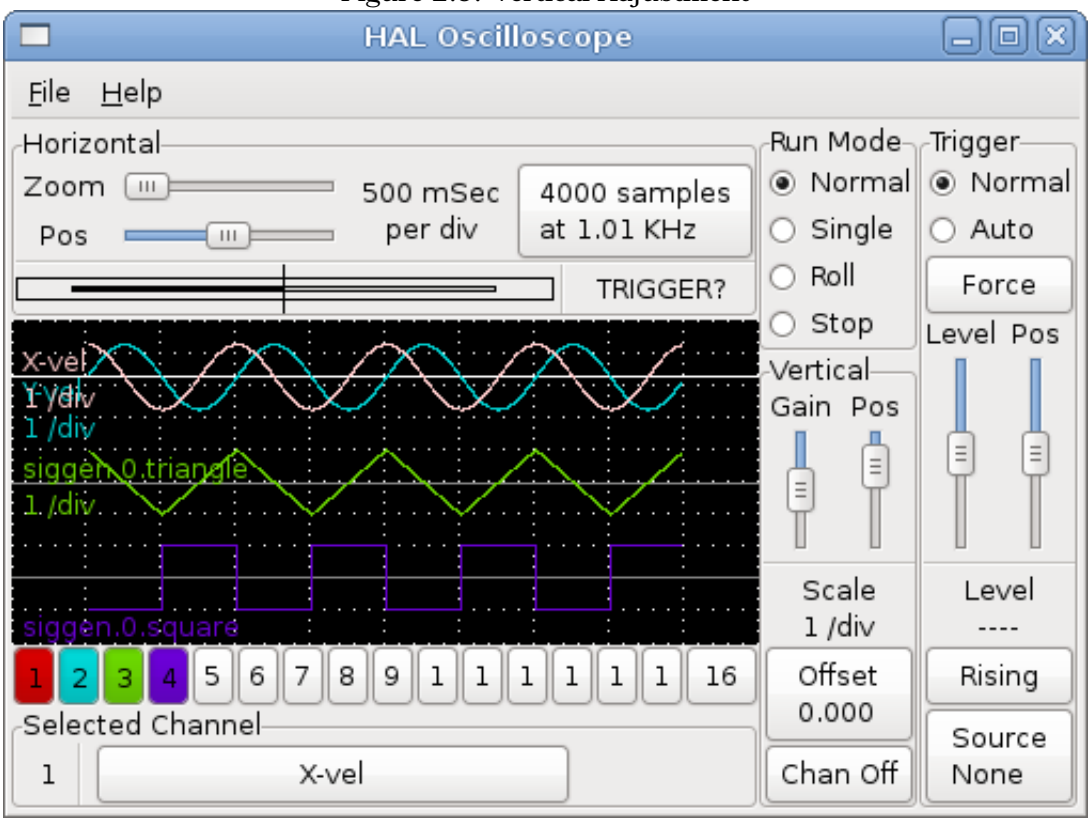


The "Selected Channel" box at the bottom tells you that the purple trace is the currently selected one, channel 4, which is displaying the value of the pin "siggen.0.square". Try clicking channel buttons 1 through 3 to highlight the other three traces.

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 (see the halscope reference in section 5.3 for details).

Figure 2.9: Vertical Adjustment



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.

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 2.10: Trigger Source Dialog

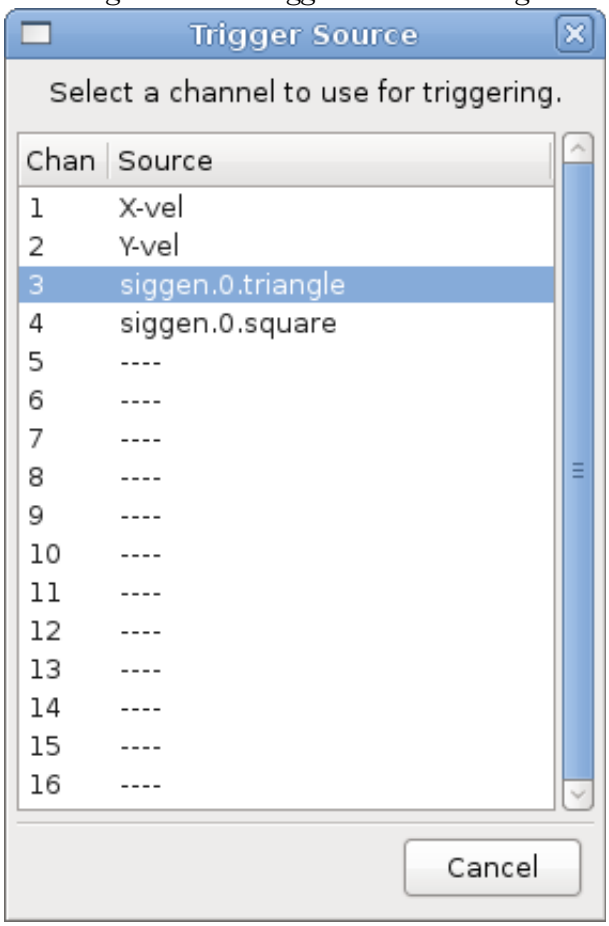
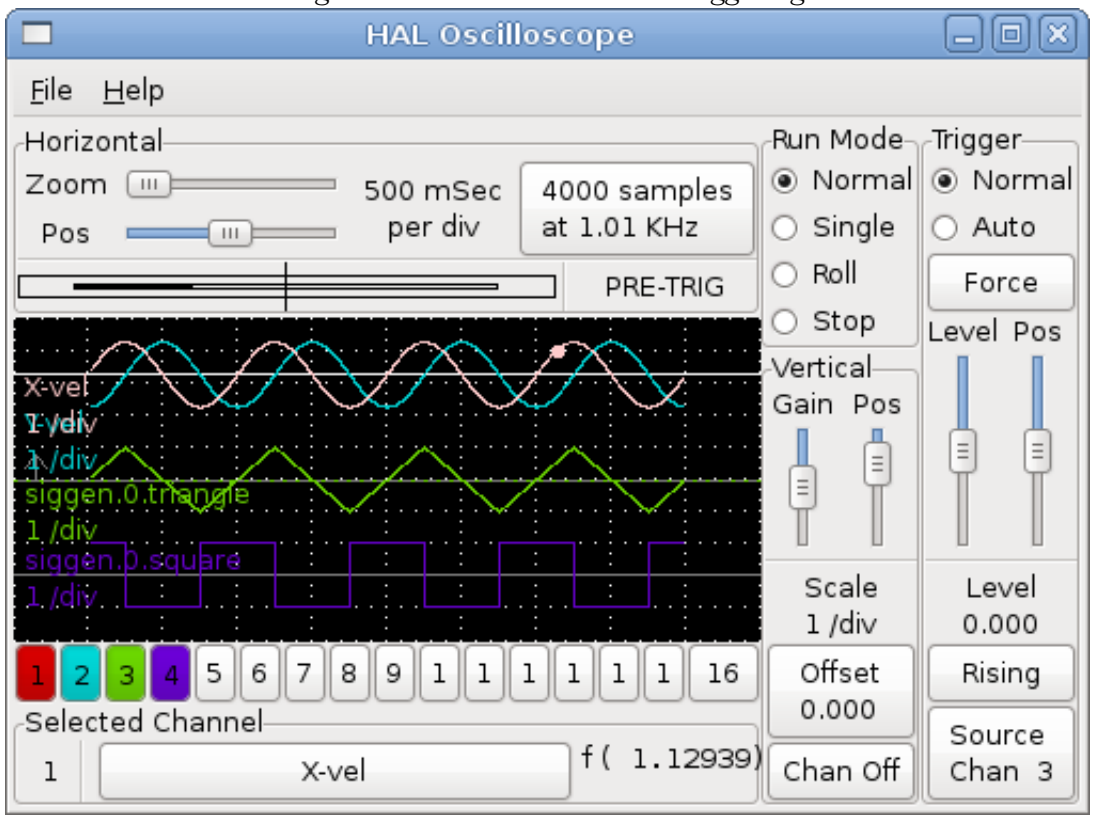


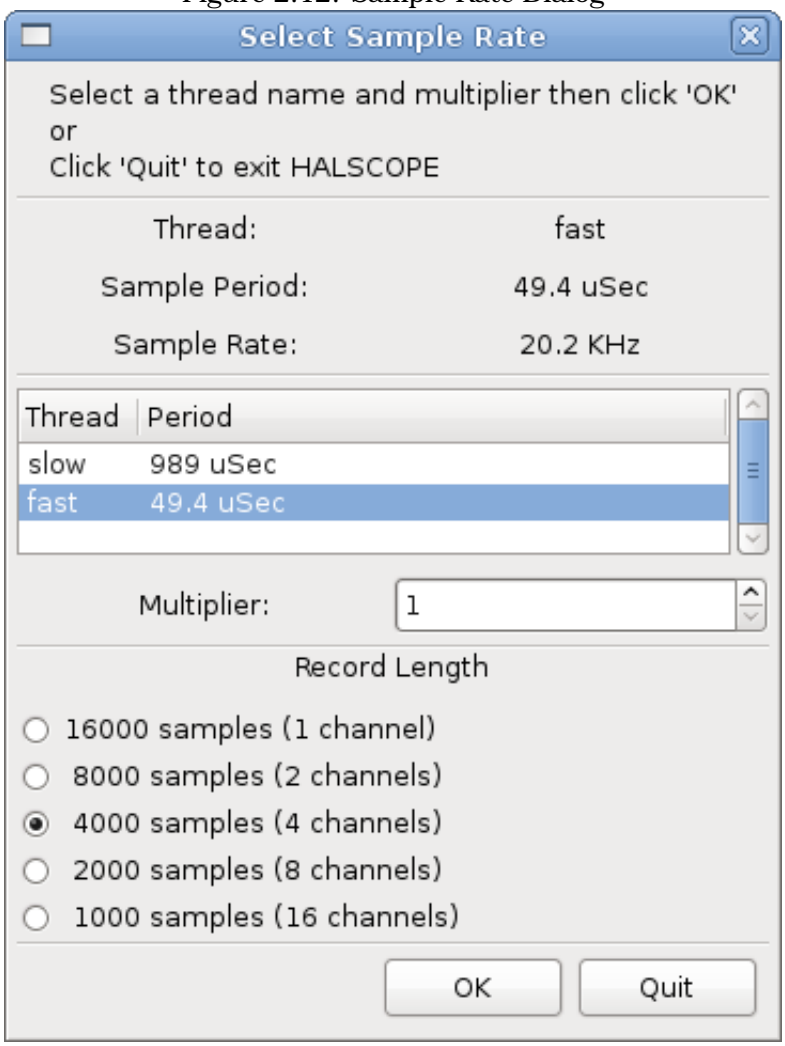
Figure 2.11: Waveforms with Triggering



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 50uS 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 50uS 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.

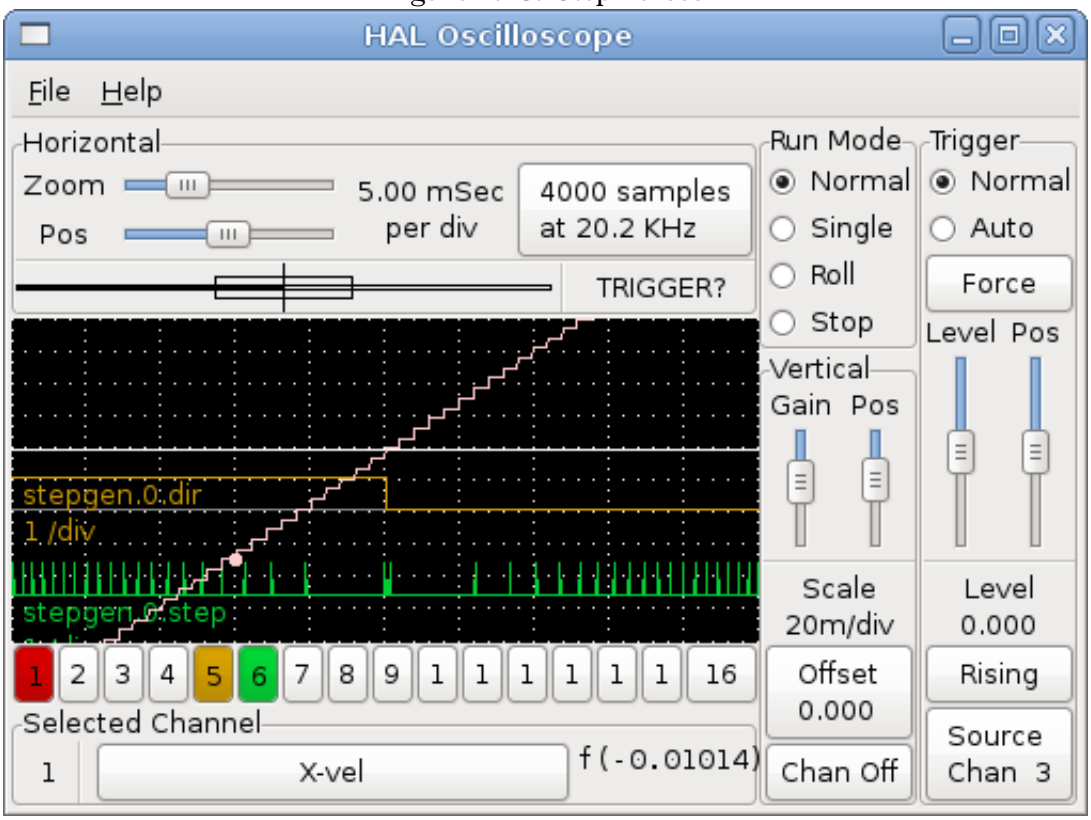
Figure 2.12: Sample Rate Dialog



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 5mS 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 20m per division to better see the change in the velocity command. The result should look like the following figure.

Figure 2.13: Step Pulses



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, such as

```
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 Information

3.1 Notation

3.1.1 Typographical Conventions

Command line examples are presented in **bold typewriter** font. Responses from the computer will be in typewriter font. As of early 2006, there are no longer commands that require root privileges, so all examples will be preceded by the normal user prompt, $. Text inside square brackets [like-this] is optional. Text inside angle brackets <like-this> represents a field that can take on different values, and the adjacent paragraph will explain the appropriate values. Text items separated by a vertical bar means that one or the other, but not both, should be present. All command line examples assume that you are in the emc2/ directory, and you configured/compiled emc2 for the run-in-place scenario. Paths will be shown accordingly when needed.

3.1.2 Names

All HAL entities are accessed and manipulated by their names, so documenting the names of pins, signals, parameters, etc, is very important. HAL names are a maximum of 41 characters long (as defined by HAL_NAME_LEN in hal.h). Many names will be presented in a general form, with text inside angle brackets <like-this> representing fields that can take on different values.

When pins, signals, or parameters are described for the first time, their names will be preceeded by their type in (SMALL CAPS) and followed by a brief description. A typical pin definition will look something like these examples:

- (BIT) parport.<portnum>.pin-<pinnum>-in – The HAL pin associated with the physical input pin <pinnum> on the 25 pin D-shell connector.
- (FLOAT) pid.<loopnum>.output – The output of the PID loop.

At times, a shortened version of a name may be used - for example the second pin above might be referred to simply as .output when it can be done without causing confusion.

3.2 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 neccessary information. To do that, all HAL modules should follow these rules:

- Dots (".") separate levels of the heirarchy. This is analogous to the slash ("/") in a filename.
- Hypens ("-") separate words or fields in the same level of the heirarchy.
- HAL modules should not use underscores or "MixedCase". [1]
- Use only lowercase letters and numbers in names.

3.3 Hardware Driver Naming Conventions[2]

3.3.1 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.[3]

<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.[4] 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,

[1] Underscores have all been removed, but there are still a few instances of mixed case, for example "pid.0.Pgain" instead of "pid.0.p-gain".

[2] Most drivers do not follow these conventions as of version 2.0. This chapter is really a guide for future development.

[3] Some devices use jumpers or other hardware to attach a specific ID to each board. Ideally, the driver provides a way for the user to specifically say "device-num 0 is the board with ID XXX", and the device numbers always start at 0. However at present some drivers use the board ID directly as the device number. That means it is possible to have a device number 2, without a device 0. This is a bug and will be fixed in version 2.1.

[4] One glaring 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.

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 chapter 4.

3.3.1.1 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.

3.3.2 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.[5]

<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.

3.3.2.1 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 ppmc board

[5]Note to driver programmers: do NOT implement separate functions for different I/O types unless they are interruptable 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.

Chapter 4

Canonical Device Interfaces[1]

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.

4.1 Digital Input

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

4.1.1 Pins

- (BIT) **in** – State of the hardware input.
- (BIT) **in-not** – Inverted state of the input.

4.1.2 Parameters

- None

4.1.3 Functions

- (FUNCT) **read** – Read hardware and set **in** and **in-not** HAL pins.

4.2 Digital Output

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

4.2.1 Pins

- (BIT) **out** – Value to be written (possibly inverted) to the hardware output.

[1] As of version 2.0, most of the HAL drivers don't quite match up to the canonical interfaces defined here. In version 2.1, the drivers will be changed to match these specs.

4.2.2 Parameters

- (BIT) **`invert`** – If TRUE, **`out`** is inverted before writing to the hardware.

4.2.3 Functions

- (FUNCT) **`write`** – Read **`out`** and **`invert`**, and set hardware output accordingly.

4.3 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.

4.3.1 Pins

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

4.3.2 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).

4.3.3 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

4.4 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.

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.

4.4.1 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 1V
- (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.

4.4.2 Functions

(FUNCT) **write** – This causes the calculated value to be output to the hardware. If enable is false, then the output will be 0, regardles 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.

4.5 Encoder

The canonical encoder interface (I/O type field: **`encoder`**) provides the functionality needed for homing to an index pulse and doing spindle synchronization, as well as basic position and/or velocity control. This interface should be implementable regardless of the actual underlying hardware, although some hardware will provide "better" results. (For example, capture the index position to +/- 1 count while moving faster, or have less jitter on the velocity pin.)

4.5.1 Pins

- (S32) **`count`** – Encoder value in counts.
- (FLOAT) **`position`** – Encoder value in position units (see parameter "scale").
- (FLOAT) **`velocity`** – Velocity in position units per second.
- (BIT) **`reset`** – When True, force counter to zero.
- (BIT) **`index-enable`** – (bidirectional) When True, reset to zero on next index pulse, and set pin False.

The "index-enable" pin is bi-directional, and might require a little more explanation. If "index-enable" is False, the index 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. If "index-enable" is True, then when the next index pulse arrives, the encoder counter will be reset to zero, and the driver will set "index-enable" False. That will let the other component know that an index pulse arrived. This is a form of handshaking - the other component sets "index-enable" True to request a index pulse reset, and the driver sets it False when the request has been satisfied.

4.5.2 Parameters

- (FLOAT) **scale** – The scale factor used to convert counts to position units. It is in "counts per position unit". For example, if you have a 512 count per turn encoder on a 5 turn per inch screw, the scale should be 512*5 = 2560 counts per inch, which will result in "position" in inches and "velocity" in inches per second.
- (FLOAT) **max-index-vel** – (optional) The maximum velocity (in position units per second) at which the encoder can reset on an index pulse with +/- 1 count accuracy. This is an output from the encoder driver, and is intended to tell the user something about the hardware capabilities. Some hardware can reset the counter at the exact moment the index pulse arrives. Other hardware can only tell that an index pulse arrived sometime since the last time the read function was called. For the latter, +/- 1 count accuracy can only be achieved if the encoder advances by 1 count or less between calls to the read function.
- (FLOAT) **velocity-resolution** – (optional) The resolution of the velocity output, in position units per second. This is an output from the encoder driver, and is intended to tell the user something about the hardware capabilities. The simplest implementation of the velocity output is the change in postion from one call of the read function to the next, divided by the time between calls. This yields a rather coarse velocity signal that jitters back and forth between widely spaced possible values (quantization error). However, some hardware captures both the counts and the exact time when a count occurres (possibly with a very high resolution clock). That data allows the driver to calculate velocity with finer resolution and less jitter.

4.5.3 Functions

There is only one function, to read the encoder(s).

- (FUNCT) **read** – Capture counts, update position and velocity.

Chapter 5

Tools and Utilities

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 EMC2 from either source or a package. If you have compiled EMC2 for "run-in-place", the man page is not installed, but it is accessible. From the main EMC2 directory, do:

$ **man -M docs/man halcmd**

Chapter 2 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 a 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. Usefull when you want a lot of meters in a small amount of screen space.

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 2.5 in the tutorial chapter, which explains the basics.

Chapter 6

comp: a tool for creating HAL modules

6.1 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 emc2 source tree, or outside it.

For instance, the "ddt" portion of `blocks` is around 80 lines of code. The equivalent component is very short when written using the *comp* preprocessor:

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

and it can be compiled and installed very easily: by simply placing `ddt.comp` in `src/hal/components` and running '`make`', or by placing it anywhere on the system and running `comp --install ddt.comp`

6.2 Definitions

component A component is a single real-time module, which is loaded with `halcmd loadrt`. One `.comp` file specifies one component.

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)

6.3 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.

6.4 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 NAME* ([*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:

HALNAME An 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:

```
param rw bit zot=TRUE
"""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.

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`.

OPT, VALUE Depending on the option name OPT, the valid VALUEs vary. 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 0. 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* (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)

If 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.

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,

```
license "GPL";
```

For additional information on the meaning of MODULE_LICENSE() and additional license identifiers, see `<linux/module.h>`.

Starting in emc 2.3, this declaration is required.

AUTHOR Specify the author of the module for the documentation.

6.5 Per-instance data storage

variable *CTYPE NAME*;

variable *CTYPE NAME*[*SIZE*];

variable *CTYPE NAME* = *DEFAULT*;

variable *CTYPE NAME*[*SIZE*] = *DEFAULT*;

Declare a per-instance variable *NAME* 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`, bool, etc.
Access to array variables uses square brackets.

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

6.6 Other restrictions on comp files

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

6.7 Convenience Macros

Based on the items in the declaration section, *comp* creates a C structure called `struct 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 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

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 uses the variable `struct state *inst` to iterate 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.

6.8 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.

6.9 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**.

6.10 Compiling `.comp` files in the source tree

Place the `.comp` file in the source directory `emc2/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 `emc2/src/hal/components` and will be built except if emc2 is configured as a userspace simulator.

6.11 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 emc2 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.

6.12 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.

6.13 Examples

6.13.1 constant

This component functions like the one in 'blocks', including the default value of 1.0. The declaration "function _" creates functions named 'constant.0', etc.

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

6.13.2 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";
;;
#include <rtapi_math.h>
FUNCTION(_) { sin_ = sin(theta); cos_ = cos(theta); }
```

6.13.3 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;

;;
#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); }
```

6.13.4 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`.

6.13.5 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";
;;
int i;
for(i=3; i>0; i--) out(i) = out(i-1);
out(0) = in;
```

6.13.6 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;
;;
#include <unistd.h>

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

6.13.7 logic

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

```
component logic;
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";
;;
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 7

Creating Userspace Python Components with the 'hal' module

7.1 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
```

7.2 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–for instance, 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).

7.3 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 7.1: 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.

7.3.1 Changing the prefix

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

7.4 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']
```

7.4.1 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.

7.4.2 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.

7.5 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.

7.6 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 `http://pyserial.sourceforge.net/` (Ubuntu package name "python-serial", in the universe repository)
- Attach a LCDProc `http://lcdproc.omnipotent.net/`-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)

Appendix A

Legal Section

Appendix B

Legal Section

B.1 Copyright Terms

B.2 GNU Free Documentation License

GNU Free Documentation License Version 1.1, March 2000

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the

Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission. B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five). C. State on the Title page the name of the publisher of the Modified Version, as the publisher. D. Preserve all the copyright notices of the Document. E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices. F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below. G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice. H. Include an unaltered copy of this License. I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence. J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission. K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein. L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles. M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version. N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License,

provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http:///www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright (c) YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being LIST"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Index