GNU Bayonne Administration Guide

David Sugar
Open Source Telecom.
sugar@gnu.org, http://www.gnu.org/software/bayonne

2003-01-03

Contents

1 Introduction

This document is new and very incomplete. It currently has material leftover from the original Bayonne manual, but will be written in a better form to support the other two manuals, installation and scripting, as soon as this is possible to do.

1.1 History of Bayonne

GNU Bayonne was developed from ACS (Adjunct Communication Server), which had been created as a free telephony server for specialized IVR use in early 2000. ACS was originally started as GPL licensed multi-line voice telephony server, with the goal to be the most flexible and advanced telephony voice messaging server available. Bayonne expanded upon that mission to become a general purpose platform for deploying all forms of voice application services within the GNU project.

The most current release represents milestone #8, and represents the final form of GNU Bayonne for the 1.0 release, scheduled for early July. Between now and

July the existing package will be refined and this documentation will be extensively revised.

With Milestone #7 we introduced broad support for high density digital telephony hardware, primarily from Intel/Dialogic and Aculab. With high density digital (PRI) support, it became possible to use GNU Bayonne for various carrier class applications.

Milestone #6 introduced support for XML based scripting. XML support was created by directly transcoding XML content into the Bayonne ccScript virtual execution environment, and then running standard Bayonne applications.

Starting With milestone #5, Bayonne supported FreeBSD in addition to GNU/Linux (and possibly Solaris). 0.5.0 is the first release which demonstrability works under FreeBSD and also happens to support the Voicetronix 4 port analog DSP card which will be used as the core of Bayonne VoIP development in coming weeks. In addition, support for Aculab and Dialogic hardware has been initiated, though it is not yet complete.

The fourth Bayonne milestone featured many changes, starting with international support and the new French language vocabulary. There are many less visible changes that effect and expand the scope of DSO capabilities for the Bayonne server as well as greatly improve stability of the server. With 0.4.2, expanded support for initiating dial-out sessions has been added along with preliminary support for locating streams and the Dialogic SDK.

The 3rd milestone release of Bayonne is being made available from `ftp://www.voxilla.org/pub/bay` immediately. This release represents both a more rigorously tested server, and a "usable" or "deployable" server. An audio library has been recorded for use with the new phrasebook system, and some documentation has been drafted. Many features have also been clarified and greatly expanded upon for better use in deployable servers. A lot of bugs were also found and fixed between 0.2.1 and 0.3.

In particular a number of new issues were identified for both QuickNet and Pika hardware. In the case of QuickNet cards, it was found to be necessary to do a full close/reopen of the `/dev/phone` devices to reset the card otherwise they stop responding between calls, and this was done in 0.2. When the server is started as root, the /dev/phone devices are initially opened under root, but then the server reduces its privilege to "mail" and can no longer re-open /dev/phone. In 0.3, the server now resets file ownership of /dev/phone devices.

1.2 The Name

Some have asked why the package name was changed from ACS to Bayonne. I felt there are several reasons for doing this. First, I have received numerous complaints over the use of the name ACS in that it is similar to Al's Circuit Simulator and several other common packages. Also, I had originally wanted a name that was more than a 3 letter acronym and clearly distinct. Bayonne is taken from the name of the Bayonne "bridge", and represents the idea of this package as a bridge between the computer and telephony worlds. Finally, I wanted to bring across the idea that Bayonne is different than a "1.x" or "2.0" derived release of ACS.

2 Configuration Files

3 Directory Layouts

3.1 System Directories

System directories are installed by the distribution of the Bayonne package itself. While they can include application specific files that are placed there later, this is generally not recommended except as noted. Generally Bayonne may be installed in either /usr, or /usr/local. The following examples are based on GNU Bayonne being installed under /usr.

/usr/share/bayonne{/voice}
: Bayonne supplied default prompt library. These include phrasebook prompts and anything else found useful to distribute with Bayonne itself. User application supplied voice libraries should never be installed here.

/usr/share/bayonne{/voice}/appname
: Phrase library that is specific to a specified script file. For example, a "play xx" command in yy.scr would look for xx.au in /usr/share/aaprompts/yy{/voice}. This allows script files to have their own local voice libraries.

/home/bayonne{/voice}
: Maybe optionally created for single product turnkey servers. This directory is generally referenced by alt: prompts, as in "play m̃yprompt".

/usr/share/bayonne
: Default system scripts supplied with Bayonne. These include test scripts and any simple applications supplied entirely with Bayonne. Turnkey applications may place script files here, though it is not generally recommended except for turnkey product servers.

/usr/share/bayonne/aawrappers
: This is a special directory to hold user supplied executables that will be executed thru the bayonne_wrapper program. These programs are generally initiated under the apache user id and converted to the bayonne user id by bayonne_wrapper.

/usr/libexec/bayonne
: This is the default directory for Bayonne supplied TGI applications. Bayonne distributions will in time add a number of useful tgi libexec files. User supplied TGI applications must also be installed here, as these run under the Bayonne server's user id.

/usr/lib/bayonne/{version}/
: This is the install directory for all loadable modules and plugins. There is only one directory, and DSO's are assumed to be compiled against a specific Bayonne version. User created DSO's should be installed here.

3.2 VAR Directories

/var/lib/bayonne is used to host all modifiable files, both as distributed with Bayonne, and for user applications. Any reference to a partial path with at least two components will refer to here.

/var/lib/bayonne/tmp
: A place to record temporary files

/var/lib/bayonne/save
: A save filespace for serialized classes.

/var/lib/bayonne/prompts
: Used by the play and record script.

/var/lib/bayonne/maps
: Used for loadable map files.

/var/lib/bayonne/users
: Used to store user preference data.

/var/lib/bayonne/hunting
: Used to store hunt group preference data.

/var/lib/bayonne/lines
: Used to store line preference data.

3.3 HOME Directories

Bayonne itself has a home directory and user id, starting fairly recently. This is typically found as /home/bayonne. A simple "adduser -r bayonne" will create the Bayonne user. The Bayonne server itself runs under the bayonne group when started under root, and tgi executables (and wrappers) will execute with the unprivileged bayonne user id. users may install their own home directories as well. These may be placed under /.bayonne, and can hold a local bayonne script, /.bayonne/bayonnrc, as well as a user's own voice samples, tgi programs, etc. To activate this directory, the user must be added to the "bayonne" group in /etc/group.

/home/bayonne/apps
: Starting with 0.5.20, this is now the preferred base prefix for installing new Bayonne applications.

/home/bayonne/php
: A prefix for PHP include and drop files.

/home/bayonne/admin
: A working directory for plug-in PHP admin forms.

/home/bayonne/html
: The primary document root for Bayonne integrated web applications. These are typically execute using an Apache server running under the "bayonne" user id rather than the nobody or http user. This is used for applications that include web services like unified messaging. Other kinds of

Bayonne apps might integrate with an existing web application by using bayonne_wrappers instead. "Click-to-dial" typically uses wrappers rather than a seperate apache server.

3.4 Bayonne Home Directory

Starting with 0.5.20, /home/bayonne has become a standard prefix for installing applications. A number of directories exist for this purpose, and a number of special requirements are needed to use this fully.

/home/bayonne
: User supplied application scripts are installed here.

/home/bayonne{/voice}
: User applications gets their own private voice library, which is also fully multi-lingual.

3.5 User HOME Directories

When starting Bayonne under an ordinary user id, Bayonne will use the local user's home directory rather than /home/bayonne and /var/run, for various things. These are stored under ~/.bayonne. This has not been changed in awhile, and is only useful if one either always runs bayonne unprivileged, or if one is starting multiple instances for different users.

$HOME/.bayonnerc
: Can be used to override some things found in /etc/bayonne.conf.

$HOME/.bayonne{.ctrl,.nodes,.mixers}
: Local user privileged versions of /var/run files.

$HOME/.bayonne/schedule.conf
: User's private scheduler.

$HOME/.bayonne/tgi
: User's private tgi directory when non-privileged Bayonne is ran.

$HOME/.bayonne/aascripts
: User's private scripts.

$HOME/.bayonne/aaprompts/{%voice}
: User's private "$HOME" prompt path. Hence, a play $HOME/myprompt" should go to here.

$HOME/.bayonne/aaprompts/APPS/{%voice}
: When not running under a privileged user id, the Bayonne application specific working directory will be located in the user's home rather than in /home/bayonne/apps. Hence, a "play myapp::intro" will go to " /.bayonne/aaprompts/myapp/UsEngM/intro.au"

4 The Bayonne control script

Virtually all basic server administration functions are performed through a single script file, which runs and manages the bayonne service on your server. This script file is named "bayonne" and lives in /usr[/local]/sbin. This script can often be executed either by the root user or by a user under the effective id or with shared group permission with the "bayonne" user, which should have been created in /etc/passwd and /etc/groups.

To start "bayonne" in a runtime installation, it is often only nessisary to execute /usr/sbin/bayonne. The script will then run the server based on the current configuration file settings. The default driver specified in /etc/sysconfig/bayonne will be used for telephony control, and the server will start in deamon mode.

To shutdown the bayonne service, you can use the command "bayonne –down". This sends a shutdown request to the server, which then exits from daemon mode.

To start the server in runtime configuration for testing, you can use the command "bayonne –trace". This starts bayonne in trace mode, and runs the server in the foreground. All server activity will then be shown on the console.

To issue control requests to a bayonne server running on your machine in daemon mode, you can use "bayonne –control". Following "–control" you would enter a fifo control command, and one or more arguments. For example, to start an outbound call session from the daemon, you can use “bayonne –control start 2 myscript”.

To get the daemon to recompile scripts you can use “bayonne –compile” as a convenient shortcut. You can also use “bayonne –service [mode]” to set the service level of the daemon while running.

Since we no longer distribute most voice libraries with Bayonne itself, we now use the bayonne script to install these remotely from the gnu repository. To install the “RussianF” voice library, for example, one would use “bayonne –voices RussianF”. This will then fetch and install the specified voice library from ftp.gnu.org.

5 TGI Services

5.1 Introduction

The TGI system used in Bayonne is very similar to the CGI protocol used for integrating dynamic executables with web servers. The primary difference is that where CGI uses environment variables, Bayonne TGI uses standard input and standard output instead.

The TGI equivalent of “Hello World” in Perl using the TGI module is:

hello.scr:

```
answer 2 # pick up the phone
libexec 10 hello.pl
slog %return
```

hello.pl:

```
use TGI;
use lib $ENV{'SERVER_LIBEXEC'};
```

```
TGI::set(return => ’Hello World!’);
```

This will cause the message ’Hello World!’ to be slogged from Bayonne. Here’s a more complex example:
hello.scr:

```
answer 2
set %a 1
set %b 1
libexec 10 adder.pl %a %b
if % return -eq -1 ::error
slog %return # outputs ‘‘2’’

::error
play *::999 # ‘‘An error has occured, please see the server error log for details.’
slog ‘‘Something went wrong a couple lines ago.  Bailing out.’’
hangup
exit
```

adder.pl:

```
use TGI;
use lib $ENV{’SERVER_LIBEXEC’};
my $a = $TGI::QUERY{’a’};
my $b = $TGI::QUERY{’b’};
if(! defined $a || ! defined $b) {
  printf(‘‘Missing either A or B or both.’’);
  TGI::set(return => -1);
  exit(1);
}
TGI::set(return => $a + $b);
```

This should output the number 2. Note the extensive error checking.

5.2 TGI Errors

Telephony applications are very unlike other computer applications in that they need to be more “fail-soft” than a traditional windowed application. In other words, only very serious errors such as an inability to contact a database server or a critical transaction failure should prevent the user from continuing with the telephone call. If a possible error really isn't that serious and it's possible to continue with the phone call even if it occurs, make sure to insert a comment noting that fact in your code.

The rule of thumb for handling TGI errors is: hanging up on users without telling them why is just plain rude. Don't do it. Make sure that you have consistency in your error handlers and that you always play a speech message indicating that something went wrong before hanging up. It's a good idea to have the generic error message include an instruction for the caller to "call the people who are hosting the application with the current time" as that will help them search the error logs for the output that occurred during the failed call.

5.3 Return Values

Your TGI script passes return values in the TGI variable "return". For example, using the TGI.pm Perl module, you would write:

```
use TGI.pm;
use lib $ENV{'SERVER_LIBEXEC'};

TGI::set(return => 0); # 0 is the return value
exit(0);
```

Variables set with the TGI::set method become accessible in the calling ccScript via the ccScript variable of the same name. In this example, the %return variable will be set to 0. These return values can be numbers or characters. You can return as many or as few variables as you want, but it's typical for an application to return a numeric error code in the %return variable so that error checking code after libexecs is more consistent.

6 Wrapper

6.1 Introduction

So, you want to integrate Bayonne with the Web? There are numerous reasons for wanting to do this. Perhaps you are building a unified messaging server with Bayonne, or are trying to build a "click to dial" front end for a Bayonne call agent server or as a compliment to online customer service. While on the surface these may sound similar, they actually are not at all in operation or implementation.

6.2 Using Bayonne with an existing site

Starting with 0.5.15, a new Bayonne executable, the "wrapper", was introduced. This is typically used to bridge between a web server, typically running under user id "nobody" or "http", and a Bayonne server, typically running under user id "bayonne". The wrapper (bayonne_wrapper) acts as a simple "sudo" facility, where you specify which scripts the bayonne wrapper will be permitted to execute based on the original requestors "user id".

This is done in the [wrappers] section of Bayonne.conf. Basically one enters the user id and a list of permitted scripts. This list of permitted scripts always execute from /usr(/local)/share/bayonne/aawrappers. Bayonne_wrapper itself runs as a setuid executable, and the scripts or shell commands found in aawrappers are given the user id of the Bayonne server. These scripts also receive useful information in their environment space that is similar to "TGI", but without port specific information.

In particular, a wrapper initiated image receives the server software and version information (BAYONNE_SOFTWARE, BAYONNE_PROTOCOL, BAYONNE_PLATFORM), the token separator to use for fifo control (BAYONNE_TOKEN), the Bayonne node id (BAYONNE_NODE), and the location of the Bayonne fifo and control files (BAYONNE_CONTROL). In addition, the defined trunk group policies are passed as "BAYONNE_POLICIES". No "PORT_xxx" symbols are passed since the wrapper script is not necessarily executing on behalf of a specific and dedicated port resource.

NOTE: These symbols were originally named "SERVER_" in bayonne_wrapper, but were changed to "BAYONNE_" starting with 0.5.17 to avoid conflicts with CGI initiated scripts.

A web server or any other application can use wrapper to invoke executables in aawrappers that manipulate the Bayonne server in a controlled and secure fashion. A wrapper can be invoked thru a CGI and then receive both the web server's CGI state information, and the Bayonne symbol set. A CGI executed wrapper of course can send html back thru stdout to form a web page for the user.

Starting with "0.5.17", Bayonne wrappers can also be used to execute scripts that require interpreters in a fairly direct fashion. For example, if we have a "webdial.cgi" script written in Perl, you will be able to specify as the first line of your script file:

```
#!/usr/sbin/bayonne --wrapper -perl
```

This will re-execute the script with the Perl interpreter under the specified user id of Bayonne. Similarly, one can then use -bash for /bin/bash, and others for other common shells and interpreters. This provides a simple and direct means to integrate cgi operation with a Bayonne server. In fact, one could use the alias map of apache or other web servers to map the aawrappers directory under a "/bayonne" url as a convenient means to access commonly used wrappers in this way, and in the future some scripts will be provided in the default distribution of Bayonne for this purpose.

6.3 Running Bayonne separate from your web server

When you have an existing site, you may not wish to run web servers on Bayonne themselves. You might have a e-commerce "site" you wish to add Bayonne to provide some service, but this need not mean that you install Bayonne on your web server farm, or that Bayonne need even run a publically accessible web server of it's own that has seperate or additional security issues.

Bayonne –wrapper support the use of ”ssh” to invoke a bayonne wrapper application on a Bayonne server from a remote machine. This is done by placing a truncated /etc/bayonne.conf file which indicates the node name of the Bayonne server it should contact, and the wrapper permissions to use. The ”bayonne_wrapper” will assume to automatically use ssh if the /usr/share/aawrappers directory is not found on the local machine.

On the remote (web server) machine, you will need to create a bayonne ”user id”. This user id should contain a .ssh/config ”entry” with the same name as the Bayonne ”node” id in the local /etc/bayonne.conf file, and should list the hostname to actually contact.

You will also need a passwordless ssh key to use between the Bayonne server and the web server. This will allow the bayonne_wrapper to automatically hop between the machines.

The bayonne_wrapper can be symlinked or placed in the web server's cgi directory under the name you will execute the program in ”aawrapper” on the Bayonne server under. Hence, you would take bayonne_wrapper, call it for example ”webdial.cgi”, and place it on the cgi directory of your web server. Since there is no local aawrappers directory, it will initiate a ssh to the Bayonne box and then execute the ”real” webdial.cgi from the aawrappers directory there.

SSH does not automatically preserve environment state variables. While bayonne_wrapper has supported ssh for awhile, it is only with 0.5.17 and above does it now preserves key CGI environment variables when it hops between the local and remote machine.

6.4 More on running web services on a Bayonne server

While wrappers do allow one to execute Bayonne in conjunction with a web server, this is not necessarily the most efficient manner to do this. A higher demand application might suffer from the performance hit of executing cgi. However, there is one other option currently available.

Certainly one can execute and access Bayonne resources directly if one places a web server on a machine running Bayonne and then has it execute under the same user id that Bayonne does. This allows one to use one of the various highly efficient "mod_xxx" interpreters (like mod_php or mod_perl) to build a web integrated Bayonne service rather than requiring cgi wrappers everywhere.

Incidentally, Bayonne can of course alternately be executed under the same user id as the web server. Which one to change probably depends on your actual need. You can of course also run multiple web server daemons, and have one of them execute under Bayonne's user id.

7 System Architecture

7.1 Introduction

This section provides an overview of the system architecture of a GNU Bayonne server. This overview while useful is not necessary to understand it's operation or successfully deploy services with it. It is meant to help better understand the functionality of Bayonne for those that are interested.

7.2 Component Libraries

To create GNU Bayonne we needed a portable foundation written in C++. I wanted to use C++ for several reasons. First, the highly abstract nature of the driver interfaces seemed very natural to use class encapsulation for. Second, I found I personally could write C++ code faster and more bug free than I could write C code.

Why we choose not to use an existing framework is also simple to explain. We knew we needed threading, and socket support, and a few other things. There were no single framework that did all these things except a few that were very large and complex which did far more than we needed. We wanted a small footprint for Bayonne, and the most adaptable framework that we found at the time typically added several megabyte of core image just for the runtime library.

GNU Common C++ (originally APE) was created to provide a very easy to comprehend and portable class abstraction for threads, sockets, semaphores, exceptions, etc. This has since grown into it's own and is now used as a foundation of a number of projects as well as being a part of GNU.

In addition to having portable C++ threading, we needed a scripting engine. This scripting system had to operate in conjunction with a non-blocking state-transition call processing system. It also had to offer immediate call response, and support several hundred to a thousand instances running concurrently in one server image.

Many extension languages assume a separate execution instance (thread or process) for each interpreter instance. These were unsuitable. Many extension languages assume expression parsing with non-deterministic run time. An expression could invoke recursive functions or entire subprograms for example. Again, since we wanted not to have a separate execution instance for each interpreter instance, and have each instance respond to the leading edge of an event call-back from the telephony driver as it steps through a state machine, none of the existing common solutions like TCL, Perl, guile, etc, would immediately work for us. Instead, we created a non-blocking and deterministic scripting engine, GNU ccScript.

GNU ccScript is unique in several ways. It is step executed, and is non-blocking. Statements either execute and return immediately, or they schedule their completion for a later time with the executive. A given "step" is executed, rather than linearly. This allows a single thread to invoke and manage multiple interpreter instances. While GNU Bayonne can support interacting with hundreds of simultaneous telephone callers on high density carrier scale hardware, we do not require hundreds of native "thread" instances running in the server, and we have a very modest CPU load.

Another way GNU ccScript is unique is in support for memory loaded scripts. To avoid delay or blocking while loading scripts, all scripts are loaded and parsed into a virtual machine structure in memory. When we wish to change scripts, a brand new virtual machine instance is created to contain these scripts. Calls currently in progress continue under the old virtual machine and new callers are offered the new virtual machine. When the last old call terminates, the entire old virtual machine is then disposed of. This allows for 100 % uptime even while services are modified.

Finally, GNU ccScript allows direct class extension of the script interpreter. This allows one to easily create a derived dialect specific to a given application, or even specific to a given GNU Bayonne driver, simply by deriving it from the core language through standard C++ class extension.

7.3 Class instantiation

Since each vendor of telephony hardware has chosen to create their own unique and substantial application library interface, we needed GNU Bayonne to sit above these and be able to abstract them. Ultimately we choose to create a driver plug-in architecture to do this. What this means is that you can get a card and API from Aculab, for example, write your application in GNU Bayonne using it, and later choose, say, to use Intel telephony hardware, and still have your application run, unmodified. This has never been done in the industry widely because many of these same telephony hardware manufacturers like to produce their own middle-ware solutions that lock users into their products.

To do this, we made Bayonne itself operate as a series of base classes, and then create derived classes which implement functionality as separate plug-in modules with static initialized objects. The server exports it's own symbol map much like a library, and so when the derived plug-in is loaded, the static objects are instantiated, and their constructors are called. These constructors link with and are resolved to base class constructors in the server image which are then automatically invoked and are used to register the plug-in with the server.

The server itself also instantiates some objects at startup even before main() runs. These are typically objects related to plug-in registration or parsing of the configuration file.

8 Driver Architecture

8.1 Introduction

The Bayonne driver architecture uses a “plug-in” style interface to abstract driver-specific details from the ccScript application accessing the hardware. Each plug-in provides state handlers for each call state (ringing, playing audio, conferenced, etc) and possibly an event thread.

8.2 Driver Lifecycle

Each driver begins life when plugins.loadDriver() is called from initial() in server.cpp. A configuration lookup is done on the driver key (set in /etc/bayonne.conf) and the path to the .ivr file for the specified driver is calculated. The .ivr file is then loaded via a call to dlopen() (or the equivalent on your system) via Common C++'s DSO class.

All Bayonne drivers have a static instance of their driver class defined at the bottom of driver.cpp. For example, the Voicetronix driver has the following statement at the bottom of its driver.cpp file:

```
VPBDriver vpbivr;
```

The VPBDriver class (which represents the driver for all Voicetronix cards in the system) inherits from the Driver class, so a call to the Driver class constructor is triggered when the Voicetronix driver is opened. This is a feature of most UNIX implementations of C++. On those platforms (which are at least BSD and Linux), if at run-time, a shared object containing a static instance of a class is loaded, that class' constructor will be called as well as constructors for any parent classes.

The Driver class constructor is defined as follows:

```
Driver::Driver() :
aaScript()
{
        active = false;
        if(driver)
                throw(this);

        groups = NULL;
        driver = this;
        status = NULL;

        memset(spans, 0, sizeof(spans));
        memset(cards, 0, sizeof(cards));
}
```

The “driver” variable seen here is of type “Driver *” and is a static global variable defined in bayonne.h. After pointing the “driver” global pointer to the instance of the driver containined in the driver plug-in, the result is that the rest of the bayone code (mainly in server.cpp) can now access the underlying driver via the abstraction functions defined in class Driver. The two most important functions are start() and stop(), which allow Bayonne to control a driver’s active state. There are also several “get” functions defined in class Driver which allows Bayonne to interrogate a driver’s capabilities and port count. To recap, here’s a summary of the driver start-up sequence:

1) The driver DSO is loaded via plugins.loadDrivers().

2) This results in a call to Driver::Driver().

3) This causes the global “driver” pointer to point to the newly created instance of the child class driver.

4) The child class driver constructor (e.g. VPBDriver::VPBdriver()) is called. This is where ports actually get initialized and the hardware is prepared to begin receiving calls.

5) Bayonne calls driver-¿start().

6) This results in a call to the driver-specific start() function, because Driver::start() is declared virtual. In the case of the Voicetronix driver, this is VPBDriver::start(). Any event threads are started up and the hardware begins processing calls.

8.3 Win32

One of the reasons why the Win32 build is not going to work for a while is that David Sugar and Jason Spence have determined via a few test cases that it is not possible to link C++ object files into i386 PE DLLs (the format used by DSOs on Win32) unless all of the classes have their parent class constructors symbols resolved in the DLL. That means that the current way of doing things where the Driver class code is in the main Bayonne executable and the driver-specific class code (e.g. VPBDriver) is in a DSO is simply not possible on Win32.

The proposed solution is to create a libbayonne.so and have a simple wrapper program that starts up Bayonne and links in the drivers at run-time. This has not been completed as of yet.

8.4 State Handlers

Bayonne, like many telephony applications, uses a state machine to model the behavior of calls. Each port starts out in the "idle" state, and changes to other states via Driver::postEvent() (if the current thread is not the thread that will be executing the handler) or by simply setting the global "handler" variable for that port (the current thread will be executing it).

Each state's state handler is a function of the name "Trunk::stateHandler", where Trunk is the driver's trunk class (DialogicTrunk, VPBTrunk), and "state" is replaced with the state name that will be handled. For example, the ring handler for the capi driver is CapiTrunk::ringHandler. The state handlers are handed arguments of type TrunkEvent, which contains the event ID and a union containing data the event handler will need to serve the event (such as digits to be dialed or timeout durations).

Each state handler is initially called with an event type of TRUNK_ENTER_STATE by postEvent, which gives the state handler a chance to set up anything it needs to service other events while in that state. postEvent and the event thread(s) (if any) then continue to call the event handler as events come in and need to be processed. Note that the execution of the state handler (or rather the "handler" function pointer defined in the Trunk coass for the driver) is protected by a mutex so that it is not possible to have more than one state handler executing concurrently on a given trunk.

However, state handlers do execute concurrently across trunks. That is, a machine with 4 analog trunks will most likely have four trunk threads going simultaneously, possibly served by a single event thread. The programmer is urged to keep in mind the current thread context whenever dealing with data that could possibly be accessed by several trunks simultaneously, and use mutexes as appropriate to protect the data.

8.5 State Handler Reference

Here is the list of states currently present in Bayonne. Note that this list is generated by grepping the source code, so it may not be entirely accurate. Among other things, the dummy driver only has stubs for its states, and the list of "Comes from" and "Goes to" states is for all drivers. This means that some of the states listed in those lists may not actually call or go to the state they are listed under.

accept
Supported by: dialogic
Comes from: step
Goes to: step Accepts an incoming ISDN call and opens up a timeslot for it. See reject.

answer
Supported by: aculab, capi20, dialogic, dummy, phonedev, pika, vpb
Comes from: step
Goes to: step
Places the port off-hook.

busy
Supported by: dialogic, dummy, phonedev, pika, vmodem, vpb
Comes from: idle
Goes to: idle
Busies out the port so incoming calls are denied. For analog cards, places the port off-hook. For digital cards, FIXME does what?

collect
Supported by: aculab, capi20, dialogic, dummy, phonedev, pika, vpb
Comes from: step
Goes to: step
Begins collecting digits into the digit buffer. When the specified number of digits has been collected or the specified timeout has occured, stops collecting digits and continues executing the script.

detect
Supported by: capi20 (broken), pika (broken)
Comes from: dial, step
Goes to: step
For the Pika driver, handles tone events from the event thread. The reference in the capi20 driver is an anomaly and will be removed.

dial
Supported by: aculab, capi20, dialogic, dummy, phonedev, pika, vpb
Comes from: flash, step
Goes to: step
Dials DTMF digits.

duplex
Supported by: capi20, pika
Comes from: step
Goes to: step, exit
Begins recording and playing audio. Its presence in the capi20 driver is an anomaly.

exit
Supported by: pika
Comes from: duplex, play, record
Goes to: play
FIXME: not sure

flash
Supported by: aculab (broken),

Comes from: step
Goes to: step, dial
Does a hookflash.

flashon
Supported by: capi20, dialogic, dummy, phonedev, pika, vpb
Comes from: step
Goes to: dial, step, flashoff
Places the port temporarily on-hook to start a hookflash.

flashoff
Supported by: dialogic, dummy, phonedev, pika, vpb
Comes from: flashon
Goes to: dial, step
Places the port back off-hook at the end of a hookflash.

hangup
Supported by: aculab, capi20, dialogic, dummy, oh323, phonedev, pika, rtp, vpb
Comes from: hangup, seize, dial, step
Goes to: idle
Places the port on-hook (analog cards) or shuts down the timeslot (digital cards).

idle
Supported by: all drivers (this is a required state)
Comes from: driver startup, hangup, ring
Goes to: busy, seize, step, release, ring
Stops activity on the port and shuts down any executing script images.

join
Supported by: aculab, capi20, dialogic, pika, vpb
Comes from: step
Goes to: step
Joins two ports together so they can hear and speak to each other. The target port must be in the “wait” state.

load
Supported by: aculab, capi20, dialogic, dummy, phonedev, pika, vpb
Comes from: step
Goes to: step
Loads an XML service. See the section on web services in this manual.

play
Supported by: aculab, capi20, dialogic, dummy, phonedev, pika, vpb

Comes from: step
Goes to: step
Begins playing audio from a file.

playwait
Supported by: aculab, capi20, dialogic, dummy, phonedev, pika, vpb
Comes from: step
Goes to: step, play
Waits for the completion of a play service thread.

record
Supported by: aculab, capi20, dialogic, dummy, phonedev, pika, vpb
Comes from: step
Goes to: step
Begins recording audio to a file.

reject
Supported by: dialogic
Comes from: step
Goes to: step
Rejects an incoming call via the digital signalling channel. See accept.

release
Supported by: dialogic
Comes from: (nowhere)
Goes to: idle
FIXME: Doesn't seem to be called from anywhere in the Dialogic driver.

ring
Supported by: aculab, capi20, dialogic, dummy, phonedev, pika, vmodem, vpb
Comes from: idle
Goes to: step, idle
Recognizes an incoming ring event and increments the ring counter, then determines whether to start up a script. In some drivers (such as Voicetronix), captures caller ID.

rtp
Supported by: vpb (broken)
Comes from: (broken)
Goes to: (broken)
Connects a call with a RTP stream. Doesn't work right now.

seize
Supported by: dialogic, dummy, phonedev, vmodem, vpb
Comes from: idle
Goes to: step, hangup
Grabs a port in preparation for outbound dialing.

sleep
Supported by: aculab, capi20, dialogic, dummy, phonedev, pika, vpb
Comes from: step
Goes to: step
Waits for a specified duration or number of rings.

step
Supported by: all drivers (this is a required state)
Comes from: all states
Goes to: most states
Executes the next step in the script image and changes the current state if needed. This is probably the most frequently used state.

tone
Supported by: aculab, capi20, dialogic, pika, vpb
Comes from: step
Goes to: step
Generates a tone of user-specified frequency and duration.

wait
Supported by: phonedev, vpb
Comes from: step
Goes to: step, dial
FIXME: I'm not sure what's going on in the phonedev driver, but the vpb driver is placed into the wait state in preparation for a join from another port.

9 Copyright