CHAPTER 9

The Asterisk Gateway Interface (AGI)

Even he, to whom most things that most people
would think were pretty smart were pretty dumb,
thought it was pretty smart.
—Douglas Adams, *The Salmon of Doubt*

The Asterisk Gateway Interface, or AGI, provides a standard interface by which external programs may control the Asterisk dialplan. Usually, AGI scripts are used to do advanced logic, communicate with relational databases (such as PostgreSQL or MySQL), and access other external resources. Turning over control of the dialplan to an external AGI script enables Asterisk to easily perform tasks that would otherwise be difficult or impossible.

This chapter covers the fundamentals of AGI communication. It will not teach you how to be a programmer—rather, we'll assume that you're already a competent programmer, so that we can show you how to write AGI programs. If you don't know how to do computer programming, this chapter probably isn't for you, and you should skip ahead to the next chapter.

Over the course of this chapter, we'll write a sample AGI program in each of the Perl, PHP, and Python programming languages. Note, however, that because Asterisk provides a standard interface for AGI scripts, these scripts can be written in almost any modern programming language. We've chosen to highlight Perl, PHP, and Python because they're the languages most commonly used for AGI programming.

Fundamentals of AGI Communication

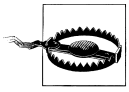
Instead of releasing an API for programming, AGI scripts communicate with Asterisk over communications channels (file pointers, in programming parlance) known as `STDIN`, `STDOUT`, and `STDERR`. Most computer programmers will recognize these channels, but just in case you're not familiar with them we'll cover them here.

What Are STDIN, STDOUT, and STDERR?

STDIN, STDOUT, and STDERR are channels by which programs in Unix-like environments receive information from and send information to external programs. STDIN, or "standard input," is the information that is sent to the program, either from the keyboard or from another program. In our case, information coming from Asterisk itself comes in on the program's STDIN file handle. STDOUT, or "standard output," is the file handle that the AGI script uses to pass information back to Asterisk. Finally, the AGI script can use the STDERR ("standard error") file handle to write error messages to the Asterisk console.

Let's sum up these three communications concepts:

- An AGI script reads from STDIN to get information from Asterisk.
- An AGI script writes data to STDOUT to send information to Asterisk.
- An AGI script may write data to STDERR to send debug information to the Asterisk console.



At this time, writing to STDERR from within your AGI script writes the information only to the *first* Asterisk console—that is, the first Asterisk console started with the -c or -r parameters.

This is rather unfortunate, and will hopefully be remedied soon by the Asterisk developers.

If you're using the *safe_asterisk* program to start Asterisk (which you probably are), it starts a remote console on TTY9. (Try pressing Ctrl-Alt-F9, and see if you get an Asterisk command-line interface.) This means that all the AGI debug information will print on only that remote console. You may want to disable this console in *safe_asterisk* to allow you to see the debug information in another console. (You may also want to disable that console for security reasons, as you might not want just anyone to be able to walk up to your Asterisk server and have access to a console without any kind of authentication.)

The Standard Pattern of AGI Communication

The communication between Asterisk and an AGI script follows a predefined pattern. Let's enumerate the steps, and then we'll walk through one of the sample AGI scripts that come with Asterisk.

When an AGI script starts, Asterisk sends a list of variables and their values to the AGI script. The variables might look something like this:

```
agi_request: test.py
agi_channel: Zap/1-1
agi_language: en
agi_callerid:
agi_context: default
agi_extension: 123
agi_priority: 2
```

After sending these variables, Asterisk sends a blank line. This is the signal that Asterisk is done sending the variables and it is time for the AGI script to control the dialplan.

At this point, the AGI script sends commands to Asterisk by writing to `STDOUT`. After the script sends each command, Asterisk sends a response that the AGI script should read. This action (sending commands to Asterisk and reading the responses) can continue for the duration of the AGI script.

You may be asking yourself what commands you can use from within your AGI script. Good question—we'll cover the basic commands shortly.*

Calling an AGI Script from the Dialplan

In order to work properly, your AGI script must be executable. To use an AGI script inside your dialplan, simply call the `AGI( )` application, with the name of the AGI script as the argument, like this:

```
exten => 123,1,Answer( )
exten => 123,2,AGI(agi-test.agi)
```

AGI scripts often reside in the AGI directory (usually located in */var/lib/asterisk/agi-bin*), but you can specify the complete path to the AGI script.

AGI( ), EAGI( ), DeadAGI( ), and FastAGI( )

In addition to the `AGI( )` application, there are several other AGI applications suited to different circumstances. While they won't be covered in this chapter, they should be quite simple to figure out once you understand the basics of AGI scripting.

The `EAGI( )` (enhanced AGI) application acts just like `AGI( )`, but allows your AGI script to read the inbound audio stream on file descriptor number three.

The `DeadAGI( )` application is also just like `AGI( )`, but it works correctly on a channel that is dead (i.e., a channel that has been hung up). As this implies, the regular `AGI( )` application doesn't work on dead channels.

The `FastAGI( )` application allows the AGI script to be called across the network, so that multiple Asterisk servers can call AGI scripts from a central location.

In this chapter, we'll first cover the sample *agi-test.agi* script that comes with Asterisk (which was written in Perl), then write a weather report AGI program in PHP, and then finish up by writing an AGI program in Python to play a math game.

* To get a list of available AGI commands, type **`show agi`** at the Asterisk command-line interface. You can also refer to Appendix C for an AGI command reference.

Writing AGI Scripts in Perl

Asterisk comes with a sample AGI script called *agi-test.agi*. Let's step through the file while we cover the core concepts of AGI programming. While this particular script is written in Perl, please remember that your own AGI programs may be written in almost any programming language. Just to prove it, we're going to cover AGI programming in a couple of other languages later in the chapter.

Let's get started! We'll look at each section of the code in turn, and describe what it does.

```
#!/usr/bin/perl
```

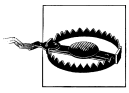
This line tells the system that this particular script is written in Perl, so it should use the Perl interpreter to execute the script. If you've done much Linux or Unix scripting, this line should be familiar to you. This line assumes, of course, that your Perl binary is located in the */usr/bin/* directory. Change this to match the location of your Perl interpreter.

```
use strict;
```

`use strict` tells Perl to act, well, strict about possible programming errors, such as undeclared variables. While not absolutely necessary, enabling this will help you avoid common programming pitfalls.

```
$|=1;
```

This line tells Perl not to buffer its output—in other words, that it should write any data immediately, instead of waiting for a block of data before outputting it. You'll see this as a recurrent theme throughout the chapter.



You should *always* use unbuffered output when writing AGI scripts. Otherwise, your AGI may not work as expected, because Asterisk may be waiting for the output of your program, while your program thinks it has sent the output to Asterisk and is waiting for a response.

```
# Set up some variables
my %AGI; my $tests = 0; my $fail = 0; my $pass = 0;
```

Here, we set up four variables. The first is a hash called `AGI`, which is used to store the variables that Asterisk passes to our script at the beginning of the AGI session. The next three are scalar values, used to count the total number of tests, the number of failed tests, and the number of passed tests, respectively.

```
while(<STDIN>) {
        chomp;
        last unless length($_);
        if (/^agi_(\w+)\:\s+(.*)$/) {
                $AGI{$1} = $2;
        }
}
```

As we explained earlier, Asterisk sends a group of variables to the AGI program at startup. This loop simply takes all of these variables and stores them in the hash named AGI. They can be used later in the program or simply ignored, but they should always be read from STDIN before continuing on with the logic of the program.

```
print STDERR "AGI Environment Dump:\n";
foreach my $i (sort keys %AGI) {
        print STDERR " -- $i = $AGI{$i}\n";
}
```

This loop simply writes each of the values that we stored in the AGI hash to STDERR. This is useful for debugging the AGI script, as STDERR is printed to the Asterisk console.*

```
sub checkresult {
        my ($res) = @_;
        my $retval;
        $tests++;
        chomp $res;
        if ($res =~ /^200/) {
                $res =~ /result=(-?\d+)/;
                if (!length($1)) {
                        print STDERR "FAIL ($res)\n";
                        $fail++;
                } else {
                        print STDERR "PASS ($1)\n";
                        $pass++;
                }
        } else {
                print STDERR "FAIL (unexpected result '$res')\n";
                $fail++;
        }
```

This subroutine reads in the result of an AGI command from Asterisk and decodes the result to determine whether the command passes or fails.

Now that the preliminaries are out of the way, we can get to the core logic of the AGI script.

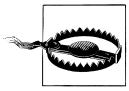
```
print STDERR "1.  Testing 'sendfile'...";
print "STREAM FILE beep \"\"\n";
my $result = <STDIN>;
&checkresult($result);
```

This first test shows how to use the STREAM FILE command. The STREAM FILE command tells Asterisk to play a sound file to the caller, just as the Background( ) application does. In this case, we're telling Asterisk to play a file called *beep.gsm*.†

* Actually, to the first spawned Asterisk console (i.e., the first instance of Asterisk called with the –c or –r option). If *safe_asterisk* was used to start Asterisk, the first Asterisk console will be on TTY9, which means that you will not be able to view AGI errors remotely.

† Asterisk automatically selects the best format, based on translation cost and availability, so the file extension is never used in the function.

You will notice that the second argument is passed by putting in a set of double quotes, escaped by backslashes. Without the double quotes to indicate the second argument, this command does not work correctly.



You must pass *all required arguments* to the AGI commands. If you want to skip a required argument, you must send empty quotes (properly escaped in your particular programming language), as shown above. If you don't pass the required number of arguments, your AGI script will not work.

You should also make sure you pass a line feed (the `\n` on the end of the `print` statement) at the end of the command.

After sending the `STREAM FILE` command, this test reads the result from `STDIN` and calls the `checkresult` subroutine to determine if Asterisk was able to play the file. The `STREAM FILE` command takes three arguments, two of which are required:

- The name of the sound file to play back
- The digits that may interrupt the playback
- The position at which to start playing the sound, specified in number of samples (optional)

In short, this test told Asterisk to play back the file named *beep.gsm*, and then checked the result to make sure the command was successfully executed by Asterisk.

```
print STDERR "2.  Testing 'sendtext'...";
print "SEND TEXT \"hello world\"\n";
my $result = <STDIN>;
&checkresult($result);
```

This test shows us how to call the `SEND TEXT` command, which is similar to the `SendText( )` application. This command will send the specified text to the caller, if the caller's channel type supports the sending of text.

The `SEND TEXT` command takes one argument: the text to send to the channel. If the text contains spaces (as in the example above), the argument should be encapsulated with quotes, so that Asterisk will know that the entire text string is a single argument to the command. Again, notice that the quotation marks are escaped, as they must be sent to Asterisk, not used to terminate the string in Perl.

```
print STDERR "3.  Testing 'sendimage'...";
print "SEND IMAGE asterisk-image\n";
my $result = <STDIN>;
&checkresult($result);
```

This test calls the `SEND IMAGE` command, which is similar to the `SendImage( )` application. Its single argument is the name of an image file to send to the caller. As with the `SEND TEXT` command, this command works only if the calling channel supports the reception of images.

```
print STDERR "4.  Testing 'saynumber'...";
print "SAY NUMBER 192837465 \"\"\n";
my $result = <STDIN>;
&checkresult($result);
```

This test sends Asterisk the `SAY NUMBER` command. This command behaves identically to the `SayNumber( )` dialplan application. It takes two arguments:

- The number to say
- The digits that may interrupt the command

Again, since we're not passing in any digits as the second argument, we need to pass in an empty set of quotes.

```
print STDERR "5.  Testing 'waitdtmf'...";
print "WAIT FOR DIGIT 1000\n";
my $result = <STDIN>;
&checkresult($result);
```

This test shows the `WAIT FOR DIGIT` command. This command waits the specified number of milliseconds for the caller to enter a DTMF digit. If you want the command to wait indefinitely for a digit, use `-1` as the timeout. This application returns the decimal ASCII value of the digit that was pressed.

```
print STDERR "6.  Testing 'record'...";
print "RECORD FILE testagi gsm 1234 3000\n";
my $result = <STDIN>;
&checkresult($result);
```

This section of code shows us the `RECORD FILE` command. This command is used to record the call audio, similar to the `Record( )` dialplan application. `RECORD FILE` takes seven arguments, the last three of which are optional:

- The filename of the recorded file.
- The format in which to record the audio.
- The digits that may interrupt the recording.
- The timeout (maximum recording time) in milliseconds, or `-1` for no timeout.
- The number of samples to skip before starting the recording (optional).
- The word `BEEP`, if you'd like Asterisk to beep before the recording starts (optional).
- The number of seconds before Asterisk decides that the user is done with the recording and returns, even though the timeout hasn't been reached and no DTMF digits have been entered (optional). This argument must be preceded by `s=`.

In this particular case, we're recording a file called *testagi* (in the GSM format), with any of the DTMF digits 1 through 4 terminating the recording, and a maximum recording time of 3,000 milliseconds.

```
print STDERR "6a.  Testing 'record' playback...";
print "STREAM FILE testagi \"\"\n";
my $result = <STDIN>;
&checkresult($result);
```

The second part of this test plays back the audio that was recorded earlier, using the STREAM FILE command. We've already covered STREAM FILE, so this section of code needs no further explanation.

```
print STDERR "================= Complete ====================\n";
print STDERR "$tests tests completed, $pass passed, $fail failed\n";
print STDERR "================================================\n";
```

At the end of the AGI script, a summary of the tests is printed to STDERR, which should end up on the Asterisk console.

In summary, you should remember the following when writing AGI programs in Perl:

- Turn on strict language checking with the use strict command.*
- Turn off output buffering by setting $|=1.
- Data from Asterisk is received using a while(<STDIN>) loop.
- Write values with the print command.
- Use the print STDERR command to write debug information to the Asterisk console.

The Perl AGI Library

If you are interesting in building your own AGI scripts in Perl, you may want to check out the *Asterisk::AGI* Perl module written by James Golovich, which is located at *http://asterisk.gnuinter.net*. The *Asterisk::AGI* module makes it even easier to write AGI scripts in Perl.

Creating AGI Scripts in PHP

We promised we'd cover several languages, so let's go ahead and see what an AGI script in PHP looks like. The fundamentals of AGI programming still apply; only the programming language has changed. In this example, we'll write an AGI script to download a weather report from the Internet and deliver the temperature, wind direction, and wind speed back to the caller.

```
#!/usr/bin/php -q
<?php
```

The first line tells the system to use the PHP interpreter to run this script. The –q option turns off HTML error messages. You should ensure that there aren't any extra lines between the first line and the opening PHP tag, as they'll confuse Asterisk.

```
# change this to match the code of your particular city
# for a complete list of US cities, go to
# http://www.nws.noaa.gov/data/current_obs/
$weatherURL="http://www.nws.noaa.gov/data/current_obs/KMDQ.xml";
```

* This advice probably applies to any Perl program you might write, especially if you're new to Perl.

This tells our AGI script where to go to get the current weather conditions. In this example, we're getting the weather for Huntsville, Alabama. Feel free to visit the web site listed above for a complete list of stations throughout the United States of America.*

```
# don't let this script run for more than 60 seconds
set_time_limit(60);
```

Here, we tell PHP not to let this program run for more than 60 seconds. This is a safety net, which will end the script if for some reason it takes more than 60 seconds to run.

```
# turn off output buffering
ob_implicit_flush(false);
```

This command turns off output buffering, meaning that all data will be sent immediately to the AGI interface and will not be buffered.

```
# turn off error reporting, as it will most likely interfere with
# the AGI interface
error_reporting(0);
```

This command turns off all error reporting, as it can interfere with the AGI interface. (You might find it helpful to comment out this line during testing.)

```
# create file handles if needed
if (!defined('STDIN'))
{
   define('STDIN', fopen('php://stdin', 'r'));
}
if (!defined('STDOUT'))
{
   define('STDOUT', fopen('php://stdout', 'w'));
}
if (!defined('STDERR'))
{
   define('STDERR', fopen('php://stderr', 'w'));
}
```

This section of code ensures that we have open file handles for `STDIN`, `STDOUT`, and `STDERR`, which will handle all communication between Asterisk and our script.

```
# retrieve all AGI variables from Asterisk
while (!feof(STDIN))
{
    $temp = trim(fgets(STDIN,4096));
    if (($temp == "") || ($temp == "\n"))
    {
        break;
    }
```

* We apologize to our readers outside of the United States for using a weather service that only works for U.S. cities. If you can find a good international weather service that provides its data in XML, it shouldn't be too hard to change this AGI script to work with that particular service. Once we find one, we'll update this script for future editions.

```
    $s = split(":",$temp);
    $name = str_replace("agi_","",$s[0]);
    $agi[$name] = trim($s[1]);
}
```

Next, we'll read in all of the AGI variables passed to us by Asterisk. Using the `fgets` command in PHP to read the data from `STDIN`, we'll save each variable in the hash called `$agi`. Remember that we could use these variables in the logic of our AGI script, although we won't in this example.

```
# print all AGI variables for debugging purposes
foreach($agi as $key=>$value)
{
    fwrite(STDERR,"-- $key = $value\n");
    fflush(STDERR);
}
```

Here, we print the variables back out to `STDERR` for debugging purposes.

```
#retrieve this web page
$weatherPage=file_get_contents($weatherURL);
```

This line of code retrieves the XML file from the National Weather Service and puts the contents into the variable called `$weatherPage`. This variable will be used later on to extract out the pieces of the weather report that we want.

```
#grab temperature in Fahrenheit
if (preg_match("/<temp_f>([0-9]+)<\/temp_f>/i",$weatherPage,$matches))
{
    $currentTemp=$matches[1];
}
```

This section of code extracts the temperature (in Fahrenheit) from the weather report, using the `preg_match` command. This command uses Perl-compatible regular expressions* to extract out the needed data.

```
#grab wind direction
if (preg_match("/<wind_dir>North<\/wind_dir>/i",$weatherPage))
{
    $currentWindDirection='northerly';
}
elseif (preg_match("/<wind_dir>South<\/wind_dir>/i",$weatherPage))
{
    $currentWindDirection='southerly';
}
elseif (preg_match("/<wind_dir>East<\/wind_dir>/i",$weatherPage))
{
    $currentWindDirection='easterly';
}
elseif (preg_match("/<wind_dir>West<\/wind_dir>/i",$weatherPage))
{
    $currentWindDirection='westerly';
```

* The ultimate guide to regular expressions is O'Reilly's *Mastering Regular Expressions*, by Jeffrey Friedl.

```
}
elseif (preg_match("/<wind_dir>Northwest<\/wind_dir>/i",$weatherPage))
{
    $currentWindDirection='northwesterly';
}
elseif (preg_match("/<wind_dir>Northeast<\/wind_dir>/i",$weatherPage))
{
    $currentWindDirection='northeasterly';
}
elseif (preg_match("/<wind_dir>Southwest<\/wind_dir>/i",$weatherPage))
{
    $currentWindDirection='southwesterly';
}
elseif (preg_match("/<wind_dir>Southeast<\/wind_dir>/i",$weatherPage))
{
    $currentWindDirection='southeasterly';
}
```

The wind direction is found through the use of `preg_match` (located in the `wind_dir` tags) and is assigned to the variable `$currentWindDirection`.

```
#grab wind speed
if (preg_match("/<wind_mph>([0-9.]+)<\/wind_mph>/i",$weatherPage,$matches))
{
    $currentWindSpeed = $matches[1];
}
```

Finally, we'll grab the current wind speed and assign it to the `$currentWindSpeed` variable.

```
# tell the caller the current conditions
if ($currentTemp)
{
    fwrite(STDOUT,"STREAM FILE temperature \"\"\n");
    fflush(STDOUT);
    $result = trim(fgets(STDIN,4096));
    checkresult($result);
    fwrite(STDOUT,"STREAM FILE is \"\"\n");
    fflush(STDOUT);
    $result = trim(fgets(STDIN,4096));
    checkresult($result);
    fwrite(STDOUT,"SAY NUMBER $currentTemp \"\"\n");
    fflush(STDOUT);
    $result = trim(fgets(STDIN,4096));
    checkresult($result);
    fwrite(STDOUT,"STREAM FILE degrees \"\"\n");
    fflush(STDOUT);
    $result = trim(fgets(STDIN,4096));
    checkresult($result);
    fwrite(STDOUT,"STREAM FILE fahrenheit \"\"\n");
    fflush(STDOUT);
    $result = trim(fgets(STDIN,4096));
    checkresult($result);
}
```

```
if ($currentWindDirection && $currentWindSpeed)
{
    fwrite(STDOUT,"STREAM FILE with \"\"\n");
    fflush(STDOUT);
    $result = trim(fgets(STDIN,4096));
    checkresult($result);
    fwrite(STDOUT,"STREAM FILE $currentWindDirection \"\"\n");
    fflush(STDOUT);
    $result = trim(fgets(STDIN,4096));
    checkresult($result);
    fwrite(STDOUT,"STREAM FILE wx/winds \"\"\n");
    fflush(STDOUT);
    $result = trim(fgets(STDIN,4096));
    checkresult($result);
    fwrite(STDOUT,"STREAM FILE at \"\"\n";)
    fflush(STDOUT);
    $result = trim(fgets(STDIN,4096));
    checkresult($result);
    fwrite(STDOUT,"SAY NUMBER $currentWindSpeed \"\"\n");
    fflush(STDOUT);
    $result = trim(fgets(STDIN,4096));
    checkresult($result);
    fwrite($STDOUT,"STREAM FILE miles-per-hour \"\"\n");
    fflush(STDOUT);
    $result = trim(fgets(STDIN,4096));
    checkresult($result);
}
```

Now that we've collected our data, we can send AGI commands to Asterisk (checking the results as we go) that will deliver the current weather conditions to the caller. This will be achieved through the use of the `STREAM FILE` and `SAY NUMBER` AGI commands.

We've said it before, and we'll say it again: when calling AGI commands, you must pass in all of the required arguments. In this case, both `STREAM FILE` and `SAY NUMBER` commands require a second argument; we'll pass empty quotes escaped by the backslash character.

You should also notice that we call the `fflush` command each time we write to `STDOUT`. While this is arguably redundant, there's no harm in ensuring that the AGI command is not buffered and is sent immediately to Asterisk.

```
function checkresult($res)
{
    trim($res);
    if (preg_match('/^200/',$res))
    {
        if (! preg_match('/result=(-?\d+)/',$res,$matches))
        {
            fwrite(STDERR,"FAIL ($res)\n");
            fflush(STDERR);
            return 0;
        }
        else
```

```
        {
            fwrite(STDERR,"PASS (".$matches[1].")\n");
            fflush(STDERR);
            return $matches[1];
        }
    }
    else
    {
        fwrite(STDERR,"FAIL (unexpected result '$res')\n");
        fflush(STDERR);
        return -1;
    }
}
```

The `checkresult` function is identical in purpose to the `checkresult` subroutine we saw in our Perl example. As its name suggests, it checks the result that Asterisk returns whenever we call an AGI command.

```
?>
```

At the end of the file, we have our closing PHP tag. Don't place any whitespace after the closing PHP tag, as it can confuse the AGI interface.

We've now covered two different languages, in order to demonstrate the similarities and differences of programming an AGI script in PHP as opposed to Perl. The following things should be remembered when writing an AGI script in PHP:

- Invoke PHP with the `–q` switch; it turns off HTML in error messages.
- Turn off the time limit, or set it to a reasonable value (newer versions of PHP automatically disable the time limit when PHP is invoked from the command line).
- Turn off output buffering with the `ob_implicit_flush(false)` command.
- Open file handles to `STDIN`, `STDOUT`, and `STDERR` (newer versions of PHP may have one or more of these file handles already opened—see the code above for a slick way of making this work across most versions of PHP).
- Read variables from `STDIN` using the `fgets` function.
- Use the `fwrite` function to write to `STDOUT` and `STDERR`.
- Always call the `fflush` function after writing to either `STDOUT` or `STDERR`.

The PHP AGI Library

For advanced AGI programming in PHP, you may want to check out the PHPAGI project at *http://phpagi.sourceforge.net*. It was originally written by Matthew Asham and is being developed by several other members of the Asterisk community.

Writing AGI Scripts in Python

The AGI script we'll be writing in Python, called "The Subtraction Game," was inspired by a Perl program written by Ed Guy and discussed by him at the 2004 AstriCon conference. Ed described his enthusiasm for the power and simplicity of Asterisk when he found he could write a quick Perl script to help his young daughter improve her math skills.

Since we've already written a Perl program using AGI, and Ed has already written the math program in Perl, we figured we'd take a stab at it in Python!

Let's go through our Python script.

```
#!/usr/bin/python
```

This line tells the system to run this script in the Python interpreter. For small scripts, you may consider adding the –u option to this line, which will run Python in unbuffered mode. This is not recommended, however, for larger or frequently used AGI scripts, as it can affect system performance.

```
import sys
import re
import time
import random
```

Here, we import several libraries that we'll be using in our AGI script.

```
# Read and ignore AGI environment (read until blank line)

env = {}
tests = 0;

while 1:
   line = sys.stdin.readline( ).strip( )

   if line == '':
      break
   key,data = line.split(':')
   if key[:4] <> 'agi_':
      #skip input that doesn't begin with agi_
      sys.stderr.write("Did not work!\n");
      sys.stderr.flush( )
      continue
   key = key.strip( )
   data = data.strip( )
   if key <> '':
      env[key] = data

sys.stderr.write("AGI Environment Dump:\n");
sys.stderr.flush( )
for key in env.keys( ):
   sys.stderr.write(" -- %s = %s\n" % (key, env[key]))
   sys.stderr.flush( )
```

This section of code reads in the variables that are passed to our script from Asterisk, and saves them into a dictionary named env. These values are then written to STDERR for debugging purposes.

```
def checkresult (params):
   params = params.rstrip( )
   if re.search('^200',params):
      result = re.search('result=(\d+)',params)
      if (not result):
         sys.stderr.write("FAIL ('%s')\n" % params)
         sys.stderr.flush( )
         return -1
      else:
         result = result.group(1)
         #debug("Result:%s Params:%s" % (result, params))
         sys.stderr.write("PASS (%s)\n" % result)
         sys.stderr.flush( )
         return result
   else:
      sys.stderr.write("FAIL (unexpected result '%s')\n" % params)
      sys.stderr.flush( )
      return -2
```

The checkresult function is almost identical in purpose to the checkresult subroutine in the sample Perl AGI script we covered earlier in the chapter. It reads in the result of an Asterisk command, parses the answer, and reports whether or not the command was successful.

```
def sayit (params):
   sys.stderr.write("STREAM FILE %s \"\"\n" % str(params))
   sys.stderr.flush( )
   sys.stdout.write("STREAM FILE %s \"\"\n" % str(params))
   sys.stdout.flush( )
   result = sys.stdin.readline( ).strip( )
   checkresult(result)
```

The sayit function is a simple wrapper around the STREAM FILE command.

```
def saynumber (params):
   sys.stderr.write("SAY NUMBER %s \"\"\n" % params)
   sys.stderr.flush( )
   sys.stdout.write("SAY NUMBER %s \"\"\n" % params)
   sys.stdout.flush( )
   result = sys.stdin.readline( ).strip( )
   checkresult(result)
```

The saynumber function is a simple wrapper around the SAY NUMBER command.

```
def getnumber (prompt, timelimit, digcount):
   sys.stderr.write("GET DATA %s %d %d\n" % (prompt, timelimit, digcount))
   sys.stderr.flush( )
   sys.stdout.write("GET DATA %s %d %d\n" % (prompt, timelimit, digcount))
   sys.stdout.flush( )
   result = sys.stdin.readline( ).strip( )
   result = checkresult(result)
```

```
    sys.stderr.write("digits are %s\n" % result)
    sys.stderr.flush( )
    if result:
       return result
    else:
       result = -1
```

The getnumber function calls the GET DATA command to get DTMF input from the caller. It is used in our program to get the caller's answers to the subtraction problems.

```
limit=20
digitcount=2
score=0
count=0
ttanswer=5000
```

Here, we initialize a few variables that we'll be using in our program.

```
starttime = time.time( )
t = time.time( ) - starttime
```

In these lines we set the starttime variable to the current time and initialize t to zero. We'll use the t variable to keep track of the number of seconds that have elapsed since the AGI script was started.

```
sayit("subtraction-game-welcome")
```

Next, we welcome the caller to the subtraction game.

```
while ( t < 180 ):

   big = random.randint(0,limit+1)
   big += 10
   subt= random.randint(0,big)
   ans = big - subt
   count += 1

   #give problem:
   sayit("subtraction-game-next");
   saynumber(big);
   sayit("minus");
   saynumber(subt);
   res = getnumber("equals",ttanswer,digitcount);

   if (int(res) == ans) :
      score+=1
      sayit("subtraction-game-good");
   else :
      sayit("subtraction-game-wrong");
      saynumber(ans);

   t = time.time( ) - starttime
```

This is the heart of the AGI script. We loop through this section of code and give subtraction problems to the caller until 180 seconds have elapsed. Near the top of

the loop, we calculate two random numbers and their difference. We then present the problem to the caller, and read in the caller's response. If the caller answers incorrectly, we give the correct answer.

```
pct = float(score)/float(count)*100;
sys.stderr.write("Percentage correct is %d\n" % pct)
sys.stderr.flush( )

sayit("subtraction-game-timesup")
saynumber(score)
sayit("subtraction-game-right")
saynumber(count)
sayit("subtraction-game-pct")
saynumber(pct)
```

After the users are done answering the subtraction problems, they are given their scores.

As you have seen, the basics you should remember when writing AGI scripts in Python are:

- Flush the output buffer after every write. This will ensure that your AGI program won't hang while Asterisk is waiting for the buffer to fill and Python is waiting for the response from Asterisk.
- Read data from Asterisk with the `sys.stdin.readline` command.
- Write commands to Asterisk with the `sys.stdout.write` command. Don't forget to call `sys.stdout.flush` after writing.

The Python AGI Library

If you are planning on writing lot of Python AGI code, you may want to check out Karl Putland's Python module, *Pyst*. You can find it at *http://www.sourceforge.net/projects/pyst/*.

Debugging in AGI

Debugging AGI programs, as with any other type of program, can be frustrating. Luckily, there are two advantages to debugging AGI scripts. First, since all the communications between Asterisk and the AGI program happen over `STDIN` and `STDOUT` (and, of course, `STDERR`), you should be able to run your AGI script directly from the operating system. Second, Asterisk has a handy command for showing all the communications between itself and the AGI script: `agi debug`.

Debugging from the Operating System

As mentioned above, you should be able to run your program directly from the operating system to see how it behaves. The secret here is to act just like Asterisk does, providing your script with the following:

- A list of variables and their values, such as `agi_test:1`.
- A blank line feed (`/n`) to indicate that you're done passing variables.
- Responses to each of the AGI commands from your AGI script. Usually, typing **`200 response=1`** is sufficient.

Trying your program directly from the operating system may help you to more easily spot bugs in your program.

Using Asterisk's agi debug Command

The Asterisk command-line interface has a very useful command for debugging AGI scripts, which is called (appropriately enough) `agi debug`. If you type **`agi debug`** at an Asterisk console and then run an AGI, you'll see something like the following:

```
    -- Executing AGI("Zap/1-1", "temperature.php") in new stack
    -- Launched AGI Script /var/lib/asterisk/agi-bin/temperature.php
AGI Tx >> agi_request: temperature.php
AGI Tx >> agi_channel: Zap/1-1
AGI Tx >> agi_language: en
AGI Tx >> agi_type: Zap
AGI Tx >> agi_uniqueid: 1116732890.8
AGI Tx >> agi_callerid: 101
AGI Tx >> agi_calleridname: Tom Jones
AGI Tx >> agi_callingpres: 0
AGI Tx >> agi_callingani2: 0
AGI Tx >> agi_callington: 0
AGI Tx >> agi_callingtns: 0
AGI Tx >> agi_dnid: unknown
AGI Tx >> agi_rdnis: unknown
AGI Tx >> agi_context: incoming
AGI Tx >> agi_extension: 141
AGI Tx >> agi_priority: 2
AGI Tx >> agi_enhanced: 0.0
AGI Tx >> agi_accountcode:
AGI Tx >>
AGI Rx << STREAM FILE temperature ""
AGI Tx >> 200 result=0 endpos=6400
AGI Rx << STREAM FILE is ""
AGI Tx >> 200 result=0 endpos=5440
AGI Rx << SAY NUMBER 67 ""
    -- Playing 'digits/60' (language 'en')
    -- Playing 'digits/7' (language 'en')
AGI Tx >> 200 result=0
AGI Rx << STREAM FILE degrees ""
```

```
AGI Tx >> 200 result=0 endpos=6720
AGI Rx << STREAM FILE fahrenheit ""
AGI Tx >> 200 result=0 endpos=8000
    -- AGI Script temperature.php completed, returning 0
```

You'll see three types of lines while your AGI script is running. The first type, prefaced with `AGI TX >>`, are the lines that Asterisk transmits to your program's `STDIN`. The second type, prefaced with `AGI RX <<`, are the commands your AGI program writes back to Asterisk over `STDOUT`. The third type, prefaced by `--`, are the standard Asterisk messages presented as it executes certain commands.

To disable AGI debugging after it has been started, simply type **`agi no debug`** at an Asterisk console.

Using the `agi debug` command will enable you to see the communication between Asterisk and your program, which can be very useful when debugging. Hopefully, these two tips will greatly improve your ability to write and debug powerful AGI programs.

Conclusion

For a developer, AGI is one of the more revolutionary and compelling reasons to choose Asterisk over a closed, proprietary PBX. Still, AGI is only part of the picture. For those of us who are less developers and more systems integrators or power users, Chapter 10 will explore the wealth of accoutrements that make Asterisk compelling to so many people.