1. Introduction

Jammer is an interactive script language interpreter designed to enable testing of the prototype ATM switches built at Washington University. Jammer was originally created for testing of the first Washington University prototype ATM switch [1,2] and has been updated for testing of the new Gigabit ATM switch [3]. To provide the test engineer with as much flexibility and testing power as possible, Jammer enables full access to all tables and registers within the switch. It also provides basic programming constructs to allow iterative and conditional testing.

2. Context

To understand how Jammer works, it is important to understand the communication structure shown in Figure 2.1 below.

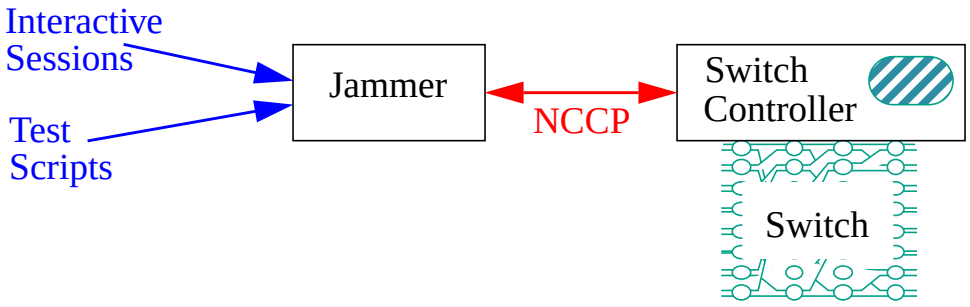

Figure 2.1: *Jammer, Switch Controller, and Switch Communication*

As can be seen, Jammer provides the test engineer with a direct interface to the switch controller via the Node Controller Communications Protocol (NCCP) [7]. Direct access to all registers and tables in the switch is thus provided to the user for both interactive and batch testing.

3. Language Overview

Jammer provides a number of constructs which enable flexible and thorough testing of the Gigabit switch. These constructs can be divided into programming constructs, session constructs, and test constructs. Programming constructs enable iterative and conditional control of the test session. Session constructs provide control of session-related variables such as current host, node, and address. Test constructs allow direct access to the internal hardware tables and registers of the switch for testing and manipulation of the switch. Each of the three construct groups will be described in detail in the following sections.

The Jammer syntax is a loose adaptation of the C and shell language syntaxes. To provide a certain degree of simplicity, some of the C syntax was modified and certain additions were made. However, programmers familiar with C or shell programming should have little difficulty adapting to writing Jammer scripts. For experienced programmers, the most challenging part of writing Jammer scripts will be learning the low level switch details. To study the Jammer syntax in detail, see the source code (jammer.y).

Before explaining the different constructs, it is necessary to explain some of the notation which will be used throughout this document. When describing Jammer commands, the courier font will

be used to show the command. Any variables or user-selectable values will be denoted by italics. For example, to make Jammer display verbose output, the following command would be used:

```
set print VERBOSE
```

In this case, `set print` is the primary part of the Jammer command. For the `set print` *volume* command, the user could have input `VERBOSE`, `NORMAL`, `QUIET_ACK`, or `QUIET` as the volume.

When there are a limited number of options for a command setting, each option will be described in the section which documents the use of the corresponding construct. In the case that a numerical range is involved, the range may be described in either the switch specification [3], the language documentation (Section 3), or both. Because these ranges may be difficult to find in the switch specification, every attempt will be made to make sure that all relevant ranges and limitations be explained in this document.

3.1. Program Constructs

3.1.1 A Basic Script

The basic Jammer procedure consists of a single procedure of arbitrary name. For example, a script which prints to the screen takes the form:

```
#this sample prints the Hello! message to screen
proc  PrintToScreen()
     echo “Hello!”
end
```

To run this script, using a text editor, create a file named `print_to_screen.js`. (The file extension `.js` is typically used for Jammer scripts.) Next, invoke Jammer. At the Jammer prompt, type:

```
include print_to_screen.js
```

to import the *PrintToScreen()* procedure into the current Jammer session. At the next Jammer prompt, invoke the procedure by typing:

```
PrintToScreen()
```

This will cause Jammer to execute the *PrintToScreen()* procedure. More complicated and sophisticated procedures can be written, but each must follow the basic format of the one above. That is, each must begin with the word `proc` followed by the procedure name followed by a parenthesized list of parameters. (The example above does not have any parameters.) The body of the procedure follows and is concluded by `end`. Local variables are typically declared in a group on the first lines of the procedure. The functional part of the procedure follows the variable declarations. Still more complicated scripts can be created by linking a number of Jammer procedures using the `include` command at the head of the script file.

Once Jammer has executed the procedure of a script file, the Jammer prompt will be redisplayed and interactive control will be returned to the user.

3.1.2 Comments

Comments in Jammer are like the comments of the Unix Shell language, they begin with the # character and are one line each. Any characters from the # character to the end of the line are considered part of the comment text.

3.1.3 Variable Declarations

Before writing more useful Jammer scripts, it is necessary to understand how to define and use variables. To fully test the switch, only integer values are needed, so Jammer only supports integer variables, in scalar and vector form. A scalar integer variable is defined as:

```
int variable
```

Similarly, the vector form of this declaration is:

```
int var_array[size]
```

In this case, *size* indicates the number of elements in the integer array *var_array* and ranges from 0 to *size*-1. This value must be indicated before the script can be run. Dynamic memory allocation is not currently supported in Jammer.

Depending on the scope desired, variables can be defined locally or globally. Local variables are defined at the top of each procedure while global variables are defined outside of the procedure. Additionally, variables can be passed between various procedures as parameters.

3.1.4 Variable Evaluation

In order to fully utilize the variables once they have been defined, it is necessary that they be evaluated. In a Jammer script, to indicate that a variable should be evaluated, a dollar sign is prepended to the variable name. For example, to evaluate the number represented by the variable node, the programmer types:

```
$node
```

Evaluation of array variables can be performed in similar fashion:

```
$node_array[2]
```

This command causes Jammer to evaluate the value of the second element (0th, 1st, **2nd**,...) of the node array. Note that if the index is out of range, Jammer will return a value of -1 and output an error message.

3.1.5 Assignment Operators

In Jammer, the assignment operator is a single equal sign. Assigning the number 10 to the variable *node* is accomplished by typing:

```
node = 10
```

Setting the value of a variable to that of another variable is done like this:

```
node = $variable
```

3.1.6 Mathematical Operations

During testing, a number of mathematical operations on numbers and variables are necessary. To allow full modification and manipulation, Jammer offers the four basic mathematical operations:

```
+     - addition
-     - subtraction
*     - multiplication
/     - division
```

Additionally, Jammer also provides postincrement and postdecrement (but not preincrement or predecrement) operators familiar to C programmers:

```
++    - postincrement
--    - postdecrement
```

The use of the basic operations is shown in the following example which increments the value of the variable node by 10:

```
node = $node + 10
```

To decrement the value by one:

```
node--
```

3.1.7 Negation Operator

Jammer also supports logical negation of a variable in a manner similar to that of C. This is achieved by prepending an exclamation point to a variable name. For example, to negate the value of the variable *result*, type:

```
!($result)
```

3.1.8 Comparison Operators

Like C, Jammer uses the double equals '==' to test for equality between two values. The set of comparison operators supported by Jammer is:

```
==    - equal to
!=    - not equal to
>     - greater than
<     - less than
>=    - greater than or equal to
<=    - less than or equal to
```

The usage of the comparison operators is similar to that of the assignment operator except that both the left and the right side variables must be evaluated. That means that to compare the value of the *node* variable, the left side must be evaluated:

```
$node <= 10
```

and:

```
$node <= $variable
```

Used in conjunction with the conditional constructs, the comparison operators enable conditional

control of Jammer scripts.

3.1.9 Conditional Constructs

The basic conditional control structure provided by Jammer is the `if-else` statement. The Jammer syntax for this structure differs from the C syntax in that an `if-else` statement must be terminated by the word ‘fi’. A number of if-else statements may be nested together as:

```
if( conditional )
    operations
else
    if( conditional )
        operations
    else
        operations
    fi
fi
```

or the else part may be omitted:

```
if( conditional )
    operations
fi
```

This construct provides a great deal of flexibility in testing one or many conditions and performing different operations based on the result of the tests. In syntax, it differs little from the `if-else` statements provided by other programming languages.

3.1.10 Iterative Constructs

A second structure provided for program control is the `while-done` loop. Again, the syntax for this construct differs from C syntax only in the termination string and its lack of curly braces. In use, it appears as:

```
while( conditional )
    operations
done
```

Since the `while` loop can be generalized to perform the same function as a C `for` loop, Jammer does not provide a separate `for` construct. To enable further control of the `while` loop, Jammer also provides a `break` command analogous to the C command of the same name. For example, the `break` command can be used to exit a `while` loop upon satisfaction of a certain condition:

```
while( conditional )
    if( conditional )
        operations
        break
    else
        operations
    fi
    operations
done
```

In providing conditional and iterative constructs, Jammer allows the programmer to exhaustively

test the switch and to devise tests which would not be possible otherwise.

3.1.11 Creating Large Scripts

At the beginning of this section a basic script was presented to illustrate some of the elementary characteristics of Jammer scripts. In order to support dynamic switch testing, Jammer was designed to allow the test engineer great latitude in designing any number of tests. To facilitate this, Jammer procedures may be linked to provide a hierarchical test facility for the Gigabit Switch. The mechanism which enables this capability is the `include` command. This command is analogous to the C `#include` directive, but it might be better compared to the `import` command in Java. The syntax for the include statement is:

`include` *filename*

The function of this statement is to cause Jammer to interpret multiple script files as a single test script. This allows the user to create modular testing procedures which can then be utilized in a number of different situations. In this way, the script writer is not bound to immense files of procedures and is freed to configure the test suite in any fashion.

3.2. Session Constructs

The session constructs provide access to session-related variables.

3.2.1 set switch *switch_id host port*

Jammer can communicate with multiple switches during the same session. To change which switch is being manipulated, the `set switch` command is used. The first time a switch is contacted it is necessary to tell Jammer on what host it is running and on what TCP port it is listening for connections. After this information has been provided once, it does not need to be entered again. Just the switch_id is sufficient. For example:

```
set switch 1.3 wugs1 5345
echo “on switch 1.3 now”
set switch 1.2 wugs2 5340
echo “on switch 1.2 now”
set switch 1.3
echo “back to 1.3”
```

3.2.2 ! *system_command*

At times, it is desirable to execute a Unix system command without exiting from Jammer. For example, in the instance that last minute changes are needed in a script file, the user can edit the script file without exiting from Jammer or opening a new window by invoking a text editor using the Jammer system command.

As an example of the usage of this construct, it is frequently useful to be able to do a file listing to find all script files which are available for inclusion in the current test session. To do this, one would type:

```
! ls
```

Optionally, to list all script files saved with the extension `.js`, this command would take the form:

```
! "ls *.js"
```

When the system command requires more than one token, it is necessary to include the command in quotations.

3.2.3 `cd` *path*

Jammer provides the `cd` command to allow the user to change the current working directory from within Jammer. Its *path* argument conforms to the Unix path syntax.

3.2.4 CTRL-C

In Jammer, the signal sent in response to typing CTRL-C is caught and handled by resetting the Jammer process. In this way, Jammer forces the user to exit from Jammer using the `quit` command. This ensures a clean exit which will release all communication structures that were allocated when Jammer was invoked. It also gives the user a mechanism for interrupting a running script without actually killing the Jammer session.

3.2.5 `date`

This command prints the current time and date to the screen. This command enables timestamping operations during testing which is useful when compiling test logs.

3.2.6 `echo`

Jammer provides limited IO facilities. Like the Unix command of the same name, the `echo` command allows the script writer to output desired messages to the screen in the middle of a script run. The message output is not limited strictly to static status messages, since it can also be used to output the current value of a program variable. For instance, in testing the switch, if an error occurs in a loop test of VCXT tables, the tester will want to know which VCXT experienced the failure. The easiest way to do this is to include an error test within the test loop. If a failure occurs, it can be reported using the `echo` command as shown below:

```
if( failed )
    echo "Failure in testing VCXT number $vcxtIndex"
fi
```

In the above code fragment, `failed` is a Jammer session variable which can be checked to see if an error occurred and `vcxtIndex` is a user defined loop counter used to iterate through the different VCXT table entries.

3.2.7 `help`

Typing help from the Jammer prompt will cause Jammer to output a listing of all Jammer commands. This is a rather terse listing and provides minimal information. For more detailed information, consult the appropriate sections of this manual, the Jammer script documentation and the switch specification.

3.2.8 `pause` and `upause`

To provide temporal script control, Jammer provides the `pause` and `upause` commands. The `pause` invocation takes the form `pause` *value* where *value* is an integer value indicating the de-

sired duration of the command in seconds. Once invoked, Jammer will idle for the requested amount of time before continuing with script processing. The `upause` command takes the same form and provides the same function, except that it causes Jammer to idle for *value* microseconds, not seconds.

3.2.9 `ping` *arg*

To check the status of the switch controller or the switch module, the `ping` command is used. To do this, the command is invoked as shown below:

`ping` *entity*

where *entity* takes one of two arguments, either `SC` or `SM` depending on whether one wishes to check the status of the switch controller or the switch module. The difference between these two is that when the switch controller is the target of the `ping`, the `ping` message goes to the switch controller and immediately back to Jammer. When the switch module is the target, the switch controller sends a test cell through the switch and returns the result of the test cell.

This is quite often the first command entered when Jammer is started.

3.2.10 `prompt` *"string"*

Just as `echo` provides the output functions of Jammer, `prompt` provides the input functions. This command can be used to cause Jammer to request and retrieve user input. Currently, user input is restricted to integer requests. For instance, in a script which allows the user to indicate which maintenance register will be tested will contain a line similar to:

```
mr = prompt "Please indicate which MR should be tested next: "
```

This will display the message within quotes and wait until the user has typed in the desired value. The script will then continue processing. Note that if the user responds by typing a non-integer string, Jammer will not explicitly handle the error, so the script programmer will have to provide any desired error checking.

A prompt command of the form:

```
prompt "Hit return when you are ready to continued..."
```

takes no integer value but merely waits for the user to hit a <return>. This can be useful when running a test that requires ports to be connected in a certain way. In this case, a message can be printed indicating exactly how things should be connected and then Jammer waits for the user to indicate that all is ready.

3.2.11 `quit`

This command causes Jammer to shut down and close the appropriate communication links. Note that Jammer will handle CTRL-C by resetting and returning to the Jammer prompt. In this way it forces the user to use the `quit` command to exit from Jammer.

3.2.12 `reset` *entity*

Occasionally it is necessary to reinitialize and clear the state of the switch or switch controller to

which Jammer is attached. As with the `ping` command, *entity* can take the form of `SC` or `SM` depending on whether the switch controller or switch module should be reset. This command is used most frequently during interactive testing. It should be noted that hitting CTRL-C while running Jammer resets only Jammer itself.

`reset SC` currently does nothing to the switch controller, it just returns a positive response.

3.2.13 `shutdown` *switch_address*

During testing, the user may choose to disable a switch by issuing a `shutdown` command. The desired switch is indicated by the *switch_address* variable. This variable takes the form of two numbers separated by a period, like *1.1* or *0.0*. This command causes the switch controller to be shut down, thus disabling that access to the corresponding switch.

3.2.14 `set print` *volume*

The `set print` *volume* command allows the user to select the amount of feedback printed by Jammer during the course of a test session. The four *volume* options are:

- `QUIET` - display only echo information
- `QUIET_ACK` - display failure messages
- `NORMAL` - display acknowledgments also
- `VERBOSE` - display everything possible

During the course of any test session, the print volume may be changed at any time according to the needs of the user. It is advisable to reset the print volume to NORMAL at any point where a Jammer script might terminate so that when Jammer returns to interactive mode it will display enough information to communicate with the user.

Also, it should be noted that the result of different Test commands will not provide the same amount of output. For example, issuing a `read` command with the print volume set to `NORMAL` will produce more output than issuing the corresponding `write` command. Because of the nature of these commands, the `read` command will cause Jammer to output a list of the current values of the table being read while the `write` command will cause Jammer to output the success or failure of the attempted write.

3.2.15 `status`

When invoked, this command causes Jammer to output the current status of the session variables SWITCH, FAILED and PRINT.

3.2.16 `wait` and `waiting`

The `wait` and `waiting` commands are closely related in functionality. Like `pause`, the `wait` command provides temporal control of script execution, but unlike `pause`, `wait` does not involve any time specifications. When invoked, `wait` causes Jammer to cease processing new commands until all outstanding commands have succeeded, timed out, or failed. This functionality can be approximated using `waiting` and `pause` together as shown below.

```
while( waiting )
    pause 1
done
```

While `wait` causes Jammer to pause until all outstanding commands have completed, `waiting`

checks for the existence of outstanding commands.

The importance of the wait command may not be immediately apparent, but it is very helpful in many test situations where large groups of tables or registers are tested. If a loop test iterates through all possible VPXTs and writes to every VPXT in the table, Jammer will not wait for the result of a `write` before writing to the next VPXT unless specifically instructed to do so by the `wait` command.

Even though `wait` makes it possible to retrieve a response after each operation, it is typically inefficient to do so. Instead, it is usually more efficient to go through the entire test, then issue the `wait` command, and finally to test for success of the entire test. The drawback to this method is that in case of a failure or timeout, it is difficult to verify which of the individual tests caused the failure without somehow iterating through the loop again, checking explicitly for failure at each test. However, experience has shown that this method is more efficient than checking for failures the first time through.

3.2.17 `clear failed`

Jammer keeps a session variable which indicates whether a command failure has occurred. During a test session, it is useful to reset the value of this variable when entering a new phase of testing. For this reason, the `clear failed` command is provided to allow the user to set the variable back to the non-failure state.

3.3. Test Constructs

3.3.1 `clear`

The function of this command is to allow the test engineer to clear (i.e. set to zero) the entries of the tables and registers of the switch. For detailed information on the tables and registers controlled with this command, refer to Chapter 7 of the System Architecture Document [3].

At invocation, the `clear` command takes either two or three arguments where the first argument is one of the following four selection variables:

- `mr` - maintenance register
- `vcxt` - virtual circuit table
- `vpxt` - virtual path table
- `vxt` - both virtual circuit and virtual path tables

When clearing the maintenance registers of the switch, this command takes the form:

`clear mr` *[switch] port_proc [field]*

The *switch* variable is an optional variable which can be used to specify which switch's registers will be cleared. This variable takes the form *x.x* where *x* is an integer. This address is not an IP address but is an address assigned to the switch for the benefit of switch testing.

The *port_proc* variable indicates which of the eight port processors contains the maintenance registers to be cleared. It will take a value ranging from 0 to 7 or the string `all` where `all` indicates that every field of each port processor will be cleared.

If *port_proc* is not `all`, then the *field* variable must also be included. The field variable indicates which of the twenty-one maintenance register fields will be cleared. It can take a value from 1 to

22 or the string `all`. Similar to the *port_proc* variable, when the *field* variable is set to `all` each field of the indicated port processor will be cleared.

Great care should be taken when clearing or even setting the maintenance register fields. Setting some fields to inappropriate values can make the switch inaccessible. A hardware reset might be necessary to regain access to the switch.

The form of the command will be similar when clearing the virtual circuit and virtual path tables:

`clear vcxt` *[switch] port_proc vci*
`clear vpxt` *[switch] port_proc vpi*

Once again, the *switch* variable is an optional variable which can be used to specify which switch's tables will be cleared. This variable takes the form *x.x* where *x* is an integer. This address is not an IP address but is an address assigned to the switch for the benefit of switch testing.

As before, the *port_proc* variable represents the port processor whose tables will be cleared. However, when clearing the tables, the port processor must be chosen singly, so the only valid values are integers from 0 to 7. Therefore, the third argument, *vci* or *vpi* depending on whether virtual circuit or virtual path tables are the target, is mandatory. The third argument can take values from 0 to 1023, or the string `all`.

The fourth selection variable option is `vxt` which effectively combines clearing of virtual path and virtual circuit tables. This option takes the form:

`clear vxt` *[switch] port_proc* `[all]`

Here, the variable *port_proc* can be set to values from 0 to 7 or the string `all`. If *port_proc* is not all, then the command requires a final argument whose only valid value is the string `all`.

3.3.2 `read`

The read command allows the user to read entries from the registers and tables of the switch's port processors. The full syntax of this command is:

`read` *sel_var [switch] port_proc field*

Where the variable *sel_var* is one of the following selection variables:

- `mr` - maintenance register
- `vcxt` - virtual circuit table
- `vpxt` - virtual path table

As in the `clear` command, the first variable is used to indicate whether the maintenance register, the virtual circuit table, or the virtual path table will be selected.

As before, the *switch* variable is an optional variable which can be used to specify which switch's registers or tables will be read. This variable takes the form *x.x* where *x* is an integer. This address is not an IP address but is an address assigned to the switch for the benefit of switch testing.

The *port_proc* variable indicates which of the switch's port processors will be tested. Again, this

value can be any integer from 0 to 7.

The final argument is the *field* variable. When *sel_var* is `mr`, this variable indicates which of the maintenance register's fields will be read and can be any number from 1 to 22. When *sel_var* is `vcxt` or `vpxt`, the field variable ranges from 0 to 1023 and represents the virtual circuit or virtual path table entry to be read.

For virtual circuit and virtual path table reads, the output of this command will take the form:

```
Jammerlist: Read VCXT Operation Completed Successfully
    bi = 0
    rc = 0
    d = 0
    cyc1 = 0
    cyc2 = 0
    cs = 0
    ud1 = 0
    ud2 = 0
    sc = 0
    vpt = 0
    rco = 0
    mapt1vpi = 0
    mapt1vci = 0
    bdi1 = 0
    mapt2vpi = 0
    mapt2vci = 0
    bdi2 = 0
    adr1 = 0
    adr2 = 0
```

Of course, the values for each of the table entries will depend on the actual contents at the time of the read operation. See Section 7.1 of the System Architecture Document [3] for more detailed information on the table entries.

For maintenance register reads, the fields output by the read command will depend on which of the twenty-two fields was read. Each has a different number of entries. For information on the nature of these entries, refer to Section 7.2 of the System Architecture Document [3].

3.3.3 `test`

In order to enable testing of the contents of switch registers and tables, Jammer provides the `test` command. This command allows the user to specify the contents of each of the register or table entries. These values are then compared with the current values stored in the switch. If all of the values are the same, the command will succeed. If one or more of the values differ, it will fail.

This command can be somewhat difficult at first since it takes a variable number of arguments for maintenance register testing and a different number of arguments for virtual circuit and virtual path table testing. For maintenance register testing, the number of arguments required depends on which of the maintenance register field being tested.

All tests begin with the same core arguments regardless of the table or register to be tested. This core takes the same form as the `read` command syntax:

`test` *sel_var [switch] port_proc field*

Each of the arguments in the core of the `test` command has the same limitations as the corresponding argument in the `read` command (Section 3.3.2). In the `test` command, this core is then followed by a string of integer values whose length is dependent on the value of the variables *sel_var* and *field*.

When testing the contents of the maintenance register, it can be difficult to remember which fields are correlated with which maintenance registers. Section 7.2 of the System Architecture Document [3] documents the various fields. During interactive testing, Jammer will prompt the user for the appropriate fields. When testing the virtual circuit and virtual path tables, the field values are those described in Section 3.3.2 above.

3.3.4 `write`

The final table manipulation command is the `write` command. The `write` command is identical in syntax and format to the `test` command described in Section 3.3.3. Using this command, the user can selectively manipulate any bit in the switch tables and registers by choosing the appropriate field values for each field of the table or register being written. When invoked, this command will succeed if the indicated values can be successfully written into the desired tables or registers. If the values cannot be written into the switch, the command will fail.

3.3.5 `get cells`

The `get cells` command allows the user to retrieve and test data cells from the switch. These cells are typically created using the put cells command and allow the user to fully define the data sent through the switch. By allowing the user to specify the data cells in a bitwise fashion, Jammer provides the user with a powerful tool for checking for the existence of various errors which are detectable only by careful analysis of the data in the switch.

The syntax for this command is:

`get cells` *vpi vci cell_count timeout repeat_count offset arg_list*

The *vpi* and *vci* fields indicate from which VPI/VCI pair the cells will be retrieved. Using the *cell_count* field, it is possible to specify an arbitrary number of cells to be retrieved at any time. The amount of time to wait before failing is specified in seconds using the *timeout* field. Any time greater than one second may be specified. However, the default maximum timeout value (approximately two years) can be selected by setting the *timeout* field to zero.

Additionally, the user may designate any bit pattern for the cells using the remaining fields. The *repeat_count* field indicates how many times the pattern is repeated in the cell while the *offset* field indicates the number of bytes in the cell before the start of the pattern. Finally, the pattern is indicated using the comma separated argument list of the final field of this command.

This command is typically issued prior to invoking the accompanying `put cells` command, otherwise the cells placed in the switch may have already passed through the switch before the `get cells` command can capture them. Because of this, it is necessary to specify a timeout value large enough that the `get cells` command will not expire before the `put cells` command can be

invoked.

For a detailed example, see Section 5.2.1 later in this document.

3.3.6 `put cells`

The `put cells` command allows the user to create data cells and inject them into the switch. Used in conjunction with the `get cells` command, this allows the user to test the ability of the switch to correctly transmit data. By allowing the user to specify the data cells in a bitwise fashion, Jammer provides the user with a powerful tool for checking for the existence of various errors which are detectable only by careful analysis of the data in the switch.

This command has a syntax which is approximately the same as the `get cells` command in the previous section. It is:

`put cells` *vpi vci cell_count offset repeat arg_list*

The only difference between the syntax of the two commands is the lack of the *timeout* field in this command. Because the cells are placed immediately in the switch when this command is invoked, a user-specified timeout is not necessary. For further detail on the other fields, refer to the preceding section.

3.3.7 `build`

The `build` command is the first of the Jammer commands used to build multipoint connections.These commands are intended to make it easier to build complex general multipoint to multipoint connections. (It is still unclear how much easier these commands make it but experience will tell more.)

This command is used to indicate the number of transmitters and receivers which will make up the multipoint tree, as illustrated in detail in Section 5.2.2. This command takes four arguments and is followed by a combination of `merge`, `xmit`, `recycle`, and `receive` commands. Its syntax is:

`build` *vxt* `mpt` *num_tx num_rx cs*

The first argument can take one of two values, either `VCXT` or `VPXT`, depending on whether the connection will be a virtual circuit or virtual path multipoint connection. The next two fields take integer arguments and specify respectively the number of transmitters in the connection and the number of receivers in the connection. The final argument indicates whether the connection is to be a continuous stream connection. As values, the *cs* field accepts values of one if it is to be a continuous stream connection and zero if it is not.

For further information on multipoint connections, refer to Section 3.2 of the System Architecture Document [3] as well as Section 5.2.2 of this document.

Note that multipoint connections can be created without using any of these specialized commands. They are intended only as an aid to the construction of multipoint connections. Using the basic table manipulation commands, the switch tables can be modified in a manner which will provide the same functionality as the suite of multipoint commands.

3.3.8 `merge`

The `merge` command is one of the suite of commands used to create multipoint connections. The role `merge` plays in the creation of multipoint connections is to specify how the multiple inputs to

a switch are to be merged and rebroadcast to multiple output ports.

The syntax of this command is:

`merge` *port vpi vci*

This command indicates that all transmitters in the multipoint connection on the indicated VPI/VCI will be merged at the switch port indicated by the *port* argument. Each multipoint connection will have either zero or one merge ports, so at most one `merge` command is needed for any multipoint connection. Any multipoint connection with multiple transmitters and more than two receivers requires a merge command.

The `merge` command is similar to a `recycle` command in that it makes use of the recycling capabilities of the switch. The primary difference is that the `merge` command multiplexes the input streams while the `recycle` command duplicates the resulting multiplexed stream, but the underlying mechanism which enables these operations is the same. For further information on the recycling capabilities of the switch, see Section 3.2 of the System Architecture Documentation [3].

3.3.9 `receive`

For each receiver in a multipoint connection, one receive command will have to be issued. This command specifies the port, the VPI, the VCI and upstream discard flag for the receiver. The format of this command is:

`receive` *lr_string port vpi vci ud*

The port, vpi and vci arguments should be straightforward. The *ud* field takes as a value of either 1or 0 and indicates whether upstream cell discarding is turned on or not. However, the purpose of the *lr_string* field is less intuitive. This field is used to indicate the path followed through the binary multipoint tree to the receiver represented by this command. Each multipoint connection tree with *n* receiver nodes will have at most *n-2* recycling nodes. Each receiver node is a leaf in the tree, and can be reached by making successive decisions of whether to follow the path to the left or the right child from the current node. The length of this string is equal to the depth of the tree.

As an example, consider that the receiver in question is the left child of a single transmitter of a multipoint connection on VPI 0 and VCI 40 and is attached to port 5. In this case, the command would take the form:

```
receive l 5 0 40 1
```

Alternatively, if there are two more receivers in the connection, one attached to port 6 and the other to port 2, their receive commands would take the form:

```
receive rl 6 0 40 1
receive rr 2 0 40 1
```

For each of these receivers we have turned the upstream discard feature on.

A more illustrative example can be found in Section 5.2.2 of this document where an entire multipoint connection is constructed.

3.3.10 `recycle`

This command takes the same arguments as the preceding receive command. However, instead of indicating the position and port for a receiver leaf in the multipoint tree, this command indicates the position and port for a recycling or interior node in the multipoint tree.

For example, considering the illustration initiated in the previous section, if a multipoint connection has three receivers, it will have one recycling node as the right child of the transmitter and the parent of the second and third receivers. In this case, if the recycling port for this node is port 5, then the command to indicate this is:

```
recycle r 5 0 40
```

Again, for a more illustrative example, refer to Section 5.2.2.

3.3.11 `xmit`

The final command required to create a multipoint connection is the `xmit` command. This command must be invoked for each of the transmitters in the `build` command. This command takes five arguments which will characterize the input traffic. Its syntax is:

`xmit` *port vpi vci clp*

Again, the *vpi* and *vci* indicate the VPI/VCI pair on which the data is transmitted and the *port* indicates the switch port to which the transmitter is attached. The *clp* field takes a value of either 1 or 0 and indicates whether the CLP bit in the cells for this stream should be set or not.

A complete example of the construction of a multipoint connection is included in Section 5.2.2 of this document.

4. Running Jammer

Before running Jammer it is necessary to first run the software which enables the communication and control necessary for switch testing. This software has been consolidated into two programs:

- `NodeSim` - simulator of node controller
- `GBNSC` - GBN Switch Controller

Both NodeSim and GBNSC require certain configuration files for test network configuration and initialization. Also, NodeSim is only needed if a real switch is not available.

Once these programs are running successfully, Jammer can be invoked by typing:

`Jammer` *switch_address SC_host SC_port [include_file]*

where *switch_address* is the address in the form 0.0 (note that this is *not* an IP address), *SC_host* is the hostname on which the switch controller is running , SC_port is the TCP port on which the switch controller is listening for new connections and *include_file* is an existing Jammer test script. For further information, consult the installation documentation [4].

After invoking Jammer, the Jammer prompt will appear on the screen indicating that Jammer is ready and in interactive mode. Interactive commands may now be entered. To begin working with an existing test script which was not included at startup, type:

`include` *filename*

at the Jammer prompt where *filename* is the file containing the desired test script. Once this completes, type the procedure invocation of the main procedure in the file (if one exists). For instance, continuing the print-to-screen example from Section 3, this would be:

```
Enter command: include print_to_screen.js
Enter command: PrintToScreen()
    Hello!
Enter command:
```

After running the script, Jammer returns to interactive mode.

5. Testing

This section will contain a few general comments on testing. This section will grow as more of the testing functions mature. For now, basic information on iterative and conditional testing will be included to provide insight into the usage of Jammer. Also, detailed examples of data cell testing and multipoint connection building are given.

5.1. Script Basics

To expand on the discussion begun in Section 3.1, this section is intended to illustrate the basic test and program constructs necessary for switch testing. It is not intended to be an all inclusive illustration of the Jammer programming environment, but just to offer a some basic examples of test script constructs.

5.1.1 Conditional Testing

Perhaps the most important programming construct provided by Jammer is the conditional `if-else-fi` construct. Using this construct, the test engineer can check for failure of a command, for correct variable value, or for valid input from the user. For instance, to request user input regarding which port number to test, the test engineer could write:

```
int response

response = prompt “Indicate port number to be tested (0-7):”

if( $response < 0 || $response > 7 )
    echo “ERROR: Invalid port choice $response\n”
    echo “Port number must be value from 0 to 7\n”
else
    test_port( $response )
fi
```

This construct will appear in almost every test script.

5.1.2 Iterative Testing

As described in Section 3.1.10, the iterative control construct provided by Jammer is the `while - done` loop. This construct is most useful when it is necessary to test all register or table entries. This kind of testing would be difficult using interactive control.

For example, to set all entries of the virtual circuit table to zero, the following script excerpt could be used:

```
# define port processor and vcxt control variables
int pp
int vcxt_num
pp = 0

#loop through all port processors
while ($pp < 8)
    vcxt_num = 0
    echo “vcxt_test: Testing VCXT table $pp at all entries.”

    #loop through all vcxts
    while ($vcxt_num < 1023)

         #write zero values to each field
         write vcxt $pp $vcxt_num 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

         vcxt_num++
    done

    # wait for all outstanding tests to complete
    wait

    #test for failures
    if (failed)
        echo “vcxt_test: write to VCXT index $pp FAILED:”
        clear failed
    fi
    pp++

done
```

Of course, this is not the easiest way to zero out all bits in the VCXT table since the same operation could be accomplished using the `clear vcxt` command.

5.2. Intermediate Control

The next two sections will cover setting up multipoint connections and inserting data and control cells into the switch.

5.2.1 Data Cell Testing

One of the most powerful capabilities provided by Jammer is the ability to test not only the content of the tables and registers of the switch but also the content of the data cells being passed through the switch. The commands which enable this functionality are the `get cells` and `put cells` commands.

To illustrate the use of the `get` and `put cells` commands, the following example shows how to test cells placed on a certain VPI/VCI channel. In this example, the VPI to be tested is VPI 0, and the corresponding VCI is VCI 50. These values were chosen arbitrarily to show the usage of the get and put cells commands in interactive mode.

Before these commands can be used, the maintenance registers and translation tables in the switch must first be initialized to enable reading and writing of cells to the appropriate VPI/VCI pair. Next, the `get cells` command must be issued to instruct the switch controller to retrieve the

cells. Finally, using the `put cells` command, the cells are placed in the stream. These commands are shown in the following trace of an interactive Jammer session.

```
Enter command: write mr 0 2 0 128 32 0 255 1 1 1 1 1048576 0
Enter command: write vpxt 0 0 1 2 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
Enter command: write vcxt 0 50 1 2 1 0 0 0 0 0 0 1 0 0 50 1 0 0 0 0 0
Enter command:
Enter command: get cells 0 50

    Number of Cells to expect[1]: 1
    Number of seconds to wait for cells to return(0 for MAX)[1]: 0

          Now we begin to build the pattern to match
       in the payload field.
       If you don't care what was in the payload field enter 0 for the
       number of times to repeat the pattern:

    Number of times to repeat pattern(0 for NO pattern)[1]: 0

Enter command:  put cells 0 50

    Number of Cells to send[1]: 1

          Now we begin to build the pattern to put in
       the payload field.
       If you don't care what goes in the payload field
       enter 0 for the number of times to repeat the pattern:

    Number of times to repeat pattern(0 for NO pattern)[1]: 0

Enter command:
JAMMERLIST: Put Cells Operation Completed Successfully

Enter command:
JAMMERLIST: Get Cells Operation Completed Successfully
```

In the first three commands of this trace, the `write mr` serves to set up the software link which will enable to connection. Next, the `write vpxt` sets the VPT bit to terminate at VP 0 and the `write vcxt` sets cell routing to port 0 on VPI/VCI 0/50. Once the registers and tables have been initialized, the `get` and `put cells` commands are issued. For this example, defaults are accepted at the prompt with no pattern being specified.

With `get cells` and `put cells,` failures can take two forms. First, the cells can be lost. If this is the case the `get cells` operation will time out and report how many of the expected cells returned. The second failure mode is if the data in any of the cells was corrupted. Again the `get cells` operation will time out and report how many of the expected cells returned, but also the switch controller will print error messages for cells that returned on the appropriate VPI/VCI but that did not have the expected payload. This type of information can be very valuable when testing new hardware that might have data path errors.

5.2.2 Creating Multipoint Connections

One of the unique abilities of the gigabit switch is its facility for cell recycling and replication for the creation of multipoint connections in an efficient manner. The facilities provided by Jammer

for constructing multipoint connections are basic abstractions in the form of the `build`, `merge`, `receive`, `recycle`, and `xmit` commands. As noted previously, these commands are built on the basic table and register manipulation commands, but it is hoped that they will simplify the construction of multipoint connections to simplify switch testing.

To fully illustrate the usage of these commands, two multipoint connections will be described. The first is the same example shown in Section 3.2 of the System Architecture Document [3]. This example provides insights into the basics of multipoint connection construction. The second example is slightly more complex to illustrate a multipoint connection with multiple transmitters and multiple receivers.

The form of the multipoint connection of the first example can be viewed as a binary connection tree whose root is the transmitter and whose leaves are the receivers. This view is shown in Fig. 5.2.2.1 below.

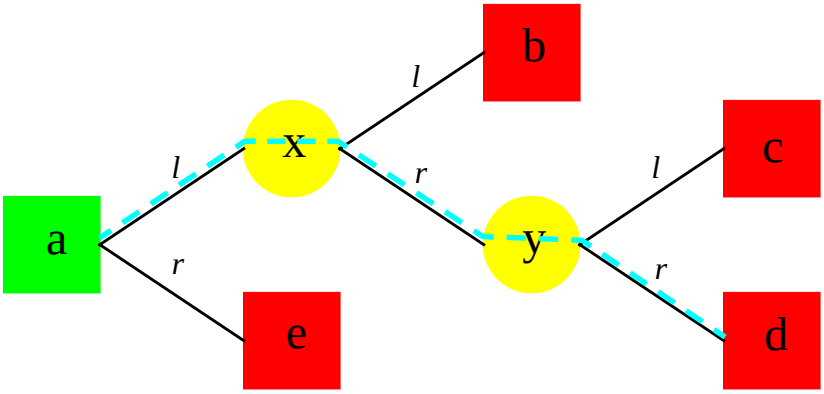

Figure 5.2.2.1: *Simple Multipoint Connection Tree*

In the above view, node *a* is the transmitter node, nodes *x* and *y* are recycling nodes, and the remaining nodes are receiver nodes. As in the example above, the multicast architecture above seeks to maintain a configuration where every internal node has two children to maintain switching efficiency. Because of this, the number of recycling nodes in a multicast connection will be two less than the number of receivers in the connection.

To create this connection as a continuous stream virtual circuit connection, the series of commands will take the following form:

```
build VCXT mpt 1 4 1                    (1)
xmit 1 0 40 1                           (2)
recycle l 2 0 40                        (3)
receive r 3 0 40 0                      (4)
receive ll 5 0 40 0                     (5)
recycle lr 4 0 40                       (6)
receive lrl 7 0 40 0                    (7)
receive lrr 6 0 40 0                    (8)
```

The first line serves to initiate the construction of the multipoint connection. It will have a single transmitter and four receivers. Data will be broadcast as a continuous stream. The transmitter is attached to port 1 and transmits on VPI/VCI 0/40. Upstream discard is allowed and the cell loss

priority bit is not set. The two recycling ports are ports 2 and 5. The *lr_string* argument for each of the recycling and receiving ports indicates how they are placed in the tree. For example, in line 8 it can be seen that node *d* is reachable by following the *lrr* path shown by the dashed line in Fig. 5.2.2.1.

In the above code sample, note that the tree is represented in a depth-first manner. The order of the statements which make up a multipoint connection call is not dictated by Jammer. The only requirement is that the build command must precede the other commands. Beyond that, any order is permitted so long as the entire connection is specified. Once Jammer receives information on all transmitters, recyclers, and receivers in the connection it issues the appropriate `write` commands to instruct the switch to construct the indicated connection.

To illustrate the use of multiple transmitters in a connection, consider the tree shown in Fig. 5.2.2.2 below.

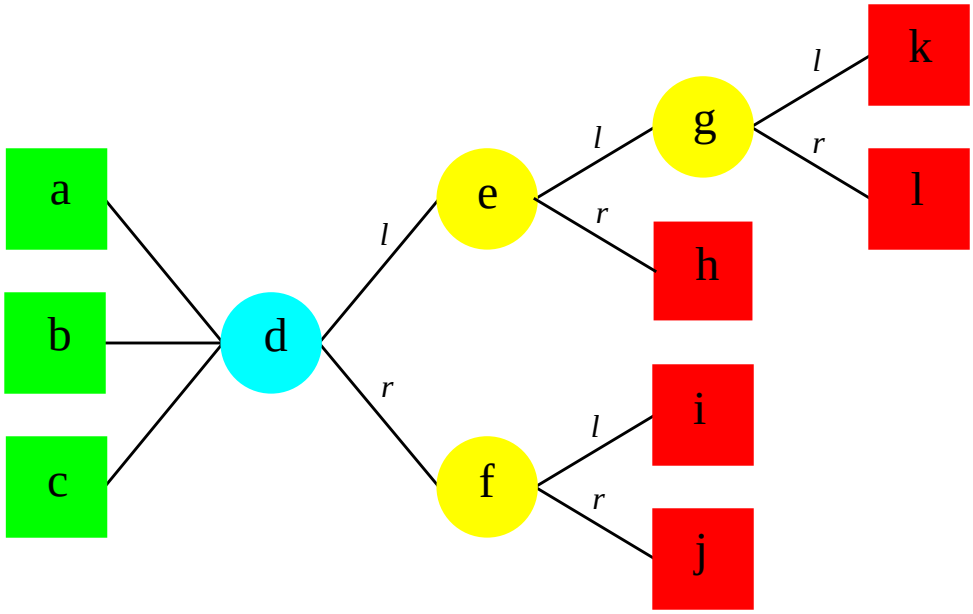

Figure 5.2.2.2: *Multipoint Connection with Multiple Transmitters*

For this example, the same basic settings like VPI/VCI, cell loss priority, and so forth will be maintained from the previous example in order to concentrate on the new issues introduced. In this example, nodes *a-c* are transmitter nodes, *d* is a merge node, nodes *e-g* are recycling nodes, and nodes *h-l* are receiver nodes. Again, note that the number of receiver nodes is two greater than the number of recycling nodes. Also, regardless of the total number of transmitters greater than one, there will only be one merge node in the tree.

To build this multipoint tree, the code fragment is:

```
build VCXT mpt 3 5 1                    (1)
xmit 1 0 40 1                           (2)
xmit 3 0 40 1                           (3)
xmit 5 0 40 1                           (4)
merge 4 0 40                            (5)
recycle l 0 0 40                        (6)
recycle r 2 0 40                        (7)
```

```
recycle ll 1 0 40                    (8)
receive lr 3 0 40 0                  (9)
receive rl 4 0 40 0                  (10)
receive rr 6 0 40 0                  (11)
receive lll 5 0 40 0                 (12)
receive llr 7 0 40 0                 (13)
```

As can be seen, there is little difference functionally between this code segment and that of the previous example. The addition of more transmitters and a merge node only adds one level to the height of the tree. The `merge` command itself is relatively simple requiring only the specification of the merge port and the VPI/VCI pair.

6. Conclusion

Jammer is a powerful, flexible tool for exhaustively testing Washington University's prototype ATM switches. By allowing the test engineer read and write access to each bit in the tables and registers as well as the data and control cells of the switch, Jammer provides complete control of all switch functions. Jammer's batch and interactive modes offer maximum flexibility in designing thorough test schemes to verify proper functionality of the switch. Together, these qualities make Jammer an indispensable tool in the verification of the design and operation of the prototype switches.

7. References

[1] Turner, J.S., “Fast Packet Switching System,” U.S. Patent 4 494 230 , January 15, 1985.

[2] Turner, J.S., “Design of a Broadcast Packet Switching Network,” in IEEE Transactions on Communications, 36 (6): 734-743, June 1988.

[3] Turner, J.S., ARL Staff, ANG Staff, “A Gigabit Local ATM Testbed for Multimedia Applications,” ARL Working Note-94-11, January 1995.

[4] DeHart, John D., “Washington University GigaBit Network Software Installation and Startup,” ARL Working Note 96-02.

[5] DeHart, John D., “Connection Management Software System (CMSS) Architecture,” ARL Working Note 95-03, July 1996.

[6] Wu, Dakang, DeHart, John D., Cox, Ken C., “GBNSC: The GigaBit Network Switch Controller,” ARL Working Note 94-12, July 1996.

[7] Wu, Dakang, DeHart, John D., “Node Controller Managed Object (NCMO) and Node Controller Communication Protocol (NCCP),” ARL Working Note 96-03, July 1996.

Appendix A: Jammer Grammar

Please see the current source code for information on the grammar of the Jammer language.