RIACS

Research Institute for Advanced Computer Science
NASA Ames Research Center

BIRD: A General Interface For Sparse Distributed Memory Simulators

David Rogers

January 1990

Research Institute for Advanced Computer Science
NASA Ames Research Center

RIACS Technical Report 90.3

NASA Cooperative Agreement Number NCC 2-387

BIRD: A General Interface For Sparse Distributed Memory Simulators

David Rogers

January 1990

Research Institute for Advanced Computer Science
NASA Ames Research Center

RIACS Technical Report 90.3

NASA Cooperative Agreement Number NCC 2-387

BIRD: A General Interface For Sparse Distributed Memory Simulators

David Rogers

Research Institute for Advanced Computer Science
NASA Ames Research Center - MS: 230-5
Moffett Field, CA 94035

RIACS Technical Report 90.3

January 1990

The Research Institute of Advanced Computer Science is operated by Universities Space Research Association, The American City Building, Suite 311, Columbia, MD 244, (301)730-2656

Work reported herein was supported in part by Cooperative Agreements NCC 2-387 between the National Aeronautics and Space Administration (NASA) and the Universities Space Research Association (USRA).

BIRD: A general interface for Sparse Distributed Memory simulators

David Rogers
1 August 1990

Document version 1.0 for
BIRD program release 3.169 syntax version 1.1

Research Institute for Advanced Computer Science
Mail Stop Ellis, NASA Ames Research Center
Moffett Field, CA 94035
(415) 604-6363
(Internet address: drogers@riacs.edu)

RIACS Technical Report 90.3

1. Purpose

Kanerva's sparse distributed memory (SDM) has now been implemented for at least six different computers, including SUN3 workstations, the Apple Macintosh, and the Connection Machine. A common interface for input of commands would both aid testing of programs on a broad range of computer architectures and assist users in transferring results from research environments to applications. A common interface also allows secondary programs to generate command sequences for a sparse distributed memory, which may then be executed on the appropriate hardware. The BIRD program is an attempt to create such an interface. Simplifying access to different simulators should assist developers in finding appropriate uses for an SDM.

BIRD can currently be used (currently) to access three different simulators for Kanerva's Sparse Distributed Memory (SDM). The first of these simulators is based on the local UNIX host; the second, on the SDM hardware prototype developed at Stanford; the third, on the Connection Machine. I anticipate future systems, such as MasPar and Cray, may be added to this interface in the future.

2. Introduction

The command language used by BIRD has several basic organizing features. It is text-based; no special characters are used. It is line-oriented; a command is composed of a series of command words, separated by whitespace characters (tab or space), and ended with a carriage-return. It is English-based; the command lines can be read as pseudo-English "sentences". It is case-independent; most command tokens can be written in any mixture of upper and lower case. It is extensible; future implementations may add new commands to the language to accommodate enhancements.

Given the diversity of simulators that currently run some version of the sparse distributed memory algorithm, it is inevitable that some commands in the interface will be functional in some implementations and not in others. For example, a given hardware prototype may require a specific memory size, or a software simulator may not have implemented features such as input masks and dont-care bits. In these cases, the BIRD will try to do the "appropriate" thing, whatever that may be, and to warn the user of the situation.

In this document, sample command lines will be shown preceded by the BIRD> prompt.

3. Description of a command line

The command lines are described by giving a line of boldface command keywords and bracketed special tokens. For example:

Set Memory Random-Seed <number>

This command is composed of three keywords (**Set, Memory,** and **Random-Seed**) and one special token (<number>). This description can be used to instantiate an instance of a command by filling in legal values for each special token in the command. For example, a legal command instantiated from the description above would be:

```
BIRD> Set Memory Random-Seed 113
```

Oftentimes, a given command has different legal tokens which are allowed; for example, the description may read:

Set System Time-Commands [Off | On]

The square brackets "[]" are used to delimit a list of options. You must select one of the options to make a legal command, such as:

```
BIRD> Set System Time-Commands On
```

It is also possible to have optional fields, where the user is allowed to request one of a set of options, but is not required to. In the case where the user does not specify any selection, the first item from the list is taken. Curly brackets "{}" are used to delimit such a list. An example is:

Set Memory Radius {Automatically | <number>}

In this case, the user can either specify the keyword **Automatically** or give a number. If the user does not give the fourth argument, the BIRD program selects **Automatically**. For example, the command:

```
BIRD> Set Memory Radius
 Radius set to 111
```

causes the BIRD program to automatically select a radius.

4. Interacting with the command line parser

We have already seen simple commands that the command line parser accepts. The parser has a number of features that assist the user in inputting a correct command line.

Nonsyntactic command lines generate an error. The program checks for syntax every time the user types a whitespace character (space, tab, or newline). When an error occurs, the parser signals the user, and leaves the user at BIRD top-level with a fresh prompt:

```
BIRD> Set Mmeory
?No match - "mmeory"
```

Empty command lines are accepted, but do no operation. The program simply returns again with the BIRD> prompt and waits for more.

The semicolon ";" is the comment character; it is ignored, and any characters following it are ignored, up to the end of line.

```
BIRD> Set Memory Radius    ; this command sets the radius
 Radius set to 111
```

The BIRD command parser does not require that full keywords be typed; only enough of the keyword must be typed to disambiguate it from the other choices. For the previous example, we could have typed:

```
BIRD> Set Mem Rad    ; this command sets the radius
 Radius set to 111
```

If in doubt, you can hit the "?" character at any time to get help as to what the legal completions of the line are. For example:

```
BIRD> set mem r? a keyword, one of the following:
Radius        Random-Seed     Reference-Type
BIRD> set mem r
```

In this case, the user typed "`set mem r?`" and the program responded by giving the list of possibilities, followed by retyping the command line (without the "?"). You can now complete the command line, or use backspace to rub out any or all of the current line.

The "?" will also show if there are any defaults. A default is the value assigned to the argument if it is not given by the user:

```
BIRD> dotimes ? the count
 or a string, beginning with "$"
 (default is "1")
BIRD> dotimes
```

There are several other characters you can use to interact with the command parser.

You can hit ESCAPE to have the program attempt to fill-out the current field. For example, if the command expects a file, and the file name is unique but long, typing ESCAPE will complete the name:

```
BIRD> Input File very-long-? an existing file name, one of the following:
very-long-file-name
BIRD> Input File very-long-<ESCAPE>
```

results in:

```
BIRD> Input File very-long-file-name
```

In this example, the "?" showed that only one very long name matched. Rather than typing in that name, we hit ESCAPE, and the command parser filled the name in, then beeped the terminal.

The "^U" (control-U) character erases the entire command line.

The "^R" (control-R) character retypes the command line.

DELETE and "^H" (control-H or backspace) are equivalent; these characters erase the previously typed character.

5. Description of some command tokens

The descriptions of BIRD commands contain a number of tokens, which are words delim-

ited by angle brackets "<>". We have already seen <number>, which represents an integer number; here is a list of additional tokens used in the command descriptions.

<number>: This token represented a positive or negative integer number. However, the fact that a command is syntactically valid does not mean that your command makes sense; for example, **set memory radius -1234** is syntactically valid, but is likely to signal an error when executed, or produce incorrect results later.

<variable>: This token represents an integer variable name, which is represented as a "$" followed by a string. The variables "$1", "$2", ..., "$9" are predefined; the user can create others with the **Create Variable** command. Variables can be used in BIRD nearly any place that numbers can be used.

<newfile>: This token represents the name of a file that is not expected to be currently on the file system. If it does not exist, it will be created; if it does exist, it will be removed and a new file created.

<oldfile>: This token represents the name of a file that is expected to be currently on the file system. An error is signalled and the command is not executed if the file does not exist.

<address>: This token represents a string that can be translated into a binary or trinary address. It expects a string of "0", "1", and possibly "*" characters, or a special text string which refers to an existing address.

If the string contains only "0", "1", and "*" characters, it is read as a series of binary digits. The "*" character is called a *dont-care bit*; in a Hamming-distance calculation, it matches either a 0 or a 1. If the string ends before the expected address size is reached, the remainder of the address is filled with "0" bits. If the string ends with a hyphen "-", packing is continued restarting at the beginning of the string. Some examples of addresses:

000111-	is read as	000111000111000111000111000111...
1*-	is read as	1*1*1*1*1*1*1*1*1*1*1*1*1*1*1*1*...
1111	is read as	11110000000000000000000000000000...

If the address contains more than these characters, it may be an *address tag*, which is a string of characters that represents some address. For example, the string "A" represents the last address given to the **Address** command. The user can create addresses using the **Create Address** command.

6. Why the BIRD program has so many commands

The BIRD program was developed as a research tool; I have made no attempt to trim the program for this release. This leads to two problems: first, the number of commands in the program is large, and second, many of the research-oriented commands are not robust and may fail.

Furthermore, many of the commands have been well-exercised on some simulators and not on others, leading to further potential problems.

I have resisted the temptation to strip the program, instead deciding to leave the system intact. This should allow both the simple user to access the stable, simple commands, and the more advanced user to risk using more powerful, advanced commands.

To assist the user in dealing with the large number of commands, I have organized chapters in this manual to first focus the user on the most-used and most-reliable commands in the BIRD program, then to develop more advanced topics.

To assist the user in accessing many of the more obscure commands, Appendix A contains a listing of all (or nearly all!) possible commands for the BIRD program in alphabetical order. The user can also use the **Help** command in the bird program to access most of the command descriptions on-line.

The user is encouraged to use the **Bug** command in the BIRD program to report bugs. This will allow future releases to incorporate known bug fixes.

I anticipate that this program will be used primarily as a research tool. If this expectation proves wrong, a future release may be made of a reduced BIRD program for application programmers who require more stability in their test environment.

7. Simple operation of the BIRD program

This section is designed to point the user to the most basic commands and command sequences that are useful for sparse distributed memory.

For most uses of the BIRD program, the basic chain of command events is as follows:

- **Set Default** commands are used to set up the defaults for the memory.
- The **Create Memory** command is used to create one or more memories.
- (Otherwise, the **Input Memory** command may be used to read a memory from a file).
- **Address**, **Read**, and **Write** commands are used to operate on the memory.
- **Show** commands are used to inspect the memory.
- The **Output Memory** command is used to save the memory to a file.

A sample sequence of BIRD commands is:

```
BIRD> Set Default Address-Size 256      ; 256 bits of address
BIRD> Set Default Data-Size 256         ; 256 bits of data
BIRD> Set Default Number-Of-Locations 1000  ; 1000 locations
BIRD> Create Memory Test                ; Memory named "Test"
 Memory Test, 1000 locations, size 256/256
```

```
BIRD> Address 1010-                          ; Address with 1010...
 Selected 21 locations, radius used was 111
BIRD> Write 11110000-                        ; Write
BIRD> Read                                   ; See if it's there
 Read data
 1111000011110000 1111000011110000 1111000011110000 1111000011110000
 1111000011110000 1111000011110000 1111000011110000 1111000011110000
 1111000011110000 1111000011110000 1111000011110000 1111000011110000
 1111000011110000 1111000011110000 1111000011110000 1111000011110000

BIRD> Show Memory                            ; Show memory info
 [lots of information about the setup of the memory]
BIRD> Show Output-Sums                       ; Show output sums
 [information about the output sums]
BIRD> Output Memory Test.memory              ; Output to a file
```

(You may wish to try to proceeding command sequence to ensure that your local version of the BIRD program is functioning. You should type the commands which are on the lines beginning with the BIRD> prompt (but don't retype the prompt!)).

(It is perfectly normal for the number of selected locations to be different from the 21 selected in the above examples.)

7.1. Using the Set Default command

The **Set Default** command is used to setup the parameters of the memory before it is created. (Many of these commands, but not all of them, can be reset using the **Set Memory** command after the memory is created.)

You can type "`Set Default ?`" for a full listing of the possible commands; the most useful of the commands are:

Set Default Address-Size [*<number>* | *<variable>*]

This sets the default address size for future memories. The address size is the number of bits used for addressing the memory. The initial value for this default is 256.

Set Default Data-Size [*<number>* | *<variable>*]

This sets the default data size for future memories. The data size is the number of bits used for reading from and writing to the memory. The initial value for this default is 256.

Set Default Number-Of-Locations [*<number>* | *<variable>*]

This sets the default number of locations for future memories. The program will attempt

to create future memories with that many hard memory locations.

Set Default Hardware [Software-Simulator | Connection-Machine | Stanford-Hardware]

This sets the default simulator for future memories. The program will attempt to create future memories using the specified simulator.

For example, we can set the defaults for memory creation to make a memory with 200 address bits, 10 data bits, and 8192 locations, and to use the Connection Machine:

```
BIRD> Set Default Address-Size 200
BIRD> Set Default Data-Size 10
BIRD> Set Default Number-Of-Locations 8192
BIRD> Set Default Hardware Connection-Machine
```

You can look at the current list of defaults by using the **Show Defaults** command.

Remember, you do not have to type the complete command, only enough for uniqueness. For example, the following command is acceptable:

```
BIRD> set def hard conn
```

7.2. Creating, inputting, and outputting memories

Once the memory defaults are assigned, we can create a memory using the **Create Memory** command.

Create Memory *<memory-name>* {*<address-size>*} {*<data-size>*}

This command creates new memory. The memory is assigned the given name. The address size and data size arguments are optional; if not given, the default address size and data size are used.

For example, we can create a new memory with the command:

```
BIRD> Create Memory Tweety 256 256
 Memory Tweety, 1024 locations, size 256/256
```

If the defaults were already set to an address size of 256 and a data size of 256, we could have typed to get the same result:

```
BIRD> Create Memory Tweety
 Memory Tweety, 1024 locations, size 256/256
```

If we try to create two memories with the same names, the program signals an error and

refuses to create the new memory:

```
BIRD> Create Memory Sylvester
 Memory Sylvester, 1024 locations, size 256/256
BIRD> Create Memory Sylvester
?Memory Sylvester already exists
```

Instead of creating a new memory, we can read in a memory previously output to a file (using the **Output Memory** command). We do this with the **Input Memory** command.

Input Memory *<oldfile>*

No memory name is supplied by the user; the memory is given the name it had when saved. If that name conflicts with a memory already in use, the string "New-" is prepended to it. For example, if we had previously saved a memory named "Age" in file Age.memory, we could type:

```
BIRD> Input Memory Age.memory
 [Addresses: 10% 20% 30% 40% 50% 60% 70% 80% 90% 100%]
 [Counters: 10% 20% 30% 40% 50% 60% 70% 80% 90% 100%]
 Memory New-Age, 0 writes, created Fri Dec 29 12:44:07 1989
```

You may wish to try to following command sequence to ensure that your local version of the BIRD program is functioning. You should type the commands which are on the lines beginning with the BIRD> prompt (but don't retype the prompt!):

```
BIRD> Set Default Number-of-locations 1000
BIRD> Create Memory Test1 256 256
 Memory Test1, 1000 locations, size 256/256
BIRD> Output Memory Test1.memory
 [Addresses: 10% 20% 30% 40% 50% 60% 70% 80% 90% 100%]
 [Counters: 10% 20% 30% 40% 50% 60% 70% 80% 90% 100%]
BIRD> Input Memory Test1.memory
 [Addresses: 10% 20% 30% 40% 50% 60% 70% 80% 90% 100%]
 [Counters: 10% 20% 30% 40% 50% 60% 70% 80% 90% 100%]
 Memory New-Test1, 0 writes, created Fri Dec 29 12:44:07 1989
```

Finally, you can switch from one memory to another using the **Set Active-Memory** command. It takes the name of a currently defined memory (that is, defined using either **Create Memory** or **Input Memory**) and makes that the "current" memory. The current memory is the one that all memory-related commands act upon; for example, **Output Memory** writes the current memory to a file, and **Address** does an addressing operation on the current memory.

Set Active-Memory *<memory-name>*

For example, the following sequence of commands creates two memories, then activates each of them in turn and does an addressing and write operation:

```
BIRD> Create Memory Test-A 256 256
 Memory Test-A, 1000 locations, size 256/256
BIRD> Create Memory Test-B 256 256
 Memory Test-B, 1000 locations, size 256/256
BIRD> Set Active-Memory Test-A
BIRD> Address 1100-              ; Address at 11001100...
 Selected 19 locations, radius used was 111
BIRD> Write 11110000-            ; Write 1111000011110000...
BIRD> Set Active-Memory Test-B
BIRD> Address 1100-
 Selected 18 locations, radius used was 111
BIRD> Write 11110000-            ; Write the same as for Test-A
```

It is possible to create the different memories on different hardware devices; thus, you could create one memory on the Connection Machine, and one memory on the front-end computer using the software simulator, and compare their results.

7.3. Using the Address, Read and Write commands

Once a memory has been created, we can perform memory operations on it. The primary memory operation are **Address**, **Read**, and **Write**.

Address does an addressing operation: it selects a subset of memory locations from the memory; these locations are then active for subsequent **Read** and **Write** operations. **Read** sums the data counters in the selected locations, and returns the result as an address. **Write** writes an address to the memory; that is, it changes the data counters in the selected locations so that future reads will likely return the written address.

(For a more detailed description of how a sparse distributed memory works, you should refer to one of the papers or books in the reference list.)

Now, for the full descriptions or **Address**, **Read**, and **Write**:

Address <*address*> {**Verbosely** | **Silently**}

The **Address** command takes two arguments: an address, and an optional keyword. The address is mandatory; it is used to change the state of the current memory. This command causes the selection a number of memory locations, which are then active for future **Read** and **Write** operations.

The second argument to **Address** is optional; it determines whether the command will print out helpful information about the addressing operation when completed. The default is for that information to be given.

Write *<address>*

The **Write** command writes the given address to the currently active memory. You must have some time previously used the **Address** command for this command to have any effect on the state of the memory.

Read {*<address>*} {**Verbosely** | **Silently**}

The **Read** command takes two arguments: an optional address, and an optional keyword. This command reads the memory; if no first argument is given, the command prints the read address. If the first argument is given, the read address is compared against that address and the distance between them is printed.

The second argument to **Read** is optional; it determines whether the command will print out distance information when the read operation is completed. The default is for that information to be given.

The simplest process for using the BIRD program involves creating a memory, addressing it at some sample address, writing a sample data address to it, then reading from the same sample address to confirm that the sample data address can be retrieved correctly:

```
BIRD> Create Memory Test                    ; Memory named "Test"
 Memory Test, 1024 locations, size 256/256
BIRD> Address 1010-                         ; Address with 1010...
 Selected 21 locations, radius used was 111
BIRD> Write 11110000-                       ; Write
BIRD> Read                                  ; See if it's there
 Read data
 1111000011110000 1111000011110000 1111000011110000 1111000011110000
 1111000011110000 1111000011110000 1111000011110000 1111000011110000
 1111000011110000 1111000011110000 1111000011110000 1111000011110000
 1111000011110000 1111000011110000 1111000011110000 1111000011110000
```

It is sometimes difficult to manually study the read data pattern for errors; in this case, you can use the first argument to **Read** to compute the distance between the read address and any given address:

```
BIRD> Read 11110000-                        ; Compare
 Distance is 0
BIRD> Read 11111111-                        ; Compare against 1's
 Distance is 128
```

Of course, the memory can store more than one pattern if they are stored at different addresses:

```
BIRD> Address 1010-                         ; Address with 1010...
```

```
Selected 21 locations, radius used was 111
BIRD> Write 11110000-                          ; Write
BIRD> Address 1110-                            ; Address with 1110...
Selected 20 locations, radius used was 111
BIRD> Write 00000001-                          ; Write
```

Now we can read at the two sample addresses:

```
BIRD> Address 1010-                            ; Address with 1010...
BIRD> Read                                     ; See what's there
Read data
1111000011110000 1111000011110000 1111000011110000 1111000011110000
[... etc ...]
BIRD> Address 1110-                            ; Address with 1110...
BIRD> Read                                     ; See what's there
Read data
0000000100000001 0000000100000001 0000000100000001 0000000100000001
[... etc ...]
```

7.4. Using the Show command

The **Show** command is used to show the user things about the current state of the BIRD program and the memories it contains. Here is a subset of useful **Show** commands:

Show All Memories

This command shows all currently defined memories. Example:

```
BIRD> Show all memories
Memory Woof, 1024 locations, size 256/256
Memory Arf, 8192 locations, size 256/256, on Connection Machine
```

Show Defaults

This command shows all the current defaults. Example:

```
BIRD> Show Defaults
Set Default Address-Size 256
Set Default Address-Type Bit-Addresses
Set Default Area 15
[... etc ...]
```

The **Show Defaults** command prints out the defaults in a format that looks like system commands; this allows the user to capture the commands for future use.

Show Distance *<address-1> <address-2>*

This shows the distance between two addresses. Example:

```
BIRD> Show Distance 11110000- 11110001-
Distance is 32
```

This example illustrates something we haven't previously seen: how does the system assign the length to recursive addresses (that is, addresses that end in "-")? In most cases, it is done by context: for example, if we use an address in the **Address** command, then the expected length is the address size of the memory. If there is no expected length given by the context, then the value of the default address size (as set by **Set Default Address-Size**) is used for the size.

Show Input-Sums

Most of the simulators keep values called an input sum. Whenever a write operation is conducted, the input sum counters are also incremented and decremented. The input sums are equivalent to a memory location which is always selected during writing, though it does not contribute during reading.

This command also shows the total number of data addresses written into the memory.

Show Location [<*number*> | <*variable*>]

This command gives extensive information about the given location. The command requires an argument, which is the index of the location in the current active memory. Locations are numbered from 1 to the number of locations in the currently active memory. Example:

```
BIRD> Show Location 333
[Lots of stuff about the location, including the location address and counters.]
```

Show Memory

This command gives a sequence of commands which should be able to recreate the given memory, though it does not keep track of the read and write operations which have taken place following memory creation.

```
BIRD> Show Memory
```
[Lots of **Set Default**, **Create Memory**, and **Set Memory** commands.]

Show Output-Sums

When a read operation is requested, the memory collects the data counters columnwise into sums, then thresholds these sums to create the read address. You can see these sums before thresholding with this command. For example:

```
BIRD> Show Output-Sums
```

[NOW: 16] 16 -16 [...]

In this example, NOW gives the number of individual location writes which contributed to the sums. The output sums are then printed out. In this case, the memory likely had the address 10101010... written to it near where we were reading.

Show Selected

This command shows a list of the selected memory locations from the last **Address** operation. it also gives the distance from that memory location address to the sample address. For example,

BIRD> `Address 1010-`
Selected 16 locations, radius used was 111
BIRD> `Show Selected`
Format is: index <distance>
59 <106>, 90 <110>, 175 <111>, 194 <110>, 298 <111>, 339 <109>, 390 <106>,
516 <110>, 583 <108>, 600 <111>, 611 <110>, 617 <108>, 662 <111>, 716 <106>,
776 <110>, 791 <109>

We could confirm that a given distance is correct by typing:

BIRD> `Show distance L59 1010-    ; dist btw loc 59 and address`
Distance is 106

7.5. Using the Set Memory command

Once a memory has been created, it is useful to be able to change parameters of its functioning. For example, we may wish to change the radius used by the memory, or switch the memory to area-based addressing, so that it uses the number of selected locations rather than a radius during addressing.

The following is a list of some of the more commonly-used commands.

Set Memory **Radius** {**Automatically** | *<number>* | *<variable>*}

When standard radius-based addressing is used, this command sets the radius. For example, we may decide that we wish to have a larger radius for our 256-bit address size memory than the standard radius of 111 bits; we could change it by:

BIRD> `Set Memory Radius 114`

Future **Address** commands will now use this new radius in calculating the set of selected

locations.

If the keyword **Automatically** is given, the program calculates what is considers a reasonable radius, and uses that value.

Set Memory Type-Of-Addressing [Area | Radius]

The Kanerva model assumes that the location addresses are randomly placed. When this is true, statistics operate well so that any address contains roughly the same number of locations within a given radius.

When the location addresses are not randomly placed, this is no longer true, and a one given address may select many more locations than another address. Because of this breakdown, I proposed the use of *area-based addressing*. Specifically, area-based addressing selects the radius only once the address is given to the **Address** command; it selects the radius to get as close to a desired number selected locations as possible. The default type of addressing is **Radius**; to use area-based addressing, type:

```
BIRD> Set Memory Type-of-addressing Area
```

(Area addressing is only available when using the software simulator or the Connection Machine simulator. It has not yet been implemented for the Stanford SDM hardware.)

Set Memory Area {Automatically | *<number>* | *<variable>*}

When area-based addressing is used, this command sets the desired area, that is, the desired number of selected locations. Experiments by Kanerva suggest that a reasonable number of locations is 1/2 * sqrt (number of locations); thus, for a 1000-location memory, an area of 1/2 * sqrt (1000) or 16 locations would be useful:

```
BIRD> Set Memory Area 16
```

Future **Address** commands will now use this area in calculating the set of selected locations when the memory has been set to area-based addressing mode.

If the keyword **Automatically** is given, the program calculates what is considers a reasonable area, and uses that value.

Set Memory Threshold [*<number>* | *<variable>* | Type]

The threshold of a memory is the number the memory uses to determine whether the data counter sum represents a "0" or a "1" in the read address. The default is zero; if the data sum is greater than or equal to zero, that bit is taken to be a "1", else it is taken to be a "0". Zero is a good threshold only if that bit is, on the average, "0" half the time and "1" half the time in the written data addresses.

Thresholding is a complex topic, and one which I won't get into there. You can refer to the section on the **Set Memory Threshold** command in Appendix A.

Set Memory Input-Mask [*<address>* | **None**]

The input mask is a set of bits that we should ignore during the addressing operation. Normally, the hamming distance is calculated using all the bits in the sample address; the input mask says that some of the bits should be ignored, that is, always be counted as matching during the hamming distance calculation. You will have to readjust the radius after you do this... the more bits are masked, the greater the number of addresses will be within a given radius.

Giving the keyword **None** as the argument unmasks all the bits.

This command is useful in conjunction with area-based addressing, as the radius will be dynamically changed to select the proper number of locations. For example, we can mask off every other bit:

```
BIRD> Set Memory Type Area
BIRD> Set Memory Area 16
BIRD> Addr 1010-
 Selected 16 locations, desired 16, radius used was 112
BIRD> Set Memory Input-Mask 101010-
BIRD> Addr 10010110011101011010010111111010010 1-
 Selected 19 locations, desired 16, radius used was 52
```

Note that the radius changed from 112 to 52. This is a sensible result; with half of the bits masked, the distances were roughly halved. Thus, one would expect the radius to also drop by about half.

7.6. Using the Output Memory command

Once we have written data to a memory, it is nice to be able to save that memory, in that state, for later restoration.

Output Memory *<newfile>* {**Binary** | **Hex**}

This command takes the current memory and outputs it to the given filename. That memory file can later be restored using the **Input Memory** command. If the default second argument is given, it is the format we wish to use for the output of the location addresses. The default output format is **Hex**.

The memory file is an ASCII file that can be transferred from machine to machine. While bulky, this is intentional: the desire was to allow a memory to be created using one simulator, and then loaded into another simulator, which may be residing on a different computer. A "transfer" of a memory currently in BIRD to another simulator can be done by saving the memory, setting a new default hardware type, and reading the memory back in:

```
BIRD> Output Memory Test1.memory
BIRD> Set Default Hardware Connection-Machine
BIRD> Input Memory Test1.memory
 [Addresses: 10% 20% 30% 40% 50% 60% 70% 80% 90% 100%]
 [Counters: 10% 20% 30% 40% 50% 60% 70% 80% 90% 100%]
 Memory New-Test1, 333 writes, created Fri Dec 29 12:44:07 1989
BIRD> Show All Memories
 Memory New-Test1, 8192 locations, size 256/1, on Connection Machine
 Memory Test1, 8192 locations, size 256/1
```

In this way, a memory can be "copied".

Someday, when the **Delete Memory** command works; I could suggest that you now delete the original memory to reclaim the space, but those days aren't here yet...

If you simply wish to save the memory location addresses, you will save space and time by using the **Output Location-Addresses** command:

Output Locations-Addresses <*newfile*> {**Binary** | **Hex**}

This command outputs all of the location addresses of the currently active memory to the given file. They can later be restored in a memory using the command sequence **Create Array** followed by **Initialize Memory Using-Address-Array**. The default output format is **Hex**.

```
BIRD> Output Location-Addresses LOCS Hex

[Then, much later...]
BIRD> Create Array locs LOCS
 Array locs contains 1024 addresses
BIRD> Initialize Memory Using locs
```

7.7. Using the Help command

A large number of on-line help files come with the BIRD program. The goal is to allow any user questions to be answered on the spot, without having to consult this manual.

The simplest use of the **Help** command is to just type "help" followed by the top-level command name. For example, we could get help on the **Plot** command:

```
BIRD> Help plot
BIRD> Help Plot
  PLOT [Address | Array | Data-File | Memory | Plot-File | Self-Array |
        Weather-Item]

    The PLOT command is used to create or display information best shown
```

as a graph.

The help file for the **Plot** command tells you a few things about the command, and also gives you the acceptable arguments for the command.

However, many commands have a command structure that goes down many layers. This command allows you to follow all keyword arguments by giving multiple arguments to the **Help** command. Thus, from looking at the **Plot** help file, we may see the argument **Address**; we can now get more help on the **Plot Address** command:

```
BIRD> Help Plot Address
   PLOT ADDRESS <address> <array> (Distances | Integrated-Distances | Both)

     This command plots the distribution of distances between the given
     address and the addresses in the given array. You can plot the distance
     distribution, the integration of the distance distribution, or both.
     It creates the files Address-info.data and Address-info.plot.
     If possible, it displays the created plot file.

     Example:

     BIRD> Plot Address 1010- W Distances  ; This plots the distance from
                                           ; 10101010... to each weather address
```

Thus, typing **Help Plot Address** got us a blurb on the command, and an example of its use. Such use of the **Help** command can reduce the need to reference this manual, and speed the learning of the BIRD program syntax.

Finally, some special topics not associated specifically with any one command are available by typing **Help On-Topic** <*topic-name*>.

7.8. Using the initialization file .birdrc and the Input File command

Often, the user begins a program run by typing a number of initialization commands. For example, a certain memory size might be a preferable default for your uses, and so you start by typing a sequence of **Set Default** commands. This process can be automated by using the file ".birdrc".

When started, the BIRD program looks for two files, in sequence, until it finds one that exists: ".birdrc", then "birdrc". If it finds a file, it assumes that it is a file of BIRD commands. It executes these commands before it displays the first prompt.

For example, my .birdrc file looks like this:

```
Set Default Address-Size 256
Set Default Data-Size 1
```

```
Set Default Type-of-addressing Area
Set Random Seed 333
Set Random Address-Seed 444
```

The first two commands set the default size of the memory. The third line makes the default for the memory to use area-based addressing. The last two lines set the random number generators to always given the same sequence, which aids in debugging.

It also is possible to create a file of BIRD commands and to execute them on demand, using the **Input File** command.

Input File *<oldfile>* {**Verbosely** | **Silently**}

This command assumes that the given file name is a file of BIRD commands. It opens the file and executes the commands until it reaches end of file.

The default is to show you the commands as they are executed. If you specify the second argument as **Silently**, the commands are not shown, though any output that the commands generate is still shown.

Appendix B contains a number of script files that can be executed in this fashion. The BIRD source directory contains these files as script1.incl, script2.incl, etc.

8. Advanced operation of the BIRD program

While the commands of the previous chapter allow the user some access to a memory, to run a full range of experiments the user needs to have a wider selection of commands. As a research tool, the BIRD program has grown enormously over the past year, and many of the advanced commands were added in response to a real experimental need. They are not always simple, but much effort was made to integrate them cleanly into the system, and to make their syntax obvious.

This chapter of the manual is composed of a number of sections. In each section, I assume that the reader has familiarity with material that was covered in previous sections, and has little familiarity with material that has not yet been presented. Thus, while not required, I encourage you to at least skim the earlier sections before jumping to a later topic that may interest you.

I have always disliked having to drop down into source code level to run my experiments; many of the features to be described are meant to bring the power of that level up to the user, so that simple experiments no longer require expertise in the "C" programming language or an understanding of the internals of the BIRD program, the Connection Machine, or of the Stanford hardware.

For your convenience, I offer a thumbnail sketch of each of the sections contained in this chapter:

8.1. Arithmetic operations: variables, assignment, and functions

This chapter discusses the creation and use of integer variables, and how to use simple arithmetic functions contained in the BIRD program. The use of variables is key to writing command scripts for the running of experiments.

8.2 Iteration and control commands

This chapter discusses the primitives available for iteration and control. Combined with variables, these commands give the BIRD command language enough power to be able to avoid writing code on the "C" level and having to integrate it into the BIRD source code.

8.3 Address tags and address arrays

The concept of "address" appears again and again when manipulating memories. I created a symbolic representation for an address, called an *address tag*, and a similar form for arrays of addresses. These address tags are much like variables; they can be created, assigned, and used in any place where an address is valid.

8.4 K nearest neighbor searches

Kanerva's sparse distributed memory is related to another scheme of associative memory called "K nearest neighbor" searching. Because it is common to compare the performance of newer associative memories with K nearest neighbor memories, I included K nearest neighbor style memories in the BIRD program. The mothod is to load the array of addresses into a SDM memory, with the memory read type (set using **Set Default Read-From**) set to use the associated array. The memory is set to area-based addressing (using the **Set Memory Type-Of-Addressing Area**) with a desired area of K (set using **Set Memory Area**).

I have not yet rewritten this section of the manual after my wholesale revision of how K nearest neighbor searches are done in the program.

8.5 Definition of new commands

A limited capability to define new commands is given. Another section of the program still under development, though what is there should work rather well.

8.6 Operating system calls: Bug and Exec

A few commands don't fit into other categories well, and operating system calls are among this group. A quick round-up of some random operating system calls that are accessible through the BIRD command level.

8.7 Plotting information

I usually prefer to see my information graphically. this is especially important when one is dealing with large amounts of statistically-based data. Thus, the BIRD program contains hooks for interfacing with a number of plotting programs.

8.8 Genetic recombination

There has been much recent interest in combining neural-network approaches with a class of algorithms known as Genetic Algorithms. This section of the BIRD program is the most experimental, and the user is advised to stay away from it unless they are ready to start playing with "C" source code internals, as many of these commands are still under development and have bugs. If you are interested, however, you should read my paper in the Proceedings of the 1989 Neural Information Processing Systems Conference.

8.1. Arithmetic operations: variables, assignment, and functions

To run experiments using the BIRD program, it is very useful to be able to do some simple arithmetic operations. To give the user this capability, the BIRD program includes commands which allow integer variables to be created, some simple arithmetic operations, and the assignment and testing of variables. These features (with the exception of testing, which will be discussed in a following section on Control) will be discussed in this section.

8.1.1. Integer variables

Variables in the BIRD program are denoted by a string name proceeded with the dollar sign "$". The program comes with the variables "$1", "$2", ..., "$9", and "$10" already created. These variables can be used in nearly any place where an integer value is legal. For example:

```
BIRD> Show Variable $5
 Value is 112
BIRD> Set Memory Radius $5
BIRD> Address 1010-
 Selected 24 locations, radius used was 112
```

In this example, we set the memory radius not to a number, but to the value of the variable "$5".

We can assign the value of a variable using the **Assign Variable** command:

```
BIRD> Assign Variable $5 3333
BIRD> Show Variable $5
 Value is 3333
```

Assign Variable *<variable>* [*<function>* | *<variable>* | *<number>*]

This command assigns the variable to the value of the given number, variable, or function.

(The legal functions will be presented in the following section.)

While 10 variables are useful, users may wish to create their own variables, either because they need more than 10, or because they wish to give the variables more descriptive names. You can create variables using the **Create Variable** command:

Create Variable *<variable>* [*<function>* | *<variable>* | *<number>*]

This command creates a new variable of the given name. For example, we could rewrite the above example to be more descriptive:

```
BIRD> Create Variable $Radius 112
BIRD> Set Memory Radius $Radius
BIRD> Address 1010-
 Selected 24 locations, radius used was 112
```

As shown above, you can see the value of any variable using the **Show Variable** command:

Show Variable [*<function>* | *<variable>* | *<number>*]

For example:

```
BIRD> Show Variable $Radius
 Value is 112
```

If a variable that has not been created is given to this command, the value "0" is printed. This is a widespread practice; using a variable that has not been defined is not an error, but uses the value zero. This is true for all commands except assignment; it is an error to try to assign the value of a variable that does not exist. For example:

```
BIRD> Assign Variable $never-created 12345
?Not a variable: $never-created
```

We can delete variables using the **Delete Variable** command:

Delete Variable *<variable>*

This command deletes the given variable name from its table of variables. Variables are fairly inexpensive to keep around, so you shouldn't need to worry about this command unless you are used to cleaning up after yourself.

(Some system-created variables are undeletable; the system will signal an error and refuse to delete them if you try.)

We can get a list of all the defined variables using the **Show All Variables** command:

Show All Variables

This command prints a list of all currently defined variables and their current value:

```
BIRD> Show All Variables
$NUMBER-OF-WRITES: 0, $TOTAL-NUMBER-OF-WRITES: 0,
$DISTANCE: 0, $GENETIC-DONE: 0, $LOCATION-SELECTED: 0,
$UNWEIGHTED-DISTANCE: 0, $LOCATION-DISTANCE: 0,
$GENETIC-FAILED: 0, $GENETIC-OK: 0, $LOCATION-AVG-VOTE: 0,
$BAD: 0, $GOOD2: 0, $GOOD1: 0, $LOCATION-RADIUS: 0,
$SELECT-COUNT: 24, $SDM-RADIUS: 111,
$PROGRAM-ID: 649180109, $LOCATION-BIRTHDATE: 0,
$LOCATION-WEIGHT: 0
```

Finally, we can do two other operation with variables, called pushing and popping.

Push Variable *<variable>* [*<function>* | *<variable>* | *<number>*]

Pop Variable {*<variable>*} {*<variable>*} {*<variable>*} {*<variable>*} {*<variable>*} {*<variable>*} {*<variable>*} {*<variable>*} {*<variable>*}

The **Push Variable** command pushes a new value onto the given variable name; future references to that variable will use that new variable. However, the old value is retained, and is restored when the **Pop Variable** command is given. These commands are useful when you wish to use a variable, but are not sure if other routines have already assigned its value.

The **Pop Variable** command can take up to 9 variables to pop.

Variable pushing and popping is mostly used in script files to avoid smashing the values of variable between script files.

8.1.2. Arithmetic functions

The previous section showed how to create and manipulate variable. To get full use of these variables, however, it is necessary to be able to perform simple arithmetic operations on variables and numbers. The BIRD program contains some commands which allow simple arithmetic operations; these commands are available anywhere the command accepts the *<function>* token. Wherever this token is legal, any of the following list of commands is legal:

<Function>: {**Not**} [**Address-Size** | **Array-Size** | **Count** | **Distance** | **EQ** | **NEQ** | **GT** | **GE** | **LT** | **LE** | **If-Undefined** | **Location-Selected** | **Minus** | **Mod** | **Not** | **Plus** | **Quotient** | **Random** | **Real-Index** | **Same** | **Selected-Location** | **Times** | **Variable-Exists** | *<number>* | *<variable>*]

Some commands which accept an argument of an integer value allow simple mathematical

functions to be given. For example, **Assign Variable** allows the variable to be assigned to the value of a function:

```
BIRD> Assign Variable $ARF Plus $Dog $Bone
BIRD> Echo SUM: $Dog "plus" $Bone "equals" $Arf
SUM: 11 plus 22 equals 33
```

Another example is the creation of a new variable:

```
BIRD> Create Variable $NewArf Plus $arf 10
```

The ability to perform simple arithmetic, when combined with simple control constructs such as **If** and **Do** (see following section for a description of Control), allows the user to write command scripts directly to the BIRD command interpreter, without having recode program internals.

The following is a simple explanation of the possible functions:

Not [*<function>* | *<variable>* | *<number>*] returns "0" if the argument has a non-zero value, and returns "1" otherwise.

```
BIRD> Assign Variable $1 Not Equal 333 444
BIRD> Show Variable $1
 Value is 1
BIRD> Assign Variable $1 Not EQ 333 333
BIRD> Show Variable $1
 Value is 0
```

Address-Size *<address>* returns the number of elements in the given address.

```
BIRD> Create Memory Aardvark 256 256
  Memory Aardvark, 100000 locations, size 256/256, on Connection Machine
BIRD> Address 11110000-
 Selected 376 locations, desired 350, radius used was 102
BIRD> Print A
 Address:
 1111000011110000 1111000011110000 1111000011110000 1111000011110000
 1111000011110000 1111000011110000 1111000011110000 1111000011110000
 1111000011110000 1111000011110000 1111000011110000 1111000011110000
 1111000011110000 1111000011110000 1111000011110000 1111000011110000

 Bit count: 1:128 0:128
BIRD> Assign Variable $ASize Address-Size A
BIRD> Show Variable $Asize
 Value is 256
```

In this example, we create a memory and address it. The tag "A" contains a copy of the

address we used. We assign the variable $ASize to the size of this address. This may be useful if we have multiple memories of different sizes, and need to change our commands depending on the address size.

Array-Size *<address-array>* returns the number of addresses in the given address array.

For example, the array "L" (L1, L2, etc) is the size of the number of memory locations. We can test:

```
BIRD> Create Memory Aardvark 256 256
 Memory Aardvark, 100000 locations, size 256/256, on Connection Machine
BIRD> Assign Address $1 Array-Size L
BIRD> Show Variable EQ $1 100000
 Value is 1
```

In this example, we create a memory, and then assign variable "$1" to the size of L. This should be equal to the number of locations in the memory; we test that in the final statement, and it returns "1" (true).

Count *<address>* returns a count of the number of non-zero elements in the given address.

The **Count** keyword is useful for counting bits. Here is an example (which admittedly uses some commands we haven't seen yet):

```
BIRD> Create Address AAA 256 Bit
BIRD> Create Address BBB 256 Bit
BIRD> Create Address CCC 256 Bit
BIRD> Assign Address AAA Z
BIRD> Assign Address BBB Z
BIRD> Assign Address CCC XOR AAA BBB
BIRD> Show Variable Count CCC
 Value is 123
```

In this example, we create three 256-bit binary addresses, AAA, BBB, and CCC. We assign the first two addresses to random values, then assign the last to the XOR of the first two. We can then count the number of bit positions at which AAA and BBB differ. A more direct way of doing this example would have been to use the **Distance** keyword:

Distance *<addr1>* *<addr2>* returns the L1 distance between the addresses. (For binary addresses, this is the Hamming distance.)

```
BIRD> Create Address AAA 256 Bit
BIRD> Create Address BBB 256 Bit
BIRD> Assign Address AAA Z
BIRD> Assign Address BBB Z
BIRD> Show Variable Distance AAA BBB
```

Value is 123

In this example, we create two addresses, and set them to random values. We can then calculate the distance between them.

EQ	[*<variable1>* | *<number>*]	[*<variable2>* | *<number>*]
NEQ	[*<variable1>* | *<number>*]	[*<variable2>* | *<number>*]
GT	[*<variable1>* | *<number>*]	[*<variable2>* | *<number>*]
GE	[*<variable1>* | *<number>*]	[*<variable2>* | *<number>*]
LT	[*<variable1>* | *<number>*]	[*<variable2>* | *<number>*]
LE	[*<variable1>* | *<number>*]	[*<variable2>* | *<number>*]
Minus	[*<variable1>* | *<number>*]	[*<variable2>* | *<number>*]
Plus	[*<variable1>* | *<number>*]	[*<variable2>* | *<number>*]
Quotient	[*<variable1>* | *<number>*]	[*<variable2>* | *<number>*]
Times	[*<variable1>* | *<number>*]	[*<variable2>* | *<number>*]
Mod	[*<variable1>* | *<number>*]	[*<variable2>* | *<number>*]

A large block of functions involve simple arithmetic operations on integer arguments. I won't go into detailed explanations of these keywords; suffice it to say that each of them returns the ''obvious'' value from a computation on its arguments. A simple example is:

```
BIRD> Assign Variable $ARF Plus $Dog $Bone
BIRD> Echo SUM: $Dog "plus" $Bone "equals" $Arf
SUM: 11 plus 22 equals 33
```

If-Undefined <variable1> <variable2> returns the value of <variable1> if it is undefined; else, it returns the value of <variable2>.

This keyword is most useful in script files. Arguments are given to scripts through integer variables; it is often useful to have the equivalent of "optional" arguments that some languages support. This command allows that; for example, let's say we wish to use a variable "$Size", and to have it default to a value of 1000 if it was not defined previously:

```
BIRD> Create Variable $Size If-Undefined 1000 $Size
```

This command creates a variable $Size; if the variable already existed, it is simply assigned to its previous value, else it is initialized to 1000.

This command is especially useful with the **Push Variable** command.

Location-Selected [*<variable>* | *<number>*] returns 1 if the given location is selected in the current memory, and 0 otherwise.

```
BIRD> Address 11110000-
 Selected 376 locations, desired 350, radius used was 102
BIRD> If Location-Selected 10
```

```
BIRD>   Echo "Location 10 was selected"
BIRD> End
```

Random [<*variable*> | <*number*>] returns a number in the range [0, <variable>).

Sometimes randomness can be useful.

```
BIRD> Show Variable Random 10
 Value is 3
BIRD> Show Variable Random 10
 Value is 9
BIRD> Show Variable Random 10
 Value is 1
```

(Of course, this command sequence would have been easier to write using the **DoTimes** command, which will be explained in a following section:)

```
BIRD> DoTimes 3
DOTIMES> Show Variable Random 10
DOTIMES> End
 Value is 3
 Value is 9
 Value is 1
```

Real-Index <array-element> returns the physical index of the array element, ignoring any reordering which have taken place.

This keyword isn't very useful to anyone other than real hackers. As we reorder arrays and create array aliases, it is sometimes useful to be able to access the physical index of an array in a file. This command allows that.

Same <address1> <address2> returns "1" if the addresses are the same, and "0" otherwise.

This keyword tests for the identicalness of the two addresses. Useful with control statements such as **If**; we can make a block of commands conditional on the equality (or nonequality, if proceeded with **Not**) of two addresses.

Selected-Location [<*variable*> | <*number*>] returns the index of the <*number*>th selected location.

This is used to walk-through the list of selected locations. The first selected location is index **1**. For example:

```
BIRD> Address 11110000-
 Selected 2 locations, desired 3, radius used was 102
BIRD> Show Selected
```

```
Format is: index <distance>
565 <102>, 864 <102>
BIRD> Show Variable Selected-Location 1
 Value is 565
BIRD> Show Variable Selected-Location 2
 Value is 864
```

Variable-Exists <variable> returns "1" if the variable name has been defined, and "0" otherwise.

This command is related to the **If-Undefined** keyword. It similarly tests for the existence of variables; for variables with more complex initialization routines, this keyword, combined with a control statement such as **If**, can make a block of command conditional on the existence or non-existence of a variable.

8.2. Iteration and control commands

Iteration (the execution of a command sequence a number of times) and control (the optional execution of commands) are essential if the BIRD interface is to be general enough to allow the user to perform useful experiments. The BIRD interface has four such commands: **Do**, **DoTimes**, **If**, and **While**.

These commands differ from most other commands in that they take more than one line to fully specify. For each one, after the initial command line is typed, the user is prompted for a set of syntactically-correct command lines, which *are not executed*, but are stored. The user completes the input of command lines with the **End** command. It is only at this point that the command sequence is executed.

DoTimes [<*number*> | <*variable*>]

The simplest command of this class is **DoTimes**; it simply repeats the execution of a given block of commands a given number of times:

```
BIRD> Dotimes 3
DOTIMES> Echo "ARF!"
DOTIMES> End
ARF!
ARF!
ARF!
```

This command illustrates the basic features of this class of commands. First, the **DoTimes** command is entered, which places the user into command-input mode. This mode change is made apparent to the user by a change of prompt to DOTIMES>. Next, the user inputs a command sequence, in this case, the single **Echo** command, then terminates the input with an **End** command. At this point the command is executed the given number of times.

Do *<do-variable>* {*<start>* | *<variable>*} {*<end>* | *<variable>*} {*<step>* | *<variable>*}

A more complex iteration command is the **Do** command. It repeatedly executes a block of commands, but also assigns a variable to a new value for each iteration.

The do variable is the name of a currently existing variable. The next argument is the starting value of the do variable; if not given, this defaults to "1". The next argument is the final value of the do variable; when the iteration results in a variable value over this number, the command finishes. This value defaults to "1" if not given. The final argument is the step size, which is the amount by which the do variable is incremented each step. The value defaults to "1" if not given.

For example, we could have rewritten the command sequence given in the **DoTimes** explanation by having it print the iteration number:

```
BIRD> Do $1 1 3
DO> Echo "ARF: " $1
DO> End
ARF: 1
ARF: 2
ARF: 3
```

The **Do** command is useful for manipulating address arrays. If an array name ends with the name of an integer variable, the variable is replaced by its value. For example, we could write a memories own location addresses to itself:

```
BIRD> Create Variable $NOL Array-Size L
BIRD> Do $1 1 $NOL
DO> Address L$1
DO> Write L$1
DO> End
[Print out of each addressing operation follows]
```

In this sequence, "L$1" is replaced each step by the value of "$1"; for example, if the value of "$1" is 123, then "L$1" is equivalent to L123, which is the address of the 123rd memory location.

This issue is discussed more in the section on Address Tags and Address Arrays.

If [*<function>* | *<variable>* | *<number>*]

This command is used to conditionalize a sequence of commands. "True" is defined as a non-zero value, and "false" a zero value. Thus, "**If** 1" always executes its block of commands, and "**If** 0" never executes its block of commands.

For an example of the use of this command, we may wish to print out a message whenever the read address (kept in the address tag "R") and the written address:

```
BIRD> If Not Same R 1010-
IF> Echo "The addresses are not the same"
IF> End
```

It is also possible to use the **Else** command to separate the block of commands into two sections, the first of which is executed if the condition is true, and the second executed if the condition is false.

Else

The else command is only legal inside of an **If** command block. For example:

```
BIRD> If Same R 1010-
IF> Echo "The addresses are the same"
IF> Else
IF> Echo "The addresses are not the same"
IF> End
```

This sequence of command will print out a message every time it is executed, but the message printed will vary depending on whether the addresses are the same or not.

While [*<function>* | *<variable>* | *<number>*]

Finally, we have a command which executes its block as long as the test is true (non-zero). For example, we may wish to write an address to the memory was long as reading the address does not result in the written address:

```
BIRD> Read
BIRD> While Not Same R 1010-
WHILE> Write 1010-
WHILE> Read
WHILE> End
```

8.3. Addresses, address tags and address arrays

Many commands in the BIRD program require an address; to this end, the program has methods for creating, inputting, and manipulating addresses. This section will discuss the use of addresses, address tags, and address arrays.

Most important to the user is the ability to specify an address in a command. For example, assuming a memory has been created, the user may wish to do an **Address** command to the memory.

The simplest form of inputting an address requires that the user type in a string of the char-

acters "0" and "1". If the memory requires 32 bit input addresses, we could given the command:

```
BIRD> Address 10101010101010101010101010101010
 Selected 15 locations, radius used was 12
```

In this case, we typed out the full 32 character string. However, the input routines recognize the character "-" as a signal for recursion: reading continues at the beginning of the string. Thus, the above example is equivalent to:

```
BIRD> Address 10-
 Selected 15 locations, radius used was 12
```

All addresses are implicitly padded with zeroes at the end; that is, if there are not enough characters to complete the address, the remaining bits are loaded with "0" bits.

8.3.1. Address tags

It would be tiring to always refer to addresses by their full typed expansion. Thus, the BIRD program contains alphanumeric "tags" which represent addresses. It is also possible to create new "tags" using the **Create Address** command. For example, we could have written the example in the previous section using the **Create Address** and **Assign Address** commands.

Create Address *<name>* [*<address-size>* | *<variable>*]
{**Bit-Addresses** | **Float-Addresses**}

This command creates an address tag of the size and type. In the future, you can use the tag name in any command that requires an address.

Assign Address *<destination-address>* [*<address-op>* | *<address>*]

This command assigns the destination address to the value of the source address or the given operation. (The definition of legal address operation is given later this chapter.)

Given these two commands, we can now write:

```
BIRD> Create Address AAA 32 Bit-Address
BIRD> Assign Address AAA 1010-
BIRD> Address AAA
 Selected 15 locations, radius used was 12
```

In this case, we created a new address tag named AAA, assigned a value to it, then used that tag to address the memory.

The BIRD program also contains several built-in address tags, which are automatically created by the program:

"A" refers to the last address used to address the memory. For example, after the previous example, the tag A would contain the value 10101010101010101010101010101010.

"R" refers to the last address read from the memory.

"W" refers to the last address written to the memory

"M" refers to the current input mask for the current memory.

"Z": returns a random address with no dont-care bits. Each reference using "Z" returns a different random address.

8.3.2. Address arrays

Some address come naturally in the form of arrays rather than individually. For example, the location addresses of a memory can be best considered as an array of addresses. The BIRD program allows the manipulation of arrays, both as individual elements and as blocks.

Create Array *<array-name> <oldfile>*
Create Array *<array-name>* [*<array-length>* | *<variable>*] [*<address-size>* | *<variable>*]
{**Bit-Addresses** | **Float-Addresses**}

This first command creates an array using the addresses in the given file. The addresses are not read into the memory; instead, a list of pointers to the addresses is kept, and the addresses are read in as needed. This allows the system to deal with very large databases of addresses, at a price of some speed.

The second command create an array in memory with the given number of addresses in it. While more memory-consumptive, this array is faster to access; for arrays that will be used a lot, you may wish to transfer them into an internal array by copying using the **Assign Array** command.

Assign Array *<address-array> <source-address-array>* {*<start>* | *<variable>*}
{* | *<end>* | *<variable>*}

This command is used to assign a block of addresses from one array to another. The command tries to copy as many addresses as it can; the user can limit the range of addresses in the source array by giving a start and end index.

For example, if we had two arrays, an internal array named INTERNAL and a file array named TEST, we could copy from the file array to the internal array:

```
BIRD> Assign Array INTERNAL TEST
 Copied 1024 addresses
```

It is also possible to assign only a range of addresses:

```
BIRD> Assign Array INTERNAL TEST 100 199
Copied 100 addresses
```

The BIRD program also contains several built-in address arrays, which are automatically created by the program:

"L" followed by a number, such as "L100", refers to the address of the corresponding memory location. (Locations are numbered starting at 1).

"Z" followed by a single-digit number, such as "Z5", returns a random address with no dont-care bits. Each reference returns the same random address.

When a memory is created, the memory name can be used to reference the location addresses of that memory. (If the name is already taken, a new name is given to that memory's location addresses.) For example, if we had typed **Create Memory ARF**, then we could use "ARF10" to refer to the tenth memory location address.

If the Darwin weather data is included in your BIRD release, then two additional address arrays are defined:

"W" followed by a number, such as "W2345", returns a weather state from the Darwin weather file, coded into a 256-bit address. (Note that "W" with no trailing number is a different type of address, namely the last address written to the memory.)

"R" followed by a number, such as "W2345", returns rain information from the Darwin weather file, coded into a 1-bit address. The bit is "1" if it rained during that weather period, and "0" otherwise. (Note that "R" with no trailing number is a different type of address, namely the last address read from the memory.)

8.3.3. Address operations

The previous section showed how to create and manipulate addresses and address arrays. To get full use of these addresses, however, it is necessary to be able to perform simple address operations on them. The BIRD program contains some commands which allow simple address operations; these commands are available anywhere the command accepts the *<address-op>* token. Wherever this token is legal, any of the following list of commands is legal:

<address-op>: [**And** | **Coords** | **Cross** | **Mutate** | **Not** | **Or** | **Random** | **Xor** | *<address>*]

Some commands which accept an argument of an address allow these address operations to be given. For example, **Assign Address** allows the address to be assigned to the value of an address operation:

```
BIRD> Assign Address ARF XOR DOG 10101010-
BIRD> Echo "The XOR of BOG and 101010- is"
BIRD> Print ARF
```

Address:
1011010001111110 0110101000001101 1011101010110000 1000110001011011
0001010101010000 1110001111011111 0011111001111101 0010100000000110
0011100010001110 1111111001011011 1000011011101011 0110000100110110
1001001111101101 0111010110010000 1111100101110001 1000100000010000

Bit count: 1:130 0:126

The following is a simple explanation of the possible address operations:

And *<address-1>* *<address-2>* returns the boolean AND of the given addresses. If the addresses are float addresses, it returns the MIN of the addresses.

Coords [*<number>* | *<variable>*] returns an address which contains exactly *number* bits "1" and the remainder "0"..

Cross *<address-1>* *<address-2>* returns a crossover between the two addresses.

Mutate *<address-1>* [*<number>* | *<variable>*] returns a copy of the first address with *number* bits mutated.

Not *<address-1>* returns the boolean NOT of the first address. If the address is a float address, it returns an address with each bit having the value (1.0 - b) of the original address..

Or *<address-1>* *<address-2>* returns the boolean OR of the given addresses. If the addresses are float addresses, it returns the MAX of the addresses.

Random returns a random address.

Xor *<address-1>* *<address-2>* returns the boolean XOR of the given addresses. If the addresses are float addresses, I'm not sure what I do, perhaps average.

8.3.4. Using dont-care bits

If the string contains only "0", "1", and "*" characters, it is read as a series of binary digits. The "*" character is called a *dont-care bit*; in a Hamming-distance calculation, it matches either a 0 or a 1. If the string ends before the expected address size is reached, the remainder of the address is filled with "0" bits. For example, we could address a 32-bit memory while ignoring the first 6 bits:

```
BIRD> Address ******10101010101010101010101010
Selected 345 locations, radius used was 12
```

Dont-care bits in the address are equivalent to a one-time input mask.

Of course, since the dont-care bits always match, you need to reduce the radius to avoid

selecting too many locations. The easiest way to use dont-care bits is to set the memory for *area-addressing*: that is, to select a radius automatically to give the proper number of selected locations. In the previous example, we selected too many locations. It would have worked better if we had set the memory to area addressing:

```
BIRD> Set Memory Type-Of-Addressing Area
BIRD> Set Memory Area 15
BIRD> Address ******10101010101010101010101010
 Selected 10 locations, desired 15, radius used was 9
```

8.4. K nearest neighbor searches

(This section not yet written.)

8.5 Definition of new commands

As BIRD developed, I found myself creating command files which performed specific functions. For example, I wanted a command that would fill a memory with a given number of random addresses. Tiring rather quickly of constantly using the **Input File** command to input this sequence of commands, I created the **Define** command to allow the definition of new top-level command sequences.

Define *<name>* {**Variable** *<variable>*} {**Variable** *<variable>*} {**Variable** *<variable>*
{**Variable** *<variable>*} {**Variable** *<variable>*} {**Variable** *<variable>*}
{**Variable** *<variable>*} {**Variable** *<variable>*} {**Variable** *<variable>*}

This command is used to define new top-level commands. The arguments allowed are currently all variables; when the command is given by the user, the user's arguments are bound to these variables, then the command is executed.

It is usually more efficient to define a command rather than to constantly use the **Input File** command.

For example, we could create the **Fill** command, which writes a given number of random data patterns into the memory at random locations:

```
BIRD> Define Fill Variable $How-Many
DEFINE> Dotimes $How-Many
DOTIMES2> Address Z Silently
DOTIMES2> Write Z
DOTIMES2> End
DEFINE> End
```

At this point, the **Fill** command is defined. We can now fill the memory with a given number of patterns with:

```
BIRD> Fill 100
```

You can see all command definitions using the **Show All Definitions** command. You can see a specific definition using the **Show Definition** command.

Show All Definitions

This command shows all commands defined with the **Define** command. For a more detailed description of an individual definition, use the **Show Definition** command.

```
BIRD> Show All Definitions
There is 1 command defined.
 Define FILL Variable $How-Many
```

Show Definition {*<command-name>*}

This command shows the definition for the given command, if the command was previously defined using the **Define** commands. For example:

```
BIRD> Show Definition Fill
 Define FILL Variable $How-Many
  DoTimes $How-Many
   Address Z Silently
   Write Z
  End
 End
```

Finally, we can delete obsolete command using the **Delete Definition** command:

Delete Definition *<defined-command>*

This command deletes the given defined command from the system. The command must have been previously created using the **Define** command.

8.6 Operating system calls: Bug and Exec

The BIRD program contains some commands that call operating system routines. The most important of these commands is **Bug**, which sends bug reports:

Bug

This command is used to send a bug report. Please include in the bug report all the information needed to reproduce the bug, along with the version numbers and site of your BIRD program.

```
BIRD> bug
Executing: /usr/ucb/mail -s "Bug report for BIRD" drogers@riacs.edu
Please type in the problem:
In BIRD version 3.0, on host brain.damaged.edu, the following
command sequence caused the program to crash:

Set default number-of-locations 999
Create memory test-memory 257 257
Output location-addr file.mem hex

Signed,
A. User
user@brain.damaged.edu
^D
EOT
Mail sent OK
```

The user is encouraged to send bug reports whenever any issue, big or small, arises concerning the BIRD program. We will try to quickly fix and send patches for any bugs we find in the BIRD program sources.

We also appreciate solutions to bugs found by more advanced users who find problems inside the source code. The more sharing of problems and solutions we do, the better this public-domain program will be!

The BIRD program also contains methods for executing UNIX programs, or getting to the UNIX shell:

Exec Program <*string*>

This command tries to execute the given program name. When completed, the user is returned to the BIRD program.

```
BIRD> Exec Program date
Wed Jan  3 16:10:13 PST 1990
```

Exec UNIX

This command starts off a UNIX shell and leaves the user there. Upon exiting that shell, the user is returned to the BIRD program.

8.7 Plotting information

This section not yet completed.

8.8 Genetic recombination

This section not yet completed.

APPENDIX A: Alphabetical list of BIRD commands

Abort

The **Abort** command is used to abort out of a block of input from **Define, Do, DoTimes, If,** and **While,** without executing any commands. It is for the occurrence that you change your mind after initiating a block of input.

```
BIRD> DoTimes 1000
DOTIMES> Address Z    ; oops... changed my mind here...
DOTIMES> Abort
?No commands to execute in DOTIMES statement
```

Address <*address*> {**Verbosely** | **Silently**}

The **Address** command takes two arguments: an address, and an optional keyword. The address is mandatory, and changes the state of the memory. It selects a number of memory locations, which are then active for future **Read** and **Write** operations.

The second argument to **Address** is optional; it determines whether the command will print out helpful information about the addressing operation when completed. The default is for that information to be given.

```
BIRD> Address 1010-
 Selected 19 locations, radius used was 111
```

Assign [**Address** | **Array** | **Variable**]

This command is used to assign the value of predefined objects: address tags, address arrays, or integer variables. See the specific help for the given type of assignment for more details.

Assign **Address** <*destination-address*> [<*address-op*> | <*address*>]

This command assigns the destination address to the value of the source address or the given operation.

For help on the available address operation, type **Help On-Topic Address-Op.**

For example, we could assign the value of the previously created address TEST-X by giving the command:

```
BIRD> Assign Address Test-X 1010-
```

We could also assign the value of the address using an address operation:

```
BIRD> Assign Address Test-X XOR Test-Y Test-Z
```

In this case, Test-X is assign to a value which is the XOR of the addresses Test-Y and Test-Z.

You can delete a created address using the **Delete Address** command.

Assign Array *<address-array>* *<source-address-array>* {*<start>* | *<variable>*} {* | *<end>* | *<variable>*}

This command is used to assign a block of addresses from one array to another. The command tries to copy as many addresses as it can; the user can limit the range of addresses in the source array by giving a start and end index.

For example, if we had two arrays, an internal array named INTERNAL and a file array named TEST, we could copy from the file array to the internal array:

```
BIRD> Assign Array INTERNAL TEST
 Copied 1024 addresses
```

It is also possible to assign only a range of addresses:

```
BIRD> Assign Array INTERNAL TEST 100 199
 Copied 100 addresses
```

Finally, we can start the block copy into a place inside the destination array, by giving the index of the starting position in the destination array name:

```
BIRD> Assign Array INTERNAL100 TEST
 Copied 924 addresses
```

In this case, the first address of TEST was copied into the 100th address of INTERNAL, and so on.

Assign Variable *<variable>* [*<function>* | *<variable>* | *<number>*]

This command assigns the variable to the value of the given number, variable, or function.

For help on the available functions, type **Help On-Topic Functions.**

For example, we could assign the value of the variable "$1" by typing the command:

```
BIRD> Assign Variable $1 333
```

Or we could assign the value of the variable to a function result:

```
BIRD> Assign Variable $1 Plus $2 $3
```

You can delete a created variable using the **Delete Variable** command.

Bug

This command is used to send a bug report. Please include in the bug report all the information needed to reproduce the bug, along with the version number and site of your BIRD program.

```
BIRD> bug
Executing: /usr/ucb/mail -s "Bug report for BIRD" drogers@riacs.edu
Please type in the problem:
In BIRD version 3.0, on host brain.damaged.edu, the following
command sequence caused the program to crash:

Set default number-of-locations 999
Create memory test-memory 257 257
Output location-addr file.mem hex

Signed,
A. User
user@brain.damaged.edu
^D
EOT
Mail sent OK
```

Clear [Addresses | Data | Folds | History | Location]

This command clears things. **Clear Addresses** resets the address in the memory. **Clear Data** clears out all data written to the memory. **Clear Folds** is not yet implemented. **Clear History** clears out the history list in a memory. See the specific help for the given clear command for more details.

Clear Addresses

This command restores the addresses of the current memory, erasing any changes that occurred since creation.

Clear Data

This clears the current memory of all data, this is, makes it appear that the memory never had anything written into it.

Clear Folds

(Not yet implemented in BIRD 3.0).

Clear History

This command clears the history list of the given memory.

Clear Location [*<variable>* | *<location-index>*]

This command clears the given location, that is, makes it appear as if it was never written to.

Create [Address | Alias | Array | Memory | State-File | Variable]

This command is used to create objects in the BIRD environment, which can then be manipulated by other BIRD commands. See the specific help for the given type of creation for more details.

Create Address *<name>* [*<address-size>* | *<variable>*]
{**Bit-Addresses** | **Float-Addresses**}

This command creates an address tag of the size and type. In the future, you can use the tag name in any command that requires an address. For example, we could create an address of size 256 bits, assign it to a random sequence of bits, and then use it to address the memory:

```
BIRD> Create Address TEMP 256 Bit
BIRD> Assign Address TEMP Z
BIRD> Address TEMP
 Selected 19 locations, radius used was 111
```

Create Alias *<alias-name>* *<address-array>* {*<start>* | *<variable>*}
{* | *<end>* | *<variable>*}

This command is used to create new names for previously defined address arrays. The alias array can be of smaller size than the original array, and can "capture" the current ordering of the source address array.

For example, we may have a script that uses the array ADDR to denote memory addresses. If we have a currently defined array TEST that we wish to use in the script, we could create ADDR as an alias of TEST:

```
BIRD> Create Alias ADDR TEST
```

We could also have created ADDR to be some subset of TEST:

```
BIRD> Create Alias ADDR TEST 1 100
```

Array aliasing has another feature, which is that is captures the current ordering of a source array. For example, assume that we had randomly ordered TEST, and then created the alias

ADDR, then again reordered, and created an alias :

```
BIRD> Reorder TEST Randomly
BIRD> Create Alias ADDR-A TEST
BIRD> Reorder TEST Randomly
BIRD> Create Alias ADDR-B TEST
```

In this case, the arrays ADDR-A and ADDR-B will contain the same addresses, but in different orderings.

The **Reorder** command can also be used to reorder alias arrays; restoring their order to their **Original-Order** restores the order to the order at the time of their creation.

Aliases can be deleted with the **Delete Array** command.

Create Array *<array-name> <oldfile>*
Create Array *<array-name>* [*<array-length>* | *<variable>*] [*<address-size>* | *<variable>*]
{**Bit-Addresses** | **Float-Addresses**}

This first command creates an array using the addresses in the given file. The addresses are not read into the memory; instead, a list of pointers to the addresses is kept, and the addresses are read in as needed. This allows the system to deal with very large databases of addresses, at a price of some speed.

The second command create an array in memory with the given number of addresses in it. While more memory-consumptive, this array is faster to access; for arrays that will be used a lot, you may wish to transfer them into an internal array by copying using the **Assign Array** command.

For an example, assume that we have a file of addresses called test.addr. We can now create an array from this file with the command:

```
BIRD> Create Array TEST test.addr
 Array TEST contains 1024 addresses
```

We can now print out (or otherwise use) any member of this array:

```
BIRD> Print test222
 Address:
 1110011111101010 1010000110001101 1001111101011010 1011000000110001
 1100101010100111 0001001100110000 1010000011010111 0110111010111111
 1010001111000111 1100100100111011 1101011100100110 0111100111100001
 0011001100111111 0001000000000111 1100011110110110 1001010000010010

 Bit count: 1:133  0:123
```

We could create an internal array of the same size, and copy the file array item-by-item into

it:

```
BIRD> Create Array INTERNAL 1024 256
 Array internal contains 1024 addresses
BIRD> Do $1 1 1024
DO> Assign Address INTERNAL$1 TEST$1
DO> End
```

(Though this copying is done easier using the **Assign Array** command.)

Create Memory *<name>* {*<address-size>*} {*<data-size>*}

This command creates new memory. The memory is assigned the name *<name>*. The address size and data size arguments are optional; if not given, the default address size and data size are used.

For example, we can create a new memory with the command:

```
BIRD> Create Memory Tweety 256 256
 Memory Tweety, 1024 locations, size 256/256
```

If the defaults were already set to an address size of 256 and a data size of 256, we could have typed to get the same result:

```
BIRD> Create Memory Tweety
 Memory Tweety, 1024 locations, size 256/256
```

If we try to create two memories with the same names, the program signals an error and refuses to create the new memory:

```
BIRD> Create Memory Tweety
 Memory Tweety, 1024 locations, size 256/256
BIRD> Create Memory Tweety
?Memory Tweety already exists
```

Creation of a memory automatically creates a new address array. The address array is given the name of the memory if possible. Thus, a memory created with the name ARF allows us to reference the tenth location address with the address tag "ARF10".

The location addresses of the current active memory can be referenced using the tag "L"; thus, the tenth location addresses can be referenced with "L10".

Create State-File

(This command is not yet functional in BIRD 3.0.)

Create Variable *<variable>* [*<function>* | *<variable>* | *<number>*]

This command creates a new variable of the given name. For example, we could rewrite the above example to be more descriptive:

```
BIRD> Create Variable $Radius 112
BIRD> Set Memory Radius $Radius
BIRD> Address 1010-
Selected 24 locations, radius used was 112
```

Define *<name>* [**Variable** *<variable>*] [**Variable** *<variable>*] [**Variable** *<variable>*] [**Variable** *<variable>*] [**Variable** *<variable>*] [**Variable** *<variable>*] [**Variable** *<variable>*] [**Variable** *<variable>*]

This command is used to define new top-level commands. The arguments allowed are currently all variables; when the command is given by the user, the user's arguments are bound to these variables, then the command is executed.

It is usually more efficient to define a command rather than to constantly use the **Input File** command.

For example, we could create the **Fill** command, which writes a given number of random data patterns into the memory at random locations:

```
BIRD> Define Fill Variable $How-Many
DEFINE> Dotimes $How-Many
DOTIMES2> Address Z Silently
DOTIMES2> Write Z
DOTIMES2> End
DEFINE> End
```

At this point, the **Fill** command is defined. We can now fill the memory with a given number of patterns with:

```
BIRD> Fill 100
```

You can see all command definitions using the **Show All Definitions** command. You can see a specific definition using the **Show Definition** command.

Delete [Address | Array | Definition | Memory | State-ID | Variable]

This command is used to delete objects created by the **Create** or **Define** command. See the specific help for the given type of deletion for more details.

Delete Address *<address>*

This command deletes the given address from the system. The address must have been previously created using the **Create Address** command.

Delete Array *<address-array>*

This command deletes the given address array from the system. The address array must have been previously created using the **Create Array** or **Create Alias** commands.

Delete Definition *<defined-command>*

This command deletes the given defined command from the system. The command must have been previously created using the **Define** command.

Delete Memory *<name>*

This command deletes the given memory name and reclaims the resources used by that memory.

(This command is not yet functional in BIRD 3.0.)

Delete State-ID [*<ID-number>* | *<variable>*] *<oldfile>*

(This command is not yet functional in BIRD 3.0.)

Delete Variable *<variable>*

This command deletes the given variable name from its table of variables. Variables are fairly inexpensive to keep around, so you shouldn't need to worry about this command unless you are used to cleaning up after yourself.

(Some system-created variables are undeletable; the system will signal an error and refuse to delete them if you try.)

Do *<do-variable>* {*<start>* | *<variable>*} {*<end>* | *<variable>*} {*<step>* | *<variable>*}

A more complex iteration command is the **Do** command. It repeatedly executes a block of commands, but also assigns a variable to a new value for each iteration.

The do variable is the name of a currently existing variable. The next argument is the starting value of the do variable; if not given, this defaults to "1". The next argument is the final value of the do variable; when the iteration results in a variable value over this number, the command finishes. This value defaults to "1" if not given. The final argument is the step size, which is the amount by which the do variable is incremented each step. The value defaults to "1" if not given.

For example, the following commands print a text string and the iteration number:

```
BIRD> Do $1 1 3
DO> Echo "ARF: " $1
DO> End
ARF: 1
ARF: 2
ARF: 3
```

The **Do** command is useful for manipulating address arrays. If an array name ends with the name of an integer variable, the variable is replaced by its value. For example, we could write a memories own location addresses to itself:

```
BIRD> Create Variable $NOL Array-Size L
BIRD> Do $1 1 $NOL
DO> Address L$1
DO> Write L$1
DO> End
[Print out of each addressing operation follows]
```

In this sequence, "L$1" is replaced each step by the value of "$1"; for example, if the value of "$1" is 123, then "L$1" is equivalent to L123, which is the address of the 123rd memory location.

DoTimes [*<number>* | *<variable>*]

The is the simplest iteration command; it simply repeats the execution of a given block of commands a given number of times:

```
BIRD> Dotimes 3
DOTIMES> Echo "ARF!"
DOTIMES> End
ARF!
ARF!
ARF!
```

First, the **DoTimes** command is entered along with the number of iterations; the user is placed into command-input mode. This mode change is made apparent to the user by a change of prompt to DOTIMES>. Next, the user inputs a command sequence, in this case, the single **Echo** command, then terminates the input with an **End** command. At this point the command is executed the given number of times.

Echo {**To-TTY** | **No-CR** | *<string>*} {*<string>*} {*<string>*} {*<string>*} {*<string>*} {*<string>*} {*<string>*} {*<string>*} {*<string>*}

This command echoes the given series of strings to the current echo device, followed by a carriage return. By default, the echo goes to the terminal; however, it can be reset using the **Out-**

put Echo and **Push Output Echo** commands.

If the string is a variable name, then the value of that variable is printed rather than the string. For example:

```
BIRD> Do $1 1 3
DO> Echo "ARF: " $1
DO> End
ARF: 1
ARF: 2
ARF: 3
```

If the first argument of **Echo** is **To-TTY**, then the echo is done to the TTY no matter what echo device is set. If the first argument of **Echo** is **No-CR**, then the echo is done to the TTY without a carriage return.

Echo is commonly used in script files to let the user know what is happening.

Else

The else command is only legal inside of an **If** command block. For example:

```
BIRD> If Same R 1010-
IF> Echo "The addresses are the same"
IF> Else
IF> Echo "The addresses are not the same"
IF> End
```

This sequence of command will print out a message every time it is executed, but the message printed will vary depending on whether the addresses are the same or not.

End

The **End** command is used to end a block of input from **Define, Do, DoTimes, If,** and **While.** It is also used to end a command level initiated with the **Exec BIRD** command.

Exec [**BIRD | Program | Restart | Unix**]

This command is used to execute new system command levels, UNIX programs or shells, or to restart the program. See the specific help for the given type of **Exec** command for more details.

Exec **BIRD**

This command executes a new command level of the BIRD executive. This is most useful in command files, where you wish to allow the user a chance to interact with the program before

continuing with the command script.

Exec Program <*string*>

This command tries to execute the given program name. When completed, the user is returned to the BIRD program.

```
BIRD> Exec Program date
Wed Jan  3 16:10:13 PST 1990
```

Exec Restart

This command restarts the BIRD program from scratch. All current state is lost.

Exec UNIX

This command starts off a UNIX shell and leaves the user there. Upon exiting that shell, the user is returned to the BIRD program.

Genetic {**No-Address** | <*address*>}

This command attempts to do one genetic recombination of the location addresses in the current memory. Still pretty experimental.

Help {**On-Topic** <*topic-name*> | <*defined-command*>}

The **Help** command is used to access on-line help documents.

For most commands, you simply type the keyword sequence to get the help document. For example, to get help on the **Set Status** command, type;

```
BIRD> Help set status
 [Lots of help.]
```

The **On-Topic** keyword is used to access help files that are not directory associated with a given command.

If [<*function*> | <*variable*> | <*number*>]

This command is used to conditionalize a sequence of commands. "True" is defined as a non-zero value, and "false" a zero value. Thus, "**If** `1`" always executes its block of commands, and "**If** `0`" never executes its block of commands.

For an example of the use of this command, we may wish to print out a message whenever the read address (kept in the address tag "R") and the written address:

```
BIRD> If Not Same R 1010-
IF> Echo "The addresses are not the same"
IF> End
```

It is also possible to use the **Else** command to separate the block of commands into two sections, the first of which is executed if the condition is true, and the second executed if the condition is false. For example:

```
BIRD> If Same R 1010-
IF> Echo "The addresses are the same"
IF> Else
IF> Echo "The addresses are not the same"
IF> End
```

This sequence of command will print out a message every time it is executed, but the message printed will vary depending on whether the addresses are the same or not.

Initialize Memory {**Old-Addresses** | **New-Addresses** | **Using-Address-Array** *<array>* | **From-Memory** *<memory-name>*}

This command initializes the current memory. If we specify **Old-Addresses**, the current location addresses in the memory are not changed. If we specify **New-Addresses**, the memory is given new random location addresses. If we specify **Using-Address-Array**, the addresses are set from the addresses in the given address array. If we specify **From-Memory**, the addresses are taken from the addresses of the given memory.

Input [**Address** | **File** | **Memory** | **State-Info**]

This command is used to input objects in the BIRD environment from external files. See the specific help for the given input command for more details.

Input Address *<address>* *<oldfile>*

This command inputs a single address from a file. The address file can be created using the **Output Address** command.

Input File *<oldfile>* {**Verbosely** | **Silently**}

This command assumes that the given file name is a file of BIRD commands. It opens the file and executes the commands until it reaches end of file.

The default is to show you the commands as they are executed. If you specify the second argument to **Silently**, the commands are not shown, though any typing that the commands do is still printed to the terminal.

Input Memory *<oldfile>*

No name is needed; the memory is given the original name it had when saved. If that name conflicts with a memory already in use, the string "New-" is prepended to it. For example, if we had previously saved a memory named "Tweety" in file Tweety.memory, we could type:

```
BIRD> Input Memory Tweety.memory
[Addresses: 10% 20% 30% 40% 50% 60% 70% 80% 90% 100%]
[Counters: 10% 20% 30% 40% 50% 60% 70% 80% 90% 100%]
Memory New-Tweety, 0 writes, created Fri Dec 29 12:44:07 1989
```

Input State-Info [*<state-ID>* | *<variable>*]

(Not yet fully implemented.)

Mutate [Location-Addresses]

This command currently has only one argument, which must be **Location-Addresses**. It is used to change a given number of bits in each location address.

Mutate Location-Addresses [*<#-of-bits>* | *<variable>*]

This command causes the given number of bits to be changed in each location address.

Output [Address | Alias-Mapping | Array | Array-Mapping | Command-Echo | Command-Record | Echo | Location-Addresses | Memory | Session]

This command is used to output objects from the BIRD environment to external files. See the specific help for the given output command for more details.

Output Address *<address>* *<newfile>* {**Binary** | **Hex**}

This command outputs a single address to a file in the given format. It can later be read back into the system using the **Input Address** command. Default is **Hex**.

Output Alias-Mapping *<alias-array>* *<newfile>*

This command outputs the currently active alias mapping of the array to the given file. This mapping is set up using the **Create Alias** command. This alias mapping is a copy of the mapping in force on the source array at the time the alias is created.

Output Array *<address-array>* *<newfile>* {**Binary** | **Hex**}

This command outputs a address array to a file in the given format. It can later be read back into the system using the **Create Array** command. Default is **Hex**.

Output Array-Mapping *<address-array>* *<newfile>*

This command outputs the currently active mapping of the array to the given file. This mapping is set up using the **Reorder** command.

Output Command-Echo *<newfile>*

This command redirects the command-echo output to the given file name. Command prompts and your typing will now go into this file rather than to the TTY. However, command results will still be printed to the TTY.

Output Command-Record *<newfile>*

This command redirects a record of all executed commands to the given file name.

Output Echo *<newfile>*

This command redirects the echo output file to the given file name. Future output of the **Echo** command will go to this file.

Output Locations-Addresses *<newfile>* **{Binary | Hex}**

This command outputs all of the location addresses of the currently active memory to the given file. They can later be restored in a memory using the command sequence **Create Array** followed by **Initialize Memory Using-Address-Array**. Default is **Hex.**

```
BIRD> Output Location-Addresses LOCS Binary

[Then, much later...]
BIRD> Create Array locs LOCS
 Array locs contains 1024 addresses
BIRD> Initialize Memory Using locs
```

Output Memory *<newfile>* **{Binary | Hex}**

This command takes the current memory and outputs it to the given filename. That memory file can later be restored using the **Input Memory** command. Default is **Hex.**

The memory file itself is an ASCII file that can be transferred from machine to machine. While bulky, this is intentional: the desire was to allow a memory to be created using one simulator, and then loaded into another simulator, which may be residing on a different computer. A "transfer" of a memory currently in BIRD to another simulator can be done by saving the memory, setting a new default hardware type, and reading the memory back in:

```
BIRD> Output Memory Test1.memory
BIRD> Set Default Hardware Connection-Machine
BIRD> Input Memory Test1.memory
```

```
[Addresses: 10% 20% 30% 40% 50% 60% 70% 80% 90% 100%]
[Counters: 10% 20% 30% 40% 50% 60% 70% 80% 90% 100%]
Memory New-Test1, 333 writes, created Fri Dec 29 12:44:07 1989
BIRD> Show All Memories
Memory New-Test1, 8192 locations, size 256/1, on Connection Machine
Memory Test1, 8192 locations, size 256/1
```

In this way, a memory can be "copied".

Output Session *<newfile>*

This command redirects the session output file to the given file name. Future output will go to that file.

The session output contains copies of the commands executed and the results of those executions. The results of the execution are not printed to the terminal.

For example, you can get the copy of a given command using the sequence:

```
BIRD> Output Session STATUS
BIRD> Show Status
BIRD> Output Session TTY
```

At this point, the file STATUS contains the following:

```
BIRD> Show Status
Set Command-Timing False
Set Print-Text-With-Addresses False
Set Random Seed 631408685
Set Random Address-Seed 333
Set Random Dont-Care-Bits-On 0
Set Random Address-Bits-On Random
Set Weather-Expansion 1
BIRD> Pop Output Session
```

Plot [Address | Array | Data-File | Memory]

This command tries to create a plot file for the given object. This plot file may be displayed or saved for later use. See the specific help for the given plot command for more details.

Plot Address *<address>* *<address-array>* **{Distances | Integrated-Distances | Both}**

This command plots the distribution of distances between the given address and the addresses in the given array. You can plot the distance distribution, the integration of the distance distribution, or both. It creates the files Address-info.data and Address-info.plot. If possible, it displays the created plot file. Example:

```
BIRD> Plot Address 1010- W Distances   ; This plots the distance from
                                       ; 10101010... to each weather
                                       ;address
```

Plot Array *<array-1>* *<array-2>* **{Distances | Integrated-Distances | Both}**

This command plots the distribution of distances between the address and the addresses in the given array. You can plot the distance distribution, the integration of the distance distribution, or both. It creates the files Array-info.data and Array-info.plot. If possible, it displays the created plot file. Example:

```
BIRD> Create Alias WA W 1 100      ; Create 100 element array WA
BIRD> Create Alias WB W 101 200    ; Create 100 element array WB
BIRD> Plot Array WA WB Distances   ; This plots the distance distribution
                                   ; between elements of WA and WB
```

Plot Data-File *<oldfile>*

This command creates a plot file from the given data file. It does NOT try to plot the generated plot file.

Plot Memory [Distances | Location-Weights | Number-Of-Writes]

This command tries to create a plot file for the given memory object. This plot file may be displayed or saved for later use. See the specific help for the given command for more details.

Plot Memory Distances {Distribution | Integrated-Distribution | Both}

This command plots the distribution of distances between the last **Address** operation and the memory locations. You can plot the distance distribution, the integration of the distance distribution, or both. It creates the files Memory.dists.data and Memory.dists.plot. If possible, it displays the created plot file.

Plot Memory Location-Weights {Distribution | Integrated-Distribution | Both}

This command plots the distribution of average location weights for each memory location. The "weight" is a measure of how predictive the location is. You can plot the distribution, the integration of the distribution, or both. It creates the files Memory.loc-weights.data and Memory.-loc-weights.plot. If possible, it displays the created plot file.

Plot Memory Number-Of-Writes {Distribution | Integrated-Distribution | Both}

This command plots the distribution of number of writes to each memory location. You can plot the distribution, the integration of the distribution, or both. It creates the files Memory.-num-writes.data and Memory.num-writes.plot. If possible, it displays the created plot file.

Pop [Output | Variables]

This command undoes the effect of a previously-given **Push** command. See the specific help for the given command for more details.

Pop Output [Command-Echo | Command-Record | Echo | Session]

This command undoes the effect of a previously-given **Push** command. See the specific help for the given command for more details.

Pop Output Command-Echo

This command restores the state of the command echo to that which was in place before the last **Push Output Command-Echo** command.

Pop Output Command-Record

This command restores the state of the command recording to that which was in place before the last **Push Output Command-Record** command.

Pop Output Echo

This command restores the state of the echo to that which was in place before the last **Push Output Echo** command.

Pop Output Session

This command restores the state of the session recording to that which was in place before the last **Push Output Session** command.

Pop Variables {*<variable>*} {*<variable>*} {*<variable>*} {*<variable>*} {*<variable>*} {*<variable>*} {*<variable>*} {*<variable>*} {*<variable>*}

The **Pop Variable** command restores the original value of a variable before a previous **Push Variable** command. The **Pop Variable** command can take up to 9 variables to pop.

Variable pushing and popping is mostly used in script files to avoid smashing the values of variable between script files.

Print [*<address-op>* | *<address>*]

This command takes one argument, which is the address to print. The address is printed along with the a count of the number of bits of each type. For example, we could print the address of location 1:

```
BIRD> Print L1
Address:
1011010001111110 0110101000001101 1011101010110000 1000110001011011
0001010101010000 1110001111011111 0011111001111101 0010100000000110
0011100010001110 1111111001011011 1000011011101011 0110000100110110
1001001111101101 0111010110010000 1111100101110001 1000100000010000

Bit count: 1:130 0:126
```

Push [Output | Variable]

This command is used to push the level of some object; a following **Pop** command restores the state. See the specific help for the given command for more details.

Push Output [Command-Echo | Command-Record | Echo | Session]

This command is used to push the level of an output device; a following **Pop** command restores the output device to its previous state. See the specific help for the given command for more details.

Push Output Command-Echo [*<newfile>* | **Error-Output | TTY | Null**]

This command pushes the command-echo output to the given file name. Command prompts and your typing will now go into this file rather than to the TTY. However, command results will still be printed to the TTY.

Push Output Command-Record [*<newfile>* | **Error-Output | TTY | Null**]

This command pushes a record of all executed commands to the given file name.

Push Output Echo [*<newfile>* | **Error-Output | TTY | Null**]

This command pushes the echo output file to the given file name. Future output of the **Echo** command will go to this file.

Push Output Session [*<newfile>* | **Error-Output | TTY | Null**]

This command pushes the session output file to the given file name. Future output will go to that file.

The session output contains copies of the commands executed and the results of those executions. The results of the execution are not printed to the terminal.

For example, you can get the copy of a given command using the sequence:

```
BIRD> Push Output Session STATUS
```

```
BIRD> Show Status
BIRD> Pop Output Session
```

At this point, the file STATUS contains the following:

```
BIRD> Show Status
Set Command-Timing False
Set Print-Text-With-Addresses False
Set Random Seed 631408685
Set Random Address-Seed 333
Set Random Dont-Care-Bits-On 0
Set Random Address-Bits-On Random
Set Weather-Expansion 1
BIRD> Pop Output Session
```

Push Variable *<variable>* [*<function>* | *<variable>* | *<number>*]

The **Push Variable** command pushes a new value onto the given variable name; future references to that variable will use that new variable. However, the old value is retained, and is restored when the **Pop Variable** command is given. This command is useful when you wish to use a variable, but are not sure if other routines have already assigned its value.

Variable pushing and popping is mostly used in script files to avoid smashing the values of variable between script files.

Quit {**Program** | **Input-File** | **Connection-Machine**}

This command is used to quit, either the BIRD program or just input from a given input file.

The special command **Quit Connection-Machine** is only useful when using the Connection Machine version of the simulator; it runs the system program "cmdetach" to detach the Connection Machine from the user.

Read {*<address>*} {**Verbosely** | **Silently**}

The **Read** command takes two arguments: an optional address, and an optional keyword. This command reads the memory; if no second argument is given, the command prints the read address. If a second argument is given, the read address is compared against that address and the distance between them is printed.

The second argument to **Read** is optional; it determines whether the command will print out distance information when the read operation is completed. The default is for that information to be given.

```
BIRD> Create Memory Test                    ; Memory named "Test"
 Memory Test, 1024 locations, size 256/256
```

```
BIRD> Address 1010-                          ; Address with 1010...
 Selected 21 locations, radius used was 111
BIRD> Write 11110000-                        ; Write
BIRD> Read                                   ; See if it's there
 Read data
 1111000011110000 1111000011110000 1111000011110000 1111000011110000
 1111000011110000 1111000011110000 1111000011110000 1111000011110000
 1111000011110000 1111000011110000 1111000011110000 1111000011110000
 1111000011110000 1111000011110000 1111000011110000 1111000011110000
```

It is sometimes difficult to manually study the read data pattern for errors; in this case, you can use the first argument to **Read** to compute the distance between the read address and any given address:

```
BIRD> Read 11110000-                         ; Compare
 Distance is 0
BIRD> Read 11111111-                         ; Compare against 1's
 Distance is 128
```

The **Read** command, when given with an address argument, sets the variable $DISTANCE to the value of the distance between the read address and the argument address.

Reorder <*array-name*> [**Offset** [<*number*> | <*variable*>] | **Original-Order** | **Randomly** | **Reverse** | **Same-Order-As** <*array-name*> | **Using-Map-File** <*oldfile*>]

The **Reorder** command is used to reorder address arrays.

For example, if we had a array defined from a file that we wish to write to memory, we would want to reorder the array before retraining, so that the training would not depend on the order of presentation.

The following are legal **Reorder** subcommands:

Reorder <*array*> **Original-Order**
This command removes any previous reorder commands and restores the array to its original order.

Reorder <*array*> **Reverse**
This command removes any previous reorder commands and orders the array in reverse order, that is, the last element is 1, the second to the last is 2, etc.

Reorder <*array*> **Offset** [<*number*> | <*variable*>]
This command offsets the current ordering by the given number, that is, the pointer that pointed to element *n* now points to element *n* + *offset*. The pointer wraps back to the beginning of the array when the new pointer points off the end of the array.

Reorder *<array>* **Same-Order-As** *<source-array>*

This command removes any previous reorder commands and orders the array in the same ordering as the source array.

Reorder *<array>* **Using-Map-File** *<oldfile>*

This command removes any previous reorder commands and orders the array in the order stored in the map file. The map file was previously generated using the **Output Mapping** command.

Set [Active-Memory | Default | Directory | Erase-Weather-Codes | Genetic | Location-Info | Memory | Plot | Print-Text-With-Addresses | Random | State-Info | Storage-Checking | System | Weather-Expansion]

This command is used to set the parameters of the BIRD system. See the specific help for the given command for more details.

Set Active-Memory *<memory-name>*

For example, the following sequence of commands creates two memories, then activates each of them in turn and does an addressing and write operation:

```
BIRD> Create Memory Test-A 256 256
 Memory Test-A, 1000 locations, size 256/256
BIRD> Create Memory Test-B 256 256
 Memory Test-B, 1000 locations, size 256/256
BIRD> Set Active-Memory Test-A
BIRD> Address 1100-              ; Address at 11001100...
 Selected 19 locations, radius used was 111
BIRD> Write 11110000-            ; Write 1111000011110000...
BIRD> Set Active-Memory Test-B
BIRD> Address 1100-
 Selected 18 locations, radius used was 111
BIRD> Write 11110000-            ; Write the same as for Test-A
```

It is possible to create the different memories on different hardware devices; thus, you could create one memory on the Connection Machine, and one memory on the front-end computer using the software simulator, and compare their results.

Set Default [Address-Size | Address-Type | Area | Counter-Type | Data-Size | Dont-Care-Bits | Folds | Full-Info | Hardware | History-Delay | History-Size | Ignore-Exact-Matches | Input-Mask | Number-Of-Locations | Output-Type | Radius | Random-Seed |Reference-Type | Threshold | Type-Of-Addressing]

This command is used to set the parameters for future **Create Memory** commands. See the specific help for the given command for more details.

Set Default Address-Size [*<number>* | *<variable>*]

This sets the default address size for future memories. The address size is the number of bits used for addressing the memory. The initial value for this default is 256.

Set Default Address-Type [Bit-Addresses | Float-Addresses]

This command sets the default address type for memories. Currently, only **Bit-Addresses** are fully functional, though the software simulator has some functions that allow **Float-Addresses**.

Set Default Area [*<number>* | *<variable>*]

When area-based addressing is used, this command sets the default desired area, that is, the desired number of selected locations, for future memories. The initial value for this default is 15.

Set Default Calculate-Weights [True | False]

My work on the statistics of data counters allows a value called a *weight* to be associated with each data counter. The weight is a measure of how predictive the data counter is. Depending on this weight, a given counter may "vote" for a "1" bit value (if it is predicting above the mean) or a "0" bit value (if it is predicting below the mean).

This command is currently only functional using the software simulator and the Connection Machine simulator.

Set Default Counter-Type [16 | 32 | 8] [Float | Integer]

This command sets the type of data counter we wish to have. Depending on the simulator, the chosen value may or may not be available. If not available, the system tries to use the "nearest" legal choice.

Set Default Data-Size [*<number>* | *<variable>*]

This sets the default data size for future memories. The data size is the number of bits used for reading from and writing to the memory. The initial value for this default is 256.

Set Default Dont-Care-Bits [Random | *<#-of-bits>* | *<variable>*]

This command sets the number of dont-care bits in the location addresses. If we give it a specific number, then that exact number of dont-care bits (randomly selected) is in each location address. If **Random**, then a random number of dont-care bits are placed in each address.

Set Default Folds [*<number>* | *<variable>*]

(Not implemented yet.)

Set Default Full-Info [True | False]

This tells the memory whether we wish to calculate full information concerning the write operation, specifically, the number of writes information.

This is only useful with the Stanford simulator.

Set Default Hardware [Software-Simulator | Connection-Machine | Stanford-Hardware]

This sets the default number of locations for future memories. The address size is the number of bits used for addressing the memory. The

For example we can set the defaults for memory creation to make a memory with 200 address bits, 10 data bits, and 8192 locations, and to use the Connection Machine:

```
BIRD> Set Default Address-Size 200
BIRD> Set Default Data-Size 10
BIRD> Set Default Number-Of-Locations 8192
BIRD> Set Default Hardware Connection-Machine
```

Set Default History-Delay [*<number>* | *<variable>*]

The history delay is used in conjunction with **Set Default History-Size** to set the number of steps we delay before using a previous address. For example, with a delay of 0 and a size of 3, the address is composed of the current address, the previous address, and the second previous address. With a delay of 1, the address is composed of the current address, the second previous address, and the fourth previous address.

If this seems too complex, then just ignore it. It's designed to make things easier for some experiments.

Set Default History-Size [*<number>* | *<variable>*]

The history size is the number of past addresses we wish to include into the full address. That is, the actual address used in the memory is a combination of the address used in the **Address** command with past addresses given to that command.

If this seems too complex, then just ignore it. It's designed to make things easier for some experiments.

Set Default Ignore-Exact-Matches [True | False]

There are occasions where we wish to use a location address of a memory for a reference address, but wish to keep that location from being selected. For example, the memory may be doing a nearest-neighbor search, and we do not wish for the test sample to be included in the response.

This command allows us to rule out from selection any location which has a distance from the reference address of **0.**

Set Default Input-Mask [*<address>* | **None**]

The input mask is a set of bits that we should ignore during the addressing operation. Normally, the hamming distance is calculated using all the bits in the sample address; the input mask says that some of the bits should be ignored, that is, always match during the hamming distance calculation. You will have to readjust the radius after you do this... the more bits are masked, the greater the number of addresses will be within a given radius. This command sets the given mask as the default mask for future memories.

Giving the keyword **None** as the argument unmasks all the bits. The initial value of this default is **None.**

Set Default Number-Of-Locations [*<number>* | *<variable>*]

This sets the default number of locations for future memories. The program will attempt to create future memories with that many hard memory locations.

Set Default Output-Type [Bit-Addresses | Float-Addresses]

(This command not yet functional).

Set Default Radius [*<number>* | *<variable>*]

When standard radius-based addressing is used, this command sets the default radius for future memories. The initial value for this default is 111.

Set Default Random-Seed [*<number>* | *<variable>*]

This sets the random seed that is used to generate the location addresses. This allows the user to create two memories with exactly the same location addresses.

Set Default Read-From [Array *<array-name> <start-index>*| **Counters** | **Location-Addresses]**

This command sets the default for where **Read** operations get their data.

Array uses the array as a table of addresses; at read-time, the selected locations are collected and returned as the output. The starting point for the array can be offset from the default value **1**; the array must be long enough to have an array address for each memory location.

Counters makes the standard model sparse distributed memory.

Location-Addresses uses the array of location addresses for the current memory as the read array.

Once the memory has been created, the type of reading cannot be changed.

Certain memory operations that refer specifically to counters cannot be used for memories that are not counter-based. For example, **Set Default Counter-Type** is not useful if the memory is not counter-based.

Set Default Reference-Type [Bit-Addresses | Float-Addresses]

This is the type of address that the user will be giving the memory. For example, it may be useful for the user to be able to give a memory with binary location addresses a floating address.

Not implemented in all simulators.

Set Default Scaled-Distances [True | False]

This command causes distances to be scaled before selection is done. It is used in combination with locally-scoped radii in order to make area-based addressing work.

Not implemented in all simulators.

Set Default Threshold [*<number>* | *<variable>* | Type]

The threshold of a memory is the number the memory uses to determine whether the data counter sum represents a "0" or a "1" in the read address. The initial value for this default is 0.

Set Default Type-Of-Addressing [Area | Radius]

The Kanerva model assumes that the location addresses are randomly placed. When this is true, statistics operate well so that any address contains roughly the same number of locations within a given radius.

When the location addresses are not randomly placed, this is no longer true, and a one given address may select many more locations than another address. Because of this breakdown, I proposed the use of *area-based addressing*. Specifically, area-based addressing selects the radius only once the address is given to the **Address** command; it selects the radius to get as close to a desired number selected locations as possible. The initial value for this default is **Radius**.

Set Directory *<name>*

This command sets the current working directory for the BIRD program.

Set Erase-Weather-Codes [True | False]

This flag tells whether the weather addresses contain information from the weather codes fields.

Set Genetic

This command is used to set parameters for the genetic system. Not yet fully documented.

Set Location-Info [*<location-index>* | *<variable>*]

This command assigns the values of a number of variables with location-related information. The variables set are:

$Number-of-writes
$Location-distance
$Location-selected
$Location-radius
$Location-birthdate
$Location-avg-vote
$Location-weight

Set Memory [Area | Calculate-Weights | Counter | Dont-Care-Bits | Full-Info | Ignore-Exact-Matches | Input-Mask | Location | Output-Type | Radius | Random-Seed | Reference-Type | Scaled-Distance | Scope-Of-Radius | Threshold | Type-Of-Addressing]

This command is used to set the parameters for the current active memory. See the specific help for the given command for more details.

Set Memory Area {Automatically | *<number>* | *<variable>*}

When area-based addressing is used, this command sets the desired area, that is, the desired number of selected locations. Experiments by Kanerva suggest that a reasonable number of locations is 1/2 * sqrt (number of locations); thus, for a 1000-location memory, an area of 1/2 * sqrt (1000) or 16 locations would be useful:

```
BIRD> Set Memory Area 16
```

Future **Address** commands will now use this area in calculating the set of selected locations, when the memory has been set to area-based addressing mode.

If the keyword **Automatically** is given, the program calculates what is considers a reasonable area, and uses that value.

Set Memory Calculate-Weights [True | False]

My work on the statistics of data counters allows a value called a *weight* to be associated with each data counter. The weight is a measure of how predictive the data counter is. Depending on this weight, a given counter may "vote" for a "1" bit value (if it is predicting above the mean)

or a "0" bit value (if it is predicting below the mean).

This command is currently only functional using the software simulator and the Connection Machine simulator.

Set Memory Counter [*<location-index>* | *<variable>*] [*<bit-index>* | *<variable>*] [*<value>* | *<variable>*]

This command sets the value of the given location counter.

Set Memory Dont-Care-Bits [Random | *<#-of-bits>* | *<variable>*]

This command sets the number of dont-care bits in the location addresses. If we give it a specific number, then that exact number of dont-care bits (randomly selected) is in each location address. If **Random**, then a random number of dont-care bits are placed in each address.

Set Memory Full-Info [True | False]

This tells the memory whether we wish to calculate full information concerning the write operation, specifically, the number of writes information.

This is only useful with the Stanford simulator.

Set Memory Ignore-Exact-Matches [True | False]

There are occasions where we wish to use a location address of a memory for a reference address, but wish to keep that location from being selected. For example, the memory may be doing a nearest-neighbor search, and we do not wish for the test sample to be included in the response. This command allows us to rule out from selection any location which has a distance from the reference address of **0**.

Set Memory Input-Mask [*<address>* | **None**]

The input mask is a set of bits that we should ignore during the addressing operation. Normally, the hamming distance is calculated using all the bits in the sample address; the input mask says that some of the bits should be ignored, that is, always match during the hamming distance calculation. You will have to readjust the radius after you do this... the more bits are masked, the greater the number of addresses will be within a given radius.

Giving the keyword **None** as the argument unmasks all the bits.

Set Memory Location [*<location-index>* | *<variable>*] *<address>*

This command sets the location address of the given memory location to the given address.

Set Memory Output-Type [Bit-Addresses | Float-Addresses]

(This command not yet functional).

Set Memory Radius {Automatically | *<number>* | *<variable>*}

When standard radius-based addressing is used, this command sets the radius. For example, we may decide that we wish to have a larger radius for our 256-bit address size memory than the standard radius of 111 bits; we could change it by:

```
BIRD> Set Memory Radius 114
```

Future **Address** commands will now use this new radius in calculating the set of selected locations.

If the keyword **Automatically** is given, the program calculates what is considers a reasonable radius, and uses that value.

Set Memory Random-Seed [*<number>* | *<variable>*]

This sets the random seed that is used to generate the location addresses. This allows the user to create two memories with exactly the same location addresses.

Set Memory Reference-Type [Bit-Addresses | Float-Addresses]

This is the type of address that the user will be giving the memory. For example, it may be useful for the user to be able to give a memory with binary location addresses a floating address.

Not implemented in all simulators.

Set Memory Scaled-Distances [True | False]

This command causes distances to be scaled before selection is done. It is used in combination with locally-scoped radii in order to make area-based addressing work.

Not implemented in all simulators.

Set Memory Scope-Of-Radius [Global | Local]

This command sets the scope of the radius in a memory. **Global** scope is standard; all locations use the same global radius in calculating whether they are selected. **Local** scope means that each location has its own radius.

Not implemented in all simulators.

Set Memory Threshold [*<number>* | *<variable>* | Type]
Set Memory Threshold Type [Average-Value | Number | Winner-Take-All | Zero]

The threshold of a memory is the number the memory uses to determine whether the data counter sum represents a "0" or a "1" in the read address. The default is zero; if the data sum is greater than or equal to zero, that bit is taken to be a "1", else it is taken to be a "0". Zero is a good threshold only if that bit is, on the average, "0" half the time and "1" half the time in the written data addresses.

There are some additional thresholding types that may be useful. Setting the threshold type to **Average-Value** tries to use the average input value of each bit in setting a threhold. Each bit has its own threshold.

Setting the threshold type to **Winner-Take-All** has the system set a threshold so that only one bit is left "1". This is useful only if we know that the data was in that format.

Setting the threshold type to **Number** simply uses the number given by the user as the threshold.

Setting the threshold type to **Zero** is the same as using type **Number** with a threshold of 0, but may be faster in some implementations.

Set Memory Type-Of-Addressing [Area | Radius]

The Kanerva model assumes that the location addresses are randomly placed. When this is true, statistics operate well so that any address contains roughly the same number of locations within a given radius.

When the location addresses are <u>not</u> randomly placed, this is no longer true, and a one given address may select many more locations than another address. Because of this breakdown, I proposed the use of *area-based addressing*. Specifically, area-based addressing selects the radius only once the address is given to the **Address** command; it selects the radius to get as close to a desired number selected locations as possible. The default type of addressing is **Radius**; to use area-based addressing, type:

```
BIRD> Set Memory Type-of-addressing Area
```

Set Plot [Bottom-Title | Left-Title | Top-Title | Display | Pixrect-Also | X-Dimensions | Y-Dimensions]

This command is used to set the parameters for the next plot-related command. See the specific help for the given command for more details.

Set Plot [Bottom-Title <*string*> | Left-Title <*string*> | Top-Title <*string*>]

This command sets the title strings along the borders of a plot.

Set Plot Display [True | False]

This command sets whether we wish to try to immediately display all plot files as they are created.

Set Plot Pixrect-Also [True | False]

This command tells the BIRD program whether to create SUN pixrect files whenever a plot file is created.

Set Plot X-Dimensions [Machine-Settable | <*start*> <*finish*>]

This command sets the X-dimensions for future plot calls. **Machine-Settable** allows the machine to select appropriate coordinates.

Set Plot Y-Dimensions [Machine-Settable | <*start*> <*finish*>]

This command sets the Y-dimensions for future plot calls. **Machine-Settable** allows the machine to select appropriate coordinates.

Set Print-Text-With-Addresses [True | False]

The **Print** command normally does not try to print an address as text. This switch causes the command to try to print out all addresses as text along with the standard printout.

Set Random [Address-Bits-On | Address-Seed | Dont-Care-Bits-On | Seed]

This command is used to set the parameters for system randomness. See the specific help for the given command for more details.

Set Random Address-Bits-On {Random | <*#-of-bits*> | <*variable*>}

This sets the number of address bits we wish to have "1" in the random "Z" addresses. **Random** means set a random number of "1" bits.

Set Random Address-Seed {Random | <*#-of-bits*> | <*variable*>}

This sets the random seed used to generate randomness in the "Z" addresses. This allows the user to set a seed to reproduce a sequence of random addresses.

Set Random Dont-Care-Bits-On {Random | <*#-of-bits*> | <*variable*>}

This sets the number of dont-care bits we wish to have on in the random "Z" addresses. **Random** means set a random number of dont-care bits on.

Set Random Seed {Random | <*#-of-bits*> | <*variable*>}

This sets the random seed for future calls to random. By setting to random seed to specific values, the user can exactly reproduce runs of the BIRD program.

Set State-Info [Min-Population-For-Breeding | Score | Source-Directory | Spontaneous-Generation | State-Directory]

This command is used to set the parameters for state info system. See the specific help for the given command for more details.

Set State-Info Min-Population-For-Breeding [*<number>* | *<variable>*]

System state info is still under development.

Set State-Info Score [*<state-ID>* | *<variable>*] [*<number>* | *<variable>*]

System state info is still under development.

Set State-Info Source-Directory *<directory-name>*

System state info is still under development.

Set State-Info Spontaneous-Generation [*<number>* | *<variable>*]

System state info is still under development.

Set State-Info State-Directory *<directory-name>*

System state info is still under development.

Set Storage-Checking [True | False]

This command is mostly for debugging routines that overwrite memory. At a cost of a few bytes per allocation, the system is capable of keeping track of all allocated storage, and checking it for overwriting. Normally, this is set **False**.

Set System [Check-For-Idleness | Command-Structure | Priority | Required-Idle-Time | Sleep-Period-If-Not-Idle | Time-Commands | Type]

This command is used to set the system parameters. See the specific help for the given command for more details.

Set System Check-For-Idleness [*<number>* | *<variable>*]

This command sets the number of seconds between checks for system idleness.

Set System Command-Structure [Full | Standard]

This sets the command structure for the system. The default is **Full**; this is the full set of BIRD commands. The **Standard** set of command is a proposed standard which has a much-reduced command set.

Set System Priority [*<number>* | *<variable>*]

This command is used to reduce the priority of the BIRD program. It is useful when one is doing a large run and wishes to avoid competing with other uses on the system.

The maximum one can reduce priority is 20; also, one cannot regain priority once one has given it up.

Set System Required-Idle-Time [*<number>* | *<variable>*]

This command sets the number of seconds we require the system be idle (of interactive jobs) before we allow the BIRD program to run.

Set System Sleep-Period-If-Not-Idle [*<number>* | *<variable>*]

This sets the number of seconds the program sleeps if the system is not idle.

Set System Time-Commands [**True** | **False**]

If you are interested in speed, this command prints out statistics about the time it took to execute the last command.

```
BIRD> Set System Time-Commands True
BIRD> DoTimes 100
DOTIMES> Address 1010- Silently
DOTIMES> End
 [Elapsed: 16 secs; Usage: 0.080000]
```

Set Version [**BIRD** | **Syntax**] *<major-version>* *<minor-version>*

This sets the version of the program or syntax used by the following input commands. If BIRD detects a possible incompatibility, it signals the user with a warning message.

This command is mostly useful in command files; by placing the version information at the head of command files, one is warned at execution time if the program has been changed in a potentially incompatible manner.

For the program, the major version number if incremented when a major update of the software is made. The minor version is incremented whenever any change is made to the software.

For the syntax, the major version number is incremented whenever the command syntax is

changed making a previously syntactically-valid command invalid. The minor version number is incremented whenever a new command is added to the command syntax.

```
BIRD> Set Version Program 3 120
?BIRD version mismatch: program 3.163, software 3.150
?Your software may not work properly in this version
```

Set Weather-Expansion [*<number>* | *<variable>*]

The default size for weather addresses is 256 bits. This command allows one to set a multiplicative factor by which to expand the weather address. For example, you could set the factor to "2", and all weather addresses will be of size 512.

Currently, this does nothing more interesting than duplicate bits. Eventually, I hope to have larger numbers give more discrimination among weather field values.

Show [**All** | **Array-Variance** | **Average-Weights** | **Best-And-Worst** |
Best-Distribution | **Better** | **Bit-Statistics** | **Candidates** | **Comparison** | **Counter-Vote** |
Crossover | **Defaults** | **Definition** | **Difference** | **Distance** | **Fields** |
Genetic | **Input-Sums** | **Location** | **Mapping** | **Memory** |
Output-Sums | **Plot** | **Radius-Estimate** | **Score** | **Selected** | **State-Info** |
Status | **Storage-Statistics** | **System** | **Time** | **Underwritten-Statistics** |
Variable | **Version** | **Weather-Decoding** | **Weights** | **Write-Statistics**]

This command shows information about things. See the specific help for the given command for more details.

Show All [**Addresses** | **Definitions** | **Memories**]

This command shows information about all items in the given class. See the specific help for the given command for more details.

Show All Addresses

This command shows all defined addresses, including system internals, user addresses, and user arrays.

```
BIRD> Show All Addresses
 Address WOOF, address size 256, bit address
 Array SAMPLES, 1000 addresses, address size 256, bit addresses
 Address CENT is a system internal
 Array Z is a system internal
 Array R is a system internal
  Array W is a system internal
 Array L is a system internal
 Array F is a system internal
```

Address A is a system internal
Address F is a system internal
Address M is a system internal
Address R is a system internal
Address W is a system internal
Address FCENT is a system internal
Address Z is a system internal

Show All Definitions

This command shows all commands defined with the **Define** command. For a more detailed description of an individual definition, use the **Show Definition** command.

```
BIRD> Show All Definitions
There is 1 command defined.
 Define ARF Variable $loops
```

Show All Memories

This command shows all currently defined memories. Example:

```
BIRD> Show all memories
 Memory Woof, 1024 locations, size 256/256
 Memory Arf, 8192 locations, size 256/256, on Connection Machine
```

Show All Variables

This command prints a list of all currently defined variables and their current value:

```
BIRD> Show All Variables
$NUMBER-OF-WRITES: 0, $TOTAL-NUMBER-OF-WRITES: 0,
$DISTANCE: 0, $GENETIC-DONE: 0, $LOCATION-SELECTED: 0,
$UNWEIGHTED-DISTANCE: 0, $LOCATION-DISTANCE: 0,
$GENETIC-FAILED: 0, $GENETIC-OK: 0, $LOCATION-AVG-VOTE: 0,
$BAD: 0, $GOOD2: 0, $GOOD1: 0, $LOCATION-RADIUS: 0,
$SELECT-COUNT: 24, $SDM-RADIUS: 111,
$PROGRAM-ID: 649180109, $LOCATION-BIRTHDATE: 0,
$LOCATION-WEIGHT: 0
```

Show Array-Variance *<array>* {*<start>* | *<variable>*} {* | *<start>* | *<variable>*}

This command shows statistics about the variance of the given address array. For example, we can get statistics about the variance of the first 100 weather addresses:

```
BIRD> Show Array-Variance W 1 100
 Average address distance from bit centroid: 36
```

Variance in address distance about average: 182 (stddev 13.490738)
Variance in address distance about zero: 1510 (stddev 38.858718)
Average bit distance from center: 0.140000
Variance of bit distance about average: 0.028272
Average bit variation about zero: 0.048528
Bit centroid:
1111111100000000 1111111100000000 1111111100000000 0001111111100000
0011111111110000 0111111110000000 0000111111111000 0000001111111100
0000001111111110 1110000000011111 0001111111110000 0000011111111100
0111111110000000 0111111110000000 0000000000000000 0000000000000000

Float centroid:
1111111100000000 1111111100000000 1111111100000000 !346891186532000
4455555555554444 4891111163000000 00!3568119875430 000!347111997630
0000027111119950 7764222222357777 !345789187653!!0 0!!3457198876542
46789991132!0000 !7788991932!!000 000000!!!!000000 0000000000000000

In the representation of the float centroid, the numbers are rounded to the nearest tenth, so that [0.0, 0.1, 0.2, ..., 0.9, 1.0] are mapped onto the characters [0, !, 2, ..., 9, 1].

The centroids are assigned to the system addresses CENT and FCENT.

Show Average-Weights [*<min-#-of-writes>* | *<variable>*]

For all the locations which have at least the given number of writes, this command returns the average weight of the 50 best, in the first bit column.

```
BIRD> Show Average-Weight 10
 Average weight for first bit of 50 best is: 0: 33  1: 24
```

Show Best-And-Worst [*<min-#-of-writes>* | *<variable>*]

This command shows the best and worst locations for predictiveness, given the restriction that all considered locations must have at least the minimum number of writes.

```
BIRD> Show Best-And-Worst 1
Best max scores: 11:10000 14:10000 38:10000 39:10000 51:10000
Best avg scores: 11:10000 14:10000 38:10000 39:10000 51:10000
Worst max scores: 11:10000 14:10000 38:10000 39:10000 51:10000
Worst avg scores: 549:4944 810:4944 820:4944 924:4944 971:4921
```

Show Best-Distribution *<newfile>* [*<min-#-of-writes>* | *<variable>*]

This command creates a file with the distribution of distances for the best 30 predictive locations. I'm not sure how well this currently works.

Show Better [*<min-#-of-writes>* | *<variable>*]

This command shows the best 30 locations for predictiveness of the first bit. It shows two tables: those that are best for predicting "1", and those that are best for predicting "0". It only considers locations which have at least the given minimum number of writes.

```
BIRD> Show Better 10
Best max scores for [1]:
39:212 62:200 650:177 806:132 290:104 192:87 939:73 951:73 37:61 865:50 576:50
465:26 903:24 608:18 114:17 204:17 466:10 44:6 180:5 860:3 880:3 659:2 870:2
95:2 557:2 203:0 8:0 0:0 0:0 0:0 0:0 0:0 0:0 0:0 0:0 0:0 0:0 0:0 0:0 0:0 0:0
0:0 0:0 0:0 0:0 0:0 0:0 0:0 0:0 0:0
Best max scores for [0]:
537:664 381:111 432:98 528:87 464:87 69:73 284:73 676:57 214:50 969:33 109:24
292:24 491:20 1004:17 644:17 67:13 695:13 534:13 40:12 548:10 111:10 36:8
352:6 175:5 652:5 485:4 321:4 103:4 350:4 12:3 413:2 0:0 0:0 0:0 0:0 0:0 0:0
0:0 0:0 0:0 0:0 0:0 0:0 0:0 0:0 0:0 0:0 0:0 0:0 0:0 0:0
```

Show Bit-Statistics

This command shows the distribution of bits in the memory locations of the current memory.

```
BIRD> Show Bit-Statistics
 Avg numbers: 1: 127  0: 128  *:0
```

This shows that each location has roughly 127 "1" bits, 128 "0" bits, and no "*" (dont-care) bits.

Show Candidates {**No-Address** | *<address>*}

This command shows the candidates that the genetic system would have selected.

Show Comparison Memory [**Distances** | **Counter-Weights** | **Location-Addresses** | **Selected-Sets**] *<memory-1-name> <memory-2-name>*

This command compares the internal values for two memories. It is primarily useful for debugging simulators, by allowing relatively rapid testing that equivalent memories on different platforms give the same results.

Show Counter-Vote [*<location-index>* | *<variable>*] {*<bit-position>* | *<variable>*}

My work on the statistics of data counters allows a value called a *weight* to be associated with each data counter. The weight is a measure of how predictive the data counter is. Depending on this weight, a given counter may "vote" for a "1" bit value (if it is predicting above the mean) or a "0" bit value (if it is predicting below the mean).

See **Show Weights** for more detailed information about a location counter's weight.

```
BIRD> Show Counter-Vote 10 1
>= mean
```

Show Crossover [*<min-distance>* | *<variable>*] *<address-1>* *<address-2>*

This command crosses over the two addresses if the distance between then is greater than the given minimum distance. The purpose of this command was simply to test the internal routines for doing crossovers.

```
BIRD> Show Crossover 10 1010- 1111-
OK! (98 & 30 >= 10)
Address1:
1010101010101010 1010101010101010 1010101010101010 1010101010101010
1010101010101010 1010101010101010 1010101010101010 1010101010101010
1010101010101010 1010101010101010 1010101010101010 1010101010101010
1010101010101010 1010101010101010 1010101010101010 1010101010101010

Address2:
1111111111111111 1111111111111111 1111111111111111 1111111111111111
1111111111111111 1111111111111111 1111111111111111 1111111111111111
1111111111111111 1111111111111111 1111111111111111 1111111111111111
1111111111111111 1111111111111111 1111111111111111 1111111111111111

Crossover:
1111111111111111 1111111111111111 1111111111111111 1111111111111111
1111111111111110 1010101010101010 1010101010101010 1111111111111111
1111111111111111 1111111111111111 1111111111111111 1111111111111111
1111111111111111 1111111111111010 1010101010101111 1010101010111111
```

Show Defaults

This command shows all the current defaults. Example:

```
BIRD> Show Defaults
 Set Default Address-Size 256
 Set Default Address-Type Bit-Addresses
 Set Default Area 15
 [... etc ...]
```

The **Show Defaults** command prints out the defaults in a format that looks like system commands; this allows the user to capture the commands for future use.

Show Definition {*<command-name>*}

This command shows the definition for the given command, if the command was previously defined using the **Define** commands. For example:

```
BIRD> Define Arf Variable $Loops
DEFINE> DoTimes $Loops
DOTIMES2> echo HI
DOTIMES2> End
DEFINE> End
BIRD> Show Definition arf
 Define ARF Variable $loops
  DoTimes $loops
   Echo HI
  End
 End
```

Show Difference *<address-1> <address-2>*

This shows the difference between two addresses. If no dont-care bits are involved, the difference is equal to the distance. If dont-care bits are involved, the difference counts as different a dont-care bit matched against anything but another dont-care bit. Example:

```
BIRD> Show Difference 11110000- 1111000*-
 Distance is 32
```

Show Distance *<address-1> <address-2>*

This shows the distance between two addresses. Example:

```
BIRD> Show Distance 11110000- 11110001-
 Distance is 32
```

This example illustrates something we haven't previously seen: how does the system assign the length to recursive addresses (that is, addresses that end in "-")? In most cases, it is done by context: for example, if we use an address in the **Address** command, then the expected length is the address size of the memory. If there is no expected length given by the context, then the value of the default address size (as set by **Set Default Address-Size**) is used for the size.

Show Fields

This commands prints out a lot of information concerning the encoding of the weather fields.

Show Genetic Candidates

This command shows information about the last pair of candidates for genetic crossover.

Show Genetic Parameters

This command shows the values of the parameters that are settable using the Set Genetic command.

Show Genetic Status

This command shows statistics about the classifications of the potential genetic candidates.

Show Input-Sums

Most of the simulators keep values called an input sum. Whenever a write operation is conducted, the input sum counters are also incremented and decremented. The input sums are equivalent to a memory location which is always selected during writing, though it does not contribute during reading.

This command also shows the total number of data addresses written into the memory.

Show Location [*<number>* | *<variable>*]

This command gives extensive information about the given location. The command requires an argument, which is the index of the location in the current active memory. Example:

```
BIRD> Show Location 333
[Lots of stuff about the location, including the location address and counters.]
```

Show Mapping *<address-array>* {All | *<element-index>* | *<variable>*}

This command shows information concerning the reorderings that are taking place in the given address array, either for every element, or for only the given element.

```
BIRD> Show Mapping L 10
 ORIGINAL order
BIRD> Reorder L Randomly 1 *
BIRD> Show Mapping L 10
 RANDOM order
 10 --> 580
```

The first command simply tells us that the array is in original order. After reordering, the array is in RANDOM order, and the reference to "L10" would actually refer to physical memory location 580.

Show Memories

This command shows a single line of information about all created memories. For exam-

ple:

```
BIRD> Show Memories
Memory Test, 1000 locations, size 256/256
Memory CM, 8192 locations, size 256/256, on Connection Machine
Memory CM-weather, 10000 locations, size 256/1, on Connection Machine
```

Show Memory

This command gives a sequence of commands which should be able to recreate the given memory, though it does not keep track of the read and write operations which have taken place.

```
BIRD> Show Memory
[Lots of Set Default, Create Memory, and Set Memory commands.]
```

Show Output-Sums

When a read operation is requested, the memory collects the data counters columnwise into sums, then thresholds these sums to create the read address. You can see these sums before thresholding with this command. For example:

```
BIRD> Show Output-Sums
[NOW: 16] 16 -16 16 -16 16 -16 16 -16 16 -16 16 -16 16 -16 16 -16 16 -16 16 -16
16 -16 16 -16 16 -16 16 -16 16 -16 16 -16 16 -16 16 -16 16 -16 16 -16 16 -16 16 -16
16 -16 16 -16 16 -16 16 -16 16 -16 16 -16 16 -16 16 -16 16 -16 16 -16 16 -16 16 -16
16 -16 16 -16 16 -16 16 -16 16 -16 16 -16 [...]
```

In this example, NOW gives the number of individual location writes contributed to this sum. The output sums are then printed out. In this case, the memory likely had the address 1010- written to it near where we were reading.

Show Plot Parameters

This command prints the current parameters for plotting.

```
BIRD> Show Plot Parameters
Set Plot Call-Plotview True
Set Plot Pixrect False
Set Plot Top-Title
Set Plot Bottom-Title
Set Plot Left-Title
Set Plot X-Dimensions Machine-Settable
Set Plot Y-Dimensions Machine-Settable
```

Show Plot Plot-File <*oldfile*>

This command attempts to display the given plot file.

Show Plot Data-File *<oldfile>*

This command attempts to display the given plot data file.

Show Radius-Estimate [*<address-size>* | *<variable>*] [*<desired-area>* | *<variable>*]

Assuming random addresses, this command calculates what a good radius would be, given the address size and the desired number of locations to select.

```
BIRD> Show Radius-Estimate 256 10
 For address size 256 and area 10, radius is 109
```

Show Score [*<starting-index>* | *<variable>*]
{**Weather-File-Size** | *<ending-index>* | *<variable>*}

This command scores the current memory as to its ability to predict rain. It runs through the given range of weather addresses and reads the memory, then compares the single-bit output of the memory with the rain information for the next four hours. It gives a total of correct and incorrect responses, broken down into "+" (rain) and "-" (no rain) categories.

```
BIRD> Show Score 1 100
correct:   +:14 -:7
incorrect: +:53 -:26
```

In this example, we got 14 correct predictions of rain, 7 correct predictions of dryness, 53 incorrect predictions of rain, and 26 incorrect predictions of dryness.

Show Selected

This command shows a list of the selected memory locations from the last **Address** operation. it also gives the distance from that memory location address to the sample address. For example,

```
BIRD> Address 1010-
 Selected 16 locations, radius used was 111
BIRD> Show Selected
 Format is: index <distance>
59 <106>, 90 <110>, 175 <111>, 194 <110>, 298 <111>, 339 <109>, 390 <106>,
516 <110>, 583 <108>, 600 <111>, 611 <110>, 617 <108>, 662 <111>, 716 <106>,
776 <110>, 791 <109>
```

We could confirm that a given distance is correct by typing:

```
BIRD> Show distance L59 1010-     ; dist btw loc 59 and address
```

Distance is 106

Show State-Info

Not currently implemented.

Show Status

This command prints the status of a variety of different **Set** commands. For example:

```
BIRD> Show Status
Set Active-Memory Woof
Set Command-Timing False
Set Dont-Care-Bits-In-Memories False
Set Ignore-One-Exact-Match False
Set K-Value 1
Set Print-Text-With-Addresses False
Set Random Seed 631394912
Set Random Address-Seed 333
Set Random Dont-Care-Bits-On 0
Set Random Address-Bits-On Random
Set Weather-Expansion 1
```

Show Storage-Statistics

This command shows information concerning the dynamic memory usage of the BIRD program. Most of the information will not be useful to the average user unless they are acquainted with the internals of the program.

Show System

This command shows the current values of the system parameters set using the **Set System** command.

```
BIRD> Show System
Set System Priority 0
Set System Required-Idle-Time 900
Set System Check-For-Idleness 60
Set System Sleep-Period-If-Not-Idle 0
Set System Command-Structure Full
```

Show Time

This shows information about the current time.

```
BIRD> Show Time
```

Wed Jan 3 12:00:07 1990

Show Underwritten-Statistics [*<location-index>* | *<variable>*]

This command shows statistics about the utilization of the given memory location.

```
BIRD> Show Underwritten-Statistics 10
For location 10:
Age: 3  Number of writes: 0
Probability of hit/write: 0.014648  Found probability: 0.000000
Mean: 0.043945  Std dev: 0.209631  Number of std devs from mean: 0.209631
```

Show Variable [*<function>* | *<variable>* | *<number>*]

For example:

```
BIRD> Show Variable $Radius
Value is 112
```

If a variable that has not been created is given to this command, the value "0" is printed. This is a widespread practice; using a variable that has not been defined is not an error, but uses the value zero. This is true for all commands except assignment; it is an error to try to assign the value of a variable that does not exist. For example:

```
BIRD> Assign Variable $never-created 12345
?Not a variable: $never-created
```

Show Version

This shows version information about the BIRD program.

```
BIRD> Show Version
BIRD version 3.0, brain.riacs.edu, created 29 Dec 89
Syntax version 3.0, simulators: software, Connection Machine (CM not attached)
```

The first line of the version info contains the major and minor version number of the program, the host name the program is running on, and the date the program was compiled. The minor version number is automatically augmented whenever a local change to BIRD is made.

The second line of the version info contains the syntax version and the simulators compiled into this instance of the BIRD program. The major software version is incremented whenever a syntactically-incompatible change is made to the program syntax. The minor version is incremented whenever an addition to the syntax is made.

Show Weather-Decoding *<address>*

This command shows the distance between each of the 16-bit fields of the given address and the 16-bit codes used to encode weather addresses. For example:

```
BIRD> Show Weather-Decoding w1
Station number: 15 (0)          [0]
Year: 61/62 (15)                [0/1/2/3/4/5/6/7/8/9/10/11/12/13/14/15]
Month: 1 (16)                   [0/2/4/8/10/12/16/14/12/8/6/4]
Day: 1/2 (16)                   [0/2/4/6/8/10/12/14/16/14/12/10/8/6/4/2]
Time: 0 (16)                    [0/2/4/8/10/12/16/14/12/8/6/4]
Pressure: 10026/10034 (13)      [2/1/0/1/2/3/4/5/6/7/8/9/10/11/12/13]
Dry Bulb: 269/273 (8)           [7/6/5/4/3/2/1/0/1/2/3/4/5/6/7/8]
Wet Bulb: 250/252 (11)          [11/10/9/8/7/6/5/4/3/2/1/0/1/2/3/4]
Dew Point: 25 (14)              [14/13/12/11/10/9/8/7/6/5/4/3/2/1/0/1]
Wind Direction: 12 (16)         [10/12/14/16/14/12/10/8/6/4/2/0/2/4/6/8]
Wind Speed: 3 (12)              [3/2/1/0/1/2/3/4/5/6/7/8/9/10/11/12]
Cloud Cover: 7 (13)             [13/11/9/7/6/4/2/0/2]
Present Code: 1/2 (14)          [1/0/1/2/3/4/5/6/7/8/9/10/11/12/13/14]
Past Code: 2 (13)               [2/1/0/1/2/3/4/5/6/7/8/9/10/11/12/13]
Rain: 0 (9)                     [0/9/8/9/8/9/8/9/8/9/8/9/8/9/8/9]
-----------------------
```

Each line begins with the name of the weather field. What follows is the field identifier for the field which matches the closest to the corresponding 16-bit field in the address. In parenthesis is given the range of that field, which is the difference between the closest encoding and the furthest encoding from the 16-bit field in the address. Finally, we give a list of each of the 16-bit field encoding distances from the 16-bit field in the address.

Show Weights [*<location-index>* | *<variable>*] {**Statistics-Only** | **Individual-Bits**}

My work on the statistics of data counters allows a value called a *weight* to be associated with each data counter. The weight is a measure of how predictive the data counter is. This command shows the weights associated with the data counters of a given location, either averaged (**Statistics-Only**) or for each of the individual bit counters (**Individual-Bits**).

A weight of 10000 is the maximum possible weight.

```
BIRD> Show Weight 10
 The predominant bit is 1
 Weights: avg: 324 best: 10000 worst: 33
```

Show Write-Statistics

This command prints a histogram of the memory usage of the memory locations.

```
BIRD> Show Write-Statistics
 Average number of writes is 0
```

1 ---> 38; 2 ---> 12; 3 ---> 1;

In this example, 38 locations had one write, 12 had two writes, and 1 had three writes. The remaining locations had zero writes.

Sleep [*<function>* | *<variable>* | *<number>*]

This puts the BIRD program to sleep for the given number of seconds

```
BIRD> Sleep 10
[The program goes away for 10 seconds, then returns]
```

Test

This is a hook for testing out commands written in C. Not much use unless you're planning to hack the internals of the BIRD program, in which case this is a simple place to insert commands for testing.

While [*<function>* | *<variable>* | *<number>*]

This command executes its block as long as the test is true (non-zero). For example, we may wish to write an address to the memory was long as reading the address does not result in the written address:

```
BIRD> Read
BIRD> While Not Same R 1010-
WHILE> Write 1010-
WHILE> Read
WHILE> End
```

Write *<address>*

The **Write** command writes the given address to the currently active memory. You must have some time previously used the **Address** command for this command to have any effect on the state of the memory.

APPENDIX B: A sample script of BIRD commands

B.1: A sample script that creates a memory, writes, reads, and saves

```
; Sample script 1.  This script creates a memory, writes to it,
; and reads from it, then saves it to a file.

; To execute this file, type "Input file script1.incl"

Set Default Address-Size 256      ; 256 bits of address
Set Default Data-Size 256         ; 256 bits of data
Set Default Number-Of-Locations 1000  ; 1000 locations
Create Memory Test                ; Memory named "Test"
Address 1010-                     ; Address with 1010...
Write 11110000-                   ; Write
Read                             ; See if it's there
Show Memory                       ; Show memory info
Show Output-Sums                  ; Show output sums
Output Memory Test.memory         ; Output to a file
```

B.2: A slightly more complex script

```
;; Sample script 2.
; The semicolon makes this a comment line.
; (Note the use of lowercase.)

; To execute this file, type "Input file script2.incl"

show version
set default address-size 256
set default data-size 256
create memory Test

;  The memory has been created.  Write 5 random addresses to it.

address z    ;  "z" is a random address
write z
address z
write z
address z
write z
address z
write z
address z
```

```
write z

;  Write a specific random address at a specific place

address z1
write z2

; See if we can read it.  Tell us how good the match was.

read z2

; Show some stats about the memory and save it.

show memory
show selected
output memory file.memory
; delete memory                ; doesn't work yet

; End of script
```

B.3: Testing memory capacity

```
;; Sample script 3.

; To execute this file, type "Input file script3.incl"

create variable $memory-size 1000
set default number-of-locations $memory-size
set default address-size 256
set default data-size 256
create memory Test

do $1 1 $memory-size
  address z$1 silently              ; write a new sample
  write z$1
  assign variable $2 0              ; $2 is number of mistakes
  do $3 1 $1                        ; loop over all stored items
    address z$3 silently
    read z$3 silently               ; READ sets var $distance
    if GT $distance 0              ; if not read perfectly, count
      assign var $2 plus $2 1
    end
  end

; print out a message every once in a while
```

```
  if GT $2 0
    echo "After storing" $1 "items, found" $2 "error(s)"
  else
        if not mod $1 10
      echo "Stored" $1 "items"
    end
  end
end
```

APPENDIX C: Installing the BIRD program at your site

The BIRD program needs some customization for the specifics of your site before it can be installed successfully. This appendix explains the process needed to install the BIRD program.

A.1. Extracting BIRD from tape or tar file

You have received the BIRD program on either a magnetic tape or as a file named SDM.tar. In either case, you will need to run the UNIX "tar" program to extract the BIRD source files. If you have a magnetic tape, then connect to an empty directory and type the command:

```
unix% tar xv
```

to a UNIX shell after the tape is mounted. If you have a file, then create a new directory, place the file SDM.tar in that directory, and type the command:

```
unix% tar xvf SDM.tar
```

These commands unpack the contents of the BIRD program. The directory is left with a number of files, including the *.c sources to BIRD and a Makefile to construct it with. Before typing "make", there are a few customizations you will need to perform.

A.2. Customizing your local version of the BIRD program

All the local changes you make should be in two files, "Makefile" and "sdm.h". If you feel you must change other files, please contact the author with information: every attempt has been made to allow the user to setup BIRD without having to delve into the depths of the program source code.

A.2.1. The Makefile

First, you much choose which combination of the three possible simulators you wish to include. You can include the local host software simulator, the Stanford hardware simulator, and the Connection Machine simulator.

Inclusion is not exclusive; you can include any combination of the above simulators, and even create multiple memories, with each memory using different simulator hardware.

All the option lines in the Makefile are at the beginning of the file, and start with a line containing "#***". To include an option, you uncomment the block of "make" commands following the option starting line. To remove an option, you comment out the lines by inserting a "#" at the beginning of the undesired commands.

For example, to include the Connection Machine code, uncomment the second and third lines:

```
#*** This includes software for the Connection Machine
#LCM2= -lparis
#CMFLAG= -DCM
```

by removing the initial "#" characters, leaving:

```
#*** This includes software for the Connection Machine
LCM2= -lparis
CMFLAG= -DCM
```

At this time, edit your Makefile to include the software simulator, the Stanford hardware prototype, or the Connection Machine. When you are finished, continue onto the next section. The default Makefile is to include the software simulator, and to not include either the Stanford hardware or Connection Machine software.

The Makefile contains a number of other options you will need to set.

The first of these options is whether to include fake calls for the CM/Paris instructions. This is only useful for development, so you will likely leave this line commented out:

```
#*** Include this if we wish to load the CM library fake calls (not
#*** very useful outsode of development work)
#FAKECM= -DCMX
```

The next is where to find the file paris.h, the include file for CM/Paris instructions. This is not needed if you are not including the CM source code in your compilation.

```
#*** Which paris.h file to load, "paris.h" (local) or <cm/paris.h>
#*** (global)
#PARIS= -DGLOBALPARIS
PARIS= -DLOCALPARIS
```

The following flag is uncommented if the UNIX version contains the *memalign() function.

```
#*** This is whether the software contains the *memalign() function
```

```
#MEMALIGN= -DMEMALIGN
```

The next line determines whether additional software is loaded into the memory allocation routines to do dynamic memory checking for overflow errors. It hasn't been much of a problem for me, so you may wish to turn it off for speed.

```
#*** This determines whether additional (time-consuming) software
#*** is added to do dynamic memory checking
MEMALIGN= -DMEMALIGN
```

A.2.2. sdm.h

The file "sdm.h" is the major include file for the BIRD program. All system variables such as file names are contained in this file.

The file "sdm.h" consists of two sections. The first section is called the variable section, and is where we keep the machine-dependent values which may have to be changed for each installation. The second section contains the constant, which should not be changed. These two sections are clearly marked in the file.

```
#define BUILDER "drogers"
```

You should place the login name of the program builder here. The builder in some versions has special permissions in the program, and is not logged in the log file.

```
/*#undef LOGFILE */
#define LOGFILE "/u5/drogers/sdm/sim/LOG"
```

This is a file where a log of users of the BIRD program are kept. The file should be universally writable. If you do not wish this option, use the "undef" line.

```
#define FULL_HELP_DIR "/u5/drogers/sdm/sim/Full-Help/"
#define STANDARD_HELP_DIR "/u5/drogers/sdm/sim/Std-Help/"
```

These are the directories where the help command files are kept. There are two directories; one for the help with the full command set, and another for a special "standard" command set of a proposed SDM interface standard. (The program defaults to the FULL command set.)

```
#define BUGCOMMAND "/usr/ucb/mail -s \"Bug report for BIRD\" drog-
ers@riacs.edu"
```

This is the command which is executed to send a bug report. Feel free

to add your name to the list of recipients.

```
#define SITE "BIRD site RIACS.EDU, SUN implementation"
```

Information about the local site.

```
#define FIELDFILE "/u5/drogers/weather/fields"
#define WEATHERFILE "/u5/drogers/weather/weather.data"
#define SIZEFILE "/u5/drogers/weather/filecount"
#define PTRFILE "/u5/drogers/weather/dataptrs"
```

These are files used to access the weather data from Darwin, Australia. The FIELDFILE is used to map the weather information into categories. The WEATHERFILE is where the weather data is kept. The SIZEFILE is the number of entries in the weather file. The PTRFILE is a list of the starting position in the weather file for each of the fields.

```
/* #undef GRAPH */
/* #define GRAPH "/usr/bin/graph" */
#define GRAPH "/u5/drogers/sdm/naxis/naxis"
```

This is the name of a program which does the preprocessing of our data files to make them plot files. The program I use is called "axis", and is mostly replaceable by the UNIX "graph" program. If you do not have either program on your machine, leave this undefined.

```
/* #undef PLOTVIEW */
#define PLOTVIEW "/u5/drogers/sdm/pv/pv"
```

This is the name of the program that takes a UNIX plot file and displays it on the screen. Currently, we only have this for SunView. Eventually, an X version would be spiffy. If you do not have such a program on your machine, leave this undefined.

```
/* #undef PLOT2PR */
#define PLOT2PR  "/u5/drogers/sdm/plot2pr/plot2pr"
```

This is the name of the program that translates plot files to pixrect files. If you do not have such a program on your machine, leave this undefined.

```
/* #undef GENETICSTATEDIR */
/* #undef GENETICSOURCEDIR */
#define GENETICSTATEDIR "/u5/drogers/sdm/State"
#define GENETICSOURCEDIR "/u5/drogers/sdm/Source"
```

These are the files for using the genetic state stuff. Not for the faint of heart. Best to just ignore this for now.

```
/* #undef IDLELOG */
#define IDLELOG "/u5/drogers/sdm/sim/IDLELOG"
```

These files are used when we are using the run-only-when-idle stuff. This is the LOG file for idleness. Leave undefined if you don't care: you most likely won't.

```
/* #undef STATUSFILE */
#define STATUSFILE "/u5/drogers/sdm/Status"
#define STATUSCHECKPERIOD 60
```

This is the directory where the host-specific status files are kept. The directory contains host-specific files which are "touched" occasionally to signal that the program is still running on that host. Leave undefined if you don't need this, which you probably don't.

A.3. Making the BIRD program

Once the customization have been made, you can make the BIRD program by typing "make" in the source directory. The final program is an executable called (appropriately enough) "bird"; this program can be moved to whatever directory you wish.

A.4. Using the Connection Machine simulator

The connection machine software is included in the BIRD program, but you must take two steps before using it. First, you must make sure that the BIRD program was compiled with the Connection Machine software installed. To do this, check the Makefile for bird to make sure that the lines for including the Connection Machine software are not commented out. These lines look somewhat like this:

```
#*** This includes software for the Connection Machine
LCM2= -lparis
CMFLAG= -DCM
```

You should ensure that the lines starting with LCM2 and CMFLAG are not commented out, that is, do not begin with the character "#". If they do, remove the "#" character, delete the current version of BIRD (if any), and remove all the ".o" files from the BIRD directory. Typing "make" will now create a version of BIRD that includes the Connection Machine software.

To run the BIRD program using the Connection Machine, you must run BIRD from a CMATTACH subshell. To do this, type:

```
unix% cmattach
```

to the UNIX prompt. (If this does not work, consult your local Connection Machine guru.) This will attach and cold-boot the Connection Machine. It will then run a new shell, leaving you at UNIX top-level. You now run bird by typing:

```
unix% bird
BIRD version 0.9, brain.riacs.edu, create 28 Dec 89
Syntax version 1.1, simulators: software, Connection Machine
BIRD>
```

Finally, you need to set the default hardware type so that memories you create are on the Connection Machine. Type:

```
BIRD> Set Default Hardware Connection-Machine
```

From this point onward, all memories created will use the Connection Machine. When you are finished, quit out of BIRD and then quit out of the CMATTACH shell by typing "quit" to the UNIX prompt. This frees the Connection Machine to others who may wish to use it.

Because of the structure of the Connection Machine, you will get maximum performance if you choose the number of locations in a memory to be a power of two. If you do not, the extra locations are created but ignored.

APPENDIX D: Machine-independent address format

Proposed standard format for a machine-readable SDM address.

David Rogers 14 June 1989 (drogers@riacs.edu)
Updated by David Rogers, 15 September 1989

The SDM address consists of a two-line header followed by an optional block of text information, then by a block of address information, and optionally followed by a block of dont-care information. A file of SDM addresses is simply a sequence of records, each of which contains a legal SDM address.

Whitespace is defined as a space, a tab, a linefeed, or a carriage return. Whitespace is ignored in the address and dont-care blocks, to allow formatting to make them easier to read.

The header:

Line 1: "SDM Address 1"

Line 2: five numbers:
the address type, either 0 (bit) or 1 (float)
the length of the address, in bits
the number of lines in the name block
the number of lines in the address block
the number of lines in the dont-care block, 0 if none.

The header is followed by an optional name block, then an address block, and optionally a dont-care block.

The address block is a string of characters "0" and "1" (for bit addresses) or a string of floating-point numbers (for float addresses). For floating-point addresses, each number must be separated by at least one whitespace character. For bit addresses, no whitespace separation is required. The numbers in floating-point addresses must be between 0.0 and 1.0 inclusive.

If a dont-care block is given, it is in the same format as the address block, with the "0" being dont-care, and "1" being fully-care. Examples:

SDM Address 1
0 16 0 1 0
1111111111111111

SDM Address 1
0 32 2 1 0
random address
with two lines in the name block
00000000 11111111 01010101 11110000

SDM Address 1
0 16 1 1 1
address with dont-care bits
00100100101010110010101010101010
11110000111100001111000011110000

SDM Address 1
1 16 1 2 0
floating address, with the address formatted over two lines
0.30.50.30.91.01.00.00.0
1.00.90.560.550.33331.01.0.999999

HEX format binary addresses

Outputting binary addresses as bits is very time- and space-consumptive. Thus, I decided to add a definition for HEX format binary addresses. These compact four bits sequences of the address into a single hex digit, i.e., 0101 --> "9".

The header:

Line 1: "SDM Address 1"
Line 2: five numbers:
the address type, 2 (binary address in HEX)
the length of the address, in bits
the number of lines in the name block
the number of lines in the address block
the number of lines in the dont-care block, 0 if none.

The address block is a string of hex (4-bit) characters "0" through "9" and "A" (or "a") through "F" (or "f"). Extra whitespace characters are ignored. Examples:

SDM Address 1
2 16 0 1 0
FFFF

SDM Address 1
2 32 2 1 0
random address
with two lines in the name block
00FF 99F0

SDM Address 1
2 256 0 1 0
D16A118DD69A08F1E0871A507E17AADFEFE7853BDD863E41477F69E711562E52

END DATE DEC. 9, 1991