

The Radio Data Logger Company Ltd reserves the right to make changes to this manual or to the products referred to herein without notice.

The Radio Data Logger Company Ltd makes no warranty or representation with regard to this document or any product described herein including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The use of Radio Data Logger Company Ltd products for medical or safety critical applications is not authorised.

This work is the property of, and embodies information proprietary to the Radio Data Logger Company Ltd, and may not be copied, used, transferred, adapted or modified without the express permission of the Radio Data Logger Company Ltd.

This does not affect your statutory rights.

Copyright © 1997- 2004 The Radio Data Logger Company Ltd as unpublished works. All rights reserved.

CONTENTS

1	Introduction.....	1
2	An overview of <i>lucid</i>	2
3	<i>lucid</i> filenames	3
3.1	<i>lucid</i> programs	3
3.2	<i>lucid</i> output files	3
3.3	<i>lucid</i> input files	4
3.4	<i>lucid</i> macro files	4
3.5	SMS control files	4
4	Layout of <i>lucid</i> programs and macros	5
4.1	<i>lucid</i> programs	5
4.2	Layout of a <i>lucid</i> macro	6
5	Running multiple tasks: global and local resources	8
6	Representation of numbers.....	9
7	The <i>lucid</i> command set	10
7.1	Task directives	10
7.2	Logging control	11
7.3	File output functions	12
7.4	Time functions	13
7.5	Measurement i/o operations	13
7.6	Integer arithmetic	16
7.7	Floating point arithmetic	18
7.8	Variables	21
7.9	Repeated and conditional execution	23
7.10	Integer comparison operations	23
7.11	Floating point comparison operations.....	26
7.12	Boolean operations	26
7.13	Type conversion.....	28
7.14	Accumulator operations	29
7.15	Input file functions.....	31
7.16	Utility functions	33
7.17	Macro support functions	36
7.18	Communications support functions	38
7.19	RDT support functions.....	41
8	Using the trace and step facilities	42
8.1	Tracing program execution	42
8.2	Stepping through a program	43
9	<i>lc</i> - the <i>lucid</i> checking program	45
9.1	Starting <i>lc</i>	45
9.2	Checking programs which do not set a repeat interval.....	48
9.3	Error messages.....	48
9.4	Directing 'clean' output to a file and suppressing screen output	50
10	<i>lucid</i> error messages	51

1 Introduction

This manual provides a comprehensive description of the facilities available within *lucid*, the language used to program the Etherlog 3000.

Many of the *lucid* examples in this manual are complete programs, and they may be run on a logger. When loggers are supplied they have a temperature sensor connected to analogue input 1, and a push button switch attached to digital input 1. Wherever possible the examples take advantage of this, and if you have these sensors attached to your logger you can use them to verify the correct operation of the programs.

If you do intend to run the example programs on a logger you can familiarise yourself with the procedures for transferring program examples to the logger, executing them and retrieving the data that they record by consulting the companion to this manual:

Etherlog 3000 Operating Manual.

Wherever possible the material in this document does not duplicate the material in the Operating Manual, and it is suggested that you read that manual first in order to familiarise yourself with all the features of the Etherlog and the Windows host PC software.

2 An overview of *lucid*

Everything in a *lucid* program is written in plain text, and can be composed using any text editor that stores information in plain text format. This text is interpreted when the program is run. It is not necessary to compile or otherwise preprocess *lucid* programs before they are used.

lucid programs are constructed from keywords (or commands), and numeric or text character strings. Each command performs a specific function, and may expect values (formally, its arguments) on which it performs its function. All commands must appear in lower case.

The command keywords, and any numeric or text arguments supplied to commands, must be delimited by 'white space' (space, tab and new line characters). Apart from this, white space is normally ignored. Thus commands and arguments may be split across lines of a *lucid* program file, or several commands may appear on the same line.

Comments can be included anywhere in a *lucid* program, to remind you why something is being done. Whenever the '#' or '/' characters are encountered in a place where a keyword or argument is expected, the rest of the line (up to the next new line) is ignored.

In the examples in this manual we have used indenting to improve readability. This is optional (because white space is ignored), but recommended.

All *lucid* commands expect their arguments to *follow* the keyword. As an example, the instruction to divide 1000 by 10 (often written as 1000 / 10) is written in *lucid* as:

```
div 1000 10
```

Though slightly less easy to understand at first, this representation has several benefits in speed and efficiency. Try thinking of the above expression as 'divide 1000 by 10'.

A *lucid* command may return a result - in this case the `div` instruction will return 100. This result can be made available to some other command as one of its arguments by placing the `div 1000 10` expression in the position in which that command expects one of its arguments. For example:

```
mul 3 div 1000 10
```

calculates $3 \times (1000 / 10)$, and the `mul` instruction returns the value 300. Think of this as 'multiply 3 by the result of dividing 1000 by 10'.

Notice that we chose to write the expression in this way, rather than as:

```
mul div 1000 10 3
```

which, although it is functionally equivalent, is harder for a human to read. In the first example the first argument of the `mul` instruction is immediately obvious. In the second it is necessary for the human reader to remember that a second argument for the `mul` function is still pending while interpreting the `div` instruction.

The available *lucid* commands are described in detail in Chapter 7.

3 *lucid* filenames

lucid programs and macro files, and the data files that they read from and write to, are all held within the RAM filing system of the Etherlog 3000. Filenames must conform to the DOS-like 8.3 **name.extension** format, and may contain only alphanumeric characters (a..z, 0..9). Files may not have a blank name part, but may have a blank extension part.

File names are case-sensitive in the Etherlog – that is, two files filter.dat and Filter.dat can exist and have different contents. Note that this differs from the convention in many PC operating systems, where file names are not case-sensitive. If files are to be transferred to and from a PC, it is recommended to use lower-case file names only on the logger.

Files associated with various functions may be given any valid name; however there are certain conventions about the naming of files to which you will find it convenient to adhere.

3.1 *lucid* programs

lucid programs are normally held in files with the extension **.lcd**. The host PC software which operates in the Windows environment will only recognise files with this extension as valid programs. At the logger command prompt issuing the command:

```
log start test
```

will cause the logger to look for a file called **test.lcd** and run that as a logging task if the file is found. In command line mode you can, if you wish, give your program file a different extension, and then override the default behaviour by specifying the full filename in the *log start* command, for example:

```
log start trial.tst
```

will run the *lucid* program in **trial.tst**, assuming that the file can be found.

3.2 *lucid* output files

Results are sent to output files from within a *lucid* program using the `write` and `newline` commands. By default, the output file has the same name as the task, but with the extension **.dat**. For example, the default output file from the *lucid* program contained in **test.lcd** is **test.dat**. This can be overridden using the `logout` command inside a *lucid* program, described in more detail in Section 7.3.

The **.dat** extension has a special significance to the software running on the host PC. When a file with this extension is transferred from the logger to the host, the software assumes that it is likely to contain data, and as well as storing it attempts to translate that data into comma separated variable format, for import into other data processing packages. If you want to retain this automatic translation facility you should always use the **.dat** extension for data files. It should only be changed in situations where you plan to use your own software to read the *lucid* output files directly, in which case you will want to suppress the automatic translation process.

The `logout` command enables output to be directed to a file whose name is generated from a integer argument, which could be a timestamp or any other value. The filename will be the letter 'f' followed by 5 decimal digits expressing the low-order 16 bits of the value of the argument, followed by the extension **.dat**. For example, the command `logout n 1023` will set the new output file name to **f01023.dat**.

3.3 *lucid* input files

lucid programs can read data from files stored in the logger's filing system: this data can typically be used to store sensor calibration constants. By default, a task's input file has the same name as the task, but with the extension **.inp**. However, a task can use several input files by making use of the `login <filename>` command to specify files for subsequent input file operations.

The *lucid* `nlogin` command generates filenames in an identical way to the `logoutn` command described above, including the **.dat** extension. The reason for using the same extension is to permit the same file to be used for both output and input from *lucid* programs.

3.4 *lucid* macro files

Macros (which will be described later in this manual) allow groups of *lucid* commands to be run from the command line using the interactive `eval` command, or readings to be processed and displayed in engineering units by *winhost*. Macros are stored in normal files with any valid name; there is no default extension for macro files. It is usually convenient, though, to store macros in files with a descriptive name and no extension, so that the command-line instruction to evaluate the macro is more readily understandable. For example, the command

```
eval humidity
```

is clear and concise.

Macros may also be evaluated from within *lucid* programs. The readability of programs will also benefit from the use of this type of filename.

3.5 SMS control files

There is no default extension for SMS control files. However, using the extension **.sms** for such files will help to indicate the purpose of the file – for example, **temphigh.sms** has a descriptive name and its extension identifies its function.

4 Layout of *lucid* programs and macros

In order for the Etherlog to run them, *lucid* programs and macros must be laid out in a particular way. This section describes their format.

4.1 *lucid* programs

The simplest format for a program is as follows (recall that ‘#’ characters introduce comments):

```
# specify the task repetition interval: set to 10 seconds in this example
repeat 10

# main task
begintask
    lucid instructions to be executed each time the main task executes
endtask
```

If the `repeat <n>` command is not present, the logger will not know how often to execute the main task, and the main task will never be started; so it is normally necessary to specify a repeat interval before the main task (which is delimited by `begintask` and `endtask`) is defined.

4.1.1 The initialisation section

Often there are instructions which should be executed before the first execution of the task, for example to initialise variables. These instructions can be placed at the beginning of the program in an initialisation section. The resulting code looks like this:

```
# initialisation section
lucid instructions to be executed before the first execution of the task

# specify the task repetition interval: set to 10 seconds in this example
repeat 10

# main task
begintask
    lucid instructions to be executed each time the main task executes
endtask
```

It is quite possible to write a *lucid* program which consists only of an initialisation section, and has no main task. This is the only situation in which it is sensible to omit the `repeat <n>` command. In this case the code in the initialisation section will be run when the program is started, and the task will then be stopped. This can be a useful way of testing short segments of *lucid* code. For example, the following program demonstrates the function of the `div` and `mul` instructions described in Chapter 2:

```
# this program demonstrates the div and mul functions
newline
write div 1000 10
write mul 3 div 1000 10

# there is no repeat interval and no main task
```

If you run this program, the logger will complain that no repeat interval has been given, but there will be a data file in the filing system which contains the results from the initialisation section.

4.1.2 Using subtasks

lucid allows you to define up to four subtasks, which are groups of instructions which can be executed simply by ‘calling’ the subtask. In this case the resulting program has the following form:

```

# initialisation section
lucid instructions to be executed before the first execution of the task

# subtask section - up to four subtasks may be defined in this way
subtask nameofsubtask
    lucid instructions to be executed when subtask is called
endsubtask

# specify the task repetition interval: set to 10 seconds in this example
repeat 10

# main task
begintask

    lucid instructions to be executed each time the main task executes

# sample call to the subtask defined above
call nameofsubtask

    more lucid instructions

endtask

```

It is not possible to pass parameters to a subtask other than in task-local variables. Subtasks are purely a way of grouping sets of instructions. This may be desirable if a group of instructions is used more than once, as it saves entering them each time they are required in the program. Alternatively, they may just be used to improve program readability by breaking the program into a series of smaller units, each of which may be given a name which summarises its purpose.

It is possible to call a subtask from within itself, but this is a feature which must be used with great care. There are relatively few logging tasks which require such recursion, and we strongly suggest that its use is avoided.

4.2 Layout of a *lucid* macro

Macro files differ from *lucid* programs in the following ways:

1. no repeat interval is required;
2. there is no initialisation section;
3. the macro body is delimited by `macro` and `endmacro` directives.

Since macros are evaluated on demand either from the command line or from a *lucid* program, there is no need to define a repeat interval. The macro is evaluated once and then does no more until it is next invoked by whatever means, so there is no purpose for an initialisation section either. Any local variables used during the execution of a macro lose their values when the macro has completed running, so cannot be used to store values from one evaluation to the next.

The work of the macro is defined by the *lucid* commands appearing between the `macro` and `endmacro` commands: these take the place of the `begintask` and `endtask` commands in a conventional *lucid* program.

An example macro skeleton is shown below.

```

# macro body code
macro
    lucid instructions to be executed each time the macro is evaluated
endmacro

```

Subtasks may be defined inside macros if required, and have identical functionality to subtasks used in *lucid* programs. When present, they must appear before the `macro` command (similarly to appearing before the `begintask` command in *lucid* programs). An example skeleton is shown below.

```
# subtask section
subtask nameofsubtask
    lucid instructions to be executed each time the subtask is called
endsubtask

# macro body code
macro
    lucid instructions to be executed each time the macro is evaluated

    # sample call to the subtask defined above
    call nameofsubtask

    more lucid instructions
endmacro
```

5 Running multiple tasks: global and local resources

Up to four *lucid* programs can be run simultaneously. Each program is as self contained as possible, with its own timekeeping, variable storage etc. These resources are ‘local’ to each program as it runs. This ensures that one program cannot interfere with the functioning of another by, for example, changing the value of a variable that is already being used for something else. This has the advantage that the author of one program does not have to consider what other programs might be running at the same time when allocating, for example, a variable. In this context, note that running a macro is equivalent to starting and stopping a task – in other words, a macro uses one task’s worth of resources (but only while running).

Inevitably, there are some program resources which relate to physical aspects of the data logger, and by definition these have to be global. Examples are the data acquisition hardware, the logger filing system, and the radio and modem communications resources.

The logger input hardware can generally be used independently by all tasks. In the case of pulse counters, if a task could reset a counter this could disrupt the operation of any other tasks also using that counter. In fact *lucid* protects you from this: there is no pulse counter reset instruction. Instead, if a task wishes to know the number of pulse which have arrived over a given interval it must store the cumulative count (in a local variable) at the start of that interval, and then calculate the number of pulses which have arrived as the difference between that value and the cumulative count at the end of the interval.

For the logger outputs, however, it is not possible to build in this sort of protection. If a task sets a digital output and leaves it set then it will be in that state when the next task is executed. Thus it is wise for a task not to assume that a digital output will be in the state that it was left in when the task last executed - another task may have changed it since.

The logger filing system is also clearly a shared resource. Perhaps the most obvious example of how one task can affect the operation of others is by filling up the available space, making it impossible for any task to write further data. It is also possible that two tasks might write data to the same file. The logger will not object to this, but each task will effectively be interfering with the format of the data stored by the other.

Eight global variables called `arg 1` to `arg 8` are provided to allow tasks to pass data values to macros and to the SMS subsystem. The fact that they are global – visible to all tasks – permits any task to use a given macro. It is possible for a task to store data in these variables, but it is inadvisable to expect that the value will not have been overwritten in the time between task executions. The `arg` variables should only be used for passing values to macros and the SMS subsystem.

The radio is another example of a hardware resource shared by all the tasks that are running at any given time. Any task may initiate the transmission of a radio advertisement without risk of interfering with another task, although if several tasks do this it may result in increased logger power consumption. Associated with each advertisement however is a radio alarm level. If a task sets this level in order to alert the host PC to an alarm condition it will over-ride the alarm level set by any other task. One way around this is to allocate different bits of the alarm code to different tasks, which then set or clear only the bits that they are allowed to use. An example of this is given in Section 7.18 where the generation of alarm codes is treated in more detail.

In conclusion, wherever possible *lucid* encourages the programmer to keep resources local to a task, where they cannot interfere with, or be interfered with, by other tasks. In some cases however, global resources inevitably have to be used. In these situations the programmer should take care not to make changes to these resources which will disrupt the operation of other tasks, and at the same time to make the task as immune as possible to such interference from other tasks.

6 Representation of numbers

Every number held within *lucid* has associated with it a type. Three types are currently defined: integer, floating point, and time. Each type of number is stored in 32 bits:

1. integers are represented as signed 32 bit numbers, with a range of about $\pm 2\,000\,000\,000$,
2. floating point numbers are stored in IEEE 754 standard 32-bit single-precision format with 1 sign bit, 8 bits of exponent and 23 bits of mantissa, giving a range of approximately 10^{-38} to 10^{38} before overflow or loss of accuracy occurs due to denormalisation, and approximately 7 decimal digit accuracy.
3. time is stored as an unsigned 32 bit integer, interpreted as seconds elapsed since 00:00:00 01/01/1970 UTC.

Numbers may be specified within a *lucid* program in decimal or hexadecimal notation. If a string of characters starts with '-', '+', or 0., then it is treated as a number. If such a string starts 0x it is interpreted as a hexadecimal number, and converted to an integer. If not the string is interpreted as a signed decimal number. If it contains a decimal point or the character 'e', it is treated as floating point; otherwise it is read as an integer. For example:

10 is interpreted as the integer 10

-10 is interpreted as the integer -10

0x10 is interpreted as the integer 16

128.0 is interpreted as the floating point number 128.0

1.562e2 is interpreted as the floating point number 156.2

Every *lucid* command expects and returns arguments of a given type. For example, the integer `add` instruction expects to be given two integers, and returns their integer sum. If one or both arguments are of the wrong type then they will be converted automatically to the type required before the instruction is executed. For example before the statement:

```
add 1 1.562e2
```

is executed, the second argument is converted to the integer 156 because `add` expects integer arguments. The `add` instruction actually evaluated is thus:

```
add 1 156
```

returning the integer 157. The floating point `fadd` instruction, on the other hand, expects two floating point arguments and returns a floating point result. In the expression:

```
fadd 1 1.562e2
```

the first argument is automatically converted to a floating point value, and when:

```
fadd 1.000e0 1.562e2
```

is evaluated it returns the floating point number 157.2.

In some cases the programmer may wish to 'force' the type conversion of a particular value, and functions are provided within *lucid* to facilitate this.

7 The *lucid* command set

The commands available in *lucid* are described in the following section. Examples are usually given to demonstrate the use of each command.

The arguments required by each command and the type of result it returns are shown using the following notation:

<n>	type integer
<f>	type floating point
<t>	type time
<a>	arbitrary type
<name>	character string argument
<time>	character string representing time, in hh:mm:ss format (eg 12:31:45)

Where the type of value returned by a command is shown as arbitrary it indicates that it can vary depending on the type of arguments supplied. In this case the rules used for determining the return type are summarised.

7.1 Task directives

Command	Argument(s)	Return	Description
begintask			Never executed - identifies the start of the main task
endtask		<n>	Returns 0 and stops logging
macro			Never executed - identifies the start of a macro
endmacro		<a>	Returns control to the caller (command line interface or running task). Returns result of value and type of result of last executed command
subtask	<name>		Never executed - identifies the start of a subtask
endsubtask		<a>	Returns control to the main task following a call to a subtask. Returns result of value and type of result of last executed command
call	<name>	<a>	Transfers flow of control to a subtask. . Returns result of value and type of result of last executed command
start	<name>	<n>	Starts a named task, returning 1 if started, 0 if not
stop	<name>	<n>	Stops a named task, returning 1

The `begintask`, `endtask`, `subtask`, `endsubtask` and `call` instructions are used as described in Chapter 4 to lay out a *lucid* program. Likewise, the `macro` and `endmacro` instructions define the body of a *lucid* macro.

The `start` and `stop` instructions can be used within a *lucid* program to control the execution of other *lucid* tasks. When a task is started from within a program it becomes that program's 'child'. A *lucid* program can only stop itself and its own children, not other tasks which have been started independently from the command line or from the `startup.cmd` file. When a task is stopped, its children are also stopped.

7.2 Logging control

Command	Argument(s)	Return	Description
<code>repeat</code>	<code><n></code>	<code><n></code>	Sets task repeat interval to <code>n</code> seconds, returns value of <code>n</code> .
<code>window</code>	<code><n1><time1> <time2></code>	<code><n></code>	Sets logging window <code>n1</code> (<code>n1=1</code> or <code>2</code>) to start at <code>time 1</code> and end at <code>time 2</code> .

Almost all *lucid* programs repeat a main task at a given interval, and the `repeat` command, which defines this interval is therefore vital to the correct operation of a task. The interval is specified in seconds and must have a value of at least 5. The `repeat` command can be used within the main body of a task if required, to vary the repeat interval under program control.

The `window` command allows a task to be executed only at selected times of day. You can associate up to two logging windows with each task that you define. This is done in the initialisation section of the program. The following task executes only between 6:00am and 6:00pm:

```
# set daytime logging window
window 1 6:00 18:00

repeat 60

begintask
  lucid instructions here
endtask
```

If the `window` command was changed to:

```
# set nighttime logging window
window 1 18:00 6:00
```

then data would be recorded only between 6:00pm and 6:00 am. The following example uses two windows, and the associated task will execute from 9:00pm to 3:00am, and from 9:00am through to 3:00pm:

```
# set logging windows
window 1 21:00 3:00
window 2 9:00 15:00
```

As with the `repeat` command, `window` commands may be used within the main body of a task to vary logging windows under program control.

7.3 File output functions

Command	Argument(s)	Return	Description
logout	<name>	<n>	Sets the name of the results file to name and returns 1
nlogout	<n1>	<n>	Sets the output filename to fnnnnn.dat where “nnnnn” is the decimal representation of the low-order 16 bits of <n1>. Returns 1.
newline		<n>	Writes a start of record marker, or newline, to the output file. Returns 1 if successful, 0 if not.
write	<a1>	<n>	Writes the value of a1 to the output file. Returns 1 if successful, 0 if not.
logsize		<n>	Returns the size of the current output file, in bytes.
logdel		<n>	Deletes the first record in the current output file, and returns the number of bytes deleted. Does nothing if the first record does not start with newline.

By default a *lucid* program outputs data to a file which has the same name as the program, but has extension **.dat** rather than **.lcd**. The `logout` command can be used to change this, and route data to a different file. For example, the *lucid* program `test1.lcd` would normally route data to the file `test1.dat`. However, if it contained the instruction:

```
logout alarm.dat
```

data would instead be routed to the file **alarm.dat**.

If you want the output file to be named according to the value of a number, the `nlogoutn` command can be used. This takes its argument, evaluated as an integer, and represents the low-order 16 bits of this as a decimal string for the name part of the filename. The name is prefixed with ‘f’, and the extension is always **.dat**. For example, the command

```
logoutn 15
```

sets the output filename to `f00015.dat`.

The `newline` command is used to signal the start of a new data record in a file. If the data file has the extension **.dat** then the host PC software will attempt to convert it to comma separated variable format when it is transferred. In this case, each instance of the `newline` flag in the data file will force the start of a new line in that file.

In cases where you need to restrict the size of the output file, the `logsize` and `logdel` commands are provided. `logsize` returns the size, in bytes, of the current output file (either the default or one set by `logout` or `logoutn`). `logdel` deletes the first record in the current output file, providing it starts with a `newline` record marker. Using conditional control structures described later, it is simple to limit the size of the output file by discarding the oldest records in the file. For example:

```

while > logsize 50000
  logdel
endwhile

```

deletes records from the current output file while its size is greater than 50 kbytes.

7.4 Time functions

Command	Argument(s)	Return	Description
<code>time</code>		<t>	Returns the time in seconds since 00:00:00 01/01/1970
<code>sett</code>		<t>	Sets the local variable t to current time and returns its value
<code>t</code>		<t>	Reads the value of the local variable t

The most common use of the `time` function is when timestamping data written to an output file. This is simply achieved by the code:

```
write time
```

The `sett` command allows you to ‘snapshot’ the value of the clock at any stage. For example a program which contained a long section of code which took several seconds to execute might want to write the time at which the task actually started to the output file, rather than the time that the write actually took place. The code below achieves this by making a record of the time at which the task started:

```

repeat 10
  begintask
    # make a record of the time the task started
    sett
    time consuming instructions
    # write new line and time task started to output file
    newline
    write t
    instructions to write remainder of data record to output file
  endtask

```

7.5 Measurement i/o operations

7.5.1 Etherlog on-board i/o

Command	Argument(s)	Return	Description
---------	-------------	--------	-------------

ain	<n1>	<n>	Returns the reading of analogue input n1 (n1 in the range 1 to 8) in microvolts
lin10k	<n1>	<n>	Converts n1, an analogue reading in microvolts, to a temperature in °C/100 in the range -2500 to 10000 using the linearisation curve for a 10kΩ curve matched thermistor in series with a 10kΩ resistor.
din	<n1>	<n>	Returns the state of digital input n1 (n1 in the range 1 to 8), 1 if input is active, 0 if inactive
dout	<n1>	<n>	Returns the state of digital output n1 (n1 in the range 1 to 4), 1 if output is on, 0 if off
doutrd		<n>	Reads all four digital output states and returns an integer with bits 0..3 corresponding to digital output 1..4 states
doutld	<n>		Transfers bits 0..3 of n to digital outputs 1..4 respectively
dset	<n1>	<n>	Sets digital output n1 (n1 in the range 1 to 4), returns 1
dclr	<n1>	<n>	Clears digital output n1 (n1 in the range 1 to 4), returns 0
led	<n1>	<n>	Turn on the LED for n1 ticks (1 tick = 10ms); return n1
wait	<n1>	<n>	Wait for n1 ticks (=n1/100 seconds); return n1
pulse	<n1>	<n>	Returns the value of pulse counter n1

These *lucid* functions are clearly crucial to the operation of the Etherlog, and armed with them we can at last write a complete data acquisition program:

```
repeat 10
begin task
# flash the led for 10 ticks (=0.1 seconds)
led 10

# create a new record in the datafile
newline

# write the time
write time

# and the linearised value of analogue input 1
write ain 1
```

```
endtask
```

This program repeats every ten seconds. It flashes the on-board led and then writes a new line in the data file, consisting of the time, and the value recorded on analogue input 1 (returned by the *lucid* command `ain 1`). These are the values which will appear in the comma separated `.txt` file when the `.dat` file that this task writes is transferred back to the host PC.

The `din` function returns the state of a logger digital input. If the input is powered it returns 1, otherwise 0. This can be written directly to an output file:

```
write din 1
```

or, as we will see later, it can be used to control some other function within a *lucid* program.

The `dset` and `dclr` instructions allow you to set and clear the Etherlog's four digital outputs. The `wait` command forces a program to wait for a given number of clock ticks (1 clock tick = 10ms). It is commonly used in conjunction with the `dset` and `dclr` commands to allow external equipment powered by the logger time to stabilise. For example, suppose that an amplifier attached to channel 1 is powered *via* digital output 4, but that it takes one second to warm up before use. The following code uses the `wait` command to allow the amplifier to warm up before use:

```
# switch on the amplifier
dset 4

# wait one second for it to warm up
wait 100

# take the reading and write it to the output file
write ain 1

# turn the amplifier back off
dclr 4
```

The `dset` and `dclr` instructions can only control one digital output at a time. If you need to change the state of two or more outputs simultaneously, the `doutrd` and `doutld` instructions can be used: `doutrd` reads digital outputs 1..4 to bits 0..3 of its integer return value, and `doutld` transfers bits 0..3 of its argument to digital outputs 1..4. You should be careful to read the values of outputs that you do not wish to modify so they can be restored when writing the new values that you do want to control

7.5.2 RDT expansion i/o

Expansion of the analogue and digital input capabilities of the Etherlog is achieved using RDT 3000 telemetry modules, each of which has eight analogue inputs and four digital inputs. Each RDT has a unique identifier (or address) which is assigned at the time of manufacture. Up to eight RDTs can be configured for used with each Etherlog, and they are distinguished within it by their addresses.

Each RDT periodically transmits a message containing the values of all inputs and pulse counts. The Etherlog will store this information for each RDT it has been configured to receive. Each new message overwrites the previous one from the same RDT, so the data values available are the latest ones from each RDT.

The following commands give access to RDT data for use in *lucid* programs.

Command	Argument(s)	Return	Description
<code>rdtain</code>	<code><n1> <n2></code>	<code><n></code>	Return the current value of RDT <code><n1></code> analogue channel <code><n2></code> in microvolts (<code>n2</code>)

			in the range 1..8)
rdtdin	<n1> <n2>	<n>	Return the state of RDT <n1> digital input <n2> (n2 in the range 1..4)
rdtpulse	<n1> <n2>	<n>	Return RDT <n1> pulse count channel <n2> (n2 in the range 1..4)
rdtttime	<n>	<t>	Return the timestamp of the last message received from RDT <n>

The `rdtain`, `rdtdin` and `rdtpulse` commands are used in a similar way to the `ain`, `din` and `pulse` commands used to read the on-board Etherlog inputs. The important difference is that the first argument to each command is the address of the RDT whose data is to be read; the channel number follows. The following example illustrates their use.

```

newline
write rdtain 35 1      # write selected data values from RDT 35
write rdtain 35 2
write rdtdin 35 1
write rdtpulse 35 2

```

The `rdtttime` command is not normally used to write timestamps to data files: its intended use is to provide a means of determining how old the data from an RDT is, so that appropriate action can be taken if messages are no longer being received from it.

7.6 Integer arithmetic

Command	Argument(s)	Return	Description
add	<n1> <n2>	<n>	Returns $n1 + n2$
sub	<n1> <n2>	<n>	Returns $n1 - n2$
mul	<n1> <n2>	<n>	Returns $n1 \times n2$
div	<n1> <n2>	<n>	Returns $n1 / n2$ (if $n2$ is zero, returns <code>maxval</code> or <code>minval</code>)
max	<n1> <n2>	<n>	If $n1 > n2$ returns $n1$, otherwise returns $n2$
min	<n1> <n2>	<n>	If $n1 < n2$ returns $n1$ otherwise returns $n2$
mod	<n1> <n2>	<n>	Returns remainder of $n1 / n2$
neg	<n1>	<n>	Returns $-n1$
abs	<n1>	<n>	If $n > 0$ returns n , otherwise returns $-n$
minval		<n>	Returns the smallest value that can be represented as a <i>lucid</i> integer, which is <code>0x80000000</code> or <code>-2 147 483 648</code>

<code>maxval</code>		<code><n></code>	Returns the largest value that can be represented as a <i>lucid</i> integer, which is <code>0x7FFFFFFF</code> or <code>+2 147 483 647</code>
<code>random</code>		<code><n></code>	Returns a random integer in the range <code>maxval</code> to <code>minval</code>
<code>randseed</code>	<code><n1></code>	<code><n></code>	Seeds random number generator with <code>n1</code>
<code>band</code>	<code><n1> <n2></code>	<code><n></code>	Returns the bitwise <code>and</code> of <code>n1</code> and <code>n2</code>
<code>bor</code>	<code><n1> <n2></code>	<code><n></code>	Returns the bitwise <code>or</code> of <code>n1</code> and <code>n2</code>
<code>bxor</code>	<code><n1> <n2></code>	<code><n></code>	Returns the bitwise <code>xor</code> of <code>n1</code> and <code>n2</code>
<code>bnot</code>	<code><n1></code>	<code><n></code>	Returns the bitwise complement of <code>n1</code>
<code>bshl</code>	<code><n1> <n2></code>	<code><n></code>	Returns <code>n1</code> shifted left by <code>n2</code> bits
<code>bshr</code>	<code><n1> <n2></code>	<code><n></code>	Returns <code>n1</code> shifted right by <code>n2</code> bits
<code>brotl</code>	<code><n1> <n2></code>	<code><n></code>	Returns <code>n1</code> rotated left by <code>n2</code> bits
<code>brotr</code>	<code><n1> <n2></code>	<code><n></code>	Returns <code>n1</code> rotated right by <code>n2</code> bits

The integer arithmetic facilities allow readings to be processed before they are written to a file. In this way sensor offsets and/or gains can be readily incorporated into a measured value before it is written to the logger data file. Suppose, for example that the microvolt output from a given sensor has to be multiplied by 5, and then have an offset of 130 subtracted from it before it is recorded. This is quickly achieved using the integer `mul` and `sub` instructions. The expression:

```
mul ain 1 5
```

provides the scaled value. The expression

```
sub mul ain 1 5 130
```

subtracts 130 from this, and so the line of code:

```
write sub mul ain 1 5 130
```

writes the required value to the output file.

Integer arithmetic is fast, and precise. The penalty paid for this is that the range of numbers which can be stored is relatively small, approximately $\pm 2\,000\,000\,000$. It is the user's responsibility to ensure that integer calculations do not overflow this range *at any stage in their evaluation*. In the example above this is simple. To retain maximum accuracy analogue inputs to the logger should be in the range $\pm 512\,000\ \mu\text{V}$, but values up to $\pm 645\,000\ \mu\text{V}$ can be registered as the converter goes over range. The largest value which `ain 1` can return is 645 000, and the scaling and offset calculation shown can never produce a result which exceeds the integer size limit. However, there are other situations where it is not so simple to be absolutely confident that a calculation will not overflow, for example when a polynomial is evaluated to linearise a sensor. Then it is much simpler to use floating point arithmetic.

7.7 Floating point arithmetic

Command	Argument(s)	Return	Description
fadd	<f1> <f2>	<f>	Returns f1 + f2
fsub	<f1> <f2>	<f>	Returns f1 – f2
fmul	<f1> <f2>	<f>	Returns f1 x f2
fdiv	<f1> <f2>	<f>	Returns f1 / f2
fmod	<f1> <f2>	<f>	Returns f1 mod f2
fmax	<f1> <f2>	<f>	If f1 > f2 returns f1, otherwise returns f2
fmin	<f1> <f2>	<f>	If f1 < f2 returns f1, otherwise returns f2
fneg	<f1>	<f>	Returns -f1
fabs	<f1>	<f>	If f1 > 0 returns f1, otherwise -f1
fpwr	<f1> <n2>	<f>	Returns f1 raised to the integer power n2
fminval		<f>	Returns the smallest value that can be represented as a floating point number
fmaxval		<f>	Returns the largest value that can be represented as a floating point number
frandom		<f>	Returns a random number in the range 0 to 1. frandom may be seeded using the integer randseed instruction
fsqrt	<f1>	<f>	If f1 > 0 returns the square root of f1, otherwise 0
fln	<f1>	<f>	If f1 > 0 returns the natural logarithm of f1, otherwise returns most negative number that can be represented (fneg fmaxval).
fexp	<f1>	<f>	Returns e raised to the power f1.
fe		<f>	Returns the value of e: 2.71828
fsin	<f1>	<f>	Returns the trigonometric sine of f1. f1 is assumed to be in radians.
fcos	<f1>	<f>	Returns the trigonometric cosine of f1. f1 is assumed to be in radians.
ftan	<f1>	<f>	Returns the trigonometric tangent of f1. f1 is assumed to be in radians.
fasin	<f1>	<f>	If f1 < 1 returns the inverse sine of f1, otherwise returns $\pi/2$. Result is in radians.
facos	<f1>	<f>	If f1 < 1 returns the inverse cosine of f1, otherwise returns 0. Result is in radians

			returns 0. Result is in radians.
<code>fatan</code>	<f1>	<f>	Returns the inverse tangent of f1 in radians.
<code>fpi</code>		<f>	Returns the value of π : 3.14159
<code>fquadr</code>	<f1> <f2> <f3> <f4>	<f>	Returns $f1 + f2 f4 + f3 f4^2$
<code>fcubic</code>	<f1> <f2> <f3> <f4> <f5>	<f>	Returns $f1 + f2 f5 + f3 f5^2 + f4 f5^3$
<code>fquart</code>	<f1> <f2> <f3> <f4> <f5> <f6>	<f>	Returns $f1 + f2 f6 + f3 f6^2 + f4 f6^3 + f5 f6^4$
<code>fcmt2c</code>	<n1>	<f>	Converts n1, an analogue reading in microvolts, to a temperature in °C in the range -25 to 100 using the linearisation curve for a 10k Ω curve matched thermistor in series with a 10k Ω resistor.
<code>fpt2c</code>	<n1>	<f>	Converts n1, an integer reading in microvolts from a 100 Ω PRT, to floating point temperature in °C
<code>fpt1k2c</code>	<n1>	<f>	Converts n1, an integer reading in microvolts from a 1k Ω PRT, to floating point temperature in °C

Floating point arithmetic goes some way to avoid the problems of overflow which may occur when using integer arithmetic. It also has the advantage that, because it can represent numbers which have a fractional part, it can produce results in commonly used units. The price paid for these advantages is that the precision of integer arithmetic is lost. The floating point number system used in the Etherlog has a precision of about 7 decimal digits, and any calculation which requires more than this must be regarded as approximate.

Consider the following simple example of unit conversion. We have already seen that the `lin10k` function returns its result as an integer in °C/100. This can easily be converted to °C by using the floating point `fdiv` instruction to divide it by 100, returning a floating point result:

```
write fdiv lin10k ain 1 100
```

Temperature measurement and linearisation has been enhanced with the addition of `fcmt2c`, `fpt2c` and `fpt1k2c` commands, which each take an integer value representing the voltage (in μV) across a curve-matched 10k Ω thermistor, 100 Ω PRT or 1k Ω PRT respectively (assumed to be connected in an appropriate resistor network to V_{ref}) and return a linearised temperature in °C in floating point format. This avoids the need for calculations like the previous example, which becomes:

```
write fcmt2c ain 1
```

The floating point arithmetic instructions can be used to apply more complex processing to a reading. Consider a humidity sensor which provides a millivolt output, but which must be temperature compensated using the equation:

$$\text{humidity} = \text{sensor output voltage} / (1.0546 - 0.00216 \times \text{temperature})$$

Suppose that the humidity sensor is connected to channel 2, and the temperature sensor to channel 1. We have already seen that `fcmt2c ain 1` returns the temperature in °C. The expression

```
fmul 0.00216 fcmt2c ain 1
```

therefore returns $0.00216 \times \text{temperature in } ^\circ\text{C}$, and

```
fsub 1.0546 fmul 0.00216 fcmt2c ain 1
```

returns $1.0546 - 0.00216 \times \text{temperature}$, the compensation term we require. Obtaining the reading of channel 2 in millivolts is simple:

```
fdiv ain 2 1000
```

and dividing this by the temperature compensation factor gives the result required. This program line takes the two readings, does the arithmetic, and writes the result to the data file:

```
write fdiv fdiv ain 2 1000 fsub 1.0546 fmul 0.00216 fcmt2c ain 1
```

At first, such convoluted code might seem daunting. In the next section we see how the calculation can be broken down into more manageable stages by using variables to store intermediate results.

The floating point functions can also be used to linearise sensors. For example, many flow measurement devices produce an output which is proportional to the square of the flow. If the flow is to be accumulated between recording intervals then the sensor output must be linearised before it is added into the running total: accumulating the sensor outputs directly and carrying out the square root operation later will give results which are seriously in error. Consider a combination of a flow grid and pressure transducer, which gives a voltage output in microvolts such that:

$$\text{Flow (m}^3/\text{s)} = 0.015 \times \sqrt{\text{Output voltage } (\mu\text{V})}$$

The following line of *lucid* reads the sensor (assuming it is attached to analogue input 1), linearises it using the square root function. The value could then be added to a running total held in a *lucid* variable, as described in the next section:

```
fmul 0.015 fsqrt ain 1
```

The *fexp* and *fln* functions allow readings to be expanded or compressed. Since *fexp* doubles as the inverse natural logarithm the two functions can be used in tandem to raise a number to an arbitrary power. For example, suppose that in order to convert to engineering units the μV output of a sensor connected to analogue input 1 has to be raised to the power 1.6. Recall that to raise a number to a power you must first take its logarithm, then multiply by the power, then take the inverse logarithm:

```
fexp fmul 1.6 fln ain 1
```

reads the output of the sensor and performs the desired manipulation.

The trigonometric functions also facilitate non-linear signal processing on-line. Applications might include resolving wind velocity measurements into components, or converting a measured angular displacement into a distance.

7.8 Variables

Command	Argument(s)	Return	Description
setvar	<n1> <a2>	<a>	Set local variable n1 (n1 in the range 1 to 16) to a2, returning the value and type of a2.
var	<n1>	<a>	Returns the value and type of local variable n1 (n1 in the range 1 to 16) or 0 if n1 is outside this range

When processing data it is often necessary to store intermediate values; each *lucid* task has 16 local variables associated with it for this purpose. A value is stored in variable n using the `setvar n` command, and may then be accessed at any time using the `var n` command

One use of a variable is to ‘snapshot’ a value during execution of a task. In the floating point arithmetic example above the reading of one channel was used to compensate another. It may be that the value of the first channel is also to be recorded. To this end it can be held temporarily in a variable:

```
repeat 10
begintask

# store temperature in deg C in variable 1
setvar 1 fcmt2c ain 1

# write time, temperature and compensated humidity to data file
newline
write time
write var 1
write fdiv fdiv ain 2 1000 fsub 1.0546 fmul 0.00216 var 1
endtask
```

As well as ensuring that the temperature recorded to the output file is exactly the same as the one used to compensate the humidity reading, the use of the variable has the effect of breaking up the code, and making it easier to read and debug. If desired, this process can be taken further, and the calculation of the compensation factor can be broken down into stages:

```
repeat 10
begintask

# store temperature in deg C in variable 1
setvar 1 fcmt2c ain 1

# now write time and temperature to output file
newline
write time
write var 1

# generate temperature compensation factor in variable 1
setvar 1 fmul 0.00216 var 1
setvar 1 fsub 1.0546 var 1

# write compensated humidity reading to output file
write fdiv fdiv ain 2 1000 var 1
endtask
```

As well temporarily storing a value as a task is executed, variables can store information between successive executions of tasks. This opens up the opportunity to carry out signal processing operations on values recorded over time. This facility is demonstrated in Section 7.10.

The most normal use of the `setvar` command is simply to place a value into a variable. However, like almost all other *lucid* commands, `setvar` returns a result, the value stored. In some cases this can be exploited to provide benefits in storage requirements and/or execution speed. In the companion to this manual there is an example in which a variable is used to store the value of a pulse counter in order that the incremental value can be evaluated when the counter is next examined. This was done as follows:

```
# on initialisation copy the starting pulse total into variable 1
setvar 1 pulse 1

repeat 10

begintask

# create a new record in the datafile and write the timestamp
newline
write time

# capture the current cumulative pulse count in variable 2
setvar 2 pulse 1

# write the change in the pulse total since last time
write sub var 2 var 1

# and update the old total
setvar 1 var 2
endtask
```

In fact the program can be streamlined quite considerably by exploiting the fact that when the value of pulse counter 1 is captured into a variable the value of that counter is returned, and can be used in further calculations, in this case the subtraction to generate the incremental count. Consider the code:

```
write neg sub var 1 setvar 1 pulse 1
```

The subtraction command takes the value of variable 1 as its first argument. It then updates variable 1 with the current value of pulse counter 1, and the result of this update (the value of the pulse counter) is passed back as the second argument for the subtraction. The order of arguments is critical here: variable 1 must appear as an argument before it is updated, or the subtraction will always return zero. Placing the arguments in this order actually always returns a non-positive result, the previous count minus the new one. However, this is quickly rectified using the `neg` function before the result is written to the output file. The logging program becomes:

```
# on initialisation copy the starting pulse total into variable 1
setvar 1 pulse 1

repeat 10

begintask

# create a new record in the datafile and write the timestamp
newline
write time

# write the change in the pulse total since last time and update
write neg sub var 1 setvar 1 pulse 1
endtask
```

This gives shorter, more efficient code, and also frees up variable 2 for another job.

7.9 Repeated and conditional execution

Command and arguments	Return	Description
<pre>loop <n1> [list1] endloop</pre>	<a>	Perform the commands in list1 n1 times. Return the last performed evaluation result/type.
<pre>if <n1> list1 [else list2] endif</pre>	<a>	If <n> is non-zero, then execute the commands represented by list1. If not, then if there is an 'else' keyword, execute the commands represented by list2. Return the last performed evaluation result/type.
<pre>while <n1> [list1] endwhile</pre>	<a>	Perform the commands in list as long as n1 evaluates to non-zero. Return the last performed evaluation result/type.

The `loop` instruction allows a list of instructions to be repeated a given number of times. For example an input can be read many times and its average determined. This is a good way of eliminating noise:

```
# initialise variable 1 to zero
setvar 1 0

# take sixteen readings and total them in variable 1
loop 16
  setvar 1 add var 1 lin10k ain 1
endloop

# write the average reading to the current output file
write div var 1 16
```

It is also possible to use the built-in accumulator variables to generate averages. This is described in Section 7.14. The conditional execution commands are almost always used in conjunction with comparison functions and examples of their use are deferred until those instructions have been described.

7.10 Integer comparison operations

Command	Argument(s)	Return	Description
<	<n1><n2>	<n>	If n1 < n2 returns 1 (TRUE) otherwise returns 0 (FALSE)
<=	<n1><n2>	<n>	If n1 <= n2 returns 1 (TRUE) otherwise returns 0 (FALSE)
=	<n1><n2>	<n>	If n1 = n2 returns 1 (TRUE) otherwise returns 0 (FALSE)

>=	<n1><n2>	<n>	If n1 >= n2 returns 1 (TRUE) otherwise returns 0 (FALSE)
>	<n1><n2>	<n>	If n1 > n2 returns 1 (TRUE) otherwise returns 0 (FALSE)

All integer comparison operations in *lucid* return a value of type integer. A value of zero is used to denote FALSE, that is it is returned if a comparison condition is not fulfilled. If the comparison is fulfilled the operators in the table return 1, although any non-zero value will be interpreted as TRUE by the conditional and Boolean operators in *lucid*.

Used in conjunction with the `if` and `while` commands, the comparison functions can be used to log data conditionally, depending on, for example, the reading on a given channel. As an example suppose that the voltage on analogue input 2 is to be recorded only when the temperature measured on analogue input 1 is above 25°C. The following program code uses the `>` function to determine whether this is the case (recall that the `lin10k` function returns the temperature in °C/100):

```
repeat 10
begintask

# test to see if temperature on input 1 is greater than 25 deg C
if > lin10k ain 1 2500

# if so write a the time and the value of analogue input 2
newline
write time
write ain 2
endif
endtask
```

The integer comparison instructions can also be used in conjunction with the `time` function. Together with the `if` function, this can be used to allow part of a task to be executed only intermittently. This is commonly used when a task is reading a value frequently, but recording a summary of those readings less frequently. For example, suppose that a task executes every ten seconds, but that part of it is only required to run every five minutes. The code below reads the time, and calculates how far past the most recent five minute boundary it is:

```
mod time 300
```

Even when a task is scheduled to run on the five minute boundary there is no guarantee that this quantity will evaluate to exactly zero. There may have been time consuming code to carry out before this instruction was reached, and the clock may have advanced while this code was executing. Alternatively there may be other tasks running which also executed on the five minute boundary and therefore delayed the execution of this task. However, if the task is running at an interval of once a minute then the code below, which checks that we are less than 30 seconds past the boundary when the tasks which are to be executed every five minutes are due to be run, will ensure that they are executed correctly:

```
if < mod time 300 30

list of tasks for execution every five minutes

endif
```

The example below shows how to use this technique to write the maximum and minimum of a measurement made every minute to a file once every five minutes:

```
# initialise the variables to be used to hold min and max
setvar 2 maxval
```

```

setvar 3 minval

repeat 60
begintask

# read linearised value of analogue channel 1 into variable 1
setvar 1 lin10k ain 1

# minimum is stored in variable 2: update it
setvar 2 min var 1 var 2

# maximum is stored in variable 3: update it
setvar 3 max var 1 var 3

# if we are on the five minutes write data to file
if < mod time 300 30

# write timestamp
newline
write time

# write minimum and maximum
write var 2
write var 3

# and reset minimum and maximum
setvar 2 maxval
setvar 3 minval
endif
endtask

```

The `while` command also examines whether its argument is `TRUE` or `FALSE` before it decides whether to execute a series of commands. However, unlike the `if` construct, if the argument is `TRUE` those commands are executed repeatedly until the argument becomes `FALSE`. Suppose that a sensor has to be powered from digital output 4 using 50ms pulses until its temperature, measured using analogue input 1, has reached 25°C:

```

# pulse the sensor while its temperature is below 25C
while < lin10k ain 1 2500

# apply 50ms pulse and then wait 50ms
dset 4
wait 5
dclr 4
wait 5
endwhile

```

The `while` command also provides the facility to put a *lucid* program into an endless loop, from which it never emerges. In the example above the power supply could become disconnected from the sensor, and the program would then ‘hang’ forever, as it tried in vain to heat it up. One way of insuring against this sort of problem is to make a note of the time at which the program started, and then to abandon the attempt if it has not succeeded within a given interval. Implementing this sort of ‘timeout’ behaviour will be demonstrated in Section 7.12.

The `while` instruction can also be used to create your own loops. This is useful when you want to access the value of the loop counter, a facility not available when the *lucid* `loop` command is used. Depending on the type of sensor attached to the logger digital inputs it may be necessary to read all eight inputs, and generate a result in the range 0 to 255. The following code uses a loop constructed with the `while` command in conjunction with the binary shift left instruction described in Section 7.6 to achieve this. Variable 1 is used to build up the result, with variable 2 acting as a loop counter:

```

# initialise accumulated value to 0 and loop counter to 1
setvar 1 0
setvar 2 1

```

```

# go through this code for inputs 1 to 8
while <= var 2 8

    # shift accumulated value left ready for next value
    setvar 1 bshl var 1 1

    # read digital input and add to accumulated value
    setvar 1 add var 1 din var 2

    # increment the loop counter
    setvar 2 add var 2 1

endwhile

```

7.11 Floating point comparison operations

Command	Argument(s)	Return	Description
f<	<f1><f2>	<n>	if f1 < f2 returns 1 (TRUE), otherwise returns 0 (FALSE).
f>	<f1><f2>	<n>	if f1 > f2 returns 1 (TRUE), otherwise returns 0 (FALSE).

The floating point comparison operators follow closely their integer counterparts, and are applied in a similar way. However, due to the fact that the floating point representation is not generally precise, it is inadvisable to search for exact equality between two numbers. For this reason the f<=, f= and f>= operators are not defined in *lucid*.

7.12 Boolean operations

Command	Argument(s)	Return	Description
and	<n1><n2>	<n>	If n1 <> 0 (TRUE) and n2 <> 0 (TRUE) returns 1 (TRUE) otherwise returns 0 (FALSE)
or	<n1><n2>	<n>	If n1 <> 0 (TRUE) or n2 <> 0 (TRUE) returns 1 (TRUE) otherwise returns 0 (FALSE)
xor	<n1><n2>	<n>	If n1 and n2 are both TRUE or both FALSE returns 0 (FALSE), otherwise returns 1 (TRUE).
not	<n1>	<n>	If n1 <> 0 (TRUE) returns 0 (FALSE) otherwise returns 1 (TRUE)

The Boolean operators allow the results of comparisons to be combined. For example it may be that the value of analogue input 2 is to be written to the output file only if the temperature measured on input 1 is above the threshold of 25°C *and* digital input 1 is set. This can be achieved by using the *and* function to combine the results of two separate comparisons:

```

repeat 10
begintask

# test to see if temperature is above 25 and din 1 is set
if and > lin10k ain 1 2500 = din 1 1

# if so, write a new record consisting of timestamp and ain 2
newline
write time
write ain 2
endif
endtask

```

In fact the `din` function returns a 1 or 0, which can be treated as TRUE or FALSE, removing the need for the second comparison operation:

```

repeat 10
begintask

# test to see if temperature is above 25 and din 1 is set
if and > lin10k ain 1 2500 din 1
newline
write time
write var 1
endif
endtask

```

The Boolean functions can also be used in conjunction with comparison operations on the current time to cause events to occur during selected periods of the day. The expression:

```
setvar 1 mod time 86400
```

places a number representing the number of seconds since midnight in variable 1. Suppose that a certain operation is to be executed only between 6:00am (21 600 seconds past midnight) and 6:00pm (64 800 seconds past midnight). The following code achieves this:

```

setvar 1 mod time 86400
if and >= var 1 21600 =< var 1 64800

    lucid instructions to be executed between 6:00am and 6:00pm
endif

```

When an entire task is to be executed only during certain periods of the day the `window` command, described in Section 7.2, provides a much more convenient way of achieving this goal.

In Section 7.10 we described an application in which it was necessary to warm up a sensor before taking a reading. It was noted that if the heater became disconnected the program would hang indefinitely as it tried to warm up the sensor. The example below solves this problem. It only heats the sensor while its temperature is below 25°C and less than five seconds has elapsed:

```

# note the time at which sensor heating started by copying time into t
sett

# pulse the sensor while its temperature is below 25C
# or until 5 seconds has elapsed (ie heating operation has failed)
while and < lin10k ain 1 2500 < time add t 5

# apply 50ms pulse and then wait 50ms
dset 4
wait 5
dclr 4
wait 5
endwhile

```

The next part of the program should, of course, check that the sensor has reached the required operating temperature before measuring its output.

Finally, note that the Boolean operations described in this section do not necessarily give the same results as the bitwise versions listed in Section 7.6. For example the bitwise `and` of 3 and 12 is given by:

$$\begin{array}{rcl}
 3 & = & \dots 0011 \\
 12 & = & \dots 1100 \\
 \text{Bitwise and: } & \overline{\dots 0000} & = & 0
 \end{array}$$

and the *lucid* code:

```
band 3 12
```

does indeed return the value 0. However, both 3 and 12 are non-zero, and hence are interpreted by *lucid* as TRUE. Thus the code:

```
and 3 12
```

returns the value 1 (TRUE). It's important to make sure that you use the right kind of operator.

7.13 Type conversion

Command	Argument(s)	Return	Description
<code>float</code>	<code><a1></code>	<code><f></code>	returns a variable of floating point type regardless of type of argument
<code>int</code>	<code><a1></code>	<code><n></code>	returns a variable of integer type regardless of type of argument
<code>timet</code>	<code><a1></code>	<code><t></code>	returns a variable of time type regardless of type of argument
<code>floatx</code>	<code><a1></code>	<code><f></code>	returns a variable of floating point type without carrying out the actual conversion
<code>intx</code>	<code><a1></code>	<code><n></code>	returns a variable of integer type without carrying out the actual conversion
<code>timetx</code>	<code><a1></code>	<code><t></code>	returns a variable of time type without carrying out the actual conversion

In general, the automatic type conversion which all *lucid* functions perform on their arguments means that it is seldom necessary for the programmer explicitly to force type conversion. One area in which the facility is useful, however, is in controlling the way in which data is written to an output file. Writing a small integer value to the output file uses considerably less space than writing a floating point number. Suppose that after signal processing a value is held in local variable 1 as a floating point number in the range -1 to +1. Writing this directly to the output file using the code:

```
write var 1
```

will require 5 bytes of disk space. However, if the value of var 1 is first converted to an integer in the range -10 000 to 10 000 it can be stored using only 3 bytes:

```
write int fmul var 1 10000
```

You might consider it more simple to use an integer multiplication to achieve the scaling and type conversion in a single operation:

```
write mul var 1 10000
```

Unfortunately the integer multiplication instruction carries out its type conversion *before* it performs the multiplication. The contents of variable 1 will be converted to -1, 0 or +1, and the value written to the output file will be either -10000, 0 or 10000, not quite the result intended !

The facility to convert variables to the time type opens up the possibility of performing arithmetic on the time, and then writing the result as a time. For example, suppose that every hour the maximum and minimum values of an input over the preceding hour are being written to the output file. If this is preceded by a simple:

```
write time
```

instruction then the time associated with the values calculated recorded between say 9:00am and 10:00am will be 10:00am. Sometimes, however, it might be preferable to associate the time 9:30 with this value (so called time-centering). This can easily be achieved using the `timet` conversion function:

```
write timet sub time 1800
```

Here the `sub` command is relied upon to automatically convert the current time from type `time` to type `integer`. It then deducts 1800 seconds from it. The `timet` function then forces conversion back to type `time` before the value is written to the output file.

It is important to remember that numbers stored in floating point format may be outside the range of the integer or time types. When converting from floating point it is the programmer's responsibility to ensure that the result will fit into the destination type.

The `floatx`, `intx` and `timetx` functions change the type of a variable without converting its value. For example, if the `intx` instruction is used on a floating point number it returns the string of ones and zeroes which represent that number in an integer. This could be used, for example, to extract the exponent of a floating point number (held in bits 23 to 30 when the number is stored):

```
# store exponent of floating point number in var 1 in var 2
setvar 2 bshr band intx var 1 0x7F800000 23
```

7.14 Accumulator operations

Command	Argument(s)	Return	Description
<code>areset</code>	<n1>	<n>	Clear total and count of accumulator n1 (n1 in range 1 to 8); return 1, or 0 if n1 is invalid
<code>accum</code>	<n1> <a2>	<a>	Add value of a2 to accumulator n1 (n1 in range 1 to 8), incrementing count of accumulator n1 and setting type of accumulator n1 to that of a2. If the initial type of accumulator n1 is different from

			that of a2, it is converted before the addition. Returns value and type of a2, or 0 if n1 is invalid.
account	<n1>	<n>	Return number of accumulations into accumulator n1 (n1 in range 1 to 8); or 0 if n1 is invalid
atotal	<n1>	<a>	Return total accumulated sum in accumulator n1 (n1 in range 1 to 8), or 0 if n is invalid. Result is of same type as most recently accumulated value.
avge	<n1>	<a>	Return the average of the results accumulated to accumulator n1, ie atotal(n1)/account(n1) (n1 in range 1 to 8), or 0 if n1 is invalid or account(n1) is zero Result is of same type as most recently accumulated value.

It is frequently desirable to be able to incorporate values into an average as they are measured. We have already seen that it is be quite straightforward to accomplish this using the variables associated with a task to hold a running total of a given quantity. In some situations it may not be immediately obvious how many readings have been incorporated in that total, and it will be necessary to use a second variable to record the number of entries made, and update it each time a value is incorporated into the average. Generating averages is a common requirement, and so *lucid* provides special facilities. This results in simpler programs, and also frees up local variables for other jobs.

The simple example below reads the temperature sensor connected to analogue input 1 every ten seconds, and writes the average of the readings to an output file every five minutes:

```
repeat 10

# initialise the accumulator to be used
areset 1
begintask

# add linearised value of analogue input 1 into accumulator 1
# note this also increments the counter associated with accumulator 1
accum 1 lin10k ain 1

# if we are on the five minutes write data to file
if < mod time 300 5

# write timestamp
newline
write time

# write average to output file
write avge 1

# reset accumulator total and count
areset 1
endif
endtask
```

If, as above, a series of integers is accumulated the accumulator will remain of type integer. When the average is calculated it will inherit the type of the accumulator, and thus also will be an integer. This may give an unexpected result. For example the code:

```
# reset accumulator 1
areset 1

# accumulate three integer values
accum 1 3
accum 1 4
accum 1 3

# write their average to the output file
write avge 1
```

writes the integer 3 (obtained from the calculation `div 10 3`) to the output file. If you wished to obtain a floating point result you should force the accumulator to be of type floating point by ensuring that at least the last value accumulated is of type floating point, forcing conversion of the accumulator to floating point:

```
# reset accumulator 1
areset 1

# accumulate two integer values and a floating point number
accum 1 3
accum 1 4
accum 1 float 3

# write their average to the output file
write avge 1
```

This program writes the floating point number 3.333333 to the output file. In more realistic examples, in which the values are being accumulated within a loop, it will probably be simpler just to float every value before accumulating it if a floating point result is required.

If averages are not being generated it is possible to use the accumulators as additional variables, increasing the storage facilities available in a task from 16 to 24 values. The following program segment stores the linearised value of analogue input 1 in the first accumulator:

```
areset 1
accum 1 lin10k ain 1
```

The result can be accessed using either the `atotal 1` or `avge 1` instruction - only one reading has been accumulated and the two instructions are equivalent. Using `atotal 1` is more efficient, as it saves the unnecessary division by 1. The stored value can be written to the current output file using the instruction:

```
write atotal 1
```

The next time you want to store a value in accumulator 1 you must remember to first reset it with the `areset 1` command.

7.15 Input file functions

Command	Argument(s)	Return	Description
login	<name>	<n>	Sets the name of the input file to name and returns 1.
nlogin	<n1>	<n>	Sets the name of the input file to fnnnnn.dat where "nnnnn" is the

			fnnnnn.dat where “nnnnn” is the decimal representation of the low-order 16 bits of n1. Returns 1.
read	<n1>	<a>	Reads value from offset n1 in the current input file. If value is not found returns integer 0.
readdef	<n1> <a2>	<a>	Read value from offset n1 in the current input file. If entry is not found returns a2.
insert	<n1> <a2>	<n>	Inserts the value <a2> in the current input file, at offset n1. Other entries are shuffled up. If offset is past the end of the file empty locations are filled with 0. Returns 1 if successful, otherwise 0.
replace	<n1> <a2>	<n>	Replaces value at offset n1 in current input file with a2. If offset is past the end of the file empty locations are filled with 0. Returns 1 if successful, otherwise 0.
delete	<n1>	<n>	Deletes the value at offset n1. Any entries beyond that point are shuffled down. Returns 1 if successful, otherwise 0.
append	<a1>	<n>	Appends <a1> to the end of the current input file. Returns 1 if successful, otherwise 0.

These functions are most commonly used to read calibration values from a file. For example, suppose that the scale of a sensor connected to analogue input 1 is the first entry in the current input file. The following code names the calibration data input file, reads the output of the sensor, reads the scale from the input file, calculates the result and then writes it to the current output file:

```
login sensor.cal
write fmul read 0 ain 1
```

It is important to note that the offset is numbered from zero. Thus `read 0` returns the first value in the file, `read 1` the second and so on. The `readdef` function can be used to check whether a file exists, by defaulting to a value which would normally never occur. For example, if a sensor scale factor is the first entry in a file and is always known to be positive then if the function:

```
readdef 0 -1
```

returns -1 it is clear that the file does not exist. This facility can be used to ensure that data is not lost simply because a calibration file is missing. If the above example is changed to:

```
write fmul readdef 0 -1 ain 1
```

then a missing calibration file will result in the sensor output being recorded in negative μV . Thus the reading is not lost, and can be appropriately scaled after the data has been collected from the logger.

For a description of how these functions can be used for other applications, including implementing lookup tables and establishing inter-task communication see our Application Note 4: Using the *lucid* input file capabilities.

7.16 Utility functions

Command	Argument(s)	Return	Description
getrpt		<n>	Return the current setting of the task's repeat interval
freespace		<n>	Return the number of bytes available in unallocated sectors for use in the file system
address		<n>	Returns the radio address of the logger
error			Forces an execution error and halts current execution of task
nosleep	<n1>	<n>	If n1 <> 0 then the logger is prevented from sleeping between logging tasks.
ticker		<n>	Returns the current value of the logger ticker in units of seconds/100
trace	<n1>	<n>	If n <> 0, enable log tracing, else disable it; return n
brkpt			Causes the program to pause until the user types <return>
sstepon			Causes the program to execute one step at a time, waiting for the user to type <return> each time
sstepoff			Cancels sstepon.

The `getrpt` command returns the current value of the task's repeat interval, in seconds. This can be useful where a time-dependent function, such as a filter with a fixed time constant, is required to operate in situations where the repeat interval can change. The following code illustrates how the filter constant for a simple first-order filter is calculated:

```
repeat 10
setvar 1 0

begintask
# calculate filter coeff: k = exp(- repeat_interval / time_constant)
setvar 2 fexp fneg fdiv getrpt 50

# output = (k x last_value) + ((1 - k) x new_value)
setvar 1 fadd fmul var 2 var 1 fmul fsub 1 var 2 ain 1

# test for different repeat interval, and set here as required
if din 1
repeat 5
```

```

    else
        repeat 10
    endif
endtask

```

This code applies a first-order low-pass filter to `ain 1` with a time constant of 50s independent of the sampling rate (which in this case is the repeat interval).

The `freespace` command returns the amount of filing space available for data storage. A typical use is to generate a radio alarm condition, as will be discussed in section 7.18.

The `address` command returns the radio address of the logger. It can be used to stamp a datafile with the logger identity:

```

write address

```

This can be done when the task is started, by placing the code in the initialisation section, or every time data is written to the file by placing it in the main body of the task. The `address` function can also be used to create programs which behave differently on different loggers. For example, suppose that in a given installation only the logger with address 35 has a particular sensor attached to analogue input 8. Then the code:

```

# test to see if this is the logger with the additional sensor on ain 8
if = address 35
    write ain 8
endif

```

will only write the additional value to the output file when it is running on logger 35. To simplify subsequent data processing it is probably advisable to place this piece of code after the other data writing operations. In that way the additional value will be added to the end of the line, and the layout of the other data values will be the same on every logger.

The `nosleep` command keeps the logger awake between tasks, and is most commonly used when pulses are to be counted on digital inputs 3 to 8. For example the following program counts pulses on all eight digital inputs, and records the cumulative count every half hour:

```

# logger must not be allowed to sleep between task executions
nosleep 1

repeat 1800

begintask

# create a new record in the datafile with a timestamp
newline
write time

# write the values of the eight pulse counters
write pulse 1
write pulse 2
write pulse 3
write pulse 4
write pulse 5
write pulse 6
write pulse 7
write pulse 8

endtask

```

In this case the same effect could have been achieved by issuing the command:

sleep disable

at the logger command line, or including it in the `startup.cmd` file. However, there may be situations in which, for example, pulses are only to be counted at certain times of day and the logger is to be allowed to sleep at other times. The following program records data from all pulse channels from 6:00am to 6:00pm, and allows the logger to sleep at other times:

```
repeat 1800
beginntask

  setvar 1 mod time 86400
  if and >= var 1 21600 <= var 1 64800

    # inside the recording interval - record the pulse counters
    newline
    write time
    write pulse 1
    write pulse 2
    write pulse 3
    write pulse 4
    write pulse 5
    write pulse 6
    write pulse 7
    write pulse 8

    # do not allow logger to sleep during recording period
    nosleep 1

  else

    # outside of recording period - allow logger to sleep
    nosleep 0
  endif
endntask
```

Note that the number of pulses accumulated on channels 3 to 8 during the period that the logger has been allowed to sleep will be undefined. Thus the data produced by the program should only be interpreted over the period 6:00am to 6:00pm. The change in cumulative pulse count between 6:00pm and 6:00am the following day cannot be interpreted as the number of pulses which arrived during that period - the logger may have missed some whilst it was sleeping.

The `ticker` command returns the current value of the logger's internal ticker, which increments every 1/100 second while the logger is awake. It can be used to measure the time it takes for a *lucid* task, or part thereof, to execute. This information may be of use in assessing the time for which a given task keeps the logger awake, and hence the power consumption and battery life on a given logging schedule. The following code measures the time taken to execute a main task:

```
beginntask
  # snapshot the ticker at the start of the task
  setvar 1 ticker

  main body of task goes here

  # calculate time taken to execute the task and record in the data file
  write sub ticker var 1
endntask
```

Naturally, the timing code itself takes some time to execute. This source of error can be eliminated by first running the program with no instructions between the timing commands. This evaluates the overhead associated with the timing instructions, which can then be subtracted from subsequent results with code in place.

You may also use the ticker function to time events occurring within a task. It is important to remember that, in its interpreted form, *lucid* is relatively slow, and that the resolution of any such measurement will be limited.

Finally, always remember that the logger ticker is only incremented while the logger is awake. It can therefore only be used to time events whilst a task is executing, and not between tasks. For measuring the time between events occurring in different executions of a task the `time` function should be used.

The `error` function forces the logger to abandon the current execution of a task. An error message will appear in the current output file, and the task will be executed again when it is next due. This results in a data file which is not 'rectangular', and in general it is better to respond to an error condition by writing a reserved numerical value to the datafile, thus retaining its normal format.

The `trace`, `brkpt` and `sstep` commands are used to control the output of trace information and to control the execution of a program during debugging. Their use is described in detail in the next chapter.

7.17 Macro support functions

Command	Argument(s)	Return	Description
<code>eval</code>	<code><name></code>	<code><a></code>	Evaluate the macro in filename <code><name></code> . Returns whatever the macro returns
<code>units</code>	<code><name></code>	<code><a></code>	Set engineering units text string to <code><name></code> when macro is evaluated from the command prompt; no effect when macro is evaluated from a task. Return value and type are from the last expression evaluated before <code>units</code> command
<code>text</code>	<code><name></code>	<code><name></code>	Set return value to <code><name></code> when macro is evaluated from the command prompt
<code>arg</code>	<code><n1></code>	<code><a></code>	Return value and type of arg <code><n1></code> (in range 1..8) if <code><n1></code> is valid, else integer 0
<code>setarg</code>	<code><n1><a2></code>	<code><a></code>	Set arg <code><n1></code> (in range 1..8) to value and type of <code><a2></code> , returning <code><a2></code> if <code><n1></code> valid, else integer 0

The macro mechanism in the Etherlog provides the capability to run sequences of *lucid* commands both from tasks and in interactive mode. This permits blocks of code to be written and tested interactively, then used without modification from any task. An example macro might read battery voltage, on the assumption that it is connected to analogue input 8 *via* a resistor divider, and return a floating point value in volts:

```
macro
  fmul ain 8 3.125e-5      # scale uV reading to equivalent battery V
endmacro
```

Using the `units` command, an interactively-evaluated macro can return a text string representing engineering units which will be displayed after the macro's return value. Note that this string is limited to 14 characters and may not contain any white space characters. For example:

```
macro
  units Volts
  fmul ain 8 3.125e-5
endmacro
```

It may sometimes be preferable to display text instead of a numeric return value. The example **pump** macro below displays “running” or “stopped” depending on the state of digital input 1:

```
macro
  if din 1
    text running
  else
    text stopped
  endif
endmacro
```

Once a macro has been tested and works, it can be used by any task to get the current battery voltage, as in the following example. It is assumed that the battery voltage macro is stored in a file called **battery**:

```
# check battery voltage and take appropriate action
if f< eval battery 7.5
  lucid code to execute when battery < 7.5V
endif
```

When evaluated from a task, the `units` command has no effect. The `text` command is intended only for use in interactive mode, and should not be used to return a text value to a task.

In many cases a macro needs input values in order to be more useful, as might be the case where several identical temperature sensors are fitted to analogue inputs and one macro could be used to measure temperature on a channel identified by an argument. The following macro illustrates this:

```
# macro temp: arg 1 = number of analogue channel to read
macro
  if and >= arg 1 1 <= arg 1 4 # validate channel no. in arg 1
    fcmt2c ain arg 1
  else
    100.0
  endif
endmacro
```

This macro expects the first `arg` to identify the analogue input channel to measure. If valid, a linearised thermistor temperature is returned in °C, otherwise a value of 100 is returned. When evaluated from *winhost* using the Interactive View derived data values facility, or from the command line by typing:

```
eval temp 2
```

the Etherlog reads the first argument to the `temp` macro into `arg 1`, then invokes the `temp` macro. This in turn expects, and finds, the channel number in `arg 1`. If the macro required three arguments, they would be read in turn into `arg 1`, `arg 2` and `arg 3`. Macros may take up to 8 arguments.

When calling a macro that takes arguments from within a task, use the `setarg` command to set up the arguments for the macro, as follows:

```
# read temp from channel 3 into var 1
setarg 1 3
setvar 1 eval temp
```

Note the important distinction between the command line and task invocations of the macro: when called from a task, it is the responsibility of the task to set up the args before calling the macro.

7.18 Communications support functions

Command	Argument(s)	Return	Description
setalarm	<n1>	<n>	Set the radio alarm code to n1 (n1 in the range 0 to 65 535), and return its value
getalarm		<n>	Return the current value of the radio alarm code
radio		<n>	Send a radio advertisement; return 1
modem		<n>	Try to establish a modem connection; return 1 if trying to connect, 0 if failed
mdmconn		<n>	Return 1 if modem is off-hook (dialling, connected or disconnecting) or 0 if on-hook (disconnected)
sms	<name>	<n>	Try to send an SMS message <i>via</i> GSM modem using control file <name>; return 1 if message sent to network, 0 if failed

7.18.1 Radio alarm codes

The `setalarm` function allows you to set (and clear) the radio alarm level associated with a radio connection, and this can modify the automated behaviour of the host PC software. For example, the code below sets the logger radio alarm level to 1 if there is less than 2000 bytes of space remaining in the filing system:

```
# set a radio alarm code if filing space is low
if < freespace 2000
    setalarm 1
endif
```

If this code is installed then when the filing space falls below 2000 bytes subsequent radio advertisements will have an alarm level of 1 associated with them. In this case, the logger relies on whoever intervenes (human or software) to clear more space, and also to reset the alarm level to zero when they have finished. This could also be done by the logger itself:

```
if < freespace 2000
    # set a radio alarm if filing space is low
    setalarm 1
else
    # clear the alarm if space is no longer low
    setalarm 0
endif
```

The radio alarm code can be set to any value between 0 and 65 535 (it is a 16-bit number), allowing different codes to be used to signal different alarm conditions. The host PC software currently provides the facility to prioritise alarms, and *lucid* can readily be used to set any alarm level required in response to different alarm conditions. As discussed in Chapter 6, the radio alarm code is a global resource, and if several tasks running simultaneously are to have access to it a way of generating independent alarm codes must be devised. The most obvious way to do this is to allocate separate

bits to separate alarm conditions. The code below uses the integer bitwise or and and commands to set or clear bit 4 of the alarm code depending on the amount of filing space available:

```
# set radio alarm bit 4 if filing space is low
if < freespace 2000
    setalarm bor getalarm 0x0010
else

    # clear alarm bit 4 if space is no longer low
    setalarm band getalarm 0xFFEF
endif
```

7.18.2 Radio advertisements

The `radio` command instructs the logger to send out a radio advertisement immediately, in addition to any advertisements that might already be going out as a result of the background radio probe interval. This feature is commonly used in conjunction with the `window` function to send out radio advertisements more frequently during selected periods of the day. This guarantees rapid connection when required, but conserves power at other times. The following simple task, which would normally be run in addition to any logging tasks, achieves this, sending out radio advertisements at 60 second intervals between 9:00am and 12:00noon, and 2:00 and 5:00pm:

```
# set up logging window
window 1 9:00 12:00
window 2 14:00 17:00

repeat 60

beginntask

    # all this task does is send out a radio advertisement
    radio

endntask
```

At other times the interval between advertisements can be increased to, say, ten minutes by including the command:

```
radio probe 600
```

in the logger `startup.cmd` file.

The ability to send out a radio advertisement from within a *lucid* program can also be useful if an alarm condition occurs when, in order to conserve power, the logger is sending out advertisements relatively infrequently. For example, using task windowing advertisements may be sent out once a minute during the day, but only once every ten minutes at night. If an alarm condition occurs at night it could be up to ten minutes before the host PC learns of it. The code below solves this problem by sending out a radio advertisement as soon as the alarm condition is detected, and on subsequent executions of the task until the condition is cleared.

```
# set a radio alarm code and advertise if filing space is low
if < freespace 2000
    setalarm 1
    radio
endif
```

7.18.3 Modem connections

The `modem` command is used to establish an outgoing modem connection from the logger to a user's computer when a conventional or GSM modem is connected to the serial port. It is assumed that the

modem initialisation and dial strings have been correctly configured in the logger, and that the modem interface has been enabled. The command returns the value 1 if the dial string was successfully sent, otherwise 0.

In some cases, it may be desirable to turn on the power supply to a GSM modem for limited periods in order to reduce the logger's power consumption. The task shown below turns on the modem for the first 5 minutes of each hour by means of an external switch connected to digital output 4:

```
repeat 60
beginntask
  if < mod time 3600 300
    dset 4      # inside first 5 minutes of hour: turn GSM on
  else
    dclr 4      # else turn it off
  endif
endntask
```

This works well until a user calls the logger during the first 5 minutes, but does not complete (say) downloading data by the end of this period. The task above pays no regard to whether anyone is using the logger, and just switches off the modem: not very user-friendly behaviour.

The problem is rectified by using the `mdmconn` command, which returns 1 if the modem is off-hook, otherwise 0. Off-hook modem states include dialling, connected and disconnecting; on-hook means the modem is not connected and idle. The following task illustrates this:

```
repeat 60
beginntask
  if < mod time 3600 300
    dset 4      # inside first 5 minutes of hour: turn GSM on
  else
    if not mdmconn # check to see if not in use
      dclr 4      # safe: turn it off
    endif
  endif
endntask
```

7.18.4 SMS messages

The SMS facilities available with GSM phones and modems are supported by the Etherlog, and are fully described in our Technical Information Note 8 "Using SMS with the Etherlog". The `sms` command in *lucid* permits a task to send an SMS message *via* a GSM modem. The contents of the message are defined by the format of a file stored on the logger, and this name is supplied as the argument to the `sms` command. This section illustrates the use of the `sms` command: please see the Technical Information Note for full details of the SMS facilities implemented in the Etherlog.

Suppose that you want to be informed as soon as possible by SMS if a temperature goes too high. If you set up a file called `temphigh.sms`, you can send it as follows:

```
if f> fcmt2c ain 1 50      # if temp on channel 1 exceeds 50 deg C
  sms temphigh.sms
endif
```

The `temphigh.sms` file looks like this:

```
07777123456
Inlet temp overrange
```

and the message sent to the number on the first line of the file will be the text on the second line of the file.

The message can optionally contain ‘escape’ sequences which allow numeric values from the task to be inserted into the text: this is useful if you want to inform the callee of the present value of the temperature (and possibly other information as well). The `arg` mechanism is used for this, in the same way as for setting up the arguments to a macro when calling one from a task. The following example sets up the first `arg` with the temperature which went overrange, and the second `arg` with another value, both expected by the message format escape sequences. The following illustrates this.

The `temphigh.sms` file looks like this:

```
07777123456
Inlet temp %i overrange (outlet temp %i)
```

The task fragment to send the message looks like this:

```
setvar 1 fcmt2c ain 1      # read inlet and outlet temps
setvar 2 fcmt2c ain 2
if f> var 1 50            # inlet overrange ?
    setarg 1 int var 1    # yes: set args to integer temp values
    setarg 2 int var 2
    sms temphigh.sms     # and send to user
endif
```

The message received will look like this:

```
Inlet temp 51 overrange (outlet temp 37)
```

There are several other escape sequences which are preconfigured to insert time, logger address, alarm code etc into messages, but only three sequences require their values to be set up using `setarg`. These are `%i` (integer value), `%f` (floating point value), and `%x` (hexadecimal value).

The task can verify that the message has been sent into the network by checking the return value of the `sms` command: this is 1 if the message has been accepted by the network, and 0 if not. This permits the task to retry sending the message at a later time if it failed.

Note that success of this command is not an indication that the message has reached the user’s phone or that it has been read – only that it has been accepted by the network for forwarding to the user’s phone.

Reasons for failure include problems with the SMS file, communications session active so unable to send, phone number problems, and network problems.

7.19 RDT support functions

Command	Argument(s)	Return	Description
<code>rdtenable</code>	<code><n1></code>	<code><n></code>	If <code><n1></code> is 0, disable all RDT reception, else enable it. Return <code>n1</code>

The `rdtenable` command is used to enable and disable reception of RDT broadcasts. Disabling RDT reception using `rdtenable 0` allows the Etherlog to sleep, saving battery power when RDT data is not required. With RDT reception enabled (using `rdtenable` with any value other than 0), the Etherlog is prevented from sleeping.

8 Using the trace and step facilities

The trace and step facilities provide a way of checking that a *lucid* program performs as intended, or alternatively of determining why a program doesn't give the results expected. The trace facility allows the results of each function evaluation to be tabulated for checking. It may produce a large volume of information, and in this case the facility to generate a record file offered by both *radhost* and *serhost* provides a useful way of recording the trace output. Alternatively the step facilities may be used to halt execution at any stage whilst the trace output is inspected.

8.1 Tracing program execution

The file `test4.lcd` contains the following *lucid* program, which calculates average, minimum and maximum values and tabulates them every five minutes:

```
# initialise the variables to be used
areset 1
setvar 2 maxval
setvar 3 minval

repeat 60

begintask

# read linearised value of analogue channel 1 into variable 1
setvar 1 lin10k ain 1

# add value to accumulator 1 and increment associated counter
accum 1 var 1

# turn trace output on
trace 1

# minimum is stored in variable 2: update it
setvar 2 min var 1 var 2

# maximum is stored in variable 3: update it
setvar 3 max var 1 var 3

# turn trace output back off
trace 0

# if we are on the five minutes write data to file
if < mod time 300 30

# create a newline and write the timestamp
newline
write time

# write average, minimum and maximum
write avge 1
write var 2
write var 3

# reset total, count and minimum and maximum
areset 1
setvar 2 maxval
setvar 3 minval
endif
endtask
```

Initially the `trace` instructions will have no effect. However, if you are running the Windows based host PC software then selecting Utilities Log trace will start the trace output and the logger will display the result of each step in the execution of the part of the program bracketed by the `trace 1` and `trace 0` commands. You can interrupt the trace at any stage using the Pause button, and then

restart it using `Continue`. Alternatively, if you are operating the logger from the command line you should type:

```
log trace on
```

at the `rdl>` prompt to enable tracing. The corresponding command:

```
log trace off
```

will disable the trace.

A sample of the trace output is shown below, with explanatory remarks alongside. On this occasion the value measured and stored in variable 1 is less than the previously calculated maximum, and so this is not updated. It is, however, also less than the previously calculated minimum, and so this is updated.

```
f=1                <-- the trace 1 command has returned 1
setvar n=2         <-- executing setvar instruction - first argument required evaluates to 2
  min              <-- the min instruction requires two arguments
    var n=1        <-- the first is a var function - its argument evaluates to 1
      f=2385        <-- var function returns most recent measurement 2385 (23.85°C)
    var n=2        <-- second argument is also a var function - argument evaluates to 2
      f=2355        <-- var function returns old minimum 2355 (23.55°C)
    f=2355         <-- min function returns 2355, the maximum of the two
f=2355            <-- returned as second argument to setvar: running minimum unchanged
setvar n=3         <-- executing the second setvar function - first argument evaluates to 3
  max              <-- the max function requires two arguments
    var n=1        <-- the first is a var function - its argument evaluates to 1
      f=2385        <-- var function again returns most recent measurement 2385 (23.85°C)
    var n=3        <-- second argument is also a var function - argument evaluates to 3
      f=2362        <-- var function returns old maximum 2362 (23.62°C)
    f=2385         <-- max function returns 2385, the maximum of the two
f=2355            <-- and this is returned as second argument to setvar: maximum is updated
trace n=0         <-- trace turned back off - returned value will not be displayed
```

Once you are satisfied that the program works as intended, turn the trace facility back off by closing the trace window, or typing:

```
log trace off
```

at the `rdl>` prompt. In practice the amount of information produced by the trace instruction makes its use rather slow on the radio link, and if you propose to make extensive use of it you may prefer to connect *via* the RS232 port.

8.2 Stepping through a program

The instructions described in this section allow a *lucid* program to be paused during its execution. It is clearly important that any instructions inadvertently left in a program do not cause it to pause indefinitely when it is out in the field. For this reason the instructions only have an effect when:

1. `log trace` is switched on, and
2. trace is enabled in the program as described in the previous section, and

3. a radio or RS232 connection exists.

If these criteria are all satisfied, then the `brkpt` instruction forces the program to pause, displaying the breakpoint prompt `<>`, until the user instructs it to proceed by clicking on the Step button in *winhost*, or typing `<return>` at the command line. This gives the programmer the chance to inspect the results of the most recently evaluated functions, examine the state of digital outputs or alter input levels. For example, with trace enabled the code:

```
write time
brkpt
```

produces the output:

```
write
time
f=12:34:50 16/01/98
f=1
brkpt <<B>>
```

and then pauses until the user clicks the Step button or types `<return>`.

The `sstepon` command goes one stage further than `brkpt`. It forces the program to pause after every function execution, at which stage the step prompt `<<S>>` will be displayed. Once again, clicking Step or typing `<return>` causes execution to resume. `sstepoff` turns single step mode back off. As an example, the code:

```
setvar 1 20
sstepon
newline
write time
sstepoff
setvar 1 21
```

pauses after the `newline`, `time` and `write` commands (in that order) and then resumes normal program execution.

9 *lc* - the *lucid* checking program

The *lucid* checker is a utility program which provides a means of spotting programming errors or oversights in your *lucid* programs. It is not comprehensive, but it will find many common problems, including:

1. essential keywords not present in the expected places
2. programming structures incorrectly defined
3. invalid formatting of integer, hexadecimal and floating point numbers
4. out-of-range numeric arguments
5. illegally nested subtasks

The program normally sends its output to the screen, but it can optionally be directed to write a 'clean' version of the checked program to disk.

9.1 Starting *lc*

In order to check the *lucid* program in, for example, the file `test1.lcd`, type:

```
lc test1
```

Like the Etherlog, the *lucid* checking program assumes a default extension of `.lcd` for a *lucid* program file. *lc* runs and displays its output, consisting of a cleaned-up version of the program `test1.lcd` and any error messages, on the screen. As with the Etherlog it is possible to over-ride the default file extension, and typing:

```
lc test1.tst
```

will check the program in the file `test1.tst`. If the input filename contains any directory path information (a backslash character or a colon), then this is preserved and *lc* will look for the input file only in the specified directory. If there is no directory information in the filename, as above, *lc* will look for the given filename only in the current working directory.

Once it is running, *lc* performs checks of the various elements of your *lucid* program to ensure that the essential components are in place. There must be a `repeat` command before `begintask` indicates the start of the main task, and `begintask` and `endtask` must occur at the bottom level of control structure nesting. If you define subtasks then these must appear before the main task, and be properly delimited with `subtask` and `endsubtask` commands.

Any numbers or time strings that you have entered are checked for validity. Additionally, if you supply a numeric argument to a command which expects a number within a given range, then it is checked against that range. This can be very useful in spotting argument errors in analogue and digital input/output channel numbers, variable numbers, accumulator numbers and pulse channel numbers.

Typographical errors will usually report an invalid instruction or unrecognised token, or an invalid number. The checker can save you a lot of time tracing a program with this sort of error.

Note that the checker stops when `endtask` is found at the bottom level of control structure nesting and no errors have been found. This mimics the way that the *lucid* program will be executed in the Etherlog. You can include anything you like after the `endtask`: it will not appear in the checked output, and will not be processed by the checker.

The formatting of the checker's output is independent of the appearance of the input file. All comments are removed. Control structures are printed with indenting to show the layout of the program more clearly. This can be invaluable in verifying that your program has the form you intended, and we strongly recommend that you indent your *lucid* programs when you write them.

As an example, consider the short program below, written without any indenting. The programmer may originally have had the intention of the program in mind, but seems to have become confused at some point during development of the code. Certainly, it is not entirely clear what is supposed to happen when `ain 1` is more than 100 000 and `ain 2` is more than 200 000.

```
# example to show lucid checker indenting
repeat 10
begintask
if > ain 1 100000
if > ain 2 200000
setvar 1 else setvar 1 20
endif
else setvar 2 300
endif
endtask
```

Running this through the *lucid* checker produces the following output:

```
repeat 10
begintask
  if > ain 1 100000
    if > ain 2 200000
      setvar 1
line 6: unexpected 'else'
```

lc points out that it was not expecting to find an `else`, and on inspection we can see that the second argument of `setvar` was expected. We correct the program (the intention was to set variable 1 to 100 under these conditions) and run *lc* again to get the following output:

```
repeat 10
begintask
  if > ain 1 100000
    if > ain 2 200000
      setvar 1 100
    else
      setvar 1 20
    endif
  else
    setvar 2 300
  endif
endtask
```

This time, no errors are reported. Note how the indenting of the program structures (the `if/else/endif` clauses) visually partitions the various sections of the program. Now we can see immediately what will happen if `ain 1` is less than or equal to 100 000 by skipping down to the 'else' at the same indent. In a similar vein, *lc* can spot redundant arguments which serve no purpose other than tying up the Etherlog's computing resources unnecessarily. The following program contains an unnecessary argument which might be missed without the checker program. Here is the original source fragment:

```
write time
setvar 1 fadd 3.14159 fdiv ain 1 1000000 20
write var 1
```

which is apparently trying to set variable 1 to the sum of π and the voltage on analogue input 1 in Volts (recall that the `ain` command returns its result in microvolts). However, we have an extra argument of 20 at the end of the line, which will be shown up thus by *lc*:

```

write time
setvar 1 fadd 0.314159e+01 fdiv ain 1 1000000
20
write var 1

```

We can immediately see that the 20 serves no useful purpose, and may be deleted. It would be evaluated by the Etherlog as the number 20 and then discarded, since no preceding function wants its value. Of course, if by mistake you include extra arguments *inside* otherwise valid *lucid* statements these may be impossible for *lc* to identify, but the very existence of spurious arguments should alert you to the possibility of an incorrect *lucid* expression.

The situation can get slightly more complicated if you use the return values of control structures as arguments. Remembering that the `if`, `loop`, `while` and `call` constructs return the result of the last evaluation within them, it possible to use them to return arguments to other functions. Consider the following program fragment:

```
setvar 1 if > var 2 10 1000 else 2000 endif
```

In this, we intend to set variable 1 to 1000 if variable 2 is greater than 10, otherwise to set it to 2000. This is quite acceptable in *lucid*, but you may not expect the checker to output the following:

```

setvar 1 if > var 2 10
  1000
else
  2000
endif

```

Here, the `if` command, placed in the position of an argument to the `setvar` command, is intentionally left in the argument's position. However, its expressions evaluated if the test is true or false *are* indented (in this case, the numbers 1000 and 2000), and the keywords `else` and `endif` are unindented to the same level as the `setvar` command. This is normal behaviour, confirmed by the fact that no overall indenting from the `setvar` level has occurred. The entire `if .. else .. endif` structure is treated as a single argument to `setvar`, but the printed output can look confusing unless you know why it happens.

It follows that if you see `else .. endif` apparently without an `if` at the same level in checked output, and you did not intend to use the technique just described, there is likely to be a missing argument whose position has been adopted by the `if`.

As an aside to this example, beware of using `if` constructs like this when there is no `else` clause, otherwise the `if` command will return the result of the last evaluation performed, which is the result of the `if` test (which is false, or zero) if the `else` clause would have been performed. Ensure you always specify the intended result !

While *lc* can find most problems associated with programming errors, there are certain things that it cannot find. These will typically be problems that will not show up until the program is run, such as supplying a channel number to the `ain` command from a variable which has been set to an inappropriate value somewhere else in the program, or not initialised properly. The following example illustrates this:

```

repeat 10
begintask
  newline
  write time
  write var 1           # write current channel number to data file
  write ain var 1      # read analogue channel and write to data file
  setvar 1 add var 1 1  # increment the channel number
  if > var 1 8         # any more channels left ?
    setvar 1 1        # if not return to channel 1
  endif

```

endtask

This program is intended to cycle round all eight analogue inputs, reading one channel each 10 seconds. When a task is started, all of its local variables are set to 0: the programmer obviously forgot this, because at the first execution of the main task variable 1 is set to 0, and an attempt is therefore made to read analogue input 0 (which does not exist). This kind of run time problem is impossible for *lc* to find.

9.2 Checking programs which do not set a repeat interval

The vast majority of programs use the `repeat` command to set the interval at which the main task will be executed. If this is omitted the main task will not be started. In normal use *lc* therefore classes the absence of a `repeat` command as a programming error.

However, as described in Chapter 4, there are some circumstances in which a programmer may deliberately wish to leave out the `repeat` command. In order to check such programs it is necessary to use the `-r` directive to disable this particular check when *lc* is run. For example:

```
lc -r norepeat
```

checks the program `norepeat.lcd` on the assumption that the `repeat` command has been intentionally omitted.

9.3 Error messages

When *lc* encounters what it considers to be an error in a *lucid* program, it stops checking and prints an error message on the screen. All error messages indicate the input file line number where the error was encountered, and give a description of the error. These error messages are described in detail below.

```
invalid instruction <command>
```

lucid commands are distinguished from their arguments by virtue of starting with a lower case letter. This message is displayed when a keyword starting in this way could not be matched in the internal command table. This probably indicates mistyping of the program.

```
unrecognised token <token>
```

If a program item does not start with a lower case letter and is not a number, this message will be displayed. Again, this will generally be because of a typing problem.

```
invalid integer number <number>
```

Integer numbers may contain only the digits 0..9, and may optionally start with either '+' or '-' to indicate sign.

```
invalid floating point number <number>
```

Floating point numbers consist of an optional sign followed by a decimal mantissa (optionally containing a decimal point), followed optionally the exponent. If the exponent is present then the indicator 'e' follows without spaces, followed by an optional '+' or '-' exponent sign and at least one decimal digit of exponent.

```
invalid hexadecimal number <number>
```

A hexadecimal number must start with the indicator '0x' followed (without spaces) by at least one hexadecimal digit in the range 0..9, A..F. The letters are not case-sensitive.

```
invalid window start time <time>  
invalid window stop time <time>
```

The time strings supplied to a 'window' command must be in the format hh:mm:ss or hh:mm (the seconds are optional). The hh part must be in the range 0..23, mm in the range 0..59, and ss (if supplied) in the range 0..59.

```
numeric argument out of range
```

The analogue, digital and pulse input commands, digital output commands, and the variable and accumulator commands all expect integer arguments in a particular range (for example, `ain` expects an argument in the range 1 to 8). If your program specifies the argument explicitly but it is out of range for the command, this message is displayed.

```
no 'repeat' command before 'begintask'
```

A `begintask` command was found, but the initialisation section does not contain a `repeat` command. This situation would result in the Etherlog refusing to start the task. The `-r` directive suppresses the generation of this error message.

```
unexpected end of input file
```

Indicates that `lc` was expecting another command or argument, but reached the end of the input file. For example, if you omit the `endtask` from an otherwise correct program, this message will be displayed because there could be more program statements to process.

```
unexpected 'else' 'endif' 'endloop' 'endwhile' 'endsubtask' or 'endtask'
```

One of the commands shown (only the offending item is printed - see the example above) was encountered where it would not have its intended effect. In the example above, the `else` does not belong with its intended `if` and would cause incorrect behaviour of the program.

```
'begintask' inside control structure
```

The `begintask` command did not appear at the bottom level of structure nesting - for example, an `endsubtask` might be missing, effectively placing the `begintask` in the middle of a subtask.

```
more than one 'else' clause
```

An `if` statement may have either zero or one `else` clauses; a second one was found in the current `if` level. This may indicate a missing `endif` from a previous or subsidiary `if` statement: the indented checker output will help in finding the problem.

```
subtask <name> defined inside main task
```

Subtasks can be defined only in the initialisation section.

```
subtask <name> defined inside another subtask
```

Subtask definitions cannot be nested - this message probably indicates a missing `endsubtask` before the subtask being checked.

```
subtask <name> already defined
```

Subtask names must be unique.

```
attempt to define more than <n> subtasks
```

The Etherlog has a maximum number of subtasks which it can support for each task. This maximum is currently 4; an attempt was made to define a fifth one.

```
call to undefined subtask <name>
```

The named subtask has not been defined. This probably indicates a mistyped subtask name in the subtask definition or in the call.

9.4 Directing 'clean' output to a file and suppressing screen output

The *lucid* checker allows you to write the cleaned up program to a file. If you want to save the cleaned-up version of `test1.lcd`, type:

```
lc -c test1
```

As before this will open the file `test1.lcd` for input. Because you have not specified a name for the output file *lc* will take the name part of the input file, append the extension `.lcc` to it, and provided that no errors are encountered during checking will write the checked output to the file `test1.lcc`. Note that as well as inserting spaces to produce the correct indenting *lc* removes all comments from `test1.lcd`, so the file `test1.lcc` may be larger or smaller than `test1.lcd`.

If you supply a name with the `-c` option but no extension, *lc* will append the `.lcc` extension to the name you specified. For example:

```
lc -ccheck test1
```

opens `test1.lcd` for input, and `check.lcc` for output. There must not be any spaces between the 'c' and the filename. Finally, you can specify the full filename, including the extension for the checked output file:

```
lc -ccheck.out test1
```

opens `test1.lcd` for input, and `check.out` for output.

Note that if you try to write the output back to the input file, *lc* spots this, warns you of the situation and stops.

If you do not want to see the checked program on the screen, type

```
lc -s test1
```

which suppresses the output of the cleaned up program and displays only error messages and summary information.

These options can be combined to suppress screen output but write a checked file (if there are no errors):

```
lc -s -ccheck.out test1
```

writes the checked program to `check.out` but not to the screen.

10 *lucid* error messages

In normal operation the Etherlog 3000 will spend most of its time collecting data unattended, under the control of one or more *lucid* programs. In view of this the language has been made as robust as possible. If an error is encountered during the execution of a program an error message is written to the current output file and *lucid* abandons the current execution of the task. This is an important feature: it may be that a program fails only occasionally, for example when some particular combination of input conditions occurs. The fact that *lucid* abandons only the current execution of the task means that the program will try to continue gathering data when the main task is next due to run. When the resulting data is retrieved from the logger the user should spot the error message, and investigate the reason for it. Version 3.00 of *lucid* can generate three such error messages, and these are described below.

```
Error 1: unknown command encountered - this execution of main task
abandoned
```

An unknown command was encountered in the logging program. Systematic use of the *lucid* checker, as described in Chapter 10, should allow you to avoid this source of error.

```
Error 2: stack overflow - this execution of main task abandoned
```

A *lucid* task uses stack space every time a command is executed. This is used to store the return addresses from functions and subtask calls as they are nested. If they are nested too deeply then the stack space provided may prove inadequate. In this situation there is nothing that the interpreter can do except terminate execution and declare an error. The most common cause of this error is a subtask which calls itself, but it may also be caused by nesting arithmetic operations too deeply (the current limit is about 22 functions). This may be cured by breaking the offending expression down into a series of smaller evaluations, storing intermediate results in a variable.

```
Error 3: error command encountered - this execution of main task abandoned
```

In spite of the warnings in Chapter 7 you chose to use the *lucid* `error` command within a program, and it was duly executed.