

Oxford Oberon-2 Compiler

# Users' Manual

Mike Spivey

Another vital P<sub>t</sub>R.G. project

Oxford University Computing Laboratory

Draft of 13.iii.2007

Copyright © 1999 J. M. Spivey

---

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Using the Oberon compiler</b>	<b>4</b>
2.1	Introducing the compiler	4
2.2	Programs with more than one module	5
2.3	Managing compilation with make	7
2.4	When things go wrong	9
2.5	More about the Oberon compiler	11
2.6	Profiling	11
2.7	Language differences	13
<b>3</b>	<b>OBC Library Reference</b>	<b>14</b>
3.1	Module <i>SYSTEM</i>	14
3.2	Module <i>In</i> : Standard input	14
3.3	Module <i>Out</i> : Standard output	14
3.4	Module <i>Err</i> : Standard error	15
3.5	Module <i>Files</i> : File input/output	15
3.6	Module <i>Math</i> : Mathematical functions	16
3.7	Module <i>MathL</i> : Mathematical functions for <i>LONGREAL</i>	17
3.8	Module <i>Args</i> : Program arguments	17
3.9	Module <i>Random</i> : Random numbers	17
3.10	Module <i>XYplane</i> : Simple bitmap graphics	18
3.11	Module <i>Conv</i> : Numerical conversions	18
3.12	Module <i>String</i> : Operations on strings	19
3.13	Module <i>Bit</i> : Bitwise operations on integers	19
3.14	Module <i>Coroutines</i> : Multiple threads	19
<b>4</b>	<b>Language extensions</b>	<b>21</b>
<b>5</b>	<b>Interfacing with C code</b>	<b>22</b>

## Introduction

This manual explains how to use the Oxford Oberon-2 compiler **obc** to compile and run Oberon-2 programs. The content is largely extracted from the laboratory manual used by our undergraduates in Oxford for their second programming course (their first course is functional programming with Haskell).

The following typographical conventions are used throughout this manual:

- Names of programs are shown in **bold face** type.
- File names are shown in *italic* type.
- The text of Oberon programs is shown in **sans serif** type, except that identifiers in the running text are shown in *italic*.
- The text of UNIX commands is shown in **typewriter** type. Where an interaction with the computer is shown, the characters you type are shown in *italic typewriter* type, and the computer's responses are shown in **upright typewriter** type.

## Using the Oberon compiler

This chapter will tell you how to use the Oberon compiler **obc** to compile and run Oberon programs.

### 2.1 Introducing the compiler

Before your Oberon program can be run, it must first be translated by the Oberon compiler into a form that can be directly obeyed by a machine. For example, Figure 2.1 shows a little Oberon program for computing factorials. As you can see, the name of the file exactly matches the name of the module *Fac* that it contains, even down to the capitalization of the name. To translate this program with the Oberon compiler, give the command<sup>1</sup>

```
% obc -o fac Fac.m
```

This command runs the Oberon compiler **obc**, asking it to translate the program found in *Fac.m* and place the translation (“**-o fac**”) in the file *fac*.<sup>2</sup>

When you have done this, you can run the program by giving the UNIX command *./fac*, like this:

```
% ./fac  
Gimme a number: 4  
The factorial of 4 is 24  
Gimme a number: 5  
The factorial of 5 is 120  
Gimme a number: 6  
The factorial of 6 is 720  
Gimme a number: -1  
%
```

As you can see in Figure 2.1, the program contains a loop that repeatedly asks for a number and computes its factorial, until a negative number is input. With our usual setup at the Computing Laboratory, you have to type *./fac* as the UNIX command for starting **fac**, to reflect the fact that the file *fac* is in the current directory, not one of the directories where UNIX usually keeps executable programs.

---

<sup>1</sup> In this manual, I’ve shown the shell’s prompt as *%*, but it may appear differently, depending on the shell that your account has been set up with. The characters you type are shown in *italics*.

<sup>2</sup> If you don’t say “**-o fac**”, the compiler rather unhelpfully puts the translation in a file called *a.out*, according to a long-standing UNIX tradition. If you’re using **obc** under MS-DOS or Windows, you’d better replace **-o fac** by **-o fac.exe**, so that the resulting program can be executed from the *C:\>* prompt.

```

MODULE Fac;
IMPORT In, Out;
VAR n, i, f: INTEGER;
BEGIN
  LOOP
    Out.String("Gimme a number: "); In.Int(n);
    IF n < 0 THEN EXIT END;

    i := 0; f := 1;
    WHILE i < n DO i := i+1; f := f*i END;

    Out.String("The factorial of "); Out.Int(n, 0);
    Out.String(" is "); Out.Int(f, 0); Out.Ln
  END
END Fac.

```

**Figure 2.1:** Contents of file `Fac.m`

The file *fac* contains low-level instructions that are equivalent to the high-level program in *Fac.m*. Once the translation has been done, you could delete the file *Fac.m* and still run the low-level version in *fac*; or you could copy the file *fac* and give it to a customer, who could run the program without needing to have the source code.

There's something a little unusual about the Oberon compiler we'll be using, and that's the fact that it doesn't actually translate your program into instructions for the physical computer you are using. Instead, the designer of the compiler has invented a simple 'abstract' computer specially for implementing Oberon, and the file *fac* contains instructions for that computer instead. The gap between the abstract computer and the actual, physical computer is bridged by a special program that uses the physical computer to carry out a simulation of the abstract computer. The biggest advantage of this is that the compiler can be moved ("ported") to a new computer just by making a new simulator – and since the simulator is itself written in a highish-level language (C), that's usually just a matter of re-compiling it on the new computer.

The biggest disadvantage of this scheme is that the software simulation of the abstract computer is much slower than a real computer would be, so your programs will run 5 to 10 times more slowly than they would if the compiler really translated them into machine code. But [in the course that I teach at Oxford] we won't be writing any programs that run for a very long time, so that won't be too much of a disadvantage.<sup>3</sup>

## 2.2 Programs with more than one module

Bigger programs are often divided into several modules for ease of understanding. If this division into modules is done judiciously, it should be possible to understand a lot about each module without looking at the other ones. A well-designed module has a small interface, and it is possible to say *what* the module does without giving the details of *how* it does it.

As a contrived example, let's look at a variation on our program for calculating factorials that calculates Fibonacci numbers instead. Calculating Fibonacci

<sup>3</sup> Actually, there's a hidden advantage: having an artificially slowed-down computer will encourage you to learn how to write programs that make economical use of computer time.

```

MODULE FibMain;
IMPORT In, Out, FibFun;
VAR n, i, f: INTEGER;
BEGIN
  LOOP
    Out.String("Gimme a number: "); In.Int(n);
    IF n < 0 THEN EXIT END;

    f := FibFun.Fib(n);

    Out.String("fib("); Out.Int(n, 0);
    Out.String(") = "); Out.Int(f, 0); Out.Ln
  END
END FibMain.

```

**Figure 2.2:** *Contents of file FibMain.m*

numbers is a simple task that can be done in several ways, some more complicated and efficient than others. That makes it sensible to introduce a procedure, so as to separate the *what* – the Fibonacci function – from the *how* – the algorithm for calculating it.

Putting the procedure in its own module is over-kill, since there are no details to hide that are not already hidden inside the procedure. But doing so will let us show how to split a program into modules, so let's do it anyway. Figure 2.2 shows a program like our earlier factorials program, but with the code to calculate the factorial replaced by a call to the function *FibFun.Fib*, imported from a module *FibFun*. Figure 2.3 shows the text of that module, including a procedure that calculates Fibonacci numbers. Note the export mark *\** in the heading of this procedure.

To compile this program, we must separately translate the two modules into machine code, then combine the two files of machine code into one program.

```

MODULE FibFun;

PROCEDURE Fib*(n: INTEGER): INTEGER;
  VAR a, b, c, i: INTEGER;
BEGIN
  IF n = 0 THEN
    RETURN 0
  ELSE
    a := 0; b := 1; i := 1;
    WHILE i < n DO
      c := a + b; a := b; b := c;
      i := i + 1
    END;
    RETURN b
  END
END Fib;

END FibFun.

```

**Figure 2.3:** Contents of file *FibFun.m*

Although all this can be done with the single command

```
% obc -o fib FibFun.m FibMain.m
```

it is better to separate the three steps, so that we need not repeat all the steps if we change only some of the modules. With a small example like this, it doesn't make a significant difference: but as I've already admitted, using modules at all in this example is over-kill. To compile the two modules separately, we need the three commands

```

% obc -c FibFun.m
% obc -c FibMain.m
% obc -o fib FibFun.k FibMain.k

```

The first command translates the *FibFun* module, producing a file *FibFun.k* that contains both the machine code and a description of the module's interface that is used when compiling the other module *FibMain.k*. Similarly, the second command translates the *FibMain* module into a file *FibMain.k*. During the translation, the compiler reads the file *FibFun.k* to check that the function *FibFun.Fib* is used in a way that is consistent with its definition. The third command combines the two files of machine code with the code for the library modules *In* and *Out* to produce the executable file *fib*.

## 2.3 Managing compilation with make

If you are writing a program, perhaps in several modules, and trying to get it to work, it becomes very boring to have to type out a long sequence of commands each time you want to compile it. Also, if we make a change to only one of the modules in a big program, we want to avoid recompiling all of it so as to save time.

A partial solution to this problem would be to write a *shell script* for compiling the program: that is, a sequence of UNIX commands that is stored in a file.<sup>4</sup> This solves the problem of typing a long command or sequence of commands each time we want to compile our program; but there's a better way that uses a

<sup>4</sup> MS-DOS aficionados will recognize shell scripts as being what they call "batch files".

```

1 # Makefile for fib
2
3 FIB = FibFun.k FibMain.k
4 fib: $(FIB)
5     obc -o fib $(FIB)
6
7 FibMain.k: FibFun.k
8
9 %.k: %.m
10    obc -c $<
ing

```

Figure 2.4: Contents of file *Makefile*

nifty program called **make**. This program lets you write a script that, like a shell script, contains the UNIX commands for compiling your program. Unlike a shell script, the **make** script associates each of these commands with a file that it produces. The **make** program uses the modification time that UNIX associates with each file to work out which files in your program have been changed, and issues only the commands that are needed to bring everything up to date. Usually, there's just one script that describes how to compile all the programs in a directory, and the script is stored in the same directory under the name *Makefile*.

For example, Figure 2.4 shows a makefile for our little Fibonacci program: I've numbered the lines for ease of reference. In this course, you will not need to know how to write makefiles, because a working makefile will be provided when it is needed. Nevertheless, **make** is such a useful program that it's good to understand something of what it can do.

Line 1 of the makefile is a comment: it starts with **#** and continues to the end of the line. Lines 3–4 describe the last of the three commands needed to build the **fib** program, using the list of files **FIB** defined on line 2. Line 3 records the fact that the file *fib* is made from the two files *FibFun.k* and *FibMain.k*, so that it should be rebuilt if either of these two files is newer than it; and line 4 (starting with a tab character) gives the UNIX command to execute in this case.

The other two commands needed to build **fib** are summarized in the 'pattern rule' on lines 6–7; this says that any file *???.k* can be made from the corresponding file *???.m* by running the command `obc -c ??? .m`. Line 5 records the fact that rebuilding *FibFun.k* requires *FibMain.k* to be rebuilt too, because the interface of the *FibFun* module may have changed.

Suppose we were to modify our program, perhaps to make it more acceptable to English tastes by replacing the word "Gimme" by the polite phrase "Kindly enter". To do this, we would use a text editor on the file *FibMain.m*, and our final action with the editor would be to save the new file over the old one. This would cause UNIX to change the modification time of the file, making it newer than the file *FibMain.k* that was generated by the compiler last time we built the program. Now we simply give the command

```
% make
```

The **make** program examines the rule that says how to make *FibMain.k* from *FibMain.m* and sees that the time-stamps are wrong. So it runs the Oberon compiler and makes an up-to-date version of *FibMain.k*. This causes the file *FibMain.k* to be newer than the final program *fib*, so **make** also runs the command to rebuild *fib*. Since we haven't touched the *FibFun* module, it does not need to be rebuilt, and **make** omits the command to build it. If we immediately run **make** again, it will do nothing the second time, because each file will already be

up to date with respect to the files from which it is built. So it's quick and easy to use **make** simply to check that everything is up to date.

Two very common mistakes in using editors and **make** are these: forgetting to save a file in the editor before running **make**, and forgetting to run **make** after changing your program. The first kind of mistake is noticeable, because **make** does less work than you expected: if you change a source file, it ought to be recompiled. Both kinds of mistake become noticeable when your program continues behaving exactly as it did before you made your changes.

## 2.4 When things go wrong

There are many kinds of things that can go wrong in writing a program:

- You might write a program that does not conform to the grammatical rules of Oberon.
- You might write a program that, although grammatically correct, does not make any sense because you have (for example) forgotten to declare some of the variables you use.

Because of the way the Oberon compiler has been written, it checks that your program is grammatically correct before it begins to address the question of whether the program makes sense, and it checks the whole program for sense before it starts to generate the machine code translation. So if you write a grammatically incorrect program that also contains undeclared identifiers, then the compiler will not bother to mention the missing declarations until you have put the grammar right; and if the program contains grammatical errors, if declarations are missing or if the types do not match up properly in expressions, the compiler will not generate any machine code for your program until you have put it all right.<sup>5</sup>

After detecting one grammatical error in your program, the compiler tries to continue with its analysis of the rest of your program, so that you can correct more than one error each time you run the compiler. The compiler works by reading your program from beginning to end, and it produces its *first* error message when it reaches the first word or symbol in your program that could not be the next symbol in any valid Oberon program. So the first error message at least is reliable.

After the first error, the compiler has the problem of re-construing your program in such a way that it can continue and look for other errors. The methods used for this are necessarily crude, because after all your program is wrong, and the compiler can only guess at what you meant. The compiler uses rules for recovery like “skip to the next semicolon, and see if what follows looks like the beginning of another statement”. This means that the compiler can diagnose as another error in your program a problem that is in fact caused by the compiler's own state of confusion. For example, the compiler might skip over an *END* keyword as it looks for a place to restart, and that might cause it to think that a later procedure declaration appears in the middle of the current procedure. There's no way to avoid this problem in general, so it's best to be aware that only the first error message is really worth relying on.

When errors are reported by the compiler, the combination of the **obc** compiler, **make** and the **emacs** editor is particularly productive. **Emacs** and its X version **xemacs** provide a command that you can run by using the menus or toolbar under X, or by typing **M-x compile**;<sup>6</sup> this uses **make** to recompile your program, and displays the output of that process, including any messages from

<sup>5</sup> This is a bit annoying when the only error is a missing semicolon, I admit.

<sup>6</sup> Hold down the Alt key and press **x**, then release both, type the word **compile**, and press Return.

the compiler, in an **Emacs** buffer. The **obc** compiler puts its error messages in a standard format, like this:

```
"Planner.m", line 92: missing ';' at token 'IF'
>      IF verbose THEN ShowLink(u, "green") END;
>      ^^
>
```

Another **emacs** command **next-error** (bound to the key sequence<sup>7</sup> C-x ‘) recognizes these error messages and finds the relevant file and the relevant line, putting the editor’s cursor there ready to correct the error. Subsequent invocations of **next-error** will take you to the locations of other errors reported by the compiler.

Other things can go wrong when you run your program:

- You might write a program that performs an illegal operation when it runs. Examples of such illegal operations are accessing an array with an index that is negative or greater than the length of the array, and following a pointer that has been set to *NIL*.

Errors like these are detected when the program runs, because there is no generally applicable way for the compiler can work out in advance whether such errors can happen.<sup>8</sup> The advantage of a language like Oberon is that the compiler can generate code that checks for such errors, and stops the program immediately. For an array reference  $a[i]$ , the compiler generates a check that  $0 \leq i < N$ , where  $N$  is the length of  $a$ . If this is not true, then the program is stops with a message such as

```
Runtime error: array bound error
                on line 123 in module Directory
```

The Oberon language has been very cleverly designed so that errors of this kind can be detected immediately with simple compiler-generated checks. This property does not hold for C, for example, so C programmers are always trying to track down the cause of baffling problems.<sup>9</sup>

Some things that can go wrong do not result in any helpful message:

- Your program may run forever, without producing any output. A common cause of this is that you’ve written a loop

```
WHILE i < N DO ...
```

but have forgotten to put  $i := i + 1$  in the loop body.

- Your program may produce output that is wrong. In this case, there’s no substitute for careful thought.
- Your program may produce the right output, but run too slowly to be useful. In this case, the solution is *not* to start madly ‘optimizing’ everything in sight in the hope of speeding things up. If you do that, you will probably spend most of your effort on parts of the program that have almost no effect on the overall speed, and you will certainly make your program more complicated, perhaps so much so that it stops working properly, and you end up in a worse position than when you started. A better alternative is to use *profiling* to find out where the program is spending its time: see Section 2.6.

<sup>7</sup> Hold down the control key while pressing letter x, then release both and press ‘ (grave accent).

<sup>8</sup> Actually, we can *prove mathematically* that no compiler could do this, because it is equivalent to solving the ‘halting problem’ for Turing machines. The intriguing theory of such things is covered in the third year course on ‘Complexity’.

<sup>9</sup> That’s why debuggers are so popular with C programmers: when a program crashes, they use debuggers to try and work out what has happened.

## 2.5 More about the Oberon compiler

There are two switches that can improve the performance of programs compiled with **obc** at the expense of a little risk. The safer of the two switches turns on an optimizer in the compiler. Actually, the optimizer is so effective that it is turned on by default, and there is a switch `-O-` that can be used to turn it off if necessary. For compatibility with previous versions of **obc**, there is a switch `-O`, but it does nothing. It's perfectly acceptable to use the optimizer on some modules of your program but not others.

Using the optimizer tends to improve the running speed of programs by about 50%, at the cost of making the compiling process take longer. The improvement mostly comes from places where the compiler finds a way to combine several small operations in your program into one larger operation that is faster overall. There ought to be no risk associated with using the optimizer, because the behaviour of an optimized program ought to be the same as one that has not been optimized. But if you suspect a compiler bug, the first thing to do is turn off the optimizer: if this fixes the problem, you have a good case for a complaint.

A more dangerous switch is `-b`, which tells the compiler not to arrange for any run-time checks for illegal operations in your program, thereby saving the time that would be spent on making the checks. If a run-time error does happen (such as an array index out of bounds), the program just continues to run, perhaps crashing later in a way that is very difficult to diagnose. For example, the program might print a unhelpful message like

```
Segmentation fault
```

and stop abruptly at an unpredictable time long after the array bound error. Perhaps worse, the program may appear to work correctly, but simply give wrong answers for some inputs. The risks are obvious.

Other useful run-time checks that are suppressed with `-b` include the check that the pointer  $p$  is not *NIL* in a pointer reference  $p↑$ , and the check that a *CASE* statement contains a label that matches the value of the controlling expression. The cost of these checks might amount to about 10% of the running time of a program. Some people like to keep the checks in while they are debugging a program, then remove them before putting it to productive work. Tony Hoare once compared this practice to wearing a life-jacket while training on land, then taking it off before going to sea.

## 2.6 Profiling

Profiling is a process of gathering statistics about where a program spends its time, as the first step of speeding it up. Profiling makes for more effective optimization, because it lets you concentrate your effort on the things that will really make a difference. In most large programs, the majority of the code accounts for only a small fraction of the execution time – or to put it another way, a small part of the code consumes a big fraction of the time. If you can identify this small part of the code – sometimes called the *inner loop*, then you can devote all your energies to improving it. Perhaps there is a better algorithm or data structure that will speed up the time-consuming task. Or perhaps a different approach to the whole program will make the problem area go away. Time spent on optimizing the rest of the program, apart from the inner loop, may be entirely wasted, because the effect on the overall speed may be negligible.

Our Oberon system provides a profiling tool called **obprof** that is used as follows: if you would normally run your program through the command

```
% prog arg1 ... argn
```

Execution profile:

Time	Cumul	Calls	Procedure
32.5%	32.5%	4	In.Int
31.3%	63.9%	1	Fac..init
10.2%	74.1%	12	In.IsDigit
9.5%	83.6%	10	Out.String
6.1%	89.7%	9	In.Char
5.5%	95.1%	4	In.IsSpace
2.6%	97.7%	6	Out.Int
1.0%	98.7%	3	Out.Ln
0.5%	99.2%	1	Files..init
0.4%	99.7%	1	In..init
0.3%	100.0%	1	Args..init

Total of 1174 clock ticks

**Figure 2.5:** Execution profile

then you should use the command

```
% obprof prog arg1 ... argn
```

instead.<sup>10</sup> The **obprof** program is actually an augmented version of the usual Oberon run-time system; it runs your program exactly as usual, but it keeps statistics about what happens as the program runs. Specifically, **obprof** keeps track of what procedure in your program is active, and counts how long is spent in each procedure. When your program finishes, **obprof** summarizes the information it has gathered by listing the procedures in your program, how often each one was called, and the total time spent in the procedure. Figure 2.5 shows the output that is produced for a trial run of the **fac** program. In this profile, the notation *Fac..init* refers to the main body of the *Fac* module; similarly, *Files..Init* refers to the main body of the *Files* module from the standard library, which is executed before our program itself starts.

This profile is from too small an example program and too short a run to draw any firm conclusions, as you can see from the fact that *Out.Int* was called only six times. But you can see that a lot of the time was consumed in the process of reading numbers (procedure *In.Int*, which also calls *In.IsDigit*). Since *IsDigit* is such a simple procedure – it just tests whether a character is a digit – perhaps it would be better to do these tests where they are needed, rather than call a procedure to do them.

The profile data is obtained by counting execution cycles of the simulated abstract machine, and this is not quite the same thing as elapsed execution time. The most important difference is that some library procedures are actually implemented in C, so no cycles of the abstract machine are needed to execute them. You can see the effect in Figure 2.5, because the procedure *Out.Int* is implemented as a C primitive, and appears to take much less time than *In.Int*, which is implemented in Oberon using the primitive *In.Char*. Apart from these primitives, the other operations of the abstract machine each take a small, fixed time that has the same order of magnitude for each operation, so that the number of clock ticks is a good guide to the actual execution time. Typical operations,

<sup>10</sup> In the DOS version, you have to type something like

```
C:\> obprof prog.exe arg1 ... argn
```

and specify the **.exe** suffix explicitly.

each counted as one clock tick, are fetching the value of a variable, or adding two numbers together, or comparing two numbers and jumping if they are equal.

The **obprof** program also provides a more sophisticated form of profiling that takes into account the directed graph of calls between one procedure and another in the program. In such a *call graph profile*, the time spent in each procedure is also charged to its callers, so that you can build up a fuller picture of where time is being spent. In order to select call graph profiling, you need to give the **-g** switch to **obprof**, like this:

```
% obprof -g prog arg1 ... argn
```

For further details, see the **obprof** manual page.

Profiling fits in well with a style of programming that begins by building a simple program that computes the right answers, preferring simple but perhaps inefficient algorithms and data structures to more complex ones. Thus we might use linear search instead of a hash table, or insertion sort instead of quicksort, just as a way of getting the program working sooner. When the program is working correctly, we can use profiling to identify the places where these simplified design decisions actually have a significant effect on performance, and revise just these decisions to make the program faster. This method leads to programs that are simple, compact and reliable whilst still having good performance. A wise programmer knows when to stop looking for further improvements: there is no point working for days to shave a minute amount of runtime from a program that will only be used occasionally.

## 2.7 Language differences

The Oberon language accepted by the **obc** compiler is (with certain exceptions) that described in the document “Oberon-2 Language Definition”, an extension of the document “Oberon Language Definition” that appears as Appendix A of the book

M. Reiser and N. Wirth, *Programming in Oberon: steps beyond Pascal and Modula-2*, Addison-Wesley, 1992,

The exceptions are as follows:

- The numeric types are *SHORTINT* (16-bit integers), *INTEGER* (32-bit integers), *REAL* (single-precision floating point) and *LONGREAL* (double-precision floating point).
- Within each module or procedure, it is not necessary to order the declarations so that (nested) procedure declarations follow all others. Instead, declarations of constants, types, variables and procedures may appear in any order.
- The scope of a procedure includes the bodies of all procedures defined at the same nesting level, without the need for ‘forward declarations’. In fact, such forward declarations are not permitted.

## OBC Library Reference

This chapter contains a brief description of the library modules that are supplied with the **obc** compiler. The source code to these modules can be found in the directory `/usr/local/lib/obc`.

### 3.1 Module *SYSTEM*

The following features of the standard module *SYSTEM* are provided: types *PTR* and *BYTE*, and procedures *ADR*, *VAL*, *BIT*, *GET*, *PUT* and *MOVE*.

### 3.2 Module *In*: Standard input

This module provides simple input operations on the standard input channel.

PROCEDURE Char(VAR c: CHAR);

- Input one character.

PROCEDURE Int(VAR n: INTEGER);

- Input a decimal integer.

PROCEDURE Real(VAR x: REAL);

- Input a real number.

PROCEDURE Line(VAR s: ARRAY OF CHAR);

- Input one line of text and store it in *s*, removing the terminating newline character.

VAR Done: BOOLEAN;

- This read-only variable is set to *FALSE* when an input operation reaches the end of the input file.

### 3.3 Module *Out*: Standard output

This module provides simple output operations on the standard output channel.

PROCEDURE Int(n: INTEGER, width: INTEGER);

- Output an integer in decimal, in a field of at least *width* characters.

PROCEDURE Real(x: REAL);

- Output a real number, using scientific notation if it is very large or very small.

PROCEDURE Fixed(x: REAL; width, dec: INTEGER);

- Output a real number in decimal notation, in a field of at least *width* characters, with *dec* digits after the decimal point.

PROCEDURE Char(c: CHAR);

- Output a character.

PROCEDURE String(s: ARRAY OF CHAR);

- Output a string.

PROCEDURE Ln;

- Output a newline character.

### 3.4 Module *Err*: Standard error

This module provides simple output operations on the standard error channel, which remains connected to the user's display even when the standard output channel is redirected elsewhere. The interface is identical with that of the module *Out*: thus you can write *Out.Int(n, w)* to output an integer on the standard output, and *Err.Int(n, w)* to output it on the standard error channel.

### 3.5 Module *Files*: File input/output

TYPE File = POINTER TO FileDesc;

- This type represents a file open for reading or writing.

VAR stdin-, stdout-, stderr-: File

- These three read-only variables of type *File* are set to the standard input, output and error channels.

PROCEDURE Open(name: ARRAY OF CHAR;  
mode: ARRAY OF CHAR): File;

- This procedure opens a named file for input, output or both, or returns *NULL* if the file cannot be opened. The *mode* argument takes the same form as the corresponding argument of the C function *fopen*. Under MS-DOS, appending *b* to the mode causes the file to be opened in binary mode, with no translation of CR/LF sequences.

PROCEDURE Close(fp: File);

- Close the file *fp*.

PROCEDURE Eof(fp: File): BOOLEAN;

- Test whether the file *fp* is positioned at the end of the file.

PROCEDURE Flush(fp: File);

- Complete any pending I/O operations on file *fp*.

PROCEDURE ReadChar(f: File; VAR c: CHAR);

- Read a single character from file *f*, returning it in the variable *c*.

PROCEDURE WriteInt(f: File; n: INTEGER; w: INTEGER);

- Write the integer  $n$  to file  $f$  in decimal using a field of width at least  $w$ .

PROCEDURE WriteReal(f: File; x: REAL);

- Write a real number  $x$  to file  $f$ .

PROCEDURE WriteLongReal(f: File; x: LONGREAL);

- Write a double-precision real  $x$  to file  $f$ .

PROCEDURE WriteFixed(f: File; x: LONGREAL;  
wid: INTEGER; dec: INTEGER);

- Write real number  $x$  to file  $f$  in a field of width  $wid$ , with  $dec$  digits after the decimal point.

PROCEDURE WriteChar(f: File; c: CHAR);

- Write the character  $c$  to file  $f$ .

PROCEDURE WriteString(f: File; s: ARRAY OF CHAR);

- Write the string  $s$  for file  $f$ .

PROCEDURE WriteLn(f: File);

- Write a newline character to file  $f$ .

PROCEDURE Read(f: File; VAR buf: ARRAY OF SYSTEM.BYTE);

- Read characters from file  $f$  into  $buf$ .

PROCEDURE Write(f: File; VAR buf: ARRAY OF SYSTEM.BYTE);

- Write characters from  $buf$  to file  $f$ .

CONST SeekSet = 0; SeekCur = 1; SeekEnd = 2;

- Constants for use with *Seek*.

PROCEDURE Seek(f: File; offset: INTEGER; whence: INTEGER);

- Position the file  $f$  at a given offset from the beginning of the file (if  $whence = SeekSet$ ), from the current position (if  $whence = SeekCur$ ) or from the end of the file (if  $whence = SeekEnd$ ).

PROCEDURE Tell(f: File): INTEGER;

- Return the current position of file  $f$ .

### 3.6 Module *Math*: Mathematical functions

This module contains various mathematical functions on floating-point numbers.

PROCEDURE Sqrt(x: REAL): REAL;

- Compute the square root of the argument.

PROCEDURE Sin(x: REAL): REAL;

- The sine function.

PROCEDURE Cos(x: REAL): REAL;

- The cosine function.

PROCEDURE Tan(x: REAL): REAL;

- The tangent function.

PROCEDURE *Arctan2*(y, x: REAL): REAL;

- The function  $\text{Arctan2}(y, x) = \tan^{-1}(y/x)$ . It is defined even when one of the arguments is zero, and uses the signs of both arguments to determine the quadrant of the result. The order of the arguments is traditional.

PROCEDURE *Exp*(x: REAL): REAL;

- The exponential function  $e^x$ .

PROCEDURE *Ln*(x: REAL): REAL;

- The natural logarithm function.

CONST pi = 3.1415927;

- (Approximately.)

All angles are expressed in radians.

### 3.7 Module *MathL*: Mathematical functions for *LONGREAL*

This module provides exactly the same functions and constants and the module *Math*, but for arguments and results of type *LONGREAL*.

### 3.8 Module *Args*: Program arguments

This module provides access to the command-line arguments given when the Oberon program was started.

VAR *argc*: INTEGER;

- This read-only variable gives the number of arguments, including the program name.

PROCEDURE *GetArg*(n: INTEGER; VAR s: ARRAY OF CHAR);

- Copy the  $n$ 'th argument into the string variable  $s$ . The arguments are numbered from 0 to  $\text{argc} - 1$ , with the name of the program being argument 0.

### 3.9 Module *Random*: Random numbers

This module provides a pseudo-random number generator. Unless the procedure *Randomize* is called, the sequence of numbers will be the same on each run of the program. Actually, this is quite useful, because it makes the results reproducible. The module is implemented with a portable generator written entirely in Oberon, and does not use the generator provided by the standard C library.

PROCEDURE *Random*(): INTEGER;

- Generate a random integer, between 0 and *MAXRAND* inclusive.

PROCEDURE *Roll*(n: INTEGER): INTEGER;

- Generate a random integer, uniformly distributed between 0 and  $n - 1$  inclusive. The result is accurate only if  $n$  is fairly small.

PROCEDURE *Uniform*(): REAL;

- Generate a random real, uniformly distributed on  $[0, 1]$ .

PROCEDURE Randomize;

- Initialize the random number generator so that it gives a different sequence of pseudo-random numbers on each run of the program.

CONST MAXRAND = 07FFFFFFFH;

- This constant (equal to  $2^{31} - 1$ ) is the largest number that can be returned by *Random*.

### 3.10 Module *XYplane*: Simple bitmap graphics

Under X windows, this module provides a very simple monochrome graphics facility.

CONST W = 640; H = 480;

- These constants give the width and height of the graphics window in pixels. The origin is in the bottom left hand corner.

PROCEDURE Open;

- Open the graphics window.

PROCEDURE Clear

- Clear the graphics window to all white.

CONST erase = 0; draw = 1;

- These constants are used as the *mode* parameter of *Dot*.

PROCEDURE Dot(x, y, mode: INTEGER);

- Draw (*mode = draw*) or erase (*mode = erase*) a single pixel at coordinates  $(x, y)$ .

PROCEDURE IsDot(x, y: INTEGER): BOOLEAN;

- Test whether a pixel has been drawn at coordinates  $(x, y)$ .

PROCEDURE Key(): CHAR;

- Test whether a key has been pressed in the graphics window. If so, return the character that was typed; otherwise, return the null character *0X*.

The procedure *Key* allows simple keyboard interaction. It also handles the events generated by X when the graphics window is uncovered, so as to fill in the newly-exposed region; this means that a graphics application should call *Key* in each iteration of its main loop.

### 3.11 Module *Conv*: Numerical conversions

This module provides an interface to the procedures for converting between numbers and strings that are used for input/output.

PROCEDURE IntVal(s: ARRAY OF CHAR): INTEGER;

- Return the integer value of a string.

PROCEDURE RealVal(s: ARRAY OF CHAR): REAL;

- Return the real value of a string.

PROCEDURE ConvInt(n: INTEGER; VAR s: ARRAY OF CHAR);

- Convert an integer into a decimal string.

### 3.12 Module *String*: Operations on strings

This module provides useful operations on strings.

PROCEDURE Length(*s*: ARRAY OF CHAR): INTEGER;

- Return the length of *s* up to the first null character.

PROCEDURE Insert(*src*: ARRAY OF CHAR; *pos*: INTEGER;  
VAR *dst*: ARRAY OF CHAR)

- Insert the string *src* before the character with index *pos* in the string *dst*.

PROCEDURE Append(*src*: ARRAY OF CHAR; *dst*: ARRAY OF CHAR)

- Insert the string *src* at the end of the string *dst*.

PROCEDURE Delete(VAR *src*: ARRAY OF CHAR; *pos*, *n*: INTEGER)

- Delete *n* characters from the string *src* starting with the character at index *pos*.

PROCEDURE Replace(*src*: ARRAY OF CHAR; *pos*: INTEGER;  
VAR *dst*: ARRAY OF CHAR)

- Insert the string *src* into the string *dst* beginning at index *pos*, replacing an equal number of characters from *dst*.

PROCEDURE Extract(*src*: ARRAY OF CHAR; *pos*, *n*: INTEGER;  
VAR *dst*: ARRAY OF CHAR)

- Copy into the string *dst* up to *n* characters of string *src*, beginning at index *pos*.

PROCEDURE Pos(*pat*, *src*: ARRAY OF CHAR; *pos*: INTEGER): INTEGER

- Search the string *src* for the string *pat*, beginning at index *pos*. Return the index of the first occurrence found, or  $-1$  if there is none.

PROCEDURE Cap(VAR *src*: ARRAY OF CHAR)

- Convert letters in the string *src* to upper case.

### 3.13 Module *Bit*: Bitwise operations on integers

This module provides various operations that treat integers as arrays of 32 bits.

PROCEDURE And(*x*, *y*: INTEGER): INTEGER;

- Bitwise AND: bit *i* of the result is 1 if bit *i* is 1 in both *x* and *y*.

PROCEDURE Or(*x*, *y*: INTEGER): INTEGER;

- Bitwise OR: bit *i* of the result is 1 if bit *i* is 1 in either *x* or *y* or both.

PROCEDURE Xor(*x*, *y*: INTEGER): INTEGER;

- Bitwise XOR: bit *i* of the result is 1 if bit *i* is 1 in either *x* or *y*, but not both.

PROCEDURE Not(*x*: INTEGER): INTEGER;

- Bitwise NOT: bit *i* of the result is 1 if bit *i* of *x* is 0.

### 3.14 Module *Coroutines*: Multiple threads

This module allows a simple kind of multi-threading with explicit transfer of control.

TYPE Coroutine;

- Variables of this type hold the state of suspended threads.

TYPE Body = PROCEDURE;

- The body of each coroutine is represented by a parameterless procedure. This procedure must never return.

PROCEDURE Init(body: Body; stksize: INTEGER; VAR cor: Coroutine);

- This procedure initializes the coroutine *cor* so that when started it will execute the procedure *body*. A stack of size *stksize* bytes will be allocated for the coroutine.

PROCEDURE Transfer(VAR from, to: Coroutine);

- Save the current coroutine state in *from* and transfer to coroutine *to*. The call *Transfer(p, p)* is allowed.

## Language extensions

The `-x` flag enables a number of language extensions:

- (1) A function may be called as a proper procedure; the result returned by the function is discarded.
- (2) Enumeration types are supported, with the syntax

```
TYPE colour = (red, blue, green);
```
- (3) CASE statements are allowed for any discrete type (including enumeration types), not just *INTEGER* and *CHAR*.
- (4) FOR statements are also allowed for any discrete type.
- (5) The function *ORD* may be applied to any discrete type, not just *CHAR*.

---

## Interfacing with C code

The architecture of the run-time system allows free mixing of subroutines that are implemented as bytecode and as native code compiled from C. This makes it easy to add primitives written in C, and even to have primitives that in turn call back routines written in Oberon. The easiest way to incorporate C primitives is to link them with OBC's runtime system to make a customized virtual machine. This is done by compiling with the `-C` flag as described below. It's also possible (at least under UNIX) to load primitives as a shared library into the standard virtual machine. This is done for the primitives that implement the *XYPlane* module in the Oberon library, so that the virtual machine does not depend directly on the X libraries, and they need not be loaded or even installed unless they are really needed.

I haven't the time now to document everything about the runtime system needed to write new primitives, so I hope the examples that follow will provide just enough information for readers who want to explore for themselves. As a simple example, let's implement a primitive *Ten* that multiplies its integer argument by 10. To illustrate the use of primitives to access UNIX system calls, let's also implement a primitive *Time* with heading

```
PROCEDURE Time(VAR t: ARRAY OF CHAR);
```

that stores into the array *t* the current time as a string. Finally, we'll implement a procedure

```
PROCEDURE Alarm(d: INTEGER; p: PROCEDURE);
```

that returns immediately, but arranges for procedure *p* to be called after a delay of *d* seconds. Many further examples of C primitives can be found in the Oberon library.

In the Oberon program, these procedures are declared by giving their heading and the name of a C function that implements them. Figure 5.1 shows a single module that declares and uses our three functions. It's also possible to declare the functions and export them for use by other modules, and many of the modules of the Oberon library are implemented like this.<sup>1</sup>

It's also necessary to define the C functions that implement the primitives; Figure 5.2 shows code that does this. Each C function that implements a primitive receives two arguments *cp* and *sp*; usually *cp* is not used, but *sp* is the stack pointer for the abstract machine's evaluation stack. Conventionally, the initial value of *sp* is saved as the base pointer *bp* for the primitive, and arguments are found of the stack at fixed offsets from *bp*. Each argument corresponds to one or more words of type *value*, with the first word of the first argument at *bp[3]*.

---

<sup>1</sup> In the system source code, the C code appears as special comments interspersed with the Oberon procedure headings, and a preprocessor separates the two before compiling them.

```

MODULE Example;
IMPORT Out;
PROCEDURE Ten(n: INTEGER): INTEGER IS "Ten";
PROCEDURE Time(VAR t: ARRAY OF CHAR) IS "Time";
PROCEDURE Alarm(d: INTEGER; p: PROCEDURE) IS "Alarm";
VAR stage, count, limit: INTEGER;
PROCEDURE PrintTime;
  VAR buf: ARRAY 30 OF CHAR;
BEGIN
  Time(buf); Out.String(buf);
  INC(stage)
END PrintTime;
BEGIN
  stage := 0; count := 0; limit := 10;
  PrintTime;
  Alarm(2, PrintTime);
  WHILE stage < 2 DO
    INC(count);
    IF count = limit THEN
      Out.Int(count, 0); Out.Ln;
      limit := Ten(limit)
    END
  END
END Example.

```

Figure 5.1: Module Example

Any result returned by the primitive is pushed onto the stack, and the result of the C function is the value of the stack pointer after any result has been pushed.

The type *value* is defined in *obx.h*:

```

typedef union {
  int i;
  float f;
  value *p;
  uchar *x;
  primitive *z;
} value;

```

The implementation of *Ten* is trivial: all that is necessary is to access the argument as *bp[β].i*, then push ten times that value onto the stack as the result.

The implementation of *Time* is also straight-forward; the UNIX calls *time* and *ctime* are used to obtain the time as a string, and this string is copied into the argument variable. The code uses *strncpy* to handle the possibility that the length of this variable (passed as an additional argument for each open array parameter) might be smaller than is needed to store the whole string.

The implementation of *Alarm* illustrates the use of the function *callback* to implement a spontaneous call to an Oberon procedure that is passed as an argument.

To compile this program, we need the following steps. First, compile the Oberon code in *Example.m* to obtain the byte-code file *Example.k*:

```
$ obc -c Example.m
```

```

#include <obx.h>
#include <time.h>
#include <signal.h>

PUBLIC value *Ten(value *cp, value *sp)
{
    value *bp = sp;
    int n = bp[3].i;          /* Fetch first argument */
    (*--sp).i = 10 * n;      /* Push result */
    return sp;
}

PUBLIC value *Time(value *cp, value *sp)
{
    value *bp = sp;
    char *t = bp[3].x;       /* Fetch address of array */
    int tlen = bp[4].i;      /* Fetch length of array */
    time_t now = time(NULL);
    char *s = ctime(&now);
    obcopy(t, s, tlen);      /* Copy string into array */
    return sp;
}

PRIVATE value *alarm_proc;

PRIVATE void handler(int sig)
{
    callback(alarm_proc, 0); /* Call the saved procedure */
}

PUBLIC value *Alarm(value *cp, value *sp)
{
    value *bp = sp;
    int d = bp[3].i;         /* Fetch first argument */
    value *f = bp[4].p;      /* Fetch second argument */

    alarm_proc = f;         /* Save procedure */
    signal(SIGALRM, handler);
    alarm(d);
    return sp;
}

```

**Figure 5.2:** *File prims.c*

Next, the file *prims.c* must be compiled to obtain *prims.o*. This is done using **obc** as an interface to the system C compiler:

```
$ obc -c prims.c
```

Finally, the two files of object code must be linked together:

```
$ obc -o example -C Example.k prims.o
```

The **-C** flag causes the bytecode to be bound together with a special-purpose runtime system that incorporates the new primitives.