Oberon Pi System User Guide (PDF version)

This document is a user guide for the *Oberon* system.

To continue reading this document you need to be able to do the following things in Oberon:

1. Scroll the document up and down in its window.
2. Exit Oberon, which returns you to the Raspberry Pi operating system.

To scroll this document up or down, first move the mouse pointer into the document's scroll bar. In Oberon, the scroll bar is the narrow vertical column that appears to the left of the document text.

While the mouse pointer is inside the scroll bar, click the *left* mouse button to scroll the document down. Click the *right* mouse button to scroll it back up.

To scroll the document by fewer lines of text, click near the *top* in the scroll bar. To scroll by more lines at once, click near the *bottom* in the scroll bar.

Finally, to exit Oberon, left-click on the Close button ("x") in the upper-right corner of the Oberon system window . The window then closes, leaving you in the Raspberry Pi operating system.

The next time you start Oberon, the document you're reading now will automatically reappear, so you can continue reading it.

WHAT IS OBERON?

Oberon is a complete software environment for personal computers. It includes both the Oberon operating system and the Oberon programming language.

This name-sharing is no accident: Oberon is unusual for how tightly it integrates the operating system with the programming language. The language was expressly designed to write the system, while the system is structured to reflect key features in the language.

Here are the original goals for Oberon, as expressed by its designers:

- *Make it simple*. Oberon includes only the basic features necessary to support facilities such as windowing systems and object-oriented programming.

- *Make it small*. One person can learn and understand the entire software system. Oberon consists of less than 15,000 lines of source code (which is included as part of the system). And the compiled code is less than 200 kB.

- *Make it extensible*. The Oberon operating system is structured as a set of modules, with each module providing a specific operating system service (files, menus, etc.). But because modules are a key feature in the Oberon language, users can develop additional modules which have the same functionality as the operating system modules. This makes the Oberon system fully extensible, even at runtime.

- *Make it productive*. Oberon offers one-click access to the most common commands for any task. The compiler is fast, and no linking is required. One of the designers of the Go programming language used Oberon while he was a graduate student, and has described the system as "extremely productive".

- *Make it useful*. Oberon is a stable and reliable platform for developing and delivering large applications. It has been used on a daily basis by both technical and non-technical users. The Oberon system has been ported to Intel, ARM, RISC-V, and custom hardware processors.

Sources

"Project Oberon: The Design of an Operating System, a Compiler, and a Computer, Revised Edition 2013". Niklaus Wirth and Jurg Gutknecht.

"GopherCon 2015: Robert Griesemer – The Evolution of Go". YouTube. https://youtu.be/0ReKdcpNyQg?si=ua3WP4PUJLhDWXwU

WHY SHOULD I LEARN IT?

Oberon is a forty-year-old software system with a programming language that is now considered historical (even though most current languages offer the same features as Oberon).

So if your goal is to master today's popular languages and operating systems, then Oberon is not for you.

However, Oberon is also a complete, powerful software system written in fifteen *thousand* lines of source code. So if you are interested in studying source code to learn how system software works, then Oberon compares favorably to the thirty *million* lines of source code in the current Linux kernel.

And if you are interested in the design of user interfaces, Oberon is well worth studying as an example of a user interface which offers a distinct alternative to the standard UI paradigms of today's personal computers.

And finally, if you are interested in just learning how to use Oberon, then the version of Oberon offered here will make it easier to learn.

This version includes some changes to the Oberon user interface which will make it easier for you to learn the key parts of Oberon, by letting you avoid the non-key parts that are known to be difficult for new users.

OBERON PI

The version of Oberon you're using here was originally the stand-alone system software for a workstation built specifically for Oberon. This workstation was based on a custom RISC processor which was also built for Oberon. The result was called the "Oberon RISC Machine".

The RISC Machine version of the Oberon software was ported to the Raspberry Pi with an emulator, which in this case is a C program that executes on the Raspberry Pi operating system. This program simulates the processor, memory, and peripherals of the Oberon RISC Machine.

The compiler in this version of Oberon generates object code for the RISC processor, and it is this code that gets executed by the emulator. And since the entire Oberon system has been compiled with this same compiler, it too executes on the emulator.

The emulation extends to the Oberon file system, which exists separately from the Raspberry Pi file system. The entire Oberon Pi system is stored in the Raspberry Pi file system as a single file named “Oberon-System.dsk”. This file is called the Oberon *disk image* file.

Files cannot be copied directly between the Oberon Pi and Raspberry Pi file systems. Oberon Pi includes the Clipboard module which lets you easily exchange text data between the two file systems.

Sources

"Project Oberon: The Design of an Operating System, a Compiler, and a Computer, Revised Edition 2013". Niklaus Wirth and Jurg Gutknecht.

"Oberon RISC Emulator". Peter De Wachter. GitHub. https://github.com/pdewacht/oberon-risc-emu

THIS DOCUMENT

This document is an introduction to the Oberon operating system.

It focuses on the unique aspects of the Oberon user interface, and will be particularly useful to people studying the Oberon UI.

Note that this is not a programming guide – aside from a few brief sections on language differences in the Oberon Pi system, nothing is said about programming in Oberon, or programming in general. (Such information can, however, be found in the various books provided as PDF files with the Oberon Pi release.)

What this document *is* is a tutorial on the Oberon operating system. It is designed to be read front-to-back, with the basic concepts being presented first, and more advanced topics later.

Sources

"The Oberon System: User Guide and Programmer's Manual". Martin Reiser.

"Programming in Oberon: Steps Beyond Pascal and Modula". Martin Reiser and Niklaus Wirth.

HELLO, WORLD

In this section you will view, compile, and execute a module which writes "Hello, world" in the Oberon display. You will do all this with just *three* mouse clicks.

The module output will appear in a new window which opens below the *guide window* you are currently reading.

NOTE – Whenever a new window opens below the guide window, you may need to *scroll* this document down to continue reading it.

Click #1 – In the *project window*, move the mouse pointer over the Oberon command name "Edit.Open", and click on this name with the *middle* mouse button.

(On the Raspberry Pi, the mouse scroll wheel serves as the middle mouse button. To click this button, press down on the scroll wheel.)

When you click on "Edit.Open", a new window opens below the guide window. This new window is titled "Hello.Mod", and it contains the source code for the "Hello" module.

Click #2 – In the project window, move the mouse pointer over the Oberon command "OBC.Compile", and middle-click on it.

When you click on "OBC.Compile", the following messages appear in the terminal window:

```
OB Compiler  8.1.25
compiling Hello    15     0    6D6EC60A
```

Click #3 – In the project window, move the mouse pointer over the Oberon command "Hello.Hi", and middle-click on it.

When you click on "Hello.Hi", a second window appears below the guide window. This new window is titled "Out.Text", and it displays the output that was generated by executing the code in the "Hello" module.

You have now compiled and executed "Hello World", in three mouse clicks. Next, close the windows that were opened by this process.

In the menu bar of the "Out.Text" window, move the mouse pointer over the command "Window.Close", and middle-click on it. The "Out.Text" window then closes.

Do the same with the "Hello.Mod" window, to close it too.

At this point, the Oberon display should now look the same as it did originally (except for the compiler messages remaining in the terminal window).

COMMANDS

Oberon command names always appear as two words joined by a ".":

```
Edit.Open
OBC.Compile
Hello.Hi
Window.Close
```

The first word is the name of the service that the command belongs to (Edit, OBC, Hello, Window).

The second word is the name of a specific command within the service (Open, Compile, Hi, Close).

To execute a command, move the mouse pointer over either word of the command name, then middle-click on it.

The Oberon user interface is unusual because anywhere a command name appears in the Oberon display, that name can be used to execute the command.

For example, in the previous section ("Hello, World") you executed commands that appeared in both the project window (Edit.Open) and the window menu bars (Window.Close).

Now try executing the name "Edit.Open" that appears below in the following command:

Edit.Open Hello.Mod

It worked. To close the window that just opened, execute the command Window.Close in the new window's menu bar.

Next, type the command "Edit.Open Hello.Mod" into the terminal window, and execute the command name there. Close the new window the same way you did before.

Command names can even be executed from Oberon modules.

For example “Edit.Open CmdDemo.Mod”.

While a command can be executed anywhere in Oberon, it's important to note that the command itself must be valid. This means the following:

– The first word in the command must be the name of an executable module in the Oberon system.

– The second word must be the name of a procedure declared in the specified module.

– The procedure must be declared in its module with no parameters, and must also be *exported* from its module.

For an example of a valid command, see how the procedure "Hi" is declared in the file "Hello.Mod". The asterisk after the procedure name specifies that this procedure is exported from its module.

NOTE – The command "System.ModuleCommands" lists all the commands defined in a module. For example:

System.ModuleCommands Hello

If you execute an invalid command in Oberon, different things may happen depending on the command. For example, if you execute "Edit.Open" without a parameter, the command does nothing. In other cases (such as misspelling a command), an error message will appear:

```
Call error: Eddit module not found
Call error: Openn command not found
```

If a command does not finish executing, it may prevent you from executing any more commands in the Oberon system. If this happens, use the keyboard command Ctrl-Shift-Delete to halt the runaway command, and regain control of the Oberon system.

The halt command displays a message in the terminal window, listing the memory address where the runaway command was halted:

```
ABORT   0001189C
```

WINDOWS

Oberon uses windows to display multiple files (and even multiple applications) on the screen at the same time.

But unlike conventional GUI interfaces, windows in Oberon never overlap each other. Instead, they are *tiled* to fit next to each other on the screen.

NOTE – Because tiling windows on the screen is not as space-efficient as overlapping them, Oberon should be used only on computers with relatively large screens (20" or more is recommended).

A window in Oberon consists of a rectangular area on the display, with a *menu bar* displayed (in highlighting) across the top of the window, and an optional scroll bar on the left.

The leftmost word in a menu bar is the window *title*. The words to the right of the title are Oberon commands associated with the window.

As you've already seen, when multiple windows are open in Oberon, each window may not be big enough to usefully view or edit its contents.

For this Oberon provides two commands which let you quickly enlarge a window you want to work on, and then restore that window to its previous size when you are done using it:

– "Window.Open" opens a window to its next bigger size on the Oberon display (either full height, or full screen).

– "Window.Close" closes an opened window back to its previous size and position on the display.

NOTE – Think of these commands as working like real-life windows, which can be opened and closed incrementally.

Opening a window does not affect the other windows it covers up – they will be restored on the screen when you close the window.

Closing a window that is already in its original size and position will remove the window from the display. The close command thus serves *two* functions: restoring an opened window to its previous size, and removing a window from the display.

Fortunately, if you execute one too many close commands and accidentally remove a window you wanted to keep open, the following command will reopen the removed window:

Window.Reopen

To see how all this works, open two additional windows below this one by executing the following two commands:

```
Edit.Open Filter.Mod
Sierpinski.Draw
```

Then try out the commands "Window.Open" and "Window.Close" (and if necessary, "Window.Reopen") on all three windows.

NOTE – You can move or resize a window after it has been opened. For details see "WINDOW MANAGEMENT".

FILES

Oberon stores all of its files in a single directory – there are no subdirectories or file folders. (This type of file system is known as a *flat* file system.)

To list all the files in the directory, use the following command:

```
System.Directory
```

This opens a new window titled "Directory", which appears under the project window.

Scroll the directory window, and you'll see that the Oberon system contains a long list of files.

To make the file listing more useful, close the newly-opened directory window, and then execute the following command:

```
System.Directory Hello*
```

This time the directory window lists just the files associated with the Hello module.

The wildcard character '*****' allows the Oberon file system to work more like a hierarchical file system.

You can also use the wildcard character to list all the files of a given type:

```
System.Directory *.Mod
```

The type of a file is usually indicated by adding a file type *suffix* to the end of the file name. Here are the common file type suffixes:

.Mod	module source file
.rsc	module object file
.smb	module symbol file
.Doc	module document file
.Proj	project file
.Cmd	command file
.Guide	user guide file
.Chap	user guide TOC file
.Text	text file
.Bak	backup file
.Fnt	font file
.Graph	draw graphics file
.Lib	draw library file

Module source files are conventionally named after the module they contain ("Hello.Mod" for module “Hello").

Module object and symbol files are created by the compiler when you compile a module source file.

Backup files are created by the editor whenever you save a file with the same name as an existing file.

Be sure to learn all the file type suffixes – because file names (Hello.Mod) look so much like command names (Hello.Hi), the suffixes are the only way you can tell the difference.

NOTE – The file names "Project", "Commands", “Chapters”, and “Guide" (all with no suffix) are reserved for use by the Oberon startup windows.

DOCUMENTS

Whenever an Oberon window contains text and a scroll bar, that text is effectively a *document*, and can be navigated with the document mouse commands.

You've already been using the basic mouse commands to scroll through this document as you read it. Here are *all* the mouse commands for moving around in a document:

Left Click	Scroll document down
Right Click	Scroll document up
Shift Left Click	Jump to end of document
Shift Right Click	Jump to beginning of document
Middle Click	Jump to specific part of document

Remember that these commands work only when the mouse pointer is inside a scroll bar. (If outside, they become different commands.)

When you scroll a document down, note how the specific text line that you left-click next to gets moved to the top of the window. This lets you control how many lines to scroll each time, from a single line to an entire window of text.

NOTE – If you have trouble seeing which text line is next to the mouse pointer, hold down the left mouse button and *drag* the mouse up and down inside the scroll bar. When you do this, the current text line gets underlined in the window, to show which line will move to the top when you release the mouse button.

To quickly jump to a specific part of the document, click the middle mouse button in the scroll bar. The position of the mouse pointer in the scroll bar (top, bottom, middle, etc.) determines what part of the document (beginning, end, middle, etc.) will be jumped to.

To quickly jump to the beginning or end of the document, hold down the keyboard Shift key while you click the left or right mouse button anywhere in the scroll bar. Shift-*left*-click jumps to the end of the document, while shift-*right*-click jumps to the beginning.

PROJECTS

The project window contains all the commands you regularly use to perform a specific task in Oberon. (This could be writing a book, developing software, or just the daily use of certain applications.)

As you've already seen in previous sections, you can execute any command in the project window simply by middle-clicking on the command name.

And because the project window is a *text* window, and the window contents are a text file, you can easily create new project files, or customize existing project files to make them more useful for their tasks. (This is one of Oberon's most powerful features.)

To open an additional project file, use the following command:

```
System.Open Hilbert.Proj
```

A new window opens below the project window, displaying the project file for the module named "Hilbert".

The command "System.Open" is equivalent to "Edit.Open", except that it opens the new text window on the right side of the Oberon display. Note how the title of this new window is the project file name ("Hilbert.Proj") instead of "Project".

To make your own project file appear in the project window, rename the file as "Project" (without any file suffix). When Oberon starts up, it recognizes this file name as the system project file.

NOTE – Don't rename your project file until you first rename the current system project file back to its original project file name.

TERMINAL

The terminal window is used for two things:

- As a convenient place to type in and execute commands that aren't available in the project window.
- As a place for certain commands (such as OBC.Compile) to output their status and result information.

Like the project window, the contents of the terminal window can be edited like a text file.

Thus any commands that you have already typed into the terminal window can be executed repeatedly. Or you can edit an existing command there, to create variations on the same command.

Note, however, that the terminal window differs from the other windows in several ways:

- Its menu bar has no "Edit.Save" command, so the window will be empty every time you restart the Oberon system.
- Its menu bar includes the command "Edit.Locate", which is used to locate syntax errors while compiling a module.
- Its menu bar includes the command "Window.Clear", which erases the contents of the terminal window.

Note also that the terminal window does *not* automatically scroll down when lines of output are written below the bottom of the window. With "Window.Clear" you can erase the terminal window entirely, before the output reaches the bottom.

The commands "Window.Open" and "Window.Close" let you open and close the terminal window.

If you accidentally close the terminal window, execute the command “Window.Reopen” from another text window, and the terminal window will reappear in the Oberon display.

TEXT

To enter text on the screen, move the mouse pointer into a text window, then click on the *left* mouse button.

The mouse pointer changes into a wedge-shaped pointer named the *text cursor*.

When you enter text from the keyboard, it appears onscreen next to the text cursor.

Along with entering text, the text cursor is also used to edit text, and to mark the current window for certain commands (such as Edit.Cut, Edit.Paste, and Edit.Locate).

While different in shape, the text cursor works similarly to the "I-beam" cursor used in other operating systems. It's always positioned between two characters, and shows where the text you type will be entered.

To move the text cursor left or right on the current text line, hold down the left mouse button and *drag* the mouse left or right. When you do this, the text cursor jumps from character to character along the text line.

NOTE – While you drag the text cursor left or right, it sticks to the current text line, even if you move the mouse up or down a little while you drag. To move the cursor off the current line, release the left mouse button and then move the mouse up or down.

To remove the text cursor from the screen, press the Esc key.

To see how the text cursor works, execute the following command:

```
Edit.Open MyHello.Mod
```

Then try out the text cursor and Esc key in the newly-opened window.

NOTE – The text cursor cannot be moved with the keyboard arrow keys. It can only be moved with the mouse, as described above.

SELECTION

To select text on the screen, move the mouse pointer over the beginning of the text, then press on the *right* mouse button, and drag the pointer down or to the right.

The selected text is highlighted as you drag the pointer over it, and remains highlighted when you release the right mouse button.

To *unselect* the selected text, press the Esc key.

Text selection is used for two things:

- To specify text for the standard text-editing commands (Edit.Cut, Edit.Copy).

- To specify parameters for non-editing commands that accept text selections as parameters (Edit.Search).

To quickly select a word (hello), command name (Hello.Hi), or file name (Hello.Mod), position the mouse pointer anywhere over the item and right-click *twice*, without moving the pointer.

To quickly change an existing selection, move the mouse pointer below or to the right of the beginning of the current selection, then press the Shift key, and click on the right mouse button. (This command is usually written as shift-right-click.)

To see how the selection commands work, try using them on the text in this document.

NOTE – If you try to continue selecting text below the bottom of a window, instead of scrolling the window, the selection simply stops at the bottom of the window. Making large selections is described in "WINDOW MANAGEMENT".

EDITING

To edit a document use the standard editing commands:

- Edit.Cut
- Edit.Copy
- Edit.Paste
- Edit.Undo

Edit.Cut and Edit.Copy operate on the current text selection.

Edit.Paste inserts the recently cut or copied text at the current position of the text cursor.
Edit.Undo undoes the result of the preceding cut or paste command.

NOTE – Before you use any of these editing commands, you must first set the *text cursor* in the window where you are editing. Doing this lets these commands know which window is being used. For details see “WINDOW SELECTION”.

The standard keyboard shortcuts are defined for these editing commands:

Ctrl-X	Edit.Cut
Ctrl-C	Edit.Copy
Ctrl-V	Edit.Paste
Ctrl-Z	Edit.Undo

The Edit.Undo command is limited. It works only for the cut and paste commands. Only one level of undo is supported. And the text cursor must still be in the *same* position it was after the preceding cut or paste command.

The Backspace key can be used to delete individual characters from a text. No undo is available for backspaced text.

The text editor does not perform word-wrapping. To start a new line in the text, you must use the keyboard Enter key.

SEARCH

To search for text in a document, use the Edit.Search command. This command appears in the menu bar of most windows.

To use Edit.Search, first select an instance of the string you want to search for. (If the string doesn't appear in any text window, type it into the terminal window, and select it there.)
Then execute the Edit.Search command in the menu bar of the window you're searching in.

If the search string is found, the document scrolls to show the text cursor positioned at the end of the found string.

If the search string is not found, the command does nothing.

To find all the instances of the search string in a document, repeatedly execute the Edit.Search command. Each time you execute it, the text cursor moves to the next instance of the search string in the document.

When the search command reaches the end of a document, it stops searching. (Note that it does *not* wrap around to the beginning of the document.)

The search command always starts searching at the current position of the *text cursor*. If the text cursor is not set in the document, then the search begins at the start of the document.

NOTE – Pressing the Esc key clears the text cursor.

To see how searching works, try using it to navigate the Oberon user guide you're reading now.

First, execute the Window.Open command that appears in the menu bar of the chapter window.

The opened chapter window displays the *table of contents* for the Oberon user guide.

You're now ready to jump to a different section in the user guide.

In the chapter window, first press the Esc key, then select the title of the section about Oberon books.

Next, execute the Edit.Search command that appears in the menu bar of the guide window.

The guide window will then jump to the section in the user guide that describes the Oberon books.

NOTE - When you're jumping forward in a document, you can skip pressing the Esc key. Why?

—

This section has described only how to search for text in a document. To replace text, use the Filter.Replace command. For more information "Edit.Open Filter.Doc".

SAVING

To save a document, execute the Edit.Save command that appears in the menu bar of the document's window.

The saved document is stored in a file with the same name as the title of the the document window.

The Edit.Save command can also be executed from any document that the command name appears in.

However, before you can use the command this way, you must first specify two things:

1) The document to save
2) The file name to save to

The document is specified by marking its window with the *location marker*. This is a star-shaped symbol which can be placed at any location on the display (including inside a window).

To mark a window for saving, move the Oberon mouse pointer inside the window and press the Ctrl-A key. The star symbol then appears on the screen next to the mouse pointer.

The file name to save to is specified as a parameter following the command name. For example:

```
Edit.Save Hello.Mod
```

The command parameter for the Edit.Save command can be a text selection instead of a file name. For example:

```
Edit.Save ^
```

When the file name parameter is a caret character '**^**', the command uses the current selection as the file name to save the document to.

NOTE – The caret character is commonly known as the "hat" character. It appears on the Pi keyboard as the Shift-"6" key.

If a file is saved with the same name as a file that already exists, the Edit.Save command automatically renames the existing file to be a *backup file*. This prevents the existing file from being deleted by the newly-saved file.

A backup file is created by adding the file suffix ".Bak" to the name of an existing file. For example:

```
Document
Document.Bak
Hello.Mod
Hello.Mod.Bak
```

To preserve a backup file, you must rename it with a different file name, and no ".Bak" suffix. For example:

```
System.RenameFiles Hello.Mod.Bak => Hello2.Mod ~
```

Oberon expects you to be mindful when using the file system (especially when saving a file with a different file name).

The Oberon system will *not* notify you when an existing file gets overwritten by a new file. Instead, it simply creates a backup file so you can recover the overwritten file.

But backup files themselves offer you only one chance to recover an accidentally overwritten file. If instead you perform a *second* save on a newly-saved file, this will overwrite the current backup file with a copy of the new file, and the original file will be lost.

NOTE – Whenever you save a file, a message appears in the terminal window, listing the name and size of the saved file.

OPENING

To open a document for editing, execute the command Edit.Open. For example:

    Edit.Open Hello.Mod

The command parameter ("Hello.Mod" in this case) specifies the file name of the document.

If a file with that name already exists, then the document stored in that file opens in a new window.

If the specified file does *not* exist, a new empty window opens, with the window title set to the command parameter:

    Edit.Open Mxyzptlk.Text

The Edit.Open command opens its document window on the *left* side of the Oberon display.

To open a document window on the *right* side of the display, execute the command System.Open:

    System.Open Hello.Proj

Except for where they open document windows, the Edit.Open and System.Open commands do the same thing.

NOTE – Oberon will let you open multiple instances of the same document. This should be avoided, as you may lose some changes if you happen to edit both copies of the document.

Edit.Open and System.Open normally open their windows on the left and right sides of the display, respectively.

You can override this behavior by using the Oberon *location marker* to specify where on the display that these commands will open their document window.

To do this, move the Oberon mouse pointer to a location on the display and press the Ctrl-A key. The location marker's star symbol then appears on the screen next to the mouse pointer.

Next, use Edit.Open or System.Open to open a new document window. The new window will be opened on the display at the position of the location marker.

NOTE – You can move or resize a window after it has been opened. For details see "WINDOW MANAGEMENT".

The command parameter for Edit.Open or System.Open can be a text selection instead of a file name. For example:

```
Edit.Open ^
System.Open ^
```

When the file name parameter is a caret character '**^**', the command uses the current selection as the file name of the document to open.

NOTE – The caret character is commonly known as the "hat" character. It appears on the Pi keyboard as the Shift-"6" key.

WINDOW SELECTION

The Oberon system is unusual because anywhere a command name appears in the Oberon display, that name can be used to execute the command.

Oberon can also display multiple windows at the same time.

Taken together, these two properties raise a user interface question: when a command is executed, which window does it apply to?

The Oberon system provides several features to manage this issue:

- Menu commands
- Focus window
- Marked window

Menu commands are the command names listed across the top of a window. These commands are defined to always operate on the window they belong to.

The *focus window* is the window that all of the Oberon content commands operate on. This includes the cut, copy, and paste commands, along with all the keyboard commands that manipulate the data in a window.

To specify a window as the focus window, set the *text cursor* anywhere in the window (by left-clicking the mouse).

The *marked window* is the window that certain Oberon commands operate on. This includes the Clipboard commands, Edit.Save when it is used outside of a menu bar, and other commands that operate on entire windows.

To specify a window as the marked window, set the *location marker* anywhere in the window (by pressing the Ctrl-A key).

The focus window and marked window share an important property:

- If you try to use a command that requires a focus or marked window, but without first setting the window you're working on as focus or marked, then the command in question will either not work at all, or will work in unexpected ways (because it accessed a different window than the one you expected).

The most dramatic case of not setting the focus window involves the commands Edit.Copy and Edit.Paste. New Oberon users assume that selecting text in a window is enough for Edit.Copy to properly copy the selected text.

But this assumption is not true – a text selection does *not* set the focus window. This can only be done by setting a text cursor in the window *before* using Edit.Copy to copy the text selection.

What happens if you don't do this is that Edit.Copy appears to copy the selected text. But when you then use Edit.Paste to paste the supposedly copied text, the text that gets pasted is completely different from the text you thought you copied.

This different text is in fact the contents of the clipboard from a *previous* Edit.Cut or Edit.Copy command (which was properly done on the text in a focus window).

Once you develop the habit of setting the focus window before working on it, this kind of scenario will no longer happen to you.

The escape key (Esc) erases the current text cursor, location marker, and text selections from the Oberon display.

NOTE – Esc unfocuses the current focus window. But it does *not* unmark the current marked window. The only way to do this is to set the location marker in a different window.

Location markers specify locations in the *display*. If you move a window after setting a location marker in it, the location may no longer be in the moved window.

NOTE – For more information on moving a window, see "WINDOW MANAGEMENT".

FILE MANAGEMENT

The Oberon system includes commands used to perform the following file system operations:

- Copying/renaming/deleting files
- Listing file attributes (date, size)

To *copy* one or more files in the file system, use the command System.CopyFiles. For example:

```
System.CopyFiles  Hello.Mod  =>  Temp.Mod   ~
System.CopyFiles  A.Text => X.Text   B.Text => Y.Text   ~
```

This command accepts a parameter list of one or more *file name transforms*.

Each transform consists of three parts: the name of the file to copy ("Hello.Mod"); the transform operator ("=>"); and the name of the copied file ("Temp.Mod").

NOTE - The parameter list must end with a tilde character (‘**~**’).

Each of the specified files is copied in the file system, with the copies having their assigned file names.

When a copy is successful, a status message appears in the terminal window. For example:

```
copying
  Hello.Mod => Temp.Mod
```

To *rename* one or more files in the file system, use the command System.RenameFiles. For example:

```
System.RenameFiles  Helo.Mod  =>  Hello.Mod   ~
System.RenameFiles  A1.Text => A2.Text   B1.Text => B4.Text   ~
```

This command is identical to System.CopyFiles, except that it renames the specified files instead of copying them.

To *delete* one or more files from the file system, use the command System.DeleteFiles. For example:

```
System.DeleteFiles  Temp.Mod   ~
System.DeleteFiles  Filter.rsc  Old.Text  Draft.Doc   ~
```

This command accepts a parameter list of one or more *file names* (not file name transforms).

NOTE - The list must end with a tilde character (‘**~**’).

Each of the specified files is deleted from the file system.

When a delete is successful, a status message appears in the terminal window. For example:

```
deleting
  Temp.Mod
```

The file management commands *must* be used with care:

- They perform minimal error checking on their command parameters.

- If a file with the same name already exists, the copy and rename commands will overwrite the existing file.

- No backup files are created for any files overwritten or deleted.

- If the tilde character (‘**~**’) is omitted on a command that accepts a parameter list, the command will treat any text in the window that follows the command – even on subsequent text lines – as additional parameters to be processed by the command. This can lead to unexpected results (especially when using the delete command).

The System.Directory command is used to list the files stored in the file system. For example, the following command:

```
System.Directory Hello*
```

... lists these files in the directory window:

```
Hello.Mod
Hello.Proj
```

NOTE – For details on System.Directory see "FILES".

To list the file date and size along with the file names, add an exclamation mark character (‘**!**’) to the end of the System.Directory command parameter. For example:

```
System.Directory Hello*!
```

... creates the following directory listing:

```
Hello.Mod      23-12-17  02:38:00    145
Hello.Proj     23-10-22  21:50:37    370
```

The date shows when the file was last updated.

The size shows the file size (in bytes).

NOTE – For details on file dates see “UTILITIES".

—

WINDOW MANAGEMENT

The Oberon system includes commands used to perform the following window operations:

- Moving windows
- Splitting windows
- Making large selections

To *move* a window up or down in the Oberon display, first position the mouse pointer anywhere in the window's menu bar, then press and hold the left mouse button.

The highlighting disappears from the menu bar, to show that the window is ready to move.

Next, move the mouse pointer up or down, to where you want the window menu bar to appear in the display.

NOTE – While you can move the mouse pointer anywhere in the display, the window can only be moved up or down from its current position.

Finally, release the mouse button. The window menu bar reappears at the specified location.

To *split* a window in the Oberon display, execute the Window.Split command that appears in the window's menu bar.

This command opens a new window which has the same title and document text as the original window.

But this new window does *not* contain a copy of the original document. Instead, it provides a separate view into the *same* document that is in the original window.

With split windows, any change you make to the document in one window will automatically appear in the other window.

When you close a split window, only the specified window gets closed. The other window remains open, allowing you to continue editing the document.

NOTE – Split windows are useful when you need to view one part of a document while editing another part. They are also used to make large selections in a document.

The Oberon system has a restriction which makes working with selections more difficult than in other operating systems. Namely:

- A text selection cannot be larger than the window that contains it.

If you try to continue selecting text below the bottom of a window, instead of scrolling the window, the selection simply stops at the bottom.

To make a *large selection* in a document, first execute the Window.Split command in the document's window.

This opens a new window which provides a separate view into the document.

Next, scroll the topmost split window to where the beginning of the large selection should be, and right-click on the beginning text there. As usual, the text you selected is highlighted to show it has been selected.

Now scroll the *other* split window to where the *end* of the large selection should be, and right-click on the ending text there. As before, the text you selected is highlighted to show it has been selected.

At this point you have created a large selection. It includes not only the highlighted text in the two selections in the two split windows, but also all the *unhighlighted* text in the document that lies between the two selections.

The large selection can now be cut, copied, and pasted as if it were a single regular text selection.

NOTE – The two regular selections that define a large selection can themselves be any size (as long as they fit in their windows). Also, the cut and copy commands require the text cursor to be set in one of the split windows.

COMMAND PARAMETERS

Most commands in the Oberon system require *parameters*.

Depending on the command, these may be specified in several ways:

- One or more text items following the command name
- A text selection
- A window selected with a text cursor or location marker

This section describes the text items that commonly follow a command name.

Six types of text items follow a command name:

- File name
- Module name
- Word transform
- File name list
- Module name list
- File name transform list

A *file name* is accepted as a parameter by many commands. For example:

```
Edit.Open Hello.Mod
System.SetFont Oberon8.Scn.Fnt
Edit.Save ^
```

When the file name parameter is specified as a caret character '**^**', the command uses the current selection as the file name.

NOTE – The caret character is commonly known as the "hat" character. It appears on the Pi keyboard as the Shift-"6" key.

A *module name* is accepted as a parameter by one command:

```
System.ModuleCommands Edit
```

The parameter must be a module name ("Edit"), not a module file name ("Edit.Mod").

A *word transform* is accepted as a parameter by one command:

```
Filter.Replace village => town
```

The transform consists of three parts: the word to be transformed ("village"); the transform operator ("=>"); and the transformed word ("town").

The words in the transform are specified without any delimiting quote characters.

NOTE - For more information on the Filter.Replace command, “Edit.Open Filter.Doc".

A *file name list* is accepted by two commands:

```
System.DeleteFiles  Old.Text  Draft.Doc  ~
OBC.Compile  Out.Mod  Hello.Mod  ~
```

The list can include one or more file names.

The list must end with a tilde character (‘**~**’).

A *module name list* is accepted by one command:

```
System.UnloadModules  Hello  Out  ~
```

The list can include one or more module names.

The list items must be module names ("Hello"), not module file names ("Hello.Mod").

The list must end with a tilde character (‘**~**’).

A *file name transform list* is accepted by two commands:

```
System.CopyFiles  Hello.Mod  =>  Temp.Mod   ~
System.RenameFiles  Memo.Text  =>  Save.Text   ~
```

The list can include one or more file name transforms.

Each transform consists of three parts: the name of the file to transform ("Hello.Mod"); the transform operator ("=>"); and the name of the transformed file ("Temp.Mod").

The list must end with a tilde character (‘**~**’).

Oberon includes a *command file* which lists all the Oberon commands and their parameters.

Keeping this file open on the display gives you a quick reference for the commands. It is also useful as a source for creating *project files*.

For more information on the command file see "DISPLAY".

DISPLAY

The *display* refers to all the text and graphics that the Oberon system displays on your computer screen. It is the visible part of the Oberon *user interface* (or "UI").

A common misimpression of the Oberon system is that it has a *text user interface*, a type of UI that was common on personal computers before the *graphical user interface* became popular.

In fact, the Oberon system has a graphical user interface:

```
Sierpinski.Draw
```

The reason for the confusion is that the Oberon designers purposely set out to create a hybrid of the traditional text user interface and graphical user interface, with the goal of creating a UI that is more productive than both.

The Oberon display uses windows to display multiple documents (and even multiple applications) on the screen at the same time.

But unlike conventional graphical user interfaces, windows in Oberon never overlap each other. Instead, they are *tiled* to fit next to each other on the screen.

The display is divided by a vertical line into two parts, which are called *tracks*. Windows can appear in either track.

The *user track* is the larger area on the left side of the display. This is where users create windows to perform their computing work. (The document you're reading here is in the user track.)

The *system track* is the smaller area on the right side of the display. It is used to present system information which supports the computing work done in the user track.

The terminal window always appears in the system track. The project, command, and chapter windows conventionally appear there too.

The commands Edit.Open and System.Open are functionally equivalent, but one opens windows in the user track, while the other opens windows in the system track.

When the Oberon system first starts up, the only window that always appears in the display is the *terminal* window.

The others – namely, the guide, project, command, and chapter windows – are optional startup windows whose appearance and contents are user-defined:

- If a text file is named "Guide" (with no file suffix), it will appear in the user track as the *guide window* when the system starts up.

- If a text file is named "Project" (with no file suffix), it will appear in the system track as the *project window* when the system starts up.

- If a text file is named "Commands" (with no file suffix), it will appear in the system track as the *command window* when the system starts up.

- If a text file is named "Chapters" (with no file suffix), it will appear in the system track as the *chapter window* when the system starts up.

Copies of the default guide, project, command, and chapter files are available in the files “Oberon.Guide", "Hello.Proj", “Oberon.Cmd”, and “Oberon.Chap”.

The Oberon display normally appears as a window (named "Oberon") in the Raspberry Pi operating system.

You can change the Oberon display to appear in full-screen mode by pressing the F11 key. Pressing this key again returns the display to window mode.

NOTE – This key appears on the Pi keyboard a the Fn-"F1" key.

—

MOUSE

The Oberon system requires a three-button mouse, with the three buttons referred to as the *left*, *right*, and *middle* button.

On the Raspberry Pi, the mouse scroll wheel serves as the middle button. To click this button, press and release the scroll wheel.

NOTE – The scroll wheel works only as a mouse button, and cannot scroll an Oberon document.

Oberon defines separate commands for each mouse button (left, right, middle) while clicking or dragging the mouse.

The Shift key is used in some mouse commands.

Left Click	Set text cursor
Middle Click	Execute command
Right Click	Select text
Right Click Twice	Select word/command name/file name

Left Drag	Move text cursor one character left or right
Middle Drag	Change underlined word/cmd name/file name
Right Drag	Extend text selection

Shift Middle Click	Execute command (after unloading module)
Shift Right Click	Change text selection

The document navigation commands work only when the mouse pointer is inside a scroll bar. (If outside, these commands become different mouse commands.)

Left Click	Scroll document down
Right Click	Scroll document up
Shift Left Click	Jump to end of document
Shift Right Click	Jump to beginning of document
Middle Click	Jump to specific part of document
Left Drag	Underline current text line

KEYBOARD

In the Oberon system the *keyboard* is used for the following:

- Entering text
- Certain mouse commands (with the Shift key)
- Keyboard shortcuts for basic editing commands
- Keyboard commands for various system operations

The escape key (Esc) performs a specific operation in the Oberon system. It erases the current text cursor, location marker, and text selections from the Oberon display.

The arrow keys are *not* used in this version of the Oberon system. All cursor movement is done with the Oberon mouse commands.

The Caps Lock key is *not* used in this version of the Oberon system. To type upper-case letters you must use the Shift key.

The Shift key is used in certain mouse commands. For details see "MOUSE".

These keyboard shortcuts are defined for the basic editing commands:

Ctrl-X	Edit.Cut
Ctrl-C	Edit.Copy
Ctrl-V	Edit.Paste
Ctrl-Z	Edit.Undo

These keyboard commands perform various operations in the Oberon system:

Ctrl-A sets the location marker at the current position of the mouse pointer.

Ctrl-Shift-Delete halts a runaway Oberon command, enabling you to regain control of the Oberon system.

Alt-F4 closes the Oberon system window. It is equivalent to clicking on the window's Close button ("x").

F11 switches the Oberon system window between window mode and full-screen mode.

F12 is equivalent to Ctrl-Shift-Delete.

NOTE – F11 and F12 appear on the Pi keyboard as Fn-F1 and Fn-F2.

CHARACTERS

The character encoding used in the Oberon system is based on the ASCII standard. Each character is encoded as a 7-bit value stored in one byte.

0	NUL	55	7	81	Q	107	k
9	TAB	56	8	82	R	108	l
13	CR	57	9	83	S	109	m
32	SP	58	:	84	T	110	n
33	!	59	;	85	U	111	o
34	"	60	<	86	V	112	p
35	#	61	=	87	W	113	q
36	$	62	>	88	X	114	r
37	%	63	?	89	Y	115	s
38	&	64	@	90	Z	116	t
39	'	65	A	91	[	117	u
40	(	66	B	92	\	118	v
41	)	67	C	93	]	119	w
42	*	68	D	94	^	120	x
43	+	69	E	95	-	121	y
44	,	70	F	96	`	122	z
45	-	71	G	97	a	123	{
46	.	72	H	98	b	124	|
47	/	73	I	99	c	125	}
48	0	74	J	100	d	126	~
49	1	75	K	101	e	127	DEL
50	2	76	L	102	f		
51	3	77	M	103	g		
52	4	78	N	104	h		
53	5	79	O	105	i		
54	6	80	P	106	j		

In Oberon the ASCII caret character (94) appears as a “hat” character (‘**^**’).

The minus character (45) appears as a dash (‘**–**’), while the underscore character (95) appears as a hyphen (‘**-**’).

Character codes 26-31 are non-standardly defined to display basic media player symbols (Forward, Stop, Back).

These symbols cannot appear in Oberon text documents, but in the Oberon language their numeric codes can be assigned to character variables (with the CHR function), which in turn can be written to the Oberon display.

Oberon text documents typically contain two of the nonprinting characters (TAB, CR).

FONTS

The Oberon system supports multiple text fonts.

This version of Oberon includes a single font family named "Oberon". Fonts in this family are available in four sizes (8, 10, 12, and 16 point), with italic and bold variants available for several sizes.

Each font is stored in a separate font file in the Oberon system:

```
Oberon8.Scn.Fnt
Oberon8i.Scn.Fnt

Oberon10.Scn.Fnt       (default system font)
Oberon10i.Scn.Fnt
Oberon10b.Scn.Fnt

Oberon12.Scn.Fnt
Oberon12i.Scn.Fnt
Oberon12b.Scn.Fnt

Oberon16.Scn.Fnt
```

The following commands are used to change the font:

```
Edit.ChangeFont  Oberon8i.Scn.Fnt
System.SetFont  Oberon12.Scn.Fnt
```

Both of these commands accept a font file name as a command parameter.

Edit.ChangeFont changes the font of the current text selection to the specified font.

System.SetFont changes the current system font to the specified font.

Edit.ChangeFont affects *only* the text in the current selection. It does not affect any other text in the Oberon system, including text that already has been typed, or will be subsequently typed.

NOTE – The "hat" character (‘**^**’) cannot be used as a command parameter with Edit.ChangeFont, because this command is already using the selection to specify the text to be changed.

System.SetFont affects only the current *setting* of the Oberon system font. It does not affect any text that has already been typed in the Oberon system. But all text subsequently typed in the Oberon system will appear in the new font.

SYMBOLS

At first glance the Oberon system may appear to be wholly text-based. But it also displays several *graphic symbols*, which show the current system state.

Most of these symbols are *pointers*, which can be moved around on the Oberon display by moving the mouse:

- Mouse pointer
- Scroll pointer
- Text cursor
- Location marker

The *mouse pointer* is an arrow-shaped pointer symbol. It is the default pointer in the Oberon system, and is mainly used to set one of the other Oberon pointers at a specific location in the display. The mouse pointer is also used to select text, and to move a window in the display.

The *scroll pointer* is similar to a mouse pointer, but with a double-headed arrow symbol that points straight up and down. The scroll pointer appears only when the mouse pointer is positioned inside a window scroll bar. It is used to scroll the contents of a window up or down.

The *text cursor* is a wedge-shaped pointer symbol. It is used to edit text, and to set the current window as the focus window. While different in shape, the text cursor works similarly to the "I-beam" text cursor used in other operating systems.

The *location marker* is a star-shaped pointer symbol. It is used to select a window (or display location) for certain Oberon commands. This pointer is used less often than the other ones.

NOTE – The location marker typically gets erased by the command it affects. It is also erased by the Esc key. But even after the marker symbol is erased, the location it originally marked is still recognized by the commands that use location markers.

Along with the pointer symbols, the Oberon system displays two other graphic symbols to show the system state:

- Position marker
- Change marker

The *position marker* is a small horizontal line which appears inside a window scroll bar. When a file is longer than its window, this line shows the relative position in the file of the text that is currently visible in the window. When you scroll the window, the line moves up or down in the scroll bar.

The *change marker* is a small hollow square which appears in the upper-right corner of an Oberon text window. When you make any editing changes in a window, the square automatically appears. It disappears when you save the file.

NOTE – The change marker reminds you to save a file before closing its window (or not to save if you accidentally change a file while viewing it).

Some Oberon applications may define additional, application-specific pointer symbols which are not described here.

The escape key (Esc) performs a specific operation in the Oberon system. It erases the current text cursor, location marker, and text selections from the Oberon display.

COMPILING

The Oberon *compiler* translates module source files into executable object files.

To compile one or more source files, use the command OBC.Compile. For example:

```
OBC.Compile  Hello.Mod  ~
OBC.Compile  H1.Mod  H2.Mod  H3.Mod  ~
```

This command accepts a parameter list of one or more file names.

NOTE – The list must end with a tilde character (‘**~**’).

The compiler displays information on the compiled modules in the terminal window. For example:

```
compiling Hello    15     0    6D6EC60A
```

If the compiler finds errors in the module source file, it lists the errors in the terminal window. For example:

```
OBC.Compile HelloBad.Mod ~

compiling HelloBad
  pos 98 no match
  pos 104 no END
  pos 113 period missing
```

To locate these errors in the module source file, you first need to open the file (if not already open), and then enlarge its window with Window.Open.

Next, set the *text cursor* anywhere in the file window. Be sure to do this before continuing.

To locate one of the listed errors ("no match"), select its position value ("98") in the terminal window, then execute the Edit.Locate command in the terminal window's menu bar.

This command moves the text cursor in the file window to the position in the source code where the compiler found the error.

To locate the other errors, repeat this procedure with their listed position values.

Position values are zero-based byte offsets into the module source file. They may become invalid if you edit the file before viewing all the errors.

NOTE – Select a position value by right-clicking twice on it.

In the above example, the error listing includes three errors: "no match", "no END", and "period missing".

The text cursor for the "no match" error points to the identifier "He". In the Oberon language, this identifier must match the identifier ("Hi") declared at the beginning of the procedure.

The text cursor for "no END" points to "End". In Oberon this keyword must be "END", not "End".

"Period missing" is an invalid error message. The compiler is not always able to recover from finding an error, and so will sometimes report additional, invalid error messages. These messages automatically disappear when you fix the preceding errors in the module source code, and recompile the module.

When the compiler successfully compiles a module, it displays the following message in the terminal window:

```
compiling Hello    15     0    6D6EC60A
```

These values are defined as follows:

- Name of module compiled (Hello)
- Number of instructions in object file (15)
- Global data size of module (0)
- Module key value (6D6EC60A)

Module keys are described later in this document.

After a module has been compiled, it can be executed directly on the Oberon system – no linking is required.

The Oberon system can compile and execute modules whose source code is embedded in a text document.

For details "Edit.Open OBC.Doc".

The Oberon compiler consists of four modules:

```
OBC   compiler
OBG   code generator
OBS   scanner
OBT   symbol table
```

Only the first module contains a user command:

```
OBC.Compile          Compile modules
```

SYMBOL FILES

The Oberon compiler translates a module source file into two files: an *object file* and a *symbol file*. For example:

- Hello.rsc
- Hello.smb

The object file (.rsc) is the executable file for the module.

The symbol file (.smb) contains information that the compiler uses to perform type checking between modules.

The Oberon language is modular. It allows you to make changes to the internals of a module X, without forcing you to recompile all of the modules that import X.

However, if you make any changes to the symbols *exported* by a module X (i.e., the module *interface*), you are then required to recompile all of the modules that import X.

The compiler uses symbol files to enforce this language rule.

When you compile a module for the first time, the compiler automatically creates a new symbol file for the module. This is shown in the status message that appears in the terminal window:

```
compiling Hallo new symbol file    15     0    6D6EC60A
```

If you change the interface of an existing module and then try to compile it, the following error message appears:

```
OBC.Compile  Hallo.Mod  ~
compiling Hallo
  pos 127 new symbol file inhibited
```

The purpose of this error is to notify you that the module interface has changed (in case you changed it by accident).

To compile a module with a changed interface, use the regular compile command, but add an "/s" option to the end of the file name parameter:

```
OBC.Compile  Hallo.Mod/s  ~
compiling Hallo new symbol file    15     4    CAC1C800
```

The "/s" option directs the compiler to replace the existing module symbol file with a new symbol file that matches the changed module interface.

NOTE – No space can appear in front of the "/s".

If the symbol file is missing for a module X, compiling any module that imports X will result in the following error:

```
pos 26 import not available
```

The error will point to the name of module X.

If any of the symbols exported by a module X are themselves declared using symbols imported from a module Y, then module X *depends* on module Y.

In this case, if a module Z imports both X and Y, then the IMPORT statement in Z must be "IMPORT Y, X", not "IMPORT X, Y".

Otherwise, compiling Z will result in the following error:

```
pos 154 invalid import order
```

EXECUTING

To execute an Oberon command, move the mouse pointer over either word of the command name, then middle-click on it:

Edit.Open
OBC.Compile
Hello.Hi
Window.Close

Anywhere a valid command name appears in the Oberon display, that name can be used to execute the command.

If the command is not visible in the display, you can type the command name into any text window, then execute it.
(This is most commonly done in the terminal window.)

If you execute an invalid command in Oberon, different things may happen depending on the command. For example, if you execute "Edit.Open" without a parameter, the command does nothing. In other cases (such as misspelling a command), an error message will appear:

Call error: Eddit module not found
Call error: Openn command not found

If you execute a command whose module X needs to be recompiled because of changes made to another module Y that X imports, an error message will appear:

Call error: X imports Y with bad key

Module keys are described later in this document.

NOTE – If you are performing the usual edit/compile/execute cycle on a module, you *must* explicitly unload the module from memory each time before you execute it:

```
System.UnloadModules  Hello  ~
```

If you forget to do this, the updated module will not execute as you expect it to.

In the Oberon system, modules remain in memory after they have been executed. If you execute a module that is already in memory, the system will *not* load the module (even if it is a newer version). It will only load a new module version after the old version has been explicitly unloaded.

For details see "MEMORY".

MODULE KEYS

When a module X is compiled, the compiler generates a *module key* for X. The key is a numeric value which is unique to the set of symbols that is exported by X.

The compiler stores this key in X's object file.

When compiling a module Y which imports X, the compiler stores X's key in Y's object file.

Finally, whenever Y is executed, the Oberon system checks that the key for X in Y matches the key for X in X itself.

If the keys do not match, an error message will appear:

```
Call error: Y imports X with bad key
```

This error indicates that you need to recompile module Y.

The Oberon language is modular. It allows you to make changes to the internals of a module X, without forcing you to recompile all of the modules that import X.

However, if you make any changes to the symbols *exported* by a module X, you are then required to recompile all of the modules that import X.

The Oberon system uses module keys to enforce this language rule.

The method the compiler uses to generate module keys ensures that the same key value will be generated *every* time you recompile a module, as long as you do not make any changes to the module's exported symbols.

A module key value will change *only* after you make some change to a module's exported symbols.

Module key values are displayed by the compiler, the symbol file decoder utility, and the object file decoder utility.

DEBUGGING

The Oberon system provides the following support for program debugging:

– Source-level runtime error messages
– ASSERT statements in the Oberon language
– User-generated debug messages in the terminal window

If the Oberon system detects a *runtime error* in an executing module, it displays an error message in the terminal window. For example:

```
TrapDemo.Hi
  pos  127  TRAP   1 in TrapDemo at 00039420
```

To locate this error in the module source file (TrapDemo.Mod), you first need to open the file (if not already open), and then enlarge its window with Window.Open.

Next, set the *text cursor* anywhere in the file window.
Be sure to do this before continuing.

To locate the runtime error in the module source code, select the position value ("127") in the error message, then execute the Edit.Locate command in the terminal window's menu bar.

This command moves the text cursor in the file window to the position in the source code where the runtime error occurred.

NOTE – Select a position value by right-clicking twice on it.

The trap number ("1") in the error message identifies the specific runtime error that occurred:

1	Array index out of bounds
2	Type guard failed
3	String too long or destination array too short
4	NIL pointer reference
5	NIL procedure variable call
6	Integer divide by zero or negative value
7	ASSERT statement failed

A runtime error message includes the following data:

- Position value (127)
- Trap number (1)
- Module name (TrapDemo)
- Trap memory address (00039420)

Traps are software interrupts which halt the execution of a module, and return control to the Oberon system.

This version of Oberon supports ASSERT statements in the Oberon language:

```
ASSERT(index < 10);
ASSERT(FALSE);
```

ASSERT is a predeclared procedure which accepts a Boolean expression as a parameter.

If the expression evaluates to FALSE, the currently-executing module halts, with a runtime error showing trap number 7.

The module "Terminal" can be used to write user-generated debug messages to the terminal window from the currently-executing module.

For example “Edit.Open TermDemo.Mod”.

MEMORY

The Oberon system includes commands used to perform the following memory operations:

- Module unloading
- Font unloading
- Garbage collection
- Memory usage

When you execute a command, the Oberon system checks to see whether the command's module is already loaded in memory:

- If so, the system performs the command by simply calling the command procedure in the already-loaded module.

- If not, the system first loads the module (and all the modules it imports, if they too are also not already loaded). The system then calls the command procedure in the now-loaded module.

Once loaded, a module *remains* in memory until you explicitly unload it.

This lets you create modules which work like the Oberon system modules, providing services to the other modules in the system.

However, two cases exist where you need to unload modules from memory:

- To regain memory space in the Oberon system
- To replace a loaded module with a newer version

To *unload* one or more modules from memory, use the command System.UnloadModules. For example:

```
System.UnloadModules  Hello  Out   ~
```

This command accepts a parameter list of one or more *module names* (not file names).

The parameter list must end with a tilde character (‘**~**’).

Each of the specified modules is unloaded from memory.

When the unload command is successful, a status message appears in the terminal window. For example:

```
unloading modules
  Hello
  Out
```

If you try to unload a module X which is imported by another module that is still loaded, X does not get unloaded. Instead, the following message appears:

```
unloading modules
  X still in use
```

NOTE – This may be avoidable by reordering the command parameters so the dependent modules get unloaded first.

The Oberon system defines a mouse command which unloads a command's module before executing the command:

Middle Click	Execute command
Shift Middle Click	Execute command (after unloading module)

This mouse command is useful when you are performing the usual edit/compile/execute cycle on a module, and need to unload the old module version each time you execute the new version.

For details see "EXECUTING".

NOTE – The Shift Middle Click command is useful only when developing a module which is not imported by any other loaded module. Otherwise, you will need to use the regular unload command to unload at once *all* of the modules dependent on the one you are working on. (Include this unload command in the project window.)

To list all the modules currently loaded in memory, use the following command:

```
System.ModulesLoaded
```

This command displays the following information in the terminal window:

```
name     desc      code      refcnt
Hello    0001D800  0001D860  0
Out      0001D2C0  0001D390  1
```

"name" is the module name.

"desc" is the memory address of the module descriptor record (declared in the Oberon system as type Modules.ModDesc).

"code" is the base memory address of the module code.

"refcnt" shows how many loaded modules import this one.

When the Oberon system first starts up, it loads into memory the default system *font* (Oberon10.Scn.Fnt). Additional fonts are loaded automatically, as they get used in documents.

To unload fonts from memory, use the following command:

    System.UnloadFonts

This command unloads all fonts from the Oberon system, except for the default system font and one additional font.

To list all the fonts currently loaded in memory, use the following command:

    System.FontsLoaded

The Oberon language uses *garbage collection* to automatically release heap memory that is no longer being used by pointer variables.

The garbage collector runs as a background task in the Oberon system, and to minimize its effect on the system performance, is run only once every twenty mouse clicks.

To make the garbage collector run immediately, use the following command:

    System.FreeMemory

To view the current *status* of the Oberon memory and disk space, use the following command:

```
System.Storage
```

This command displays the following information in the terminal window:

```
Memory (modules)   121152 bytes   2%
Memory (heap)   159488 bytes   3%
Disk     3612 sectors   5%
Tasks   1
```

"Memory" lists the amount and percentage of memory used for modules and the heap.

"Disk" lists the amount (in 1024-byte sectors) and percentage of disk storage used in the Oberon Pi disk image.

"Tasks" shows how many tasks are running in the Oberon system.

UTILITIES

This version of Oberon includes a number of commands not described elsewhere in this document.

These commands perform the following functions:

- Display current date and time
- Replace words in a document
- Display document word and line counts
- Exchange text data with host system
- Convert binary files to text data
- Decode symbol and object files
- Update system boot modules

Display current date and time

To display the current *date* and *time*, use the following command:

```
System.Date
```

This command displays the following information in the terminal window:

```
system clock
  24-01-28   23:53:16
```

The date and time value shows (in order) the year, month, day, hour, minutes, and seconds.

NOTE – The displayed date and time values are read-only, and obtained from the Raspberry Pi operating system.

Replace words in a document

To *replace* all the occurrences of a word in a document, use the following command:

```
Filter.Replace village => town
```

For details "Edit.Open Filter.Doc".

Display document word and line counts

To display the number of *words* or *lines* in a document, use the following commands:

```
Filter.WordCount
Filter.LineCount
```

For details "Edit.Open Filter.Doc".

Exchange text data with host system

To *exchange* text data between the Oberon system and the Raspberry Pi operating system, use the following commands:

```
Clipboard.Import      Import text from Pi clipboard
Clipboard.Export      Export text to Pi clipboard
```

For details "Edit.Open Clipboard.Doc".

Convert binary files to text data

To *convert* binary files to text data (and also convert the text data back to binary files), use the following commands:

```
Binary.ToText   Hello.Graph
Binary.FromText  Hello.Graph.txt
```

For details "Edit.Open Binary.Doc".

Decode symbol and object files

To *decode* the binary data in Oberon symbol and object files (including the compiled object code for Oberon modules), use the following commands:

```
OBD.DecodeSymbol  Hello.sym
OBD.DecodeObject  Hello.rsc
```

For details "Edit.Open OBD.Doc".

Update system boot modules

To update the Oberon system *boot modules* (Kernel, FileDir, Files, Modules), use the following commands:

```
OBL.Link  Modules ~
OBL.Load  Modules.bin ~
```

For details "Edit.Open OBL.Doc”.

USING OBERON

The Oberon system's unique user interface poses a learning challenge for users accustomed to today's highly-standardized operating systems.

This section presents information on the following topics:

- Learning the mouse commands
- Using windows
- Executing commands
- Setting the focus window
- Commands versus programs
- Developing code
- Backing up files
- Modifying the Oberon system

Some of the information in this section appears in previous sections of this document. It is repeated here to collect together topics related to learning Oberon.

Learning the mouse commands

Oberon Pi uses the Shift key and mouse-drag operations to define multiple *mouse commands* for each mouse button (left, right, middle). To make these commands easier to learn, they have been assigned to the mouse buttons according to their command category:

- Left set cursors
- Middle execute commands
- Right select text

The *middle* button commands are the hardest mouse commands to learn in Oberon, because on most modern operating systems (including the Raspberry Pi OS) it's the *left* button that is used to execute commands.

With time and practice, you can learn to switch between these mouse buttons as you alternate between using the Oberon Pi and Raspberry Pi systems.

Using windows

The Oberon system display is based on *windows*, and while the scroll controls in an Oberon window resemble those of a standard computer window, they differ in some non-obvious ways.

Clicking the *middle* mouse button in a window scroll bar works similarly to a standard window. Middle-click where you want to move the scroll bar's *position marker* to, and the window gets updated to display the corresponding part of the document.

But scrolling an Oberon document by smaller amounts works differently than expected. Clicking the *left* mouse button anywhere in a scroll bar always scrolls the document down. Clicking the *right* mouse button always scrolls it back up.

And *where* in a scroll bar you left- or right-click determines how much scrolling occurs. Clicking near the *top* of a scroll bar scrolls by fewer lines of text at once, while clicking near the *bottom* scrolls by more lines at once.

When a window is first created, it sometimes occupies only the bottom part of the display. In this case use Window.Open to make the window bigger. But to delete this window, you will need to use Window.Close *twice*: once to return the window to its original size, and a second time to actually delete the window.

If you accidentally delete a window using Window.Close, use the (non-menu bar) command Window.Reopen to restore the window. This will work even with the terminal window.

Executing commands

When *executing* a text command, a common user error is to middle-click on one of the command parameters, instead of on the command name itself. This typically results in an error message instead of an executed command.

Another common error is to accidentally omit the *tilde* symbol ("~") from the end of a command that requires this symbol. This can lead to unexpected results. For details see "FILE MANAGEMENT".

Setting the focus window

The *focus window* is the window that all of the Oberon content commands operate on. This includes the cut, copy, and paste commands, along with all the keyboard commands that manipulate the data in a window.

If you try to use a command that requires a focus window, but without first setting the focus window, then the command may not work as expected. For details see "WINDOW SELECTION".

Commands versus programs

The Oberon system executes *commands*, not programs.

For example, while the following code is a valid Oberon module, it is *not* executable in the Oberon system:

```
MODULE Hello;
IMPORT Terminal;
BEGIN
  Terminal.String(" Hello, world")
END Hello.
```

But the following code *is* executable (as the module command "Hello.Hi"):

```
MODULE Hello;
IMPORT Terminal;

PROCEDURE Hi*;
BEGIN
  Terminal.String(" Hello, world")
END Hi;

END Hello.
```

For a procedure to work as a module command, it must be declared *without* any parameters, and have an asterisk ("*") after its procedure name:

```
PROCEDURE Hi*;
```

Developing code

When you are *developing* a module in the Oberon system (via the usual edit/compile/execute cycle), you must explicitly *unload* the module from memory each time before you re-execute it.

If you forget to do this, the updated module will not execute as you expect it to.

In the Oberon system, modules remain in memory after they have been executed. If you execute a module that is already in memory, the system will *not* load the module (even if it is a newer version). It will only load a new module version after the old version has been explicitly unloaded.

Two commands exist for unloading modules:

- System.UnloadModules unloads the modules specified as command parameters.

- Shift Middle Click executes a module command after first unloading its module.

Note that the mouse command is useful only when developing a module (such as Hello) that is not imported by any other loaded module. Otherwise, you will need to use the regular unload command to unload at once *all* of the modules dependent on the one you are working on. (Include this unload command in the project window.)

For details see "MEMORY".

Backing up files

Backing up files is a nonstandard process in Oberon Pi, because Oberon Pi files are not stored individually in the file system of the Raspberry Pi operating system.

Instead, the Oberon Pi file system is stored in a single file named "Oberon-System.dsk", which is stored in the Raspberry Pi OS file system (in the directory "Home Folder/Oberon/DiskImage"). This single file serves as a *disk image* file for the entire Oberon Pi system.

Thus when using Oberon Pi, *three* options are available for backing up files:

1) Inside Oberon Pi, use the command System.CopyFiles to create backup copies of an Oberon file within the Oberon Pi file system.

2) Inside Oberon Pi, use the command Clipboard.Export to transfer the contents of individual Oberon Pi text files out to text files in the Raspberry Pi OS.

3) Outside Oberon Pi, use the Raspberry Pi OS to create backup copies of the Oberon disk image file.

Because of Oberon Pi's minimal support for exporting data, it's often easier to use the Raspberry Pi OS to back up a single disk image file than it is to repeatedly use the Clipboard module to transfer the text for multiple files out of the Oberon Pi file system.

NOTE – Oberon Pi disk image files are useful *only* if they contain bootable, usable versions of the Oberon Pi system. If they are *not* bootable and usable, then you will *not* be able to recover any Oberon files that are stored inside the disk image file. Thus be sure to make backup copies of a disk image file while it is still working.

Binary files can be backed up outside Oberon Pi by using the Binary module to convert them to text files, and then using the Clipboard module to transfer them to the Raspberry Pi OS.

Modifying the Oberon system

Oberon Pi includes the source code files for *all* parts of the Oberon operating system. This enables you to not only study the system internals, but also *modify* them.

Successfully modifying the Oberon system requires great care, along with a solid understanding of the affected system internals.

While the system's modular structure suggests that such modifications may not be difficult, the various system modules are in fact sufficiently interdependent that seemingly minor changes to one part of the system – for instance, character input – can result in unexpected changes throughout the system, and in the worst case render the system unusable.

Before each modification to the system (no matter how small), create a backup copy of the Oberon disk image file. Doing this will ensure that you have a working version of the Oberon system to fall back to, in case your modification breaks the system.

After each modification, recompile *all* of the affected system modules (and in the proper order) before trying to restart the modified Oberon system. Failure to do so will render the system unusable, because it will no longer be able to start up. For details see "MODULE KEYS".

NOTE – If you modify the system modules Kernel, FileDir, Files, or Modules, then after compiling the updated modules, you *must* use the Oberon boot loader utility to install the updated module code in the system boot area of the Oberon file system. For details "Edit.Open OBL.Doc".

Oberon system module dependencies:

- Kernel imports SYSTEM
- Input imports SYSTEM
- Display imports SYSTEM
- FileDir imports SYSTEM, Kernel
- Files imports SYSTEM, Kernel, FileDir
- Fonts imports SYSTEM, Files
- Texts imports Files, Fonts
- Modules imports SYSTEM, Files
- Viewers imports Display
- Oberon imports SYSTEM, Kernel, Files, Modules, Input, Display, Viewers, Fonts, Texts
- MenuViewers imports Input, Display, Viewers, Oberon
- TextFrames imports Modules, Input, Display, Viewers, Fonts, Texts, Oberon, MenuViewers
- Window imports Display, Viewers, Texts, Oberon, MenuViewers, TextFrames
- Edit imports Files, Texts, Fonts, Display, Viewers, Oberon, MenuViewers, TextFrames
- System imports SYSTEM, Kernel, FileDir, Files, Modules, Input, Display, Viewers, Fonts, Texts, Oberon, MenuViewers, TextFrames

—

The following expression presents a simplified view of the Oberon system module dependencies:

SYSTEM < Kernel < Input < Display
< FileDir < Files < Fonts < Texts
< Modules < Viewers < Oberon
< MenuViewers < TextFrames
< Window < Edit < System

This expression can be used to determine what system modules to recompile (and in what order: left-to-right) after changing the interface of a specific system module.

For example, an interface change in module Input (e.g., exporting the variable Input.Shift) requires recompiling (in order) modules Oberon, MenuViewers, TextFrames, and System.

OBERON MODULES

The Oberon system contains a large number of modules, in the following categories:

- Operating system
- Utilities
- Compiler
- Libraries
- Graphics editor
- Animation
- Mathematics
- Instruction

Operating system

Kernel	memory, disk, clock, boot
FileDir	file directory manager
Files	file manager
Modules	module loader
Display	symbols, graphics primitives
Input	mouse & keyboard primitives
Fonts	font manager
Texts	files, edit, read/write, scan
Viewers	window manager
Oberon	cursors, display, commands, tasks, event loop
MenuViewers	window moving, menu bars
TextFrames	display, editing, message handling
Window	window commands
Edit	text editor
System	commands (files, display, memory, etc.)

Utilities

Filter	echo, text replace, word/line count
Clipboard	data exchange with Raspberry Pi OS
Binary	convert binary file to text data
OBD	symbol/object file decoder
OBL	boot module updater

Compiler

OBC	compiler
OBG	code generator
OBS	scanner
OBT	symbol table

Libraries

Strings	string operations
Math	math functions
In	text input from window
Out	text output to window
Terminal	terminal output
XYplane	pixel output to window

Graphics Editor

Draw	graphics editor
GraphicFrames	display, editing, message handling
Graphics	macro & command primitives
Rectangles	rectangle command
Curves	line & curve commands
Macros	macro & macro lib commands

Animation

Stars	celestial simulacrum

Mathematics

Hilbert	draw Hilbert curve
Sierpinski	draw Sierpinski curve
Checkerboard	draw square tessellation
Permutations	permutations
MagicSquares	magic squares
PrimeNumbers	prime numbers
Fractions	reciprocals
Powers	powers of 2
Harmonic	harmonic numbers

NOTE – The source code for all the non-draw math modules is stored in the single file "OBC.Doc".

Instruction

Hello	hello world
HelloBad	syntax error example
TrapDemo	debugging example
NestDemo	nested procedure example
CaseDemo	CASE statement example
H1/H2/H3	compiling examples
TermDemo	terminal output demo
XYplaneDemo	pixel output demo
FilesDemo	file I/O demo
CmdDemo	command execute demo

OBERON BOOKS

Oberon Pi includes several documents describing the Oberon operating system and programming language:

- Oberon system user guide
- Oberon draw user guide
- Oberon system internals
- Oberon language tutorial
- Oberon language reference
- Oberon language differences
- Oberon compiler internals
- Oberon philosophy
- Oberon article

These documents are stored in the Raspberry Pi file system, in the directory "Home Folder/Oberon/Documents". They can be viewed in the Raspberry Pi operating system using the PDF Viewer in the Accessories menu.

The Oberon system user guide and draw user guide documents are written specifically for the Oberon Pi system, and describe all user-level facilities of the system and the Draw application.

The remaining system and language documents were written for the RISC workstation version of Oberon. As a result, some differences exist between these documents and the Oberon Pi software. Such differences are described in the sections of this document titled "SYSTEM DIFFERENCES" and "LANGUAGE DIFFERENCES".

The Oberon compiler internals document does not describe the Oberon release compiler. Instead it uses a smaller compiler – which supports a subset of the Oberon language – to teach compiler design. In doing so, the document serves as an excellent bridge to learning the complete Oberon compiler (whose source code is included in the Oberon Pi system).

The source code for the teaching compiler can be downloaded from https://people.inf.ethz.ch/wirth/CompilerConstruction/index.html

---

The Oberon article is noteworthy for its description of how the Oberon system was used in daily life at ETH Zurich, serving the departmental computing needs for research, education, and administration. It describes a relationship between user and software which differs radically from the conventional divide that exists between software developers and users.

NOTE – Each Oberon Pi PDF document displays a table of contents in its document sidebar. If the TOC does not appear in an opened document, select the menu command View > Docks in the Raspberry Pi PDF Viewer, then select the "Outline" checkbox.

Here are the sources for the Oberon Pi documents:

Oberon system user guide
Document provided inside Oberon Pi system.

Oberon draw user guide
Document provided inside Oberon Pi system.

Oberon system internals
"Project Oberon". Niklaus Wirth, Jurg Gutknecht.
Posted in three parts (chapters 1-9, 10-15, 16-17) on
https://people.inf.ethz.ch/wirth/ProjectOberon/

Oberon language tutorial
"Programming in Oberon (a Tutorial)." Niklaus Wirth. Posted on
https://people.inf.ethz.ch/wirth/Oberon/index.html

Oberon language reference
"The Programming Language Oberon-07 (Revised Oberon)".
Niklaus Wirth. Posted on
https://people.inf.ethz.ch/wirth/Oberon/index.html

Oberon language differences
"Differences between Oberon-07 and Oberon".
Niklaus Wirth. Posted on
https://people.inf.ethz.ch/wirth/Oberon/index.html

Oberon compiler internals
"Compiler Construction". Niklaus Wirth. Posted in two parts (chapters 1-8, 9-16) on https://people.inf.ethz.ch/wirth/CompilerConstruction/index.html

Oberon philosophy
"A Plea for Lean Software". Niklaus Wirth. Pages 64-68 in the February 1995 issue of "Computer" (volume 28, number 2).

Oberon article
"Oberon – The Overlooked Jewel". Michael Franz. Pages 41-54 in "The School of Niklaus Wirth: The Art of Simplicity". Laszlo Boszormenyi, Jurg Gutknecht, Gustav Pomberger (editors). Published in 2000 by Morgan Kaufmann Publishers.

SYSTEM DIFFERENCES

The original Oberon system includes many unusual user interface elements. These elements make the system worthy of study, both as a historical software artifact, and as a case study in user interface design.

But the original system also includes a few non-key features (such as *mouse interclicking* and opaque command names) which make the system difficult for beginners to use.

The primary goal and purpose of Oberon Pi is instructional: to make the original Oberon system more accessible to beginners, by updating the non-key user interface elements to contemporary software standards, while preserving the key elements that make Oberon unique.

The automotive world uses the term *restomod* to describe vintage cars that have been restored in the traditional sense, but with modern engines, brakes, and electronics.

The Oberon Pi user interface should be considered a restomod of the original Oberon system.

Oberon Pi includes the following differences from the original Oberon system:

- No mouse interclicking
- Updated terminology
- Text-editing commands
- System dates
- System directory
- System startup windows
- Draw application

No mouse interclicking

The most significant change in Oberon Pi is its elimination of Oberon's use of *interclicking*, a feature that requires users to press various combinations of mouse buttons at the same time in order to perform certain mouse commands.

By contrast, mouse commands in Oberon Pi are constrained to pressing only one mouse button at a time on the Raspberry Pi's three-button mouse.

Oberon Pi additionally defines a second set of mouse commands, which use the keyboard Shift key to extend the basic Oberon Pi mouse commands. This approach not only utilizes a freely-available user resource (i.e., the user's other hand), but also enables the Shift-mouse commands to be defined as natural mappings of the basic mouse commands, following the traditional semantics of the Shift key.

The end result of this change is a set of mouse commands which is arguably easier to learn and use than that of the original Oberon system.

Updated terminology

The other significant change in Oberon Pi is its renaming of many of Oberon's nonstandard terms for windows, commands, and other items, with either:

a) standard terms used in contemporary computing, or
b) more consistent and descriptive names in general.

viewer -> window
key -> button (mouse)
caret -> cursor

System.Log -> Terminal
System.Tool -> Project

System.Close -> Window.Close
System.Grow -> Window.Open
System.Copy -> Window.Split
System.Clear -> Window.Clear
System.Recall -> Window.Reopen

Edit.Store -> Edit.Save
Edit.Recall -> Edit.Undo

System.ShowModules -> System.ModulesLoaded
System.Free -> System.UnloadModules

System.ShowFonts -> System.FontsLoaded
System.FreeFonts -> System.UnloadFonts

System.ShowCommands -> System.ModuleCommands

System.Watch -> System.Storage
System.Collect -> System.FreeMemory

ORP.Compile -> OBC.Compile (compiler)
ORG -> OBG (code generater)
ORS -> OBS (scanner)
ORB -> OBT (symbol table)

ORTool -> OBD (file decoder)
ORL -> OBL (system build tools)

Draw.Ticks -> Draw.Grid
Draw.Store -> Draw.Save

Text-editing commands

Oberon Pi replaces Oberon's interclicked text-editing commands with the standard text-editing commands, and defines the standard keyboard shortcuts for those commands:

Command	Shortcut
Edit.Cut	Ctrl-X
Edit.Copy	Ctrl-C
Edit.Paste	Ctrl-V
Edit.Undo	Ctrl-Z

—

As a modest productivity enhancement, Oberon Pi adds one new selection command, and changes the behavior of two existing commands.

- Shift-right-click is added as a standard command for extending an existing text selection.

- Right-clicking twice on a word now selects words, file names, command names, and procedure names.

- When you drag the text cursor left or right on a text line, the cursor sticks to the line as you drag it.

System dates

The System.Date command obtains date values from the Raspberry Pi operating system. In Oberon Pi these date values are read-only.

Oberon Pi uses the same date values to timestamp files created in the Oberon file system.

System directory

The System.Directory command treats a missing command parameter as equivalent to “*".

System startup windows

When the original Oberon system first starts up, it automatically opens any file named “System.Tool”, and displays it in a *startup* window in the system track of the Oberon display.

Oberon Pi redefines and extends this feature to support the use of multiple startup windows:

- A text file named "Project" (Oberon Pi's version of "System.Tool") gets automatically opened in the system track, below the terminal window.

- A text file named "Commands" gets automatically opened in the system track, below any project window.

- A text file named "Chapters" gets automatically opened in the system track, below any command window.

- A text file named "Guide" gets automatically opened in the user track.

Draw application

In the Oberon Pi Draw application, the selection areas for all graphic elements have been enlarged to make them easier to select with the Raspberry Pi mouse.

LANGUAGE DIFFERENCES

The Oberon language has two major versions: the original version released in 1987; and Oberon-07, released in 2007 and revised several times after, with the last revision in 2016.

Oberon Pi conforms to the 2016 language definition, which is provided as a language reference document in the Oberon Pi release.

The language differences are described in the document "*Differences between Revised Oberon and Oberon*", which is also provided as a release document.

This section presents additional language differences which are not described in the language differences document:

- CASE statements
- Nested procedures
- Type LONGINT

CASE statements

The original version of the Oberon language defined the CASE statement to include an optional ELSE statement, which handled case index values that did not match any of the case labels.

In this original version, if a case index value did not match any of the case labels, and no ELSE was defined, the CASE statement aborted the program.

But in the 2016 version of the Oberon language, the CASE statement no longer supports an optional ELSE statement, and if a case index value does not match any of the case labels, the CASE statement does nothing, and program flow continues in the following statement.

In addition to this change, the CASE statement has the following restrictions:

- Type case index variables cannot be structured variables (s[x], s(x), s.x).

- Integer case labels must be in the range 0-255.

For example “Edit.Open CaseDemo.Mod”.

Nested procedures

The Oberon language allows a procedure to be declared inside another procedure. This is called a *nested* procedure.

In the original version of the Oberon language, a nested procedure could access constants, types, or variables that were declared in the procedures containing the nested procedure.

But in the 2016 version of the language, a nested procedure can access these objects only when they are declared *locally* in the nested procedure, or *globally* in the module containing the procedure. The so-called *intermediate* objects declared in the surrounding procedures are not accessible in a nested procedure.

For example “Edit.Open NestDemo.Mod”.

Type LONGINT

The original version of the Oberon language supported the separate data types INTEGER and LONGINT. The current version supports only the data type INTEGER.

The Oberon Pi compiler still supports LONGINT, and treats it as equivalent to INTEGER.

For example "Edit.Open TextFrames.Mod".

SYSTEM LIMITATIONS

When judged by contemporary software standards, the Oberon system has a number of limitations.

These reflect both the state of computer hardware when Oberon was created, and the designers' well-documented intention to create a system tailored to their own needs, and not to those of the marketplace.

- The window system supports neither auto-scrolling nor the selection of content that exceeds the current window view. This limitation makes the Window.Split command necessary for creating large selections.

- The system undo commands (Edit.Undo, Window.Reopen) support only minimal undo operations, with a single level of undo.

- The text editor ("Edit") supports basic text editing, but is not suitable for general document preparation. Other Oberon systems have included dedicated applications for word processing.

- The scroll wheel on the Raspberry Pi mouse is limited to serving as the middle mouse button. The Oberon system was not designed to support scroll wheel input.

- Due to limitations in the RISC emulation software, the arrow keys and caps lock key are not recognized by the Oberon Pi system. (The red caps lock indicator will light up on the keyboard, however.)

- Though the Oberon system includes facilities for displaying color, the RISC emulation software supports only a monochrome display.

- The system screen fonts are bitmapped, and low-quality by contemporary software standards. Note how this document avoids using the available bold and italic font styles. For examples see "FONTS".

- The Oberon Pi system does not support the display of image files (JPG, PNG, etc.). Other Oberon systems have supported image file types.

- The Oberon system does not provide access to the internet or World Wide Web. When Oberon was first created, these features did not exist.

- This version of the Oberon Pi system lacks the ability to transfer files to and from the Raspberry Pi operating system. Currently it can only import and export text via the Pi operating system clipboard, using the Clipboard module.

- By contemporary standards, the Oberon file system is arguably too minimal to support large-scale computing projects. Other Oberon systems have included modules which provide access to networked file servers with greater storage capacity.

CREDITS

The Oberon system was developed by Niklaus Wirth (NW) and Jurg Gutknecht (JG).

The Oberon compiler was developed by Niklaus Wirth.

The Oberon RISC emulator and Clipboard module were developed by Peter De Wachter.

The Oberon compiler changes and system building tools were developed by Andreas Pirklbauer (AP).

The modules In, Out, and XYplane were developed by Martin Reiser.

The Oberon Pi documents and software changes were developed by Richard Gleaves (RG).

===