Porting Native Oberon to the Gneiss Microkernel – A Guideline for Future Ports

Jacques Eloff and Frank van Riet
eloff@cs.sun.ac.za
vanriet@cs.sun.ac.za

Department of Computer Science
University of Stellenbosch

11 February 1999

Abstract

The object of this report is to give a brief overview of how Native Oberon System 3, Release 2.3.0 was ported to the Gneiss Microkernel. It serves two main purposes. First, to describe the process of porting the system to the Gneiss kernel and the methods that were applied during this process to accomplish our goals. Second, to serve as a guideline for when future ports are attempted in order to minimize the type and number of errors that may occur during such an exercise.

Contents

List of Figures

List of Tables

1 Introduction

The report is divided into several sections. While some focus directly on the project of porting Oberon, others play a more informative part like the section on garbage collection. Section 1.2 gives a brief overview of what the project entailed. Section 2 focus on how the port was accomplished, where and how it was started and how the different modules finally came together to form a single new system. General topics such as debugging, cross platform development and the system configuration that was used during the current port will be discussed in Section 3 .

Section 4 plays only an informative part. This section should serve as a good introductory text before delving into the source code of the garbage collector and its related routines. The authors feel it is necessary to include this section as the garbage collector was one of the most complex sections of code encountered during the current port, and, without a basic understanding of how memory management functions inside Oberon, making modifications or locating errors in this section can become time consuming and difficult.

Section 5 gives a detailed description of each individual module that was involved during the current port. Although this section strictly applies to the current port, it should assist the reader in obtaining the necessary knowledge to understand the Oberon source code, especially how the different modules are integrated, and hopefully reduce the time required to perform future ports.

1.1 Conventions Used in this Report

- The term **current port** refers to the porting process of Native Oberon System 3, Release 2.3.0 to the Gneiss environment.
- The term **previous port** refers to the system that was originally ported from DOS Oberon in 1992.
- The term **kernel** is used interchangeably throughout the report and will be qualified in the context that it is used in as it can refer to both the Gneiss microkernel and the Oberon `Kernel` module.

1.2 Overview of the Current Port

The project was started to accomplish a number of goals. First, to make the new Native Oberon environment available to replace the the previous port. Second, to include the Gadgets Graphical User Interface (GUI) as an extension of the traditional viewer system thereby making it more accessible and easier to grasp for new users, especially first year students who use Oberon primarily as a teaching tool. Third, to possibly replace the `IOSvr` in the kernel making Oberon the primary display through which all information is channelled including other VM's. Last, to integrate the new system with the *Caching File Server* which replaced the *HFS* system. Due to design constraints encountered, it was decided to keep the `IOSvr` for the time being.

The `IOSvr` is considered to be lacking in performance being one of the oldest servers in the Gneiss kernel. The removal or replacement of the server would naturally hold many advantages, but would also introduce certain design problems.

The greatest disadvantage currently is that the `IOSvr` is slow and does not offer support for the latest available Super VGA (SVGA) chip sets found in most IBM Compatible PC systems today. The `IOSvr` only supports standard VGA and one SVGA chip set namely the Tseng ET4000. This being the case, removing the `IOSvr` would allow one to use the display drivers that Native Oberon provides, thus supporting a far wider variety of display hardware.

The greatest problem the authors foresaw was that the `IOSvr` supports both a text based and a graphical oriented environment. The text environment is where VM's, except Oberon, normally display information by making use of the `WindowSvr`. Once the `IOSvr` is removed, Oberon would become the only active display mechanism on the system. Other VM's would no longer be able to display themselves unless it was done inside the Oberon environment.

VM's in the text based environment allow for a certain amount of interaction as users can quickly switch from one VM to the next. In Oberon, the main scheduling loop prohibits this switching to a certain extend in that only one task can execute at a certain time, even though there may be many tasks installed in the central loop.

By removing the `IOSvr`, these two paradigms would have to be integrated into a single environment. First, the viewer system would have to be extended to allow VM's to make requests to display themselves inside the Oberon environment. In order to service such requests, the Oberon module would have to take on both the role of client and server where it traditionally only played the role of a client of the Gneiss kernel. The biggest challenge was that even if this new viewer mechanism could be introduced, VM's would no longer have the same amount of freedom with regards to user interaction.

Second, Oberon would need extensive modifications in order to distinguish between viewers owned by itself and those belonging to other VM's. One example here is that VM's might sometimes need to display critical data that requires a user's attention immediately. If Oberon was busy attending to another task, the VM's request would simply be blocked until it could be serviced. The result of this could be quite fatal, especially if the information is lost and could not be retrieved again.

Given the current status of the `IOSvr`, the authors' objectives and limited time frame of two months, it was decided that the `IOSvr` would be kept during the current port. Although the standard VGA display driver that Native Oberon provides was used, it required a certain amount of modification in order to work with the `IOSvr` and was only made possible by the fact that the `IOSvr` supported the hardware.

1.3 System Specifications

This section gives a brief overview of the different systems that were employed during the current port:

- **Source:** Native Oberon, System 3, Release 2.3.0
- **Target:** Gneiss microkernel 0.74f with *Caching Filer Server* (CFS)
- **Development:** Native Oberon, System 3, Release 2.3.0
- **Testing:** Gneiss microkernel 0.73y, stand-alone DOS based version

Section 3.2 gives a more detailed explanation on how the systems were employed to interact with one another and facilitated in the testing of the software.

1.4 Acknowledgements

We would like to extend our thanks and gratitude to the following people. They were always eager to assist, whether it was simply a meaningful suggestion, the time they took to fit us into their daily schedules or the replies to all the e-mail. It is much appreciated.

- Pieter de Villiers
- Eben Esterhuyse (Assistance during the integration of the current port with *CFS* (Caching File Server))
- Alan Webber
- Jaco Loubser
- Pieter Muller (ETH, for all the information provided regarding the garbage collector, especially for the information contained in Sections 4.3.3, 4.3.4, 4.3.5 and 4.3.6, as well for making all the Oberon source code available to us. Also involved with the previous port)
- Johan de Villiers (Data-Fusion, assistance with debugging exception handlers. Also involved with the previous port)
- De Villiers De Wet (Data-Fusion, assistance with the garbage collector and debugging suggestions. Also involved with the previous port)
- Régis Crelier and Thomas Burri (Régis Crelier drew the original heap diagrams. It was redrawn by Thomas Burri and reproduced for this report with the permission of Pieter Muller)

2 Porting Oberon

2.1 Beginning the Port

Before writing a single line of code, the authors concentrated on understanding each individual module and its part in the Oberon hierarchy. Understanding the Gneiss kernel is also very important as it partially replace some of the functions which the `Kernel` module fulfills under Native. Because the `Kernel` module forms the basis of the system, it follows that this was the first Native module that was ported.

2.1.1 The Basic Test Model

Initially the Native kernel was stripped of all functionality. Apart from the initialization code, it contained only a single implemented procedure namely `Exit` that was taken from the Oberon kernel from the previous port. From this a test VM was constructed. Stubs were implemented for the memory routines in order to allow it to link correctly. The memory routines were not considered important at this stage as no dynamic structures were implemented. Following is the code for the first VM that we used for testing the new `Kernel` module under Gneiss:

```
MODULE Test;

IMPORT
  Kernel;

BEGIN
  Kernel.Exit(0)
END Test.
```

The simplistic design was motivated by the fact that the VM facilitated in providing the means for testing the new `Kernel` module and therefore need not be a complex program itself. In order for the test VM to execute, the `Kernel` module had to be initialized correctly.

The test VM was extended as more functionality was introduced to the `Kernel` module. During these extensions it became clear that tracking differences in the implementations between the `Kernel` module of Native and that of the previous port was very important. Although locating changes that are Gneiss specific usually proves to be a fairly easy task, care must still be taken. Sometimes code might be encountered that is Gneiss specific, but which is no longer required. The same is true for Native code. Sometimes certain procedures and functions will need to be replaced with a Gneiss equivalent. Quite often sections were encountered in the `Kernel` module whose services were performed by servers inside the Gneiss kernel. Consequently, these parts of the Native `Kernel` were removed or modified to interface with Gneiss.

2.1.2 Testing Memory Management

Testing the memory management routines is very complex. Although there exist a number of simple tests that can be conducted, they only serve a general purpose and result in giving a global view of memory management.

The initial tests focused on those routines that monitored memory usage. These routines are: `LargestAvailable`, `Available` and `Used`. Once they proved to return accurate information the actual allocation and deallocation of memory was tested.

The authors again relayed on a simple test. A elementary dynamic structure like a binary tree was constructed after which it was traversed, deallocated and an explicit call was made to the garbage collector. Memory usage was reported before, during and after the construction and traversal of the structure.

During the early stages when work was still performed on the Oberon core, a number of unrecoverable errors occurred. During the booting process of Oberon, the garbage collector was activated and collected all the active file handles before the module loader had completed loading all the required modules. After every possible cause was explored, it was suspected that the error originated during garbage collection. The suspicion was verified when checks were implemented on the root pointer of the file handles. This pointer spontaneously turned `NIL` during the booting process, shortly after the garbage collector was called for the first time.

A more specialized test [1] was applied and immediately failed, thus strengthening the authors' suspicion that the garbage collector did collect blocks of memory that still contained active references, in this case, the root of all the file handles. The error was introduced by

[1] Refer to Appendix A.1 for the source code used in this test

a section of code that was omitted during the port and which was responsible for the initial setup of the heap.

As the system's complexity increased, brought on both by the number of modules that were loaded and by the fact that Gadgets and the compiler was actively used, it became harder to locate memory errors. As the system was tested over a longer period of time, it came to the authors' attention that memory fragmentation increased, especially when compiling large groups of modules like Gadgets. The situation reached a point where the largest block of contiguous memory accounted for roughly 2.5% of the total available memory, which is not considered to be an acceptable level. This clearly indicated that there was an error in the memory management system.

Although garbage collection does not completely remove fragmentation, it does limit it to a far better extent than would have otherwise been possible had memory management been left under the control of the programmer to allocate and deallocate memory as he or she saw fit.

After a detailed investigation it was determined that the implementation of the filesystem was responsible for the fragmentation. During a compile session, a number of new file handles are allocated as many temporary files are created apart from the source, symbol and object files. Since file handles are fairly large, memory is allocated from the list containing the largest free blocks. Refer to Section 4.3.2 for a more detailed explanation on basic memory allocation. Compiling a large system like Gadgets speeds the fragmentation process. Since a termination handler was used in the `Files` module, the handles were only marked for collection during an exit from Oberon. This meant that the fragmented memory could never be restored as those file handles used for the creation of the temporary files were never collected.

2.1.3 Exception Handlers

The way Oberon handles exceptions under Gneiss differs significantly from how it is implemented under Native Oberon. During initialization the `InitTrapHandling` procedure creates a local thread in the Oberon VM called `Dispatcher`. This thread effectively loops and and blocks until an exception occurs. The exception is detected by repeatedly issuing calls to the `AwaitExceptions` function. This function in turn does an IPC transaction with the exception server inside the Gneiss kernel to check if an exceptions did occur.

Once the exception is detected, the machine state is saved and the `ipTable` is checked to see if a trap handler has been installed for the exception that occurred. If so, control is transfered to the handler by simulating an interrupt and calling the `SetState` function. The `SetState` function is responsible for the actual control transfer to the handler. If the `ipTable` returns a NIL pointer for the specified exception, the `DefaultHandler` procedure is called.

The OP2 Compiler no longer supports the `+` operator to compile interrupt handlers as was the case in the previous port. The `Scheduler` procedure therefore no longer generates a stack to simulate an interrupt, but rather one to simulate a procedure call. `Transfer` and `Stackbase` can no longer be in-lined using the `-` operator and have been transformed into formal procedures. The use of normal procedure calls also mean that the stack pointer register, `ESP`, needs to be adjusted as no `RET` instruction will be executed to pop the registers from the stack that was pushed during the procedure call. Figure 1 gives an outline of how the stack will be configured before the handler for the exception is executed.

Considering the changes that were made in the `Kernel` module to the exception handling mechanism, it follows that the `System` module also needed to be modified. `System` is respon-

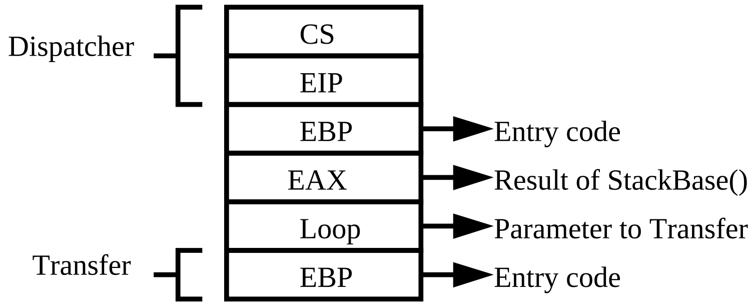

Figure 1: Stack Layout for an Exception

sible for opening a trap viewer whenever an exception occurs to inform the user of the error. Once this is done, `System` performs a traceback to locate the origin of the exception and continues this traceback from the module in which it occurred until it reaches the scheduling loop in the `Oberon` module. The following modifications were required by the `System` module due to changes made in the `Kernel` module:

- The `EFlags` and `EBP` registers are no longer read from the stack. Instead, their values are obtained from the machine state exported from the `Kernel` module in the `state` variable.
- The *detailed* option is no longer supported. As was done in the previous port, a complete register dump is written to the `Messages` window.
- An explicit `EXIT` has been inserted in the traceback loop and is called once the traceback reaches the `Oberon.Loop` procedure. Due to the changes in the stack layout, the procedure can no longer terminate as it did under Native.

Although the testing of the exception handler proved simple, correcting and debugging it posed more of a challenge. The first test involved an empty VM which simply tried to reference a `NIL` pointer. This of course leads to a page fault being generated and invoked the exception handler. The test was applied to a variety of exceptions, including issuing breakpoints (INT 3), invalid machine instructions and division by zero. What is also very important during this sort of testing is the verification of the machine state. Each VM executes in a separate address space, usually starting at `20000000H`. Once an exception is generated, `EIP` (the instruction pointer) must be verified to make sure that it does not contain a value lying outside the VM's address space. If so, care must be taken to determine the reason for this and if necessary, the required steps to correct it must be implemented and the routines must be tested again to verify the corrections.

Once the basic exception handling worked correctly, a dummy loop was installed in the test VM in order to simulate the main scheduling loop in Oberon. This was done in order to test if the handler, either an installed one or the default one, could in fact transfer control back to the main scheduling loop once the exception has been generated and handled.

Testing was completed when the basic Oberon environment was ported. Given that an exception occurred, a register dump is given in the `Messages` window and a trap viewer is opened inside Oberon with a stack trace back, indicating where the exception originated. The trace back must be carefully verified and all values, especially the `PC` [2], must be checked.

2.1.4 Completion of the Oberon Core

The Oberon core was completed by compiling and testing the necessary modules required by a very basic Oberon system. This includes those modules responsible for display management, the basic viewer system and editor as well as the module loader and scheduler in the `Oberon` module.

Although the core is fairly small, locating errors during this phase can prove difficult as there are more opportunities for errors to find their way into the code. An error in the input driver might only be detected at a later stage for example or errors might occur that could point to any number of modules that might have been the origin of the error.

Extensive use was made of the `Trace` module to obtain as much information as was possible. The module loader was extensively monitored, as were the routines for accessing the display and input drivers as well as the file system related routines.

2.2 Porting Gadgets

The Gadgets environment proved to be fairly easy to port. The only requirement being that it be recompiled on the ported system once the basic environment was considered stable.

Certain modifications were required, but proved to be few and fairly concentrated within a small group of modules.

Not all applications were considered during the current port though. Modules offering Telnet and FTP services were left out as Gneiss already provides these services to users. Also, all the network modules required to interface with the gadgets networking environment were not port as it will require a significant amount of modification to work correctly under Gneiss. A further reason for not porting network related Gadgets modules is that our objective was to port Gadgets and not Gadget based applications.

Those modules not requiring modifications were ported as it imposed no further constraints on the already limited time frame. It must also be remembered that many of the Gadgets based modules are projects that were given to students at ETH and that the source code is not distributed with the rest of the system, nor is it covered by the license agreement.

The authors do however foresee that once the network environment of Gadgets has been adapted to work under Gneiss and interface with the `NetSvr` and other networking software that these applications could easily be ported. The result of such a port would mean that web browsing utilities and email facilities would become available.

2.3 Porting Gneiss Related Modules

Although not part of the original objectives, it was decided to port as many of the older, Gneiss specific modules to the new Native environment under Gneiss. Only a limited amount of modules were ported due to time constraints. In order to lend some form of priority to the process, those modules most likely to be used were identified as suitable candidates and include:

- Patch

[2] Program Counter

- PFinder
- Env
- VM
- Print
- Andy

2.4 Integration with CFS

The final phase of the port was concluded by moving the current port from a single user environment to a distributed, multi-user environment and integrating it with the *Caching File Server* (CFS) which replaced the HFS system during the last half of 1998.

The CFS related modules required a small amount of modifications. Certain constructs that were inherited from HFS seems to have been allowed under the previous port due to possible errors in the compiler with regards to casting extended record types. The new OP2 compiler required that some modifications be made to CFS in order to allow the code to function correctly. Most of the modifications related to dynamic extended record types that were cast into other types and passed as dereferenced parameters during IPC calls.

The advantage in performance offered by CFS [2] over HFS proved to be more than adequate given the increase in the amount of file services required to boot Native under Gneiss. Table 1 gives a brief outline on the increase in system requirements for Native Oberon, especially where memory is concerned.

Version	Modules	Memory (K)	Filesystem Calls
Current Port, 2.3.0[a]	28	255	339
Current Port, 2.3.0	59	808	985
Current Port, 2.3.0[b]	68	1130	1334
Previous Port, 1.6	31	245	338
Previous Port, 1.6[c]	37	380	491

[a]Excluding Gadgets and OP2 Compiler
[b]Including the OP2 Compiler
[c]Including the Compiler

Table 1: System Resource Usage During Booting

Because of the way that CFS handles the storage of files, `Edit` and certain Gadgets related modules required modification to their storage routines. In order to create a backup file, the original copy must first be renamed before a new file is created and stored. This is required because of the way the cache is organized. Not doing so will lead to the destruction of the original file with the data contained in the backup file. CFS was also extended to give a detailed directory of files including their date and time stamps as well as the actual file size.

Testing Native Oberon under CFS introduced certain problems because of the organization and structure of the file system. When ever a user requests a file, CFS first try to locate a local copy in the users account. If unsuccessful, the group account is queried of which the user is a member. If the file could still not be located, the filesystem tries to locate the file from a special group called `ZeroGroup`. Only if the last instance fail can it be assumed that the file is not present on the server. Since `ZeroGroup` is still based on the previous port, as are many of the other groups on the system, it often happened that CFS retrieved files from `ZeroGroup` when it could not be located in the `Native` group account. This of course lead to version mismatches and many modules could not be tested until it could be ensured that all the required files were in fact inside the `Native` group.

One solution that was investigated was the creation of multiple `ZeroGroups`, but due to the filesystem, this was not possible. Once Native becomes stable, the remaining files from the previous port should be replaced with those of the current port to ensure consistency on the file server

2.5 Continuing Gneiss Kernel Development

No investigation was done as to what the influence of the current port would be on the development of the Gneiss kernel. It is however quite clear that all modules relaying on the use of assembly language would need modifications to conform with the new syntax imposed by the OP2 compiler. The authors do feel however that the development of the Gneiss kernel be continued under the previous port for the time being until the current port has proven itself as a suitable environment, but a steady and progressive transition to the new environment should be considered.

3 Porting: A General Approach

3.1 Introduction

This section concentrates on the general aspects of porting and the topics are discussed in the context of the current port, including the application and impact that these topics had on the porting process.

3.2 Cross Platform Development

Cross platform development is a topic which needs to be approached with caution. Unlike development under a single system, working across more than one platform creates ample opportunities for introducing errors, especially where the mismatch of versions are concerned. The authors often found that once an error was located it proved quite difficult to attribute the error to a specific system or simply view it as a case of pure incompatibility between two versions.

The current port required that a group of very distinct environments had to be used, either in conjunction with one another or separately. These environments include the following:

- The Native Oberon system, Release 2.3.0
- The stand-alone version of Oberon under Gneiss as implemented in the previous port. For this, version 0.73y and 0.74f of the Gneiss kernel was used.
- The distributed version of Oberon under Gneiss, including the new *Caching File Server* (CFS) as implemented in the previous port.

- The distributed version of Oberon under Gneiss, including the new *Caching File Server* (CFS) as implemented in the current port.

Although Native Oberon was the primary source of the port, the previous implementation was constantly used as a reference. First, to gain an understanding of how the system functioned in the past. As there were none of the DOS Oberon sources in existence, the authors quite often had to make assumptions to determine which code was possibly modified during the previous port or which was reproduced without modifications. It was quite common to encounter code from the previous port which either did not contain accompanying comments or which were simply fragments of ideas never implemented. Second, to serve as an indication of which procedures and functions had to be modified in order to interface with Gneiss instead of directly accessing the hardware as is the case with Native Oberon.

All the development was done under Native Oberon. Once the software needed to be tested it was transferred to disk and loaded unto a separate machine which contained the stand-alone version of Gneiss and Oberon from the previous port. In order to speed the process, customized files were created, similar to *makefiles* one would typically use with C under Unix. These files contained the necessary commands for copying, renaming and compiling all the modules. As the development process matured, all the software was compiled on the development machine and transfered in whole to Gneiss.

During the early phases of development this proved to be an effective method. As the amount of code that required testing increased, the turn around time [3] increased dramatically, sometimes to as much as forty minutes compared to an average of approximately five to ten minutes during the early stages of the porting process. The basic system that was compiled was modified to include all the required modules up to the compression utilities. Once the basic system was transferred, the remainder of the system was brought over in Oberon's Arc compression format and uncompressed on the target machine. Although this method had a significant impact on reducing the turn around time, it could only be applied effectively during a complete system transfer.

Once a stable stand-alone system was created the target platform was changed to a machine running the distributed version of Gneiss under CFS. This machine was used to test the interface to CFS. In order to create a safe environment, the development process was continued using a single account on CFS. Once this proved successful, all the system files[4], including the documentation, examples and applications were transferred to the *Native* group account. A separate group called *NativeSrc* containing the source code was also created in order to guard the code from prying users.

3.3 Debugging Techniques

Although Gneiss offers a full set of debugging tools, the authors refrained from using them since the object file format was inconsistent between the previous and current port, thus rendering the debugger incapable of giving accurate information. A more manual form of debugging was employed, utilizing every part of the system that could yield useful information in order to clarify, describe and explain program behaviour.

[3]Turn around time refers to the amount of time it takes to create a new image, reboot the test machine, upload the new image and run all the required tests before returning to the development machine

[4]Object and Symbol files

Some may consider the applied methods archaic, but the type of systems programming that was involved required a more direct and hands-on approach to debugging. In the authors' experience it proved quite successful in locating errors efficiently. The most time consuming part of the process was correcting the errors which were often very subtle.

3.3.1 The Trace Module

The `Trace` module is capable of displaying arbitrary text information in the `Messages` window. This module was often used as a method for localizing errors by placing trace code at both the entry and exit points of a procedure. The following extract of code will be used to explain this concept.

```
PROCEDURE MyProc1;
BEGIN
  Trace.String("Entering MyProc1"); Trace.Ln;
  ...
  (* Remainder of procedure body *)
  ...
  Trace.String("Leaving MyProc1"); Trace.Ln;
END MyProc1;
```

If the entry point trace code was written in the `Messages` window, but not the trace code contained at the exit point, it served as an indication that the procedure failed during it's execution. Any other procedure that might be called from inside a specific procedure also contained entry and exit trace code. Should a procedure call any other procedure one is able to localize possible errors by locating the innermost procedure with entry code, but no exit code. Apart from this, it also provided a concise overview of program flow and module interaction.

The `Trace` module was also employed to assist with more general tasks. Whenever a procedure dealt with complex calculations, the `Trace` module was used to display this information in order to verify it with the authors' precalculated-calculated values.

Too many trace code can also hinder one's debugging efforts. Once a section of code has been tested and seems to work, either remove all the related trace code or keep a short message. Too much information at once does not assist, but rather overwhelms and confuses.

Also, note that trace code written to the `Messages` window will cause the VM to grow from time to time, usually in multiples of 8K. One might easily view this change in available memory as a leak of some sort, while in effect it is quite normal. The `Messages` VM was written in Modula-2 code and designed to grow dynamically. As more trace code is written to this window, it eventually needs to allocate additional memory in order to buffer the text and consequently the VM size increases usually in blocks of 8K.

3.3.2 Serial Terminals

A serial terminal was employed as debugging tool when the display driver was ported. Certain design problems during the early stages of this phase prohibited the authors from using the `Messages` window as the primary debugging tool since there was no active display when a system crash occurred during a switch from text to graphics mode.

The serial terminal was also used to gain information on the behaviour of the garbage collector and other functions under the Native kernel that use the set of serial trace routines in the `Kernel` module to output debug information.

Since serial terminals do not provide a scroll back mechanism, care must be taken in selecting the type of information that need to be displayed. It is best to use the `Trace` module for information with a high repetition rate and reserve the serial terminal for very specific information.

3.3.3 Debugging the Display Driver

Native Oberon supplies a special display driver called `Trace.Display.Mod` which can be used during debugging. Since all the modules in the Oberon core depend on a display driver being present, testing and debugging can become a complex issue. The `Trace.Display` module does not contain a single line of code to deal with graphics hardware. Instead, the procedure bodies were replaced with trace code to output information about the specific procedure.

The advantage is that one can easily compile and test the remainder of the Oberon core without introducing an unstable device driver. Considering how the `IOSvr` functions, an unstable display driver could mean that there is no means for switching back to the `Messages` window to view important debug information or determine where a possible system crash occurred.

3.3.4 The Decoder Module

The `Decoder` module provides another means for performing low level debugging as it is capable of giving a complete assembly dump of a module including the actual hexadecimal formats of machine instructions. The `Decoder` was used in order to verify certain modules that contained assembler code. It also served in gaining a solid understanding of the type of code that is generated for procedures and functions, especially with regards to entry and exit code. This knowledge was extensively used to debug the exception handling mechanism of the `Kernel` module as a detailed description of the stack layout was required. The source text of a test module along with the `Decoder` output is given in Appendix A.2 as an example.

3.4 The OP2 Assembler

The OP2 compiler offers a far more suitable environment for using assembly language than was possible under the previous port by way of the `DInline.Assemble` construct. Since procedures and functions can no longer contain mixed sections of both Oberon and assembler code, older code conforming to the mixed layout requires the separation of the assembler and Oberon code.

Another feature is that the type of machine instructions required must be specified. If for example i486 privileged instructions need to be accessed, the procedure must specify this in terms of predefined sets like:

```
PROCEDURE MyProc;
CODE {SYSTEM.i486,SYSTEM.Privileged}
  ...
END MyProc;
```

Those instructions not supported currently by the OP2 compiler can be implemented by using the machine language values and declaring the instruction as a sequence of bytes. The technical documentation [3] at ETH also contain examples for using the new assembler. It must be remembered that under Gneiss, the restrictions for using the assembler are much stronger imposed due to the layout of the microkernel and the use of protected address spaces. Unlike Native Oberon, the `Kernel` module executes as a user process in privilege level 3 under Gneiss, whereas under Native Oberon the `Kernel` module executed at level 0 with other modules residing in level 3.

Native based modules seldom require changes to the actual assembler code unless modifications are required to interface correctly with Gneiss. It is more likely that changes would need to be made to older Gneiss specific modules that are brought into the new environment and requires to be recompiled with the OP2 compiler.

3.5 The BootLinker

The `BootLinker` module can also be used for debugging purposes, but only during the development of the statically linked image of the new Oberon VM. Any exception that may occur inside the address space of the static image can be easily located using the `BootLinker.Find` command in conjunction with the compiler. The linker also creates a log file which contains data regarding the static image that was linked, and although this file is generated for the usages of the linker only, it might help to give insight into the way modules are linked together. The source code of the `BootLinker` might also be helpful as the code is well commented, but an extensive knowledge of the object code format is a prerequisite.

4 Garbage Collection and Memory Management

4.1 Introduction

This section gives an overview of how memory allocation strategies work inside Oberon as well as the process of garbage collection and the related routines involved with this process. Most of the information in this section was obtained from Pieter Muller at ETH. Though the changes that were made to the Native kernel's memory related routines during the current port to perform the necessary functions under Gneiss were subtle, the authors feel that a detailed discussion is justifiable given the impact that these routines have on the Oberon environment. Without a basic understanding, locating errors or attempting to correct them in this part of the `Kernel` module might prove futile and extremely time consuming.

Native Oberon uses a group of routines for allocating memory. The programmer is usually just aware of making a call to the `NEW` procedure, or in rare cases to the `SYSTEM.NEW` procedure. As will become clear in this section, what actually happens during such a procedure call is much more complex and most of the details are hidden away inside the `Kernel` module.

4.2 The Initialization Process

During the initialization of the `Kernel` module there are two very important procedures named `Init` and `InitKernel`. The `Init` procedure is responsible for setting up the stack and heap and replaces the `InitHeap` procedure of Native Oberon. If no stack environment variable is

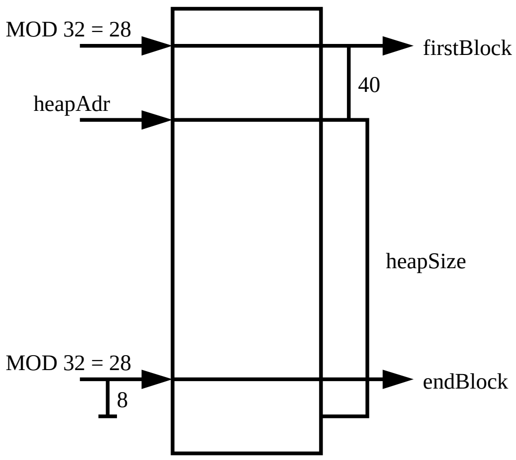

Figure 2: Initialization of the Heap

supplied the size will default to 60K. If one is specified it is checked against a lower boundary of 32K. The final value is then reduced by 4K and added to the current stack size of the VM.

The same method is applied when setting up the heap. If the heap environment variable is present, its value is used. If the variable is absent or contains a negative value, the size defaults to 256K, the default VM size. After this the total free memory is calculated and the requested size is tested against this as well. Also take note that 128K of the available memory under Gneiss is reserved. Should this part of the initialization process fail, a `CoreHalt` will be issued. The last part of `InitKernel` is responsible for initializing the `StackOrg` and `heapSize` variables.

The `InitKernel` procedure is responsible for initializing the `firstBlock` and `endBlock` variables, aligning each on a 32-byte boundary and then adjusting the address by -4 to align it on an 8 byte boundary. `firstBlock` and `endBlock` represent the lower and upper boundaries of the heap respectively. After this, the memory between these two pointers are initialized to `0`. Once this is done, the `firstBlock` variable is cast into a `FreeBlock`, initialized and a single `FreeBlock` structure is created spanning the whole of the heap. Figure 2 gives a basic layout of this process.

Unlike Native which calls the garbage collector explicitly in the `InitHeap` procedure, `InitKernel` makes a call to the `Sweep` procedure to initialize the free lists. This is done because the garbage collector under Gneiss uses a dynamic structure called `info` to perform the stack tracing of threads. Since no memory effectively exist at this stage, `Sweep` is called instead to create the free list, after which the `info` structure is allocated.

Under Native Oberon the initialization of the `firstBlock` and `endBlock` variables are much more complex as factors like low memory [5] and DMA are taken into account.

Two special type descriptors are also located right at the beginning of `InitKernel`, namely `ptrElemTag` and `dynarrElemTag` which is used during the allocation of dynamic arrays in the `NewArr` procedure. Refer to Section 4.3.5 for a more detailed explanation.

[5]Memory between the 640K and 1Mb boundary

4.3 Memory Allocation

4.3.1 Introduction

When allocating memory there are five procedures to take note of. They are:

- `PROCEDURE NewBlock(size: LONGINT): InitPtr;`
- `PROCEDURE NewRec(VAR p: ADDRESS; tag: Tag);`
- `PROCEDURE NewSys(VAR p: ADDRESS; size: LONGINT);`
- `PROCEDURE NewArr(VAR p: ADDRESS; eltag: Tag; nofelem,nofdim: LONGINT);`
- `PROCEDURE NewSysArr(VAR p: ADDRESS; size,tdsize: LONGINT);`

The `NewSysArr` procedure is the latest addition as open arrays were introduced with Native Oberon. In the previous port, `NewRec`, `NewSys` and `NewArr` used to be functions with the return type as the address. Together with `NewSysArr` they have been turned into proper procedures returning the address in a variable parameter. `NewBlock` forms the basis of the other four procedures and is either called directly as in `NewSys` or indirectly as in `NewSysArr` which calls `NewSys`, in turn issuing a call to `NewBlock`.

The compiler is responsible for generating the correct code to call the required memory allocation routine depending on the type of structure involved. When creating statically linked images using the `BootLinker`, the memory routines must be explicitly specified.

Currently, Native Oberon distinguishes between the following set of structures:

- `POINTER TO RECORD` variables allocated with `NEW`
- `POINTER TO ARRAY` OF Type, where type is a pointer type, allocated with `NEW`
- `POINTER TO ARRAY` OF Type, where type is not a pointer, allocated with `NEW`
- Blocks allocated using the `SYSTEM.NEW` procedure

Memory for the first two structures are allocated by using the *RecBlk* and *ArrBlk* structures respectively. For the last two, *SysBlks* are used [3]. All dynamic structures are allocated on the heap and under Gneiss, this is the available memory after a VM is loaded into its given address space, usually 256K[6] (The default VM size) minus the VM binary size, stack base and header. The process of setting up the heap has been discussed in Section 4.2. Figure 3 illustrates the layout of the different heap blocks.

The `SFAM` bit fields in Figure 3 refers to the special markings of a tag used during allocation and garbage collection and is explained in Table 2. The meaning and use of these bit fields will become clear in the later discussions on memory allocation and garbage collection.

[6] The current port has increased the default size to 512K

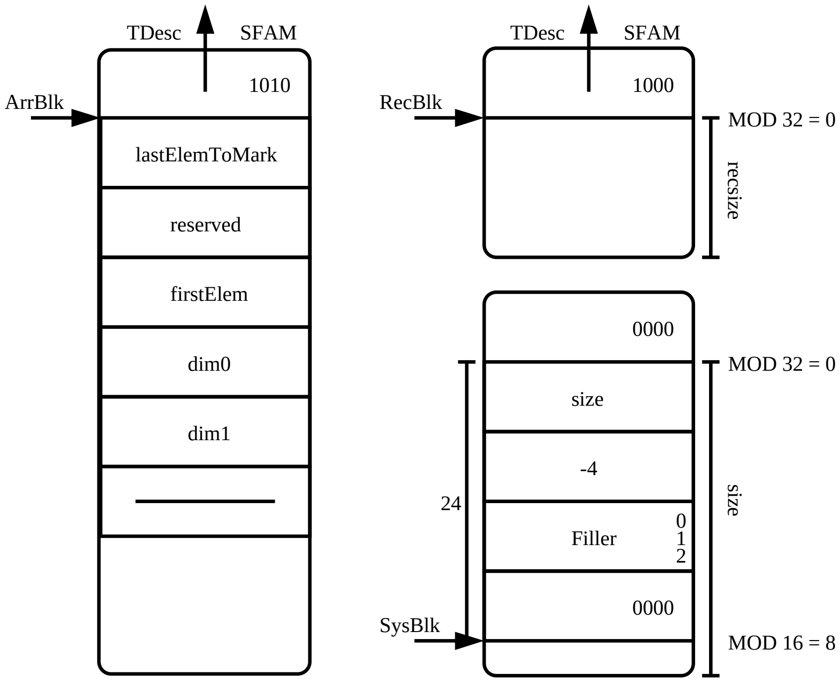

Figure 3: Heap Block Structures

Constant	Set Name	Value
SubObjBit	subobj	3
FreeBit	free	2
ArrayBit	array	1
MarkBit	mark	0

Table 2: Interpretation of the SFAM Fields

4.3.2 The Basis of Allocation: `NewBlock`

As mentioned in Section 4.3.1, `NewBlock` forms the basis of all the allocation procedures. Oberon keeps a list of pointers in a variable called `A` which points to all the free memory blocks. The `size` parameter of `NewBlock` is always rounded up to the nearest 32 bytes. Each index in `A` points to a list of free blocks belonging to a certain size except `A[0]` which is never used. `A[1]` therefore points to the list of free blocks that are 32 bytes in size, `A[2]` to those blocks being 64 bytes in size and so on. The exception being `A[N]` which points to larger blocks of variable sizes which are all a multiple of 32 and larger than those in `A[N-1]` [7].

The requested size is divided by the block size `B`, where `B=32`, to determine the list in which to search for a free block. The index into the free list array is kept in `i`. The address of the list is kept in `adr` and the address of the last list, `A[N]`, is kept in `AN`.

A first fit strategy is employed during the allocation process. `A[i]` is of course the preferred list in which the block must be allocated, but this is not always possible as there may not be any blocks available. `adr` is cast into an `InitPtr` and, if not `NIL`, it is taken as the pointer which points to the block of memory that must be used for the allocation. If this fails, the search gradually continues until the final list in `AN` is reached. If `adr` is not `NIL`, it is assumed that a valid free list has been found and the searching process terminates.

Should `AN` also be a `NIL` pointer, one of two situations might occur. If it is the first attempt at allocation, `firstTry = TRUE`, the garbage collector is called to possibly free up memory that might have become available since the last activation of the garbage collector. `firstTry` is set to `FALSE` and `NewBlock` calls itself. If the process repeats itself and fails again to satisfy the request the `reserve` pointer is set to NIL, the garbage collector is called in order to collect the reserved memory and a `CoreHalt` is issued to indicate that the request failed. The reserved memory is freed to allow Oberon to open a trap viewer to inform the user of the problem. The garbage collector will automatically reallocate the reserved memory once there is more than 64K of memory available.

The final part of `NewBlock` determines if the allocation caused fragmentation. It might be that the programmer requested 64 bytes of memory, but the smallest block available to satisfy the request was only 96 bytes. The remaining 32 bytes must be placed back into the free list structure. The remaining memory, if any, is calculated in `rest` and placed in the correct free list by repeating the set of calculations that were performed during the first part of `NewBlock`. The final step is to mark the block as free by setting the `FreeBit` in the `Tag` and inserting it into the list of free blocks. The sequence:

```
restptr^.next := A[i];
A[i] := SYSTEM.VAL(ADDRESS,restptr);
```

is responsible for the insertion into the free list.

Figure 4 gives an illustration of the process described above. It it is assumed that `A[2]` points to a `NIL` list, indicating that there are no free blocks available that are 64 bytes in size. `NewBlock` receives a request for 64 bytes. The Block is allocated from `N[3]` resulting in a block of 96 bytes being allocated. The block is split up into two blocks of 32 and 64 bytes in size respectively. The remaining 32 bytes are placed back at the head of the list in `A[1]` and a pointer in `ptr` is returned pointing to the block of 64 bytes.

[7] In the original implementation of Oberon the free list consisted out of four lists pointing to blocks of 16, 32, 64 and 128 bytes and a fifth list pointing to blocks sizes that are a multiple of 128 bytes [1].

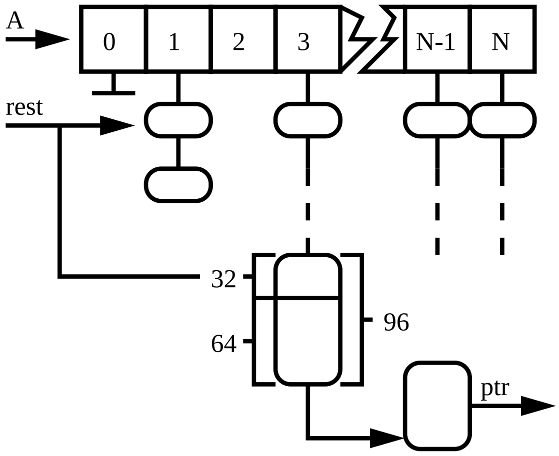

Figure 4: Basic Allocation using `NewBlock`

4.3.3 Structured Blocks: `NewSys`

The `SYSTEM.NEW` procedure is implemented in `NewSys` and uses a *SysBlk* as described in Figure 3. `NewSys` first adjusts the requested size before `NewBlock` is called to allocate a block of memory. The first adjustment is to compensate for the indirect tag and actual tag which constitutes 28 bytes in total. The second adjustment is made by clearing the lower 5 bits of the requested size in order to round it up to the nearest 32 bytes.

Once a pointer has been allocated, `NewSys` continues initializing the memory by filling in zero values. As can be seen from the code, the initialization is done in groups of 32 bytes, starting at the end of the structure and continuing until the start is reached. It is suspected that this [8] was done in order to allow for more effective register allocation on RISC based processors under the old *DECoberon* that was ported from *DOS Oberon*. This method of initialization is also present in the other routines discussed later on.

`NewSys` concludes by setting up the tag. Important fields to take note of are `z1` which is set to -4 and is used as a sentinel for the garbage collector. `z3` is set to the value of the double word (32 bits) stored at the address `[EBP+4]`. This is used for debugging purposes only in order for the `BootLinker` to locate where the allocation of the structure occurred.

4.3.4 Structured Blocks: `NewRec`

`NewRec` facilitates the dynamic allocation of records with the `NEW(ptr)` command and uses a `RecBlk` to allocate memory (Refer to Figure 3). The `tag` parameter is supplied by the compiler which generates the correct code when encountering the `NEW` statement. The `tag` parameter may be a NIL tag in which case a descriptor for an `ARRAY * OF CHAR` should be allocated. The initialization of the memory follows a similar pattern to that of `NewSys` by moving over the block of memory and clearing all the bytes to zero. The code to initialize the type descriptor is generated after the call to `NEW`.

Each record type is described by a type descriptor. The layout of the type descriptor is

[8] As per Pieter Muller, ETH

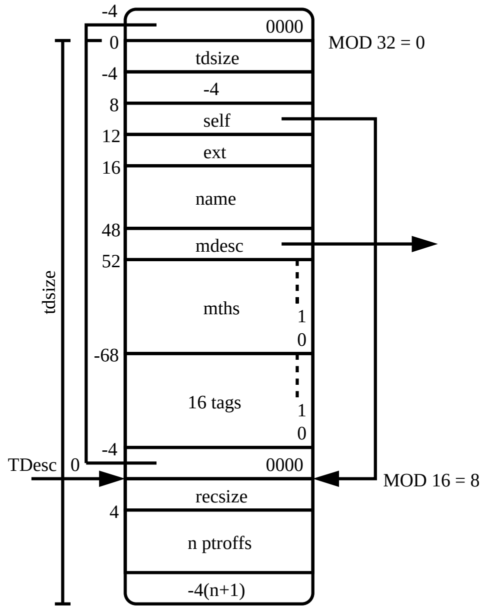

Figure 5: Type Descriptor for Records

shown in Figure 5. `tdsize` is the size of the type descriptor in bytes. The `-4` entry is used as a sentinel for the garbage collector. `ext` is the extension level of the record type. The `name` contains the name of the record type, but will be empty for anonymous types. `mdesc` is a pointer to the module descriptor in which the type was declared. `mths` are the type bound procedures for the record. The `tag` fields point to the base types of an extended record and will be `NIL` if not used. `recsize` is the size of the actual record described by the type descriptor. The `ptroffs` field contain the offsets of pointers inside the record. The negative field at the end is again used as a sentinel for the garbage collector.

4.3.5 Structured Blocks: `NewArr`

Procedure `NewArr` implements the allocation for `NEW(ptr)` where the parameter is a `POINTER TO ARRAY`. `NewArr` first checks to see if the `elTag` parameter, which is a pointer to the type descriptor of the record type used for the elements, is `NIL`. If this is the case, the element type is a pointer and not a record type and the `ptrElemTag` type descriptor is used, otherwise `NewArr` uses the `dynarrElemTag` type descriptor. Both these type descriptors are defined inside the `Kernel` module and are located during the Initialization process described in Section 4.2.

4.3.6 Structured Blocks: `NewSysArr`

`NewSysArr` is similar to `NewSys` except that the tags are marked differently after memory has been allocated with `NewSys` in order to distinguish it from normal SysBlks.

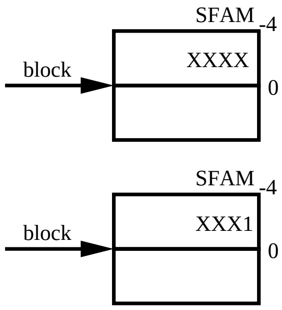

Figure 6: Marking Blocks

4.4 Monitoring Memory Usage

Memory usage is monitored by way of three functions, `Available`, `LargestAvailable` and `Used`. `Available` simply adds all the available block sizes in all the free block lists to calculate the total amount of available memory. `LargestAvailable` simply searches for the free block with the largest size. Since the free lists are ordered in an ascending fashion, the search is performed from `A[N]` towards `A[0]` in order to optimize the process. The amount of memory in use is calculated by subtracting the amount of free memory from the total heap size. `VMHeap` contains the heap size and is initialized in the `Init` procedure during the main initialization of the `Kernel` module.

4.5 Collecting Garbage

4.5.1 Introduction

Garbage collection in Oberon is carried out in two separate phases known as the *mark* and *sweep* phase. During the *mark* phase, all pointers that still contain valid references are marked and this is done on a module basis. Next the local stack is checked and possible pointers are also marked [9]. The process concludes by sweeping over the heap, collecting all blocks that were not marked, and placing them back into the free lists of memory blocks.

4.5.2 The Mark Phase

The mark phase begins inside the `GC` procedure by first marking each module so that the pointers to the module descriptors are not collected. The procedure distinguishes between those blocks that are sub-objects and those that are not.

If it is determined that `block` is not a sub-object, the tag is retrieved and a new tag called `marked` is constructed with the marked bit set. If `tag # marked`, then the tag belongs to a previously unmarked block that may require marking. Refer to Figure 6.

[9]Under Gneiss, each thread in the current VM under which the garbage collector exist is checked as each thread contains its own stack.

Tag variable	SFAM bits
`p↑.tag`	XXXX
`tag`	X0XX
`notmarked`	X0X0
`tdesc`	X000

Table 3: Tag Construction for Sweeping

4.5.3 Stack Traversal in Threads

Unlike Native Oberon which contains a single user level stack, a VM such as Oberon contains more than one thread and therefore more than one stack. Since each thread contains its own stack, the stack tracing code have to check each stack separately to locate all the pointers that requires marking in each thread. Once all the threads have been evaluated, the stack of the main thread is examined.

Each 32-bit value in the stack is taken and passed on as a possible candidate to the `Candidates` procedure. The reason for this is that it not possible to tell which stack entry does in fact contain a pointer of some sort. The whole process is repeated for every thread in the VM.

The `Candidate` procedure receives an address as a parameter which is checked against the upper and lower boundaries of the heap contained in `firstBlock` and `endBlock`. `Candidate` distinguishes between the different types of heap blocks by examining the alignment of the address which could be either `p MOD 32 = 0` or `p MOD 16 = 8`. For each address which conforms to the required alignment of a pointer the tag is retrieved and the address is placed into a list named `candidates`. Once the list becomes full, `CheckCandidates` is called.

`CheckCandidates` begins by sorting the list of addresses in the `candidates` array before examining them. Each address is examined according to the type of pointer and if valid, marked by calling the `Mark` procedure.

4.5.4 The Sweep Phase

`Sweep` begins by clearing all the free lists. This is done by assigning the `nil` constant to the root of each of the free lists kept in `A`. During this process, a duplicate list is constructed in `lastA` which simply points to the same address as `A`.

The sweep phase is carried out over the whole heap, starting at `firstBlock` and continuing until `endBlock` is reached. In order to determine which pointers must be swept, a group of tags are constructed from the original tag of the current pointer under investigation. The group of tags are represented by `tag`, `notmarked` and `tdesc`. Each tag is based on its predecessor with one additional bit cleared, except for the `SubObj` bit. Table 3 illustrates how the tags are constructed. The `X` value indicates a bit with an unknown setting.

The next step is to determine the size of the structure that is being dealt with and therefore `Sweep` must determine if it is an array or not by comparing `notmarked` and `tdesc`. If `notmarked # tdesc`, it is assumed that the structure is an array of some sort and the size is calculated by the number of elements and the basic element size. Non-array based structures contain the correct size in the `size` field of the tag. Once the size has been determined, a

`FreeBlockPtr` is constructed and inserted into `lastA` before continuing with the next pointer. Because the `lastA` structure points to `A`, the free list will automatically be reconstructed in `A`.

5 Module Modifications

5.1 Introduction

This section gives a detailed description of all the modules that were affected during the current port. Each sub-section contains a brief general overview of the module after which a detailed description of all the procedures and functions contained in the module is given.

Please note that the information contained in the underlying sub-sections only apply to the current port. It is likely that future ports would require some amount of modification. The authors have attempted to give an indication of this for each module, but this is purely speculative and therefore subject to change as new releases of Native Oberon become available.

Each module section contains a structured overview of the type of modifications that were made during the current port. Following is a list of subsections that might be encountered when dealing with a specific module. These subsections serve only as a guideline for future ports and should assist in reducing the time required to make modifications to individual modules.

- **Port Direction:** Indicates whether the module was taken from Native and modified to be compatible with Gneiss or if the opposite was done during the current port. In some cases, the port was accomplished by working in both directions.
- **System:** Indicates on which distribution the module is included.
- **Global Constants Added:** Indicates new constants added during the current port.
- **Global Constants Removed:** Indicates constants that were removed during the current port.
- **Global Types Added:** Indicates new types added during the current port
- **Global Types Removed:** Indicates types that were removed during the the current port.
- **Global Variables Added:** Indicates new variables that were added during the current port.
- **Global Variables Removed:** Indicates variables that were removed during the current port.
- **Procedures Removed:** Indicates procedures and functions that were removed during the current port.
- **Procedures Added:** Indicates procedures and functions that were added during the current port.
- **Procedures Modified:** Indicates procedures and functions that required modification during the current port.

- **Main Initialization and General Information:** Explains changes that were made to the main module initialization block as well as other general changes that might have been made to the module.

- **Future Ports:** This gives an estimate of the type of changes foreseen in order to port the module again as new releases of Native Oberon becomes available

5.2 IPC.Mod

Port Direction: Gneiss to Native.
System: CFS and Stand-alone.
Procedures Added:

- `PROCEDURE TransactionHack(porta,portb: LONGINT; req,rep: LONGINT; ms: LONGINT): LONGINT;`

- `PROCEDURE ReceiveRequestHack(porta, portb: LONGINT; req: LONGINT): LONGINT;`

- `PROCEDURE SendReplyHack(diff: LONGINT; rep: LONGINT): LONGINT;`

- `PROCEDURE CreatePortHack(port: LONGINT): LONGINT;`

- `PROCEDURE LookupGlobalPortHack(num: LONGINT; port: LONGINT): LONGINT;`

Main Initialization and General Information: Certain changes were warranted by changes in the new inline assembler provided by the OP2 Compiler. The new procedures that were added was done to accommodate the changes in the assembler. Refer to Section 3.4 for more details on the changes to the assembler. The required changes to the `IPC` module were made during 1997 by Eben Esterhuyse.
Future Ports: No major changes foreseen. Depends on changes inside the Gneiss kernel.

5.3 VMSL.Mod

Port Direction: Gneiss to Native.
System: CFS and Stand-alone.
Main Initialization and General Information: No modifications required. Only recompile with the OP2 Compiler.
Future Ports: No major changes foreseen.

5.4 ExpSL.Mod

Port Direction: Gneiss to Native.
System: CFS and Stand-alone.
Main Initialization and General Information: No modifications required. Only recompile with the OP2 Compiler.
Future Ports: No major changes foreseen.

5.5 DmaSL.Mod

Port Direction: Gneiss to Native.
System: CFS and Stand-alone.
Main Initialization and General Information: No modifications required. Only recompile with the OP2 Compiler.
Future Ports: No major changes foreseen.

5.6 GStrings.Mod

Port Direction: Gneiss to Native.
System: CFS and Stand-alone.
Main Initialization and General Information: This module was previously known as `Strings.Mod`. Native Oberon contains a module by the same name which differs in implementation. It has been renamed during the current port to avoid errors. The new `Strings` module under Native Oberon contains support for ISO strings and other extensions and is extensively used by Gadgets. The renaming was required to avoid module conflicts.
Future Ports: No major changes foreseen.

5.7 Trace.Mod

Port Direction: Gneiss to Native.
System: CFS and Stand-alone.
Main Initialization and General Information: No modifications required. Only recompile with the OP2 Compiler.
Future Ports: No major changes foreseen.

5.8 Kernel.Mod

Port Direction: Gneiss to Native. Native to Gneiss
System: CFS and Stand-alone.
Global Constants Added:

- `tCom1* = 3F8H;`
- `tCom2* = 2F8H;`
- `tCom3* = 3E8H;`
- `tCom4* = 2E8H;`
- `t7Bits* = 2;`
- `t8Bits* = 3;`
- `t1Stop* = 0;`
- `t2Stop* = 4;`
- `tParNone1* = 0;`
- `tParOdd* = 8;`

- `tParNone2* = 16;`
- `tParEven* = 24;`
- `tBaud110* = 0;`
- `tBaud150* = 32;`
- `tBaud300* = 64;`
- `tBaud600* = 96;`
- `tBaud1200* = 128;`
- `tBaud2400* = 160;`
- `tBaud4800* = 192;`
- `tBaud9600* = 224;`

The group of constants listed above are used to initialize the serial terminal. The `tComXXX` constants are used to select the COM port. `t7Bits` and `t8Bits` selects the number of data bits while `t1Stop` and `t2Stop` select the stop bits. `tParXXX` determines the parity and `tBaudXXX` selects the baud rate. **Global Constants Removed:**

- `IRQ* = 32;`
- `IDTSize = 32+16;`
- `IntA0 = 020H;`
- `IntA1 = 021H;`
- `IntB0 = 0A0H;`
- `IntB1 = 0A1H;`
- `V86EnterInt = 28;`
- `V86ExitInt = 29;`
- `KernelCodeSel = 1*8;`
- `KernelStackSel = 2*8;`
- `UserCodeSel = 3*8 + 3;`
- `UserStackSel = 4*8 + 3;`
- `DataSel = 4*8;`
- `KernelTR = 5*8;`
- `PS = 4096;`
- `PTEs = 1024;`

- KernelStackSize = 16*1024;
- PageNotPresent = 0;
- NormalPage = 7;
- V86 = TRUE;
- Operators = FALSE;
- VesaAdr = 0E0000000H;
- VesaSize = 400000H;
- DefaultPageHeap = 16384;
- DefaultStackSize = 128*1024;
- Rate = 1193180;

Global Types Removed:

- GateDescriptor
- SegmentDescriptor
- TSSDesc
- IDT
- GDT
- PageTablePtr
- PageDirectoryPtr
- Vendor
- V86Regs

Global Variables Removed:

- bt
- memTop
- dma0
- dma1
- dmafree
- handler0
- idt
- gdt

- ktss
- glue
- intHandler
- instip
- kernelpd
- v86pd
- handlingtrap
- oldcopro
- trapCR
- trapDR
- trapfpu
- mapPtr
- vregadr
- vframe
- configadr
- pspeed
- pageheap
- pageheap0
- pageheap1
- kpar
- apmofs
- powersave
- beepInit
- beeps
- cpuversion
- cpufeatures
- cpuvendor
- cpu

Procedures Removed:

- PROCEDURE StoreIDT(adr: LONGINT);
- PROCEDURE StoreGDT(adr: LONGINT);
- PROCEDURE WriteGDT;
- PROCEDURE WriteIDT;
- PROCEDURE Reboot;
- PROCEDURE -GoFrom0To3(ss,sp,cs: LONGINT; ip: Proc);
- PROCEDURE -CLTS;
- PROCEDURE LoadIDT(base,size: LONGINT);
- PROCEDURE LoadGDT(base,size: LONGINT);
- PROCEDURE SetTR(tr: LONGINT);
- PROCEDURE -ReadMSR(msr: LONGINT; lowadr,highadr: LONGINT);
- PROCEDURE -WriteMSR(msr: LONGINT; low,high: SET);
- PROCEDURE V86Exit;
- PROCEDURE V86Switch;
- PROCEDURE V86IntHandler;
- PROCEDURE -CR0(): LONGINT;
- PROCEDURE -CR2(): LONGINT;
- PROCEDURE -CR3(): LONGINT;
- PROCEDURE -CR4(): LONGINT;
- PROCEDURE -DR0(): LONGINT;
- PROCEDURE -DR1(): LONGINT;
- PROCEDURE -DR2(): LONGINT;
- PROCEDURE -DR3(): LONGINT;
- PROCEDURE -DR6(): LONGINT;
- PROCEDURE -DR7(): LONGINT;
- PROCEDURE -DS(): LONGINT;
- PROCEDURE -ES(): LONGINT;
- PROCEDURE -FS(): LONGINT;
- PROCEDURE -GS(): LONGINT;

- PROCEDURE -SS(): LONGINT;
- PROCEDURE InitProcessor;
- PROCEDURE StrToInt(s: ARRAY OF CHAR): LONGINT;
- PROCEDURE Fill4(dest,size,filler: LONGINT);
- PROCEDURE InitHeap;
- PROCEDURE IsRAM(adr: LONGINT): BOOLEAN;
- PROCEDURE CheckMemory;
- PROCEDURE ReadBootTable;
- PROCEDURE EnableEmulation;
- PROCEDURE DisableEmulation;
- PROCEDURE -StoreFPEnv(adr: LONGINT);
- PROCEDURE LoadSegRegs(data: LONGINT);
- PROCEDURE -HLT;
- PROCEDURE InterruptHandler;
- PROCEDURE InitInterrupts;
- PROCEDURE EnableMM(pd: LONGINT);
- PROCEDURE InitMemory;
- PROCEDURE -Call15;
- PROCEDURE ReadClock;
- PROCEDURE ClockHandler;
- PROCEDURE InitClock;
- PROCEDURE *TimerHandler;
- PROCEDURE InitTimer;
- PROCEDURE AllocatePage(VAR p: ADDRESS);
- PROCEDURE MapPage(pd: ADDRESS; virt,phys: LONGINT);
- PROCEDURE MappedPage(pd: ADDRESS; virt: LONGINT): LONGINT;
- PROCEDURE MapMem(pd: ADDRESS; virtAdr,size,phys: LONGINT);
- PROCEDURE GetCMOS(i: SHORTINT): INTEGER;
- PROCEDURE PutCMOS(i: SHORTINT; val: CHAR);

- PROCEDURE BCD2(x: INTEGER): LONGINT;
- PROCEDURE ToBCD(x: LONGINT): INTEGER;
- PROCEDURE WriteType(t: ADDRESS);
- PROCEDURE NMIHandler;
- PROCEDURE Unexpected;
- PROCEDURE SetupFPU;
- PROCEDURE Beep(hz: LONGINT);
- PROCEDURE Delay(ms: LONGINT);
- PROCEDURE InitBeeps;
- PROCEDURE BeepStr(msg: ARRAY OF CHAR);
- PROCEDURE Detect486(): BOOLEAN;
- PROCEDURE Detect586(): BOOLEAN;
- PROCEDURE DetectCoprocessor(): BOOLEAN;
- PROCEDURE SetupFlags;
- PROCEDURE Setup486Flags;
- PROCEDURE Setup586Flags;
- PROCEDURE CPUID(VAR vendor: Vendor; VAR version, features: LONGINT);
- PROCEDURE APM(VAR gdtofs, apmofs: LONGINT): BOOLEAN;
- PROCEDURE APMPowerOff;
- PROCEDURE -SwitchToLevel3(ss,sp,cs: LONGINT);

Procedures Added:

- PROCEDURE EnableSTrace*(t :BOOLEAN);
  This procedure is responsible for enabling or disabling IO privileges to perform tracing through the serial ports and sets `TraceOn := t`.

Procedures Modified:

- PROCEDURE CoreHalt(msg :ARRAY OF CHAR; n :LONGINT);
  `CoreHalt` now writes a short message followed by the error code to the `Messages` window. The error code contained in `n` has been left unchanged from Native Oberon.

- PROCEDURE WriteChar*(c :CHAR);
  Modified to test if `TraceOn = TRUE` before writing to the serial port.

- `PROCEDURE InitTracing*(base :INTEGER; speed :LONGINT; setting :SHORTINT);`
  Previously implemented as a parameterless procedure in Native. It now contains parameters to initialize the serial port. The parameters are selected by ORing the required constants as in the following example:
  `InitTracing(tCom2,9600,t7Bits+tParOdd);`

- `PROCEDURE MapPhysical*(physAdr,size :LONGINT; VAR virtAdr :LONGINT);`
  Although the procedure still carry the same set of parameters as the Native Oberon implementation, the actual mapping is done through the `ExpSvr`.

- `PROCEDURE NewDMA*(size :LONGINT; VAR adr, phys :ADDRESS);`
  Although the procedure still carry the same set of parameters as the Native Oberon implementation, the actual mapping is done through the `DmaSvr`.

- `PROCEDURE GC*;`
  Code related to the `FontRoot` variable that was used in the previous port has been removed. `FontRoot` was used as a dummy node to stop the garbage collector from collecting fonts. The stack tracing code was also modified to check each thread in the VM and was taken from the previous port.

- `PROCEDURE InitRuntime;`
  All code relating to `V86` mode has been removed.

- `PROCEDURE Shutdown*(code :LONGINT);`
  This procedure simply performs an `Exit(0)`.

- `PROCEDURE Idle*(code :  LONGINT);`
  Empty stub implemented for this routine.

Future Ports: The authors foresee significant changes to this module, subject to the changes in Gneiss and the `Kernel` module of Native. As already mentioned, the kernel is one of the most important modules in the Oberon hierarchy as it forms the basis for the other modules, especially device drivers.

5.9 Display.Mod

Under the previous port the display driver consisted out of two modules namely `Display` and `ColorDisplay`. `ColorDisplay` was written in pure assembler using the now outdated Assembler module. The `Display` module formed the front end of the display driver and also contained the necessary code for communicating with the `IOSvr`.

In Native Oberon, the display driver is contained in whole inside the `Display` module. Native Oberon uses a prefix in front of the module name to distinguish between the supported display hardware. For example, the standard VGA driver is contained in `VGA.Display.Mod` while the SVGA driver is contained in `SVGA.Display.Mod`.

All drivers under Native Oberon have direct access to the display memory located at an absolute address of `A0000H`. Under Gneiss, this area of memory is only accessible by mapping the absolute address unto a virtual address. The `MapDisplay` procedure is responsible for this.

Port Direction: Native to Gneiss.
System: CFS and Stand-alone.
Global Variables Added: These variables were added to allow the display driver to work with the `IOSvr`.

- `ioPort : IPC.Port;`
  Used for accessing the `IOSvr`
- `address : LONGINT;`
  Contains the virtual address for the video memory
- `displayFlag : BOOLEAN;`
  All the procedures responsible for writing to the display can only do so if the driver has access to it. This flag is toggled each time a switch occurs between Oberon and the Gneiss environment
- `update* : BOOLEAN;`
  Indicates to Oberon that a switch occurred back to the graphics mode. When set, Oberon will redraw the complete display. Switch back to text mode assigns a `FALSE` value to this variable.

Procedures Modified:

- `PROCEDURE Map*(X: INTEGER): LONGINT;`
  This function now only returns the virtual address that was used for mapping the VGA's physical address. The virtual address is contained in the `address` variable. Remember that this is an absolute address and not just the segment part.
- `PROCEDURE Dot*(col,x,y,mode: INTEGER);`
  Modified to test if the `displayFlag` variable is set.
- `PROCEDURE CopyBlock*(SX,SY,W,H,DX,DY,mode: INTEGER);`
  Modified to test if the `displayFlag` variable is set.
- `PROCEDURE CopyPattern*(col: INTEGER; pat: Pattern; X,Y,mode: INTEGER);` Modified to test if the `displayFlag` variable is set.
- `PROCEDURE ReplConst*(col,X,Y,W,H,mode: INTEGER);`
  Modified to test if the `displayFlag` variable is set.
- `PROCEDURE FillPattern*(col: INTEGER; pat: Pattern; pX,pY,X,Y,W,H, mode: INTEGER);`
  Modified to test if the `displayFlag` variable is set.
- `PROCEDURE ReplPattern*(col: INTEGER; pat: Pattern; X,Y,W,H,mode: INTEGER);`
  Modified to test if the `displayFlag` variable is set.
- `PROCEDURE DisplayBlock*(B: LONGINT; DX,DY,W,H,SX,SY,mode: INTEGER);`
  Modified to test if the `displayFlag` variable is set.

- `PROCEDURE Depth*(X: INTEGER): INTEGER;`
  Since the `IOSvr` only supports 16 colour modes, this function will only return a value of 4 indicating a depth of 16 colours (4 bits per pixel).

Procedures Added: Please note that the following procedures were taken from the Display module as implemented in the previous port.

- `PROCEDURE MapDisplay;`
  Maps the physical display memory unto a virtual address
- `PROCEDURE ReleaseDisplay*;`
- `PROCEDURE RegainDisplay*;`
- `PROCEDURE OpenDisplay;`
  Effectively switches from text to graphics mode
- `PROCEDURE CloseDisplay;`
  Close the graphics display and switch back to text mode

Main Initialization and General Information: Modified to include the initialization code required by Gneiss. This was taken 'As is' from the previous port.
Future Ports: Limited amount of changes foreseen. During the current port, Native 2.3.2 became available. As a simple exercise the new driver was adapted to the current port to investigate how quickly it could be accomplished. It took less than 3 minutes with only 5 lines of code being added or modified.

5.10 HDSL.Mod

Port Direction: Gneiss.
System: CFS and Stand-alone.
Main Initialization and General Information: No modifications required. Only recompile with the OP2 Compiler.
Future Ports: No major changes foreseen.

5.11 Disk.Mod

Port Direction: Gneiss to Native.
System: Stand-alone.
Global Variables Modified:

- `map :POINTER TO ARRAY OF LONGINT`
  Changes in the OP2 compiler prompted this modification. Memory for `map` is allocated using the `NEW(p,n)` statement instead of `SYSTEM.NEW`.

Procedures Added:

- `PROCEDURE Available*() :LONGINT;`
  Returns amount of available disk space.
- `PROCEDURE Marked*(sec :LONGINT) :BOOLEAN;`

- `PROCEDURE Size*() :LONGINT;`
  Returns the size of the local disk in terms of sectors.

Main Initialization and General Information: No modifications required. Only recompile with the OP2 Compiler.
Future Ports: No major changes foreseen.

5.12 Files.Mod

Port Direction: Gneiss to Native. Native to Gneiss
System: Stand-alone.
Global Variables Added:

- `PathChar* :Char;`
  Brought over from the `FileDir` module in order to be consistent with CFS.

Procedures Added:

- `PROCEDURE Copy(source, dest, size :LONGINT);`
  This procedure is for internal use only and facilitates a fast, low level copying function.

Procedures Modified:

- `PROCEDURE Register(f :File);`
  The procedure body was replaced with the Native implementation.

- `PROCEDURE Copy(source, dest, size :LONGINT);`
  This procedure is for internal use only and facilitates a fast, low level copying function.

- `PROCEDURE CleanUp*(f :SYSTEM.PTR);`
  Under the previous port, `CleanUp` was installed as a `GCnotifier`, but since notifiers are no longer used, `CleanUp` is installed using the `Kernel.RegisterObj` procedure. In order to stop the garbage collector from collecting the root file handle, `Kernel.DisableTracing` is used.

Main Initialization and General Information: Under the previous port, variables of type `File` was implemented using type `LONGINT`. It has been replaced with type `File` in the current port. Also note that the `Files` module still uses 1K sectors and not 2K sectors as is the case under Native Oberon. **Future Ports:** No major changes foreseen.

5.13 Files.Mod (CFS)

Port Direction: Gneiss to Native.
System: CFS.
Global Variables Added:

- `PathChar* :Char;`
  Moved from the `FileDir` module as CFS does not contain a `FileDir` module.

Procedures Added:

- `PROCEDURE Copy(source, dest, size :LONGINT);`
  Refer to Section 5.12

Procedures Modified:

- `PROCEDURE CleanUp*(f :SYSTEM.PTR);`
  Refer to Section 5.12
- `PROCEDURE FileSvrCall(Operation :LONGINT...);`
  `Buf` is now an extended type of `Core.BasicHeader` due to inconsistencies found with the compiler under the previous port. If the type is not extended, SYSTEM.VAL will not update the values of the fields of the extended record correctly under Native.
- `PROCEDURE GetName(F :File; VAR name :ARRAY OF CHAR);`

Main Initialization and General Information: Refer to Section 5.12.
Future Ports: No major changes foreseen.

5.14 CFS.Mod

Port Direction: Gneiss to Native.
System: CFS.
Procedures Modified:

- `PROCEDURE SvrCall(Operation :LONGINT...);`
  `Buf` is now an extended type of `Core.BasicHeader` due to inconsistencies found with the compiler under the previous port. If the type is not extended, SYSTEM.VAL will not update the values of the fields of the extended record correctly under Native.

Main Initialization and General Information: Longint's replaced with File pointers. Still uses 1K sectors, not 2K as under Native. Added disable tracing code. Modified to use the default viewer system installed instead of the System module viewers.
Future Ports: No major changes foreseen.

5.15 FileDir.Mod

Port Direction: Gneiss to Native.
System: Stand-alone.
Global Variables Removed:

- `PathChar* :Char;`
  Removed to be consistent with CFS which does not contain the FileDir module.

Main Initialization and General Information: No modifications required. Only recompile with the OP2 Compiler.
Future Ports: No major changes foreseen.

5.16 Input.Mod

Port Direction: Gneiss to Native.
System: CFS and Stand-alone.
Procedures Added:

- PROCEDURE KeyState(VAR keys :SET);
  Empty stub implemented to maintain compatibility with Native

Main Initialization and General Information: No modifications required. Only recompile with the OP2 Compiler.
Future Ports: No major changes foreseen.

5.17 Buffers.Mod

Port Direction: Gneiss to Native.
System: CFS and Stand-alone.
Main Initialization and General Information: No modifications required. Only recompile with the OP2 Compiler.
Future Ports: No major changes foreseen.

5.18 Modules.Mod

Port Direction: Native to Gneiss.
System: CFS and Stand-alone.
Main Initialization and General Information: No modifications required. Only recompile with the OP2 Compiler.
Future Ports: No major changes foreseen.

5.19 Viewers.Mod

Port Direction: Native to Gneiss.
System: CFS and Stand-alone.
Main Initialization and General Information: No modifications required. Only recompile with the OP2 Compiler.
Future Ports: No major changes foreseen.

5.20 Fonts.Mod

Port Direction: Native to Gneiss.
System: CFS and Stand-alone.
Main Initialization and General Information: No modifications required. Only recompile with the OP2 Compiler.
Future Ports: No major changes foreseen.

5.21 Reals.Mod

Port Direction: Native to Gneiss.
System: CFS and Stand-alone.
Main Initialization and General Information: No modifications required. Only recompile with the OP2 Compiler.
Future Ports: No major changes foreseen.

5.22 MenuViewers.Mod

Port Direction: Native to Gneiss.
System: CFS and Stand-alone.

Main Initialization and General Information: No modifications required. Only recompile with the OP2 Compiler.
Future Ports: No major changes foreseen.

5.23 Objects.Mod

Port Direction: Native to Gneiss.
System: CFS and Stand-alone.
Main Initialization and General Information: No modifications required. Only recompile with the OP2 Compiler.
Future Ports: No major changes foreseen.

5.24 Texts.Mod

Port Direction: Native to Gneiss.
System: CFS and Stand-alone.
Main Initialization and General Information: No modifications required. Only recompile with the OP2 Compiler.
Future Ports: No major changes foreseen.

5.25 TextFrames.Mod

Port Direction: Native to Gneiss.
System: CFS and Stand-alone.
Main Initialization and General Information: No modifications required. Only recompile with the OP2 Compiler.
Future Ports: No major changes foreseen.

5.26 Fonts.Mod

Port Direction: Native to Gneiss.
System: CFS and Stand-alone.
Main Initialization and General Information: No modifications required. Only recompile with the OP2 Compiler.
Future Ports: No major changes foreseen.

5.27 FPA.Mod

Port Direction: Native to Gneiss.
System: CFS and Stand-alone.
Main Initialization and General Information: No modifications required. Only recompile with the OP2 Compiler.
Future Ports: No major changes foreseen.

5.28 Oberon.Mod

Port Direction: Native to Gneiss.
System: CFS and Stand-alone.
Global Variables Added:

- `ioport :IPC.Port`

Procedures Added:

- `PROCEDURE SetPalette;`
  Performs the actual palette switch.
- `PROCEDURE Redraw;`
  This procedure was brought in from Gneiss and is responsible for refreshing the display.
- `PROCEDURE SetTimer(ms :LONGINT;`
  This procedure was brought in from Gneiss. It is used to reschedule the main loop in Oberon.
- `PROCEDURE Loop*;`
  `Redraw` is now called upon receiving a signal from the `Display` module. If there is any network input available, the related handler for the `NetTask` will be called. The procedure was also modified so that an idle loop would be allowed to be rescheduled by Gneiss.

Procedures Modified:

- `PROCEDURE ResetPalette;`

Main Initialization and General Information: Added `NetTask` for possible future expansion and compatibility with Gneiss implementation of previous port.
Future Ports: No major changes foreseen.

5.29 System.Mod

Port Direction: Native to Gneiss.
System: CFS and Stand-alone.
Global Variables Added:

- `pos :INTEGER;`
- `pat :ARRAY 32 OF CHAR;`

Procedures Modified:

- `PROCEDURE List*(name :ARRAY OF CHAR; time, date, size :LONGINT; VAR cont :BOOLEAN);`
  Native code replaced with Gneiss code of previous port.
- `PROCEDURE Directory*;`
  Native code replaced with Gneiss code of previous port.
- `PROCEDURE Watch*;`
  Removed code that reports any disk related information.
- `PROCEDURE Trap*(error,fp,pc, page :LONGINT);`
  Refer to Section 2.1.3.

Main Initialization and General Information: No modifications required. Only recompile with the OP2 Compiler.
Future Ports: No major changes foreseen.

5.30 Edit.Mod

Port Direction: Native to Gneiss.
System: CFS and Stand-alone.
Main Initialization and General Information: This module required modifications under CFS due to the way the the filesystem routines are implemented. No modifications were made for the Stand-alone implementation.
Future Ports: No major changes foreseen.

5.31 Strings.Mod

Port Direction: Native to Gneiss.
System: CFS and Stand-alone. Gadgets related module
Main Initialization and General Information: All code relating to the `FileDir` module was removed for the CFS as the `FileDir` module does not exist under CFS. No modifications were made for the Stand-alone implementation.
Future Ports: No major changes foreseen.

5.32 NamePlates.Mod

Port Direction: Native to Gneiss.
System: CFS and Stand-alone. Gadgets related module
Main Initialization and General Information: All code relating to the `FileDir` module was removed under CFS as the `FileDir` module does not exist under CFS. No modifications were made for the Stand-alone implementation.
Future Ports: No major changes foreseen.

5.33 TextDocs.Mod

Port Direction: Native to Gneiss.
System: CFS and Stand-alone. Gadgets related module
Main Initialization and General Information: This module required modifications under CFS due to the way the the filesystem routines are implemented. No modifications were made for the Stand-alone implementation.
Future Ports: No major changes foreseen.

5.34 PanelDocs.Mod

Port Direction: Native to Gneiss.
System: CFS and Stand-alone. Gadgets related module
Main Initialization and General Information: This module required modifications under CFS due to the way the the filesystem routines are implemented. No modifications were made for the Stand-alone implementation.
Future Ports: No major changes foreseen.

5.35 Directories.Mod

Port Direction: Native to Gneiss.
System: CFS and Stand-alone. Gadgets related module

Main Initialization and General Information: All code relating to the `FileDir` module was removed for the CFS as the `FileDir` module does not exist under CFS. No modifications were made for the Stand-alone implementation.
Future Ports: No major changes foreseen.

6 Future Work

This section tries to give suggestions for future work that might be attempted under the current port.

- Extending the `IOSvr` to include support for certain SVGA chip sets in order to allow Gadgets to work in a higher resolution or the possible replacement of the IOSvr by a new display mechanism.
- Modifying the networking support offered by Native Oberon in order to extend the Gadgets environment to allow the inclusion of modules that provide email and web based services, to do so under Gneiss.
- A detailed study to determine if Gneiss kernel development under the current port would proof to be a viable option to consider.
- Updating the `Scope` debugging tool designed by *de Villiers de Wet* to comply with the new object code format. As release 2.3.3 introduced yet another object file format, the authors suggest waiting until the stable release 2.3.4 is available before attempting this.

References

[1] N. Wirth and J. Gutknecht. *Project Oberon — The design and implementation of an Operating system and compiler.* Addison-Wesley, 1991.

[2] E. Esterhuyse. *The Caching File Server (CFS): CacheSVR v1.2a — A Technical Report.* Technical Report, 1999.

[3] http://www.oberon.ethz.ch/native/Tech.html - Technical notes regarding the current Oberon release in HTML format.

A Source Listings

A.1 Locating Garbage Collection Errors

```
(* pmuller 13.02.95/13.11.95
   jeloff 20.1.99 modified to work under Gneiss *)

MODULE TGC;

IMPORT
  Kernel,Trace,SYSTEM;

PROCEDURE Check(p: LONGINT);
VAR
  b: POINTER TO ARRAY 10000000 OF LONGINT;
  s: LONGINT;
  inside: BOOLEAN;
BEGIN
  Kernel.GC; (* this will put p^ on the free list if GC is broken *)
  inside := FALSE;
  LOOP
    s := Kernel.LargestAvailable();
    IF s <= 500 THEN EXIT END; (* 500 < 4000 *)
    DEC(s,32); (* allow space for type descriptor added by SYSTEM.NEW *)
    SYSTEM.NEW(SYSTEM.VAL(SYSTEM.PTR,b),s);
    IF (p >= SYSTEM.VAL(LONGINT,b)) & (p < SYSTEM.VAL(LONGINT,b)+s) THEN
      inside := TRUE;
      EXIT
    END
  END;
  IF inside THEN
    Trace.String("Inside!  Error found."); Trace.Ln
  ELSE
    Trace.String("Not inside.  Error not found."); Trace.Ln
  END;
  Kernel.GC
END Check;

PROCEDURE Inside1*;
VAR
  p: POINTER TO ARRAY 4000 OF CHAR;
BEGIN
  NEW(p);  p^ := "quite safe";
  Check(SYSTEM.VAL(LONGINT,p));
  Trace.String("p is ");  Trace.String(p^);
  Trace.Ln
END Inside1;
```

```
PROCEDURE Inside2*;
VAR
  p: POINTER TO ARRAY 4000 OF CHAR;
BEGIN
  SYSTEM.NEW(SYSTEM.VAL(SYSTEM.PTR,p),4000);  p^ := "quite safe";
  Check(SYSTEM.VAL(LONGINT,p));
  Trace.String("p is "); Trace.String(p^);
END Inside2;

BEGIN
  Inside1;
  Inside2;
  Kernel.Exit(0)
END TGC.

Compiler.Compile \s TGC.Mod~

BootLinker.Link tgc
  \new Kernel.NewRec \sysnew Kernel.NewSys \newarr Kernel.NewArr
  \newsysarr Kernel.NewSysArr
  \list Kernel.modules
  \integrate 20000000H
  IPC GStrings VMSL ExpSL RandomNumbers Trace DmaSL Kernel tgc ~
```

The test module `TGC` conducts a very general test of the garbage collector. If the test worked correctly, one should see the messages `p is quite safe` and `Not inside. Error not found`. The absence of any one of these messages indicates that there are still errors in the garbage collector.

A.2 The Decoder Module

```
MODULE Test;

PROCEDURE Sum(x,y :INTEGER) :INTEGER;
BEGIN
  RETURN x+y
END Sum;

PROCEDURE Go*;
VAR
  x,y,z :INTEGER;
BEGIN
  x := 50; y :=25; z := Sum(x,y)
END Go;
```

```
End Test.

.....
PROCEDURE Sum
0007H: 55                 push    ebp
0008H: 8B EC              mov     ebp,esp
000AH: 66|8B 5D 0C        mov     bx,12[ebp]
000EH: 66|8B 55 08        mov     dx,8[ebp]
0012H: 66|8B C3           mov     ax,bx
0015H: 66|03 C2           add     ax,dx
0018H: 8B E5              mov     esp,ebp
001AH: 5D                 pop     ebp
001BH: C2 08 00           ret     8
001EH: 6A 03              push    3
0020H: CC                 int     3
.....
```