Eidgenössische
Technische Hochschule
Zürich

Departement Informatik
Institut für
Computersysteme

Jacques Supcik

HP-Oberon™

The Oberon Implementation for Hewlett-Packard Apollo 9000 Series 700

February 1994

212

ETH Zürich
Departement Informatik
Institut für Computersysteme
Prof. Dr. N. Wirth

Author's address:
Institut für Computersysteme, ETH Zentrum, CH-8092 Zürich, Switzerland
e-mail: supcik@inf.ethz.ch

Table of contents

Abstract 5

1 Introduction 7

2 The HP-UX Environment 8
HP-UX Shared Libraries 8
Calling Conventions 9

3 The PA-RISC Compiler 14
The Portable Oberon-2 Front-end 14
Nullification 15
Instruction-Set 16
Register Usage 19
Stack Frame 24
Millicodes 25

4 The Bootstrap Process 28

5 Performance 32

6 Conclusions 34

Acknowledgements 35

References 36

Availability 37

Appendices 38
Appendix A: Calling Conventions in Oberon 39
Appendix B: Objects Allocated on the Heap. 44
Appendix C: HP Object File Format 47
Appendix D: Symbol File Format 49
Appendix E: Boot-file Format 51

Abstract

This technical report describes the implementation of HP-Oberon, a new member of the Oberon family running on Hewlett-Packard Apollo 9000 series 700 workstations.

It describes the HP-UX environment and the integration of Oberon in this environment. A description of the PA-RISC processor is then given and how the HP-Oberon compiler uses the new features offered by PA-RISC. After some performance measurements, the appendices contain implementation details for those who want to understand the heart of the system.

Trademark Notice

1 Introduction

Oberon is both a programming language and an operating system. As successor of Pascal and Modula-2, the Oberon programming language allows type extension, providing for an object-oriented programming style. The Oberon system features single-process multitasking, automatic garbage collection and dynamic loading of modules. "Make it as simple as possible", this quotation from A. Einstein has been the guideline during all the development of the Oberon project and the result is an extensible programming environment that is both compact and efficient.

In 1993 a project was started to port Oberon onto Hewlett-Packard workstations. At that time Oberon was already available for many architecture including the Apple Macintosh, IBM PC Compatibles running DOS or Windows, SPARCstations. DECstations, IBM RS/6000's and the SGI Indigo and the HP Apollo 9000 was an important UNIX workstation for which Oberon was still not available.

The first step of this project was to implement an Oberon-2 compiler generating native Precision Architecture (PA-RISC) code. Based on the portable Oberon-2 compiler [Cre91], this task has been reduced to writing a new back-end for the PA-RISC. As there are many similarities between the PA-RISC and the MIPS architecture, the MIPS back-end of DECoberon was taken as a starting point for the PA-RISC back-end.

After its completion, work on implementation of the Oberon operating system was started. Again the sources of DECoberon were used as a guideline. A new interface-module for HP-UX supporting shared libraries was written, and the module loader and the trap handler were adapted to the new object file format and the new memory layout. The rest of the system was completely portable and only needed to be recompiled with the PA-RISC compiler. In October 1993, the implementation of HP-Oberon was completed and was released on November 1st, 1993 after one month of intensive testing.

This report describes the implementation of HP-Oberon, focusing on interesting features of the PA-RISC architecture as well as problems encountered during the realization of the project.

2 The HP-UX Environment

HP-UX is the Hewlett-Packard implementation of the UNIX operating system. It is based on UNIX System V.3 with some BSD4.2 and BSD4.3 extensions, but in order to remain compatible with the System V UNIX standard, HP-Oberon does not use any of these BSD features. Hewlett-Packard is also a participant in the developing POSIX standard and intends to make HP-UX support this standard. Likewise, Hewlett-Packard is committed to follow the X/OPEN standard.

HP-UX Shared Libraries

HP-UX, like many modern UNIX implementations, offers the required functionality to support dynamic linking with shared libraries. A program using these features is no more a huge executable file into which all used libraries are statically linked. It is rather a small executable program calling functions distributed in dynamically loaded shared libraries. Using shared libraries, programs are much smaller than their *'archive libraries'* equivalent. They need less main memory, less disk space and are generally loaded faster.

Dynamic loading is also an important feature of the Oberon system and Oberon object files are similar to HP-UX shared libraries. Though it would have been possible to implement HP-Oberon object files being compatible with HP-UX shared libraries, it was much easier to implement them following the model of DECoberon object files. Thus the module loader could be ported without major changes.

Even if HP-Oberon object files are not compatible with HP-UX shared libraries, it still provides the functionality needed to open shared libraries and to obtain the address of procedures or variables included in them. This feature has been implemented to allow HP-Oberon programs to call procedures of the standard C library, the UNIX library, the X11 user interface library, or any other available shared library.

Listing 1 shows an example of calling a shared library from HP-Oberon. The exported procedure *WriteChar* of this *Demo* module calls the Unix function *write* via the *Write* procedure variable. This variable is initialized by *Init* using the procedure *Unix.dlsym*.

```
MODULE Demo;

    IMPORT SYSTEM, Unix;

    CONST StdOut = 1;

    VAR Write: PROCEDURE(fd, adr, n: LONGINT): LONGINT;

    PROCEDURE WriteChar*(ch: CHAR);
    VAR res: LONGINT;
    BEGIN
        res := Write(StdOut, SYSTEM.ADR(ch), 1);
    END WriteChar;

    PROCEDURE Init;
    VAR handle: LONGINT;
    BEGIN
        handle := Unix.dlopen("libc.sl", 0);
        Unix.dlsym(handle, "write", SYSTEM.VAL(LONGINT, Write));
    END Init;

BEGIN
    Init;
END Demo.
```

Listing 1: Example of using shared libraries

Calling Conventions

To live in harmony with HP-UX, HP-Oberon must follow the same calling conventions as the standard C, Pascal, or Fortran compiler. Concerning parameter passing, the convention is very similar to the one used by DEC-Oberon and ULTRIX: If parameters are 32 bits long INTEGERs or REALs, the first four are passed in registers (R26 to R23 for integer and F4 to F7 for floating-points) and the other ones are passed on the stack. If some parameters are arrays or records, their address is given as parameter[1] and the convention is the same as the one for LONGINTs. The convention for 64 bits LONGREALs is more complex and can be found in [HP2]. More details about register usage and the stack frame are given in

[1] In the case of a *value-parameter*, a copy of the parameter is done by the called procedure.

the next chapter. The convention concerning the jump mechanism in HP-UX is shown in figure 1.

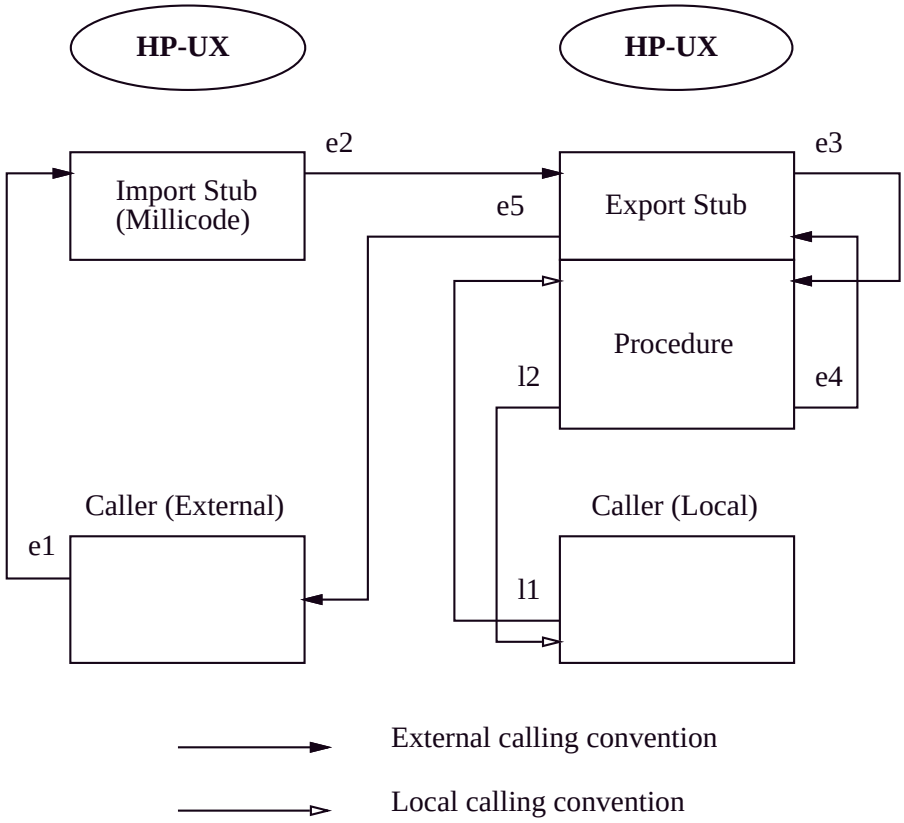


Figure 1: Calling conventions used in HP-UX

Local calls are simple and efficient; just one branch (l1) to call a procedure and a return (l2) to the caller are needed. On the other hand, external calls are rather expensive in the sense of the number of branches. A first branch (e1) is a local call to the import stub, which then makes an external call (e2) to the export stub of the called procedure. The export stub finally makes a local branch (e3) to the effective procedure. Once the procedure is ended, control is returned to the export stub (e4), which then returns to the caller(e5). Appendix A describes how the import and the export stub are implemented. The advantage of such a complicated mechanism is that the code generated for procedures is the same if the procedure is called locally or externally. The only difference is in calling the procedure directly or calling its export stub via an import stub. However the five branches needed for external calls make this scheme unattractive for Oberon where many external calls occur. To obtain better performance, a faster mechanism had to be implemented in HP-Oberon. Because all modules of HP-Oberon reside in the Oberon heap, that is in one memory

segment[2], the local calling convention could also be used for calls between different Oberon modules. This simpler convention has been used and is shown in figure 2.

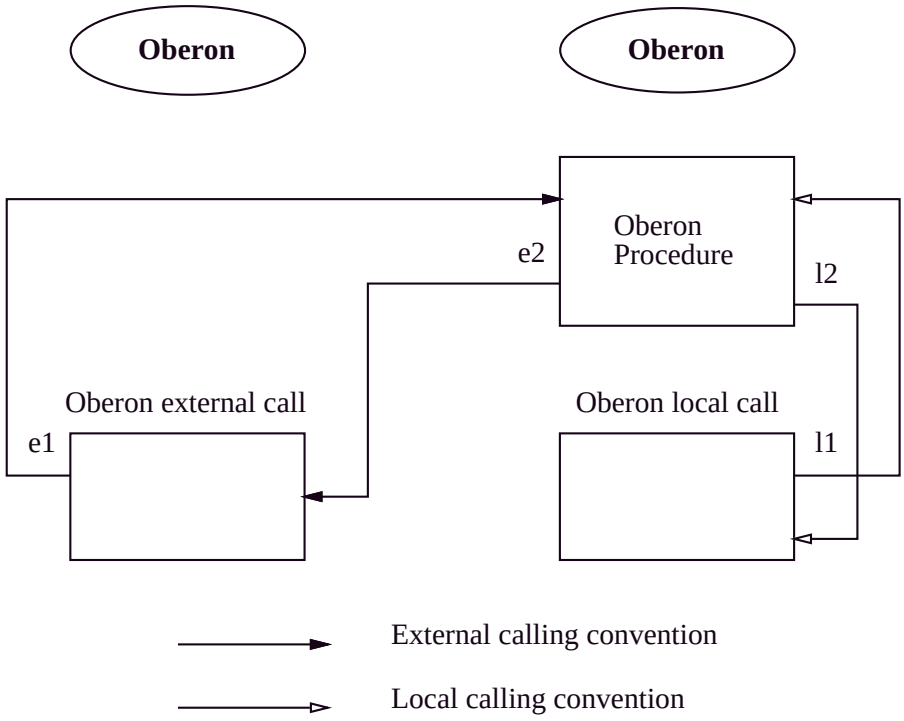


Figure 2: Calling conventions in HP-Oberon

This works well for direct procedure calls, but if the call is done via a procedure-variable, Oberon needs a way to find out if the procedure to be called is an Oberon or a 'foreign' HP-UX procedure. An Oberon procedure can be called using the local convention whereas a foreign procedure needs to be called via the import/export stub. This problem has been solved in the following way: As the address of a procedure is always a multiple of four, its last two bits are always zero. Bit 30 has already a special meaning in HP-UX but bit 31 can be freely used to differentiate Oberon from foreign HP-UX procedure addresses. An assignment of the form *procVar := Procedure*, where *procVar* is a procedure variable and *Procedure* an Oberon procedure, is then patched by the loader to ensure that bit 31 of *ProcVar* is set. When a procedure variable call occurs, this last bit is checked and the appropriate calling convention is used. Appendix A describes this mechanism with an example. The external call via an import stub is shown in figure 3.

[2] See section *Space Registers* in the next chapter

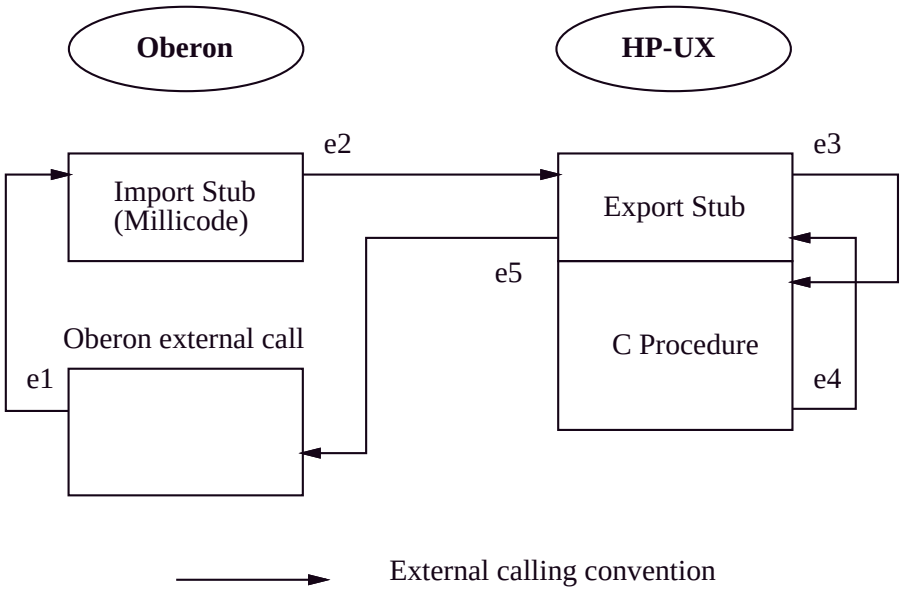


Figure 3: Oberon calls HP-UX

In the case where Oberon procedures are called by HP-UX — for low level features such as the trap handler or cleanup-procedures that have to be called by HP-UX when Oberon terminates — HP-UX makes the call using the standard convention for external calls, requiring that an export stub must be inserted just before the called Oberon procedure. Such a procedure is specially marked with a plus sign between the keyword *PROCEDURE* and the procedure name. Figure 4 shows this last situation.

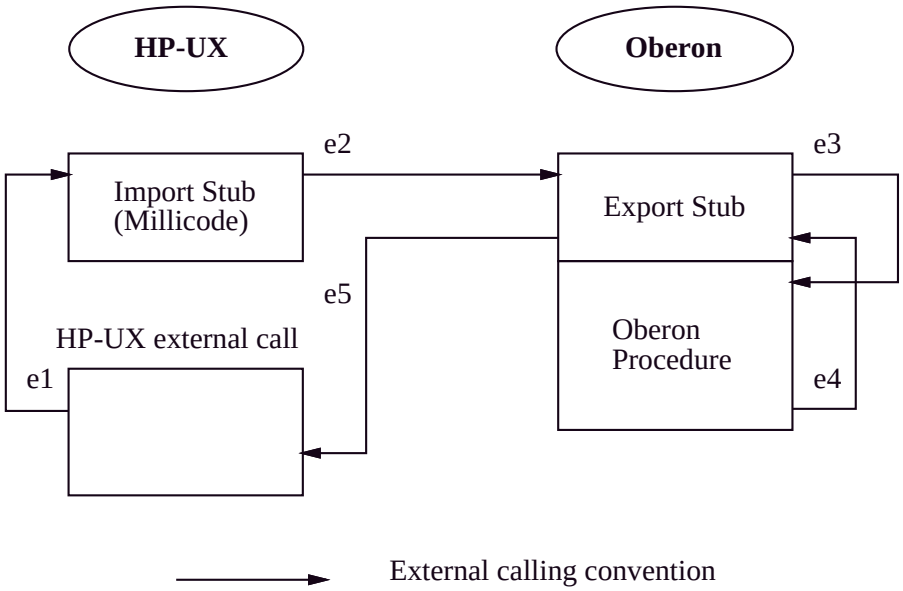


Figure 4: HP-UX calls HP-Oberon

To summarize, HP-Oberon is able to call any HP-UX procedure contained in a shared library and conversely, any HP-UX procedure can call an HP-Oberon procedure. If a new shared library is available, Oberon can use it without changes in its kernel. However the HP-UX calling conventions are rather inefficient and between two Oberon procedures a faster mechanism is used.

3 The PA-RISC Compiler

The Portable Oberon-2 Front-end

A compiler can be seen as two separate parts. The first one, the front-end, reads the source file, does syntactical and semantical analysis, and then generates an intermediate structure representing the program. The second part, the back-end, takes this intermediate representation and generates the object file consisting of the machine's object code and some additional information needed at run-time. Figure 5 shows these two phases.

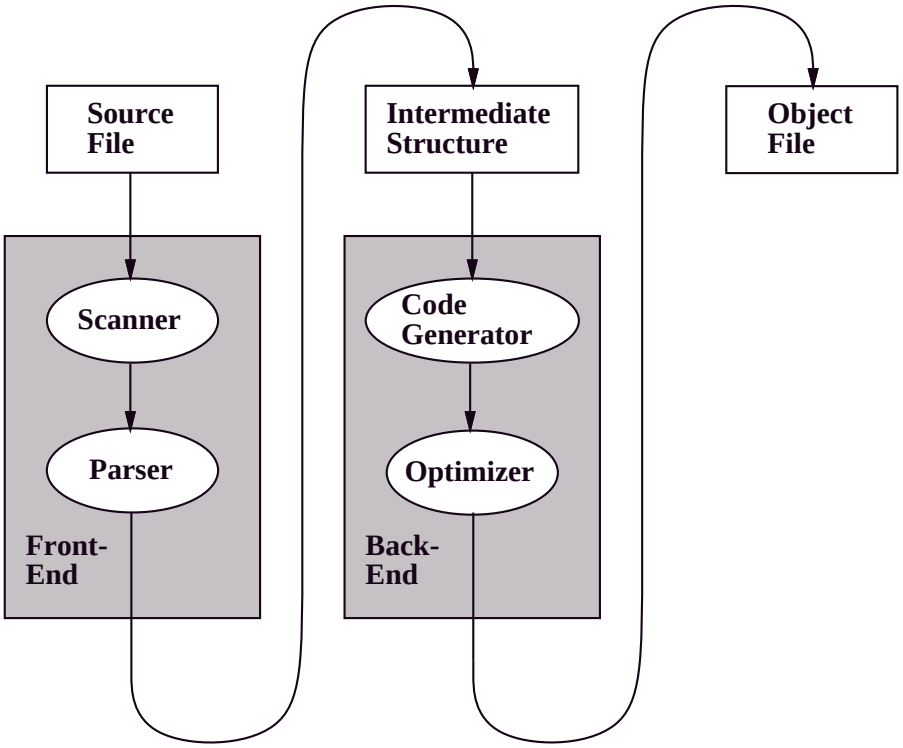


Figure 5: Compiler Front-end and Back-end

As shown in the figure, an optional optimization pass can be introduced during the compilation process. However, to keep the compiler small and fast, only simple optimizations have been included in the HP-Oberon compiler.

HP-Oberon, like almost all other Oberon implementations is based on the portable OP2 compiler [Cre91]. The OP2 front-end consists of the scanner, the parser, and the symbol table management. It generates a syntax-tree and a symbol

table as intermediate program representation. This part of the compiler is exactly the same in all implementation. The back-end then uses this syntax-tree and produces native object code. As this code is target machine specific, the second part is not portable and has to be implemented anew for each different architecture.

Like in DEC-Oberon and SGI-Oberon, the *Object-Model* has been used in the HP-Oberon compiler to check for module compatibility giving much more flexibility than the previously used *time-stamp* method. It is beyond the scope of this report to describe this model, however more information about this topic can be found in the PhD thesis of Régis Crelier [Cre94].

Nullification

Precision-Architecture is in many points similar to the MIPS architecture. However, it has some interesting new features not found in MIPS processors like for example the concept of *nullification*.

A nullified instruction is an instruction that will not be executed. This instruction will just be skipped, not affecting the machine state. All branch and computational instructions can nullify the execution of the following instruction. For branch instructions, nullification is specified explicitly. In computational instructions, nullification is performed conditionally, based on the outcome of a test.

Nullification is a very powerful tool and it is used in many places of the compiler like range checking or long-jumps. The following listings show examples of nullification.

```
COMCLR,<=   rb, rc, ra      ra := 0; IF rb <= rc THEN skip
LDO         1(0), ra        ELSE ra := 1 END;
```

Listing 2: Assign the logical expression (rb > rc) to ra

```
COMICLR,>   len, index, r0  IF index < len THEN skip
HALT        15              ELSE HALT
```

Listing 3: Index check

```
COMB,cond       target      (12-bit Target ⇒ jump distance is 8k)

COMCLR, ~cond   r0
BL              target, r0  (17-bit Target ⇒ jump distance is 256k)
```

Listing 4: Conditional long jumps

Instruction-Set

This section gives an overview of interesting instructions of the PA-RISC processor and their use in the Oberon compiler. Some of these instructions are of special interest because they are not found in other popular RISC architectures. A detailed description of the complete PA-RISC instruction-set can be found in the HP-Reference manual [HP1]

Extract/Deposit

Even though they are not new in PA-RISC, the *extract* and the *deposit* instruction are not available on all RISC processors and deserve special attention. The first one, like its name says, extracts *len* bits of a given source register and puts them in the last bits of a destination register. This instruction comes in two variants, either signed or unsigned, where the most significant bit of the extracted block is replicated to the left of the block. The next figure shows the result of the EXTR* r, p, len, t instruction.

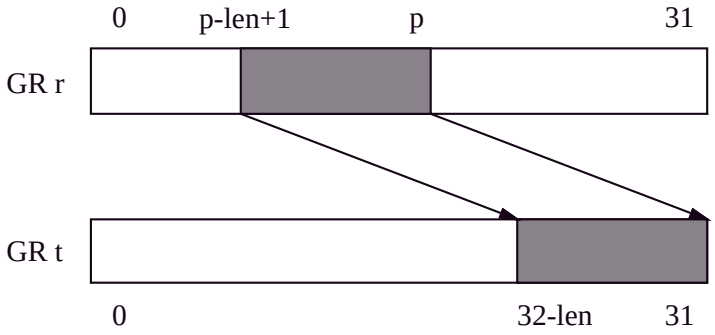


Figure 6: Extract instruction

The Extract instruction is used by the compiler to implement logical and arithmetical right shifts, sign extension, range check and bit-test.

The next listing shows how range checks are implemented. This little code segment finds out if the value in register *r* is in the *INTEGER* range (between -32768 and 32767). The idea is that *r* is in this range if its bits 0 to 17 are the same (all one or all zeroes). This can be implemented by testing if the value of the register remains the same after sign extending its last 16 bits.

```
EXTRS       r, 31, 16, r1   (sign extended extract of the 16 last bits)
COMCLR, =   r, r1, r0       (nullify the next instruction if r = r1)
TRAP        RangeTrap
```

Listing 5: INTEGER range check

The *Deposit* instruction is the opposite of *Extract*. It takes an immediate value or a register as source and deposits its last *len* bits left from position *p* in a target register. The other bits of the destinations are either unchanged or cleared according to the kind of the extract instruction. The semantic of this instruction is better explained by the following figure.

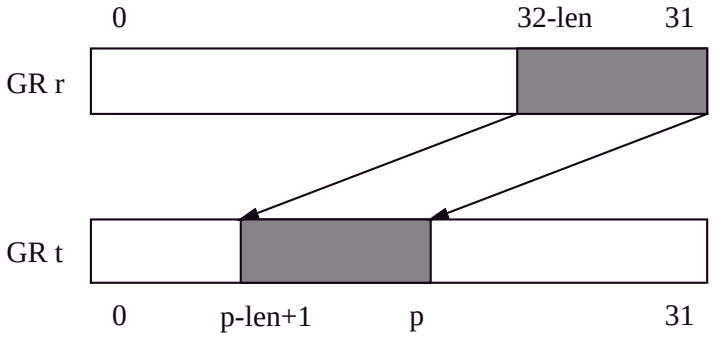


Figure 7: Deposit instruction

The Oberon compiler uses Deposit to implement left shifts, and to set or clear a bit in a given register.

SHnADD

Another interesting instruction is *Shift-and-Add*, which shifts the first operand *n* bits to the left and adds the result to the second operand. The result is then stored in the destination register. The shift *n* can be either one, two, or three.

PA-RISC has neither a multiply nor a multiply-step instruction that could be used to implement the integer multiplication, therefore this operation must be achieved using a series of shift-and-add instructions. Replacing those two instructions by only one performing these two operations simultaneously, will allow for an efficient implementation of integer multiplication.

Multiplication with a Constant

If one of the operands is a constant, the sequence of shifts and adds is known in advance and the compiler can generate SHnADD instructions. The following listing shows the MIPS and the PA-RISC pseudo instruction sequence to multiply register *x* with the constant *10* putting the result into register *z*. Register *t* is a temporary register.

```
MIPS          PA-RISC
z := 8*x      z := 8*x
t := 2*x      z := z + 2*x     (* SH1ADD x, z, z *)
z := z+t
```

Listing 6: Constant multiplication

Register Multiplication

More complex is the problem if both operands are variables because in this case the shift and add sequence is not known in advance. The traditional *add-shift* algorithm shown in listing 7 computing *R3 := R1 * R2* could be implemented, but its poor efficiency would not be acceptable.

```
R3 := 0;
FOR i := 0 TO 31 DO
      CASE R1{0} OF              (* R1{0} is the MSB of R1 *)
        0: R3 := 2 * R3 + 0
      | 1: R3 := 2 * R3 + R2
      END;
      R1 := 2 * R1;
END
```

Listing 7: Add-Shift multiplication

A faster way to implement the multiplication is to shift R1 by more than one bit per iteration. A shift by two would not be much better but a shift by four or even by eight as shown in listing 8 would make the multiplication much faster. On the other hand, the price to pay for this speedup is the need for a big case-table. In its standard system library, HP decided to implement the integer-multiplication with a shift by eight, needing a 4Kbyte case-table. HP-Oberon uses the same code for its multiplication.

```
R3 := 0;
FOR i := 0 TO 3 DO
        CASE R1{0..7} OF        (* R1{0..7} are the 8 MSB of R1 *)
          0: R3 := 256 * R3 + 0
        | 1: R3 := 256 * R3 + R2
        | 2: R3 := 256 * R3 + 2*R2
        | 3: R3 := 256 * R3 + 3*R2
        ...
        | 254: R3 := 256 * R3 + 254*R2
        | 255: R3 := 256 * R3 + 255*R2
        END;
        R1 := 256 * R1
END;
```

Listing 8: More efficient multiplication

Load/Store Operation

Load and *Store* operations are instructions moving data between main memory and a register. For an *integer-load*, there are two possible addressing modes for accessing memory. In the first one, *Based-Long*, the effective address is the sum of a register and a constant between -8192 and +8191. In the second one, *Indexed*, the effective address is the sum of a register and another register multiplied by a scaling constant. Unfortunately the *Indexed* mode is not available for an *integer-store*; the only possible addressing mode is *Based-Long*. There is another irregularity with *floating-point* Load/Store operations where both *Indexed* and *Based* addressing mode are available, but in the *Based* mode the constant offset must be between -16 and +15. This addressing mode is known as *Based-Short*. These irregularities are rather uncomfortable, but despite this, all of the above addressing modes are used by the Oberon compiler.

Register Usage

To remain compatible with the host operating system, HP-Oberon must follow the HP-UX conventions regarding register-usage. The PA-RISC 1.1 processor has of 32 general integer registers, 32 floating-point register, 7 space registers, and 25 control registers.

General and floating-point registers are used to hold local variables, to pass arguments to procedures, and to hold temporary values during expression evaluation. Space and control registers contain system state information.

General Registers

The 32 general integer registers of PA-RISC are each 32 bits wide. R0 is a special register whose value is always zero. Twenty registers (R3-R22) are general purpose registers and are used to hold local variables or temporary values, and four registers (R26-R23) are reserved to pass parameters to procedures. The following table shows in detail the convention imposed by HP-UX.

Register	Name	Usage Convention	Saved by
R0		Zero value register	
R1		Scratch register	caller
R2	RP	Return pointer and scratch register	
R3-R18		General purpose	callee
R19-R22		General purpose	caller
R23	arg3	Argument register 3	caller
R24	arg2	Argument register 2	caller
R25	arg1	Argument register 1	caller
R26	arg0	Argument register 0	caller
R27	DP	Global data pointer	stubs
R28	ret0	Function return register on exit Function result address on entry	caller
R29	SL ret1	Static link register on entry Millicode function return Function return register for upper part function result	caller
R30	SP	Stack pointer	
R31		Millicode return pointer and scratch register	caller

Floating-Point Registers

PA-RISC processors are very powerful in floating point intensive applications. To achieve its good performance, the processor includes a big set of 32 floating-point registers, each of them 64 bits wide. The following table summarizes the HP-UX conventions regarding floating-point registers.

Register	Name	Usage Convention	Saved by
F0		Coprocessor status	
F1-F3		Exception registers (cannot be modified)	
F4	fret farg0	Floating-point return register Single-precision argument register 0	caller
F5	farg1	Single-precision argument register 1 Double-precision argument register 0	caller
F6	farg2	Single-precision argument register 2	caller
F7	farg3	Single-precision argument register 3 Double-precision argument register 1	caller
F8-F11		General purpose	caller
F12-F15		General purpose	callee
F16-F21		General purpose (only on PA-RISC 1.1)	callee
F22-F31		General purpose (only on PA-RISC 1.1)	caller

Space Registers

With 32 bit wide addresses, processors can access 4 GBytes of virtual memory. Because this seems not to be enough, PA-RISC processors have extended the memory addressing modes using the concept of *spaces*. The global virtual memory is then organized as a set of linear spaces, each being 4 gigabytes long. Depending on the level of PA-RISC architecture, there are 2^{16}, 2^{24} or 2^{32} virtual spaces allowing 48 bit, 56 bit, or 64 bit wide addresses. Their usage convention is shown in the following table.

Register	Usage Convention
SR0	Link code space ID
SR1-SR3	General use
SR4	Program Text
SR5	Process private data
SR6	Shared data
SR7	Operating System's public code, literal, and data

Figure 8 shows how the space is specified using the first 2 bits of the address, and the convention for SR4-SR7.

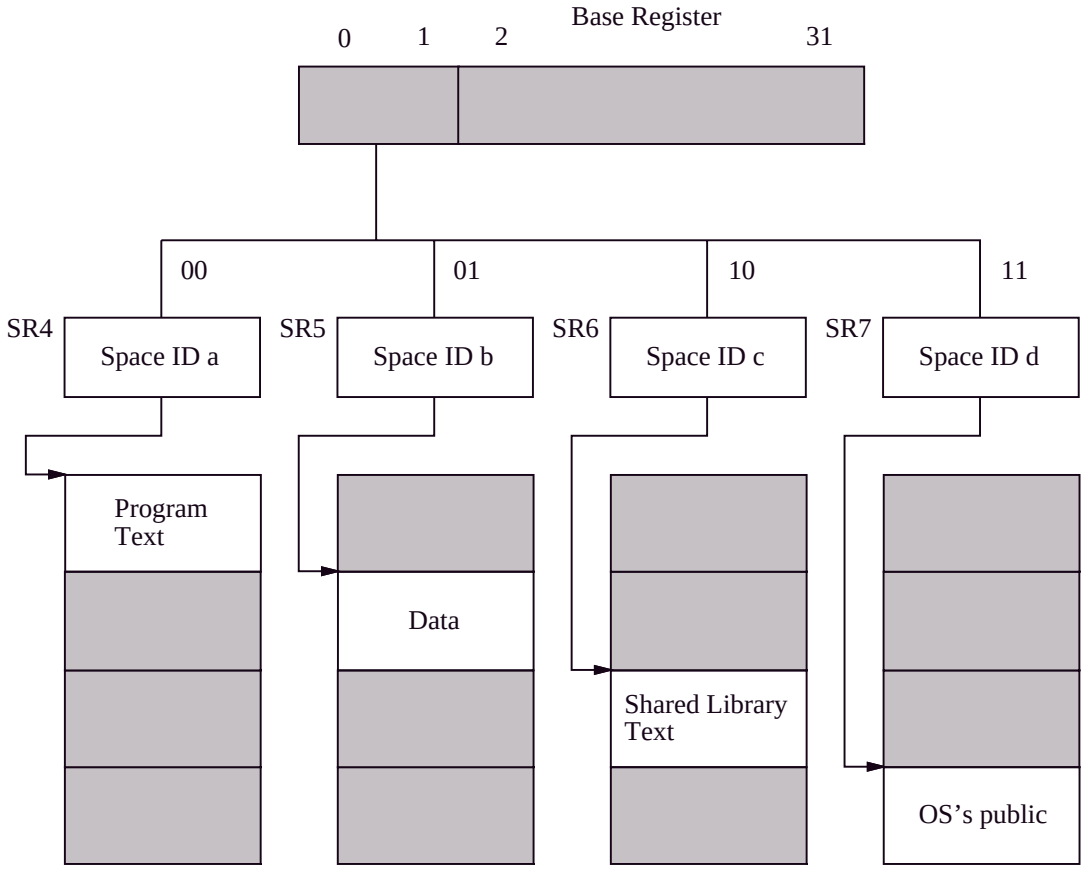


Figure 8: Space registers convention

Control Registers

There are 25 control registers labeled CR0 and CR8..CR31, which contain system state information. The only Control Register used by HP-Oberon is CR11, the *Shift Amount Register* (SAR). This control is readable and writable by code running at any privilege level and is used to determine the displacement by *variable shift*, *variable extract,* and *variable deposit* instructions.

```
VAR
   i: INTEGER;    (* R22 *)
   s: SET;        (* R21 *)
BEGIN
   IF i IN s THEN
      ...
   END;
END.
```

Translates to

```
   MTCTL R22, SAR              Move i (R22) to SAR (CR11).
   VEXTRU, OD  R21, 1, R0      Extract 1 bit from s (R21) starting at SAR
                               and put the result in R0 (no effect)
                               Nullify the next instruction if the rightmost
                               bit of the result is 1.
   BL,n  END, R0               Branch to the end of the if statement..
```

Listing 9: Examples of variable extract using the SAR control register

Stack Frame

As seen before, the arguments to a procedure are passed by the four registers R26 to R23. However, if there are more than four arguments, some of them have to be passed on the stack. The same is true for local variables; some can be held in registers and others have to be put on the stack. The next figure shows the structure of the stack during the execution of a procedure.

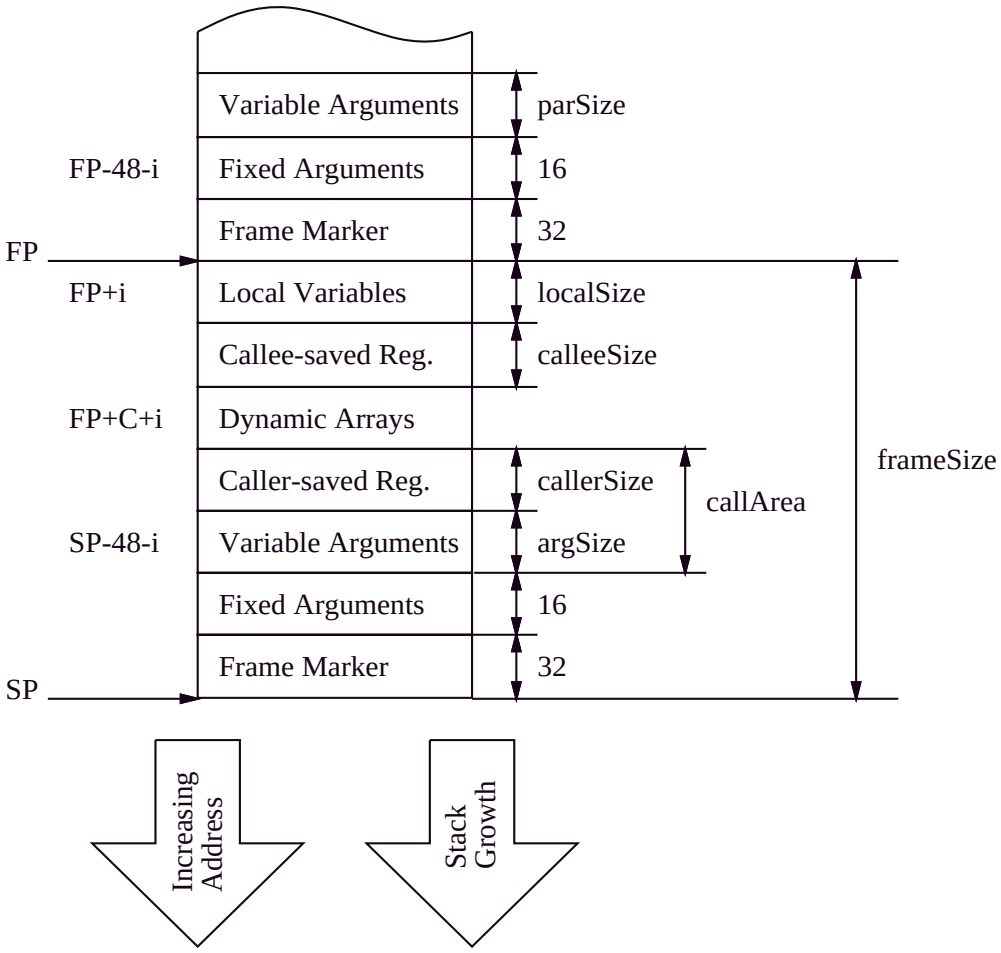


Figure 9: Stack Frame

The next table gives a more detailed view of the Frame Marker.

Variable arguments	...		
	SP-56	arg word 5	
	SP-52	arg word 4	
Fixed arguments	SP-48	arg word 3	
	SP-44	arg word 2	
	SP-40	arg word 1	
	SP-36	arg word 0	
Frame marker	SP-32	External data/LT Pointer (LPT)	Set before call
	SP-28	External-stub RP (RP')	Set after call
	SP-24	External RP	Set after call
	SP-20	Current RP	Set after call
	SP-16	Static Link	Set before call
	SP-12	Clean Up	Set before call
	SP-8	Relocation Stub RP (RP")	Set after call
	SP-4	Previous SP	Set before call
Top of frame	SP-0	Stack pointer	

A complete example of a procedure call, showing how the stack is involved during the calling process, is given in Appendix A.

Millicodes

A millicode is very similar to a local procedure or a local function. The only difference is that arguments are only passed in registers, no stack frame is needed, no registers need to be saved, and the result is returned in register 29 instead of register 28. Register 1 is used as a scratch register and register 31 for the return address. The advantage of millicode compared to normal functions is that the calling conventions are simpler and much more efficient.

A millicode call with two arguments is done in 5 steps :

1 Save R26 and R25 in temporary registers if they are live.
2 Load R26 and R25 with the arguments.
3 Call the millicode
4 Relocate the result R29 in another temporary register.
5 Restore R26 and R25 if they were saved.

In the current version of HP-Oberon, millicodes are used to implement integer multiplication, division, modulo, and the export stub *(DynCall)*. However, because of the difference between the Oberon-2 [Moe91] and the C [Ker88] definition of the division and modulo, the millicodes provided by HP-UX can not be used. The following listing shows how to compute the division and the modulo according to the Oberon definition if the numerator is negative.

```
PROCEDURE OberonDIV(a, b: LONGINT): LONGINT
BEGIN
   IF a < 0 THEN
      RETURN -((-a-1) CDIV b) - 1
   ELSE
      RETURN a CDIV b
   END
END OberonDIV;

PROCEDURE OberonMOD(a, b: LONGINT): LONGINT
BEGIN
   IF a < 0 THEN
      RETURN -((-a-1) CMOD b) + b - 1
   ELSE
      RETURN a CMOD b
   END
END OberonMOD;
```

Listing 10: Oberon DIV and MOD functions using C functions

Because the PA-RISC architecture is in some points similar to the MIPS architecture, one can think that porting the Oberon compiler for HP was a rather easy task and actually no big problem occurred during its implementation. However there are many new features in PA-RISC and, in order to produce efficient code, the compiler has to use them. On the other hand, there are subtle differences between MIPS and PA-RISC that were not obvious at the beginning of the project.

Most of the time needed for the implementation of the compiler was spent in these two areas: Using the new features of PA-RISC and solving problems due to small differences.

4 The Bootstrap Process

As already explained before, the HP-Oberon compiler does not generate standard HP-UX shared libraries, and there is no way for HP-UX to directly use these Oberon-specific object files. So how can Oberon be started from HP-UX? The solution to this bootstrap problem has been solved by a small C program called the *boot-loader*. The task of the boot-loader is to allocate the Oberon-heap with a standard *malloc* call, then to install the system modules of Oberon into the heap, and finally to start Oberon by jumping to the entry-point of the system modules previously loaded. The system modules consist of the following Oberon modules: *Unix*, *Console*, *Kernel*, *Files*, and *Modules*. To keep the C boot-loader as simple as possible, these modules are previously linked by the *Boot-Linker* to form the *Oberon-Boot-file*.

As the boot-loader is written in C, it can be compiled and linked to make an executable file that can be then started from HP-UX.

To build the boot-file, the boot-linker loads in a pseudo heap all modules composing this boot-file. The pseudo heap is then compressed and written to the boot-file, together with the fixup information needed to relocate this pseudo heap. Figure 10 shows how the pseudo heap looks like when the boot-file is generated.

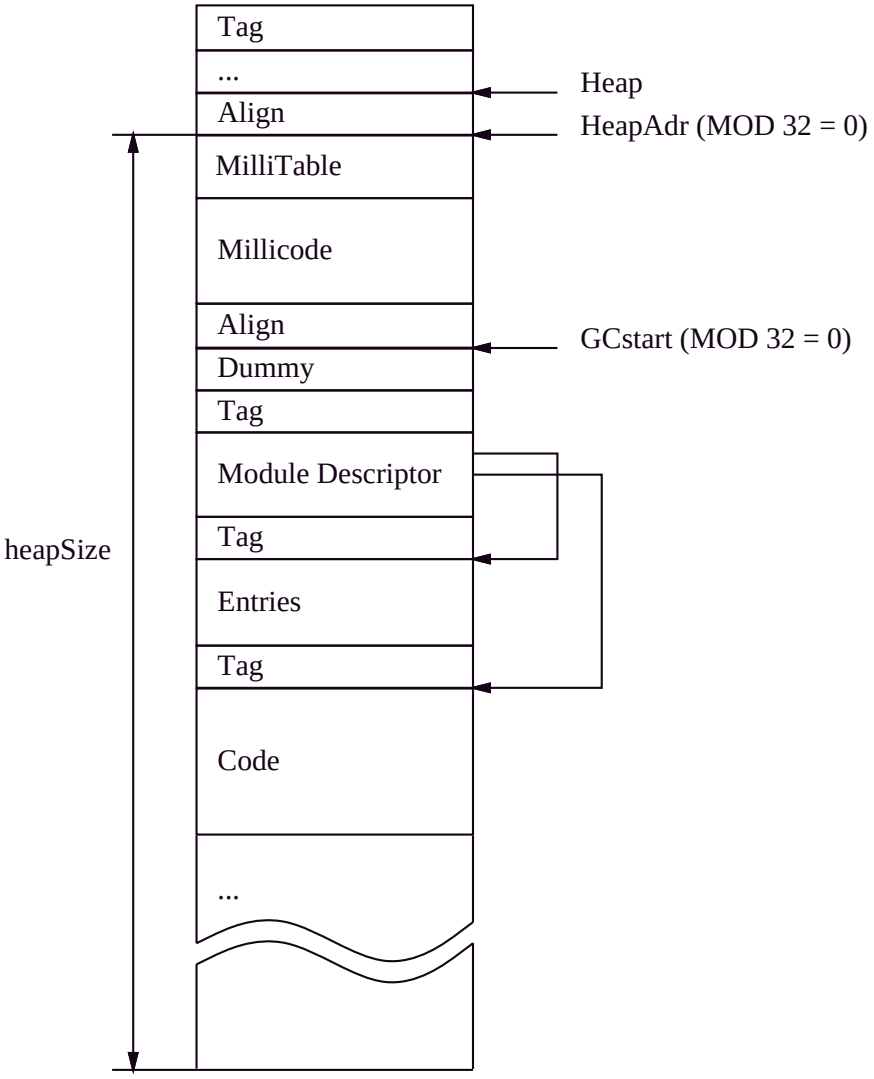


Figure 10: Image of the boot file in memory

Some Oberon modules need information from the boot-loader, for example the Oberon garbage-collector needs to know the start address of the Oberon heap. This link between Oberon and the boot-loader is made via the *Unix.dlsym* procedure. *Dlsym* is the first global variable of *Unix*, the first loaded module, and during the boot process, this variable is initialized by the boot-loader to point to the *dlsym C-procedure* of the boot-loader. Oberon can then call the Unix.dlsym procedure to obtain information from the boot-loader. The next figure shows how this link is done.

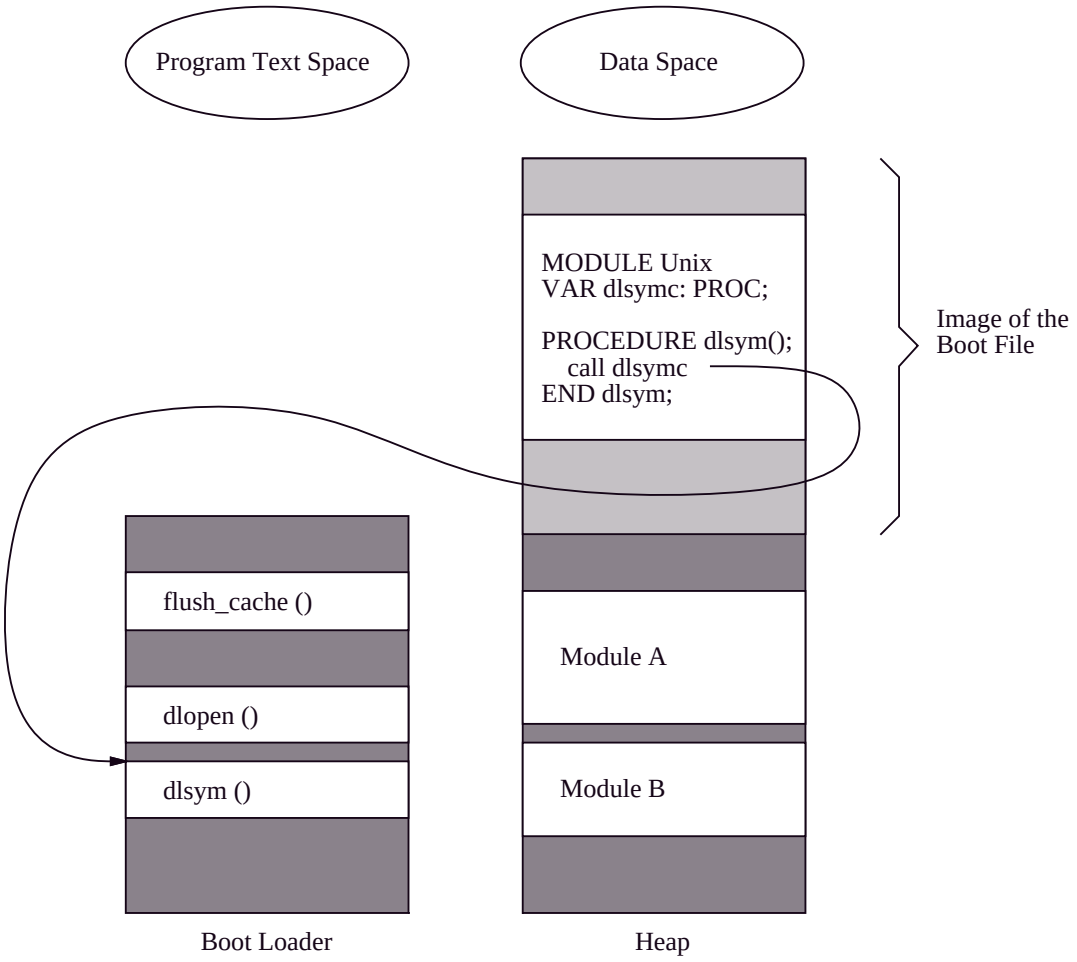


Figure 11: Link between HP-Oberon and HP-UX

The following listing shows an extract of the dlsym procedure of the boot-loader.

```
int dlsym(int handle, char *symbol, int *adr)
{
   int res;

   if (strcmp("dlopen", symbol) == 0) *adr = (int)dlopen;
   else if (strcmp("dlclose", symbol) == 0) *adr = (int)dlclose;
   ...
   else if (strcmp("heapAdr", symbol) == 0) *adr = heapAdr;
   ...
   else res = shl_findsym (...)    /* search in the shared-library */
}
```

Listing 11: Extract of the boot-loader

To summarize, the following steps are executed during the loading of Oberon:

1 Parsing of the command line to find out the startup options.
2 Allocation of the Oberon heap.
3 Initialization of the file searching path by parsing the OBERON environment variable.
4 Loading of the boot-file into the heap.
5 Relocation of the code of the boot-file.
6 Initialization of the millicode table.
7 Flushing of the caches.
8 Execution of the startup code, usually the body of module *Modules*.

5 Performance

The Dhrystone benchmark has been used to measure the performance of the various Oberon implementations. Like any other benchmark, dhrystone is not perfect, but it gives a good image of the CPU and compiler efficiency. On HP Apollo 9000, a C version of the dhrystone benchmark has also been compiled using the HP C compiler with and without optimization.

Platform	Processor	Clock	Dhrystones/sec		
			Oberon	C	C + opt.
HP 735	PA-RISC	99 MHz	113'636	93'677	170'940
SGI Indigo-2	MIPS R4000	100 MHz	80'645		
SPARC Classic	Micro SPARC	50 MHz	50'632		
PC-Compatible	i486 DX2	66 MHz	46'986		
HP 715	PA-RISC	33 MHz	37'950	26'316	55'633
IBM RS6000	RIOS I	25 MHz	34'013		
DECstation	MIPS R3000	25 MHz	33'037		
PC-Compatible	i486 DX	33 MHz	23'494		
Sun Sparc	SPARC I	20 MHz	18'400		
Macintosh II fx	MC 68030	40 MHz	11'338		
Ceres-2	NS 32532	25 MHz	6'677		

The comparison with other machines has to be considered very critically. Dhrystone is not a global performance index, it just gives a value reflecting the efficiency of a little part of the compiler and of the CPU.

Much more interesting is the comparison with C, optimized C, and Oberon. Figure 16 shows these values graphically, together with the compilation time.

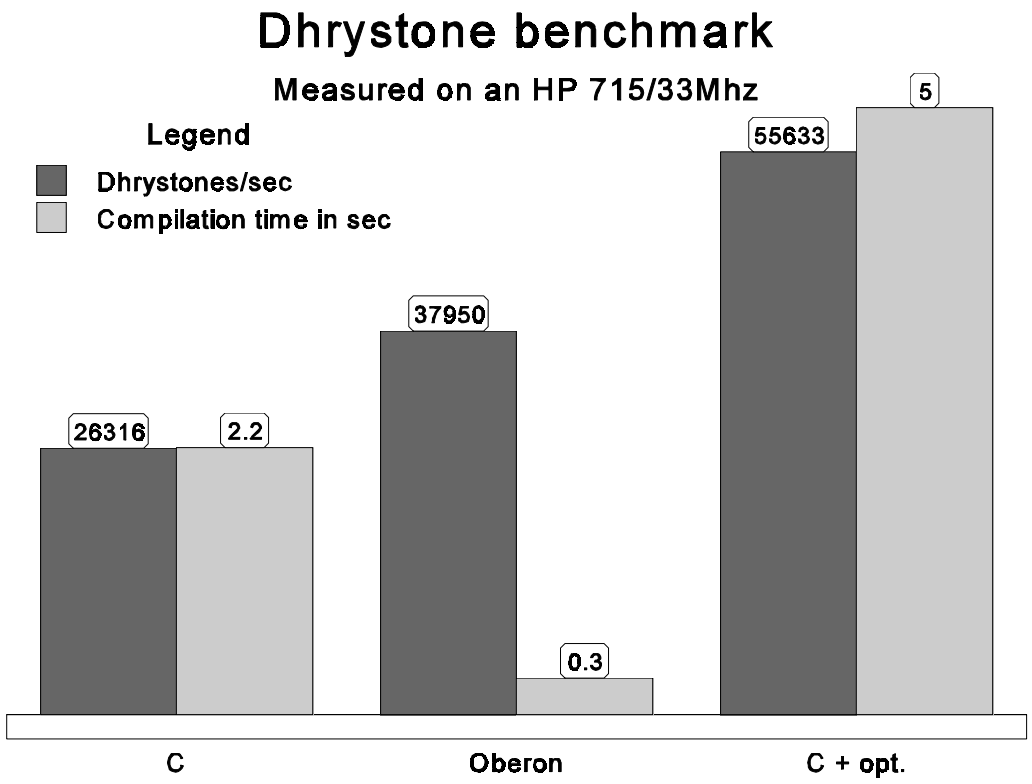


Figure 12: Oberon versus C in dhrystone benchmark

To keep Oberon simple and efficient, only few optimizations have been implemented in the compiler. No common subexpression elimination, no instruction scheduling, no loop unrolling have been implemented. Clearly the C optimizing compiler, in which many optimization techniques are used, generates better code than Oberon.

However, with only few simple optimizations, the code generated by the Oberon compiler is appreciably better than the one generated by the standard C compiler, even though Oberon requires only a fraction of the time used by the C compiler.

To give another idea of the compiler's speed, the complete Oberon system, including the text-editor, graphic-editor, compiler, display-driver, network-manager, etc..., altogether representing 88 modules, 1.38 MB source code and 1.19 MB object code, compiles itself in about 14 sec. on a 99MHz HP 735 and in less than 45 sec. on a 33MHz HP 715. The Oberon compiler compiles itself in 2.1 sec. on a 735 and in 7 sec. on a 715.

6 Conclusions

In the light of compiler construction the PA-RISC processor is not significantly different from the other RISC processors. Like all RISC processors it has many registers and a quite simple instruction format. Like almost all RISC processors, it transparently deals with register interlocking, freeing the compiler from this complicated and tedious task. The more interesting feature in respect of compiler writing is perhaps nullification. Nullification is very simple and useful for some fixed sequences of instructions like index-check or long jump, but it is difficult to use this concept more generally in a single pass compiler. The same is true for the *shift-and-add* instructions and generating a good sequence for constant multiplication demands a lot of reflection.

The main reproaches a compiler designer could direct to Precision Architecture are certainly the lack of integer multiplication and the irregularities in the load/store operations. Despite the rather complex conventions dictated by Hewlett-Packard, the integration of Oberon in HP-UX was not really difficult. The challenge was rather to allow HP-Oberon making use of the HP-UX millicode library and of the standard UNIX/X11 shared libraries in a simple and efficient way.

The simplicity and the portability of Oberon made the implementation of HP-Oberon in less than one man-year possible. The PA-RISC Oberon compiler has been first cross-developed on a DECstation running DEC-Oberon. Despite the fact that the MIPS processor of the DECstation is little-endian and PA-RISC is big-endian, once the boot-strap was done, the HP-Oberon compiler could be compiled on the PA-RISC machine without changes.

HP-Oberon is now a new member of the Oberon family and contributes to making even more people discover and use the Oberon system.

Acknowledgements

My thanks first go to Régis Crelier for providing the OP2 front-end, the MIPS back-end, and the source of all DECoberon modules. His great experience with RISC architectures helped me a lot. I am grateful for the patience he had in answering all my questions.

I would like to thank Niklaus Wirth for being the initiator of the project and for the precious advice he gave me during the implementation of HP-Oberon.

Thanks to their experiences with RISC implementations Marc Brandis and Josef Templ always had the right answer to my questions.

I would like to acknowledge Stefan Ludwig and Marc Brandis for having proofread this report, making many suggestions and constructive criticisms.

I am also greatly indebted to Roland Dietiker at Hewlett-Packard Schweiz for providing me with an Apollo 9000 715/33 PA-RISC workstation.

References

[HP1] *PA-RISC 1.1 Architecture and Instruction Set, Reference Manual*
HP Part No. 09740-90039
First Edition, November 1990

[HP2] *PA-RISC Procedure Calling Conventions, Reference Manual*
HP Part No. 09740-90015
Second Edition, January 1991

[HP3] *HP-UX Portability Guide*
HP Part No. B2355-90025
Second Edition

[HPj4] *Hewlett-Packard Journal*
June 1992

[Cre94] Régis Crelier
Separate Compilation and Module Extension
Ph.D. dissertation, ETH Zürich. To be published

[Cre91] Régis Crelier
OP2: A Portable Oberon-2 Compiler
Proceeding of the Second International Modula-2 Conference.
Loughborough University, UK. September 1991

[Moe91] Hanspeter Mössenböck, Niklaus Wirth
The Programming Language Oberon-2
Technical Report, 160. Institute for Computer Systems, ETH Zürich, CH.
May 1991

[Wir90] Niklaus Wirth
The Programming Language Oberon
Technical Report, 143. Institute for Computer Systems, ETH Zürich, CH.
November 1990

[Wir92] Niklaus Wirth, Jürg Gutknecht
Project Oberon. The Design of an Operating System and Compiler
ACM Press, Addison-Wesley Publishing. 1992

[Coo1] S.M. Cooper, J. M. Goertz, S. E. Levine, J. L. Mosher, S. R. Sieler, Jr., J. Van Damme
Beyond RISC ! An Essential Guide To Hewlett-Packard Precision Architecture
Software Research Northwest, Inc.

[Ker88] Brian W.Kernighan, Dennis M. Ritchie
The C Programming Language
2nd edition, Prentice Hall, 1988

Availability

HP-Oberon as well as all other implementations of Oberon are freely available from the ftp server of the department of computer science. The name of this server is neptune.inf.ethz.ch and its IP number is 129.132.101.33. The directory Oberon is subdivided as follows:

Oberon/HP700	contains the implementation of Oberon for Hewlett-Packard Apollo 9000 series 700 workstations.
Oberon/<Machine>	contains the implementation of Oberon for <Machine>
Oberon/Docu	contains some documentation about the Oberon language and the Oberon System.

Appendices

The following appendices give deeper information about some points of the HP-Oberon implementation. They are intended for people needing to have more details about the heart of the system. Because HP-Oberon inherited a lot from DEC-Oberon, some of these appendices can also be useful to DEC-Oberon users.

Appendix A: Calling Conventions in Oberon

Given the following Oberon module:

```
MODULE Test;

    IMPORT SYSTEM;

    TYPE Proc = PROCEDURE(x: INTEGER; VAR y: INTEGER);

    VAR p: Proc;

    PROCEDURE P(x: INTEGER; VAR y: INTEGER);
    BEGIN
        x := y;
    END P;

    PROCEDURE+ Q(x: LONGINT);
    BEGIN
    END Q;

    PROCEDURE Do;
        VAR a, b: INTEGER;
    BEGIN
        P(a, b);
        p(a, b);
    END Do;

BEGIN
    p := P;
END Test.
```

The Decoder produces this output:

```
Test

code size: 240
data size: 8
const size: 0
sysCall link: 000CC
data link: 000B4
```

```
imports:   (pos 26)

exports:   (pos 27)
LinkProc entry=00040 link=000A0

commands:   (pos 32)

pointers:   (pos 33)

constants:   (pos 34)

uses:   (pos 276)

refs:   (pos 277)
$$  pc=00000
  saved.r=2
  saved.f=
  frameSize=48  callArea=0
  p  ProcTyp  adr=0
P  pc=00040
  saved.r=
  saved.f=
  frameSize=48  callArea=0
  x  Int  adr=r22
  VAR y  Int  adr=r21
Q  pc=00060
  saved.r=
  saved.f=
  frameSize=48  callArea=0
  x  LInt  adr=r22
Do  pc=0008C
  saved.r=2 17
  saved.f=
  frameSize=72  callArea=8
  a  Int  adr=r17
  b  Int  adr=2
```

```
code:   (pos 35)

(* BODY *)
00000000H  E8400020H BL             00000018H,RP
00000004H  08000240H OR             r0,r0,r0
00000008H  4BC23FD1H LDW            -24(s*,SP),RP
0000000CH  004010A1H LDSID          (s*,RP),r1
00000010H  00011820H MTSP           r1,s0
00000014H  E0400002H BE,n           0(s0,RP)

00000018H  6BC23FD9H STW            RP,-20(s*,SP)
0000001CH  B7DE0060H ADDI           48,SP,SP

00000020H  22DF1FF5H LDIL           FFA3E800H,r22
00000024H  36D60000H LDO            0(r22),r22
00000028H  20200000H LDIL           00000000H,r1
0000002CH  68360000H STW            r22,0(s*,r1)

00000030H  B7DE07A1H ADDI           -48,SP,SP
00000034H  4BC23FD9H LDW            -20(s*,SP),RP
00000038H  E840C000H BV             r0(RP)
0000003CH  08000240H OR             r0,r0,r0

(* PROCEDURE P *)
00000040H  B7DE0060H ADDI           48,SP,SP
00000044H  081A0256H OR             r26,r0,r22
00000048H  08190255H OR             r25,r0,r21

0000004CH  46B60000H LDH            0(s*,r21),r22
00000050H  D2D61FF0H EXTRS          r22,31,16,r22

00000054H  B7DE07A1H ADDI           -48,SP,SP
00000058H  E840C000H BV             r0(RP)
0000005CH  08000240H OR             r0,r0,r0

(* PROCEDURE Q *)
00000060H  E8400020H BL             00000078H,RP
00000064H  08000240H OR             r0,r0,r0
00000068H  4BC23FD1H LDW            -24(s*,SP),RP
0000006CH  004010A1H LDSID          (s*,RP),r1
00000070H  00011820H MTSP           r1,s0
00000074H  E0400002H BE,n           0(s0,RP)

00000078H  B7DE0060H ADDI           48,SP,SP
0000007CH  081A0256H OR             r26,r0,r22
```

```
00000080H  B7DE07A1H ADDI            -48,SP,SP
00000084H  E840C000H BV              r0(RP)
00000088H  08000240H OR              r0,r0,r0

(* PROCEDURE Do *)
0000008CH  6BC23FD9H STW             RP,-20(s*,SP)
00000090H  B7DE0090H ADDI            72,SP,SP
00000094H  6BD13F89H STW             r17,-60(s*,SP)

00000098H  37D93F75H LDO             -70(SP),r25
0000009CH  0811025AH OR              r17,r0,r26
000000A0H  205F1EF5H LDIL            F7A3E800H,RP
000000A4H  E4406000H BLE             0(s5,RP)
000000A8H  081F0242H OR              r31,r0,RP

000000ACH  37D93F75H LDO             -70(SP),r25
000000B0H  0811025AH OR              r17,r0,r26
000000B4H  20200000H LDIL            00000000H,r1
000000B8H  48360000H LDW             0(s*,r1),r22
000000BCH  C7F6C012H BB,>=,n         r22,31,000000CCH
000000C0H  D6C01C1EH DEPI            0,31,2,r22
000000C4H  E6C06000H BLE             0(s5,r22)
000000C8H  B7E20018H ADDI            12,r31,RP
000000CCH  20BF1FF5H LDIL            FFA3E800H,r5
000000D0H  E7E06000H BLE             0(s5,r31)
000000D4H  081F0242H OR              r31,r0,RP

000000D8H  4BD13F89H LDW             -60(s*,SP),r17
000000DCH  B7DE0771H ADDI            -72,SP,SP
000000E0H  4BC23FD9H LDW             -20(s*,SP),RP
000000E4H  E840C000H BV              r0(RP)
000000E8H  08000240H OR              r0,r0,r0
000000ECH  08000240H OR              r0,r0,r0
```

Comments :

0000-0014	Export stub of the module
0018-001C	Entry of the module body
0020-002C	Assignation: p := P
0030-003C	Exit code of the body
0040-0048	Entry of the P procedure
004C-0050	Body of P
0054-005C	Exit of P

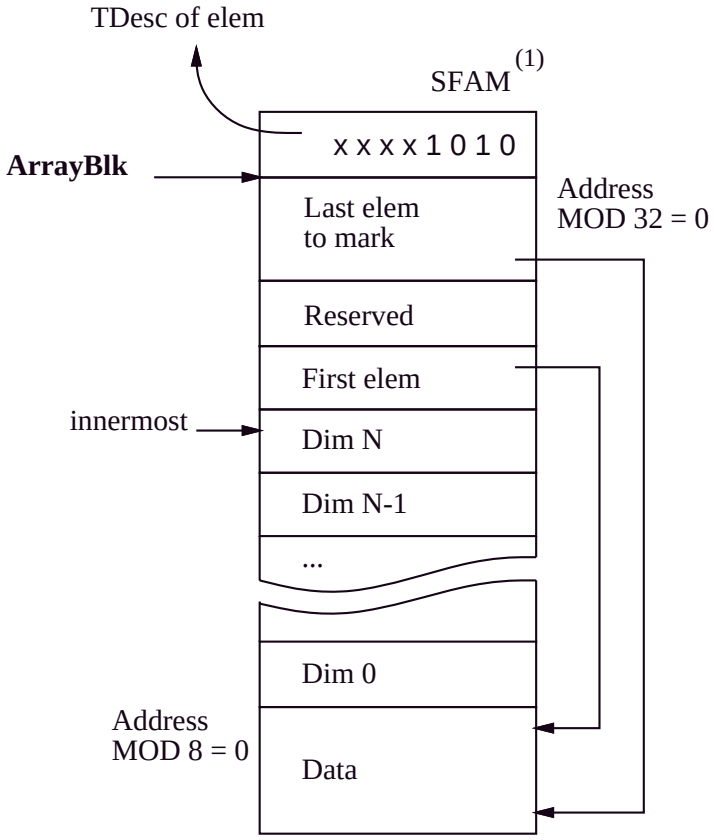
0060-0074 Export stub
0078-007C Entry of Q
0080-0088 Exit of Q

008C-0094 Entry of the Do procedure
0098-00A8 Direct call to P
00AC-00D4 Indirect call to P via p
00D8-00E8 Exit of Do

As can be seen at the address 00B8, the address of the called procedure passed to the import stub is stored in register R22. The following listing shows the standard HP-UX import stub also used by HP-Oberon.

```
        BB >=,n  R22,30,simple  If bit 30 is not set, then simple call
        DEPI     0,31,2,R22     Else, address is a shared library label
        LDW      4(0,R22),19    Load R19 (linkage table pointer)
        LDW      0(0,R22),R22   Load target
simple: LDSID    (0,R22), R1    Store space-ID of target...
        MTSP     R1,SR0         ...into SR0
        BE       0(0,R22)       Interspace branch
        STW      2,-24(0,SP)    Save the return pointer on the stack
```

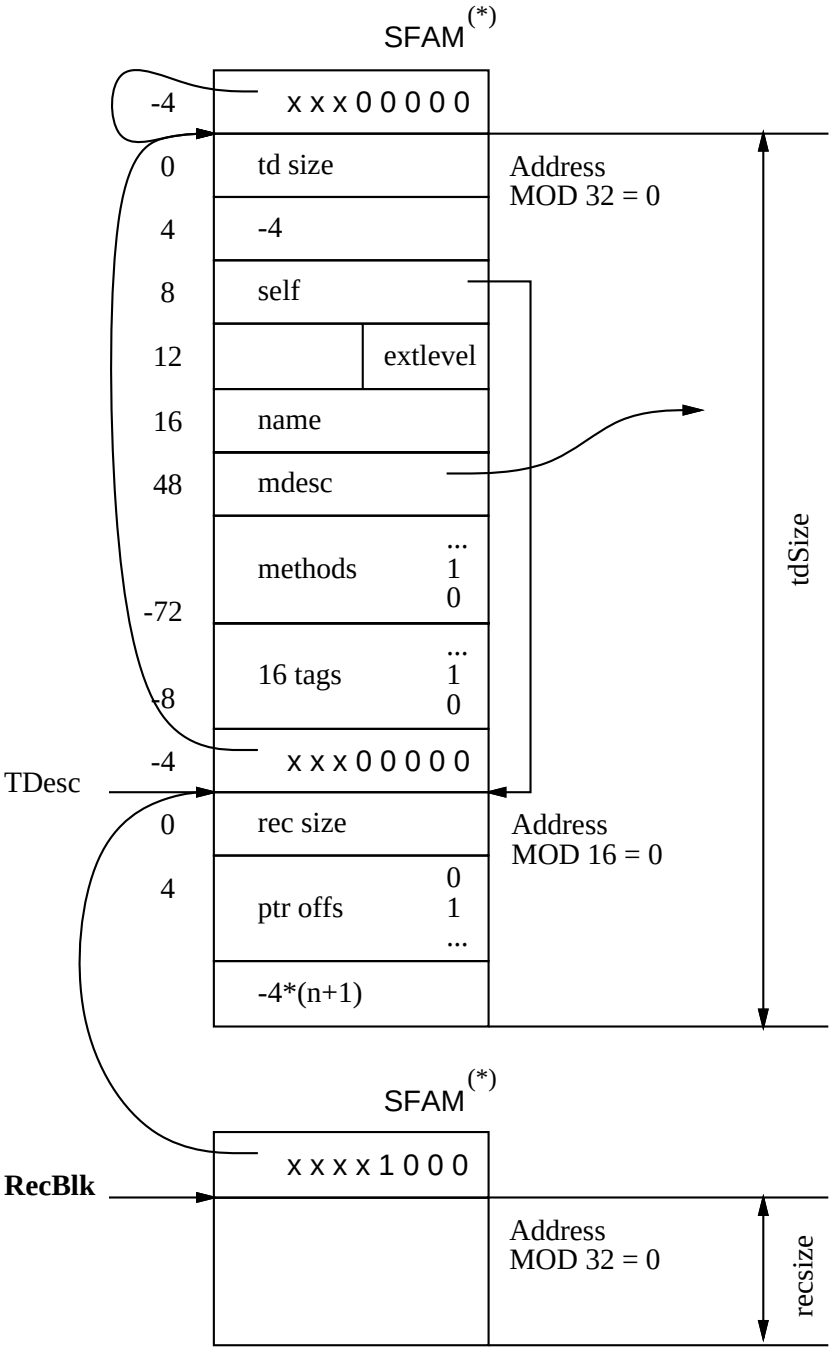
Listing 12: HP-UX import stub



Note 1: S(SubObject), F(Free), A(Array) and M(Marked) are the 4 last bits of the Tag.

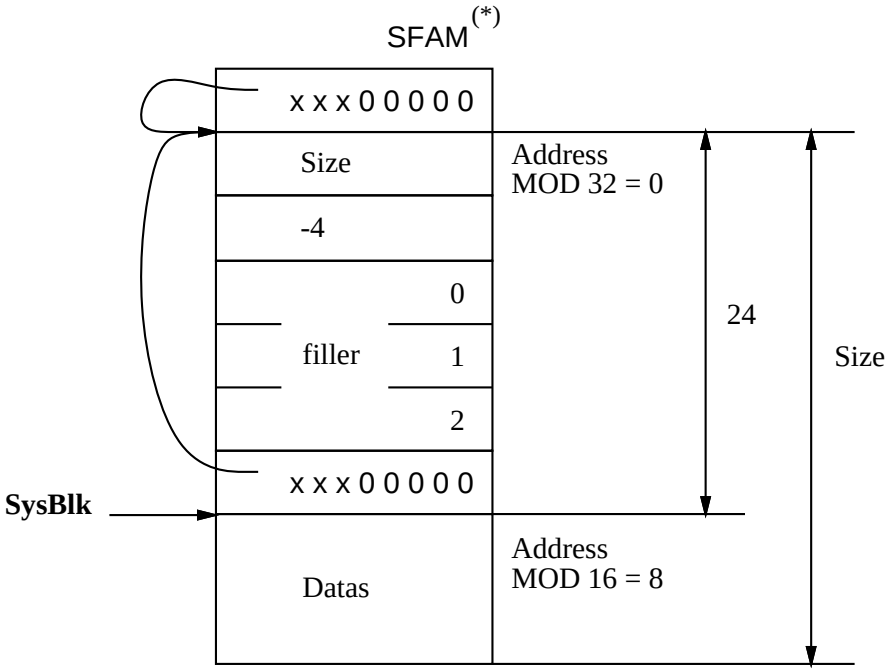
Note 2: SysBlk is used instead of ArrayBlk if there is no pointer in the element.

Figure 13: Array block



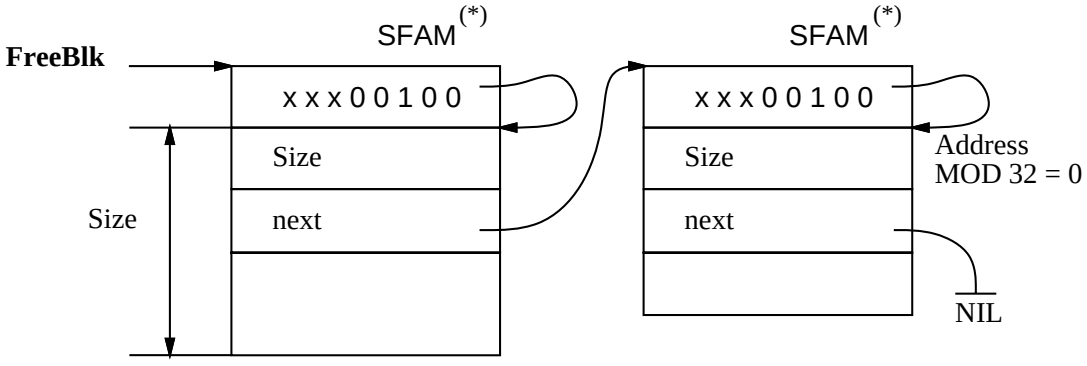
Note *: S(SubObject), F(Free), A(Array) and M(Marked) are the 4 last bits of the Tag.

Figure 14: Record block and Type Descriptor



Note *: S(SubObject), F(Free), A(Array) and M(Marked) are the 4 last bits of the Tag.

Figure 15: System block



(Size MOD 32) + 4 = 0

Note *: S(SubObject), F(Free), A(Array) and M(Marked) are the 4 last bits of the Tag.

Figure 16: Free block

Appendix C: HP Object File Format

Régis Crelier, Jacques Supcik, 5-May-93

```
ObjFile     = OFtag HeaderBlk ImpBlk ExpBlk CmdBlk PtrBlk ConstBlk
              CodeBlk UseBlk RefBlk.

OFtag       = 0F9X 37X.

HeaderBlk   = refsize:4 nofexp:2 noftdesc:2 nofcom:2 nofptr:2 nofimp
              syscalllink datalink datasize consize codesize modname.

ImpBlk      = 81X {name}.

ExpBlk      = 82X {EConst | EType | EVar | EProc | ECProc | EStruct | TDesc |
              LinkProc} 0X.
EConst      = 1X name fprint.
EType       = 2X name fprint.
EVar        = 3X name fprint offset.
EProc       = 4X name fprint entry.
ECProc      = 5X name fprint.
EStruct     = 6X name pbfprint pvfprint.
TDesc       = 8X (name | 0X pvfprint) link recsize
              ( -1 | basemod (name | 0X pvfprint))
              nofmth nofinhmth nofnewmth nofptr {mthno entry} {ptroff}.
LinkProc    = 9X entry link.

CmdBlk      = 83X {name entry}.

PtrBlk      = 84X {off}.

ConstBlk    = 87X {con1}.

CodeBlk     = 88X {instr:4}.

UseBlk      = 89X {{UConst | UType | UVar | UProc | UCProc | UpbStr |
              UpvStr | LinkTD} 0X}.
UConst      = 1X name fprint.
UType       = 2X name fprint.
UVar        = 3X name fprint link.
UProc       = 4X name fprint link.
```

```
UCProc   =  5X name fprint.
UpbStr   =  6X name pbfprint.
UpvStr   =  7X name pvfprint.
LinkTD   =  8X (name | 0X pvfprint) link.

RefBlk   =  8AX {0F8X procend savedr savedf frame callarea name {Mode
            Form adr name}}.

Mode     =  Var | VarPar.

Var      =  1X.
VarPar   =  3X.

Form     =  Byte | Bool | Char | SInt | Int | LInt | Real | LReal | Set | Pointer |
            String.

Byte     =  1X.
Bool     =  2X.
Char     =  3X.
SInt     =  4X.
Int      =  5X.
LInt     =  6X.
Real     =  7X.
LReal    =  8X.
Set      =  9X.
String   =  0AX.
Pointer  =  0DX.
```

Names are sequences of characters terminated by 0X. Lower case identifiers denote numbers. A digit appended to an identifier indicates the length of the number in bytes (LSByte first). Otherwise, the number is compressed into a variable number of bytes (LSByte first, base 128, cleared MSBit is stop bit). Sets (savedr and savedf) are coded like integers. Floating point numbers are in IEEE format (LSByte first).

Appendix D: Symbol File Format

Régis Crelier, 29-May-92

```
SymFile    =  0FBX Module {Object}.

Module     =  0 | negmno | MNAME name.

Constant   =  CHAR value:1
              |BOOL (FALSE | TRUE)
              |(SINT | INT | LINT | SET) value
              |REAL value:4
              |LREAL value:8
              |STRING name
              |NIL.

Object     =  Constant name
              |TYPE Struct
              |ALIAS Struct name
              |(RVAR | VAR) Struct name
              |(XPRO | IPRO) Signature name
              |CPRO Signature len {code:1} name.

Field      =  ((RFLD | FLD) Struct name | (HDPTR | HDPRO)) offset.

Method     =  (TPRO Signature name | HDTPRO) methno.

Struct     =  negref
              |STRUCT Module name [SYS value]
              (PTR Struct
              |ARR Struct nofElem
              |DARR Struct
              |REC Struct size align nofMeth {Field} {Method} END
              |PRO Signature).

Signature  =  Struct {(VALPAR | VARPAR) Struct offset name} END.
```

MNAME	=	16.	XPRO	=	31.	predefined refs:	
not used		17.	IPRO	=	32.		
END	=	18.	CPRO	=	33.	BYTE	= 1.
TYPE	=	19.	STRUCT	=	34.	BOOL	= 2.
ALIAS	=	20.	SYS	=	35.	CHAR	= 3.
VAR	=	21.	PTR	=	36.	SINT	= 4.
RVAR	=	22.	ARR	=	37.	INT	= 5.
VALPAR	=	23.	DARR	=	38.	LINT	= 6.
VARPAR	=	24.	REC	=	39.	REAL	= 7.
FLD	=	25.	PRO	=	40.	LREAL	= 8.
RFLD	=	26.				SET	= 9.
HDPTR	=	27.				STRING	= 10.
HDPRO	=	28.	boolean constants:			NIL	= 11.
TPRO	=	29.	FALSE	=	0X.	NOTYP	= 12.
HDTPRO	=	30.	TRUE	=	1X.	POINTER	= 13.

Names are sequences of characters terminated by 0X. Lower case identifiers denote numbers. A digit appended to an identifier indicates the length of the number in bytes (LSByte first). Otherwise, the number is compressed into a variable number of bytes (LSByte first, base 128, cleared MSBit is stop bit). Sets are coded like integers. Floating point numbers are in IEEE format (LSByte first).

Appendix E: Boot-file Format

All module names must be listed and topologically sorted. If the optional heap address is present, no relocation information is generated.

Boot File Format:

```
heapAdr4
heapSize4
GCstart4
{adr4 len4 {int4}}(* len4/4 times int4 *)
entryAdr4 0X 0X 0X 0X
nofPtr {adr}(* nofPtr times adr, etc... *)
nofWord {adr}
nofProc {adr}
nofBl {adr}
nofMilli {adr}
dlsymAdr
```

All numbers in the relocation information part are in compact format. adr*4 is an offset from heapAdr4.