OSSI
A Portable Operating System Interface and Utility Library for Modula–2

Technical Report 86–005

February, 1986

E. Biagioni, G. Heiser, K. Hinrichs, C. Muller

The University of North Carolina at Chapel Hill
Department of Computer Science
New West Hall 035 A
Chapel Hill, N.C. 27514

Abstract

The absence of a standard layer between operating systems and Modula-2 programs makes most software written in Modula-2 non-portable. We propose **OSSI,** a set of library modules which hides the machine-dependent details from Modula-2 application programs. OSSI, for **O**perating **S**ystem **S**tandard **I**nterface, has been implemented on several computers and operating systems, and allows us to port large Modula-2 programs without any changes to their source code. OSSI standardizes I/O operations, memory management, and utilities such as string handlers and large-set operations.

Keywords

modules and interfaces, software libraries, programming environments

current addresses of authors:

E. Biagioni, K. Hinrichs
Department of Computer Science, University of North Carolina, Chapel Hill, NC 27514, USA

G. Heiser
Institut für Informatik, ETH, CH-8092 Zürich, Switzerland

C. Muller
BBC Brown Boveri & Co. Ltd., Research Center, Dept. KLR-CI, CH-5405 Baden, Switzerland

1 Introduction

The main emphasis in the design of Modula-2 has been put on ease of implementation and efficiency. This results in a compact language that is easy to master and can be implemented on practically any computer. The disadvantage of this approach is that essential features - such as I/O and memory management - have been left out or are not sufficiently defined. Hence the interface to the host system differs between various implementations of the language. This poses serious problems when trying to write portable software, as can be seen in Example 1.

Since the language only supports a few commonly required operations, most Modula-2 programmers have to implement their own libraries containing the functions they need most frequently. This results in unnecessary duplication of effort and a proliferation of very specialized, non-standard libraries. Further complications arise due to small but significant differences among existing compilers. For instance the type `WORD` is sometimes defined according to the original definition of the language - `WORD` is compatible with types `CARDINAL`, `INTEGER` and `POINTER` - and sometimes according to the revised definition - `WORD` is the smallest addressable unit of storage. Both problems, i. e. the lack of functionality and the differences among compilers, can be solved by providing a generally available standard library. It is one of the strengths of Modula-2 that standard libraries can be easily and efficiently incorporated into existing Modula-2 environments.

The need for a standard, machine-independent library was made obvious to us when porting the interactive system XS-2[1] from the Lilith personal computer[2] to a VAX™/VMS system (work done by H. Sugaya of the Brown Boveri Research Center and one of the authors). The compilers on both computers were derived from the same PDP-11 compiler. However, the libraries supplied with the compilers were different, incompatible and system dependent. In each case they mirrored the capabilities provided by the underlying operating system. In the XS-2 project, the portability of input and output had been carefully analyzed; for these, we defined a machine-independent interface. XS-2 however used other features of the Lilith library including runtime loading of programs, bit sets defining cursor patterns, and polling to track cursor movements. It took over four man-months to produce a running version of XS-2 on the VAX™. Every time a new version was installed on the VAX™, it took several days to adapt the non-portable features of the Lilith library which were used throughout the code.

As a solution to this kind of problems we specified **OSSI** - **O**perating **S**ystem **S**tandard **I**nterface. OSSI is a set of standard modules to be used by portable programs; these modules define a virtual machine and supply a set of commonly needed standard operations. This minimal programming environment suffices for a large class of applications. By adding further modules, it can be extended to support any kind of portable application. The programmer is assured that the features

provided by this library are portable. OSSI provides program portability, not data portability.

In section 2 we discuss the basic requirements for a standard Modula-2 library. Section 3 presents the ideas behind our concept of a standard library and evaluates other proposed standard libraries. In section 4 we describe our implementation in more detail. In section 5 we relate our experience with OSSI.

2 Requirements for an operating system interface and standard library

A standard programming support library consists of several building blocks. First there must be some low-level support to ensure the programmer *can* write applications independent of hardware and operating system characteristics. Typical operating system services such as input and output must also be included. A third building block is a library of general utilities. Finally an optional part should support the use of special hardware and software that may not be available on all systems.

2.1 Low-level system interface

A standard library must provide a shell that insulates applications from low-level hardware and operating system features. This includes constants defining the storage requirement of simple types. Some of these constants are often required in type declarations, precluding usage of the predefined function `SYSTEM.TSIZE`. The ranges of standard types, e.g. `CARDINAL` or `REAL`, should also be defined. This eases type declarations and the writing of programs that need to know parameters such as the relative precision of reals or the number of bits available in a bit set.

Data base systems and other applications often need to pack and unpack data to optimize storage utilization. This must be done differently for different types; while it is sufficient to drop the most significant bytes when packing a `CARDINAL` value, the sign bit has to be treated separately in the case of `INTEGER` values. Data packing is not supported by the language and must therefore be incorporated into the library.

Dynamic memory allocation is not sufficiently specified in Modula-2. Most environments provide the procedures `ALLOCATE` and `DEALLOCATE`, but their implementations differ in the way they respond to memory requests that cannot be satisfied. Some will do nothing, others will return a `NIL` pointer, some will simply crash.

Furthermore the size of the allocation unit is highly system-dependent; it can be

anything from a single bit to a number of memory words. This implies that there is no general way to compute the size of an object, as is often necessary in constant declarations and in storage-management applications. To allocate memory for a string of say 100 characters, it is necessary to determine the number of storage units corresponding to 100 bytes.

2.2 Standard input/output interface

It is clearly necessary to have a properly defined interface for terminal input and output. The Modula-2 report does not include such an interface, and the modules `Terminal` and `InOut` as described in Wirth[3] are not available in most Modula-2 environments or are incompatible in their specifications.

The situation is worse in the case of file I/O, as illustrated by Example 1. Portable application programs must be based on a consistent, *hardware-* and *operating system-independent* view of secondary storage. The I/O interface should provide simple and convenient sequential file operations, sufficient for a large class of applications. More sophisticated programs require record-oriented random access files. Access to the underlying file management system is also necessary for such operations as rename, delete and file search. Finally, programs that implement their own disk storage management, e.g. database systems, must be able to build their own access structures and make full use of the physical properties of disk storage such as block structure and asynchronous random access.

A standard library should therefore provide several entry points to the I/O system, differing in the level of abstraction they represent and the convenience of their usage. To ensure portability, the library modules must not depend on a particular operating system, but implement the underlying abstract model, e.g. the stream concept.

2.3 Standard utilities

Many programming languages provide built-in string operations. Modula-2 does not, so the standard library should offer the necessary functionality. The basic string operations are:
- determining the length of a string;
- copying and concatenating strings;
- inserting, deleting and extracting substrings;
- comparing strings;
- searching for an occurrence of a string within another one.

Conversions between numbers and strings are useful for formatted input and output of numbers and should also be included.

A library of mathematical functions standardizing the computations with the Modula-2 numerical types should include:

- functions returning the largest or smallest of two or more numbers;
- square root;
- exponential;
- function for real numbers raised to integral powers;
- logarithms;
- trigonometric and hyperbolic functions and their inverses;
- random number generators.

Most implementations of the language only support sets up to a cardinality equal to the computer's word size in bits. This is unsatisfactory for two reasons: the maximum set size is implementation-dependent, and in general too small for most applications. To make full use of the powerful set concept without modifying the language, a library module should define a set type of large cardinality which supports all the usual set operations.

2.4 Support for advanced I/O devices

In recent years powerful new devices such as graphic displays and analog input devices have come into widespread use. While keyboard and typewriter-like devices are supported by most programming languages, these new interactive features are not yet integrated into most programming environments.

A comprehensive Modula-2 library should therefore contain optional modules implementing concepts such as screen windows or pointing devices. The latter can be used for input selections via menus and for manipulating objects in highly interactive application programs. Many physical devices could play the role of a pointing input device, for instance mouse, trackball, lightpen or even cursor keys. Because of this variety it is necessary to give the programmer a unified view of these concepts and to implement them with device-independent modules.

2.5 Error handling

An error condition occurs when an operation cannot be performed as intended due to either lack of resources or invalid inputs. The caller of a library procedure must be able to detect such errors. This should be achieved by the called procedure signalling an error to the caller. There are cases, however, where this is too inefficient, e.g. in time critical procedures like mathematical functions. If such an operation cannot be completed with the correct result, the calling program must be able to check the validity of the call beforehand.

3 The philosophy behind OSSI

OSSI consists of a set of modules created to meet the above-mentioned requirements. Its design is based on several fundamental ideas:

Consistent programmer interface
Convenience of usage as well as clarity and reliability of the applications are enhanced by clear and consistent definitions. Names are chosen in a systematic manner, errors are treated systematically via result parameters, and side effects are avoided.

Independence
The modules are complete in the sense that they can be used independently of each other. Functions that operate on different types of objects, e. g. strings or sets, are provided by independent modules.

Stability
Great care was taken to prevent the library procedures from crashing due to user program bugs, lack of system resources or meaningless input parameters. The data packing routines are excepted from this rule, since they only operate on resources that can be controlled by the user program, and they need to be extremely efficient. The same holds for the mathematical function library, for which efficiency precludes checking the validity of the input on each call.

Efficiency and minimality
The procedures are defined in such a way that they can make efficient use of system resources on almost any system. The unavoidable penalty for using the standard library, in terms of memory usage and execution time, has been kept to a minimum. This was made possible by providing minimal but comprehensive services based on simple concepts, rather than building a huge system that does everything, as for example GKS, the graphical kernel system standard.

Extensibility
The system is open: it can easily be expanded to meet requirements posed by new hardware or special applications, e. g. robot control or sound generation.

Portability
The library has been implemented to make portability easy. System-dependent constructs, e. g. operating system services, are used only where necessary. This allows re-use of large portions of the library without changes when transporting OSSI to another computer or operating system. The remainder can be adapted with little effort, so that an experienced programmer familiar with the target environment can port OSSI - and test it - in a week's time. Testing and debugging efforts are greatly reduced by providing system-independent test programs for many of the OSSI modules. The test programs perform mostly automated checks on all procedures exported by a specific OSSI module. The tests have been carefully selected to check a wide range of typical and borderline cases. If a test fails, the test program gives an indication of where to look for the problem. With these tools, the trustworthiness of a new implementation of OSSI can be established quickly, typically in less than an hour.

Other library proposals

Other proposals for Modula-2 standard libraries can be found in Modus[4] and in Craig et al.[5]. The goals of the Modus proposal are, among others, to "allow one to write portable Modula-2 programs", "allow one to describe algorithms which refer to a standard environment", "do not specify the operating system...". Craig et. al. specify their goals as "(1) provide enough facilities to write general purpose applications programs which are intended to be portable, (2) allow efficient implementation on many different systems, (3) allow expansion of the kernel as needed".

The specification of OSSI includes all these features and goes beyond them to specify a virtual machine. This allows OSSI to be considerably more uniform in its appearance to the programmer: all the modules share the same naming conventions and error handling is uniform. Each module corresponds to a single abstract concept.

This is not the case with the other libraries. In the Modus proposal for instance, separate modules provide for text and binary I/O to files; however, they use a single type `File` defined in a third module. Hence, several modules must be imported to perform conceptually related operations. Files can be manipulated by operations provided by several modules, and these can influence each other via side effects. This implies that misuse of files cannot be detected, leading to an unsafe system. For instance, writing binary information onto a text file can make the file unusable for text editors.

Craig et al. do not distinguish between text files and binary files: "FileIO provides procedures for reading, writing, and positioning files. These procedures are patterned after the best of the Ada KAPSE packages, UNIX™ file functions and UCSD Pascal I/O procedures." This is contrary to the OSSI philosophy of supplying operating system independent concepts rather than accumulating "nice" features from different operating system. Not distinguishing between text and binary files works fine in the UNIX™ world, on other systems, e.g. VAX™ /VMS, it is impossible to implement files that can store any kind of data while still being compatible with standard text editors.

Probably the least portable feature found in Modus is the module `Program`. This module cannot be implemented on some virtual memory systems (for instance VAX™/VMS). Moreover both libraries fail to provide many of the features we consider essential (cf. section 2). These include a complete set of hardware or compiler dependent constants, packing and consistent error handling.

4 Description of OSSI modules

We now describe the basic features of our library. Currently OSSI consists of roughly a dozen modules, each covering one logical domain. The structure of the OSSI modules is shown in Figure 1. The names of all modules start with the letters "SI". This is to avoid name clashes with system or library modules of particular Modula-2 environments. The first six letters of the module names are all unique, to avoid problems with operating systems that only allow relatively short file names.

Error handling is uniform across the library: every procedure which may fail due to conditions not under control of the calling program returns a result variable. This variable is of a global result type. Its value reflects the completion status of the operation. If it is equal to a global constant `SIDone`, the operation has completed successfully; in all other cases an error has occurred. The mapping from errors to error results is system-dependent; a global procedure `SIMessage` converts error results into message strings. This allows easy detection of errors and provides a convenient mechanism for reporting them to the user. In general one cannot expect that each possible error condition can be mapped onto a predefined set of errors, and many operating systems do not allow user programs to distinguish between different errors. OSSI's concept of error handling is sufficient because a programmer can always test all error conditions for which recovery is possible, i.e. testing the existence of a file before trying to open it.

4.1 Low-level system interface

The low level interface consists of the modules `SISystem`, `SIMemory` and `SIPacking`.

The module `SISystem` exports generally useful, system-dependent constants, types and functions. These include constants for defining the range of the standard types and their storage requirements as postulated in section 2.1. Storage requirements are always given in units of bytes rather than the system-dependent storage units. Further constants define such compiler-dependent features as the size of a storage unit and the maximum range allowed for an array index. Exported functions include `IntTrunc` and `IntFloat` as well as `Cap` and `UnCap`. The former two are the `INTEGER` equivalents of the predefined functions `TRUNC` and `FLOAT`, which are defined for `CARDINAL` values only. `Cap` is like the predefined function `CAP`, but defined on the whole range of the type `CHAR`, not only for letters. `UnCap` performs the inverse operation. Finally, the type `BYTE` can be used in place of the implementation-dependent type `WORD`.

`SIMemory` provides memory allocation and deallocation procedures. It exports the procedures `ALLOCATE` and `DEALLOCATE` which most compilers substitute for the predefined macros `NEW` and `DISPOSE`. `ALLOCATE` returns a `NIL`

pointer if a memory request cannot be satisfied, so the client program can recover from a heap overflow. Both procedures use the same storage unit as the standard procedures `SIZE` and `TSIZE`.

In addition `SIMemory` provides the procedures `AllocateBytes`, `DeallocateBytes` and `PartialDeallocateBytes`. The first two are like `ALLOCATE` and `DEALLOCATE`, except for the storage unit, which is the byte. This is compatible with the size constants exported by the module `SISystem`. The procedure `PartialDeallocateBytes` lets the programmer return to the system a part of a storage segment - previously allocated by a call to `AllocateBytes` - without having to perform system-dependent address computations.

The module `SIPacking` satisfies the requirements for data packing and standard type conversions stated in section 2.1. It exports procedures which allow packing of `CARDINAL` and `INTEGER` values, and the storage of any kind of structured or unstructured data into an `ARRAY OF BYTE`.

4.2 Input/output interface

Standard terminal I/O is done using the module `SITerminal`. It exports procedures for reading single characters from the keyboard with or without echo on the screen, and for writing single characters to the screen. In addition `SITerminal` also supports reading and writing of strings. Another procedure tells the read routine that the last character returned from the keyboard should be buffered and returned again on the next read operation. Finally the function `Keypress` reports whether a key has been pressed, i.e. whether a character is available for reading. The input/output procedures of the module `SITerminal` are all exported as procedure variables. This lets the user redefine the procedures to operate on alternate devices or according to different specifications. The default procedures may support system conventions for input/output redirection, to improve integration of the Modula-2 programs in a given environment.

The modules `SIStreams`, `SIRandomIO` and `SIBlockIO` represent conceptually different ways of accessing secondary storage. `SIStreams` is a convenient interface to sequential files, supporting line-oriented text files and byte-oriented binary files (see Example 2). `SIRandomIO` implements record-oriented random access files, allowing the use of more complex but more useful file system capabilities. `SIBlockIO` defines a virtual disk, allowing block-oriented, asynchronous random access to secondary storage. It is tailored to the requirements of database systems, which implement their own disk access structures. The overriding consideration is minimization of the number of physical I/O operations needed to perform block reads and writes.

4.3 Utilities

The utility library consists of the modules `SIStrings`, `SIFiles`, `SIConversions`, `SINumberIO`, `SISets` and `SIMathLib`.

`SIStrings` provides all the basic string operations given in section 2.3.

`SIFiles` views files as atomic units and gives access to the file management of the underlying operating system. It allows searching, deleting, renaming and copying of files and provides a portable means to convert logical into physical filenames.

`SIConversions` allows conversions between strings and numbers, that is, between binary and human-readable representations. Result parameters give the user program the opportunity to recover from invalid inputs. When converting strings to numeric types, the caller can distinguish whether the number exceeds the range of the particular type or whether the string is syntactically incorrect. With the inverse conversions, the user has extensive control using well-defined formatting parameters. An additional procedure determines whether or not a particular `REAL` can be represented as a `CARDINAL` or `INTEGER`. This is done by checking whether or not it has an integral value (within the numerical accuracy of type `REAL`) and lies within the range of the respective types. `SINumberIO` allows formatted I/O of numeric types. It essentially combines the functions of the modules `SIConversions` and `SITerminal`.

`SISets` exports operations on large sets of cardinals. This allows for instance sets of character codes. All the operations available for standard `SET` types are available for these large sets. In addition there are procedures which return the complement and the cardinality (the number of elements) of a set.

`SIMathLib` is a library of mathematical functions which satisfies the requirements stated in section 2.3.

4.4 Optional modules

Advanced hardware and software which is not available everywhere is supported by the optional modules `SIFullScreen`, `SIWindows`, `SITextWindows` and `SIGraphicWindows`. Other modules support special features such as menus and cursors. `SIFullScreen` allows random access to an alphanumeric display such as provided by most terminals. `SIWindows` supports the creation and management of windows on a raster display. The management functions include moving and resizing windows and control of window visibility. The windows are refreshed automatically after these operations. `SITextWindows` and `SIGraphicWindows` are built on top of `SIWindows` and provide the functions needed for text and graphic input and output, respectively.

5 Experience with OSSI

5.1 The virtual machine OSSI

OSSI is an improvement over libraries supplied with current Modula-2 compilers, because those libraries often make use of operating system dependent details. OSSI was developed on the basis of our experience porting software across a variety of computers and operating systems, for instance Lilith, VAX™, PDP-11, IBM-PC, Apple Macintosh™, Sun and other 68000-based systems; under VMS, UNIX™, MS-DOS. This experience guided us in deciding what features are truly common to most systems and what features, possibly useful, are too specific to be included in the library. The resulting set of modules defines a virtual machine that can be realized on most physical machines in use today.

A programmer planning to write a portable application must keep in mind that the language Modula-2 offers many features that lead to non-portable programs. The complete program design should therefore be based on a virtual machine defined by the modules of a standard library. OSSI provides enough functionality for writing general-purpose portable application programs. Even large and complex software packages from the domain of systems programming can be built on top of OSSI, because the virtual machine can be accessed at a level close to the hardware (modules `SISystem`, `SIMemory` and `SIPacking`). The next section gives an example of such a software package.

If an application needs facilities that are too specialized to be included in the library, one should proceed as follows:
- set up precise specifications of the non-standard requirements;
- use the module concept of Modula-2 to concentrate the non-portable facilities in a separate special module;
- build the application on top of the virtual machine which is now extended by this special module.

If the application has to be installed on a different computer system, only the special module needs to be adapted. A programmer familiar with the target machine can adapt the module without knowing anything about the application, because the definition part of the special module provides exact specifications of what the module is supposed to do.

5.2 Software packages based on OSSI

The **Smart Data Interaction** package[6] is an interactive software tool for deductive data manipulation. It allows
- efficient access to data on secondary storage;
- deductive queries and data manipulations;
- effective human-computer interaction.

The package is built on top of the virtual machine OSSI and runs on several different computer systems. It is based on three software building blocks that can be used alone or in various combinations:

The *grid file* is a data structure for secondary storage designed to efficiently handle large amounts of multidimensional data. Efficient access to secondary storage is achieved by minimizing the number of disk read and write operations required to accomplish a given task. The *grid file system*[7] is implemented in Modula-2 and based on OSSI. Since the grid file views secondary storage as a randomly accessible, block-structured device, it uses `SIBlockIO` to access the disk.

Modula--Prolog[8] is an interpreter for the logic programming language *Prolog*. It can interact in many ways with other Modula-2 programs, e.g. it provides tools for including new built-in predicates (written in Modula-2) in the Prolog language, or it can be used by Modula-2 programs as a deductive problem solving component in the background. It is completely based on OSSI, and within the Smart Data Interaction package it plays the role of a deductive query and manipulation language.

Easy[9] is a user interface for interactive applications. The user sees structured commands and data, a command history and a set of universal commands applicable to both commands and data for all applications that use Easy. The application receives commands with their appropriate parameters, and can display data and feedback in a window reserved for it. Since Easy is meant to work satisfactorily using only a minimal hardware configuration, it uses `SIFullScreen` for screen output and positioning.

Current software development using OSSI includes the kernel of an extended relational data base system for storing geometric objects. It incorporates the grid file system and uses only OSSI modules otherwise.

5.3 Portability of OSSI

OSSI was developed between the fall of 1984 and fall of 1985, with minor revisions since. The size of the source files of the OSSI kernel (definition and implementation modules, including all comments) adds up to some 330 KByte.

The library was developed on the Lilith and was ported to a VAX™ under VMS during the early stages of development. Since its completion versions have been produced for a variety of computer systems, including IBM-PC under MS-DOS, Macintosh™ using the MacLogimo compiler and a general UNIX™ version. The implementation of the UNIX™ version took one of the authors, already familiar with the system, just a week, testing included. The Macintosh™ version was developed by a student (C. Gianotti) with no prior knowledge of either OSSI or the Macintosh™ as part of a two month project. This made all the OSSI based packages

immediately available on the new computer systems. The reader is encouraged to compare these results with the time and effort usually required for porting large systems, as shown for example in the introduction.

6 Concluding remarks

It is clearly desirable to have one generally available standard library for Modula-2. In order for any library to be accepted as a standard, it must be based on a consistent and clean design. The library modules should be implementations of abstract concepts. Most present libraries suffer from the lack of a consistent design philosophy and appear to be collections of modules which simply reflect the underlying operating system. The examples given show OSSI to be well suited for large and complex system programs, as well as application programs. Unlike other proposals, OSSI consists of a kernel and an optional part. The kernel reflects the features generally provided by operating systems, the optional part interfaces to hardware that is not available on all systems. OSSI is easily ported and provides most of the commonly-used features of present operating systems, in addition to many common utility routines. As such, OSSI is a valid candidate for a standard Modula-2 library.

References

1. J. Stelovsky, J. Nievergelt, H. Sugaya, and E. S. Biagioni, "Can an Operating System Support Consistent User Dialogs? - Experience with the Prototype XS-2," Proc. 1985 ACM Annual Conf., 1985, Denver CO, pp. 476-483.

2. N. Wirth, "The personal computer Lilith," Proc. 5th Intern. Conf. on Software Engineering, IEEE Computer Society Press, 1981, pp. 2 - 15.

3. N. Wirth, "Programming in Modula-2," 3rd ed., Springer Verlag, Berlin, Heidelberg, New York, Tokyo, 1985.

4. "Modula-2 Standard Library Definition Modules," Modula-2 News, Issue # 1, January 1985, pp. 22 - 34. Modus, the Modula-2 Users Association, c/o Pacific Systems, PO Box 51778, Palo Alto, CA 94303, USA.

5. J. M. Craig, J. M. Martel, and K. B. Richan, "Design of a Modula-2 Standard Library," Journal of Pascal, Ada and Modula-2, Vol. 4, No. 2, 1985, pp. 33 - 46.

6. E. S. Biagioni, K. Hinrichs, C. Muller, and J. Nievergelt, "Interactive deductive data management - the Smart Data Interaction package," Wissensbasierte Systeme, GI - Kongress 1985, München, Informatik - Fachbericht 112, Springer Verlag, Berlin, Heidelberg, New York, Tokyo, 1985, pp. 208 - 220.

7. K. Hinrichs, "Implementation of the grid file: design concepts and experience," BIT, Vol. 25, 1985, pp. 569 - 592.

8. C. Muller, "Modula--Prolog," Report No. 63, Institut für Informatik, ETH, CH-8092 Zürich, Switzerland, July 1985.

9. E. S. Biagioni, "Easy - a front end for interactive programs," diploma thesis, Institut für Informatik, ETH, CH-8092 Zürich, Switzerland, February 1985.

Example 1: PDP-11 and Lilith file interfaces

Wirth[3] describes two different Modula-2 interfaces to file systems. One provides access to the DEC PDP-11/RT-11 system and consists of the modules `Files` and `Streams`, the other is the module `FileSystem` of the Medos-2 operating system for the personal computer Lilith[2]. The module `Files` mirrors the structure of the RT-11 file system and is restricted to block-oriented input/output. `Streams` is a sequential file interface based on the module `Files`. A user of `Streams` must explicitely use procedures exported by `Files`, e. g. `Create` or `Lookup`. The Medos-2 module `FileSystem`, in contrast, provides the functionality of both `Streams` and `Files`; its syntax and semantics differ from those of the RT-11 implementation.
In order to open one file for input and one for output, the code for RT-11 might look like

```
VAR f1, f2: FILE;    (* FILE = CARDINAL *)
    s1, s2: STREAM;
    reply:  INTEGER;
   . . . .
f1 := 1; f2 := 2;
Lookup (f1, inName, reply);
Create (f2, outName, reply);
Connect (s1, f1, TRUE); Connect (s2, f2, TRUE);
```

The same is achieved under Medos-2 using the following statements:

```
VAR f1, f2: File;     (* File = RECORD ... END; *)
   . . . .
Lookup (f1, inName, FALSE);     (* open existing file *)
Lookup (f2, outName, TRUE);     (* create new file *)
```

In the first case `f1` and `f2` denote RT-11 channel numbers, while `s1` and `s2` are variables of an abstract type `STREAM` associated with the channels. A result status is returned via the parameter `reply`. In the Lilith case, `f1` and `f2` are variables of a structured type `File`, which contains a field that represents the result status of the last operation on the file. This record also contains fields that are highly specific to the Lilith file system, such as <address, boolean> pairs that are used as byte pointers, since Lilith can only directly address two-byte words; similarly, pairs of cardinals are used for file read/write positions since a single 16-bit cardinal is not sufficient to define a file position. It is obvious that neither module is system-independent, nor easily portable. Even though the above two program fragments are equivalent, it is clear that neither would compile in the other environment. This is just a simple example to show that programs which use different interfaces are generally not portable.

Example 2: The OSSI module SIStreams

As an example of an OSSI module we give the definition module `SIStreams`. It implements the abstract concept *stream;* it is a simple interface to sequential files. The procedure `ConnectStream` connects a stream to a file on secondary storage. This stream can then be used for input or for output to the file. Only sequential file access is supported and a given stream can only be used either for input or for output. We also distinguish between text streams and binary streams. The former are compatible with standard text editors available on the host system but may contain padding or formatting characters required by the system. Table 1 shows the procedure calls that are valid for a stream variable of a given mode and type.

StreamType	StreamMode	valid procedure calls
textStream	inputStream	SRead, SReadString, EndOfStream
textStream	outputStream	SWrite, SWriteString, SWriteLine
binaryStream	inputStream	SReadByte, SReadRecord, EndOfStream
binaryStream	outputStream	SWriteByte, SWriteRecord

Table 1

In addition to the above, a call to `DisconnectStream` is always possible. It disconnects the stream from the physical file and closes the latter.
Each procedure returns a result status through the result parameter. The value returned in the result parameter will be `SISystem.SIDone` if and only if the operation was successful. Otherwise result has a different value, which can be converted to a textual description of the error by a call to `SISystem.SIMessage`.

The definition module, with all comments edited out, is:

```
DEFINITION MODULE SIStreams;

FROM SYSTEM IMPORT WORD;
FROM SISystem IMPORT BYTE, SIResult;

TYPE StreamType = (textStream, binaryStream);
     StreamMode = (inputStream, outputStream);
     Stream;

PROCEDURE ConnectStream (VAR s: Stream;
                         VAR fileName: ARRAY OF CHAR;
                         type: StreamType; mode: StreamMode;
                         VAR result: SIResult);

PROCEDURE DisconnectStream (VAR s: Stream; VAR result: SIResult);

PROCEDURE EndOfStream (s: Stream; VAR eos: BOOLEAN;
                       VAR result: SIResult);
```

```
PROCEDURE SRead (s: Stream; VAR ch: CHAR; VAR result: SIResult);

PROCEDURE SReadString (s: Stream; VAR string: ARRAY OF CHAR;
                       VAR result: SIResult);

PROCEDURE SReadRecord (s: Stream; VAR record: ARRAY OF WORD;
                       VAR result: SIResult);

PROCEDURE SReadByte (s: Stream; VAR byte: BYTE;
                     VAR result: SIResult);

PROCEDURE SWrite (s: Stream; ch: CHAR; VAR result: SIResult);

PROCEDURE SWriteString (s: Stream; VAR string: ARRAY OF CHAR;
                        VAR result: SIResult);

PROCEDURE SWriteLine (s: Stream; VAR result: SIResult);

PROCEDURE SWriteRecord (s: Stream; VAR record: ARRAY OF WORD;
                        VAR result: SIResult);

PROCEDURE SWriteByte (s: Stream; byte: BYTE; VAR result: SIResult);

END SIStreams.
```

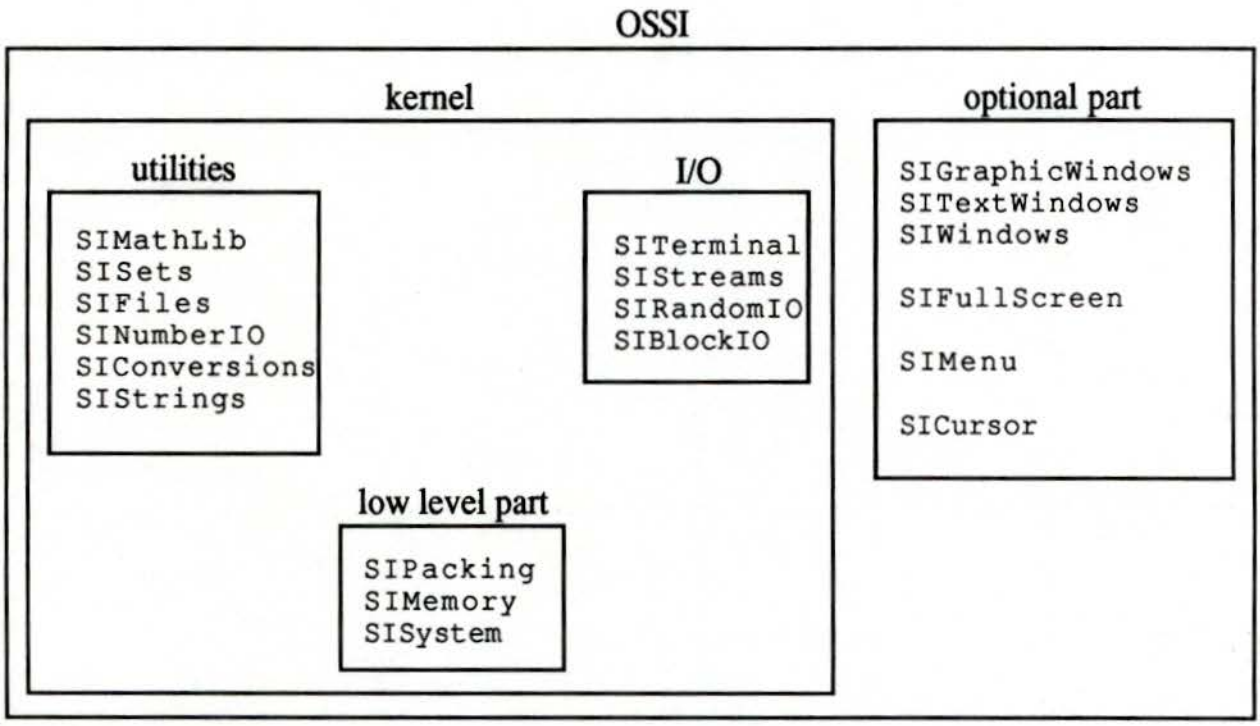

Fig. 1: module structure of the OSSI library