$\mathcal{A}_2$ Programming Quickstart Guide

Felix Friedrich, Ulrike Glavitsch, Florian Negele, Sven Stauber

March 12, 2010

This is a brief introduction to the programming language Active Oberon and the operating system $\mathcal{A}_2$. This introduction does neither replace a language report nor is it an introduction to programming. It rather should provide quick access to the $\mathcal{A}_2$ system by ways of comments and examples. It has been prepared for the Operating System Course, Spring Semester 2009, at ETH Zurich as accompanying material to the practical exercises.

Contents

1 The $\mathcal{A}_2$ Operating System

$\mathcal{A}_2$ is a modern, multi-processor operating system in the tradition of the ETH operating systems Medos, Oberon and Aos. It is written in Active Oberon with only very few exceptions: little parts of the kernel and some low-level optimized routines are implemented in assembler. $\mathcal{A}_2$ is composed of light-weight processes that are supported by *Active Objects* in the language Active Oberon. Active Oberon is a highly readable, abstract programming language with built-in concurrency support. Its abstraction protects programmers from artificial technical complexities while system-near programming is still possible by ways of a special module named ‘SYSTEM’.

1.1 Modules and Commands

Active Oberon is a modular programming language that is supported by a dynamic loading concept in the operating system $\mathcal{A}_2$. The role of programs and applications is played by *modules*. Instead of loading and executing a 'program', *modules* are loaded and *commands* are executed. Once a module is loaded, it stays loaded until it is explicitly unloaded or until the system goes down. Before we give more details on modules and commands, we show an example of a very simple module.

```
MODULE Hello;

IMPORT Commands, KernelLog;

   (* a command taking a context parameter *)
   PROCEDURE World*(context: Commands.Context);
   BEGIN
      context.out.String("Hello World");
      context.out.Ln
   END World;

   (* a command without parameters *)
   PROCEDURE InnerWorld*;
   BEGIN
      KernelLog.String("Hello Kernel World");
      KernelLog.Ln;
   END InnerWorld;

END Hello.
```

Fig. 1: A Simple Oberon Module

Commands are executed by activating a command string of the form `ModuleName.CommandName` within any text displayed on the screen. In the simple example above, this would be the strings `Hello.World` and `Hello.InnerWorld`. A command is usually activated by a middle-mouse click or by selecting the command string and using the key-combination `Ctrl+Return`. When a command is activated, the module with name `ModuleName` is dynamically loaded on demand and the procedure named `CommandName` in module `ModuleName` is executed.

Note that a module name has nothing to do with the name of the file that contains the source code of the module. It is pure convention that source code files are usually denominated as `ModuleName.Mod`.

1.2 Loading and Unloading Modules

When a module is compiled, an *object file* is generated and is made persistent in a file. It is not immediately loaded. Basically, a module `M` is loaded when a command of the form `M.CommandName` is activated. A module X is also automatically loaded when any module importing X is loaded (cf. Paragraph 2.1.1).

A module stays loaded until it is unloaded explicitly. Modules can be unloaded with the command

```
SystemTools.Free ModuleName1 ModuleName2 ... ~
```

where the tilde '~' indicates the end of the list. Note that a module `A` can only be unloaded if no other currently loaded module imports `A`.

2 The Language Active Oberon

2.1 Module Structure

A module consists of an import list, constant declarations, type declarations, variable declarations, procedure declarations and a body, all optional. In this Section, we describe these parts.

2.1.1 Import Section

The exported symbols of a module can be made available to other modules. To make the symbols of module `A` available in module `B`, module `A` has to be imported in module `B`. A module import graph is a directed graph where cycles are forbidden.[1]

Symbols in one module that are to be used in any other module must be exported. A symbol is exported by suffixing the name with an asterisk '`*`'. It is exported read only by suffixing with a minus symbol '`-`'. We provide a little example of a module `A` being imported by a module `B`.

```
MODULE A;
   CONST ExportedConst*=100;
END A.

MODULE B;
   IMPORT A;
   VAR c-: LONGINT;
BEGIN
   c := A.ExportedConst;
END B.
```

Fig. 2: B imports A

2.1.2 Constant Declaration Section

Although static values can be used in expressions, it is possible to associate an identifier with a constant expression, in other words: to *declare a constant.* Only values of basic type and

[1]Cycles in the import graph are detected by the compiler.

strings[2] can be declared as constants[3]. The following module contains some examples.

```
MODULE Random;

CONST
   (* ------ global constants --------- *)
   Max     = 2147483647;
   Allbits = 7FFFFFFFH; (* hex number *)

VAR
   (* ------ global variable ---------- *)
   z: LONGINT;

   PROCEDURE Rand*(): LONGINT;
   (* constants can also be declared in procedure scope: *)
   CONST A = 16807; Q = 127773; R = 2836;
   VAR t: LONGINT;
   BEGIN
      t := A * (z MOD Q) - R * (z DIV Q);
      IF t > 0 THEN z := t ELSE z := t + Max END;
      RETURN z;
   END Rand;
BEGIN
   z := Allbits;
END Random.
```

Fig. 3: Declaration and Usage of Constants

2.1.3 Type Declaration Section

Type declarations bind an identifier to a type that can be used in variable declarations to name the type of the variables (see section 2.1.4). Type declarations are used to provide synonyms for user-defined composite types or existing types such as all predeclared basic types (see sections 2.3 and 2.2 respectively). The following example uses a type declaration to assign a new name to a composite record type.

```
MODULE TypeDeclarations;

TYPE
   NumberType = REAL;
   ComplexNumber = RECORD re,im: NumberType END;

VAR
   res,a,b: ComplexNumber;
```

[2]In Active Oberon, strings are array of characters, cf. Section 2.3.1

[3]There are exceptions: special mathematical arrays can also be declared as constants but this goes beyond the scope of this short reference.

```
   PROCEDURE Multiply(VAR res: ComplexNumber; x,y: ComplexNumber);
   BEGIN
      res.re := x.re*y.re - x.im*y.im;
      res.im := x.re*y.im + x.im*y.re;
   END Multiply;

BEGIN
   (* ... *)
   Multiply(res,a,b);
   (* ... *)
END TypeDeclarations.
```

Fig. 4: Using Type Declarations

Note that the call of procedure `Multiply` with variables `res`, `a` and `b` requires that the variables are assignment compatible to the respective parameters. This is guaranteed by using the name of a type declaration which by definition always refers to the same type.

2.1.4 Variable Declaration Section

Variables have to be declared before use in a variable declaration section. A variable declaration consists of an identifier and a type. Variables can be declared globally, in a procedure scope, in an object or in a record scope.

```
MODULE Variables;
TYPE
   (* variables in a record *)
   Pair = RECORD x,y: REAL END;
   PairPointer = POINTER TO Pair;
   ExampleObject = OBJECT
      (* object variables *)
      VAR x,y: REAL;
      END ExampleObject;

VAR (* global variables *)
   a: LONGINT;
   b: Pair;
   c: PairPointer;

   PROCEDURE Example;
   (* local variables *)
   VAR a,b: REAL;
   BEGIN
   END Example;

END Variables.
```

Fig. 5: Variables and Parameters

2.1.5 Procedure Declaration Section

Procedures contain the executable code of a module. They can be declared in a module scope, within procedure scopes ('nested procedures') or in object scopes ('methods'). Parameters are special variables that refer to expressions that have been passed to the procedure from its caller. Parameters are declared in the head of the procedure declaration.

Parameters can be *value parameters*[4], in which case they represent the actual value of an expression being passed to the respective procedure. Or parameters can be *variable parameters*[5], in which case they stand for the reference of a designator that has been passed to the procedure. The modification of a value parameter within a procedure does only have a temporary effect within the procedure while the modification of a variable parameter implies the modification of the passed variable.

```
MODULE Procedures;

TYPE
   O = OBJECT

   PROCEDURE Method;
      PROCEDURE NestedProcedure;
      BEGIN
      END NestedProcedure;
   BEGIN
   END Method;

   END O;

   PROCEDURE GlobalProcedure;
      PROCEDURE NestedProcedure;
      BEGIN END NestedProcedure;
   END GlobalProcedure;

   PROCEDURE Square(x: REAL): REAL;
   BEGIN x := x*x; RETURN x
   END Square;

   PROCEDURE Inc(VAR i: INTEGER);
   BEGIN i := i+1
   END Inc;

   PROCEDURE Test*;
   VAR x,sq: REAL; i: INTEGER;
   BEGIN
      x := 10; i := 0;
      sq := Square(x);
      Inc(i);
```

[4]Value parameters are passed over the stack.

[5]Variable parameters are references being passed over the stack.

```
    END Test;

END Procedures.
```

Fig. 6: Procedures

2.1.6 Body

A module can provide a body. The body will be executed once after the module has been loaded but prior to any of its commands. Consider the following example:

```
MODULE Hello;

IMPORT KernelLog;

    PROCEDURE World*;
    BEGIN KernelLog.String("Hello World"); KernelLog.Ln;
    END World;

BEGIN
    (* this is a special 'procedure': the module body
    it is executed if and only if a module has been freshly loaded *)
    KernelLog.String("Hello has been loaded"); KernelLog.Ln;
END Hello.
```

Fig. 7: A simple Oberon Module

A first call of `Hello.World` results in the output `Hello has been loaded` and `Hello World` in the kernel log. Each subsequent call will result in an output of `Hello World`, but no further execution of the body. Only if the module has been unloaded, the body is executed again. Note that, consequently, in order to see the effects of a modification and re-compilation of a module it has to be unloaded. (cf. Section 1.2).

2.2 Basic Types

There are nine predeclared basic types in Active Oberon. Their names and valid values are shown in table 1.

2.2.1 Numeric types

The *numeric types* are comprised of the integer and real types and form a set hierarchy

$LONGREAL \supset REAL \supset HUGEINT \supset LONGINT \supset INTEGER \supset SHORTINT$

The range of the larger type includes the ranges of the smaller types. The smaller type is said to be *compatible* with the larger one in the sense that it can without danger of loss of leading digits be converted. In assignments and in expansions the conversion of internal representations is automatic.

Type name	Size	Valid values
BOOLEAN	1 byte	TRUE or FALSE
CHAR	1 byte	characters of the extended ASCII set (0X ... 0FFX)
SHORTINT	1 byte	integers between -2^7 and $2^7 - 1$
INTEGER	2 bytes	integers between -2^{15} and $2^{15} - 1$
LONGINT	4 bytes	integers between -2^{31} and $2^{31} - 1$
HUGEINT	8 bytes	integers between -2^{63} and $2^{63} - 1$
REAL	4 bytes	floating point value between -3.4028^{38} and $+3.4028^{38}$
LONGREAL	8 bytes	floating point value between -1.7976^{308} and $+1.7976^{308}$
SET	4 bytes	any set combination of the integer values between 0 and 31

Table 1: Predeclared Basic Types

2.2.2 BOOLEAN

A Boolean value is one of the *logical truth values* which are represented in Active Oberon by the standard identifiers TRUE and FALSE.

2.2.3 SET

The values which belong to the type SET are *elements of the power set of* $\{0, 1, \ldots, N\}$ where N is equal to MAX(SET), a constant defined by the implementation. It is typically the word length of the computer (or a small multiple of it). In fact, sets are efficiently implemented as bit operations.

Examples of set constants are:

```
MODULE SetConstants;

CONST
  EmptySet = {};
  SomeElements = {1, 6, 10};
  SomeOthers = {0, 2..4, 8};
END SetConstants.
```

Fig. 8: Using set constants

where `{}` denotes the empty set and the expression `2..4` refers to the elements 2, 3 and 4.

2.2.4 CHAR

A major portion of the input and output of computers is in the form of character strings that contain values of type CHAR. The value range of the type CHAR consists of the characters of the roman alphabet and a small set of special symbols used frequently in commerce and mathematics.

The set representing the type CHAR is ordered and each character has a fixed position or *ordinal number*. This is reflected in the notation for character constants which may be written as "a" or 61X for the letter a. The first representation denotes the value of the variable of type CHAR, the second its (hexadecimal) *ordinal* number.

2.3 Composite Types

In Active Oberon, there are basically three composite types available: Arrays, Records and Objects.

2.3.1 Arrays

Arrays can be declared static, open or dynamic. Static arrays are declared as ARRAY number OF BaseType, where number must be constant. The BaseType can be an array, which allows the declaration of multi-dimensional arrays. Arrays can be declared open only in a parameter section. Dynamic arrays are basically references pointing to an array with lengths that may be provided during runtime.

```
MODULE Arrays;
VAR
a: ARRAY 32 OF ARRAY 20 OF INTEGER; (* static, two-dimensional *)
b: POINTER TO ARRAY OF INTEGER; (* dynamic, one-dimensional *)

PROCEDURE Print(x: ARRAY OF INTEGER); (* open *)
VAR i: INTEGER;
BEGIN
   i := 0;
   WHILE i<LEN(x) DO
      (* Printout x[i] *)
      INC(i);
   END;
END Print;

PROCEDURE Example;
BEGIN
   NEW(b,100); (* allocate array *)
   (* ... *)
   Print(b^);
END Example;

END Arrays.
```

Fig. 9: Using arrays

2.3.2 Records

Records are containers of data. Records are value types and are declared as RECORD (variables) END. Records can also be declared as reference types using POINTER TO. Record fields are referred to via recordName.variableName.

```
MODULE Records;
VAR
   a: RECORD x,y: LONGINT END;
   p: POINTER TO RECORD a,b: REAL END;

BEGIN
   NEW(p);
   p.b := a.x;
END Records.
```

Fig. 10: Using records

2.3.3 Objects

Objects are basically *reference* records that can additionally be equipped with procedures. Procedures in an object are methods: they reside in the object scope and have access to the object's variables. An object can be explicitly referred to in its method using the SELF identifier. A method prefixed by an ampersand character & is an *object initializer*. This method is automatically called when an instance of the object is created and processed before the object becomes publicly available. An object may have at most one initializer. If absent, the initializer of the base type is inherited. Initializers can be called like methods.

```
MODULE Objects;
TYPE
MyObject = OBJECT (* class *)
   VAR x,y: REAL;

   PROCEDURE Equals(o: MyObject): BOOLEAN;
   BEGIN
      IF o = SELF THEN RETURN TRUE
      ELSE RETURN (o.x = x) & (o.y = y)
      END;
   END Equals;

   PROCEDURE &Init(x, y : REAL); (* initializer *)
   BEGIN
      SELF.x := x; SELF.y := y;
   END Init;
END MyObject;

VAR p,q: MyObject; (* objects / instances *)
```

```
BEGIN
   (* instantiate object p with x and y parameters for initializer *)
   NEW(p, 1.0, 9.99);
   (* ... *)
   IF p.Equals(q) THEN (* ... *)
   END;
END Objects.
```

Fig. 11: Usage of Objects

2.3.4 Inheritance

Active Oberon is an object oriented language. Inheritance is supported for records and objects. It is possible to extend records and objects and to use type checks, type guards and overriding of methods. Record and object extension is explicated as in the following example:

```
MODULE Extensions;

TYPE
Rectangle = RECORD
   x,y,w,h: INTEGER
END;

FilledRectangle = RECORD(Rectangle)
   color: INTEGER
END;

Window = OBJECT
VAR x,y,w,h: INTEGER;

   PROCEDURE Print;
   BEGIN
      (* draw frame *)
   END Print;
END Window;

FilledWindow = OBJECT (Window)
VAR color: INTEGER;

   PROCEDURE Print; (* overrides Window.Print *)
   BEGIN
      Print^; (* supercall *)
      (* fill *)
   END Print;
END FilledWindow;

END Extensions.
```

Fig. 12: Type extension of records and objects

2.4 Concurrency Support

Active Oberon provides built-in concurrency support. Threads are represented as *Active Objects* and language constructs for ensuring mutual exclusion and thread synchronization are provided.

2.4.1 Active Objects

The declaration of an object type may include an *object body.* The body is the object's activity, to be executed whenever an object instance is allocated after the initializer (if any) completed execution. The object body is annotated with the *ACTIVE* modifier. At allocation, a new process is allocated to execute the body concurrently. If the ACTIVE modifier is not present, the body is executed synchronously, i.e. NEW returns only after the body has terminated execution. The active object activity terminates whenever the body execution terminates. As long as the body executes, the object is kept alive (in particular it cannot be garbage collected).

```
MODULE Example;
TYPE
  ActiveObject = OBJECT
  BEGIN {ACTIVE}
     (* do some useful work *)
  END ActiveObject;

VAR
  o : ActiveObject;

BEGIN
  (* instantiate active object o *)
  NEW(o);
END Example.
```

Fig. 13: Example of an active object

2.4.2 Protection

Like a (procedure-, module- or object-) body, a Statement Block is a sequence of statements delimited by BEGIN and END. It can be used anywhere like a simple statement. An *EXCLUSIVE* modifier turns a statement block (or body) into a critical region to protect the statements against concurrent execution. Our protection model is an instance-based monitor: Every object instance is protected and the protection granularity is any statement block inside the object's method, ranging from a single statement to a whole method.

Upon entering an exclusive block, an activity is preempted as long as another activity stands in an exclusive block of the same object instance. An activity cannot take an object's lock more than once (re-entrance is not allowed). Modules are considered singleton objects, thus procedures in a module can also be protected. In this case the scope of protection is the whole module.

```
MODULE Demo;

TYPE
   SomeObject = OBJECT

      PROCEDURE O1;
      BEGIN {EXCLUSIVE}
         (* critical section o1 *)
      END O1;

      PROCEDURE O2;
      BEGIN
         (* non-critical section *)
         BEGIN {EXCLUSIVE} (* critical section o2 *) END;
         (* non-critical section *)
      END O2;

   END SomeObject;

PROCEDURE P1;
BEGIN {EXCLUSIVE}
   (* critical section p1 *)
END P1;

PROCEDURE P2;
BEGIN
   (* non-critical section *)
   BEGIN {EXCLUSIVE} (* critical section p2 *) END;
   (* non-critical section *)
END P2;

END Demo.
```

Fig. 14: Usage of exclusive blocks

In this example, at most one thread can be in one of the critical sections of the module *p1* or *p2* at any time. For each instance of SomeObject, at most one thread can be in *o1* or *o2* at any time. Note that there is no relation of *p1*, *p2* and *o1*, *o2* in different object instances since the protection is based on *instance-based* monitors.

2.4.3 Synchronization

The built-in procedure *AWAIT* is used to synchronize an activity with a state of the system. AWAIT can take any boolean condition. The activity is allowed to continue execution only when the condition is true. While the condition is not established, the activity remains suspended. The lock on the protected object is released, as long as the activity remains suspended[6]. The

[6]Releasing the lock upon waiting allows other activities to change the state of the object and thus establish the condition

activity is resumed if and only if the lock can be taken. The system is responsible for evaluating the conditions and for restarting suspended activities. The conditions inside an object instance are re-evaluated whenever some activity leaves a protected block inside the same object instance. This implies that changing the state of an object outside a protected block does not imply a re-evaluation of its conditions. When several activities compete for the same object lock, the activities whose conditions are true are scheduled before those that only want to enter a protected region.

```
MODULE Example;

TYPE
    Synchronizer = OBJECT
    VAR awake : BOOLEAN;

        PROCEDURE &Init;
        BEGIN
            awake := FALSE;
        END Init;

        PROCEDURE Wait;
        BEGIN {EXCLUSIVE}
           AWAIT(awake); (* suspend caller until awake = TRUE *)
           awake := FALSE;
        END Wait;

        PROCEDURE Wakeup;
        BEGIN {EXCLUSIVE}
           awake := TRUE;
        END Wakeup;

    END Synchronizer;

END Example.
```

Fig. 15: Example of object synchronization

2.4.4 Examples

The following example shows the implementation of a bounded-buffer.

```
MODULE Example;

TYPE
  Item* = OBJECT END Item;

  Buffer* = OBJECT
  VAR head, num: LONGINT; buffer: POINTER TO ARRAY OF Item;

    PROCEDURE Append*(x: Item);
    BEGIN {EXCLUSIVE}
      AWAIT(num # LEN(buffer));
      buffer[(head+num) MOD LEN(buffer)] := x;
      INC(num)
    END Append;

    PROCEDURE Remove*(): Item;
    VAR x: Item;
    BEGIN {EXCLUSIVE}
      AWAIT(num # 0);
      x := buffer[head];
      head := (head+1) MOD LEN(buffer);
      DEC(num);
      RETURN x
    END Remove;

    PROCEDURE &Init*(n: LONGINT);
    BEGIN
      head := 0; num := 0; NEW(buffer, n)
    END Init;
  END Buffer;

END Example.
```

Fig. 16: Bounded-buffer

3 Application Programming Interfaces

This section comprises a brief overview of often used application programming interfaces.

3.1 Streams

Streams provide an abstraction for easily accessing various resources that can be accessed byte-wise. Streams can be opened on files, network connections, serial port connections, strings, memory and many other resources. Streams are unidirectional, i.e. it is possible to either read from a resource (*Readers*) or write to a resource (*Writers*). Readers maintain an input buffer and writers maintain an output buffer that has to be flushed explicitly by the programmer. Streams maintain an internal position that is updated automatically whenever a read or write operation is performed. Streams may support random access, depending on the underlying resource.

```
MODULE Streams;

 (** A reader buffers input received from a Receiver.
    Must not be shared between processes. *)
Reader* = OBJECT
```

```
(** result of last input operation *)
res*: LONGINT;

(** Current position. *)
PROCEDURE Pos*( ): LONGINT;

(** Returns TRUE if this stream supports random access *)
PROCEDURE CanSetPos*( ): BOOLEAN;

(** Set position to <pos> (only if resource supports random access) *)
PROCEDURE SetPos*( pos: LONGINT );

(** Return number of bytes currently available in input buffer. *)
PROCEDURE Available*( ): LONGINT;

(** Read one byte. *)
PROCEDURE Char*( VAR x: CHAR );

(** Read one byte but leave the byte in the input buffer. *)
PROCEDURE Peek*( ): CHAR;

(** Read size bytes into x, starting at ofs.
   The len parameter returns the number of bytes that were actually
      read. *)
PROCEDURE Bytes*( VAR x: ARRAY OF CHAR; ofs, size: LONGINT; VAR len:
   LONGINT );

(** Skip n bytes on the reader. *)
PROCEDURE SkipBytes*( n: LONGINT );

(** -- Read formatted data (uses Peek for one character lookahead) --
   *)

(** Read an integer value in decimal or hexadecimal.
   If hex = TRUE, recognize the "H" postfix for hexadecimal numbers. *)
PROCEDURE Int*( VAR x: LONGINT; hex: BOOLEAN );

(** Read all characters until the end of the line (inclusive).
   If the input string is larger than x, read the full string and
   assign the truncated 0X-terminated value to x. *)
PROCEDURE Ln*( VAR x: ARRAY OF CHAR );

(** Skip over all characters until the end of the line (inclusive). *)
PROCEDURE SkipLn*;

(** Skip over space, TAB and EOLN characters. *)
PROCEDURE SkipWhitespace*;

(** Read an optionally "" or '' enquoted string.
      Will not read past the end of a line. *)
   PROCEDURE String*( VAR string: ARRAY OF CHAR );

   (** First skip whitespace, then read string *)
   PROCEDURE GetString*(VAR string : ARRAY OF CHAR);

   (** First skip whitespace, then read integer *)
   PROCEDURE GetInteger*(VAR integer : LONGINT; hex : BOOLEAN);

 END Reader;

 Writer* = OBJECT

   (** result of last output operation *)
   res*: LONGINT;

   (** Current position. *)
   PROCEDURE Pos*( ): LONGINT;

   (** Returns TRUE if this stream supports random access *)
   PROCEDURE CanSetPos*( ): BOOLEAN;

   (** Set position to <pos> (only if resource supports random access) *)
   PROCEDURE SetPos*( pos: LONGINT );

   (** Flush output buffer *)
   PROCEDURE Update*;

   (** Write one byte. *)
   PROCEDURE Char*( x: CHAR );

   (** Write len bytes from x, starting at ofs. *)
   PROCEDURE Bytes*(CONST x: ARRAY OF CHAR; ofs, len: LONGINT );

   (** -- Write formatted data -- *)

   (** Write an ASCII end-of-line (CR/LF). *)
   PROCEDURE Ln*;

   (** Write a 0X-terminated string, excluding the 0X terminator. *)
   PROCEDURE String*(CONST x: ARRAY OF CHAR );

   (** Write an integer in decimal
      right-justified in a field of at least w characters. *)
   PROCEDURE Int*( x, w: LONGINT );

   (** Write an integer in hexadecimal
      right-justified in a field of at least ABS(w) characters. *)
   PROCEDURE Hex*(x: HUGEINT; w: LONGINT );
```

```
END Writer;

END Streams;
```

Fig. 17: Simplified Reader and Writer Interface

3.2 Commands

Commands are procedures that can be invoked directly by the command interpreter. As has already been indicated in Section 1.1, a procedure can act as a command if and only if it is exported, if it is declared in the module scope and if it has a command signature:

```
(** Command with no arguments *)
PROCEDURE ProcedureName*;

(** Command with arguments and output stream *)
PROCEDURE ProcedureName*(context : Commands.Context);
```

Fig. 18: Command Signatures

An excerpt of the Commands module interface reads as follows:

```
MODULE Commands;
TYPE
   Context* = OBJECT
   VAR
      in-, arg- : Streams.Reader;
      out-, error- : Streams.Writer;
      caller- : OBJECT;
   END Context;
END Command;
```

Fig. 19: Command Context

Command line arguments can be accessed using the *arg* stream. The stream *in* is the command input stream. For output, the streams *out* and *error* are used. The caller field optionally contains a reference to an object that is responsible for the command invocation. Note that both output stream buffers (out and error) are flushed automatically by the command interpreter.

```
MODULE Example;
IMPORT Commands;

  PROCEDURE Add*(context : Commands.Context);
  VAR a, b : LONGINT;
  BEGIN
    context.arg.GetInteger(a, FALSE);
    context.arg.GetInteger(b, FALSE);
    context.out.Int(a, 0); context.out.String(" + "); context.out.Int(b,
       0);
    context.out.String(" = "); context.out.Int(a + b, 0);
    context.out.Ln;
  END Add;
END Example.
```

Fig. 20: Command Example

After compilation of module Example, the command interpreter is able to process the command `Example.Add 3 5 ~`. Note that the tilde character indicates the end of the command line arguments.

3.3 Files

Module Files provides both the interface to be implemented by file system drivers and $\mathcal{A}_2$ file API.

The most useful operations are

```
MODULE Files;
TYPE
  File* = OBJECT;

  (** Open file <filename>. Returns NIL if file cannot be opened *)
  Old*(filename : ARRAY OF CHAR) : File;

  (** Create file <filename>. Returns NIL if file cannot be created *)
  New*(filename : ARRAY OF CHAR) : File;

  (** Register a new file (create entry in file system directory) *)
  Register*(f : File);
END Files;
```

Fig. 21: Basic Files API

The actual access to files is achieved by using the low-level file riders or by using streams on a file. We describe both methods in the subsequent sections.

File Interface The low-level file interface uses so-called riders as context for accessing files. Multiple riders can be positioned independently on a given file. Essentially, riders are used for tracking the position in a file and store the result of the last operation on the file. As for streams, read and write operations automatically update the current position.

```
 (** A rider points to some location in a file,
    where reading and writing will be done. *)
Rider* = RECORD (** not shareable between multiple processes *)
  (** has end of file been passed *)
  eof*: BOOLEAN;

  (** leftover byte count for ReadBytes/WriteBytes *)
  res*: LONGINT;
END;

File* = OBJECT (** sharable *)
  (* ... *)

  (** Position a Rider at a certain position in a file.
     Multiple Riders can be positioned at different locations in a file.
     A Rider cannot be positioned beyond the end of a file. *)
  PROCEDURE Set*(VAR r: Rider; pos: LONGINT);

  (** Return the offset of a Rider positioned on a file. *)
  PROCEDURE Pos*(VAR r: Rider): LONGINT;

  (** Read a byte from a file, advancing the Rider one byte further.
     R.eof indicates if the end of the file has been passed. *)
  PROCEDURE Read*(VAR r: Rider; VAR x: CHAR);

  (** Read a sequence of len bytes into the buffer x at offset ofs,
     advancing the Rider. Less bytes will be read when reading over the
     end of the file. r.res indicates the number of unread bytes. *)

  PROCEDURE ReadBytes*(VAR r: Rider; VAR x: ARRAY OF CHAR; ofs, len:
     LONGINT);

  (** Write a byte into the file at the Rider position,
     advancing the Rider by one. *)
  PROCEDURE Write*(VAR r: Rider; x: CHAR);

  (** Write the buffer x containing len bytes (starting at offset ofs)
     into a file at the Rider position. *)
  PROCEDURE WriteBytes*(VAR r: Rider; CONST x: ARRAY OF CHAR; ofs, len:
     LONGINT);

  (** Return the current length of a file. *)
  PROCEDURE Length*(): LONGINT;

  (** Flush the changes made to a file from its buffers. *)
  PROCEDURE Update*;

END File;
```

Fig. 22: Low-level Files API

Using Streams On Files First we display a part of the Files interface:

```
MODULE Files;

TYPE
```

```
  File* = OBJECT; (* as described in last section *)
  Reader* = OBJECT (Streams.Reader);
  Writer* = OBJECT (Streams.Writer);

  OpenReader*(VAR r: Reader; f : File; position : LONGINT);
  OpenWriter*(VAR w : Writer; f : File; position : LONGINT);
END Files.
```

Fig. 23: Using Streams On Files

Next we provide an example of how to use streams on files.

```
MODULE Example;
IMPORT Commands, Files;

PROCEDURE CreateFile*(context : Commands.Context);
VAR
  filename : Files.FileName;
  file : Files.File; writer : Files.Writer;
  ch : CHAR;
BEGIN
   context.arg.GetString(filename);
   file := Files.New(filename);
   IF (file # NIL) THEN
     Files.OpenWriter(writer, file, 0);
     context.arg.Char(ch); (* skip argument delimiter character *)
     REPEAT
       context.arg.Char(ch);
       writer.Char(ch);
     UNTIL (ch = 0X);
     writer.Update;
     Files.Register(file);
   END;
END CreateFile;

END Example.
```

Fig. 24: Example: Create A New File

In this example, executing the command `Example.CreateFile HelloWorld.txt Hello World ~` would create (or overwrite) the file ”HelloWorld.txt” and write the 0X-terminated string “Hello World ” into it.

3.4 Strings

The module Strings provides procedures for ASCII string manipulation.

```
MODULE Strings;

 (** returns the length of a string *)
```

```
PROCEDURE Length* (CONST string: ARRAY OF CHAR): LONGINT;

(** appends appendix to s: s := s || appendix *)
PROCEDURE Append* (VAR s: ARRAY OF CHAR; CONST appendix: ARRAY OF CHAR);

(** concatenates s1 and s2: s := s1 || s2 *)
PROCEDURE Concat* (CONST s1, s2: ARRAY OF CHAR; VAR s: ARRAY OF CHAR);

(** converts an integer value to a string *)
PROCEDURE IntToStr*(i: LONGINT; VAR s: ARRAY OF CHAR);

(** Simple pattern matching with support for "*" and "?" wildcards
   returns TRUE if name matches mask. *)
PROCEDURE Match*(CONST mask, name: ARRAY OF CHAR): BOOLEAN;

END Strings.
```

Fig. 25: Strings

A Appendix

A.1 Built-in Functions

There are some built-in procedures and functions in Active Oberon. We give a short overview in the following table. Note that Integer stands for SHORTINT, INTEGER, LONGINT and HUGEINT and Number stands for integers plus REAL and LONGREAL.

Function	Argument Types	Result Type	Description
INC(x)	x: Integer		increment x by 1
DEC(x)	x: Integer		decrement x by 1
INC(x,n)	x: Integer; n: Integer		increment x by n
DEC(x,n)	x: Integer; n: Integer		decrement x by n
ASSERT(x)	x: BOOLEAN		assert trap, if x not true
COPY(x,y)	x,y: ARRAY OF CHAR		0X-terminated copy of x to y
INCL(s,e)	s: SET, e: Integer		include element e in set s
EXCL(s,e)	s: SET, e: Integer		exclude element e from set s
HALT(n)	n: Integer		generate a trap with number n
NEW(x,...)	x: Object or Pointer		allocate x
ABS(x)	x: Number	Number	return absolute value of x
ASH(x,y)	x,y: Integer	Integer	return arithmetic shift of x by y bits
CAP(x)	x: CHAR	CHAR	return capital letter of x
CHR(x)	x: Integer	CHAR	return character with ascii-number x

Function	Argument Types	Result Type	Description
ENTIER(x)	x: REAL or LONGREAL	LONGINT	return largest integer not greater than x
LEN(x)	x: ARRAY OF CHAR	LONGINT	return length of x
MAX(t)	t: Number or SET	Number	return maximal number of basic type t
MIN(t)	t: Number or SET	Number	return minimal number of basic type t
ODD(x)	x: Integer	BOOLEAN	return if x is odd
ORD(x)	x: CHAR	LONGINT	return ascii-number of x
SHORT(x)	x: Number	Number	number conversion down
LONG(x)	x: Number	Number	number conversion up

The number conversion routines SHORT and LONG operate with respect to the relations

$$LONGREAL \supset REAL, HUGEINT \supset LONGINT \supset INTEGER \supset SHORTINT.$$

A.2 The Module SYSTEM

The (pseudo-)module SYSTEM contains definitions that are necessary to directly refer to resources particular to a given computer and/or implementation. These include facilities for accessing devices that are controlled by the computer, and facilities to override the data type compatibility rules otherwise imposed by the language definition. The functions and procedures exported by this module should be used with care! It is recommended to restrict their use to specific low-level modules. Such modules are inherently non-portable and easily recognized due to the identifier SYSTEM appearing in their import list. The subsequent definitions are applicable to the $\mathcal{A}_2$ operating system.

A.2.1 BIT Manipulation

Function	Argument Types	Result Type	Description
BIT(adr,n)	adr: ADDRESS; n: LONGINT	BOOLEAN	Returns TRUE if bit n at adr is set, FALSE otherwise
LSH(x,n)	x: Integer; n: LONGINT	Integer	Returns value x logically shifted left n bits (shifts right for $n < 0$)
ROT(x,n)	x: Integer; n: LONGINT	Integer	Returns value x rotated left by n bits (rotates right for $n < 0$)

A.2.2 SYSTEM Types

Type	Description
ADDRESS	Representation of memory addresses. Currently, this is an alias to either LONGINT or HUGEINT
SIZE	Representation of results of arithmetic operations on memory addresses.
BYTE	Representation of a single byte.

A.2.3 Addresses, Sizes and Unsafe Typecasts

Function	Argument Types	Result Type	Description
ADR(v)	v: ANY	ADDRESS	Returns the address of v
SIZEOF(v)	v: ANY	SIZE	Returns the size of type v
VAL(T,x)	T: Type; x: ANY	T	Unsafe type cast. Returns x interpreted as type T with no conversion

A.2.4 Direct Memory Access Functions

Function	Argument Types	Result Type	Description
PUT(adr,x)	adr: ADDRESS; x: Type		Mem[adr] := x where the size of type x is 8, 16, 32 or 64 bits

PUT8(adr,x)
PUT16(adr,x)
PUT32(adr,x)
PUT64(adr,x)
	adr: ADDRESS; x: SHORTINT
	adr: ADRESS; x: INTEGER
	adr: ADDRESS; x: LONGINT
	adr: ADDRESS; x: HUGEINT
			Mem[adr] := x

GET(adr,x)	adr: ADDRESS; VAR x: Type		x := Mem[adr] where the size of type x is 8, 16, 32 or 64 bits

GET8(adr)
GET16(adr)
GET32(adr)
GET64(adr)
	adr: ADDRESS
	adr: ADDRESS
	adr: ADDRESS
	adr: ADDRESS
		SHORTINT
		INTEGER
		LONGINT
		HUGEINT
			RETURN Mem[adr]

MOVE(src, dst,n)	dst: ADDRESS; n: SIZE		Copy ”n” bytes from address ”src” to address ”dst”

A.2.5 IA-32 Specific Functions

Function	Argument Types	Description
PORTIN(adr,x)	adr: LONGINT; VAR x: Type	Perform a port input instruction at the specified I/O address. The size of type x must be 8, 16 or 32 bits
PORTOUT(adr,x)	adr: LONGINT; x: Type	Perform a port output instruction at the specified I/O address. The size of type x must be 8, 16 or 32 bits
CLI()		Disable interrupts on the current processor
STI()		Enable interrupts on the current processor
GETREG(reg,x)	reg: LONGINT; VAR x: Type	x := REGISTER(reg) where the size of type x is 8, 16, 32 or 64 bits depending on the register
PUTREG(reg,x)	reg: LONGINT; x: Type	REGISTER(reg) := x; where SIZEOF(x) is 8, 16, 32 or 64 bits
EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI		32 bit registers
AX, CX, DX, BX, SP, BP, SI, DI		16 bit registers
AL, CL, DL, BL, AH, CH, DH, BH		8 bit registers

A.3 Active Oberon EBNF

We display the syntax of Active Oberon in the Extended Backus-Naur-Form (EBNF). We present productions (syntactic equations) as equations with a single equal sign `=`. On the left hand side of a production stands the defined nonterminal symbol, the right hand side contains the substitution rule and is terminated by a period. Terminal symbols are embraced by single or double quotes (for example `':='`, `"'"` and `'begin'`). An alternative in a production is denoted by a vertical bar `|`. Brackets `[` and `]` denote optionality of the enclosed expression, while braces `{` and `}` denote its repetition (possibly 0 times). Additionally, parentheses `(` and `)` are used to enclose expressions and thereby control additional precedence.

```
Module              = 'MODULE' Identifier ['IN' Identifier]';'
                        [ImportList] DeclarationSequence [Body]
                      'END' Identifier '.'.

ImportList          = 'IMPORT' Import { ',' Import } ';'.

Import              = Identifier [':=' Identifier] ['IN' Identifier].

DeclarationSequence =  { 'CONST' [ConstDeclaration] {';' [ConstDeclaration]}
                        |'TYPE'  [TypeDeclaration] {';' [TypeDeclaration]}
                        |'VAR'   [VariableDeclaration] {';' [VariableDeclaration]}
                       }
                       [ProcedureDeclaration | OperatorDeclaration]
                       {';' [ProcedureDeclaration | OperatorDeclaration] }.
```

```
ConstDeclaration      = IdentifierDefinition '=' Expression.

TypeDeclaration       = IdentifierDefinition '=' Type.

VariableDeclaration   = VariableNameList ':' Type.

ProcedureDeclaration = 'PROCEDURE' ['&'|'-'|SystemFlag] IdentifierDefinition
    [FormalParameters]';'
                        DeclarationSequence  [Body] 'END' Identifier.

OperatorDeclaration  = 'OPERATOR' String ['*'|'-'] FormalParameters ';'
                        DeclarationSequence [Body] 'END' String.

SystemFlag           = '{' Identifier '}'.

IdentifierDefinition = Identifier ['*'|'-'].

FormalParameters     = '('[ParameterDeclaration {';' ParameterDeclaration}]')' [':' Type].

ParameterDeclaration = ['VAR'|'CONST'] Identifier {',' Identifier}':' Type.

Type                 = ArrayType | RecordType | PointerType | ObjectType | ProcedureType
                        | QualifiedIdentifier.

ArrayType            = 'ARRAY' [Expression {',' Expression}
                        | '[' MathArraySize {',' MathArraySize} ']' ] 'OF' Type.

MathArraySize        = Expression | '*' | '?'.

RecordType           = 'RECORD' ['(' QualifiedIdentifier ')']
                        [VariableDeclaration {';' VariableDeclaration}] 'END'.

PointerType          = 'POINTER' 'TO' Type.

ObjectType           = 'OBJECT' ['(' QualifiedIdentifier ')'] DeclarationSequence [Body]
                        'END' [Identifier]
                      | 'OBJECT'.

ProcedureType        = 'PROCEDURE' [SystemFlag] [FormalParameters].

Body                 = 'BEGIN' ['{' BlockModifiers '}'] StatementSequence
                        ['FINALLY' StatementSequence]
                      | 'CODE' {any}.

BlockModifiers       = [Identifier ['(' Expression ')'] {',' Identifier ['(' Expression ')']
    }]

StatementSequence    = Statement {';' Statement}.

Statement            =
                      [
                      Designator [':=' Expression]
                      | 'IF' Expression 'THEN' StatementSequence
                         {'ELSIF' Expression 'THEN' StatementSequence} 'END'
                      | 'WITH' Identifier ':' QualifiedIdentifier 'DO'
                          StatementSequence 'END'
                      | 'CASE' Expression 'OF' ['|'] Case
                         {'|' Case} ['ELSE' StatementSequence] 'END'
                      | 'WHILE' Expression 'DO' StatementSequence 'END'
                      | 'REPEAT' StatementSequence 'UNTIL' Expression
```

```
                       | 'FOR' Identifier ':=' Expression 'TO' Expression ['BY' Expression]
                           'DO'
                           StatementSequence 'END'
                       | 'LOOP' StatementSequence 'END'
                       | 'EXIT'
                       | 'RETURN' [Expression]
                       | 'AWAIT' Expression
                       | 'BEGIN' StatementBlock 'END'
                       ].

StatementBlock         = ['{' BlockModifiers '}'] StatementSequence.

Case                   = Element {',' Element} ':' StatementSequence.

Expression             = SimpleExpression [RelationOp SimpleExpression].

RelationOp             = '=' | '.=' | '#' | '.#'
                       | '<' | '.<' | '<=' | '.<=' | '>' | '.>' | '>=' | '.>='
                       | 'in' | 'is'

SimpleExpression       = ['+'|'−'] Term {AddOp Term}.

AddOp                  = '+' | '−' | 'OR'.

Term                   = Factor {MulOp Factor}.

MulOp                  = '*' | '**' | '.*' | '+*' | '/' | './' | 'DIV' | 'MOD' | '&'.

Factor                 = Number | Character | String | 'NIL' | 'TRUE' | 'FALSE' | Set
                       | '(' Expression ')' | '~' Factor | Factor '`' | Designator
                       | MathArrayExpression.

MathArrayExpression    = '[' Expression {',' Expression} ']'.

Set                    = '{' [Element {',' Element}] '}'.

Element                = Expression ['..' Expression].

Designator             = ('SELF' | Identifier)
                         {'.' Identifier|'[' RangeList ']' | '('[ExpressionList]')' | '^'}.

RangeList              = Range {',' Range}.

Range                  = Expression | [Expression] '..' [Expression] ['by' Expression] | '?' |
    '*'.

ExpressionList         = Expression {','Expression}.

VariableNameList       = IdentifierDefinition [SystemFlag] {',' IdentifierDefinition
    [SystemFlag]}.

IdentifierDefinition = Identifier [ '*' | '−' ].

QualifiedIdentifier    = Identifier ['.' Identifier].


Identifier             = Letter {Letter | Digit | '_'}.

Letter                 = 'A' | 'B' | .. | 'Z' | 'a' | 'b' | .. | 'z'.

String                 = '"' {Character} '"' | "'" {Character} "'".
```

```
Number              = Integer | Real.

Integer             = Digit {Digit} | Digit {HexDigit} 'H'.

Real                = Digit {Digit} '.' {Digit} [ScaleFactor].

ScaleFactor         = ('E' | 'D') ['+' | '-'] digit {digit}.

HexDigit            = Digit | 'A' | 'B' | 'C' | 'D' | 'E' | 'F'.

Digit               = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' .
```