Abstract

The Oberon environment [Wir89] offers only limited possibilities for inspecting run time data and for determining the cause for an exception. Only variables of basic types such as INTEGER or CHAR can be examined. It is not possible to execute a program step by step.

This report describes a run time debugger that allows the inspection of structured types such as RECORD or ARRAY as well as step by step execution. The debugger supports views on local and global variables of any type. Run time types are supported as well as dynamic arrays. It is possible to follow pointers in order to traverse complex data structures. The debugger supports single stepping as well as breakpoints.

Another goal was to embed the debugger into the Oberon environment as smoothly as possible. This results in a look and feel typical for Oberon. Many features of the Oberon system (text frames, text elements) have been reused, resulting in small and compact code. The run time debugger was implemented for PowerMac Oberon.

Keywords: Programming techniques, Testing and Debugging; Programming Environments Operating Systems, Systems Programs and Utilities

Contents

1. Introduction

2. How to Use the Debugger
   2.1 An Example
   2.2 Commands
   2.3 Data Viewers
   2.4 Limitations

3. Implementation
   3.1 Compiler support
   3.2 Stepwise execution
      3.2.1 Patching
      3.2.2 Context switch
   3.3 Determine values of variables
   3.4 User interface
      3.4.1 Source Viewers
      3.4.2 Data Viewers

4. Extensibility
   4.1 Module hierarchy
   4.2 RTDT (low-level Trap handling, context switches)
   4.3 RTDB (Breakpoint handling)
   4.4 RTDC (Compiler Interface)
   4.5 RTDD (Data viewers)

5. Conclusions

6. References

1. Introduction

A major drawback of the Oberon system is its lack of a debugger. In Oberon there are only three means for getting information about a running program:

- The command *System.State* (showing part of the global variables of a module).
- The command *System.ShowModules* (showing the list of all loaded modules).
- The trap viewer (showing the active procedures on the stack as well as part of their variables).

One of the problems for successful debugging is buried within the words 'part of'. Only variables, for which there exists reference information in the object module, are shown. This reference information is created by the compiler. Unfortunately, the compiler does not generate this information for all variables. Structured types such as arrays or records are omitted. Therefore variables of these types cannot be shown. Pointers can only be shown as the address of the object they reference.

To circumvent these problems a post mortem debugger was implemented as a first step [Hof94]. This tool displays the following information:

- Global and local variables of all modules and active procedures regardless of their types.
- The source code position of the program counters for all active procedures on the stack.

Additionally some general design goals were set, which partially contradict each other. The main objective was to find an optimal balance between the different goals:

- As simple as possible.
- Minimal use of memory.
- No interference with normal work. No additional delays.
- Conformity with the Oberon environment and usage conventions. User interface with commands, viewers and text elements.
- No modifications of the Oberon system and the Oberon compiler which are not absolutely necessary.

These goals were met to a large degree. The post mortem debugger needed almost no memory on the Oberon heap, only one minor change in the compiler, and was built on top of the Oberon environment using text elements as well as viewers. As a result of making it as simple as possible, it needed only about 40 kBytes of disk space.

Daily usage in research as well as in teaching showed, that there was still some need for a more advanced debugging tool. This led to the development of the run time debugger presented in this paper. The run time debugger meets the same design goals as the post mortem debugger. Additionally it offers the following new features:

- Step by step execution of a program.
- Setting of breakpoints.
- Views on variables updated automatically after every debugging step.
- Reuse of parts of the post mortem debugger.

2. How to Use the Debugger

We first explain the usage of the debugger with an example. Then we describe its commands and its other abilities in depth.

2.1 An Example

Consider the following sample program:

```
MODULE Demo;

    IMPORT Out;

    PROCEDURE Do*;
        VAR i: INTEGER; arr: ARRAY 3 OF INTEGER;
    BEGIN
        Out.String ("Hello world$");
        FOR i := 0 TO 2 DO
            arr[i] := i*2;
            Out.String ("Hello again$")
        END
    END Do;

END Demo.
```

To start debugging, the debugger has to know which modules should be traced by the debugger. In our case we only want to trace the module Demo.

```
Debug.Trace Demo ~
```

After running this command everything is prepared. Debugging is started automatically upon entering one of the marked modules (here Demo). In our example we enter the traced code by starting the command *Demo.Do* with a click on the middle mouse button. After a traced module has been activated, execution is suspended and a viewer is opened showing the source text corresponding to the current program counter (PC). The exact position of the PC is marked with a special text element:

```
MODULE Demo;

    IMPORT Out;

    PROCEDURE Do*;PC
        VAR i: INTEGER; arr: ARRAY 3 OF INTEGER;
    BEGIN
        Out.String ("Hello world$");
        FOR i := 0 TO 4 DO
            arr[i] := i*2;
            Out.String ("Hello again$")
        END
    END Do;

END Demo.
```

The program may now be executed step by step by using the command *Debug.Step*. Alternatively, it can be executed up to the next breakpoint with *Debug.Run*. With *Debug.Data Demo.Do* one can open a so called data viewer. A data viewer always displays the variables of a procedure or module together with their current values. In this case it displays the values of *i* and *arr*. If one would open a data viewer of *Demo.Do* in the above situation, the values of *i* and *arr* were not defined. After stepping through some statements, one gets to the following state:

```
MODULE Demo;

    IMPORT Out;

    PROCEDURE Do*;
        VAR i: INTEGER; arr: ARRAY 3 OF INTEGER;
    BEGIN
        Out.String ("Hello world$");
        FOR i := 0 TO 2 DO
            arr[i] := i*2;
            Out.String ("Hello again$")
        END
    END Do;

END Demo.
```

From step to step, the data viewer is automatically updated, to show the current values of the variables. This results in the following display of the variables of *Demo.Do* in the data viewer:

```
arr = ▷
    0 = 0
    1 = 2
    2 = 4 ◁
i = 2
```

The components of the array *arr* are enclosed in so-called fold elements (opening and closing triangles) [MoK95]. By middle clicking on one of the triangles, one can collapse the text between these elements (giving ▶◀ instead of ▷...◁). By default, fold elements are closed to avoid long data viewers which tend to be difficult to inspect (Fold elements are distributed with the standard Oberon environment).

Instead of the command interface described above, one may also use the Dialogs package [Kna94] as a means to communicate with the debugger (Figure 1).

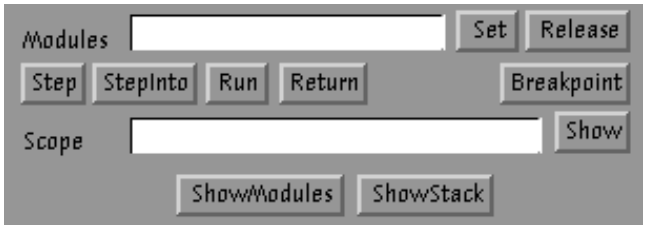


Figure 1: Debugger Dialog

2.2 Commands

Like other Oberon tools, the debugger is not a program for itself, but a collection of modules which export several commands. There are two types of commands:

- Initialisation commands define the state of the debugger.
- Debugging commands change or give information about the currently debugged program.

Initialisation Commands

- *Debug.Trace {module}*. Registers the modules that are to be traced. When a module has been registered, execution halts either when one of its statements is reached in single step mode or when a breakpoint in this module is encountered. Note that it is possible to selectively trace individual modules of a system.
- *Debug.Close*: Ends the current debugging session. The allocated memory is freed and the debugged modules are restored to their former state.
- *Debug.ShowModules*: Shows the set of currently registered modules.
- *Debug.PostMortem*: Installs the debuggers Post Mortem facility. The machine state of a later trap may then be inspected as if a debugging step has been executed.

Debugging Commands

- *Debug.StepInto*: Executes the next statement after the PC position. If this statement calls a procedure of a registered module, execution is stopped before the first statement of the procedure (step into mode).
- *Debug.Step*: Executes the next statement after the PC position. If this statement calls a procedure, this procedure is not debugged but executed completely, even if it is in a registered module (step over mode). However, an encountered breakpoint inside of the procedure will cause a program stop.
- *Debug.Breakpoint*: Sets a breakpoint in front of the statement containing the caret. The breakpoint is shown as . During debugging, an encountered breakpoint triggers a program stop.
- *Debug.Run*: Executes the debugged program until a breakpoint is encountered or the end of the program is reached (i.e. the command returns to the Oberon loop).
- *Debug.Data (module | procedure | ^)*: Opens a data viewer and shows all variables of the specified module or procedure (if that procedure has an active frame on the current stack). Note that it is also possible to inspect the global data of a module which is not registered.
- *Debug.Return*: Executes the debugged program until the current procedure returns to its caller. However, execution will be stopped, if a user-defined breakpoint is encountered.
- *Debug.Source (procedure)*: Shows the source code of the specified procedure and sets the caret to the current position within this procedure. The procedure must have an active frame on the stack. This is useful for determining the exact halt position when the debugger is used in post mortem mode.
- *Debug.Stack*: Displays a list of procedures having an active frame on the stack. After a trap these are the procedures that were active when the trap occurred.

2.3 Data Viewers

Data viewers display the values of local, global or heap variables. They are opened with the command *Debug.Data*. A data viewer is a normal text viewer. All commands applicable to text viewers are also valid for data viewers.

Variables of basic types, strings, procedure variables and NIL-pointers are displayed with their respective names and values. For example:

```
i = 100
ch = 'C'
str = "a string"
real = 3.14159265D+000
bool = TRUE
set = {3, 17}
proc = Texts.Notify
ptr = NIL
```

Variables of structured types (arrays, records) or pointers are displayed with their names and a pair of fold elements which can be expanded to show the components of these types. For example:

```
rec ▶◀
ptr^ ▶◀
arr ▶◀
```

Clicking on one of the triangles shows the components of the corresponding variable. For example, clicking on the fold element after 'arr' leads to:

```
rec ▶◀
ptr^ ▶◀
arr ▷
    0 = 17
    1 = 3
    2 = 25 ◁
```

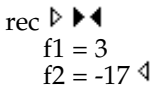
Nested structured types are displayed as nested fold elements. For example, a record with two fields of array type is displayed as:

```
rec ▷
    arr1 ▶◀
    arr2 ▷
        0 = 41
        1 = 35 ◁ ◁
ptr^ ▶◀
arr ▶◀
```

Data structures linked via pointers are displayed with nested fold elements as well. For example:

```
rec ▶◀
ptr^ ▷
    data ▶◀
    next^ ▷
        data ▶◀
        next^ ▷
            data ▶◀
            next^ ▶◀ ◁ ◁ ◁
arr ▶◀
```

A record which is an extension of another record is displayed with fold elements for its base type. For example:



2.4 Limitations

There are some limitations inherent to the structure of the debugger (see also Section 3):

- It is only possible to debug modules, whose source code is available.
- In order to use as little memory as possible, the state of the debugged modules is not saved completely. Therefore, modifying the global state of the debugged modules by commands executed between two debugging steps is potentially dangerous. However, commands not related to the debugged modules may be executed without any special considerations.
- Since a module is loaded into memory as soon as it is registered, the body of a module cannot be debugged.

3. Implementation

The implementation of the debugger consists roughly of three parts. The first part handles the stepwise execution, the second extracts the actual values of variables, and the third part presents the information generated by the other two parts to the user. Each part will be described in its own section. But first, some low level information is presented.

3.1 Compiler support

The debugger mainly does two things: Stepping through a program and displaying the values of variables. To achieve this, it needs information on how to interpret the memory (heap and stack), i.e., it needs the types and addresses of global and local variables, as well as information about statement boundaries to allow stepwise execution. Most debuggers get this information from special files created by the compiler (so called reference files). This has several disadvantages:

- For every compiled module an additional file is needed, even if a module is never debugged.
- The compiler has to be modified in order to create these files.
- The debugger has to read these files and to build a data structure similar to the symbol table built by the compiler.

To circumvent these problems another approach was chosen. The idea was to create the reference information of a module on demand by compiling the module and using the symbol table directly from the compiler. This needs only a minor modification of the compiler. Most of the above problems are avoided, but there are also some disadvantages:

- There is a slight delay when accessing debug information of a module, since the module has to be recompiled first.
- Problems may arise, if the source code of a module is modified after it has been loaded by the Oberon system. The position of the current program counter may be wrong, as its position is still calculated correspondingly to the older source code. However, debuggers using reference files have this problem as well.

The delay for recompiling a module is so short, that it is only noticeable on slow machines (The PowerMac Oberon compiler compiles approx. 2000 lines per second). In addition, the debugger maintains a cache for the debug information of the most recently used modules. The modifications to the compiler are negligible. There are only two necessary changes:

- A pointer to the symbol table must be stored in a global variable in order to prevent the garbage collector from freeing this memory. This can be done in a single line.
- A table must be generated that relates statement boundaries to PC values and vice versa. This results in another 10 lines added to the compiler.

3.2 Stepwise execution

Issuing one of the step commands results in the following actions:

- The code is patched with trap instructions at all statement boundaries.
- The context is switched to the debugged program.
- After a patched trap instruction was reached, the debug context is saved.
- Control is given back to the standard Oberon loop.

3.2.1 Patching

In order to stop execution after a debugging step, trap instructions are inserted into the code segment of the debugged modules. These instructions replace the first machine code instructions of all possibly following Oberon statements. Overwritten machine instructions are saved to a buffer in order to be able to restore them later on (Figure 2).

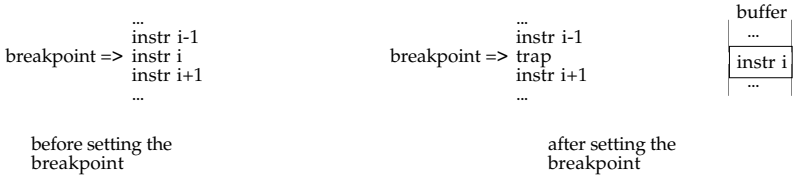


Figure 2: Patching of instruction i

Depending on the chosen step command, the set of following statements differs. Therefore not all statement boundaries are patched with trap instructions.

- *Debug.StepInto* inserts trap instructions at all statement boundaries reported by the compiler. It would be possible to scan the machine instructions to restrict the number of statements, but
- *Debug.Step* inserts trap instructions in all active procedures as well as at all breakpoint positions.
- *Debug.Run* inserts trap instructions at all breakpoint positions.
- *Debug.Return* inserts trap instructions in all active procedures as well as at all breakpoint positions except for the current procedure on the bottom of the stack..

3.2.2 Context switch

There are two kinds of context switches. When the debugged program runs into a trap instruction, control is transferred to the Oberon loop (standard mode). When the user issues a debbuger command such as *Debug.Step*, control is transferred back to the debugged program which is then running in debug mode again. Both context switches are initiated by traps and are performed in the trap handler (Figure 3).

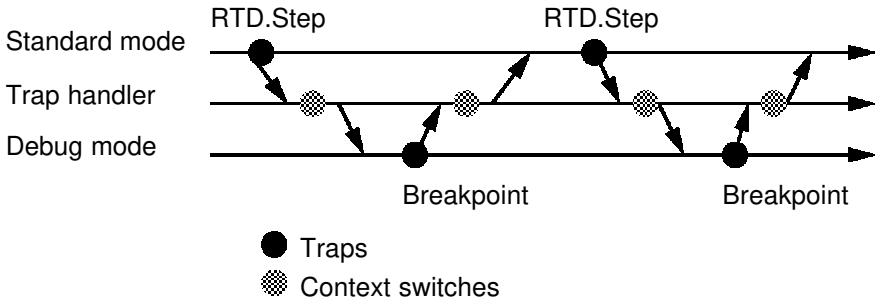


Figure 3: Two sample debugging steps

- **Debug mode to standard mode**: The debugged program runs into a trap instruction (breakpoint). Before control can be tranferred to the Oberon system, the debugger has to save information about the debugged program (stack and registers) and to update the views visible to the user. The update is not done directly in the context switch, but through a more general mechanism, in order to allow later addition of further extensions to the debugger (see also Section 4).

- **Standard mode to debug mode**: This context switch is generated explicitly through the step commands activated by the user. After patching the code with trap instructions (see above), the step command issues a trap to trigger the context switch. Before the actual context switch takes place, the trap handler makes an upcall to the user interface to allow the views to prepare themselves for the next debugging step. Then the trap handler restores the state of the debugged program (stack and registers) and resumes execution at the address where the latest trap occurred. As there may be a trap instruction at this address, execution is resumed in a separate memory location called 'save area' (Figure 4).

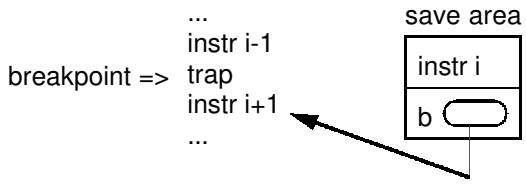


Figure 4: Launching of a debug step

In principle, this area contains two machine instructions: the instruction hidden by the trap instruction and a branch to the instruction following the trap instruction. If the hidden instruction is a relative branch, its target address has to be modified. There are three different cases to be considered (see Figure 5):

- relative branch conditional (bc)
- relative branch unconditional (b), used when a procedure is called
- other instructions

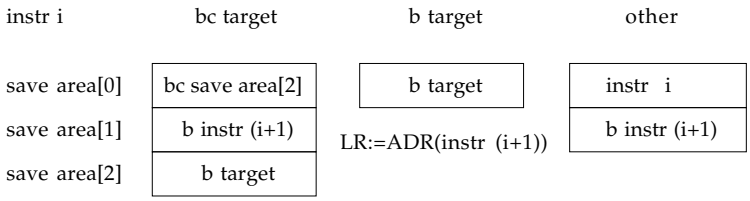


Figure 5: Structure of the save area depending on the kind of the saved instruction

By moving a conditional branch instruction to the save area, its branch distance may exceed the allowable limit. Therefore an indirection through an unconditional branch located in save area[2] is introduced. An unconditional branch overwritten by a trap instruction is typically a procedure call. In order to return correctly from the called procedure, the link register (LR) is set to the instruction following the trap instruction

3.3 Getting the values of variables

To display a variable with its actual value, the debugger needs the following information

- the effective address of the variable
- the variable's actual run time type

The effective address can be calculated from the information provided by the compiler (offset of the variable relative to the base address of the corresponding module or procedure) and the actual base address. If the static type of a variable is a record, the dynamic type may be an extension of it. The dynamic type can be obtained from the record's type descriptor whose address is stored near the effective address of the record. From the address of a variable (and possibly its dynamic type) the value of the variable can be computed (Figure 6).

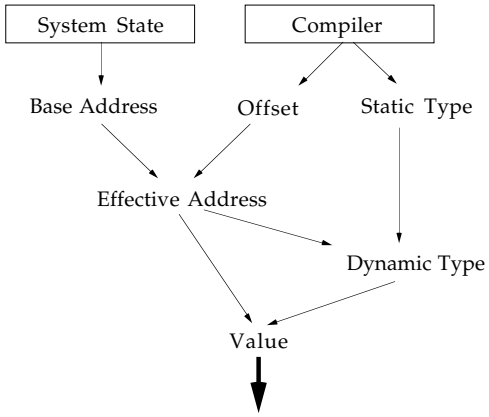


Figure 6: Determine value and run time type of variables

3.4 User interface

We set two main goals for the user interface:

- The debugger should have an Oberon-like 'look and feel'
- The user interface should be extensible

The main idea was to start with a simple and small user interface which may later be extended without changing the debugger (for details see Section 4). The minimal user interface consists of two types of viewers. Source viewers displaying the control flow and data viewers showing the current values of variables. Both views are automatically updated after each debugging step.

3.4.1 Source Viewers

Source viewers are standard text viewers. They are opened automatically by the debugger but can also be opened by the user with *Edit.Open*. To achieve the added functionality, HandlerElems are used (for further information read the file Elem.Guide.Text of the Oberon system). After every debugging step a SetPC message is broadcast. This message contains the name of the module as well as the text position corresponding to the current PC. Upon receiving this message, a text frame displaying the named source text inserts a PC element at the indicated text position and sets a flag in the message. After the broadcast this flag is tested. If it is set, no further action is taken. Otherwise a new viewer is opened showing the desired source text and a PC element is inserted at the appropriate position.

3.4.2 Data Viewers

Data viewers are normal text viewers displaying all variables of a module or a procedure. Fold elements and indentation are used to enhance readability. After every debugging step, the shown values are updated automatically. A view on the data of a no longer active procedure (no active stack frame) or an unloaded module is marked.

4. Extensibility

The user interface of the debugger consists of two parts. On the one hand there is a collection of user-startable actions (commands) and on the other hand there is a set of actions, invoked automatically by the debugger at the beginning and the end of each debugging step. One may use the commands either directly by middle-clicking on them or indirectly through a dialog.

The interface of the debugger is extensible in two ways. First, one can add new commands or dialogs (extensibility inherent to the Oberon system). Second, it is possible to add arbitrary actions that are then activated after every debugging step. In principle, one can add any new or improved feature to the debugger (e.g. a graphical representation of the current object hierarchy). If a procedure named *MyGraphicHandler* should be activated after every debugging step, one has to call

```
RTDT.debugQ.Add (MyGraphicHandler)
```

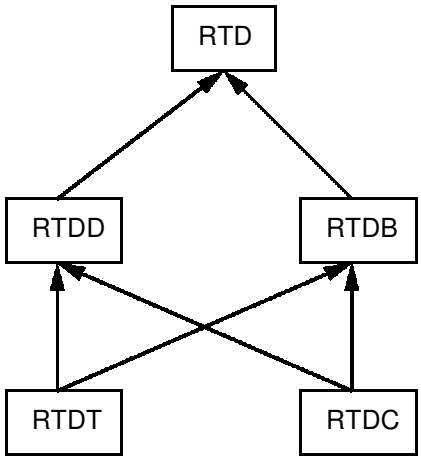
If a procedure name *MyCleanUp* should be called before every debugging step, one has to call

```
RTDT.startQ.Add (MyCleanUp)
```

Currently, there are two installed actions executed after every debugging step (one updates the source viewers and the other updates the data viewers) and one action executed before every debugging step (it deletes the current PC element in the source viewers).

The debugger modules offer several auxiliary procedures which can be used by new extensions. The following sections describe each module's interface, as far as they are important for future extensions.

4.1 Module hierarchy



4.2 RTDT (low-level Trap handling, context switches)

```
DEFINITION RTDT;

    CONST
        EnterDebugMode = 255;

    TYPE
        Proc = POINTER TO ProcDesc;
        ProcDesc = RECORD
            up: Proc;
            pc, sp: LONGINT;
            name: ARRAY 64 OF CHAR;
            modName: ARRAY 32 OF CHAR;
            regs: Sys.ExceptionInfo;
            beginPC, endPC: LONGINT;
        END ;

    VAR
        debugQ-: Kernel.Queue;
        startQ-: Kernel.Queue;
        procs-: Proc;

    PROCEDURE Debugging (): BOOLEAN;
    PROCEDURE FindProc (startPC: LONGINT; VAR name: ARRAY OF CHAR);
    PROCEDURE Launching (): BOOLEAN;
    PROCEDURE PC (): LONGINT;
    PROCEDURE PopProc;
    PROCEDURE SearchProc (pc: LONGINT; VAR mod: Modules.Module;
                          VAR refpos, refend, startpc, endpc: LONGINT);
END RTDT.
```

Constants

- *EnterDebugMode* is the trap number used to initiate a context switch to the debug mode. To write an own command which initiates a debugging step, set the desired breakpoints and call *HALT(EnterDebugMode)* in order to resume the execution of the debugged modules.

Types

- *ProcDesc* represents a procedure with an activation frame on the debugging stack. It contains the following information:
  - *up* is a link to the procedure's caller.
  - *pc* is the address of the latest executed instruction (of this procedure).
  - *sp* is the procedure's stack pointer.
  - *name* is the procedure's name
  - *modNam*e is the name of the module containing the procedure.
  - *regs* are the procedure's registers.
  - *beginPC* and *endPC* define the address and size of the procedure's code in memory.

Global variables

- *debugQ*: Procedures registered in *debugQ* are called whenever a debugging step has finished (e.g. useful to update information on the screen).
- *startQ*: Procedures registered in *startQ* are called whenever a debugging step is about to be started (e.g. to remove temporary information visible on screen).
- *procs*: Points to the list of procedures currently on the debugging stack.

Operations

- *Debugging ()* returns TRUE if a module is currently debugged.
- *FindProc (startPC, name)* returns the name of the procedure starting at the address *startPC*.
- *Launching ()* returns TRUE if modules have been registered for debugging but have not yet been entered, i. e. there exists no saved state of the debugged modules.
- *PC ()* returns the program counter at the end of the latest debugging step.
- *PopProc* removes the bottom most activation frame from the debugging stack.
- *SearchProc (pc, mod, refStart, refEnd, startPC, endPC)* returns information about the procedure which contains the address *pc*. *refStart* and *refEnd* define the address and size of the procedure's reference information and *startPC* and *endPC* the address and size of its code.

4.3 RTDB (Breakpoint handling)

```
DEFINITION RTDB;

    TYPE
        GetBPMsg = RECORD (Display.FrameMsg) END ;

        ModuleInfo = POINTER TO ModuleInfoDesc;
        ModuleInfoDesc = RECORD
            next-: ModuleInfo;
            name-: ARRAY 32 OF CHAR;
        END ;

    VAR
        modules-: ModuleInfo;

    PROCEDURE AddModules (s: Texts.Scanner);
    PROCEDURE BreakAll;
    PROCEDURE BreakAtPC (VAR name: ARRAY OF CHAR; pos: LONGINT): BOOLEAN;
    PROCEDURE CleanUp;
    PROCEDURE EntryAll;
    PROCEDURE PCToPos (pc: LONGINT; VAR name: ARRAY OF CHAR;
                       VAR pos: LONGINT);
    PROCEDURE PosToPC (VAR name: ARRAY OF CHAR; pos: LONGINT;
                       VAR pc: LONGINT);
    PROCEDURE RestoreAll;
    PROCEDURE StepAll;
    PROCEDURE StepOverAll;

END RTDB.
```

Types

- A *GetBPMsg* is broadcast in order to collect user defined breakpoints. Upon receiving this message, a handler of a text frame with breakpoints has to register them with help of the operation *BreakAtPC*.

Global variables

- *modules* points to a list of all debugged modules.

Operations

- *AddModules (s)* reads names using the scanner *s* as long as they are valid. Each name is treated as the name of a module, which is then entered into the list of debugged modules.
- *BreakAll* inserts trap instructions for user-defined breakpoints in all debugged modules. Trap instructions for stepwise execution are previously removed.
- *BreakAtPC (name, pos)* inserts a trap instruction for the user-defined breakpoint in module *name* at text position *pos*. Should always be called upon receiving a *GetBPMsg*.
- *CleanUp* removes all patches in the debugged modules and clears the list of debugged modules.
- *EntryAll* inserts trap instructions at all entry points of all debugged modules.
- *RestoreAll* removes all patches in the debugged modules.
- *StepAll* inserts trap instructions at all statement boundaries in all debugged modules.
- *StepOverAll* inserts trap instructions in all procedures with an activation frame on the debugging stack.
- *PCToPos (pc, name, pos)* returns the module name and the text position corresponding to the address *pc*.
- *PosToPC (name, pos, pc)* returns in *pc* the code address corresponding to the text position *pos* in the module *name*.

4.4 RTDC (Compiler Interface)

```
DEFINITION RTDC;

    TYPE
        ScanProc = PROCEDURE (obj: POPT.Object);

        Stat = RECORD
            pc-: POINTER TO ARRAY OF SHORTINT;
            pos-: POINTER TO ARRAY OF LONGINT;
        END ;

        Sym = POPT.Object;
        Type = POPT.Struct;

    PROCEDURE FindProc (proc: RTDT.Proc): Sym;
    PROCEDURE FindType (type: Types.Type; module: Modules.Module): Sym;
    PROCEDURE ScanScope (scope: Sym; proc: ScanProc);
    PROCEDURE Statements (name: ARRAY OF CHAR; VAR stats: Stat);
    PROCEDURE Symbols (name: ARRAY OF CHAR; VAR syms: Sym);

END RTDC.
```

Types

- *Stat* contains two open arrays containing the statement boundaries as code addresses and as text positions. These arrays are used for translating code addresses to text positions and vice versa. The text position *pos[i]* corresponds to the code address 4*Sum (*pc[0]..pc[i]*).

- *Sym* denotes an Oberon symbol (variable, procedure).
- *Type* denotes a type definition as given by the compiler.

Operations

- *FindProc (proc)* returns the symbol information for the procedure *proc*.
- *FindType (type, module)* returns the symbol information for a given type. If its module's source code is not accessible, the information is searched in the module *module*.
- *ScanScope (scope, proc)* calls the procedure *proc* for all local objects of the procedure or module *scope*.
- *Statements (name, stats)* retrieves the statement boundaries for the module *name*. If they could not be determined, *stats* is NIL.
- *Symbols (name, syms)* retrieves the symbol information for the module *name*. If an error occurs, *syms* is NIL.

4.5 RTDD (Data viewers)

```
DEFINITION RTDD;

   PROCEDURE GetLocalScope (VAR w: Texts.Writer; t: Texts.Text; syms: POPT.Object;
                            reg: Sys.ExceptionInfo; indent: INTEGER);
   PROCEDURE GetScope (VAR w: Texts.Writer; t: Texts.Text; syms: POPT.Object;
                       baseAdr: LONGINT; indent: INTEGER);

END RTDD.
```

Operations

- *GetLocalScope (w, t, syms, reg, indent)* writes all local objects of a procedure to the writer *w* using the given indentation. The procedure is defined by its symbols *syms* and the register set *reg*. When a structured type is encountered (use of fold elements), the text *t* is scanned for the corresponding fold element. If it is found and it is open, it is automatically expanded when written to *w*. Therefore a fold element in a data viewer stays open, if a step command is executed.
- *GetScope (w, t, syms, baseAdr, indent)* writes all local objects of a module to the writer *w* using the given indentation. The module is defined by its symbols *syms* and the base address *baseAdr*. When a structured type is encountered (use of fold elements), the text *t* is scanned for the corresponding fold element. If it is found and it is open, it is automatically expanded when written to *w*. Therefore a fold element in a data viewer stays open, if a step command is executed.

5. Conclusions

The strength of our debugger stems from its concepts being based on the Oberon environment. The debugger has the following main features:

- It needs no reference files.
- It is small and efficient by reusing many features of the Oberon system (the total object code size is about 30 kBytes).
- It fully supports the Oberon concept of multiple entry points into an application.
- It is extensible.

Context switches in general and hand-made code pieces in particular are highly system-dependent and processor-dependent. The debugger is currently implemented for the Power PC. This implies a high degree of non portability. However porting it to a processor with an architecture similar to the Power PC should not be too hard and time consuming.

6. References

[Ebe87] Hans Eberle: Development and Analysis of a Workstation Computer. PhD thesis, Swiss Federal Institute of Technology (ETH Zürich), 1987. Number 8431.

[Hof94] Markus Hof: Post Mortem Debugger for Oberon. Proc. 1994 Joint Modular Languages Conference.

[Kna94] Markus Knasmüller: Oberon Dialogs: User's Guide and Programming Interfaces. Institute for computer science (system software), University of Linz, Report No. 1, November 1994.

[MoK95] H.Mössenböck, K.Koskimies: Active Texts for Structuring and Understanding Source Code. Paper submitted to Software – Practice and Experience.

[Wir89] N. Wirth, J. Gutknecht: The Oberon System. Software-Practice and Experience, 19(9), 1989, 857-893.