XDS Family of Products

Native XDS-x86 for Microsoft Windows
Version 1.1

XDS Debugger User’s Guide

http://www.excelsior-usa.com
Copyright © 1999-2011 Excelsior LLC. All rights reserved.

Information in this document is subject to change without notice and does not represent a commitment on the part of Excelsior, LLC.

Excelsior’s software and documentation have been tested and reviewed. Nevertheless, Excelsior makes no warranty or representation, either express or implied, with respect to the software and documentation included with Excelsior product. In no event will Excelsior be liable for direct, indirect, special, incidental or consequential damages resulting from any defect in the software or documentation included with this product. In particular, Excelsior shall have no liability for any programs or data used with this product, including the cost of recovering programs or data.

XDS is a trademark of Excelsior LLC.

Microsoft, Windows 2000, Windows XP, Windows Vista are either registered trademarks or trademarks of Microsoft Corporation.

All trademarks and copyrights mentioned in this documentation are the property of their respective holders.
# Contents

1 Introduction ............................................. 1
   1.1 New in version 1.1 ................................. 1

2 Getting started ........................................ 3
   2.1 General considerations .......................... 3
      2.1.1 Executable lines .......................... 3
      2.1.2 Procedure prologue and epilogue ........ 4
      2.1.3 Register variables ......................... 4
      2.1.4 Program startup .......................... 4
   2.2 Preparing a program for debugging ............. 5
      2.2.1 Debug information formats ................ 5
   2.3 Starting XD ....................................... 5
   2.4 Terminating XD ................................... 6

3 Using the dialog mode ................................ 7
   3.1 Introduction ..................................... 7
      3.1.1 Using menus and shortcuts ................. 8
      3.1.2 Using information windows ................ 9
      3.1.3 Using dialogs .............................. 9
      3.1.4 Managing windows ......................... 9
      3.1.5 Refreshing the screen ..................... 10
      3.1.6 Viewing batch mode output ............... 10
      3.1.7 Smart session select ...................... 10
   3.2 Files used by XD ................................ 11
   3.3 Loading a program ................................ 12
   3.4 Navigating through your program ............... 12
      3.4.1 Cursors .................................... 12
      3.4.2 View modes ................................ 13
      3.4.3 Moving around in the Program window .... 14
      3.4.4 The Search menu ........................... 15
      3.4.5 The Components window .................... 16
3.4.6 The Modules window ........................................... 16
3.4.7 The Procedures window ....................................... 16
3.4.8 The Publics window ............................................. 17

3.5 Executing your program ............................................ 17
  3.5.1 Run ............................................................... 18
  3.5.2 Execution Trace ............................................... 18
  3.5.3 Step into ....................................................... 19
  3.5.4 Step over ....................................................... 19
  3.5.5 Step out ........................................................ 19
  3.5.6 Run to cursor .................................................. 19
  3.5.7 Run to epilogue ............................................... 20
  3.5.8 Run until address .............................................. 20
  3.5.9 Restart ........................................................ 20
  3.5.10 Restart at startup .......................................... 20
  3.5.11 Restart at entry point ..................................... 21
  3.5.12 The Call Stack window ..................................... 21
  3.5.13 The Threads window ........................................ 21

3.6 Using breakpoints and breaks ..................................... 21
  3.6.1 Sticky breakpoint ............................................. 23
  3.6.2 Breakpoint ..................................................... 24
  3.6.3 Delayed sticky breakpoints .................................. 24
  3.6.4 Delayed breakpoint .......................................... 24
  3.6.5 Expression breakpoint ....................................... 25
  3.6.6 Pass counter ................................................... 25
  3.6.7 Watchpoint ..................................................... 25
  3.6.8 Access break .................................................. 26
  3.6.9 Condition break ................................................ 26
  3.6.10 The Breaks window ......................................... 27
  3.6.11 Managing breaks and breakpoints .......................... 27

3.7 Examining your program data ....................................... 27
  3.7.1 Displaying and modifying values ............................. 29
  3.7.2 Examine variable .............................................. 29
  3.7.3 Evaluate expression .......................................... 30
  3.7.4 The Global variables window ................................ 30
  3.7.5 The Module variables window ................................ 30
  3.7.6 The Local variables window ................................ 30
  3.7.7 Memory dump windows ....................................... 31
  3.7.8 The Registers window ......................................... 31
  3.7.9 The Float Registers window .................................. 32
  3.7.10 The Stack window ............................................ 32
  3.7.11 The Watches window ......................................... 32
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.8</td>
<td>Saving scripts</td>
<td>33</td>
</tr>
<tr>
<td>3.9</td>
<td>Setting XD options</td>
<td>33</td>
</tr>
<tr>
<td>3.9.1</td>
<td>Changing window colors</td>
<td>34</td>
</tr>
<tr>
<td>3.9.2</td>
<td>Modifying keyboard shortcuts</td>
<td>35</td>
</tr>
<tr>
<td>3.9.3</td>
<td>Modifying frames</td>
<td>36</td>
</tr>
<tr>
<td>4</td>
<td>Using the batch mode</td>
<td>37</td>
</tr>
<tr>
<td>4.1</td>
<td>Control file syntax</td>
<td>37</td>
</tr>
<tr>
<td>4.2</td>
<td>Load program statement</td>
<td>40</td>
</tr>
<tr>
<td>4.3</td>
<td>Control flow statements</td>
<td>40</td>
</tr>
<tr>
<td>4.4</td>
<td>Program execution statements</td>
<td>45</td>
</tr>
<tr>
<td>4.5</td>
<td>Data management statements</td>
<td>49</td>
</tr>
<tr>
<td>4.6</td>
<td>Log control statements</td>
<td>51</td>
</tr>
<tr>
<td>4.7</td>
<td>Miscellaneous statements</td>
<td>56</td>
</tr>
<tr>
<td>5</td>
<td>Expressions</td>
<td>59</td>
</tr>
<tr>
<td>5.1</td>
<td>Constants</td>
<td>60</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Whole constants</td>
<td>60</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Real constants</td>
<td>60</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Boolean constants</td>
<td>60</td>
</tr>
</tbody>
</table>
CONTENTS

5.1.4 Character constants .................................. 61
5.2 CPU registers ............................................ 61
5.3 Referencing program entities .............................. 61
5.4 Operators .................................................. 62
5.4.1 Unary operators ........................................ 62
5.4.2 Binary operators ....................................... 62
5.4.3 Comparison operators .................................. 62
5.5 Built-in functions ......................................... 63
5.5.1 ADR - address of a memory object ..................... 63
5.5.2 SIZE - size of a memory object ....................... 63
5.5.3 LOW - low index of an array ......................... 63
5.5.4 HIGH - high index of an array ....................... 63
5.5.5 @PASS - number of passes for breakpoint ............ 64
5.5.6 @ARG - access to command-line arguments ........... 64
5.6 Type casting ................................................. 64

6 Remote debugging ............................................. 67
6.1 Preparing for remote debugging ......................... 67
6.2 Starting debug server .................................... 67
6.3 Starting XD in remote mode ............................... 68

7 Keys reference .................................................. 69
7.1 General ..................................................... 69
7.2 Window management ....................................... 69
7.3 Program window .......................................... 70
7.4 Information windows ...................................... 70
7.5 Breaks ..................................................... 71
7.6 Execution .................................................. 71
7.7 Watches ..................................................... 71
7.8 Function keys .............................................. 72

A The HIS utility ................................................ 73
Chapter 1

Introduction

From its first release in 1991, the XDS\textsuperscript{1} family of Modula-2/Oberon-2 development systems provided a set of basic debugging facilities, such as execution history and profiling. When it came to symbolic debugging, however, it relied on an "underlaying" C compiler package. Nothing was wrong about it, as long as XDS compilers were implemented as translators to ANSI C.

For owners of the first native code XDS compilers for Intel x86 line of processors, launched in the second half of 1996, having a C compiler is no longer essential. Nevertheless, XDS understands that for any serious development a symbolic debugger is an absolute requirement.

The \textbf{XDS Debugger} (XD) pretends to meet this requirement. It is a full source level symbolic debugger which provides common debugging facilities, such as execution in step mode, breakpoints and breaks setting, data examination and modification, etc.

The debugger may be used in two modes, dialog and batch. The dialog mode is interactive, while in the batch mode XD may automatically execute a debugging scenario (or a number of scenarios) described in special control files, and log the scenario execution results. This allows the debugger to be used to automate testing.

1.1 New in version 1.1

The following changes and improvements have been made to XD since version 1.0:

\textsuperscript{1}Previous versions or XDS were known as Extacy and OM2
CHAPTER 1. INTRODUCTION

- Improved locators (see 3.1.2)
- 16-bit character type and 64-bit integer type (Java char and long int) are partially supported
- Exceptions generated by the XDS run-time library (type guards, array index checks, etc.) are now detected
- The program may be restarted at entry point (see 3.5.11)
- A counter may be turned into a delayed breakpoint (see 3.6.3)
- The Float Registers window (see 3.7.9)
- Constants and symbols are displayed according to rules of the respective language (e.g. a null pointer is displayed as NIL in Modula-2 variables and as null in Java variables)
- Global variables of any module can be displayed in a separate window (see 3.4.6, 3.7.5)
- Local variables and parameters from any procedure’s stack frame can be displayed in the Local variables window (see 3.7.6)
- A memory dump may be saved to file (see 3.7.7)
- Watch expressions are saved in session files
- Detection of the actual dynamic type of an Oberon-2 pointer or a Java object (see 3.9)
- Debug scenario record and playback (see ??)
- Symbolic names may be used instead of numbers to identify breaks in the batch mode (see 4.4)
- All types of breakpoints may now be set in the batch mode (see 4.4)
- In the batch mode, breakpoints may be set at a procedure’s prologue, body, epilogue, or return instruction (see 4.4)
- Command-line arguments may now be referenced in control files (see 5.5.6)
- Remote debugging (see Chapter 6) is now supported
- Support for XD has been added to the HIS utility (see Appendix A)
Chapter 2

Getting started

2.1 General considerations

This section introduces some terms and principles to help you better understand the debugger.

2.1.1 Executable lines

In XD, a source line is considered executable if there are CPU instructions corresponding to it. Breakpoints (see 3.6) may be placed only on executable lines, only executable lines are displayed in the Mixed mode (see 3.4.2), etc. Obviously, source lines containing comments or declarations are not executable. However, native code XDS compilers may generate no code even for lines containing statements, because of optimization or, for instance, if that code would be unreachable anyway:

```
PROCEDURE P(i : CARDINAL): CARDINAL;
BEGIN
  CASE i OF
    |0: RETURN 1
    |1: RETURN P(i-1)
  END;
  RETURN 2 (* Warning: Unreachable code *)
END P;
```

In the above example, passing any value other than 0 or 1 as an input parameter to \( P \) would cause the language exception `caseSelectException` to be raised.
Hence, the code corresponding to the last \texttt{RETURN} statement would never be executed and is omitted in the resulting object file. The "Unreachable code" warning is issued during compilation.

### 2.1.2 Procedure prologue and epilogue

In addition to the code correspondent to a procedure’s body, a compiler usually inserts some initialization (so called \textit{prologue}) after its entry point and some cleanup (\textit{epilogue}) before the \texttt{RET} instruction. In particular, the XDS compiler may generate code which saves and restores stack frame, installs and removes exception handlers, etc.

The compiler maps a prologue to the source line containing the procedure’s \texttt{BEGIN}, and an epilogue to the line containing \texttt{END}.

### 2.1.3 Register variables

Even if you turn the \texttt{NOOPTIMIZE} option on, the compiler still performs some optimizations. In particular, it may place local variables onto CPU registers. The debugger prepends names of register variables with an exclamation mark in information windows (see 3.1.2). Such variables may not exist at certain points of procedure execution, so the displayed values may be incorrect and it is not possible to modify a register variable.

When register variables are used in expressions (see Chapter 5), they are treated as unsigned integers.

### 2.1.4 Program startup

In a C program, the \texttt{main()} function has the public name \texttt{main} or \texttt{_main}. To ensure compatibility with third-party debuggers and other tools, XDS compilers assign one of these names to the main module body entry point, depending on the \texttt{CC} option setting. After the program is loaded, XD searches its symbol table for these symbols and automatically executes the program’s startup code up to the point the located symbol refers to. If you wish to debug the startup code, use the \texttt{Restart at startup} (see 3.5.10) command or specify the \texttt{/s} option in XD command line.

If the program does not have a symbol table or there is no \texttt{main} or \texttt{_main} symbol in the code segment, the program will be stopped at the first possible point, as if
2.2. PREPARING A PROGRAM FOR DEBUGGING

Before invoking the debugger, ensure that your program is built with line number and symbolic debug information. To do this, switch the LINENO and GENDEBUG options on in the project file or on the command line. You may also wish to switch on the NOOPTIMIZE option in order to disable the machine-independent optimizer. Use the =ALL submode to force all modules which constitute your program to be recompiled with these option settings.

Do not forget to specify a linker option (/DEBUG for XDS Linker) which would cause debugging information to be bound to the executable. If you are taking advantage of the XDS make facility, it will be done automatically if you specify the GENDDEBUG compiler option, provided that you use a template file bundled with your XDS package. Refer to an appropriate documentation if you use a third-party linker which is not supported in standard template files.

2.2.1 Debug information formats

The Native XDS-x86 compiler is capable to produce debug information in two formats: CodeView (default) and HLL4. This is controlled by the DBGFMT compiler option. The HLL4 format is more suitable for Modula-2/Oberon-2 programs. For instance, it allows to represent sets and enumerations, which are not supported in CodeView. XDS Linker and XDS Debugger accept both formats on both platforms, so use HLL4 whenever possible.

HLL4 support under Windows is probably an unique feature, so if you use any third-party tools, CodeView is your only option.

2.3 Starting XD

To invoke XD from the command line, type

```
xd [<options>] [<executable> [<arguments>]]
```

or

```
xd /b [<options>] <control file>
```
and press Enter. Available options are:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/D</td>
<td>Invoke XD in dialog mode (default)</td>
</tr>
<tr>
<td>/B</td>
<td>Invoke XD in batch mode</td>
</tr>
<tr>
<td>/H</td>
<td>Print a brief help message</td>
</tr>
<tr>
<td>/J</td>
<td>Safe log</td>
</tr>
<tr>
<td>/L</td>
<td>Create log file</td>
</tr>
<tr>
<td>/N</td>
<td>Display file names without path</td>
</tr>
<tr>
<td>/S</td>
<td>Debug startup code</td>
</tr>
<tr>
<td>/Xnnn</td>
<td>Set width of XD session window to nnn</td>
</tr>
<tr>
<td>/Ynnn</td>
<td>Set height of XD session window to nnn</td>
</tr>
</tbody>
</table>

If you invoke XD without any parameters, it will start in dialog mode and display the Load Program dialog.

### 2.4 Terminating XD

To exit the dialog mode, select Exit from the File menu or press Alt-X. If the debugger was started in the dialog mode, it will terminate. Otherwise (the dialog mode was activated from a control file), control file processing will continue.

To interrupt XD running in the batch mode, press Ctrl+C.
Chapter 3

Using the dialog mode

3.1 Introduction

XD is a text mode application and usually operates in a Win32 console window. It may also be started in fullscreen mode. In the following text, screen refers to either fullscreen or windowed console containing XD.

The different types of information displayed by the debugger are collected in separate windows. There are five kinds of windows in XD:

- menus
- the Program window
- information windows (or just windows)
- dialogs
- message boxes

The top line of XD screen is occupied by the Main menu, which provides access to other menus. The rest of the screen is shared by the Program window and information windows. All those windows are overlapped and may be freely moved, resized, and placed above or beneath each other.

The Program window contains the source text and/or the disassembled code of the program being debugged (the debuggee). It is frameless and may not be closed or hidden.
Information windows are used to display various data about the current state of the debuggee, such as watch expressions, CPU registers, call stack, etc. They have frames and may be displayed, hidden, or closed at any time.

A dialog is a window which pops up when XD prompts you for an information it needs to complete a requested action, such as a name of a variable to examine, or an expression to watch. Dialog windows are modal: another window may not be activated while the dialog is displayed and you have to explicitly confirm or cancel the action you requested.

The debugger displays error, warning, and informational messages in message boxes. They are similar to dialogs, but the only action you may perform when a message box is displayed is to close it.

### 3.1.1 Using menus and shortcuts

To select an item from the Main menu, do one of the following:

- Click an item with the left mouse button.
- Hold down Alt and press a highlighted character of an item.
- Press F10, choose an item with left/right arrow keys, and press Enter or the down arrow key.

This will cause one of the pulldown menus to be displayed. To select an item from a pulldown menu, do one of the following:

- Click an item with the left mouse button.
- Press a highlighted character (holding down Alt is optional).
- Choose an item with up/down arrow keys and press Enter.

Finally, there are pop-up menus available in most information windows. To activate a pop-up menu, click an information window with the right mouse button or press Ctrl+F10. Then select an item as in a pulldown menu.

The most often used menu commands have shortcuts - key combinations which may be used to activate a menu command without selecting it from a menu. Shortcuts are shown to the right of menu items.
3.1.2 Using information windows

Most information windows contain lists of items, such as global variables or watch expressions. When a window becomes current, a cursor is displayed over one of its items. The following keys may be used to move the cursor:

<table>
<thead>
<tr>
<th>Key</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left/Right</td>
<td>scroll window contents horizontally</td>
</tr>
<tr>
<td>Up/Down</td>
<td>move the cursor one line up/down</td>
</tr>
<tr>
<td>PgUp/PgDn</td>
<td>move the cursor one page up/down</td>
</tr>
<tr>
<td>Ctrl+PgUp/PgDn</td>
<td>move the cursor to the first/last item of the list</td>
</tr>
<tr>
<td>Ctrl+Home/End</td>
<td>move the cursor to the top/bottom of the window</td>
</tr>
</tbody>
</table>

An item may also be selected by clicking it with the left or right mouse button. In the latter case, a pop-up menu will also be displayed.

Information windows which contain lists of program entities, for instance, a list of all modules or a list of local variables provide locators for quick cursor positioning.

To move the cursor to a certain item, start typing item’s name on the keyboard. The characters you type will appear on the bottom line of a window frame (the locator), and the cursor will position on the first item which name begins with those characters. You may use the Tab key to quickly expand the current value of a locator.

Once the cursor appears near the desired item, you may use arrow keys again to complete positioning. Character case is ignored during search, and case of characters in a locator always match the currently selected item.

3.1.3 Using dialogs

A dialog is a modal window which pops up when you initiate a certain action which requires parameters, such as variable name or address.

XD dialogs behave very similar to Windows dialogs.

3.1.4 Managing windows

At any moment during a debugging session, there is exactly one window which accepts user input - the current window. It is the topmost and is displayed with a
CHAPTER 3. USING THE DIALOG MODE

double frame instead of a single\(^1\).

To select a window to become current, click it or press Ctrl+Tab to cycle through windows. You may also select a window from the Windows menu, which always contains a list of all open windows. You may need to scroll it if there are too many windows open.

To move or resize the current window using the keyboard, press Ctrl+W. A bold frame will be drawn around the window. Use arrow keys to move the window. Holding down Ctrl while pressing arrow keys will force the window to change its size instead of position. To leave the move/resize mode, press Ctrl+W or Enter.

With the mouse, drag the top border of a window to move it or drag any other border or a corner to resize.

To close the current window, press Ctrl+F4 or click the \([\times]\) sign in its top right corner. It is also possible to close a window by selecting it in the Windows menu using arrow keys and pressing Del. The Program window may not be closed.

### 3.1.5 Refreshing the screen

If there is any garbage on XD screen, press Ctrl+E or select Refresh screen from the File menu to have all windows repainted.

### 3.1.6 Viewing batch mode output

If the debugger was invoked in the batch mode (see Chapter 4) and then switched to the dialog mode with the help of the DIALOG command (see 4.3), you may wish to look at the last lines of the log. To do this, press "/" or select Show output from the File menu.

### 3.1.7 Smart session select

Your program is likely to require user interaction. It would be convenient to have the session in which it is executed automatically brought to foreground (selected) during execution, and XD session to be selected whenever the program is stopped. However, in case of step execution modes (see 3.5) usage that would cause undesirable flickering.

---

\(^1\)The frameless Program window is marked as current by changing its header colour.
XD uses a bit smarter approach: after resuming the debuggee execution, it waits for a small amount of time before selecting the debuggee session. If control returns to the debugger prior to timeout, its session remains selected. The **Delay to select debuggee** option (see 3.9) allows you to adjust the timeout period.

When your program is stopped, you may temporarily bring it to foreground by pressing "\" or by selecting **Show debuggee** from the File menu. After a timeout, the XD session will be automatically selected again. A timeout does not occur if you select the debuggee using an operating system mechanism.

### 3.2 Files used by XD

The following files are used by the debugger:

- **xd.red** redirection file
- **xd.cfg** default session settings
- **xd.hlp** on-line help
- **xd.msg** debugger messages
- ***.ses** session settings

The debugger uses a redirection file to search for all other files the same way the compiler does, except that source files are searched via redirection only if the **Strip path from source name** option (see 3.9) is switched ON.

The **xd.cfg**, **xd.hlp**, and **xd.msg** files, if not found via redirection, are expected to be found in the directory in which XD executable file, **xd.exe**, resides. The **xd.msg** file is essential; the debugger will display an error message and terminate if it fails to locate it on startup.

The configuration file contains default values of options, window sizes, positions, and colors.

Upon termination of a dialog debugging session, the debugger stores its settings in a session file, provided that the **Save debug session layout** option (see 3.9) is set ON. A session file name matches the debugged executable name, and has the extension **.ses**. Next time you debug that executable, XD will restore option values and window layout from the session file.

Watch expressions (see 3.7.11), if any, are also saved in session files.
3.3 Loading a program

To have a program which you want to debug automatically loaded when you invoke the debugger, pass its executable name and optional arguments on XD command line (see 2.3). If you do not provide this information, XD will display the Load Program dialog upon startup.

To load a program via the Load Program dialog:

1. Type the name of the executable in the Program name entry field or select the Browse button to select it using a file dialog.

2. If you want any command line arguments to be passed to the debuggee, type them in the Arguments entry field.

3. Select the OK pushbutton or press Enter.

If you decided not to load a program, select the Cancel pushbutton or press Esc.

To load another program during a debugging session, select Load Program from the File menu or press F3.

3.4 Navigating through your program

XD displays the program you are debugging in the Program window. This window is automatically opened upon program load and remains on the screen until you finish the debugging session.

The Program window is frameless. The top line is a header in which information about the current position in the code is displayed. The two leftmost columns are reserved for breakpoint marks (see 3.6) and are not scrolled horizontally. The remaining area of the window is used to display the program source and/or the disassembled code.

3.4.1 Cursors

The Program window contains two cursors, displayed in different colors.

When the debuggee is stopped, the execution cursor is positioned over the line or instruction about to be executed. It may not be moved manually.
The *user cursor* is intended for navigation. You may freely move it through the source text or code, which is scrolled, if necessary, so that the user cursor always stays visible. It is also used to specify a breakpoint position. See 3.6 for more information.

When the two cursors coincide, only the execution cursor is displayed. The user cursor is moved to the position of the execution cursor when the debuggee execution is interrupted, for instance, when a step command is executed or a breakpoint is hit. You may also quickly move the user cursor to the position of the execution cursor by pressing **Ctrl+Home** or selecting **Home** from the **Search** menu.

### 3.4.2 View modes

The **Program** window supports three view modes: **Source**, **Disassembly**, and **Mixed**.

In the **Source** mode, the **Program** window contains source code of one of the modules which constitute your program. Its name is displayed in the header along with the number of the source line currently selected by the user cursor.

Two different colors are used to display executable and non-executable lines, unless the **Code highlight** option (see 3.9) is set to **No**. When the user cursor is positioned over an executable line, a corresponding address and a procedure name are displayed in the window header.

In the **Disassembly** mode, the code of your program is displayed as assembly instructions. A name of a procedure to which the instruction under the user cursor belongs and offset of this instruction from the procedure’s entry point are shown in the window header. In this mode, you may scroll through the entire code segment, while in two other modes module boundaries may not be crossed.

The **Mixed** mode is similar to the **Disassembly** mode, with addition that the correspondent source line is displayed above each chunk of disassembled code. The source lines, however, are treated as comments; the user cursor jumps over them as you move it, so it is not possible to step execute an entire line or set a breakpoint on it.

**Note:** In the **Source** and **Mixed** modes, only one of the program’s modules is displayed in the **Program** window. Use the **Modules** window (see 3.4.6) to display another module.

After loading a program, XD always enters the **Source** mode, unless there is no debug info at the execution point. If a source file can not be found, the window is filled with lines containing the corresponding message.
CHAPTER 3. USING THE DIALOG MODE

To toggle view modes, select a corresponding item from the Code menu, from the pop-up menu, or use the following shortcuts:

<table>
<thead>
<tr>
<th>Shortcut</th>
<th>Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alt+1</td>
<td>switch to the Source mode</td>
</tr>
<tr>
<td>Alt+2</td>
<td>switch to the Disassembly mode</td>
</tr>
<tr>
<td>Alt+3</td>
<td>switch to the Mixed (source+disassembly) mode</td>
</tr>
</tbody>
</table>

3.4.3 Moving around in the Program window

The following keys move the user cursor through the contents of the Program window:

<table>
<thead>
<tr>
<th>Key Combination</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left/Right</td>
<td>scroll window contents horizontally</td>
</tr>
<tr>
<td>Up/Down</td>
<td>move the user cursor one line up/down</td>
</tr>
<tr>
<td>PgUp/PgDn</td>
<td>move the user cursor one page up/down</td>
</tr>
<tr>
<td>Ctrl+PgUp/PgDn</td>
<td>move the user cursor to the top/bottom of the current module</td>
</tr>
<tr>
<td>Ctrl+Home/End</td>
<td>move the user cursor to the top/bottom of the window</td>
</tr>
<tr>
<td>Home/End</td>
<td>show the current line beginning/end</td>
</tr>
<tr>
<td>Ctrl+Up/Down</td>
<td>move the user cursor to the previous/next procedure in the current module</td>
</tr>
<tr>
<td>Ctrl+H</td>
<td>move the user cursor to the execution cursor position</td>
</tr>
<tr>
<td>Ctrl+G</td>
<td>move the user cursor to a source line by its number</td>
</tr>
<tr>
<td>Ctrl+A</td>
<td>move the user cursor to a certain address</td>
</tr>
<tr>
<td>Ctrl+F</td>
<td>find a string</td>
</tr>
<tr>
<td>Ctrl+N</td>
<td>repeat last find</td>
</tr>
</tbody>
</table>

Some of the key combinations listed above are shortcuts for the Search menu (see 3.4.4) commands.

In addition, the following information windows may help you to quickly move to certain points in the code:
3.4. NAVIGATING THROUGH YOUR PROGRAM

<table>
<thead>
<tr>
<th>Window</th>
<th>May be used to</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modules (see 3.4.6)</td>
<td>display another module</td>
</tr>
<tr>
<td>Procedures (see 3.4.7)</td>
<td>display a procedure in the current module</td>
</tr>
<tr>
<td>Publics (see 3.4.8)</td>
<td>display a public procedure</td>
</tr>
<tr>
<td>Call Stack (see 3.5.12)</td>
<td>display a call chain element</td>
</tr>
<tr>
<td>Breaks (see 3.6.10)</td>
<td>display code where a breakpoint is set</td>
</tr>
<tr>
<td>Components (see 3.4.5)</td>
<td>switch to another component</td>
</tr>
<tr>
<td>Threads (see 3.5.13)</td>
<td>switch to another thread</td>
</tr>
</tbody>
</table>

3.4.4 The Search menu

The Search menu commands may be used to quickly move to certain positions in windows. Most of them are available in the Program window only.

Find
Searches for a string in the currently displayed source module. Shortcut: Ctrl+F

Find next
Finds the next occurrence of a string. Shortcut: Ctrl+N

Find prev
Finds the previous occurrence of a string.

Next procedure
Moves the user cursor to the beginning of the next procedure. Shortcut: Ctrl+Down

Prev. procedure
Moves the user cursor to the beginning of the previous procedure. Shortcut: Ctrl+Up

Goto line
Moves the user cursor to a given line in the displayed module. Shortcut: Ctrl+G

Home
Moves the user cursor to the current execution cursor position. Shortcut: Alt+Home

Goto address
Moves the user cursor to a certain address. Shortcut: Ctrl+A
3.4.5 The Components window

In the Components window, the debugger displays the list of executable components (EXE and DLLs) of your program. It can be opened by selecting Components from the Code menu.

When your program is first loaded into the debugger, its EXE file becomes the current component. To choose another component, double-click it in the Components window, or select it and press Enter.

The Components window contains two marks similar to cursors (see 3.4.1): one for the currently displayed component and another for the component in which debuggee execution stopped.

Use the pop-up menu to toggle display of full paths to components on or off.

3.4.6 The Modules window

The Modules window contains the list of modules which constitute the current component. To open it, select Modules from the Code menu or press Ctrl+M. To display another module in the Program window, double-click its name, select it using arrow keys and press Enter, or select Show code from the pop-up menu.

By default, only modules for which debug information is available are displayed. Switch the Show all modules option (see 3.9) ON or select Show all from the pop-up menu.

The module that is currently displayed in the Program window and the module that contains the instruction pointer have marks in the first column. The color of each mark matches the color of the respective cursor.

To display global variables of a particular module in a separate window (see 3.7.5), press Ctrl+V or select Show variables from the pop-up menu.

3.4.7 The Procedures window

All procedures of the current module are listed in the Procedures window, which may be opened by selecting Procedures from the Code menu or by pressing Ctrl+L. To position the user cursor on a procedure, double-click its name or select it using arrow keys and press Enter.
3.5. EXECUTING YOUR PROGRAM

3.4.8 The Publics window

The Publics window contains all public names belonging to your program’s code. Double-clicking or pressing Enter on a name causes the correspondent source or disassembled code to be displayed in the Program window.

3.5 Executing your program

The debugger provides different methods to execute your program. You may use the continuous execution commands to quickly move the execution point to a certain position in the code, to watch execution flow, or to wait for an event, such as variable access. The step commands allow you to monitor your program execution.

If the Program window is in the Source mode, step commands are performed in terms of source lines, otherwise — in terms of CPU instructions.

Note: Execution of your program may be interrupted earlier than implied by the command used to initiate it:

- if your program terminates
- if an exception is raised in your program
- if a breakpoint is hit
- if a break is activated

You may also explicitly interrupt your program and have control returned to XD.

The execution commands are collected in the Run menu:

Run
Start or resume program execution at full speed.

Execution trace
Start or resume line-by-line program execution.

Step into
Execute a single source or assembly line, stepping into a procedure call, if there is one.
Step over
   Execute a single source or assembly line without stepping into procedure calls.

Step out
   Execute until return from the current procedure.

Run to cursor
   Execute until the current user cursor position is reached.

Run to epilogue
   Execute until the current procedure’s stack cleanup code.

Run until address
   Execute until a given address is reached.

Restart
   Restart the debuggee and stop at the main module body.

Restart at startup
   Restart the debuggee and stop at the very beginning - the startup code.

Restart at entry point
   Restart the debuggee and stop at the entry point.

3.5.1 Run

The Run command resumes execution of your program from the current position at full speed, provided that no condition breaks (see 3.6.9) are installed and enabled.

Shortcut key: F5

3.5.2 Execution Trace

The Execution trace command executes the debuggee line by line or instruction by instruction, moving the execution cursor and updating all information windows after each step. This enables you to watch control and data flow. Execution continues until you press Esc and may also be interrupted by a breakpoint or a break.

You may control whether code, for which no debug information is available, is to be traced, by setting the Never fall into disassembler option (see 3.9).
3.5. EXECUTING YOUR PROGRAM

The Delay in execution trace option (see 3.9) allows you to specify a delay taken before execution of each line or instruction.

Shortcut key: Ctrl+F5

3.5.3 Step into

The Step into command executes the current line or instruction. If it contains a procedure call, execution stops at the first line or instruction of a called procedure.

The Never fall into disassembler option (see 3.9) controls whether procedures for which no debug information is available are traced or executed in one step.

Shortcut key: F7

3.5.4 Step over

The Step over command executes the current line or instruction. If it contains any procedure calls, they are executed at once, unless a breakpoint is encountered or a break is activated.

Shortcut keys: Ctrl+F7, Space

3.5.5 Step out

The Step out command places a temporary breakpoint right after the call instruction which called the current procedure, and then resumes execution. If no other breakpoints or breaks are encountered, the debuggee will be executed until the current procedure finishes, and then control will be returned back to the debugger. You may use this command if you accidentally issued the Step into command instead of Step over.

Shortcut key: Ctrl+F6

3.5.6 Run to cursor

The Run to cursor command installs a temporary breakpoint at the current position of the user cursor, which has to be on an executable line, and then resumes execution of the debuggee. So execution will stop at that line, provided that no
other breakpoints or breaks are encountered, no exception is raised and the de-
buggee is not terminated.
Shortcut key: F4

3.5.7 Run to epilogue

The Run to epilogue command executes the debuggee until the epilogue (see
2.1.2) of the current procedure is reached. This allows you to examine its local
variables just before the procedure returns.
This command is available only for procedures with debug information.

3.5.8 Run until address

The Run until address command installs a temporary breakpoint at an address
you specify and then resumes execution of your program.
Warning: Do not use this command unless you know for sure that there is an
instruction starting at that address, otherwise you will corrupt the code and exe-
cution will not stop.

3.5.9 Restart

The Restart command reloads the debuggee and executes it up to the beginning
of the main module body. All breakpoints, breaks, and watches are retained.
Note: Your program may load DLLs at run-time, which will be unavailable after
restart. If there were any breakpoints set in that DLLs, they will be disabled and a
warning message will be displayed.
Shortcut key: Ctrl+X

3.5.10 Restart at startup

The Restart at startup command is similar to the Restart (see 3.5.9) command
except that no debuggee instructions are executed after restart. This allows you to
debug DLL startup code.
3.6. USING BREAKPOINTS AND BREAKS

3.5.11 Restart at entry point

The Restart at entry point command is similar to the Restart (see 3.5.9) command except the debuggee is stopped at the EXE component’s entry point, when all load-time DLLs are loaded and initialized.

3.5.12 The Call Stack window

The Call Stack window can be displayed by selecting the correspondent item from the Code menu. It contains the names of the procedures which formed the call chain that led to the current position in the code. The current procedure is the topmost in the window.

To have a certain procedure displayed in the Program window, double click its name, or select it using arrow keys and press Enter. A mark will appear in front of that procedure name in the Call Stack window, and the Local variables window (see 3.7.6), if open, will display local variables an parameters of that procedure.

Note: In some cases, the debugger may be unable to correctly trace back the call chain, so the Call Stack window may contain some garbage entries. Turn the GENFRAME compiler option ON to improve stack tracing accuracy.

3.5.13 The Threads window

The Threads window displays information about all threads started by the debuggee. It can be opened by selecting Threads from the Code menu.

One of the threads is marked as current. When the debuggee is stopped, you may switch to another thread by double-clicking it or selecting with arrow keys and pressing Enter.

3.6 Using breakpoints and breaks

A breakpoint is a mark in the debuggee’s code which, once encountered (hit) during execution, causes it to stop and control to be transferred to the debugger. The source line or instruction on which the hit breakpoint is set is not executed.

A break is a condition which is constantly checked during execution. Once it evaluates to true, the debuggee is immediately stopped.
CHAPTER 3. USING THE DIALOG MODE

The major difference between breakpoints and breaks is that a breakpoint is bound to a particular source line or instruction, while a break is activated regardless of the current execution point. Using breakpoints, you may transfer control to the debugger when a certain code is about to be executed. Using breaks, you may force the debuggee to be stopped right after a particular event, such as modification of a specific variable, happens in it.

To set a breakpoint, move the user cursor to the desired line or instruction, and select an appropriate command from the Breaks menu (or press a shortcut key). Note that optimizations may cause strange effects during debugging, for instance, some lines containing statements may be displayed as unexecutable. See 2.1 for more information.

Note: Your program may load and unload DLLs at run-time. If there were any breakpoints set in such a DLL, and your program unloads it, the breakpoints are disabled and a warning message is displayed. The same happens in the case of a restart.

For historical reasons, XD supports only seven types of breakpoints which may not be combined together. This issue will be addressed in future releases.

To set a break, select an appropriate command from the Breaks menu (or press a shortcut key). Access breaks (see 3.6.8) may also be set from pop-up menus in some of the windows.

The commands related to breakpoints and breaks are collected in the Breaks menu:

**Sticky breakpoint**
- Set a sticky breakpoint at the current user cursor position.

**Breakpoint**
- Set an automatically removable breakpoint.

**Delayed sticky breakpoint**
- Set a sticky breakpoint with a countdown of hits.

**Delayed breakpoint**
- Set a removable breakpoint with a countdown.

**Expression breakpoint**
- Set a sticky breakpoint which triggers when a given expression is true.

**Pass counter**
- Set a sticky breakpoint which only counts number of hits.
3.6. USING BREAKPOINTS AND BREAKS

**Watchpoint**
Set a sticky breakpoint which automatically resumes execution.

**Access break**
Define an access break.

**Condition break**
Define a condition break.

**Disable**
Disable the breakpoint at the current user cursor position.

**Enable**
Enable a previously disabled breakpoint.

**Delete**
Remove the breakpoint at the current user cursor position.

**View all breaks**
Display the Breaks window.

**Disable all breaks**
Disable all breaks and breakpoints.

**Enable all breaks**
Enable all breaks and breakpoints.

**Erase all breaks**
Remove all breaks and breakpoints.

### 3.6.1 Sticky breakpoint

The **Sticky breakpoint** command sets a breakpoint at the line or instruction pointed to by the user cursor. It is called "sticky" because it is not removed when hit, unlike a simple breakpoint.

The ◊ symbol is displayed in the mark column for breakpoints in this type.

Shortcut key: F9
3.6.2 Breakpoint

The **Breakpoint** command sets a simple breakpoint which is automatically removed when it is hit.

The ▶ symbol in the mark column indicates a simple breakpoint.

Shortcut key: F8

3.6.3 Delayed sticky breakpoints

The **Delayed sticky breakpoint** command sets a sticky breakpoint which stops the debuggee only after the specified number of hits. This is very useful for debugging loops.

When you select this command, a dialog window is displayed, prompting you for an initial countdown value. Type the number of passes you wish the breakpoint to skip and press Enter.

In a typical debug scenario a delayed breakpoint is used in conjunction with a counter (see 3.6.6): you first use a counter to determine how many times a particular line has been executed before crash, then establish a delayed breakpoint at that line so that your program will be stopped on the last iteration before crash, and then restart your program. So if there is already a counter (see 3.6.6) established at the current line/address, the initial countdown value in the dialog will be automatically set to the current value of the counter decremented by 1.

When the countdown reaches zero, the breakpoint turns into a regular sticky breakpoint (see 3.6.1).

For a delayed sticky breakpoint, XD displays the remaining number of passes (or the * symbol if it exceeds 9) and the ◇ symbol in the mark column.

Shortcut key: Ctrl+F9

3.6.4 Delayed breakpoint

The **Delayed breakpoint** command sets a delayed breakpoint which is automatically removed after final hit. See 3.6.3 for more information about usage of delayed breakpoints.

A digit equal to the number of passes left (or the * symbol), followed by the ▶ symbol, is displayed in the mark column for a delayed breakpoint.
3.6. USING BREAKPOINTS AND BREAKS

Shortcut key: Ctrl+F8

3.6.5 Expression breakpoint

The Expression breakpoint command installs a sticky breakpoint and associates a boolean expression with it. The expression is evaluated each time the breakpoint is hit, and execution is stopped only if the result is TRUE.

When the user cursor is positioned over a line containing an expression breakpoint, the associated expression is shown in the Program window header.

A question mark ("?") in the mark column indicates an expression breakpoint.

Shortcut key: Alt+F9

3.6.6 Pass counter

The Pass counter command establishes a sticky breakpoint which, however, does not cause execution to stop but just counts the number of hits. A pass counter initially has a value of zero which is incremented each time the line or instruction associated with it is executed.

Pass counters provide a method to determine how many times, if at all, a particular line or instruction is encountered during program execution. This information may be used, for instance, when setting delayed breakpoints.

When the user cursor is positioned over a pass counter, its value is shown in the Program window header. You may also use the Breaks window (see 3.6.10) to examine pass counter values.

For a pass counter, its current value followed by the "." symbol is displayed in the mark column; values exceeding 9 are represented with the "*" symbol.

Shortcut key: .

3.6.7 Watchpoint

The Watchpoint command sets a sticky breakpoint, upon encountering which execution is resumed automatically. The effect is that all information windows are updated when a watchpoint is passed.

This approach is in many cases more efficient than execution trace (see 3.5.2) in the sense that a program may be executed at full speed (for instance, using the
CHAPTER 3. USING THE DIALOG MODE

Run command (see 3.5.1)), and information about its current state will be updated at and only at the moments you specify.

The ◦ symbol in the mark column indicates a watchpoint.

3.6.8 Access break

The Access break command allows you to have execution of your program interrupted immediately after it accesses a certain block of memory.

Note: Since access breaks are implemented using CPU debug registers, they do not slow down execution of a debuggee, but also have certain restrictions:

- Only blocks of 1, 2, or 4 bytes may be monitored
- A block has to be aligned, i.e. its address must be a multiple of its length
- Only write or read/write access may be detected
- Only four access breaks may be enabled at a time

When you activate this command, the Access break dialog is displayed. Select the desired access type and memory block length, type an address expression (see Chapter 5) in the Location field, and select the OK pushbutton.

Note: If there are already four enabled access breaks, the new one will be created in disabled state and a warning message will be issued.

Shortcut key: Ins

3.6.9 Condition break

The Condition break command allows you to specify a boolean expression which will be constantly evaluated during execution of your program until it yields TRUE, causing execution to be stopped.

Note: if there is at least one enabled condition break, your program will be stopped after each CPU instruction, so its execution will slow down dramatically. Avoid using condition breaks whenever possible.
3.6.10 The Breaks window

The Breaks window contains a list of all breakpoints and breaks currently set. To display it, press Ctrl+B or select View all breaks from the Breaks menu.

For a breakpoint, the name of the corresponding module followed by the breakpoint mark and the source line is displayed. For an access break, access type and address range are displayed. Finally, for a conditional break, the associated expression is displayed.

Double-clicking or pressing Enter on a breakpoint moves the user cursor to the breakpoint position. You may also achieve this by selecting Go to from the pop-up menu.

Double-clicking or pressing Enter on a break displays a corresponding dialog, allowing you to edit break parameters. The Modify command from the pop-up menu does the same.

3.6.11 Managing breaks and breakpoints

A break or breakpoint may be disabled at any time. A disabled breakpoint, once encountered, does not cause the debuggee to stop executing. The associated counter, if any, is not changed either. Marks corresponding to disabled breakpoints are displayed in different color.

To disable a breakpoint, move the user cursor to the line containing it or select it in the Breaks window and press "-" or select Disable from the Breaks menu. Press "+" or select Enable from the Breaks menu to re-enable a previously disabled breakpoint.

To delete a break or breakpoint, select it in the Program or Breaks window and press Del or select Delete from the Breaks menu.

You may disable, enable, or delete all breakpoints and breaks at once by selecting an appropriate item from the Breaks menu.

3.7 Examining your program data

The Data menu contains commands which open different kinds of information windows displaying the current state of your program’s data: variables, stack, CPU registers, etc.:
CHAPTER 3. USING THE DIALOG MODE

Examine data
Display and/or modify value of a given variable, designator, or register.

Evaluate expression
Evaluate an expression.

Global variables
Display the Global variables window containing global variables of the current component.

Module variables
Display the automatic Module variables window containing global variables of the current module.

Local variables
Display the Local variables window containing local variables and parameters of a procedure.

Memory dump
Open a new Dump window, displaying raw memory at a given address.

Registers
Display the Registers window containing CPU registers.

Float Registers
Display the Float Registers window containing FPU registers.

Stack
Display the Stack window, which contains memory at the stack pointer of the current thread.

Add watch
Add an expression to the Watches window.

Delete watch
Delete an item from the Watches window.

Show watch window
Display the Watches window.

Del all watches
Clears the Watches window.
3.7. EXAMINING YOUR PROGRAM DATA

3.7.1 Displaying and modifying values

The way a value of a variable or a watch expression is displayed depends on its type.

Values of elementary types immediately follow the correspondent variables and expressions in list windows. For variables of set and structured types (arrays and records), except character arrays, only type names are shown in list windows. Double-clicking or pressing Enter on such a variable causes a structured variable window to be displayed. Each line of those windows contains a single array element or record field.

Select Show address item from the pop-up menu of a structured variable window to toggle display of variable addresses in window headers on or off.

The Use single structure window option (see 3.9) controls whether a new window is opened to display a structured variable element which, in turn, is of a structured or pointer type.

You may alter the display format of a particular element of data using the Type pop-up submenu. You may change radix for wholes, turn pointer dereference on/off, etc.

To modify a variable of a basic, range, or enumeration type displayed in a list window, double click it or select it using arrow keys and press Enter. Type a new value in a displayed dialog and press Enter or press Esc if you do not want to modify the current value.

Double-clicking or pressing Enter on a pointer variable or expression has the same effect as it would have on a pointed-to object. Use the Ctrl+Enter key combination to modify value of a pointer variable itself.

3.7.2 Examine variable

The Examine variable command displays a dialog prompting you for a variable name. Type it and press Enter (in fact, you may type not just a variable name, but a designator (see Chapter 5), such as parr^[5].field). Depending on the type of an object denoted by the designator, either dialog containing the current value or structured variable view window will be displayed. See 3.7.1 for more information.

Shortcut key: ?

provided that the pointer is not NIL.
3.7.3 Evaluate expression

The **Evaluate expression** command allows you to evaluate an arbitrary expression (see Chapter 5) using the current values of program objects. Type it in a displayed dialog and press **Enter**.

See also 3.7.1.

3.7.4 The Global variables window

The **Global variables** window displays global variables of the current component, e.g. `extern` variables. To open it, select **Global variables** from the **Data** menu.

See also 3.7.1.

3.7.5 The Module variables window

The automatic **Module variables** window displays global variables of the module currently displayed in the **Program** window. To open it, select **Module variables** from the **Data** menu or press **Ctrl+V**.

To display global variables of a particular module in a separate window, select **Show variables** from the pop-up menu of the **Modules** window.

See also 3.7.1.

3.7.6 The Local variables window

Select **Local variables** from the **Data** menu to open the **Local variables** window. That window displays parameters and local variables of the procedure currently selected in the **Call Stack** window. Name of that procedure is displayed in the window header.

A horizontal line separates parameters from locals. An exclamation mark is displayed in the first column for register variables (see 2.1.3).

If the selected procedure is nested, and the option **DBGNESTEDPROC** was set ON during compilation of the module containing it, parameters and local variables of enclosing procedures are also displayed in the **Local variables** window. Horizontal lines with procedure names are used as separators.
You may quickly traverse the call stack (see 3.5.12) when the Local variables window is active. Press Ctrl+Down to display locals and parameters of the calling procedure, Ctrl+Up — of the called procedure.

See also 3.7.1.

3.7.7 Memory dump windows

You may use Memory dump windows to examine your program’s code or data as raw memory. To open a dump window, press Ctrl+D or select Memory dump from the Data menu. A dialog will display, prompting you for a starting address. Type an address expression and press Enter.

If the current window contains a variable list or a structured variable and you are opening a dump window, the address of the currently selected element will be substituted into the entry field.

Memory may be displayed in different formats. Select Type from the pop-up menu to switch between them.

To modify a dump element, double-click it, or select it using arrow keys and press Enter.

To change dump origin, click in the address column or select Change dump origin from the pop-up menu.

You can save a memory dump to file in text or raw form by selecting Save dump to file from the pop-up menu.

3.7.8 The Registers window

The Registers window displays names and values of the CPU registers. To open it, press Ctrl+R or select Registers from the Data menu.

Each time execution of a debuggee stops, register values which have been changed since previous stop are highlighted.

To modify a register value, double click it or select it with arrow keys and press Enter.
3.7.9 The Float Registers window

The **Float Registers** window displays names and values of the FPU registers. To open it, select **Float Registers** from the **Data** menu.

To modify a register value, double click it or select it with arrow keys and press **Enter**.

3.7.10 The Stack window

The **Stack** window displays stack of the current thread in raw form, starting from the current stack pointer position. It may be opened by selecting **Stack** from the **Data** menu.

You may change the format in which stack elements are displayed by selecting **Type** from the pop-up menu.

To change value of a stack element, double click it or select it with arrow keys and press **Enter**.

3.7.11 The Watches window

The **Watches** window contains arbitrary expressions along with their values. The expressions are re-evaluated in the current execution context each time the debuggee is stopped or program data is altered by the user. If an expression contains variables which are undefined or are not visible at the current execution point, an appropriate message is displayed instead of expression value.

Expression results are displayed according to their types. See 3.7.1 for more information.

Each time execution of a debuggee stops, expressions which results have changed since previous stop are highlighted.

To enter a new watch expression, select **Add watch** from the **Data** menu or press **Ctrl+Ins**. The **Watches** window will be automatically opened. To display it any time later, select **Show watch window** from the **Data** menu.

To delete a watch expression, select it and press **Del** or choose **Del watch** from the pop-up menu. To clear the **Watches** window, select **Del all watches** from the **Data** menu.
3.8 Saving scripts

When you find a suspicious place in your program, you may want to modify it slightly in order to prove or disprove your hypothesis. Then you will have to rebuild it and repeat the steps you took during the previous debugging session. You could save some time on the last stage by using the Save script command from the File menu before exiting the debugger. This command creates a control file (see Chapter 4) that, if invoked, would restore all breakpoints, breaks, and watches currently set in the debugger.

3.9 Setting XD options

Selecting Options from the File menu causes an option sheet to be displayed. These options are used to control various aspects of XD behaviour in dialog mode.

Delay in execution trace
The delay in milliseconds used in Execution trace mode (see 3.5.2).

Exception raise on first chance
If this option is enabled, the debugger displays a dialog immediately after an exception is raised in the program, allowing you to examine its state.

Delay to select debuggee
Specifies delay in milliseconds before the debuggee session is selected (see 3.1.7).

Never fall into disassembler
Check this box to disable automatic switching from the Source mode to the Disassembly mode during execution.

Code highlight
If enabled, XD displays executable lines in different color.

Full disasm mode
If enabled, XD tries to resolve addresses to publics and variable names in disassembly mode. Uncheck this box if you have a slow system, especially if you are running Windows 95.

Dump in disassembler mode
If enabled, bytes which constitute CPU instructions are displayed in Disassembly and Mixed modes.
Dereference pointers
If enabled, the debugger displays next to pointer variables values of objects to which they refer (if the base type is simple), base type name (if the base type is named compound type), or base type sort (if the base type is anonymous compound type).

Strip path from source name
If this option is enabled, the debugger uses its redirection file to locate source files. This may be useful if you debug a program which was compiled in a different environment.

Show all modules
Defines whether the Modules window (see 3.4.6) lists all modules or only those for which debug information is available.

Auto detect actual type
Enables detection of the actual dynamic type of an Oberon-2 pointer or a Java object. If this option is disabled, actual type of a particular entity can still be determined using a pop-up menu.

Use single structure window
Controls whether a new window is opened when you select an array element or a record field of a structured or pointer type.

Warning bell
Enables or disables speaker beeping when an error message is displayed.

Save debug session layout
Enabling this option causes information about debugger options, window layout and colors, and watch expressions to be automatically stored in a .ses file upon debugging session termination.

Use keyboard layout
When enabled, forces XD to load keyboard shotcuts upon startup from the file specified in the entry field; otherwise, the default layout is used (see 3.9.2).

Use the Save config command from the File menu to store the current option settings, window layout, and colors as the default.
3.9. SETTING XD OPTIONS

3.9.1 Changing window colors

To modify XD color schemes, select Palette from the File menu. A window containing a list of all window types will appear. Selecting a window type will cause its color sheet to be displayed. A color sheet consists of two columns, the left containing names of window areas, and the right containing current colors for each area.

The following keys may be used in a color sheet:

<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Up/Down</td>
<td>move the cursor one line up/down</td>
</tr>
<tr>
<td>Left/Right</td>
<td>change foreground color</td>
</tr>
<tr>
<td>Ctrl+Left/Right</td>
<td>change background color</td>
</tr>
<tr>
<td>Esc</td>
<td>close the color sheet</td>
</tr>
</tbody>
</table>

If you have the Save debug session layout option (see 3.9) switched ON, the new color scheme will be stored in a session file upon debugging session termination. Otherwise, the changes you made will be lost unless you explicitly save the current configuration using the Save config command from the File menu.

3.9.2 Modifying keyboard shortcuts

If the option Use keyboard layout (see 3.9) is enabled, upon startup the debugger attempts to load keyboard shortcuts from the file specified in the respective entry field. If that name does not contain directories, the file is sought via redirections, then is the current directory, and then in the directory where XD executable resides.

Default shortcuts are hard-wired into the debugger, so in order to modify them, you need to save the default layout to file using the Save keyboard layout command from the File menu. A keyboard shortcut file is a plain text file that looks similar to the following:

```
[ Actions and keys ]
Main pulldown = F10
Main window = Ctrl-T
Menu: File = Alt-F
Load program = F3
Refresh screen = Ctrl-E
Show output = /
Show debuggee = \nPalette =
```
Options =

Edit the layout file in any text editor, then enable the **Use keyboard layout** (see 3.9) and specify the name of that file in the respective entry field. described above.

### 3.9.3 Modifying frames

If the default window frames do not suite you, edit the FrameImageSingle, FrameImageDouble, and FrameImageMove items in the [Options] section of the debugger configuration file.
Chapter 4

Using the batch mode

When invoked in the *batch mode*, the debugger loads a control file — a text file containing XD control language statements. Those statements perform various debugging actions: load program, set breakpoint, add watch expression, start program execution, etc.

During control file processing, the debugger creates a log, which may be saved in a file and then analyzed.

There is also a statement that temporarily switches the debugger into the dialog mode. Upon exit from it, control file processing continues.

### 4.1 Control file syntax

Each line of a control file may contain a statement or a comment (empty lines are treated as comments). Comments start from the symbol `;`.

```
Statement =
    [ label ] name [ parameter {"," parameter}] [";" comment]
```

A statement consists of a label, a statement name, a comma-separated list of parameters and a comment. All these parts except statement name are optional. Statements may not be broken across two or more lines. Statement names are not case-sensitive.

Control files are often used to prepare a program for debugging in the dialog mode, and usually perform the following actions:
CHAPTER 4. USING THE BATCH MODE

1. load the program
2. establish breakpoints, breaks, and watches
3. initiate program execution
4. upon the first breakpoint hit, switch to the dialog mode
Below, is a small control file example with detailed comments.

```
LOAD d:\xds\samples\bench\dry.exe ;1
;2
MODULE dry ;3
;4
BREAK P0,ADDR,Proc0,Dialog ;5
BREAK P8,ADDR,Proc8,Dialog ;6
;7
WATCH Char1Glob,Array1Glob[8] ;8
;9
START ;10
QUIT ;11
;12
Dialog DIALOG ;13
STOP ;14
```

In this sample, comments are used to hold the line numbers referenced below.

**Line 1:** The LOAD statement loads an executable file, specified as a parameter, into the debugger.

**Line 3:** The MODULE statement sets the current module to dry, allowing to use unqualified identifiers in the following statements.

**Lines 5 and 6:** The BREAK statements establish sticky breakpoints at the entry points of the procedures dry.Proc0 and dry.Proc8. Dialog is the name of a label from which processing of the control file will continue if the breakpoint is hit.

**Line 8:** The WATCH statement adds expressions dry.Char1Glob and dry.Array1Glob[8] to the list of watch expressions.

**Line 10:** The START statement starts the debuggee. Processing of this statement finishes when the debuggee terminates or a STOP statement is executed in a break handler.

**Line 11:** The QUIT statement terminates the debugging session.

**Line 13:** The label Dialog was specified in the BREAK statements in lines 5 and 6. If either of the breakpoints set by those statements will be encountered during execution, in our case during processing of the START statement in line 10, the debugger will process the control file starting from this line.

The DIALOG statement switches the debugger to dialog mode. Breakpoints and watch expressions remain active. Upon user-initiated exit from the dialog mode,
CHAPTER 4. USING THE BATCH MODE

the debugger will return to the batch mode and process the next statement.

**Line 14:** The STOP statement terminates the debuggee. The debugger will continue control file processing from the line following the START statement.

### 4.2 Load program statement

<table>
<thead>
<tr>
<th>LOAD</th>
<th>Load a program</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD Program [ Arguments ]</td>
<td></td>
</tr>
</tbody>
</table>

The LOAD statement loads an executable file Program into the debugger, passing optional command-line Arguments to it. This statement should be executed prior to any statements which perform debugging actions, i.e. set breakpoints, add watch expressions, etc.

If a program fails to load, a message is displayed and control file processing is terminated.

**Possible Errors**

- 127 Incorrect program name
- 165 Program name expected
- 700 Loading <...>: <...>

**Examples**

```
LOAD xc =p =a dry
LOAD dry
```

### 4.3 Control flow statements

The statements described in this section allow to change the order in which control file is processed,
4.3. CONTROL FLOW STATEMENTS

| PAUSE | Suspend control file processing |

```
PAUSE [ Delay ]
```

The **PAUSE** statement suspends control file processing for *Delay* seconds, or until a key is pressed, allowing a user to view the last debugger messages. If *Delay* is not specified, infinite delay is assumed.

**Possible Errors**

126 Incorrect parameter

**Example**

```
PAUSE
PAUSE 10
```

| GOTO | Unconditional control transfer |

```
GOTO Label
```

The **GOTO** statement unconditionally transfers control to the statement associated with *Label*.

If the specified *Label* does not exist in the control file, an error message is issued, and control is passed to the next statement.

**Possible Errors**

112 Incorrect label
114 Label not found
168 Label expected

**Example**

```
GOTO Terminate
```
CHAPTER 4. USING THE BATCH MODE

**IF**

*Conditional control transfer*

**IF** Expression THEN Label

The **IF** statement evaluates Expression, which has to be of type BOOLEAN, and, if it yields TRUE, passes control to the statement associated with Label. If the result of Expression is FALSE, the next statement is executed.

If the specified Label does not exist in the control file, an error message is issued, and control is passed to the next statement.

**Possible Errors**

128 THEN expected
134 Expression expected
168 Label expected
176 Incorrect expression

**Examples**

```
IF Dry.Char1Glob='A' THEN Write
IF (Str.Len=10)AND(Memory.Limit) THEN exit
```

**CALL/RETURN**

*Procedure call*

**CALL** Label

RETURN

The **CALL** statement transfers control to the statement associated with Label. The **RETURN** statement returns control to the statement following the last executed **CALL** statement. Thus, these two statements allow to group the common parts of a control file into subroutines. The procedure calls may be nested.

If the specified Label does not exist in the control file, an error message is issued, and control is passed to the next statement. The same happens if a **RETURN** statement is encountered, which has no corresponding previously executed **CALL** statement.
Possible Errors

112 Incorrect label
114 Label not found
115 RETURN without CALL
168 Label expected

Example

CALL Write
.
Write . . .
RETURN

\begin{table}[h]
\centering
\begin{tabular}{|l|l|}
\hline
\textbf{SIGNAL} & \textbf{Error trapping} \\
\hline
SIGNAL "+" ErrorNumber "," Label & \\
SIGNAL "-" ErrorNumber & \\
\hline
\end{tabular}
\end{table}

The \texttt{SIGNAL} statement is used for error trapping. The "+"-form enables trapping of an error \texttt{ErrorNumber}. After execution of a statement caused this error, control is passed to the statement associated with \texttt{Label}. An error is considered handled and the error counter is not incremented.

If the specified \texttt{Label} does not exist in the control file, an error message is issued, and control is passed to the next statement.

The "-"-form disables trapping of a specified error.

The \texttt{#IMPORT} directive and the \texttt{SIGNAL} statement allow to implement reusable handlers for a certain group of errors.

Possible Errors

112 Incorrect label
114 Label not found
168 Label expected
208 Error code expected
209 Incorrect error list action
210 Incorrect error code
211 Handler for error <...> already defined
212 Handler for error <...> not defined
Example

SIGNAL +700, ErrorLoadProgram
SIGNAL -700

**DIALOG**  
*Switch to the dialog mode*

DIALOG [ Message ]

The **DIALOG** statement switches the debugger to the dialog mode. **Message**, if specified, is displayed in a message box. All breaks, breakpoints, and watches set in the batch mode remain active.

Control file processing will be resumed upon exit from the dialog mode, unless you select **Stop debugging** from the **File** menu.

**Possible Errors**

116 Program not loaded

Example

**DIALOG**

**QUIT**  
*Terminate control file processing*

QUIT

The **QUIT** statement unconditionally terminates control file processing.

**Possible Errors**

None.

Example

QUIT
4.4 Program execution statements

Statements of this group are used to establish breaks and breakpoints, and execute the program being debugged.

<table>
<thead>
<tr>
<th>START/STOP</th>
<th>Start/stop execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>START</td>
<td></td>
</tr>
<tr>
<td>STOP</td>
<td></td>
</tr>
</tbody>
</table>

The START statement begins execution of the debuggee. During execution, control may be returned to the debugger, for instance, as a result of a breakpoint hit. The STOP statement may be used in a handler (see 4.4) to terminate the debuggee. In this case, control file processing continues from a statement next to the START statement.

The START/STOP statements form a pair similar to CALL/RETURN, except that nesting is not allowed. This means that a program may not be restarted until it is explicitly stopped, so START is not allowed within a handler. Conversely, STOP is not allowed outside a handler.

The STOP statement unwinds the subroutine stack up to the level from which the START statement was issued.

Possible Errors

116 Program not loaded
129 Program restart not allowed

Example

```
START
STOP
```

<table>
<thead>
<tr>
<th>RESUME</th>
<th>Resume execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>RESUME</td>
<td></td>
</tr>
</tbody>
</table>

The RESUME statement effectively terminates a handler, causing program execution to continue after a breakpoint hit or a break activation. The CALL stack,
however, is not unwound, so issuing RETURN later may cause control to be transferred to the line after the CALL statement which called the subroutine containing the RESUME statement.

**Possible Errors**

116 Program not loaded  
130 Program was not started

**Example**

RESUME

```
| BREAK | Install an event handler |
|-----------------------------------------------|
| BREAK (Name | Number) "," Event |

Event = StopByUser  
  ProgramException  
  BreakAtAddress  
  BreakAtLine  
  BreakAtProc  
  WriteAccess  
  ReadAccess  
  BreakByCondition

StopByUser = USER "," Label.  
ProgramException = XCPT "," Label.  
BreakAtAddress = ADDR "," Address ","  
  ( Label [ "," Pass] [ ","," Condition ] [",", 
  "*" | "," ] ).  
BreakAtLine = LINE "," ModuleName "," Line ","  
  ( Label [ "," Pass] [ ","," Condition ] [",", 
  "*" | "," ] ).  
BreakAtProc = PROC "," ProcName "," [ "P" | "B" | "E" | "R" ]  
  ( Label [ "," Pass] [ ","," Condition ] [",", 
  "*" | "," ] ).  
WriteAccess = WRITE "," Address "," Length "," Label.  
ReadAccess = READ "," Address "," Length "," Label.  
BreakByCondition = COND "," BooleanExpression "," Label.
```

The **BREAK** statement establishes an event handler. When the specified Event
happens during execution of the debuggee, control is returned to XD and control file processing continues from the statement associated with Label. A handler is identified by an unique symbolic Name or Number.

The following types of events may be detected:

**USER** Execution is interrupted by user

**XCPT** An exception is raised

**ADDR** A breakpoint is hit at Address

**LINE** A breakpoint is hit at Line in ModuleName

**PROC** A breakpoint is hit at Prologue, Body (default), Epilogue, or Return instruction of the procedure ProcName

**WRITE** A Length bytes block of memory at Address was written to

**READ** A Length bytes block of memory at Address was read from

**COND** BooleanExpression was evaluated to TRUE

For breakpoints, you may optionally specify:

- **Pass** to set a delayed breakpoint (see 3.6.3)
- **Condition** to set a conditional breakpoint (see 3.6.5)
- **"/".** to set a non-sticky breakpoint (see 3.6.2)

Specify "*", or "." instead of Label to set a watchpoint (see 3.6.7) or a counter (see 3.6.6) respectively.

If the specified Line is not executable (see 2.1.1), a warning is issued, and the breakpoint is set at the first executable line which number is greater, if such line exists.

In Condition, it is possible to use variables that are not visible in the current context, but they have to be visible at the specified breakpoint location.

**Note:** It is recommended to remove access breaks from local variables upon return from a procedure.
CHAPTER 4. USING THE BATCH MODE

Possible Errors

112 Incorrect label
114 Label not found
116 Program not loaded
131 Incorrect address
151 Incorrect breakpoint number
152 Breakpoint number already in use
153 Incorrect event identifier
155 Incorrect length
159 Line number expected
160 Incorrect line number
161 Break already exists
168 Label expected
169 Breakpoint number expected
170 Breakpoint type expected
171 Breakpoint attributes expected
182 Incorrect break
213 Module not found
917 This break is set but is disabled due to hardware limit
918 Wrong expression in current context

Examples

BREAK 1,USER,Break
BREAK 2,XCPT,Break
BREAK 3,ADDR,Dry.Proc8,Break
BREAK 4,WRITE,ADR(Dry.Char1Glob),1,Break
BREAK 5,READ,ADR(Dry.Char1Glob),1,Break
BREAK 6,COND,Dry.Char1Glob=’A’,Break
BREAK 7,LINE,Dry,210,Break

DEL Remove event handler(s)

DEL Number {","Number}
Possible Errors

151 Incorrect breakpoint number
162 Break number <...> not found
169 Breakpoint number expected

Examples

DEL 1
DEL 3, 4, 17

4.5 Data management statements

The statements described in this section are used to change and display values of CPU registers, variables, and memory areas.

**FILL**

Initialize memory/registers

```plaintext
FILL ( Address | Register ) "," Length "," WholeExpression
```

The FILL statement initializes a memory area or a set of CPU registers by setting all its bytes to a specified value.

Address must be an address expression, so use ADR(v) to initialize a variable v. Length is a number of bytes to fill with result of WholeExpression MOD 100H.

It is also possible to specify CPU Register instead of memory Address. In this case, registers are "mapped" onto a byte array in the following sequence:

EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP

starting from the less significant byte.

Possible Errors

107 Parameter expected
116 Program not loaded
126 Incorrect parameter
131 Incorrect address
157 Address value expected
203 Write to address <...> failed, len <...>
Example

```
FILL ADR(Dry.Char1Glob),1,0x47
FILL ADR(Dry.Array2Dim),56,0
FILL @EAX,20,0xFA
```

**SET**  
*Change variable/register value*

```plaintext
SET ( Variable | Register ) "," Expression
```

The **SET** statement evaluates an **Expression** and assigns the result to a **Variable** or loads it into a CPU **Register**.

See Chapter 5 for more information about expressions in XD.

Possible Errors

116  Program not loaded  
134  Expression expected  
187  Write to register <...> failed  
200  Variable or register expected  
201  Type incompatibility in assignment  
205  Structured variable not allowed in this context

Example

```
SET Dry.Char1Glob,’A’
SET @ESP,0xFAFBFCFD
```

**MOVE**  
*Copy memory area*

```plaintext
MOVE SourceAddress "," DestinationAddress "," Length
```

The **MOVE** statement copies **Length** bytes of memory from **SourceAddress** to **DestinationAddress**. Both addresses may be specified as expressions (see Chapter 5). Overlapped areas are copied unsafely; a warning is issued in this case.
4.6. LOG CONTROL STATEMENTS

Possible Errors

116  Program not loaded
126  Incorrect parameter
155  Incorrect length
157  Address value expected
203  Write to address <...> failed, len <...>

Example

MOVE ADR(Dry.Char1Glob),ADR(Dry.Array1Dim),1

4.6  Log control statements

When operating in the batch mode, the debugger forms a log containing executed lines or instructions, messages, etc. The statements described in this section allow to output data to the log, redirect it to a file, toggle log filters, and otherwise control the logging features.

<table>
<thead>
<tr>
<th>LOG</th>
<th>Redirect log to file</th>
</tr>
</thead>
</table>

LOG [ FileName [ "." [ FileExt ] ] ]

The LOG statement allows to redirect log to a file. By default, the log information is written to the standard output. After redirection to a file, only error messages are written to the standard output. If FileExt is not specified, the default extension .XDL will be used. If no file name is specified at all, control file name will be used with extension replaced with the default log file extension.

Possible Errors

004  Error opening log file <...>
117  Incorrect log file name

Example

LOG
LOG result.
LOG result.dat
PRINT Format \{ \',\' Expression \}.

Format = \"\" \{ Character \| Specifier \| ControlSeq \} \"\"
Character = <any printable character except \'%\' and \'\'\'>
ControlSeq = \'\' ( \'n\' \| \'r\' \| \'l\' \| \'f\' | \\
\'t\' \| \'e\' \| \'g\' \| \%' \| \'\' )
Specifier = \'%\' \{ Modifier \} Base \| MemoryRegion
Modifier = \'\$\' \| \'-\' \| \'+\' \| \'\' \| \'#\' \| \'0\' \|
\| Space \| Width \| Precision
Base = \'i\' \| \'d\' \| \'u\' \| \'x\' \| \'o\' \| \'b\' | \\
\'c\' \| \'s\' \| \'w\' \| \'f\' \| \'e\' \| \'g\' | \\
\'X\' \| \'F\' \| \'E\' \| \'G\' \| \'u\' \| \%' \\
MemoryRegion = (\'M\' \| \'m\') \"\" \"\" \"\" \"\" \"\" \"\" \"\" \"\" \\
[\', \Const \[\', \Const \] ] \""
Width = Digit \{ Digit \}
Precision = \'\.' Digit \{ Digit \}

The PRINT statement performs formatted output to the log. The format string syntax is similar to that used by the printf family of functions available in the C standard library.

Format is a string which may contain ordinary characters, control sequences (starting with \'\'\') and format specifiers (starting with \'\%'\'). During execution of a PRINT statement, Format is scanned from left to right. Ordinary characters are sent to the log unchanged, while control sequences are replaced with certain control characters. Upon encountering of a format specifier, result of a corresponding Expression is formatted according to that specifier and output to the log. Therefore, the number of expressions has to match the number of format specifiers. If there is a specifier for which no expression is supplied, an error will be raised. If there are more expressions than specifiers, a warning will be issued. A newline sequence (CR+LF) is output upon completion of a PRINT statement.
Possible Errors

134   Expression expected
143   Double quotes expected
144   String not terminated (double quotes expected)
204   Read from address <...> failed, len <...>
205   Structured variable not allowed in this context
206   Register is not allowed

Example

PRINT "Hello, world!"
PRINT "Char1Glob=’%c’,Char1Glob"
PRINT "------
	1 bell\g
	2 bell\g
	3 bell\g
------"
PRINT "Memory region: %M(’%$4X\n’,k4,120)’,ADR(Array1Glob[6])"

Control sequences

ControlSeq is a two-character sequence which causes a certain control character to be output:

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Replaced with</th>
</tr>
</thead>
<tbody>
<tr>
<td>\n</td>
<td>CR+LF (15C 12C)</td>
</tr>
<tr>
<td>\r</td>
<td>CR (15C)</td>
</tr>
<tr>
<td>\l</td>
<td>LF (12C)</td>
</tr>
<tr>
<td>\f</td>
<td>FF (14C)</td>
</tr>
<tr>
<td>\t</td>
<td>TAB (11C)</td>
</tr>
<tr>
<td>\e</td>
<td>ESC (33C)</td>
</tr>
<tr>
<td>\g</td>
<td>BEL (7C)</td>
</tr>
<tr>
<td>%</td>
<td>percent sign (45C)</td>
</tr>
<tr>
<td>\</td>
<td>backslash (134C)</td>
</tr>
</tbody>
</table>

Format bases

The following format bases are recognized:
CHAPTER 4. USING THE BATCH MODE

<table>
<thead>
<tr>
<th>Base</th>
<th>Argument</th>
<th>Output format</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;c&quot;</td>
<td>CHAR</td>
<td>single character</td>
</tr>
<tr>
<td>&quot;d&quot;, &quot;i&quot;</td>
<td>integer</td>
<td>signed decimal integer</td>
</tr>
<tr>
<td>&quot;u&quot;</td>
<td>integer</td>
<td>unsigned decimal integer</td>
</tr>
<tr>
<td>&quot;o&quot;</td>
<td>integer</td>
<td>unsigned octal integer</td>
</tr>
<tr>
<td>&quot;x&quot;, &quot;X&quot;</td>
<td>integer</td>
<td>unsigned hexadecimal integer</td>
</tr>
<tr>
<td>&quot;f&quot;</td>
<td>real</td>
<td>fixed-point real</td>
</tr>
<tr>
<td>&quot;e&quot;, &quot;E&quot;</td>
<td>real</td>
<td>floating-point real</td>
</tr>
<tr>
<td>&quot;g&quot;, &quot;G&quot;</td>
<td>real</td>
<td>the shorter of &quot;f&quot; and &quot;e&quot;</td>
</tr>
<tr>
<td>&quot;s&quot;</td>
<td>string</td>
<td>string</td>
</tr>
<tr>
<td>&quot;w&quot;</td>
<td>boolean</td>
<td>&quot;FALSE&quot; or &quot;TRUE&quot;</td>
</tr>
<tr>
<td>&quot;n&quot;</td>
<td>any type</td>
<td>default</td>
</tr>
<tr>
<td>&quot;%&quot;</td>
<td>none</td>
<td>&quot;%&quot;</td>
</tr>
</tbody>
</table>

Note: Case of hex letters and of a letter used to introduce exponent in a floating point number matches case of the corresponding Base character.

Modifiers

Modifier allow to alter the default data formatting:

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Bases</th>
<th>Meaning</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;+&quot;</td>
<td>difegEG</td>
<td>always print sign</td>
<td>only negative</td>
</tr>
<tr>
<td>space</td>
<td>difegEG</td>
<td>print space instead of &quot;+&quot;</td>
<td></td>
</tr>
<tr>
<td>&quot;−&quot;</td>
<td>all</td>
<td>left-justify the value</td>
<td>right-justify</td>
</tr>
<tr>
<td>&quot;:&quot;</td>
<td>all</td>
<td>center the value</td>
<td></td>
</tr>
<tr>
<td>&quot;0&quot;, &quot;:&quot;</td>
<td>numeric</td>
<td>print leading zeroes</td>
<td>spaces</td>
</tr>
<tr>
<td>&quot;#&quot;</td>
<td>oxX</td>
<td>print a base character</td>
<td>no base character</td>
</tr>
</tbody>
</table>

Width

Width is an unsigned decimal specifying the minimum number of character positions used for a formatted value. If the output string is shorter than Width, blanks or zeroes are added according to Modifiers. If the number of characters in an output string exceeds Width, or Width is omitted, all characters are printed.

Precision

Precision is an unsigned decimal preceded by a period, which specifies the exact number of characters to be printed or the number of decimal places. Unlike
4.6. LOG CONTROL STATEMENTS

Width, Precision may cause the output value to be truncated or floating-point value to be rounded.

Precision value is interpreted depending upon the Base:

<table>
<thead>
<tr>
<th>For bases</th>
<th>Precision specifies</th>
</tr>
</thead>
<tbody>
<tr>
<td>i d u o x X</td>
<td>The minimum number of digits to output. Shorter values are padded on the left with blanks; longer values are not truncated. Default is 1.</td>
</tr>
<tr>
<td>f F e E</td>
<td>The number of positions after the decimal point. The last digit output is rounded. If Precision is 0, decimal point is not printed. Default is 6.</td>
</tr>
<tr>
<td>g G</td>
<td>The maximum number of significant digits to output. By default, all significant digits are printed.</td>
</tr>
<tr>
<td>c</td>
<td>The number of times the character is to output, default is 1.</td>
</tr>
<tr>
<td>s</td>
<td>The maximum number of characters to output. By default, characters are output until 0C is reached.</td>
</tr>
<tr>
<td>{}</td>
<td>Has no effect.</td>
</tr>
</tbody>
</table>

MODE

MODE Mode { "," Mode }

Mode = ADDR | BAT | BREAK | DUMP | PRINT ( ",+" | ",-" ) | TRACE ( ",+" | ",-" ) [ ",," ( CODE | SOURCE | MIX ) ] | DISASM ",=" ( SMALL | FULL ) | ERRORS ",=" number

The MODE statement controls what is written to the log and in which format.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Controls</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDR</td>
<td>whether memory dump lines contain addresses</td>
</tr>
<tr>
<td>BAT</td>
<td>whether executed control file statements are logged</td>
</tr>
<tr>
<td>BREAK</td>
<td>whether info about breaks is written</td>
</tr>
<tr>
<td>DISASM</td>
<td>disassembly mode</td>
</tr>
<tr>
<td>DUMP</td>
<td>whether instruction codes are dumped</td>
</tr>
<tr>
<td>ERRORS</td>
<td>maximum number of errors</td>
</tr>
<tr>
<td>PRINT</td>
<td>whether PRINT statements are executed</td>
</tr>
<tr>
<td>TRACE</td>
<td>the way execution trace is logged</td>
</tr>
</tbody>
</table>
CHAPTER 4. USING THE BATCH MODE

Possible Errors

126 Incorrect parameter
137 Incorrect mode
138 Errors limit set to <...
172 Mode expected

Example

MODE TRACE-
MODE TRACE+,CODE,DUMP+,DISASM=FULL
MODE PRINT+,ADDR-
MODE ERRORS=10,BREAK+

4.7 Miscellaneous statements

**MODULE**

Set current module

```
MODULE [ ModuleName ]
```

In the dialog mode, you may use names of program objects belonging to the current module without qualification. Similarly, in the batch mode, when execution of the program stops and control is returned to the debugger, the current component and module are set according to the value of the instruction pointer. The **MODULE** statement explicitly changes the current module. If ModuleName is not specified, unqualified identifiers recognition is disabled.

Possible Errors

116 Program not loaded
213 Module not found

Example

```
MODULE Dry
MODULE
```
4.7. MISCELLANEOUS STATEMENTS

**WATCH**

Add watch expression

```
WATCH Expression { "," Expression }
```

The **WATCH** statement is used to fill up the list of watch expressions (see 3.7.11), which is available in the dialog mode. It has no effect in the batch mode, and usually precedes the DIALOG statement.

Each **Expression** is checked for correctness. If **Expression** may not be evaluated in the current context, a warning will be issued.

**Possible Errors**

- 116 Program not loaded
- 918 Wrong expression in current context

**Example**

```
WATCH Dry.Char1Glob
WATCH Dry:Array2Glob[1],Memory.Free<=128
```

**#IMPORT**

Control file import

```
#IMPORT ControlFile
```

XD control files are likely to have some common routines, which perform, for instance, register dump, or heap tracing. It is possible to collect these routines in separate control files and import them into any control file which requires the routines.

The **#IMPORT** directive causes the debugger to load the specified ControlFile into memory and add its labels to the global table of labels.

After the debugger loads the control file specified on the command line, it scans it for labels and **#IMPORT** directives, and recursively repeats this procedure for all imported control files.

**Note:** This directive was intentionally named **#IMPORT**, not **#INCLUDE**, in order to emphasize the fact that the contents of the ControlFile are not substituted instead of the directive.
When XD reaches the end of a control file, it terminates regardless of whether that file was imported or specified on the command line. So make sure the imported control files are ended with an explicit control transfer statement, such as RETURN or GOTO.

Possible Errors

007 Batch name missed.
009 Too many nested packages (16)
010 Package <...> recursively imported
011 Incorrect package name <...>
014 Imported package <...> not found.
018 Unknown directive <...>.
111 Label ’<...>’ already defined
112 Incorrect label
113 Label matches reserved word

Example

#IMPORT PrintRegisters
#IMPORT SetBreakpoints
#IMPORT ErrorHandler
Chapter 5

Expressions

XD accepts expressions in various places. For instance, an expression may be added to the Watches window, used in a conditional breakpoint, and even entered as an initial counter value.

The expression syntax in XD closely matches Modula-2 syntax, with the following additions and exceptions:

- the qualified identifier syntax is expanded to allow referencing of entities from scopes and components other than current (see 5.3).
- expressions may not contain debuggee function calls; only debugger built-in functions (see 5.5) may be used.
- the construct:
  
  `Type "<" Expression ">"`

  designates an object of type `Type` residing at address to which `Expression` evaluates (see 5.6).

Type compatibility rules are much more relaxed in comparison with Modula-2. Also, implicit type conversions are performed under certain circumstances.

Set and complex types are not currently supported.

A *designator* is an expression which denotes a memory object (in other words, represents an lvalue). In some places XD expects a designator to be used.
5.1 Constants

5.1.1 Whole constants

WholeConst = DecimalConst | HexConst
DecimalConst = Digit { Digit }
HexConst = Digit { HexDigit } "H" |
            "0" ("x"|"X") HexDigit { HexDigit }
Digit = "0" | "1" | "2" | "3" | "4" |
        "5" | "6" | "7" | "8" | "9"
HexDigit = Digit |
           "a" | "b" | "c" | "d" | "e" | "f" |
            "A" | "B" | "C" | "D" | "E" | "F"

A decimal constant is a sequence of decimal digits '0'..'9'. A hexadecimal constant is a sequence of hexadecimal digits '0'..'9', 'a'..'f', 'A'..'F', either prefixed with "0x" or "0X", or postfixed with "H". Examples:

0x05074 13H 1342

The debugger stores a whole constant in 4 bytes of memory, so the maximum value of a whole constant is 0FFFFFFFFH.

5.1.2 Real constants

RealConst = DecimalConst "." [ DecimalConst ]
            [ "e" | "E" [ "+" | "-" ] DecimalConst ]

A whole decimal constant immediately followed by the period, is considered to be a whole part of a real constant. The period, in turn, may be followed by another whole decimal constant representing a fixed part. Finally, an exponent may be specified by adding the "E" or "e" character and an exponent value with an optional sign.

1. 3.14 0.1234E-10

Real constants are stored in 10 bytes format.

5.1.3 Boolean constants

BooleanConst = "TRUE" | "FALSE"
5.2. CPU REGISTERS

5.1.4 Character constants

CharConst = "'" character '"' | OctalConst "C"

A character constant is a single character enclosed in apostrophes:

'!' ''' 'X'

5.2 CPU registers

You may use names of CPU registers in expressions, prefixing their names with "@".

The debugger recognizes the following register names:

EAX, AX, AL, AH, EBX, BX, BL, BH,
ECX, CX, CL, CH, EDX, DX, DL, DH
ESI, SI, EDI, DI
ESP, SP, EBP, BP
EIP
CS, SS, DS, ES, GS, FS
EFL

5.3 Referencing program entities

It is possible to reference a named program entity in an expression, provided that
the executable contains debug information for that entity. You may use in expres-
sions identifiers of global and local variables and procedures, types, and enumera-
tion type elements. In addition, it is possible to reference DLL publics.

Entities are referenced by their identifiers. In Modula-2 and Oberon-2, a program
may contain more than one entity with the given identifier in different scopes.
By default, the debugger starts searching for an entity in the scope of the current
procedure (i.e. the procedure in which the execution point is placed) and then
goes up to the scope of the compilation module which contains that procedure.
Therefore, it may be necessary to qualify an entity name:

QualIdent = [ [ Component "$" ] Module "." ] { Proc "::" } Name
5.4 Operators

5.4.1 Unary operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>For operand type</th>
<th>Denotes</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>numeric</td>
<td>addition</td>
</tr>
<tr>
<td>-</td>
<td>numeric</td>
<td>subtraction</td>
</tr>
<tr>
<td>NOT</td>
<td>whole</td>
<td>bit-wise complement (32-bit)</td>
</tr>
<tr>
<td></td>
<td>boolean</td>
<td>logical complement</td>
</tr>
</tbody>
</table>

5.4.2 Binary operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>For operand types</th>
<th>Denotes</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>numeric</td>
<td>addition</td>
</tr>
<tr>
<td>-</td>
<td>numeric</td>
<td>subtraction</td>
</tr>
<tr>
<td></td>
<td>address</td>
<td>difference</td>
</tr>
<tr>
<td>*</td>
<td>numeric</td>
<td>multiplication</td>
</tr>
<tr>
<td>/</td>
<td>numeric</td>
<td>division</td>
</tr>
<tr>
<td>DIV</td>
<td>whole</td>
<td>division</td>
</tr>
<tr>
<td>MOD</td>
<td>whole</td>
<td>modulus</td>
</tr>
<tr>
<td>REM</td>
<td>whole</td>
<td>modulus</td>
</tr>
<tr>
<td>AND</td>
<td>whole</td>
<td>bit-wise AND (32-bit)</td>
</tr>
<tr>
<td></td>
<td>boolean</td>
<td>logical conjunction</td>
</tr>
<tr>
<td>OR</td>
<td>whole</td>
<td>bit-wise OR (32-bit)</td>
</tr>
<tr>
<td></td>
<td>boolean</td>
<td>logical inclusive disjunction</td>
</tr>
<tr>
<td>XOR</td>
<td>whole</td>
<td>bit-wise XOR (32-bit)</td>
</tr>
<tr>
<td></td>
<td>boolean</td>
<td>logical exclusive disjunction</td>
</tr>
</tbody>
</table>

It is also possible to add whole numbers to and subtract from addresses.

5.4.3 Comparison operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>For operand types</th>
<th>Tests if the first is</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>scalar</td>
<td>equal</td>
</tr>
<tr>
<td>#</td>
<td>scalar</td>
<td>not equal</td>
</tr>
<tr>
<td>&lt;</td>
<td>ordinal or real</td>
<td>less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>ordinal or real</td>
<td>less than or equal</td>
</tr>
<tr>
<td>&gt;</td>
<td>ordinal or real</td>
<td>greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>ordinal or real</td>
<td>greater than or equal</td>
</tr>
</tbody>
</table>

5.5 Built-in functions

5.5.1 ADR - address of a memory object

ADR "(" Designator ")"

Returns the address of the memory object designated by Designator.

ADR(ArrayOfRecords)
ADR(ArrayOfRecords[6])
ADR(ArrayOfRecords[6].FirstField)

5.5.2 SIZE - size of a memory object

SIZE "(" Designator ")"

Returns the size of the memory object designated by Designator.

Examples:

SIZE(ArrayOfRecords)
SIZE(ArrayOfRecords[6])
SIZE(ArrayOfRecords[6].FirstField)

5.5.3 LOW - low index of an array

LOW "(" Array ")"

Returns the low index of an array designated by Array.

LOW(ArrayOfRecords)

5.5.4 HIGH - high index of an array

HIGH "(" Array ")"

Returns the high index of an array designated by Array.

HIGH(ArrayOfRecords)
5.5.5 @PASS - number of passes for breakpoint

@PASS "(" Name | Number ")"

Returns the number of passes for the breakpoint identified by Name or Number (see 4.4). Batch mode only.

5.5.6 @ARG - access to command-line arguments

@ARG "(" n ")"

Returns n-th command-line argument specified after control file name. Batch mode only.

For instance, if the breakat.xd file contains the following statements:

```
LOAD @ARG(1)
BREAK B, PROC, @ARG(2),, Dialog
START
QUIT
```

Dialog DIALOG
STOP

and XD is launched as

```
xd /b breakat myprog M.P
```

it will load the program myprog, establish a breakpoint at the body of the procedure P from module M, execute the program up to that breakpoint and switch to the dislog mode.

5.6 Type casting

Type "(" Designator ")"

This kind of type casting allows you to treat a memory object designated by Designator as if it had type Type. Type can be any type defined in your program and present in its debug information, or one of the predefined types.

```
CARD8(File.result)
PrintModes.MODE(PrintModeBox^.selected)
```
### Type Casting

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>BYTE8</td>
<td>unsigned integer</td>
<td>1</td>
</tr>
<tr>
<td>BYTE16</td>
<td>unsigned integer</td>
<td>2</td>
</tr>
<tr>
<td>BYTE32</td>
<td>unsigned integer</td>
<td>4</td>
</tr>
<tr>
<td>INT8</td>
<td>signed integer</td>
<td>1</td>
</tr>
<tr>
<td>INT16</td>
<td>signed integer</td>
<td>2</td>
</tr>
<tr>
<td>INT32</td>
<td>signed integer</td>
<td>4</td>
</tr>
<tr>
<td>CARD8</td>
<td>unsigned integer</td>
<td>1</td>
</tr>
<tr>
<td>CARD16</td>
<td>unsigned integer</td>
<td>2</td>
</tr>
<tr>
<td>CARD32</td>
<td>unsigned integer</td>
<td>4</td>
</tr>
<tr>
<td>REAL</td>
<td>real</td>
<td>4</td>
</tr>
<tr>
<td>LONGREAL</td>
<td>real</td>
<td>8</td>
</tr>
<tr>
<td>LONGLONGREAL</td>
<td>real</td>
<td>10</td>
</tr>
<tr>
<td>COMPLEX</td>
<td>complex</td>
<td>8</td>
</tr>
<tr>
<td>LONGCOMPLEX</td>
<td>complex</td>
<td>16</td>
</tr>
<tr>
<td>BOOL8</td>
<td>boolean</td>
<td>1</td>
</tr>
<tr>
<td>BOOL16</td>
<td>boolean</td>
<td>2</td>
</tr>
<tr>
<td>BOOL32</td>
<td>boolean</td>
<td>4</td>
</tr>
<tr>
<td>CHAR8</td>
<td>character</td>
<td>1</td>
</tr>
<tr>
<td>SET8</td>
<td>bitset</td>
<td>1</td>
</tr>
<tr>
<td>SET16</td>
<td>bitset</td>
<td>2</td>
</tr>
<tr>
<td>SET32</td>
<td>bitset</td>
<td>4</td>
</tr>
<tr>
<td>ADDRESS</td>
<td>address</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 5.1: Predefined types

Type "<" Expression ">

The result is a memory object residing at address Expression and having type Type.

The result of Expression must be compatible with the address type.

```
ADDRESS<ADR(result)>
DataBase.ArrayCardinal<ADR(Array2DimInteger[3])>
```
Chapter 6

Remote debugging

With XD, you may debug programs running on another computer to which you may connect via network. Table 6.1 contains the list of supported networks protocols.

<table>
<thead>
<tr>
<th>Code</th>
<th>Protocol</th>
<th>Remote system identified by</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP</td>
<td>TCP/IP</td>
<td>IP address</td>
</tr>
</tbody>
</table>

Table 6.1: Supported Protocols

6.1 Preparing for remote debugging

Copy the following files from the BIN subdirectory of your XDS installation to an empty directory on the target system:

- XD_SRV.EXE
- XDS24.DLL
- XD NB09.DLL
- XD DITLS.DLL
- XD T_TCP.DLL
- XD NB04.DLL
- XD UTL.DLL

6.2 Starting debug server

To launch the debug server on the target machine, issue the following command:
XD_SRV /R=<protocol code>

Example:

XD_SRV /R=TCP

6.3 Starting XD in remote mode

To start XD in remote mode, use the /R command line switch:

XD /R=<protocol code>,<remote system id>

For instance, if you want XD to connect via TCP/IP to the debug server running on a machine which IP address is server.mycompany.com, issue the following command:

XD /R=TCP,server.mycompany.com
Chapter 7

Keys reference

7.1 General

F1   Help
F10  Main Menu
Alt+X Exit XD or return to batch mode
/    Switch to debuggee session
\   Switch between dialog and batch mode screens

7.2 Window management

Ctrl+F4  Close active window
Ctrl+Tab Activate next window
Ctrl+T   Activate Program window
Ctrl+R   Activate Registers window
Ctrl+B   Activate Breaks window
Ctrl+M   Activate Module List window
Ctrl+L   Activate Procedure List window
Ctrl+V   Activate Global variables window
Ctrl+D   Open a new Memory Dump window
Ctrl+W   Enter window move/resize mode (see below)

In the window move/resize mode, arrow keys move window, Ctrl+arrow keys change its size. Enter or Ctrl+W exits the mode.
7.3 Program window

<table>
<thead>
<tr>
<th>Key Combination</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Up/Down</td>
<td>move the user cursor one line up/down</td>
</tr>
<tr>
<td>Left/Right</td>
<td>scroll window left/right</td>
</tr>
<tr>
<td>PgUp/PgDn</td>
<td>move the user cursor one page up/down</td>
</tr>
<tr>
<td>Ctrl+PgUp/PgDn</td>
<td>move the user cursor to the top/bottom of window contents</td>
</tr>
<tr>
<td>Ctrl+Home/End</td>
<td>move the user cursor to the top/bottom of window</td>
</tr>
<tr>
<td>Home/End</td>
<td>show current line beginning/end</td>
</tr>
<tr>
<td>Ctrl+Up/Down</td>
<td>move the user cursor to previous/next procedure</td>
</tr>
<tr>
<td>Ctrl+H</td>
<td>move the user cursor to the execution cursor position</td>
</tr>
<tr>
<td>Ctrl+G</td>
<td>move the user cursor to a line with a certain number</td>
</tr>
<tr>
<td>Ctrl+F</td>
<td>find text</td>
</tr>
<tr>
<td>Ctrl+N</td>
<td>find next</td>
</tr>
<tr>
<td>Alt+1</td>
<td>switch to source view</td>
</tr>
<tr>
<td>Alt+2</td>
<td>switch to disassembly view</td>
</tr>
<tr>
<td>Alt+3</td>
<td>switch to mixed (source+disassembly) view</td>
</tr>
</tbody>
</table>

7.4 Information windows

<table>
<thead>
<tr>
<th>Key Combination</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Up/Down</td>
<td>move cursor one line up/down</td>
</tr>
<tr>
<td>PgUp/PgDn</td>
<td>move cursor one window up/down</td>
</tr>
<tr>
<td>Ctrl+PgUp/PgDn</td>
<td>move cursor to the top/bottom of window contents</td>
</tr>
<tr>
<td>Ctrl+Home/End</td>
<td>move cursor to the top/bottom of window</td>
</tr>
</tbody>
</table>
### 7.5 Breaks

<table>
<thead>
<tr>
<th>Key</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ins</td>
<td>Add conditional break</td>
</tr>
<tr>
<td>Del</td>
<td>Remove breakpoint</td>
</tr>
<tr>
<td>F8</td>
<td>Set breakpoint</td>
</tr>
<tr>
<td>F9</td>
<td>Set sticky breakpoint</td>
</tr>
<tr>
<td>Ctrl+F8</td>
<td>Set delayed breakpoint</td>
</tr>
<tr>
<td>Ctrl+F9</td>
<td>Set delayed sticky breakpoint</td>
</tr>
<tr>
<td>Alt+F9</td>
<td>Set expression breakpoint</td>
</tr>
<tr>
<td>.</td>
<td>Set counter</td>
</tr>
<tr>
<td>Plus</td>
<td>Enable breakpoint</td>
</tr>
<tr>
<td>Minus</td>
<td>Disable breakpoint</td>
</tr>
</tbody>
</table>

### 7.6 Execution

<table>
<thead>
<tr>
<th>Key</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>F5</td>
<td>Run</td>
</tr>
<tr>
<td>Ctrl+F5</td>
<td>Trace execution</td>
</tr>
<tr>
<td>Esc</td>
<td>Interrupt execution (in trace mode)</td>
</tr>
<tr>
<td>F4</td>
<td>Run to cursor</td>
</tr>
<tr>
<td>F7</td>
<td>Step into</td>
</tr>
<tr>
<td>Ctrl+F7</td>
<td>Step over</td>
</tr>
<tr>
<td>Ctrl+F6</td>
<td>Step out from current procedure</td>
</tr>
</tbody>
</table>

### 7.7 Watches

<table>
<thead>
<tr>
<th>Key</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ctrl+Ins</td>
<td>Add watch expression</td>
</tr>
<tr>
<td>Del</td>
<td>Remove watch</td>
</tr>
</tbody>
</table>
7.8 Function keys

| F1  | Help          |
| F3  | Load program  |
| F4  | Run to location |
| F5  | Run           |
| F8  | Set breakpoint|
| F7  | Step into     |
| F9  | Set sticky breakpoint |
| F10 | Activate Main Menu |
Appendix A

The HIS utility

XDS runtime support contains routines that in case of abnormal program termination may trace back the stack of the thread that is in error and output the call chain into a file called errinfo.$$$$. To enable this, the option **GENHISTORY** has to be turned ON during compilation of the main module of the program.

The HIS utility takes the stack trace file as input and prints execution history of the crashed thread in terms of line numbers and procedure names (provided that the respective executable components contain debug information):

```
XDS History formatter, Version 2.0
(c) 1996-2000 Excelsior
F:\xds\samples\modula\hisdemo.exe
----------------------------------------
#RTS: unhandled exception #6: zero or negative divisor
----------------------------------------
Source file     LINE  OFFSET    PROCEDURE   COMPONENT
----------------------------------------
hisdemo.mod    5   00000F    Div   hisdemo.exe
hisdemo.mod   11   000037   Try   hisdemo.exe
hisdemo.mod   15   000061   main  hisdemo.exe
```

In the above example, division by zero has been detected at line 5 in the procedure **Div** that was called by the procedure **Try** at line 11, and the procedure **Try**, in turn, was called from the main module body at line 15. All these procedures are defined in file **hisdemo.mod** and their code reside in the executable file **hisdemo.exe**.

By default, HIS prints to the standard output. Use **-l=file** option to redirect it to **file**.

73
APPENDIX A. THE HIS UTILITY

The \texttt{-f} option tells HIS to output full names of source files.
HIS is also able to facilitate quick reproduction of the erroneous situation in XD. Launching HIS with the \texttt{-b=file} option creates a debugger control file (see Chapter 4) that would stop the program immediately before execution of the instruction that causes crash:

\begin{verbatim}
; ; XDS History formatter, Version 2.0 ; (c) 1996-2000 Excelsior ; ; XDS Debugger control file

; 1. Load the program
LOAD F:\xds\samples\modula\hisdemo.exe

; 2. Establish breakpoint-counter
MODULE hisdemo$
BREAK Counter, LINE, hisdemo, 5, .

; 3. Initiate program execution at the first time
START

; 4. When program has finished, establish delayed sticky breakpoint
MODULE hisdemo$
BREAK Break, LINE, hisdemo, 5, Switch_To_Dialog, @PASS(Counter)-1

; 5. Delete auxiliary breakpoint
DEL Counter

; 6. Initiate program execution for the second time
RESTART
START
QUIT

; 7. Upon the breakpoint hit, switch to the dialog mode
Switch_To_Dialog
DIALOG #RTS: unhandled exception #6: zero or negative divisor
STOP
\end{verbatim}

You may use the \texttt{-c} option if you do not want comments to appear in the control file.
The `-d` option forces HIS to immediately launch the debugger with the created control file. If the `-b` option was not specified, a temporary control file is created. The `-s` option disables trace of the control file execution in the debugger.
Index

#IMPORT, 57
BREAK, 46
CALL/RETURN, 42
DEL, 48
designator, 59
DIALOG, 44
FILL, 49
GOTO, 41
IF, 42
LOAD, 40
LOG, 51
MODE, 55
MODULE, 56
MOVE, 50
PAUSE, 41
PRINT, 52
QUIT, 44
RESUME, 45
SET, 50
SIGNAL, 43
START/STOP, 45
WATCH, 57
This page had been intentionally left blank.