DETOP Users Manual
Release 1.0

Michael Oberhuber and Roland Wismüller
Lehrstuhl für Rechnertechnik
und Rechnerorganisation (Prof. Bode)
Fakultät für Informatik
Technische Universität München
80290 München
Germany

March 6, 1995

Contents

About This Manual

This manual explains how to use DETOP, a source level debugger for parallel programs using the PARIX operating systems on PowerXplorer and GC/PowerPlus parallel computers. It is divided into three parts:

1. The first part gives an introduction to DETOP's features, explains the graphical user interface and provides a short description of all available commands. This part should enable you to use DETOP for the first time.

2. The second part describes a short example session, giving examples for the most frequently used debugger commands and also some hints on debugging strategy.

3. The third part contains a detailed description of all debugger commands that interact with your application program. This part is intended to be used as a reference manual.

Throughout this manual, the following notation is used to define the syntax of commands or other input:

- For terminal symbols, i.e. plain text you have to type in, a typewriter font is used (e.g. `break 112`).

- Nonterminal symbols, i.e. names that represent some piece of text, are marked by an italic font. These symbols (e.g. *program*) have to be replaced by the proper text (e.g. `applic.px`) when the command is issued.

- In addition, there are a couple of meta-symbols only used to define the syntax:

  [*symbol*] denotes that *symbol* is optional, i.e. may or may not by present.

  { *symbol1* | *symbol2* } indicates an alternative, i.e. either *symbol1* or *symbol2* may be specified here.

  { *symbol* }* stands for zero or more occurrences of *symbol*.

  { *symbol* }$^+$ denotes one or more occurrences of *symbol*.

  Be sure not to confuse the meta-symbols [,], |, * and $^+$ with the corresponding terminal symbols `[`, `]`, `|`, `*` and `+`, that use another font.

Chapter 1

Overview

1.1 Introduction

In sequential programming environments, as offered by UNIX, source level debuggers are one of the most important tools for program development. Source level debuggers provide a detailed view of a program's execution behavior and of its internal states at the level of the source code. Thus, debuggers dramatically speed up the process of finding and eliminating software errors.

With DETOP, there is now also an adequate debugging support for the development of parallel programs. DETOP is a source level debugger for distributed PARIX programs running on PowerXplorer and GC/PowerPlus multi-processor computing systems. The most common languages for these systems, C and FORTRAN 77, are supported. DETOP offers some unique features that are essential for efficient debugging of parallel programs:

- DETOP is a real parallel debugger providing a global view of the distributed program. The user is not restricted to monitor and control only single threads, but can inspect the program as a whole.

- The debugger is running as a single process on the PowerXplorer's or PowerGC's front end or a remote workstation with only a small monitoring system executed on the multi-processor. Thereby there is only a minimal use of node resources, like memory or CPU time.

- A graphical interface based on X Windows and Motif supports a flexible and comfortable user interaction. A single debugger window is used to view the source code and issue debugger commands. Commands apply to either a single thread at a time or a set of threads.

- The debugger's user interface is completely asynchronous, so neither user interaction is blocked while the parallel program is running, nor the debugged program has to be stopped in order to issue debugger commands. This is essential when debugging parallel programs with multiple threads.

DETOP offers a variety of commands to inspect, control and monitor the execution of a parallel program:

- List all threads that exist at the current point of the program's execution. Additional information can be requested, e.g. scheduling state and procedure traceback.

- Display the value and type of a source code variable or expression in a formatted way. Variables may also be modified.

- Detailed control of program execution using distributed breakpoints. Breakpoints may be set at any line or procedure in the source code. In addition, breakpoints may also be set on thread creation, so dynamically created threads can be stopped before executing the first instruction.

  When a breakpoint is hit, either the thread that triggered the breakpoint or all threads of the program are stopped. Stopping all threads freezes the program's global state, so it can be viewed consistently.

- Stop all threads, start or continue a set of threads, execute a thread in single step mode.

- Show listings of the program's source code. The listing is also used to define breakpoints and to show the current point of execution when a thread is stopped.

- Full on-line help.

1.2 Starting DETOP

1.2.1 An Example

Assume you want to debug a program called `applic.px` running on four nodes. Without debugger you may start this program by typing:

```
px run -f0 2 2 applic.px par1 par2
```

As a prerequisite for debugging you have to compile and link this program with the `-g` compiler option set, in order to instruct the compiler to generate the information necessary for source level debugging. Using `make` you may simply add `-g` to the `CFLAGS` or `FFLAGS` variable in your makefile. Once your program is compiled this way, you may use DETOP following this starting sequence:

- Start the monitor `$PARIX/tbin/monitor.px` on the configuration you need for your application. `$PARIX` is the directory where DETOP has been installed (usually this is the same directory where PARIX is installed). Note that the monitor needs at least eight host links, so if you are using only one node, you must add the option `-MH8` to the run command (see man-pages for `run` and `mkdesc`). In the example, since there are four nodes, 16 host links will be created as a default, so the monitor can be started by:

  ```
  px run -f0 2 2 $PARIX/tbin/monitor.px -k ke
  y4711 -h sparc2
  ```

  The option `-k key4711` defines a key that is used to ensure that the monitor connects to the correct invocation of the debugger. A connection will only be established, if both monitor and DETOP are started with the same key. If you omit this option, a built-in key will be used. The second option `-h sparc2` defines the hostname of the machine where the debugger is started.

- Start DETOP. DETOP will need a SUN SPARC workstation running X11 R5 or later to execute. For the above example, the command line will look like this:

  ```
  px detop -k key4711 -- applic.px par1 par2
  ```

  The key defined with the `-k` option must match the one supplied to the monitor. If you didn't specify a `-k` option when starting the monitor you must also omit it here.

DETOP and the monitoring system will now try to connect to each other via a TCP/IP socket. Both programs will indicate this by a short message. By using a socket connection, DETOP does not need to execute on the PowerXplorer's or GC/PowerPlus' front end. However, if you start DETOP on another machine, you must indicate the host-name where DETOP is started using the option `-h`.

To work properly, DETOP needs some auxiliary files (e.g. resource files, bitmaps and files for on-line help). When DETOP is installed in the same directory as PARIX, it will locate these files using the environment variable `PARIX`. Starting DETOP via the `px` command ensures that this variable is set properly and a proper command search path is used to search DETOP's binary. DETOP may also be installed in a different directory as PARIX. In this case, you must set an environment variable `DETOP_DIR` containing the path name of that directory. In the rest of this manual it is assumed that DETOP in istalled in the PARIX directory, so all files coming with DETOP are referred to using `$PARIX`.

If Motif is not installed at your workstation, you should define an additional environment variable `XKEYSYMDB` with the value `$PARIX/lib/X11/app-defaults/XKeysymDB`.

Once DETOP has started, your application is loaded on each node, however, all threads are stopped before executing any instruction of your program. To start the program, use the Continue All command (see Section 1.3.1).

It is convenient to use a small shell script starting both the monitor and the debugger. For the example given above, this script may look like:

```
#!/bin/sh

detop -k $$ -- applic.px par1 par2 &
run -f0 2 2 $PARIX/tbin/monitor.px -k $$ -h `ho
stname`
```

Two somewhat more general scripts are included in the `$PARIX/example/Detop` directory. `run_detop` is for use with fixed topology, while `run_detop.nrm` is for use with `nrm`. All these scripts should be started using the `px` command.

1.2.2 Command Line Options

1.2.2.1 Monitor

`px run` *run_options* `$PARIX/tbin/monitor.px`
    [`-h` *host_name*] [`-k` *key*]

run_options: The options supplied to the `run` command should be exactly the same as in the case when you start your application without the debugger. See the man page of `run` for more detail. Note, however, that the monitor needs at least eight host links, so if you are using only one node, you must add the option `-MH8` here.

`-h` *host_name*: Defines the host-name of the SUN SPARCstation where you start DETOP. If you omit this option, the monitor assumes that DETOP is started on the PowerXplorer's or GC/PowerPlus' front end. However, if you are using `nrm` and have access to more than one PowerXplorer or GC/PowerPlus it is advisable to always specify this option, since in this case you do not know the front end in advance.

`-k` *key*: This option allows to specify a key that is used to ensure that the monitor connects to the correct invocation of DETOP, if there is more than one DETOP waiting for connections on the specified host. A connection is only established if the same key is specified for both the monitor and DETOP. You may omit this option for both the monitor and DETOP. In this case, a predefined key will be used. However, this key is the same for all invocations of DETOP, so it may happen that the monitor connects to the wrong DETOP.

1.2.2.2 DETOP

`px detop` [*X-options*] [`-k` *key*] `--` *program* [*parameters ...*]

X-options: Since DETOP is a standard X application, you may add options applicable to all X window based programs. So the option `-display` *display_name* can be used to redirect DETOP's graphical output to a non-local X server. See man-page `X(1)` for more details.

`-k` *key*: This option allows to specify a key that is used to ensure that the monitor connects to the correct invocation of DETOP, if there is more than one DETOP waiting for connections on the specified host. A connection is only established if the same key is specified for both the monitor and DETOP. You may omit this option for both the monitor and DETOP. In this case, a predefined key will be used. However, this key is the same for all invocations of DETOP, so it may happen that the monitor connects to the wrong DETOP.

`--`: This option indicates the end of DETOP's options. It must be immediately followed by the name and command line parameters of the program you want to debug.

program: The name of the program you want to debug, i.e. the name of the executable file. The program has to be compiled and linked with the compiler/linker option `-g`. If you start DETOP on a remote machine, using the `-h` option, the program must be accessible on both machines with the same pathname.

parameters ...: The command line parameters supplied to the program to be debugged.

1.2.3 Using DETOP on a Remote Workstation

As already mentioned above, you can start DETOP on any workstation that is connected to the PowerXplorer's or GC/PowerPlus' front end via TCP/IP by using the `-h` option. However, if you do so, it is necessary that the executable file of your application and all executables loaded by `Execute()` are accessible on both machines with equal path names.

Even with shared home directories in an NFS environment, this may not be the case, e.g. if symbolic links are used for auto-mounting. In order to support environments with different path names, DETOP provides a simple path translation mechanism. You may define an environment variable `TOPSYS_PATH_TRANSLATION` containing a colon separated list of translations:

translation { : *translation* }*

where each *translation* is defined as:

translation ::= *front_end_prefix* `=` *host_prefix*

Now, If the file name of the initial executable is transferred from DETOP to the monitoring system, it is first converted to an absolute path name by prepending the current directory name as returned by the `getwd()` system call, if necessary. Then `TOPSYS_PATH_TRANSLATION` is scanned for a *host_prefix* that equals the prefix of that path name. If such a prefix is found, it is replaced by the proper *front_end_prefix*. The monitor then tries to load the resulting file from the front end.

If the path name of an executable file loaded via `Execute()` is transferred from some monitor to DETOP, `TOPSYS_PATH_TRANSLATION` is scanned left to right for a *front_end_prefix* that equals the prefix of that path name. If a match is found, the prefix is replaced by the proper *host_prefix*. Note that the path name of an executable file is always an absolute path.

For example, assume an environment where your application is located at `/usr/smith/src/example.px` on the PowerXplorer's or GC/PowerPlus' front end. The directory `/usr/smith/src` may then be mounted on the remote workstation as `/mnt/gcpp/smith/example`. Then you should set the `TOPSYS_PATH_TRANSLATION` environment variable by typing

```
setenv TOPSYS_PATH_TRANSLATION /usr/smith/src=/
mnt/gcpp/smith/src
```

in C shell syntax or

```
TOPSYS_PATH_TRANSLATION=/usr/smith/src=/mnt/gcp
p/smith/src
export TOPSYS_PATH_TRANSLATION
```

in Bourne shell notation.

1.3 User Interface

DETOP offers a graphical user interface based on X windows and Motif. The debugger window is centered around a source code listing. Multiple threads can be debugged simultaneously to support data parallel programs where each thread executes the same code. Future versions will allow to have more than one debugger window in order to support task parallel programs.

The debugger's main window is shown in Fig. 1.1. It consists of six major parts, that are explained in more detail in Chapters 1.3.1 to 1.3.6:

1. The **main menu** offers pull-down menus for loading source files, selecting the threads to be debugged, global commands and some configuration parameters.

2. The **thread list** shows the subset of threads currently being debugged. These threads are continuously monitored by the debugger; once one of them is stopped (either due to a breakpoint or an exception), DETOP will notify you and update the thread's state in the state display. The thread list is also used to select the threads a command will work on.

3. The **source code listing** is used to display a source code file, the current positions of threads and the breakpoints activated in this code. The listing also provides popup-menus allowing to issue commands in a source code related fashion.

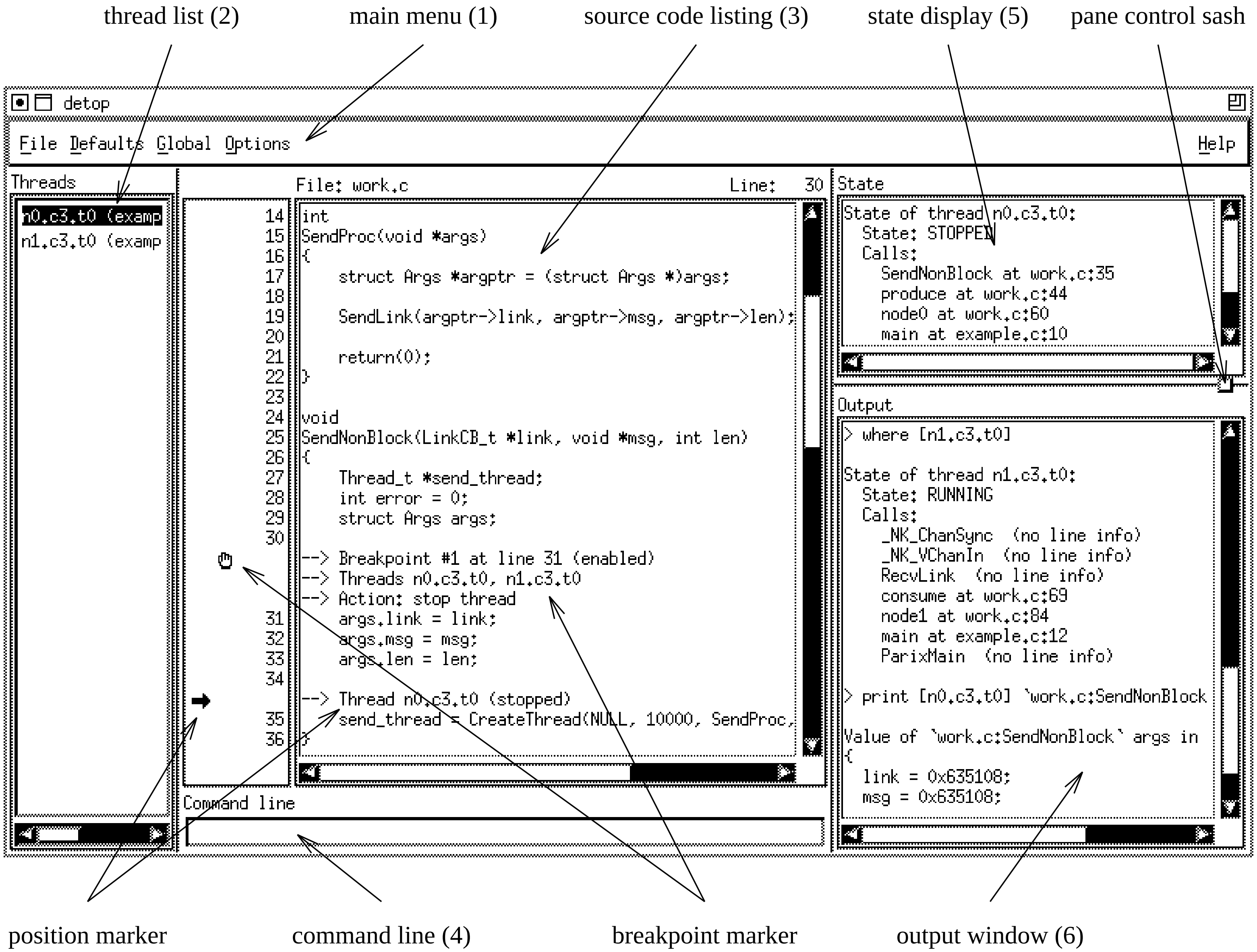


Figure 1.1: Main window

4. An additional **command line** allows to enter debugger commands via keyboard. For convenience, most command names are the same as in sequential debuggers like `gdb`. In addition, all commands may be abbreviated, most of them by just one character. The command line also provides editing facilities and an input history.

5. The **state display** shows the procedure tracebacks of all monitored threads. A thread's traceback is updated in-place whenever the thread is stopped by the debugger or due to an exception.

6. In the **output window** all issued commands and their results are displayed.

1.3.1 Main Menu

DETOP's main menu offers the following selections:

File: The File menu offers four items:

> **Load Source File:** Creates a file selection box that allows to specify the file to be displayed in the source listing window (see 1.3.3.1).
>
> **Thread Selection:** Creates a dialog window that allows to view the currently existing threads and to select the threads to be debugged (see 1.3.2).
>
> **Info:** Displays copyright information.
>
> **Quit:** Exits DETOP.

Defaults: This menu allows to change some of the debugger's default settings:

Actions: When a breakpoint is hit, the debugger can perform a couple of different actions. The submenu associated with this menu item allows you to specify the default action. There are three possible choices:

Stop Thread: Stop only the thread that has hit the breakpoint. All other threads continue operation until they are stopped by another breakpoint or a `stop` command.

Stop Node: Stop all threads on the processing node where the breakpoint was hit. This setting is useful if you use shared memory between threads. Stopping the whole node prevents shared data from being modified by other threads.

Stop All: Stop all threads on all nodes. This action freezes the global state of computation, so you can inspect this state in a consistent way.

The default actions are always used if you set a breakpoint via the listing window's popup-menu. The command line allows to set breakpoints with a non-default action (see 1.3.4 and 3.2).

Tabs: Pops up a requester to specify the tab width to be used in the listing window. Note that a new setting will not affect the source file currently displayed. To change the tab width in the current file, you have to re-load it.

Global: The global commands always affect *all* threads of your program, not just the ones displayed or selected in the thread list. There are three commands of this class:

Continue All: This command can be used to start all threads of your application after the debugger has been invoked, or to resume all threads after a Stop All command or breakpoint action.

Stop All: Stop all threads of the program being debugged.

Stop New Threads: If this option is on, all dynamically created threads will be stopped immediately before they execute their first instruction. A message will be printed in the output window. You may then add this thread to the thread list using the Thread Selection menu item. If this option is off, you will not be informed about thread creation. Any dynamically created thread will start working immediately.

Options: This menu offers some items to configure the graphical user interface according to your own preferences:

Threads on/off: Show or hide the thread list part of the debugger window. If you are debugging only one thread you may hide the thread list in order to increase the size of the program listing.

Listing on/off: You can also switch on and off the middle part of the debugger window including the source listing and the command line. This may be useful if you want to study some output that is very wide.

Output on/off: If you want more room for the program listing you can also switch off the state and output part of the debugger window. Doing this may be useful to study the source code or during single-stepping.

Breaks on/off: Usually, a breakpoint is marked in the source listing by both a little icon left of the listing and an additional text inserted into the source listing. By selecting the option Breaks off, you can suppress this additional text.

Stops on/off: Just like breakpoints, also the position where a thread is stopped is indicated both by an icon and an inserted text in the listing. This menu item allows to switch off the inserted text.

Lines on/off: You may disable the line numbers associated with the source code listing using this item.

Scroll: This option controls the source code listing's behavior when a monitored thread is stopped for some reason. DETOP will then always mark the thread's current position in the source listing (see 1.3.3.2 for more details on these markers). However, to avoid repeated scrolling when a couple of threads stops nearly at the same time, by default the listing will not scroll to make this mark visible.

If you are focusing on only one thread or a set of threads executing the same code in a very synchronous data-parallel manner, you may want to change this behavior. The Scroll menu item offers a submenu containing three possible options:

Off: This is the default behavior: The source listing will neither scroll to make a thread's position marker visible nor any attempt is made to load the file containing the thread's current point of execution.

Local: If this option is selected, the source listing will scroll to show the position marker in the currently loaded source file. However, no attempt is made to load the file containing the thread's current point of execution.

Global: With this option, DETOP will try to load the source file containing the thread's current point of execution. Since the debugger cannot determine the file's directory, but only its name, a simple search path is used: First, the current directory is searched; if this fails, the directory selected in the source file selection box (see 1.3.3.1) is examined. If a source file is found this way, it is loaded into the listing window, otherwise the old source file is kept. In any case, the listing is then scrolled in order to make the position marker visible.

Help: Select an item in this menu to get on-line help for the selected topic.

To select an item from the main menu, move the mouse cursor over the menu bar, press the left mouse button, move the cursor over the desired item and release the left button again. All menu items can also be selected via keyboard: Press the META-key and the letter underlined in the menu-bar to pull down a menu. Then press the letter underlined in the menu item (without META) to select the item.

Menu items that set an option will show the current setting in parentheses. These items either work as toggles or have an additional sub-menu to select the option.

1.3.2 Thread Selection

Thread selection is a two phase process in DETOP: In the thread selection window you can select the subset of all existing threads that is of interest for the current debugging session. From this subset that is displayed in the debugger window's thread list you can then choose the threads a command shall work on.

Building the subset keeps the main window's thread list short and clear. In future versions of DETOP it will also be possible to debug different sets of threads in different debugger windows.

1.3.2.1 Thread Identification

Since DETOP can be used to debug any thread created by your application, globally unique identifiers must be used to distinguish between different threads. The PARIX operation system offers such identifiers, called global identifiers. In the debugger interface a global identifier is a string containing a node number, a context number and a thread number, for example `n0.c3.t0`. This identifier represents the first thread created on node 0 in context 3. In the current implementation of operating system and debugger, context 3 is the first user context created on a node. Thus, `n0.c3.t0` is your application's initial thread on node 0.

The contexts on a node and threads within a context are numbered in a strictly ascending order. Thus, assuming that `n0.c3.t0` is the thread most recently created on node 0, the next thread that is created in the same context (via `CreateThread()`) will get the identifier `n0.c3.t1`. If a new thread is created in a new context, using the `Execute()`

function, its identifier will be `n0.c4.t0`.

To further ease the process of identifying threads, DETOP's thread lists will display additional information for each thread:

1. The name of the binary file executed by the thread. This is useful for applications creating different contexts via `Execute()`.

2. If the symbol table for the thread's context is already loaded, the name of the thread's root function will be shown, so you can distinguish between threads executing different codes. Static C-functions defined in different source files with the same name will be preceded by the correct file name. If no symbolic information is present, the root function's address is given as a hexadecimal number.

3. Finally, the pointer to the thread's `Thread_t` structure, as returned by `CreateThread()`, is given as a hexadecimal number. Using this information, you can identify threads belonging to a `Thread_t` pointer stored in some program variable.

In the debugger's thread lists the additional information is shown in parentheses after the thread's global identifier. Thus, a complete entry in a thread list may be:

```
n0.c4.t2 (prog.px,work.c:do_work,0x633570)
```

This entry identifies the third thread started in the context created by the first `Execute()` on node 0. The thread executes part of the binary file `prog.px`, its root function is the static function `do_work()` in source file `work.c`. The pointer

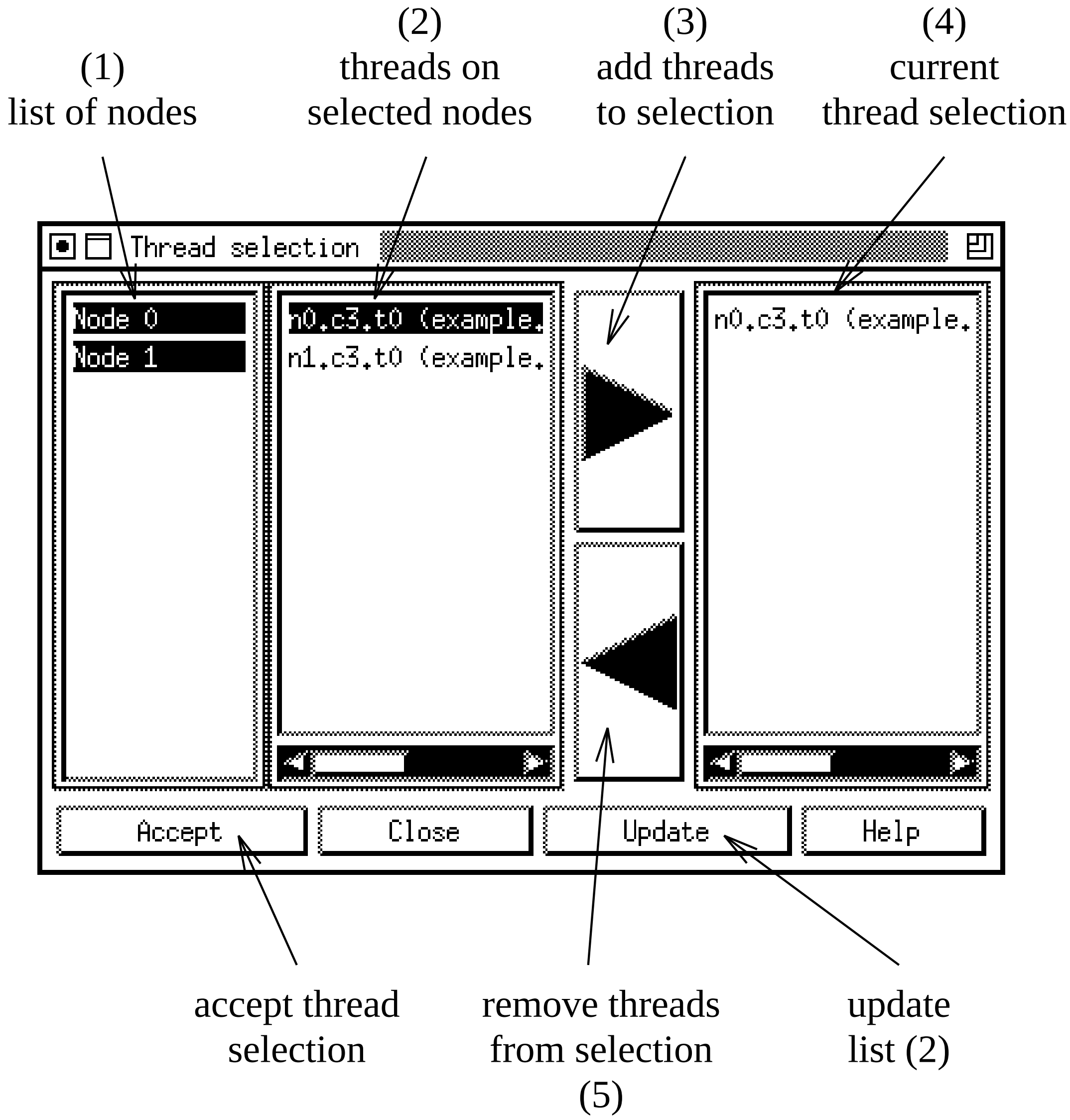


Figure 1.2: Thread selection window

to the corresponding `Thread_t` structure, and therefore the result of the call to `CreateThread()` that created this thread is 0x633570.

1.3.2.2 The Thread Selection Window

By selecting the item Thread Selection in the File menu, you can pop up the thread selection window shown in Fig. 1.2. The leftmost part (1) of this window lists the nodes in your allocated partition. You may now select one or more nodes in this list by means of the Motif extended list selection:

- To select a single node, simply click on its list entry.
- To select a continuous range of nodes, click on the first node, hold the left mouse button pressed, and drag the mouse until all nodes in the range are highlighted.
- You can also toggle the selection state of each node by clicking on the appropriate list item while pressing the CTRL key. In this way you can assemble arbitrary node sets.

Once a set of nodes is selected, list (2) will show the identifiers and the associated additional information of all threads currently existing on the selected nodes. Note that new threads are not added to this list automatically upon creation. You have to press the Update button to perform an update.

In list (2) you may now select a set of threads in the same way as in the node list. By pressing the > button (3) these threads will be added to the set of threads to be debugged that is shown in list (4). In a similar way you can delete threads from the selection: Select the threads to be deleted in list (4) and press the < button (5).

You may repeat this procedure until the set is complete. Pressing the Accept button accepts the set of threads contained in list (4). This set will then be displayed in the main window's thread list. In addition, the debugger will load the thread's symbol information, if necessary, so you can also debug threads in a new context created by the `Execute()` function. Pressing the Close button will close the thread selection window.

Note that changing the thread selection will not change the state of any thread. So if you add a thread to your selection, this thread will not be stopped automatically, but keeps its state. Likewise, if the new selection no longer contains a previously selected thread, this thread is not automatically continued. Also all breakpoints set for this thread remain active until you delete them explicitly.

1.3.2.3 The Thread List

The main window's thread list (see Fig. 1.1) shows the subset of threads you are currently debugging. All threads in this subset will be permanently monitored by the debugger, so if any of them is stopped (either due to a breakpoint or an exception), DETOP will notify you, update the thread's state in the state display and indicate the point where is has been stopped in the source listing. In addition, the thread list is used to select the threads a command will work on.

Normally, all debugger commands affect only those threads that are selected in the main window's thread list. If you set a breakpoint in the situation shown in Fig 1.1, where only thread `n0.c3.t0` is selected the breakpoint is valid only for this very thread, i.e. it can be triggered only by thread `n0.c3.t0`. A `continue` command will only (re-)start this thread, the state of any other thread will not be changed. If both threads were selected, a breakpoint you set could be triggered by any of them. Issuing a `continue` command would (re-)start both threads.

This rule does not apply to the commands in the Global menu which always affect the whole application, and in case you override the thread selection in the command line (see 1.3.4). Breakpoint actions may also affect threads not selected in the thread list.

1.3.3 Source Listing

The source listing is used to display source code and additional state information. It also provides context-sensitive popup-menus for easy command input.

1.3.3.1 Source File Selection

To load a source file into the main window's source listing part, select the Load Source File item in the File menu. This will pop up a Motif file selection box, that contains two lists allowing to select a directory and a file name (see Fig. 1.3).

To change the directory, click on an item in the directory list and press the Filter button. As a shortcut, you may double-click on a list item. Then all files matching the pattern in the text widget named Filter will be displayed in the file list. The selected directory is also used to find source files when the Scroll Global option is selected (see p.19).

To load a source file, select the appropriate item in the file list and press Ok. Again you may alternatively double-click the item. If you prefer, you may also enter the file name in the text widget named Selection and press the Ok button.

1.3.3.2 State Information

There are two different kinds of state information, that are displayed in the source listing: Breakpoints and locations where threads are stopped or have been inspected last.

Breakpoints: A breakpoint set at a source line is indicated by a small icon (a hand) left of the source line. If the

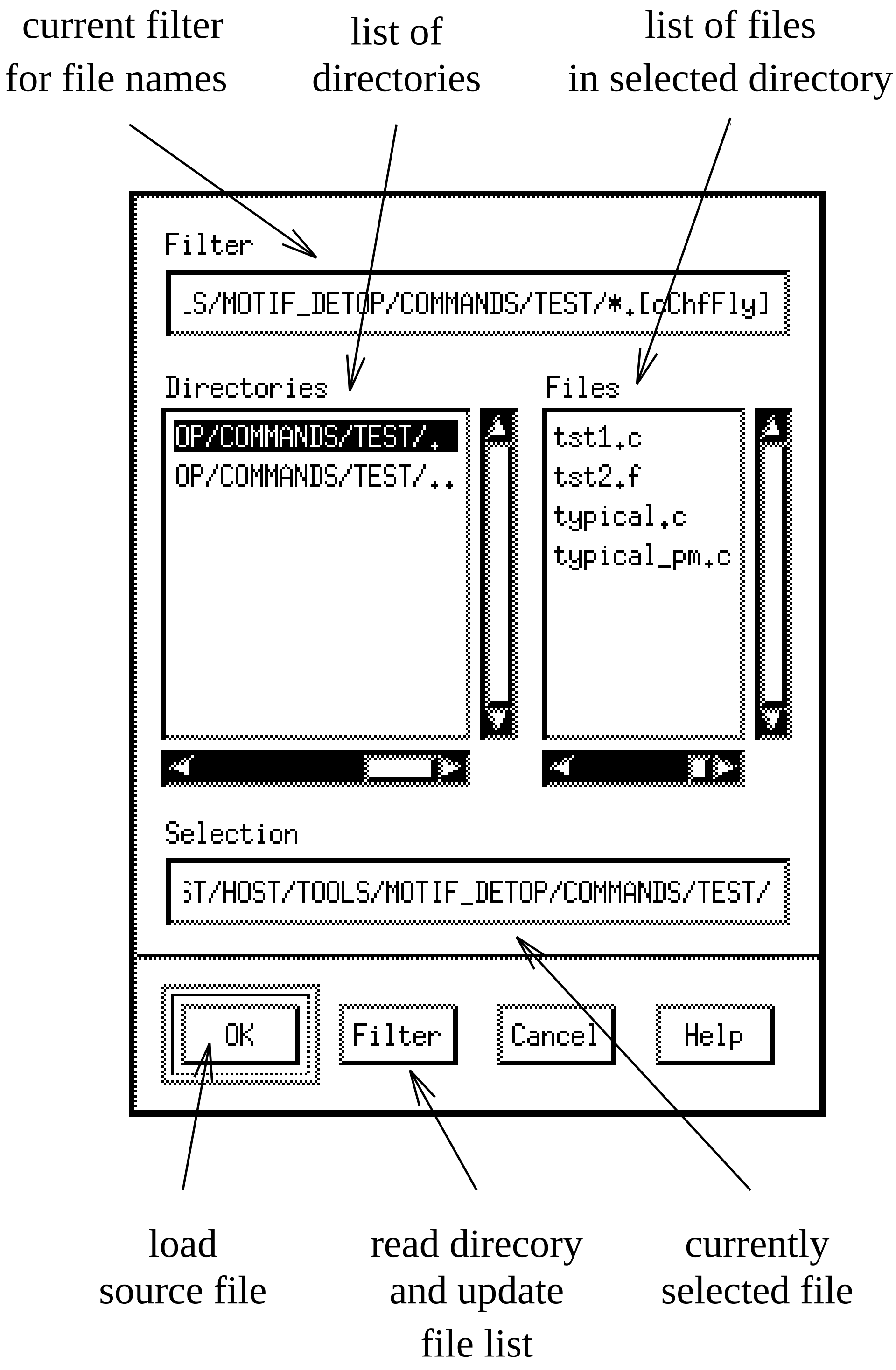


Figure 1.3: Source file selection box

Breaks on option is toggled in the Options menu, an additional text will be inserted just above the breakpoint line (see Fig. 1.1). This text shows the breakpoint number, its position and state, the threads that may trigger this breakpoint and the action to be performed once the breakpoint is triggered.

Stop locations: The point of execution of a thread is shown by an arrow left of a source line. If Stops on is enabled in the Options menu, there will be also a short additional text in the source listing showing the thread's token. The text also indicates whether the thread is currently stopped or not. In the latter case, the position shown reflects the most recent information, i.e. the last position where the thread has either been stopped or inspected using the where command.

The position shown for a thread reflects the topmost element on the procedure call stack[1] for a procedure that is contained in the loaded source file. In Fig. 1.1, line 35 is displayed as the position of thread `n0.c3.t0`, since file `work.c` is loaded and line 35 of `work.c` corresponds to the top of the call stack, as you can see in the state display. If file `example.c` was loaded, the position marker would be located at line 10. In this latter case, the inserted text would indicate that the thread is actually stopped in a procedure call (namely a call to procedure `node0()`) contained in this line.

The additional information for breakpoints and stop locations can also be retrieved by clicking at the corresponding icon with the left mouse button.

[1] Assuming the stack to be growing bottom-up.

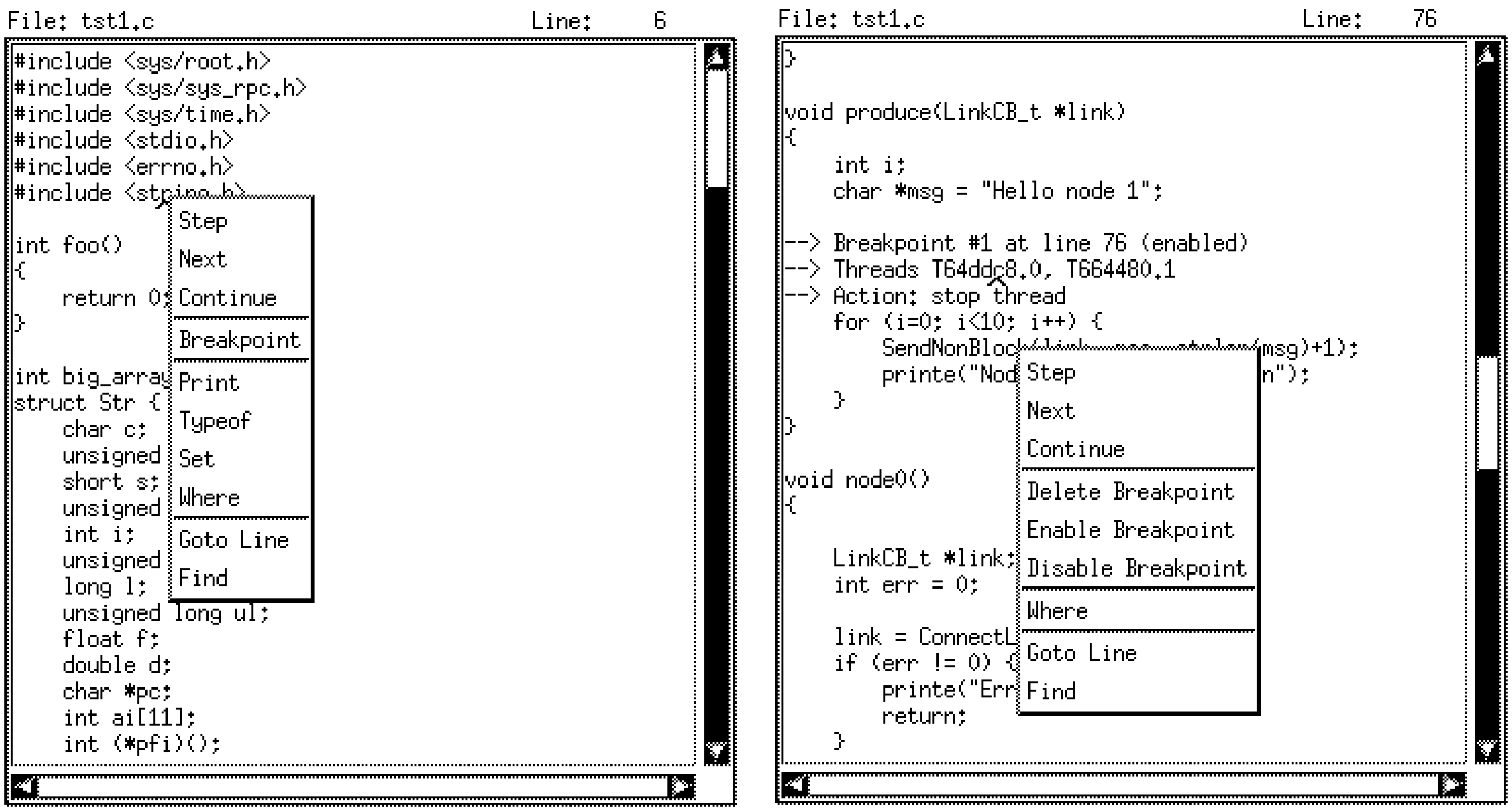
1.3.3.3 Popup-Menus

DETOP's listing window provides popup-menus for easy access to the most frequently used debugger commands. The listing window is also used to specify the parameters needed by these commands. These parameters are given by the position of the insert text cursor (the 'I'-shaped one) and/or a selected (highlighted) piece of text in the listing window. In general, the procedure to invoke a command via the popup-menu is as follows:

1. Set the insert cursor position by clicking at the appropriate position with the left mouse button or by using the cursor keys. In addition, for some commands you have to select a piece of text by dragging the mouse with the left mouse button pressed or by double-clicking a word. This step can be omitted if the insert cursor is already properly positioned or the desired text is still selected (i.e. highlighted).

2. Press the right mouse button to get the popup-menu, move the mouse cursor over the desired item and release the right mouse button to select it.

This menu is context-sensitive, so dependent on the position of the insert cursor you may get different menus. If the cursor is positioned in a line of source code, you will see the menu shown in Fig. 1.4a. If the cursor is within a breakpoint text, you will get a slightly different menu as show in Fig. 1.4b. The additional texts indicating the source positions of threads have no associated menu. Note that due to this context-sensitiveness, you will only get a popup-menu, if you have loaded a source code listing.



a) Menu for a source line b) Menu for a breakpoint text

Figure 1.4: The different types of popup menus

Here is a brief description of the menus' commands. For a more detailed description, refer to Chapter 3.

Step: Execute a single-step. All threads selected in the thread list will be executed until they reach a different source line.

Insert cursor position and selected text are ignored by this command.

Next: For each thread selected in the thread list perform a single-step; treat procedure calls as one step.

Insert cursor position and selected text are ignored by this command.

Continue: Start or resume all threads selected in the thread list.

Insert cursor position and selected text are ignored by this command.

Breakpoint: Set a breakpoint at the source code line containing the insert cursor. The breakpoint may be triggered by any thread that is selected in the thread list while all other threads pass the breakpoint line undisturbed. Once the breakpoint is reached, the default breakpoint actions are performed.

The selected text is ignored by this command.

Delete Breakpoint: Delete a breakpoint. The breakpoint to be deleted is identified by the insert cursor position.

The selected text is ignored by this command.

Enable Breakpoint, Disable Breakpoint: Enable or disable the breakpoint identified by the insert cursor position. These commands are useful to temporarily switch off some breakpoints without having to delete them. After being set, a breakpoint is enabled by default.

The selected text is ignored by this command.

Print: Display the value of a source code variable or expression that is defined by the text selected in the listing window. The position of the insert cursor defines the procedure stack frame the value is to be read from. If the cursor is located within the body of a procedure, the variable's value in the most recent activation of this procedure is printed[2]. Each variable name is searched for according to the usual scoping rules, i.e. the procedure's local variables are considered first, then global statics and last global variables.

Two examples may clarify this:

1. In Fig. 1.1, if the text `args` in line 29 of the listing is selected, the Print command will try to read the value of variable `args` in procedure `SendNonBlock()`. Since `args` is a local variable of this procedure, its value in the topmost (i.e. most recent) activation of `SendNonBlock()` is printed.
2. If the same string is selected in line 44 (that is in the body of procedure `produce()`, see the state display

[2] If the cursor is located in the procedure's header, i.e. before the opening curly brace in C or in the line containing the FORTRAN 77 keyword `subroutine`, the procedure cannot be determined due to insufficient symbol table information.

in Fig. 1.1), the `Print` command will try to read the value of `args` in procedure `produce()`. Since there is no local variable with that name in `produce()`, the debugger will consider global static and global variables. If there is still no matching variable, an error message is displayed.

Note that for recursive procedures you will only get the values of local variables in the most recent procedure incarnation in this way. To print the values of variables in previous incarnations you must use the command line (see 3.7).

Typeof: Print the type of the variable or expression defined by the selected text in the source listing.

The remarks for the `Print` command also apply here.

Set: Change the value of a variable. You will be prompted for the new value by a small dialog box. The selected text in the listing, concatenated with an assignment symbol (`=`) and the text given in the dialog box must be a valid assignment statement. So usually you will select a variable name in the listing and enter a constant in the dialog box. However, more complex assignments (including side effects in C) are also possible.

The remarks for the `Print` command apply analogously.

Where: Print the current procedure call sequence of the threads selected in the thread list. The line corresponding to the topmost position in the call stack that is contained in the loaded source file will be marked by an arrow; if the Stops on option is selected, an additional stop text will be inserted into the listing.

Insert cursor position and selected text are ignored by this command.

Show Line: Allows to quickly scroll to a given source line in the listing. The line number is requested by a small dialog box.

Insert cursor position and selected text are ignored by this command.

Find: Pops up a dialog box that allows to search the source listing for occurrence of a given text. You can search forward and backward, starting from the current insert cursor position.

The selected text is ignored by this command.

1.3.4 Command Line

All debugger commands offered by the popup-menus, except for the Goto Line and Find command can also be entered by the debugger's command line. The command line offers the standard Motif editing functions and an additional input history. You may browse through this history using the CURSOR-UP and CURSOR-DOWN keys to get the previous or next text, edit an old command and enter it again by pressing the RETURN key. An empty input results in the last input being used again.

Here is a short description of the command syntax used. More details are provided in Chapter 3. Note that all commands must be terminated by the RETURN key. Also note that all commands may be abbreviated by just a few characters. The shortest abbreviation is underlined below.

`step`, `next`, `where`

These commands don't need any parameters, so just typing the command name or an abbreviation is sufficient. The commands will then be applied to all threads selected in the thread list.

`continue` [`all`], `stop` [`all`]

These commands may be used without parameters to continue or stop the threads selected in the thread list. You may also add `all` (separated by a blank) to continue or stop all threads of your application.

`break` [*file_name*:] { *line_number* | *proc_name* } [`stop` { `thread` | `node` | `all` }]

Via the command line you can set breakpoints on either source lines or procedure entry points by specifying either a line number or a procedure name. For line breakpoints in the loaded source file and for procedures with a unique name the *file_name* and the colon may be omitted. If the action is omitted, the default break action is used (see Section 1.3.1). The breakpoint can be triggered by the threads selected in the thread list.

Examples of valid breakpoint commands are:

```
break foo.c:foo stop all

b 14

bre example.c:123 stop thread
```

`delete` { *breakpoint_number* }$^+$,
`enable` { *breakpoint_number* }$^+$,
`disable` { *breakpoint_number* }$^+$

These commands take a sequence of breakpoint numbers, separated by blanks, as an argument and will delete, enable or disable all specified breakpoints. The breakpoint number can be taken from the output window, where it is printed when a breakpoint is defined. It is also contained in the breakpoint text shown in the listing window and in the breakpoint icon's information popup window.

`print` [*context*] *expr*,
`typeof` [*context*] *expr*,
`set` [*context*] *expr*

These commands take an optional context[3] specification and a valid C or FORTRAN 77 (assignment-)expression as an argument. Without a context specification, the topmost procedure in the call stack that has been compiled with the `-g` flag defines the context for evaluating the expression and also the language that must be used for *expr*. For example, for the procedure call stack of thread `n1.c3.t0` in Fig. 1.1 this procedure is `consume()`. Since this procedure is contained in a C source file, the given expression must follow C syntax.

To access variables not in the scope of this procedure, you must provide a context specification. The exact syntax and semantic of this specification is given in Sections 3.7, 3.8 and 3.9.

[3]Do not confuse this with the term 'context' used in 1.3.2.1 in combination with the `Execute()` function. Here 'context' means a context for the evaluation of an expression within a thread.

Examples of valid commands are:

```
pr ‘int.f:integr‘ mat(i) + (1.2,2.0) ** y
p i
set ‘foo.c:foo(2)‘ p = (char **)s->c[i++]
```

In addition to the parameters mentioned above, each command accepts a comma-separated list of thread identifiers enclosed in square brackets as its first parameter. In this way you may override the selection in the main window's thread list for a command. So `continue [n1.c3.t0]` will always continue this thread, regardless of the selection in the thread list.

1.3.5 State Display

In contrast to the output window, the state display shows information that is, at least in some sense, always up-to-date. The information shown for a thread always corresponds to the last point where the thread was stopped. Whenever the thread is halted by the debugger, the information will be updated in-place. Currently, only procedure tracebacks are displayed in this window.

Note that in the current implementation, this window is cleared when you change the main window's thread list via the thread selection window.

1.3.6 Output Window

The output window shows the history of all issued commands and their output. For convenience, commands are recorded in the correct command line syntax, so you may use the cut-and-paste mechanism to re-execute a command.

Chapter 2

An Example Session

2.1 Introduction

The example used in this chapter is a small producer/consumer program. You will find this program in the `$PARIX/examples/Detop` directory, together with a makefile for its compilation and two shell scripts that can be adapted to ease the starting of DETOP. The main file of our example is also shown in figures 2.1 and 2.2. The procedures `node0()` and `node1()` are the continuation of `main()` corresponding to the nodes the program is running on. They initialize the communication link before they call the actual working procedures `produce()` and `consume()`. The producer uses a nonblocking, nonreliable send operation. This operation is implemented by creating an extra thread that uses a blocking send operation (`SendLink()`). The consumer is simply waiting at the given link using `RecvLink()`.
The intention of the producer is to send ten messages, what he seems to confirm by printing ten times the message "Node

0: sent to node 1". But, however, the consumer receives none. Even with a nonreliable send, this should not happen.
To find out more about this typical message passing problem the appropriate tool is DETOP. But to use DETOP it is necessary to compile the program with the `-g` flag, i.e. if not already done, recompile it. Generally it is a good idea to use the `-g` flag as default during program development.
Below there is an example session on how to get some more informations about the program error using DETOP. For several commands there exist two possibilities to obtain them. In this case the alternative will be described as well or there will be a short comment in brackets.

```c
#include <sys/root.h>
#include <sys/sys_rpc.h>
#include <sys/time.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>

struct Args {
        LinkCB_t *link;
        void *msg;
        int  len;
};

int
SendProc(void *args)
{
        struct Args *argptr = (struct Args *)args;

        SendLink(argptr->link, argptr->msg, argptr->len);

        return(0);
}

void
SendNonBlock(LinkCB_t *link, void *msg, int len)
{
        Thread_t *send_thread;
        int error = 0;
        struct Args args;

        args.link = link;
        args.msg = msg;
        args.len = len;

        send_thread = CreateThread(NULL, 10000, SendProc, &error,
                              (void *)&args);
}

void produce(LinkCB_t *link)
{
        int i;
        char *msg = "Hello node 1";

        for (i=0; i<10; i++) {
                SendNonBlock(link, msg, strlen(msg)+1);
                printe("Node 0: sent to node 1\n");
        }
}

```

Figure 2.1: Producer/consumer example (file work.c)

```
void node0()
{

        LinkCB_t *link;
        int err = 0;

        link = ConnectLink (1, 1, &err);
        if (err != 0) {
                printe("Error node 0\n");
                return;
        }
        produce(link);
}

void consume(LinkCB_t *link)
{
        int i, sz;
        char buf[100];

        for (i=0; i<10; i++) {
                sz = RecvLink(link,buf,100);
                printe("Node 1: received %d bytes (%s) from
                        node 0\n",sz,buf);
        }
}

void node1()
{
        LinkCB_t *link;
        int err = 0;

        link = ConnectLink (0, 1, &err);
        if (err != 0) {
                printe("Error node 1\n");
                return;
        }
        consume(link);
}
```

Figure 2.2: Producer/consumer example (file work.c, cont.)

2.2 Debugging Session

Starting The Debugger

Starting the debugger happens as follows:

start monitor

First of all you have to start the monitor:

```
px run -f0 2 1 $PARIX/tbin/monitor.px
```

This assumes that you don't use `nrm` and DETOP will be started on the front end.

start DETOP

If the monitor is up start DETOP itself:

```
px detop -- example.px
```

You may also use one of the shell scripts provided in `$PARIX/examples/Detop` to start both monitor and DETOP with a single command, e.g.

```
px run_detop 0 2 1 example.px
```

select file

After DETOP appears upon the desktop, load work.c by choosing the File:LoadFile menu (see 1.3.3.1).

Getting Closer To The Bug, Step 1

Now it is important to get closer to the error step by step. For one part this will be achieved by setting breakpoints at appropriate code lines and procedures. Additionaly the single step command is used to execute one source code line after the other. Since the receiving procedure `consume()` does not do anything, the assumption is made that the error occurs in the producing part of our example.

select threads

The first formal act is to select all threads available at the start of the program. These are the initial threads on each node. In this case you will find the threads `n0.c3.t0` and `n1.c3.t0` in the thread selection window, which you get by clicking at the File:SelectThreads menu item (see 1.3.2). For our example, select all nodes in the window's node list. Afterwards select all threads in the middle list and press the > button to add these threads to the right list. Eventually you have to press OK. All selected threads will appear afterwards in the thread list at the left side of the main window. Here you have to choose all threads the following command(s) are supposed to apply to. The resulting initial state of DETOP is shown in figure 2.3.

set breakpoint #1

The first attempt to get some informations about the communication problem is to set a breakpoint at line 44 into the body of the procedure `produce()`. Since the producer runs on node zero, the initial thread of this node must be chosen in the thread list (n0.c3.t0). Afterwards the cursor must be positioned at the line where you want to set the breakpoint. A breakpoint is always connected with a dedicated action. To select an action choose the Defaults:Actions item of the main menu. But since we only want to stop the thread triggering the breakpoint, which is the default setting, nothing has to be done. Now you can set the breakpoint by clicking into the source code window with the right mouse button and choosing

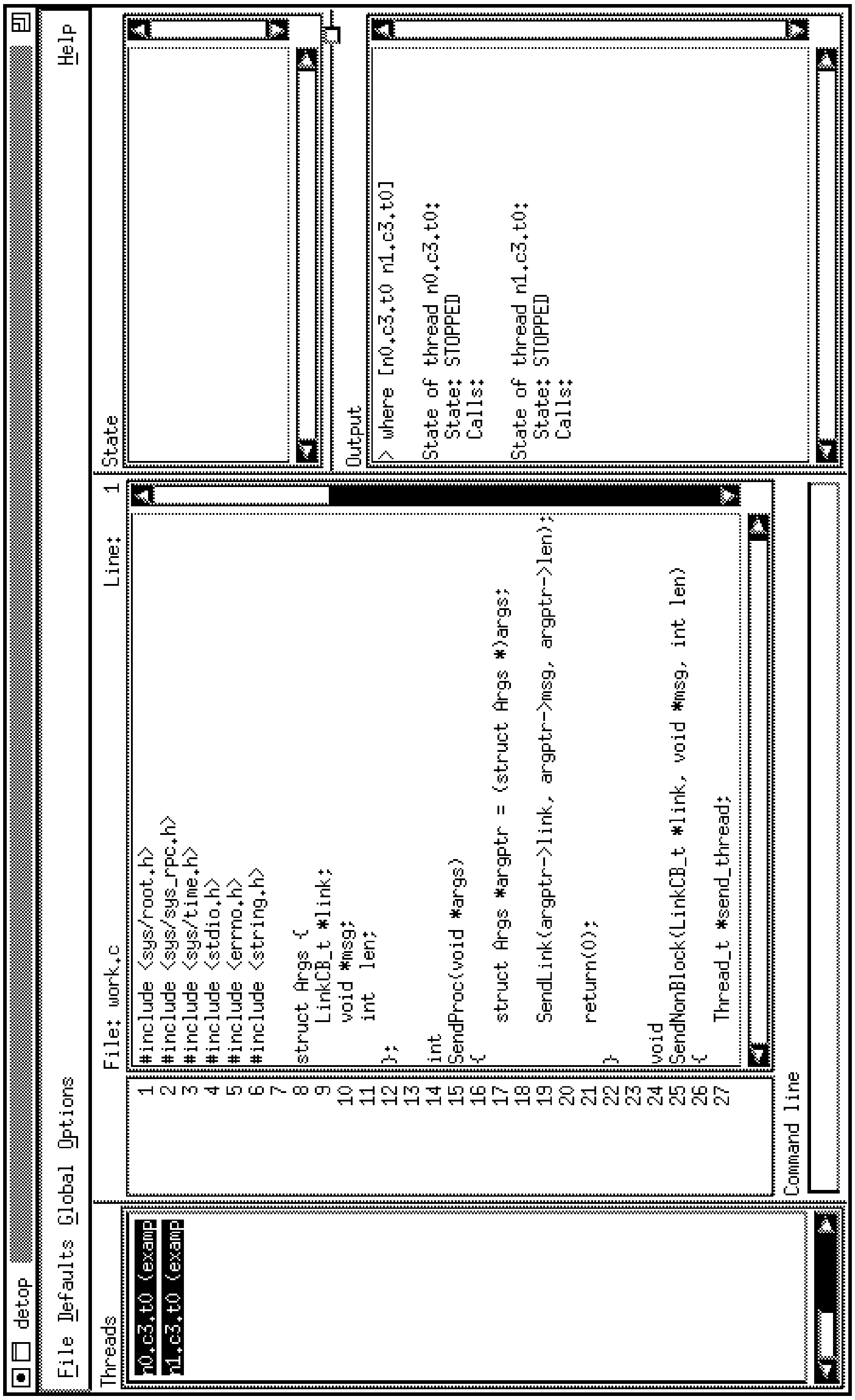


Figure 2.3: Initial state of DETOP

the Breakpoint button from the popup menu (see 1.3.3.3). But as alternative you also might use the command line beneath the source code window. Here he would have to type `break 44` (see 1.3.4). Now you can see the definition of the breakpoint in the source code and the confirmation in the output window (see figure 2.4).

continue

Now start all threads. The command to start threads the first time and to continue threads after a breakpoint is the same. This command is called Continue. First select all threads the command is supposed refer to, in this case all. Continue can again be obtained either via the source code code popup menu or the command line (see 1.3.3.3 and 1.3.4). The command line command is simply `continue`. Another possibility in this case would be to choose the Global:Continue All item of the main menu to start all existing threads.

breakpoint #1 reached

The occurrence of the breakpoint is shown in the output window together with the thread triggering the breakpoint. Since only one thread runs on the code where the breakpoint is set only this one is announced, while the other one, the consumer, is waiting for a message.

print

Now let's have a look at the variable `link` of the producer thread for two reasons. First check whether it contains a reasonable value and second to be able to verify its contents in the following procedures. Select thread `n0.c3.t0` and mark the variable `link` anywhere in the body of the

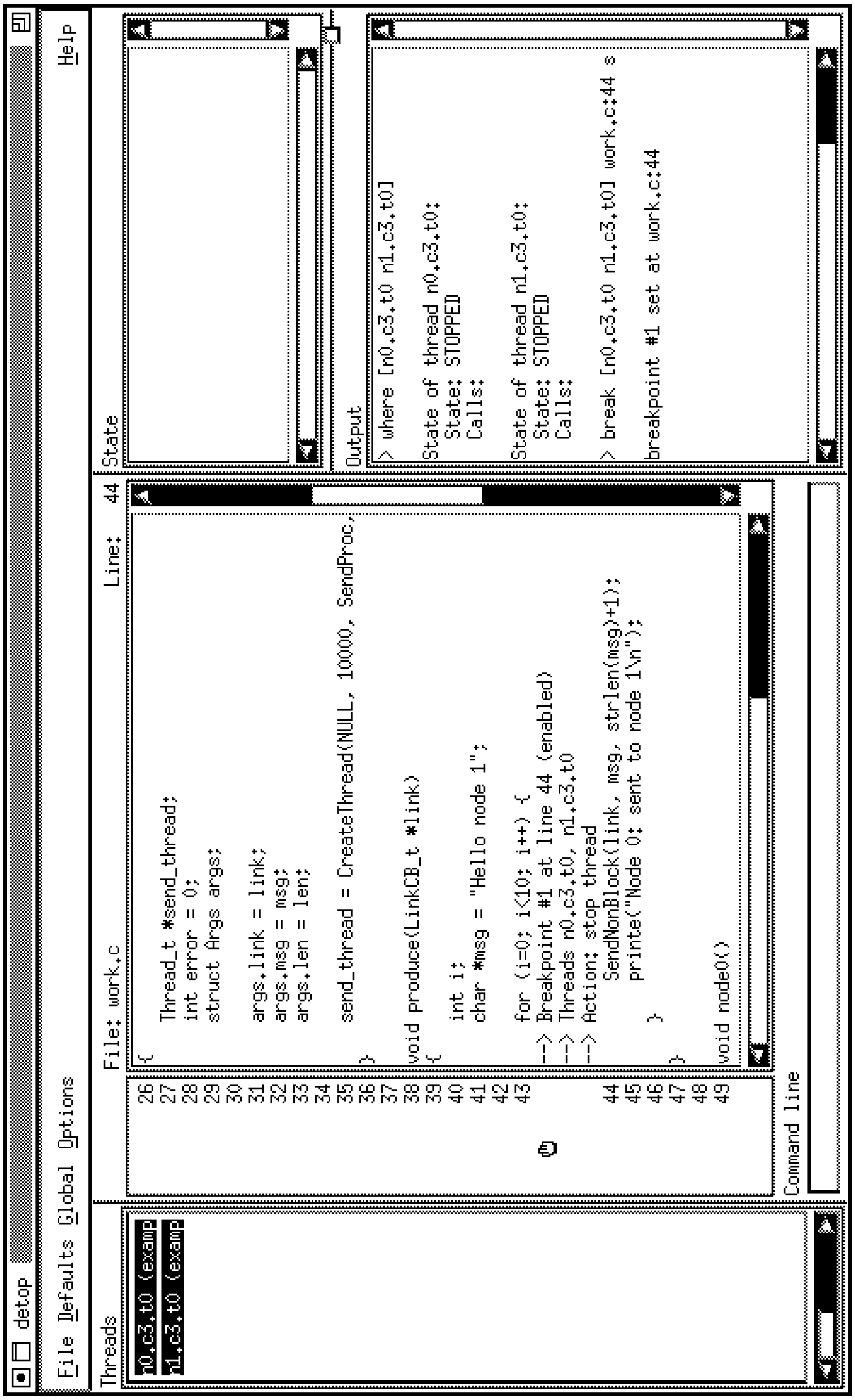


Figure 2.4: After setting the first breakpoint

procedure `produce()` with the left mouse button. Afterwards choose the popup item Print (Or select the thread and type `print link`). The contents of the variable will be displayed in the output window. It might have the value shown in figure 2.5.

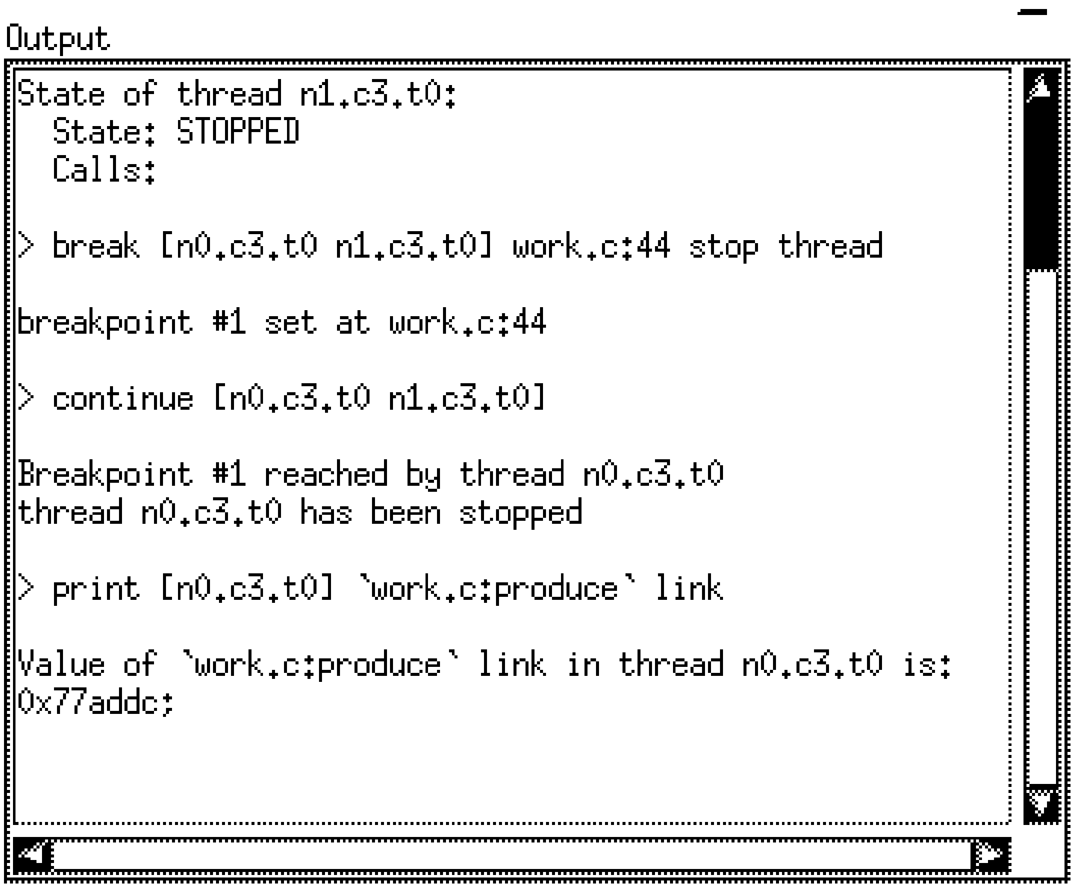


Figure 2.5: Output window displaying the value of `link`

Getting closer to the bug, Step 2

set breakpoint #2

Since the value of `link` seems to be all-right, the examination must go on. To that effect it is useful to set another breakpoint stopping the `SendNonBlock()` procedure at its beginning. You can do that in the usual manner by positioning the cursor (at line 31) and choosing the breakpoint item from the popup menu. The selected thread is `n0.c3.t0`. But it is also possible to simply type `break SendNonBlock` in the command line. The result won't exactly be the same. The thread stops before the first assignment, that is in this case in line 28. This effects that you have to perform one more single step as mentioned below.

continue

The `Continue` command has now to affect the same thread as the preceding breakpoint command. Therefore it is not necessary to select any other threads. That means merely select the `Continue` item in the popup menu (type `continue`).

breakpoint #2 reached

The breakpoint #2 should be reached at this moment by thread n0.c3.t0.

single step

The next goal is to examine the structure `args` after initialization. Therefore three (four, if `break SendNonBlock` has been used) single steps are executed. The selected thread is still valid. Therefore it is only necessary to click at the `Step` item of the popup or typing `step`. To come to the first line after assigning `args` it is necessary to repeat the single step two (three) more times. If using the command line this is easy to achieve since you only have to press RETURN to repeat the last command. You can

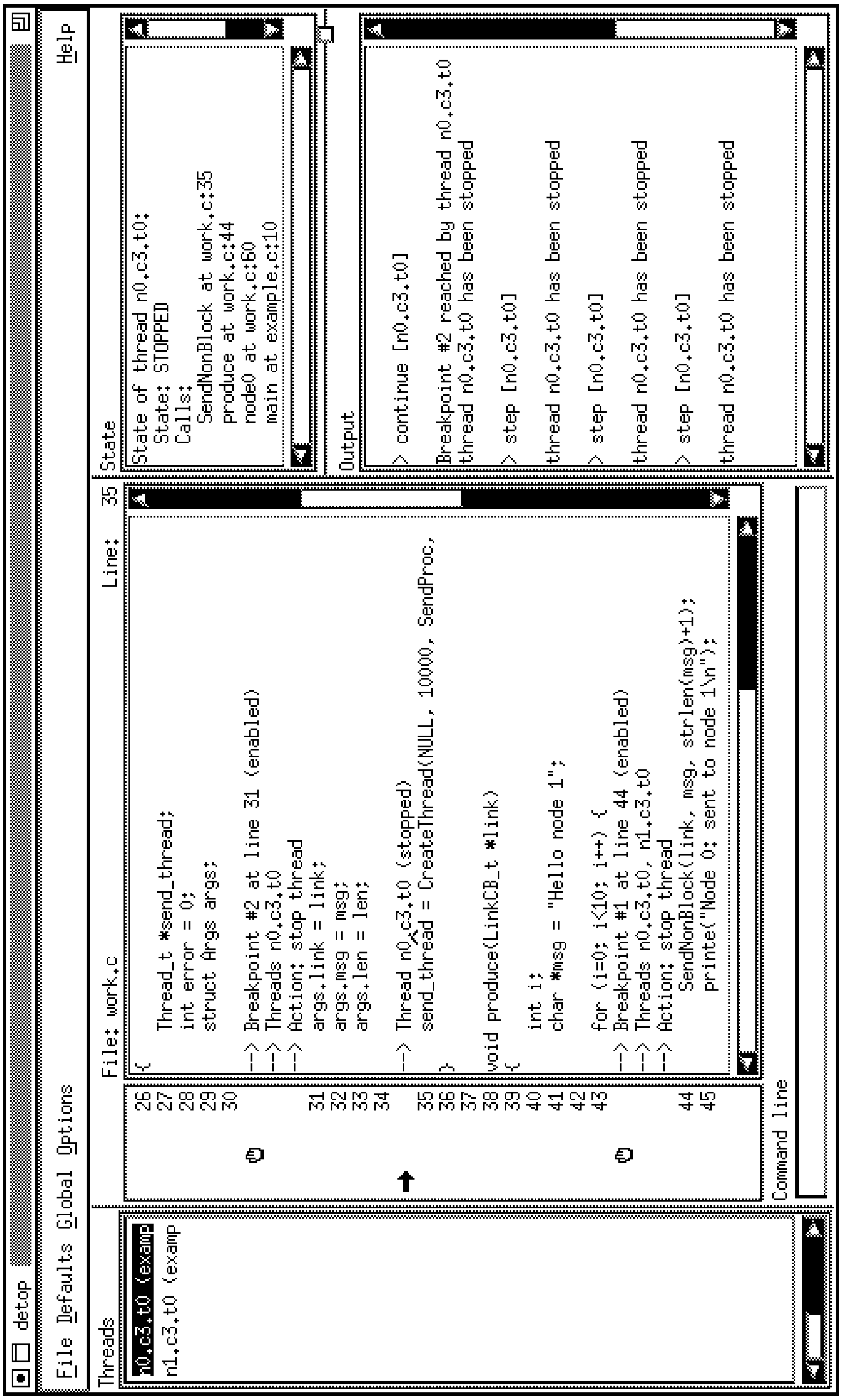


Figure 2.6: Result after three single steps

see the final result in figure 2.6.

print structure

Now it is possible to have a look at the whole structure representing `args`. Mark `args` in the body of `SendNonBlock()` and choose Print or type `print args`. The structure will again be shown in the output window (see 2.7).

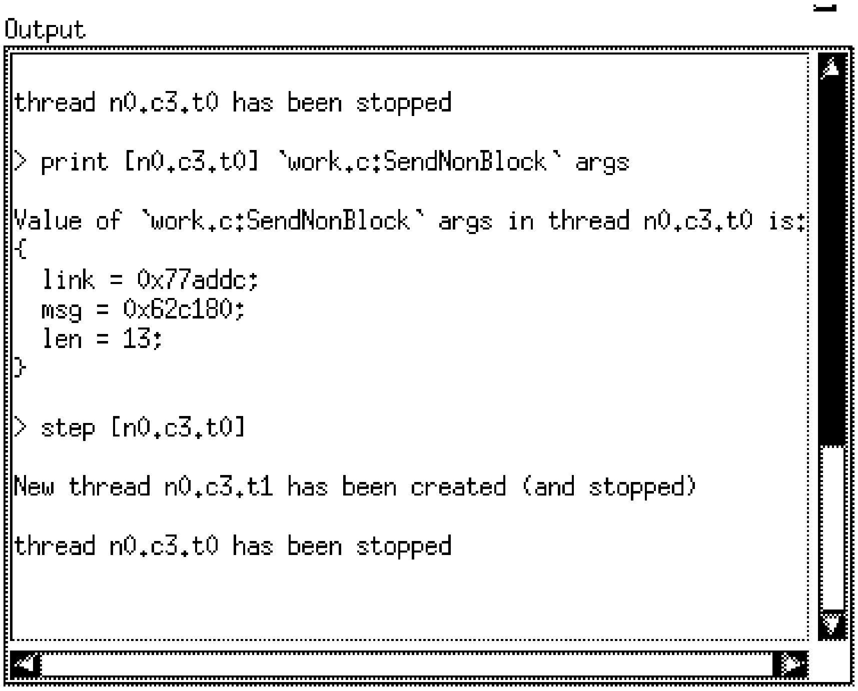


Figure 2.7: The structure `args` and the announcement of the new thread

stop new threads

The contents of `args` is still all-right. Therefore it is necessary to have a look at the new, dynamically created thread, dealing with the asynchronous send operation. To cause the debugger to stop the new thread at the beginning the Stop New Threads option must be switched on. The option can be found in the main menu Global.

single step

Now we perform a single step over the `CreateThread()` call, which creates the new thread based upon the `SendProc()` procedure. To do that choose Step in the popup or type `step`. Afterwards the output window confirms the operation and signals the appearance of a new thread. The current output can be found in figure 2.7.

select new thread

As we have seen the thread has been created. Now it is necessary to select the new thread to be able to apply commands to it. Therefore click at File:Select Threads of the main menu. The Thread selection window appears. Since the new thread is running on node zero (can be seen at the beginning of the global id: n<*node*>.c<*context*>.t<*thread*>) choose Node 0 in the left list. The new thread `n0.c3.t1` appears now in the

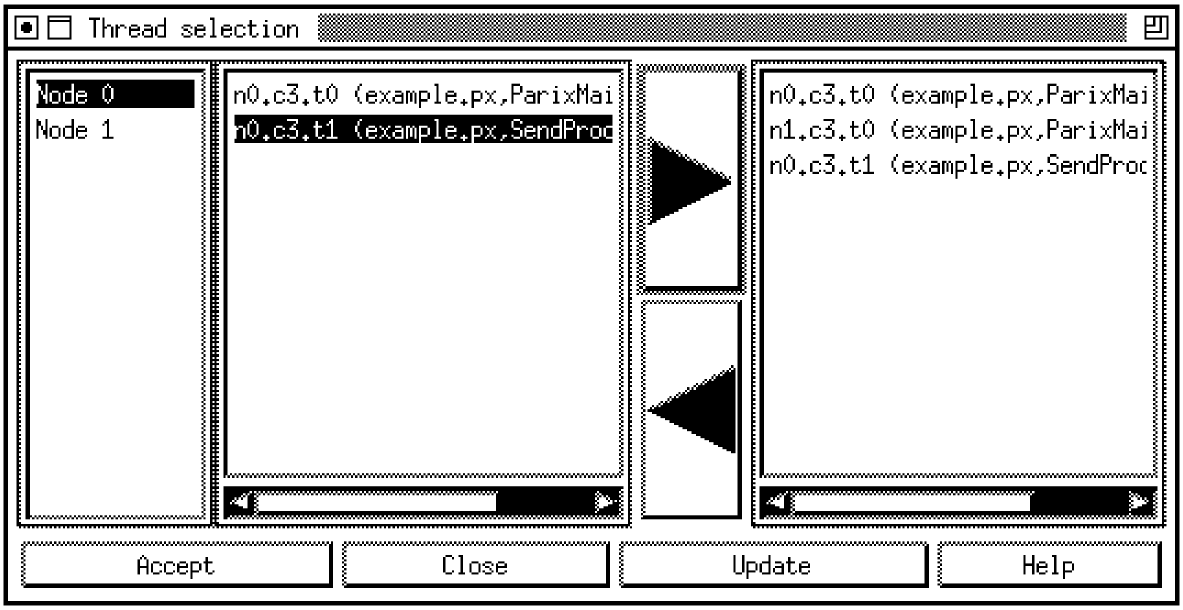


Figure 2.8: Selection of the new thread

middle list. Click on this thread, add it to the right window and confirm by clicking the Accept button (for more informations see 1.3.2). The thread will now appear in the main window's thread list window.

Inspecting the new thread

set breakpoint #3

To examine the new thread it is necessary to set a breakpoint inside the body of the corresponding procedure. To that effect select the new thread in the thread list and put the cursor to line 19. Now choose the Breakpoint item in the popup window. As alternative after selecting the thread it is possible to use the command `break 19`. The final window state is shown in figure 2.9.

continue

To reach the breakpoint select the Continue command (or type `continue`).

breakpoint #3 reached

The breakpoint is reached by thread `n0.c3.t1`. The according output is shown in figure 2.9

alternative: step over assignment

A much more easier way to reach the first line of the new thread is simply to perform a single step over the assignment in the definition part of the procedure. The result is the same, but you need only one command instead of three.

print contents of pointer

Now let's have a look at the contents of the `argptr`. To

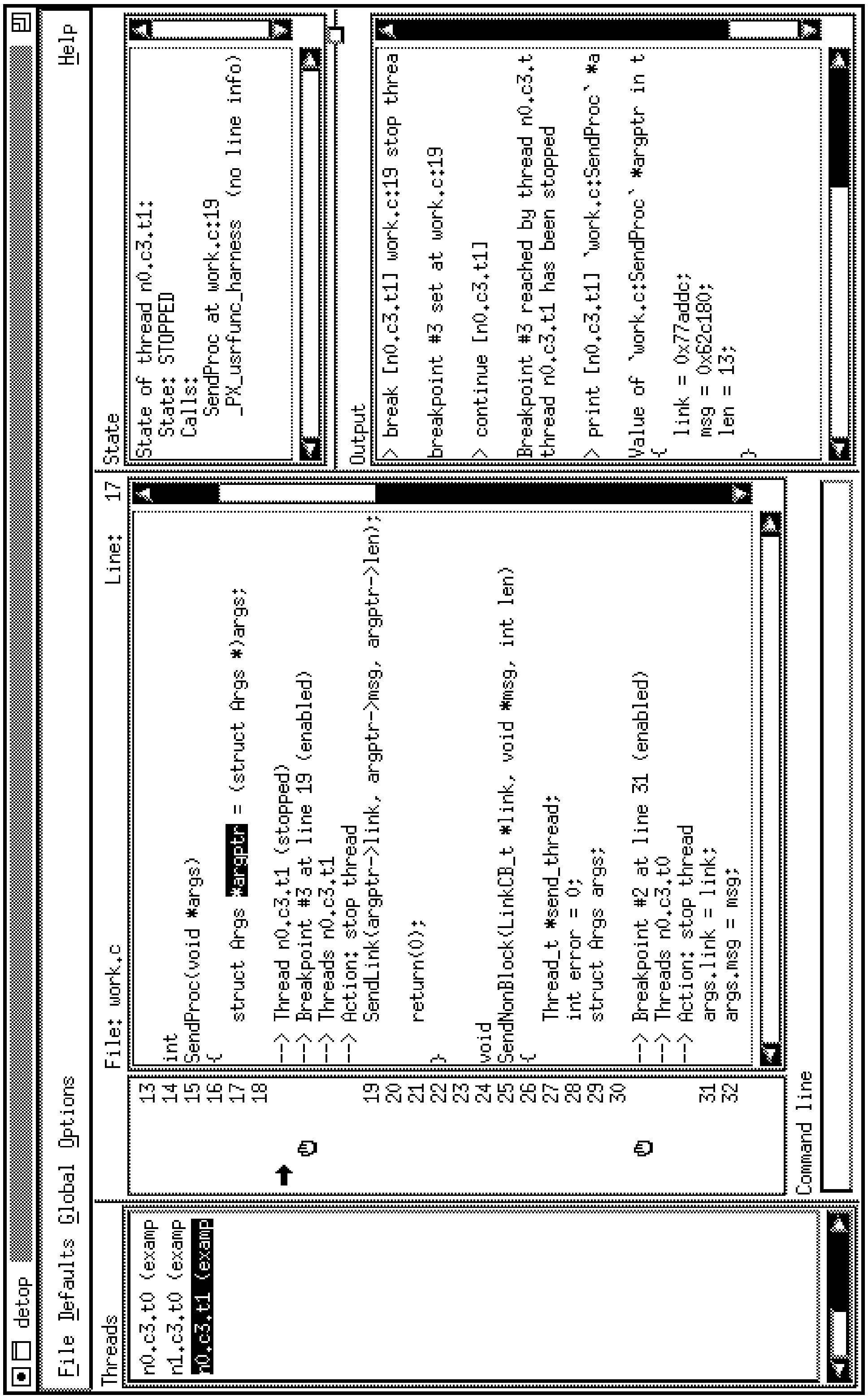


Figure 2.9: First output of argptr

get the contents of the memory the pointer points to, mark the variable including the star operator, i.e. mark `*argptr` in the body of `SendNonBlock()` and select `Print` from the popup menu (type `print *argptr`). The output window shows the same contents as before, as you can see from figure 2.9.

Fixing the bug

continue n0.c3.t0

It looks like everything is all-right so far. That means the error must have been caused by a side effect of the current program situation. On node zero there exists only one more thread currently stopped at line 31 (breakpoint #2). Continuing this thread it will subsequently reach breakpoint #1, defined at the beginning of the session. But to continue thread `n0.c3.t0` it has to be selected in the thread window. Afterwards it can again be started as usual by selecting the `Continue` item or typing `continue` in the command line.

print *argptr

After thread `n0.c3.t0` has reached breakpoint #1 we change back to thread `n0.c3.t1` in order to print `*argptr` again. That means the command above has to be performed again. The result of this request is shown in figure 2.10. The contents of the structure has changed and the values do not look appropriate. But why does the structure now contain trash and not before ? The reason is that the structure is a local variable and only a pointer is passed to `SendProc()` in `CreateThread()`. Therefore the contents is trashed upon the exit of `SendNonBlock()`.

The `*argptr` should be allocated in `SendNonBlock()` and the memory should be set free in `SendProc()`.

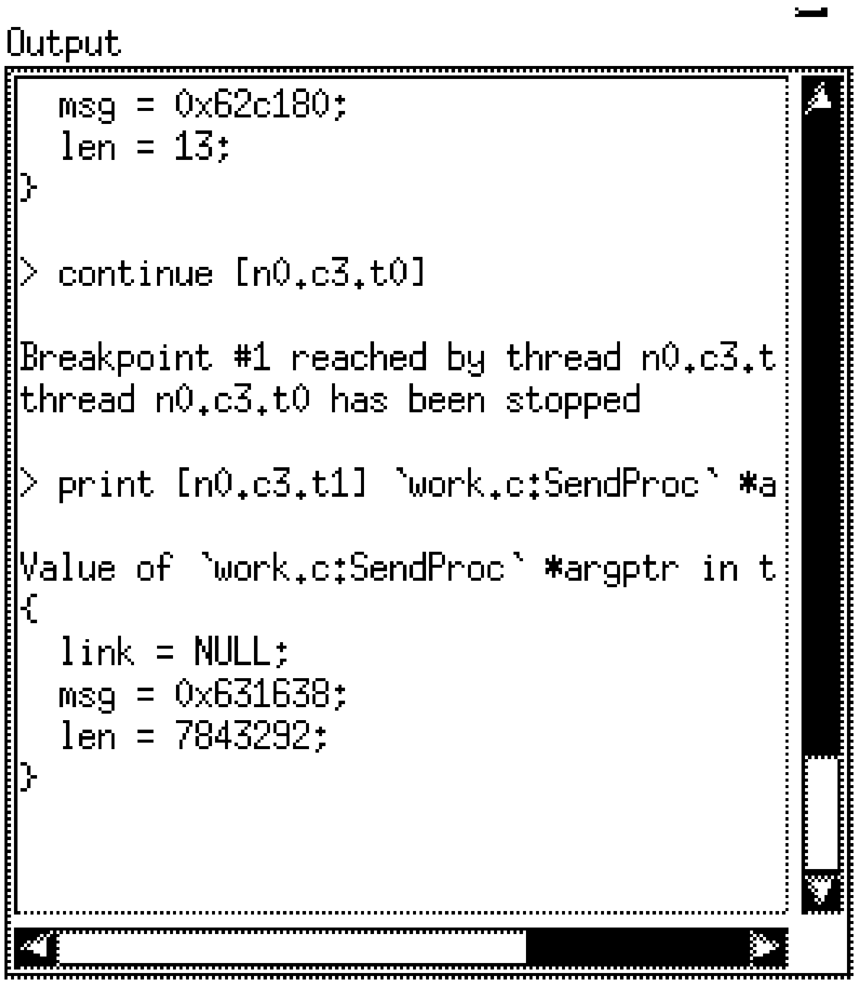


Figure 2.10: Output window showing the error

Chapter 3

Command Reference

The following pages contain a reference for all real debugger commands, i.e. all commands interacting with your application program. The descriptions contain the following sections, if appropriate:

DESCRIPTION: Gives a short description of the command's goal and effects.

MENU INPUT: Explains how the command can be invoked via the pulldown or popup menus.

COMMAND LINE INPUT: Specifies the syntax of the command when using the command line.

OUTPUT: Describes the type of information that will be displayed by DETOP upon completion of the command.

DIAGNOSTICS: Lists the error messages that may be created by this command and gives hints how to avoid them.

NOTES: This section gives information on behavior that might be slightly different from what you are expecting.

BUGS: Known bugs of this command.

3.1 Stopping New Threads

DESCRIPTION:

A large class of parallel programs dynamically creates new threads during run-time. In order to be able to debug such threads, there is an option Stop New Threads in the Global menu.

If this option is set, all subsequently created threads will be stopped immediately before they execute their first instruction. This option also works with the PARIX `Execute()` function, since PARIX will create a new thread for the loaded code. Once a new thread has been created, you will be informed about that fact and may then add this thread to the thread list using the Thread Selection item in the File menu.

If this option is off, you will not be informed about thread creation. Any dynamically created thread will start working immediately. Of course, you may still use the thread selection window to add new threads to your thread list.

MENU INPUT:

1. Move the mouse cursor to the main menu bar.
2. Pull down the Global menu by pressing the left mouse button.
3. Select Stop New Threads to toggle this option. The current setting is displayed in the menu item.

OUTPUT:

For each thread that is created while the Stop New Thread option is on, you will get a message 'New thread ... has been created (and stopped)' in the output window.

3.2 Break: Defining Breakpoints

DESCRIPTION:

In general, a breakpoint definition consists of some predicate and associated actions. The debugger will then automatically perform the given actions each time the predicate is satisfied. In the current version of DETOP, predicates are restricted to execution predicates that are satisfied when a thread reaches a given code location. Also the actions are restricted to stopping threads. These restrictions will be loosened in future versions.

The breakpoint predicate consists of the set of threads selected in the thread list or given in the command line and a position in the source code. This position is defined either by a source line that generates executable code or by a procedure. The breakpoint is triggered whenever one of the specified threads is about to execute the given source line or procedure.

When a thread has triggered the breakpoint, dependent on the selected actions, either this thread alone, all threads on the same node or the whole application will be stopped. Use the `Continue` command to resume execution of the stopped threads.

MENU INPUT:

Using the popup menu, you can only set breakpoints on source lines in the currently displayed source file. Follow these steps:

1. Select the threads that shall be able to trigger the breakpoint in the thread list.

2. Move the mouse cursor to the listing window.
3. Click at the desired line with the left mouse button or use the cursor keys to move the insert text cursor to that line.
4. Press the right mouse button to pop up the menu.
5. Select the Breakpoint item.

A breakpoint set in this way will always have the default actions associated with it. You can change these actions prior to setting a breakpoint in the following way:

1. Move the mouse cursor to the main menu bar.
2. Pull down the Defaults menu by pressing the left mouse button.
3. The current default action is shown in the Actions menu item.
4. To change the default, pull down this menu item's submenu and select the new default action.

There are three possible actions:

Stop Thread: Stop only the thread that has triggered the breakpoint. All other threads continue operation until they are stopped by another breakpoint or a Stop command.

Stop Node: Stop all threads on the processing node where the breakpoint was triggered. This action is useful if you use shared memory between threads, since stopping the whole node prevents shared data from being modified by other threads.

Stop All: Stop all threads on all nodes. This action freezes the global state of computation, so you can inspect this state in a consistent way.

COMMAND LINE INPUT:

`break` [*thread_list*
] [*file_name*:] { *line_number* | *proc_name* }
[`stop` { `thread` | `node` | `all` }]

The optional *thread_list* overrides the thread selection in the main window's thread list. It consists of a comma-separated list of thread identifiers enclosed in square brackets:

thread_list ::= `[` *thread_id* { `,` *thread_id* }* `]`

file_name and *line_number* or *proc_name* define the code location to be used for the breakpoint. To set a breakpoint on a source line, specify *file_name* and *line_number*. If you omit the file name, the breakpoint is set in the file currently displayed in the source code listing. By specifying a procedure name (*proc_name*) instead of a line number, you can also set breakpoints on entry of a procedure. Again, you may omit the file name, if there is only one procedure with that name in your program. Note, that if you specify a file name, there must be no spaces before or after the colon.

You can also specify the action associated with the breakpoint. If you omit the action, the default action is used (see MENU INPUT).

OUTPUT:

After setting the breakpoint, a message containing the assigned breakpoint number is printed in the output window. The breakpoint number is needed for later manipulations of this breakpoint via command line. In addition, the position of the breakpoint will be marked in the source code listing.

When the breakpoint is triggered by a thread, you will get a message 'Breakpoint #... reached by thread ...'. For each thread that is stopped due to the breakpoint action, you will get the following output:

1. A message 'Thread ... has been stopped' is printed in the output window.
2. The procedure stack is updated in the state display.
3. The position where the thread has been stopped is indicated in the source code listing.

DIAGNOSTICS:

No thread selected: You haven't selected a thread in the thread list. To enter the command, select the appropriate threads and try again.

No such thread: One of the selected threads doesn't exist or has already terminated.

Unknown module: The source file you have specified for the breakpoint is unknown to the debugger. Check for errors in typing and be sure that the requested file has been compiled with `-g`. Also be aware that some preprocessors (e.g. `yacc`) change the source file name. Unfortunately, the same is true if you have FORTRAN 77 files that are processed by the C-preprocessor (i.e. source files with suffix `.F`). The only name known to DETOP is the one passed to the actual compiler. You may determine this name from the output of the Where command.

Unknown procedure/function: The procedure name is not known to the debugger or there is more than one procedure with that name. In the latter case, you must also specify the name of the source file containing the desired procedure.

Unknown line: There is no source line with this number that has generated executable code. To avoid confusing behavior, DETOP will not set the breakpoint on a subsequent line in this case.

BUGS:

Due to a bug in the Motif text widget, the source listing's scrollbar is confused if a stop or breakpoint marker is inserted in front of the visible part of the listing. Therefore, you can sometimes not scroll to the very beginning of the listing using the scroll bar. However, the cursor and page-up/down keys will work.

If a thread has aquired some system semaphore in a PARIX call and is stopped due to a Stop all or Stop node action, other threads may not be able to continue before the semaphore is released again. Since in the current OS version it is not possible for the monitor to run in an environment isolated from the debugged program, this may also block the debugger in rare occasions.

3.3 Delete/Enable/Disable: Manipulating Breakpoints

DESCRIPTION:

While the Delete command is used to definitely remove a breakpoint, Disable and Enable can be used to temporarily switch off a breakpoint and switch it on again later. A disabled breakpoint has no influence on the program's execution, however, it still consumes resources in the monitoring system.

MENU INPUT:

Via the popup menu you can manipulate any breakpoint that has its breakpoint text shown in the source code listing:

1. Click with the left mouse button at the proper breakpoint text in the source code listing or use the cursor keys to move the insert cursor into that breakpoint text.
2. Press the right mouse button to pop up the menu.
3. Select the Delete Breakpoint, Enable Breakpoint or Disable Breakpoint item.

COMMAND LINE INPUT:

`delete` { *breakpoint_number* }$^+$
`enable` { *breakpoint_number* }$^+$
`disable` { *breakpoint_number* }$^+$

On the command line, you can specify a list of breakpoint numbers, separated by blanks, for each command. The breakpoint number can be taken from the output window, where it is printed when a breakpoint is defined. It is also contained in the breakpoint text shown in the listing window and in the breakpoint icon's information popup window.

OUTPUT:

The breakpoint markers of all affected breakpoints will be either deleted or will be changed to show the new breakpoint state (i.e. enabled or disabled).

DIAGNOSTICS:

There is no such breakpoint: There is no breakpoint with the given breakpoint number. Check for errors in typing.

3.4 Step/Next: Single-Stepping

DESCRIPTION:

The Step and Next commands are used to single-step the threads selected in the thread list or defined in the command line, i.e. each of these threads will execute the next source line and stop afterwards.

The difference between Step and Next is the handling of procedure calls in the executed source line. While Step will step into such a procedure, provided that procedure has been compiled with `-g`, Next will always execute the procedure call in a single step.

MENU INPUT:

1. Select the threads to be single-stepped in the thread list.
2. Move the mouse cursor to the listing window.
3. Press the right mouse button to pop up the menu.
4. Select the Step or Next item.

COMMAND LINE INPUT:

`step` [*thread_list*]
`next` [*thread_list*]

The optional *thread_list* overrides the thread selection in the main window's thread list. It consists of a comma-separated list of thread identifiers enclosed in square brackets:

thread_list ::= `[` *thread_id* { , *thread_id* }* `]`

OUTPUT:

Immediately after entering the command, the position markers of the selected threads will indicate that the threads are now executing. Once a thread has finished its step and is stopped again, you'll get the following output:

1. A message 'Thread ... has been stopped' is printed in the output window.
2. The procedure stack is updated in the state display.
3. The position where the thread has been stopped is indicated in the source code listing.

DIAGNOSTICS:

No thread selected: You haven't selected a thread in the thread list. To enter the command, select the appropriate threads and try again.

No stack: The current procedure stack cannot be determined. This is usually the case when the initial thread of an execution context has not yet entered the `main` function. In this case, set a breakpoint on `main` (`MAIN_` for FORTRAN 77 programs), continue the thread and try again. This error is also generated if your program has not been compiled with `-g`. In this case, you can't single-step.

No such thread: One of the selected threads doesn't exist or has already terminated.

NOTES:

If a thread blocks during execution of a single step, e.g.

in a call to `SendLink()`, you will not get a stop message and the thread's position shown in the source listing will not change. However, the debugger is not blocked, so you may single step or continue other threads to remove the blocking condition.

BUGS:

Stepping out of a procedure not compiled with `-g` (e.g. a library function) will result in an error message. So if you have stopped your program in such a procedure (e.g. due to a breakpoint on procedure entry) and want to finish that procedure, you must place a breakpoint after the call to that procedure.

Due to incomplete line information from the compiler, DETOP may display confusing source positions in C programs if you step to the end of a loop whose body is not enclosed in curly braces. DETOP will then mark the last line in the loop body which may be incorrect if the body contains an `if` statement.

3.5 Continue: Continuing Threads

DESCRIPTION:

The Continue command resumes the execution of all threads selected in the thread list or defined in the command line. If a thread has never been running before, it will be started and supplied with its initial parameters. If a thread is stopped by a breakpoint, it will continue operation from this point. A thread that has been stopped due to an exception will continue with normal expection processing, i.e. it will usually drop a core and terminate. The command does not affect threads that are already running.

A special version Continue All allows to continue all threads of your application program.

MENU INPUT:

To continue some threads:

1. Select the threads to be continued in the thread list.
2. Move the mouse cursor to the listing window.
3. Press the right mouse button to pop up the menu.
4. Select the Continue item.

To continue all threads:

1. Move the mouse cursor to the main menu bar.
2. Pull down the Global menu by pressing the left mouse button.
3. Select Continue All.

COMMAND LINE INPUT:

To continue some threads:

`continue` [*thread_list*]

The optional *thread_list* overrides the thread selection in the main window's thread list. It consists of a comma-separated list of thread identifiers enclosed in square brackets:

thread_list ::= `[` *thread_id* { `,` *thread_id* }* `]`

To continue all threads:

`continue all`

OUTPUT:

The position markers of all continued threads will be slightly changed to reflect the fact that the threads are no longer stopped.

DIAGNOSTICS:

No thread selected: You haven't selected a thread in the thread list. To enter the command, select the appropriate threads and try again.

No such thread: One of the selected threads doesn't exist or has already terminated.

3.6 Stop: Stopping Threads

DESCRIPTION:

Using the Stop command you can stop the execution of all threads selected in the thread list or defined in the command line: If a thread is ready to execute but is not scheduled, it will be stopped immediately, i.e. it will not be rescheduled until you continue this thread. A thread that is waiting at a link, semaphore or timer will be stopped immediately when it is freed from the waiting condition. If a thread is currently running, which is only possible on a multiprocessor node (as in the GC/PowerPlus), it will not be stopped before it is descheduled or executes a system call. If a thread is already stopped, the command has no effect.

The Stop All command will stop all threads of your application program.

MENU INPUT:

To stop all threads:

1. Move the mouse cursor to the main menu bar.
2. Pull down the Global menu by pressing the left mouse button.
3. Select Stop All.

COMMAND LINE INPUT:

To stop some threads:

`stop` [*thread_list*]

The optional *thread_list* overrides the thread selection in the main window's thread list. It consists of a comma-separated list of thread identifiers enclosed in square brackets:

thread_list ::= `[` *thread_id* { `,` *thread_id* }* `]`

To stop all threads:

`stop all`

OUTPUT:

For each thread that is stopped, you'll get the following output:

1. A message 'Thread ... has been stopped' is printed in the output window.
2. The procedure stack is updated in the state display.
3. The position where the thread has been stopped is indicated in the source code listing.

DIAGNOSTICS:

No thread selected: You haven't selected a thread in the thread list. To enter the command, select the appropriate threads and try again.

No such thread: One of the selected threads doesn't exist or has already terminated.

BUGS:

If a thread has aquired some system semaphore in a PARIX call and is stopped due to a Stop or Stop all command, other threads may not be able to continue before the semaphore is released again. Since in the current OS version it is not possible for the monitor to run in an environment isolated from the debugged program, this may also block the debugger in rare occasions.

3.7 Print: Show Value of Variables

DESCRIPTION:

The Print command allows to read the value of any source code variable existing at the program's current point of execution. In addition, you can also determine the value of expressions. The expression is evaluated for each selected thread and the result is printed in the output window.

Expressions must be given in C or FORTRAN 77 syntax, dependent on the language your program is written in. For mixed language programs, the language to be used is defined by the currently loaded source file, if you use the popup-menu, or by the context specification (see below), if you use the command line to invoke the Print command. In the current implementation, the expression syntax is slightly restricted compared to the language specification. The appendix contains a complete syntax definition, however, for the Print command expressions must not contain any side effects (e.g. `a[i++]`).

MENU INPUT:

1. Select a set of threads in the thread list.
2. Move the mouse cursor to the listing window.
3. Select a variable or expression in the body of a procedure.
4. Press the right mouse button to pop up the menu.
5. Select the Print item.

DETOP will then try to determine the procedure containing the selected expression from the expression's position in the loaded source file. If this succeeds, for each selected thread DETOP will search the current procedure call stack for the topmost invocation of this procedure. The expression will then be evaluated in the context of that procedure invocation.

All identifiers will be searched for according to the usual scoping rule: Identifiers defined locally in the procedure are considered first, then global static identifiers and last global ones. This search is case sensitive for C identifiers, but case insensitive for FORTRAN 77 identifiers.

As an example, consider the following source code listing and procedure call stack of a thread:

```
 1: int i = 99;
 2: int fac(int i)
 3: {
 4:   if (i == 1)
 5:     return(1);
 6:   return(i * fac(i-1));
 7: }
 8: main()
 9: {
10:   i = fac(4);
11: }
```

```
Calls:
  fac at example.c:5
  fac(3) at example.c:6
  main at example.c:10
  ParixMain (no line info)
```

If the string `i` is selected in line 4 or 6, DETOP will print the value of `i` in the most recent (i.e. the fourth) incarnation of the recursive procedure `fac`, that is 1. Note that using the popup-menu you can't determine the value of `i` in the other incarnations of `fac`. Use the command line instead (see below).

If `i` is selected in line 10, DETOP will print the value of the global variable `i`, since that is the `i` within scope of procedure `main`. Since the assignment in line 10 has not yet been executed at the shown point of execution, the result will be 99.

COMMAND LINE INPUT:

`print` [*thread_list*] [*context*] *expr*

The optional *thread_list* overrides the thread selection in the main window's thread list. It consists of a comma-separated list of thread identifiers enclosed in square brackets:

thread_list ::= `[` *thread_id* { `,` *thread_id* }* `]`

The language used for *expr*, the scope used to search variable names and the stack frame used to determine the values of local variables are all specified by a so called 'context'. The syntax of this context specification is given below. Note that there are no blanks allowed in this specification.

context ::= { ' '
| '*file_name*'
| '*proc_name*'
| '*file_name*:*proc_name*'

| '*proc_name*(*incarnation*)'
| '*file_name*:*proc_name*(*incarnation*)' }

These context specifications have the following meaning:

``: This string defines a global context, i.e. each variable name given in the expression is regarded as a global variable. Using this context you can inspect global variables that otherwise would be shadowed by local variables with identical name. In the example given above, `print `` i` will print the value 99.

Since FORTRAN 77 has no global variables (members of COMMON blocks are local variables!), the language used for `expr` is always C.

'*file_name*': Supplying a file name results in a global static context, i.e. each identifier in the expression is first searched for among the global static variables defined in the given source file. If it is not found there, the global variables are searched. In this way, you can explicitly request the value of a global static variable in a given source file.

The language used to parse `expr` is defined by the programming language used in the given file.

'*proc_name*':
'*file_name*:*proc_name*': Specifying a procedure name will result in the evaluation of the given expression in the context of the topmost activation of that procedure in the current procedure call stack. All identifiers will be searched for according to the usual scoping rule: Identifiers defined locally in the procedure

are considered first, then global static identifiers and last global ones.

The file name and the colon may be omitted if the procedure name is unique within your program. The language used to parse `expr` is defined by the programming language used in the source file containing the given procedure.

In the example given above, `print 'fac' i` will read the value of `i` in the most recent (i.e. the fourth) incarnation of the recursive procedure `fac`, that is 1. `print 'main' i` will print the value of the global variable `i`, since that is the `i` within scope of procedure `main`. Since the assignment in line 10 has not yet been executed at the shown point of execution, the result will be 99.

'*proc_name*(*incarnation*)':
'*file_name*:*proc_name*(*incarnation*)': You may also add an incarnation number to the context specification. In this way you can read the values in other than the most recent incarnations of recursive procedures. The incarnation number is 0 for the most recent incarnation and is incremented for each occurrence of the given procedure scanning the procedure call stack from top to bottom.

So `print 'fac(0)' i` is always the same as `print 'fac' i` and will result in the value 1 in the example above. `print 'fac(1)' i` will print 2, while `print 'fac(3)' i` reads the value in the incarnation of `fac` directly called by `main`, i.e. 4.

If no context is specified on the command line, DETOP will use the current context of each thread. The current context is the local context defined by the topmost activation of a procedure compiled with `-g`. In the example, `print i` is equivalent to `print 'fac' i`.

OUTPUT:

The value of the specified expression will be printed in the output window for each selected thread. Structured data types will be printed in full, each component is preceded by the component name or its array index.

DIAGNOSTICS:

No thread selected: You haven't selected a thread in the thread list. To enter the command, select the appropriate threads and try again.

No such thread: One of the selected threads doesn't exist or has already terminated.

Expression expected: You didn't select an expression in the source code when entering the Print command via the popup-menu.

Syntax error: The given expression does not accord to the syntax given in the appendix. The position of the error is indicated in the error message.

Unknown module: The source file you have specified for the breakpoint is unknown to the debugger. Check for errors in typing and be sure that the requested file has been compiled with `-g`. Also be aware that some preprocessors (e.g. `yacc`) change the source file name. Unfortunately, the same is true if you have FORTRAN 77 files that are processed by the C-preprocessor (i.e. source files with suffix `.F`). The only name known to DETOP is the one passed to the actual compiler. You may determine this name from the output of the Where command.

Unknown procedure/function: There are two possible causes for this error message: If you have used the popup-menu, DETOP can't find the procedure containing the selected text. Be sure you have selected an expression in the body of a procedure. When using the command line, the procedure name given in the context specification is not known to the debugger or there is more than one procedure with that name. In the latter case, you must also specify the name of the source file containing the desired procedure.

Unknown variable:
Unknown struct/union member:
Unknown type name: The indicated name is unknown to the debugger. Check for errors in typing and correct context specification.

Illegal type: The type of an operand is not allowed for the given operator. This error message is also created if you try to print a value of type `void` or an array of unknown size. Type casting may help you in some cases.

Can't display functions: To display the address of a function, use the `&` operator.

Variable is neither pointer nor array: You tried to apply the `*` or `->` operators or an index to a variable that is neither a pointer nor an array. Type casting may help you in this case.

Array bounds are unknown: DETOP can't evaluate the given expression, since the bounds of some FORTRAN 77 assumed size array is unknown. This regularly happens when in the declaration of an array the bound is specified by an expression rather than a simple variable.

L-value required: An L-value is required in order to use the `&` operator in C.

Can't take address: You have requested the address of either a variable that is stored in a register or a component of a bit field.

Modification of variables not allowed: The given expression has a side effect that would modify a variable. This is not allowed for the Print command.

No stack: DETOP can't find any procedure activation matching your context specification. If you didn't specify a context on the command line, this error is also generated when there is no active procedure that has been compiled with `-g`. This may be the case when the initial thread of an execution context has not yet entered the `main` function. In this case, set a breakpoint on `main` (`MAIN_` for FORTRAN 77 programs), continue the thread and try again.

NOTES:

The Print command handles arrays slightly different as your C compiler does. If the result of an expression is of array-type, then the whole array will be printed, whereas in a C-program the result would be the array's address. To obtain the address of an array or of a function, use the `&` operator.

BUGS:

For a function `foo()`, the command `print &foo` will print the function's starting address. Due to the PowerPC compiler's implementation of function pointers, this value is different from the value stored in a pointer to that function.

Since the compiler doesn't generate sufficient debugging information for C functions using `alloca()`, local variables in such functions cannot be correctly displayed.

3.8 Typeof: Show Type of Variables

DESCRIPTION:

The Typeof command allows to retrieve the type of any source code variable in your program, even if the variable does not exist at the current point of execution. You can also determine the type of expressions. The expression is evaluated for each selected thread and its type is printed in the output window. At a first glance, it seems to be unnecessary to determine the type for each thread, since usually types will be the same. However, threads may have different executables, types may be defined differently in different source files and last but not least assumed size arrays may have different sizes in each thread.

Expressions must be given in C or FORTRAN 77 syntax, dependent on the language your program is written in. For mixed language programs, the language to be used is defined by the currently loaded source file, if you use the popup-menu, or by the context specification (see below), if you use the command line to invoke the Typeof command. In the current implementation, the expression syntax is slightly restricted compared to the language specification. The appendix contains a complete syntax definition.

MENU INPUT:

1. Select a set of threads in the thread list.
2. Move the mouse cursor to the listing window.
3. Select a variable or expression in the body of a procedure.

4. Press the right mouse button to pop up the menu.
5. Select the Typeof item.

DETOP will then try to determine the procedure containing the selected expression from the expression's position in the loaded source file. DETOP will then determine the type of the given expression in the context of that procedure. Note that the procedure need not be active, since types are usually static. The only exception are FORTRAN 77 assumed sized arrays. If you try to determine the type of such an array, and the procedure is active, the current size of the array will be reported. Otherwise DETOP will indicate that the size is unknown.

All identifiers will be searched for according to the usual scoping rule: Identifiers defined locally in the procedure are considered first, then global static identifiers and last global ones. This search is case sensitive for C identifiers, but case insensitive for FORTRAN 77 identifiers.

As an example, consider the following source code listing:

```
1: double x;
2: int sqr(int x)
3: {
4:   return(x * x);
5: }
6: main()
7: {
8:   x = sqr(10);
9: }
```

If the string `x` is selected in line 4, DETOP will print the type of `x` in procedure `sqr`, i.e. `int`. If `x` is selected in line 8, DETOP will print the type of the global variable `x`, since that is the `x` within scope of procedure `main`. The result therefore is `double`.

COMMAND LINE INPUT:

`typeof` [*thread_list*] [*context*] *expr*

The optional *thread_list* overrides the thread selection in the main window's thread list. It consists of a comma-separated list of thread identifiers enclosed in square brackets:

thread_list ::= `[` *thread_id* { `,` *thread_id* }* `]`

The language used for *expr*, the scope used to search variable names and the stack frame used to determine the real size of FORTRAN 77 assumed size arrays are all specified by a so called 'context'. The syntax of this context specification is given below. Note that there are no blanks allowed in this specification.

context ::= { ` `
| `*file_name*`
| `*proc_name*`
| `*file_name*:*proc_name*`
| `*proc_name*(*incarnation*)`
| `*file_name*:*proc_name*(*incarnation*)` }

These context specifications have the following meaning:

` `: This string defines a global context, i.e. each variable name given in the expression is regarded as a global variable. Using this context you can inspect global variables that otherwise would be shadowed by local variables with identical name. In the example given above, `typeof `` x` will print `double`.

Since FORTRAN 77 has no global variables (members of COMMON blocks are local variables!), the language used for `expr` is always C.

`*file_name*`: Supplying a file name results in a global static context, i.e. each identifier in the expression is first searched for among the global static variables defined in the given source file. If it is not found there, the global variables are searched. In this way, you can explicitly request the type of a global static variable in a given source file.

The language used to parse `expr` is defined by the programming language used in the given file.

`*proc_name*`:
`*file_name*:*proc_name*`: Specifying a procedure name will result in the evaluation of the given expression in the context of that procedure. To determine the real size of FORTRAN 77 assumed size arrays, the topmost activation of that procedure in the current procedure call stack is used. All identifiers will be searched for according to the usual scoping rule: Identifiers defined locally in the procedure are considered first, then global static identifiers and last global ones.

The file name and the colon may be omitted if the procedure name is unique within your program. The language used to parse `expr` is defined by the programming language used in the source file containing the given procedure.

In the example given above, `typeof 'sqr' x` will retrieve the type of `x` in function `sqr`, i.e. `int`. `typeof 'main' x` will print the type of the global variable `x`, since that is the `x` within scope of procedure `main`. Therefore the result will be `double`.

'*proc_name*(*incarnation*)':

'*file_name*:*proc_name*(*incarnation*)': You may also add an incarnation number to the context specification. However, there is usually no need to do so, since the activation of a procedure is only relevant for FORTRAN 77 assumed size arrays, and FORTRAN 77 subroutines usually are not recursive. The incarnation number is 0 for the most recent incarnation and is incremented for each occurrence of the given procedure scanning the procedure call stack from top to bottom.

If no context is specified on the command line, DETOP will use the current context of each thread. The current context is the local context defined by the topmost activation of a procedure compiled with `-g`. In the example, `typeof x` is equivalent to `typeof 'sqr' x`.

OUTPUT:

The type definition of the expression's type is given in C or FORTRAN 77 like syntax. For components of composite C types that have again a composite type, DETOP will only print the full type description, if the type has no name. For example, consider the declaration `struct s x[10];`. When requesting the type of `x`, the response will be `struct s [10];`. To see the definition of `struct s`, simply request the type of a component, e.g. `x[0]`.

DIAGNOSTICS:

No thread selected: You haven't selected a thread in the thread list. However, DETOP needs the thread identifier in order to determine the proper symbol table. To enter the command, select the appropriate threads and try again.

Expression expected: You didn't select an expression in the source code when entering the Typeof command via the popup-menu.

Syntax error: The given expression does not accord to the syntax given in the appendix. The position of the error is indicated in the error message.

Unknown module: The source file you have specified for the breakpoint is unknown to the debugger. Check for errors in typing and be sure that the requested file has been compiled with -g. Also be aware that some preprocessors (e.g. yacc) change the source file name. Unfortunately, the same is true if you have FORTRAN 77 files that are processed by the C-preprocessor (i.e. source files with suffix `.F`). The only name known to DETOP is the one passed to the actual compiler. You may determine this name from the output of the Where command.

Unknown procedure/function: There are two possible causes for this error message: If you have used the popup-menu, DETOP can't find the procedure containing the selected text. Be sure you have selected an expression in the body of a procedure. When using the command line, the procedure name given in the context specification is not known to the debugger or there is more than one procedure with that name. In the latter case, you must also specify the name of the source file containing the desired procedure.

Unknown variable:
Unknown struct/union member:
Unknown type name: The indicated name is unknown to the debugger. Check for errors in typing and correct context specification.

Illegal type: The type of an operand is not allowed for the given operator. Type casting may help you in some cases.

Variable is neither pointer nor array: You tried to apply the `*` or `->` operators or an index to a variable that is neither a pointer nor an array. Type casting may help you in this case.

Array bounds are unknown: DETOP can't evaluate the given expression, since the bounds of some FORTRAN 77 assumed size array is unknown. This regularly happens when in the declaration of an array the bound is specified by an expression rather than a simple variable.

L-value required: An L-value is required in order to use the `&` operator in C.

Can't take address: You have requested the address of either a variable that is stored in a register or a component of a bit field.

No stack: This error is generated if you didn't specify a context on the command line and there is no active procedure that has been compiled with `-g`. This may be the case when the initial thread of an execution context has not yet entered the `main` function. In this case, set a breakpoint on `main` (`MAIN_` for FORTRAN 77 programs), continue the thread and try again.

3.9 Set: Modify Value of Variables

DESCRIPTION:

Using the Set command you can assign new values to variables existing at the program's current point of execution. In addition, you may also use a C expression with side effects to modify variables. The assignment expression is evaluated for each selected thread and will change the thread's variables accordingly. Currently, the `set` command only allows to assign values of simple data types, i.e. structure assignment or modification of whole arrays are not possible.

Assignment expressions must be given in C or FORTRAN 77 syntax, dependent on the language your program is written in. For mixed language programs, the language to be used is defined by the currently loaded source file, if you use the popup-menu, or by the context specification (see below), if you use the command line to invoke the Set command. In the current implementation, the expression syntax is slightly restricted compared to the language specification. The appendix contains a complete syntax definition.

Note that the expression supplied to the Set command need not be an assignment statement, but may be an arbitrary expression that may modify variables via side effects. So `i++` or `a[i--] = (j+=3)` are perfectly valid expressions. Further note that nothing will be done by the Set command, if the given expression has no side effects.

MENU INPUT:

1. Select a set of threads in the thread list.

2. Move the mouse cursor to the listing window.
3. Select a variable or left hand side expression in the body of a procedure.
4. Press the right mouse button to pop up the menu.
5. Select the Set item.
6. A dialog box will pop up, prompting you for the new value. Enter a value and press the Ok button.

DETOP will then try to determine the procedure containing the selected expression from the expression's position in the loaded source file. If this succeeds, for each selected thread DETOP will search the current procedure call stack for the topmost invocation of this procedure. The assignment will then be performed in the context of that procedure invocation.

All identifiers will be searched for according to the usual scoping rule: Identifiers defined locally in the procedure are considered first, then global static identifiers and last global ones. This search is case sensitive for C identifiers, but case insensitive for FORTRAN 77 identifiers.

As an example, consider the following source code listing and procedure call stack of a thread:

```
int i = 99;
int fac(int i)
{
  if (i == 1)
    return(1);
  return(i * fac(i-1));
}
```

```
 8: main()
 9: {
10:   i = fac(4);
11: }


Calls:
  fac at example.c:5
  fac(3) at example.c:6
  main at example.c:10
  ParixMain (no line info)
```

If the string `i` is selected in line 4 or 6, DETOP will modify the value of `i` in the most recent (i.e. the fourth) incarnation of the recursive procedure `fac`. Note that using the popup-menu you can't modify the value of `i` in the other incarnations of `fac`. Use the command line instead (see below).

If `i` is selected in line 10, DETOP will set the value of the global variable `i`, since that is the `i` within scope of procedure `main`.

COMMAND LINE INPUT:

`set` [*thread_list*] [*context*] *expr*

The optional *thread_list* overrides the thread selection in the main window's thread list. It consists of a comma-separated list of thread identifiers enclosed in square brackets:

thread_list ::= `[` *thread_id* { `,` *thread_id* }* `]`

The language used for *expr*, the scope used to search variable names and the stack frame used to store the values of local variables are all specified by a so called 'context'. The syntax of this context specification is given below. Note that there are no blanks allowed in this specification.

context ::= { ‘ ‘
| ‘*file_name*‘
| ‘*proc_name*‘
| ‘*file_name*:*proc_name*‘
| ‘*proc_name*(*incarnation*)‘
| ‘*file_name*:*proc_name*(*incarnation*)‘ }

These context specifications have the following meaning:

‘ ‘: This string defines a global context, i.e. each variable name given in the expression is regarded as a global variable. Using this context you can modify global variables that otherwise would be shadowed by local variables with identical name. In the example given above, `set ‘‘ i++` will therefore assign the value 100 to global variable `i`.

Since FORTRAN 77 has no global variables (members of COMMON blocks are local variables!), the language used for `expr` is always C.

‘*file_name*‘: Supplying a file name results in a global static context, i.e. each identifier in the expression is first searched for among the global static variables defined in the given source file. If it is not found there, the global variables are searched. In this way, you can explicitly modify the value of a global static variable in a given source file.

The language used to parse `expr` is defined by the programming language used in the given file.

`*proc_name*`:

`*file_name*:*proc_name*`: Specifying a procedure name will result in the execution of the given assignment expression in the context of the topmost activation of that procedure in the current procedure call stack. All identifiers will be searched for according to the usual scoping rule: Identifiers defined locally in the procedure are considered first, then global static identifiers and last global ones.

The file name and the colon may be omitted if the procedure name is unique within your program. The language used to parse `expr` is defined by the programming language used in the source file containing the given procedure.

In the example given above, `set 'fac' i=5` will change the value of `i` in the most recent (i.e. the fourth) incarnation of the recursive procedure `fac`. `set 'main' i=0` will set the value of the global variable `i` to 0, since that is the `i` within scope of procedure `main`.

`*proc_name*(*incarnation*)`:

`*file_name*:*proc_name* (*incarnation*)`: You may also add an incarnation number to the context specification. In this way you can modify the values in other than the most recent incarnations of recursive procedures. The incarnation number is 0 for the most recent incarnation and is incremented for each occurrence of the given procedure scanning the procedure call stack from top to bottom.

So `set 'fac(0)' i=1` does always mean the same as `set 'fac' i=1`. In the example above, the command `set 'fac(3)' i=5` will change the value in the incarnation of `fac` directly called by `main`.

If no context is specified on the command line, DETOP will use the current context of each thread. The current context is the local context defined by the topmost activation of a procedure compiled with `-g`. In the example, `set i=0` is equivalent to `set 'fac' i=0`.

OUTPUT:

None.

DIAGNOSTICS:

No thread selected: You haven't selected a thread in the thread list. To enter the command, select the appropriate threads and try again.

No such thread: One of the selected threads doesn't exist or has already terminated.

Syntax error: The given expression does not accord to the syntax given in the appendix. The position of the error is indicated in the error message.

Unknown module: The source file you have specified for the breakpoint is unknown to the debugger. Check for errors in typing and be sure that the requested file has been compiled with `-g`. Also be aware that some preprocessors (e.g. `yacc`) change the source file name. Unfortunately, the same is true if you have FORTRAN 77 files that are processed by the C-preprocessor (i.e. source files with suffix

.F). The only name known to DETOP is the one passed to the actual compiler. You may determine this name from the output of the Where command.

Unknown procedure/function: There are two possible causes for this error message: If you have used the popup-menu, DETOP can't find the procedure containing the selected text. Be sure you have selected an expression in the body of a procedure. When using the command line, the procedure name given in the context specification is not known to the debugger or there is more than one procedure with that name. In the latter case, you must also specify the name of the source file containing the desired procedure.

Unknown variable:
Unknown struct/union member:
Unknown type name: The indicated name is unknown to the debugger. Check for errors in typing and correct context specification.

Illegal type: The type of an operand is not allowed for the given operator. This error message is also created if you try to assign to an L-value of type `void`. Type casting may help you in some cases.

Variable is neither pointer nor array: You tried to apply the `*` or `->` operators or an index to a variable that is neither a pointer nor an array. Type casting may help you in this case.

Array bounds are unknown: DETOP can't evaluate the given expression, since the bounds of some FORTRAN 77 assumed size array is unknown. This regularly happens when in the declaration of an array the bound is specified by an expression rather than a simple variable.

L-value required: An L-value is required as the left hand side of an assignment and in order to use the `&`, `++` and `--` operators in C.

Can't take address: You have requested the address of either a variable that is stored in a register or a component of a bit field.

No stack: DETOP can't find any procedure activation matching your context specification. If you didn't specify a context on the command line, this error is also generated when there is no active procedure that has been compiled with `-g`. This may be the case when the initial thread of an execution context has not yet entered the `main` function. In this case, set a breakpoint on `main` (`MAIN_` for FORTRAN 77 programs), continue the thread and try again.

BUGS:

Since the compiler doesn't generate sufficient debugging information for C functions using `alloca()`, local variables in such functions cannot be correctly modified.

3.10 Where: Determine the State of Threads

DESCRIPTION:

For each thread applied to, the Where command retrieves the current point of execution. In addition, it shows whether or not the thread is currently stopped by the debugger.

MENU INPUT:

1. Select the threads to be inspected in the thread list.
2. Move the mouse cursor to the listing window.
3. Press the right mouse button to pop up the menu.
4. Select the Where item.

COMMAND LINE INPUT:

`where` [*thread_list*]

The optional *thread_list* overrides the thread selection in the main window's thread list. It consists of a comma-separated list of thread identifiers enclosed in square brackets:

thread_list ::= `[` *thread_id* { `,` *thread_id* }* `]`

OUTPUT:

For each selected thread, you will get the following output:

1. The thread's state and procedure call stack is printed in the output window.
2. The thread's current position is indicated in the source code listing.

The state shown in the output window may either be `STOPPED`, indicating that the thread is stopped by the debugger or due to an exception, or `RUNNING` otherwise. Note that the `RUNNING` state is also shown for threads that are currently descheduled either by normal scheduling or due to some waiting condition. Whether or not a thread is waiting for communication can be determined from its call stack. For a waiting thread the topmost procedure is usually `_NK_ChanSync()`.

The procedure call stack shows a thread's active procedures and, for procedures compiled with `-g`, also the point of execution in this procedure. The stack grows bottom-up, i.e. a callee is printed above its caller. To shorten the call stack for recursive procedures, a repeat counter is appended to a procedure name, if there are identical successive entries. As an example, consider the following output:

```
Calls:
  malloc (no line info)
  EnterElem at tree.c:125
  EnterElem(4) at tree.c:108
  main at main.c:54
  ParixMain (no line info)
```

This is read as follows: Procedure `ParixMain()` called `main()` that in turn called a recursive procedure `EnterElem()` at line 54 in file `main.c`. `EnterElem()` has made four recursive calls at line 108 in `tree.c`; the last invocation called the `malloc()` function at line 125. The thread is currently stopped somewhere in `malloc()`.

The thread's position shown in the source code listing depends on the source file that is displayed. The position marker will always reflect the topmost element in the call stack matching this file. So in the example above, if `tree.c` is loaded, line 125 will be marked; if `main.c` is displayed, line 54 will be marked.

DIAGNOSTICS:

No thread selected: You haven't selected a thread in the thread list. To enter the command, select the appropriate threads and try again.

No such thread: One of the selected threads doesn't exist or has already terminated.

Appendix A

Syntax Definition For Expressions

The following grammars define the syntax accepted by the `print`, `typeof` and `set` commands for both C and FORTRAN 77. Operator precedence is the same as in the language definition, therefore it is not indicated in the grammar.

Syntax for C expressions

$$
\begin{array}{rcl}
\textit{expr} & ::= & \{\ \textit{expr}\ \ \textit{binary\_operator}\ \ \textit{expr} \\
 & & \mid \textit{term}\ \} \\
\textit{term} & ::= & \{\ \textit{term}\ \ \textit{postfix\_operator} \\
 & & \mid \textit{unary\_operator}\ \ \textit{term} \\
 & & \mid \texttt{sizeof}\ \texttt{(}\ \textit{type\_definition}\ \texttt{)} \\
 & & \mid \textit{term}\ \texttt{[}\ \textit{expr}\ \texttt{]} \\
 & & \mid \textit{term}\ \texttt{.}\ \ \textit{component\_name} \\
 & & \mid \textit{term}\ \texttt{->}\ \textit{component\_name} \\
 & & \mid \textit{variable\_name} \\
 & & \mid \texttt{(}\ \textit{expr}\ \texttt{)} \\
 & & \mid \textit{number} \\
 & & \mid \textit{char\_constant}\ \} \\
\textit{binary\_operator} & ::= & \{\ \texttt{=} \mid \texttt{+=} \mid \texttt{-=} \mid \texttt{*=} \mid \texttt{/=} \mid \texttt{\%=} \\
 & & \mid \texttt{<<=} \mid \texttt{>>=} \mid \texttt{\&=} \mid \texttt{|=} \mid \texttt{\^{}=} \\
 & & \mid \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%} \mid \texttt{<<} \mid \texttt{>>} \mid \texttt{\&} \mid \texttt{|} \mid \texttt{\^{}} \\
 & & \mid \texttt{<} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{!=} \mid \texttt{==}\ \} \\
\textit{unary\_operator} & ::= & \{\ \texttt{-} \mid \texttt{*} \mid \texttt{\&} \mid \texttt{!} \mid \texttt{\~{}} \\
 & & \mid \texttt{sizeof} \mid \textit{type\_cast}\ \} \\
\textit{postfix\_operator} & ::= & \{\ \texttt{--} \mid \texttt{++}\ \} \\
\textit{type\_cast} & ::= & \texttt{(}\ \textit{type\_definition}\ \texttt{)} \\
\textit{type\_definition} & ::= & \textit{type\_name}\ \{\ \texttt{*}\ \}^{*} \\
\textit{char\_constant} & ::= & \text{'}\ \textit{character}\ \text{'}
\end{array}
$$

Syntax for FORTRAN 77 expressions

$$
\begin{array}{rcl}
\mathit{expr} & ::= & [\ \mathit{var}\ \texttt{=}\ ]\ \mathit{exp} \\
\mathit{exp} & ::= & \{\ \mathit{uexp} \\
 & & \mid\ \texttt{(}\ \mathit{exp}\ \texttt{)} \\
 & & \mid\ \mathit{complex\_constant}\ \} \\
\mathit{uexp} & ::= & \{\ \mathit{var} \\
 & & \mid\ \mathit{number} \\
 & & \mid\ \mathit{unary\_operator}\ \ \mathit{exp} \\
 & & \mid\ \mathit{exp}\ \ \mathit{binary\_operator}\ \ \mathit{exp}\ \} \\
\mathit{binary\_operator} & ::= & \{\ \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{**} \\
 & & \mid\ \texttt{<} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{/=} \mid \texttt{==} \\
 & & \mid\ \texttt{.lt.} \mid \texttt{.gt.} \mid \texttt{.le.} \mid \texttt{.ge.} \\
 & & \mid\ \texttt{.ne.} \mid \texttt{.eq.} \mid \texttt{.neqv.} \mid \texttt{.eqv.} \\
 & & \mid\ \texttt{.and.} \mid \texttt{.or.}\ \} \\
\mathit{unary\_operator} & ::= & \{\ \texttt{+} \mid \texttt{-} \mid \texttt{.not.}\ \} \\
\mathit{var} & ::= & \mathit{varaible\_name}\ [\ \texttt{(}\ \mathit{exp}\ \{\ \texttt{,}\ \mathit{exp}\ \}^{*}\ \texttt{)}\ ] \\
\mathit{complex\_constant} & ::= & \texttt{(}\ \mathit{uexp}\ \texttt{,}\ \mathit{uexp}\ \texttt{)}
\end{array}
$$

Appendix B

X Resources

This appendix contains a list of some of DETOP's X resources you might wish to change according to your own preferences. The standard values of these resources, as used by DETOP are also indicated. If DETOP uses the default resource value defined by Motif, no value is shown in the list below.

```
! Colors (value can be color name or #rgb,
! where r, g, b is a 2 digit hexadecimal
! number defining the intensity)

Detop*foreground:
Detop*background:
Detop*topShadowColor:
Detop*bottomShadowColor:

! Font (value is a font name as obtained by
! 'xlsfonts', e.g. 6x13). For well formatted
! output, use only fixed fonts.
```

```
Detop*fontList:

! Size of the main window (in pixel)

Detop*mainwin.width:      800
Detop*mainwin.height:     475

! Width of thread list (in pixel)

Detop*threadlist.width:   100

! Width of state display and output window
! (in pixel)

Detop*datatxt.width:      190

! Initial height of state display (in pixel)

Detop*datatxt.height:     100

! Size of thread selection window (in pixel)

Detop*pselw.width:        400
Detop*pselw.height:       250

! Size of help window (in pixel)

Detop*HelpWindow.width:   550
Detop*HelpWindow.height:  400
```