INQUEST
Debugger Tutorial

SGS-THOMSON
MICROELECTRONICS

72–TDS–465–01 – October 1995

This product incorporates innovative techniques which were developed with support from the European Commission under the ESPRIT Projects:

- P2701 PUMA (Parallel Universal Message-passing Architectures)
- P5404 GPMIMD (General Purpose Multiple Instruction Multiple Data Machines).
- P7250 TMP (Transputer Macrocell Project).
- P7267 OMI/STANDARDS.
- P6290 HAMLET (High Performance Computing for Industrial Applications)

Document Number: 72 TDS 465 01

Contents

Preface

The INQUEST Development Environment is a collection of powerful software development tools designed to help you build fast, bug-free code, including a debugger and execution monitors. This document is the *INQUEST Debugger Tutorial*, which takes you step-by-step through the main features of the debugger, both for C users and for occam users.

Reference material on the debugger and other INQUEST tools may be found in the *INQUEST User and Reference Manual*.

1 Introduction

The INQUEST Development Environment is a collection of powerful software development tools designed to help you build fast, bug-free code. This document contains some tutorials for one of those tools, the debugger.

Chapter 2 provides a brief introduction to the main features of the debugger. It is a fully-featured debugger that provides all the functions you need to debug sequential and parallel programs.

Chapter 3 describes, step by step, a full interactive debugging session of a simple C program.

Chapter 4 describes a post-mortem debugging session of the same program.

Chapters 5 and 6 describe similar interactive and post-mortem debugging sessions of an occam program.

To get you started quickly we have kept these tutorials brief. Inevitably this means that some features are not used.

The tutorials assume you are familiar with programming transputer systems and have some experience of using debuggers.

2 Overview of the debugger

If you have experience of using conventional debuggers then many of INQUEST's debugging features will be familiar to you. As you would expect, the debugger enables you to:

- set breakpoints and watchpoints;
- single step through source code;
- interrupt running code;
- examine variables;
- examine memory;
- examine stack traces;
- debug post-mortem.

The INQUEST debugger can debug either interactively or post-mortem.

- *Interactive* debugging means debugging as the program executes. Part or all of the program can be started, interrupted, or left to run until it hits a breakpoint or watchpoint and then restarted.
- *Post-mortem* debugging means debugging after the program has crashed or been halted. It may have crashed or halted during normal running or during an interactive debugging session. The program cannot resume.

Interactive debugging is the normal method of debugging, because it allows you to watch the behavior of the program as it executes. If the program behaves wrongly but does not crash then post-mortem debugging cannot be used without stopping the program.

Post-mortem debugging is used to find the reason for an unexpected crash. It may also be used if there is not enough memory to debug interactively or the problem only shows when the program is run without the debugger. If necessary, the program can be made to crash by inserting the assembly instruction *seterr* on an IMS T2xx/T4xx/T8xx transputer or *causeerror* on an IMS T9000 transputer or by calling the occam procedure `CAUSEERROR()` or using the functions `debug_assert` or `debug_stop`. Post-mortem debugging can either use the target hardware or be run on the host using a dump file.

In addition to these facilities the debugger has features that are specific to the needs of debugging multi-tasking and multi-processor code. These include the ability to:

- debug at the program, process, thread or frame level;
- have several windows open at once to view more than one piece of code;

- jump down a channel or link from one task to another.

Perhaps the most important difference between the INQUEST debugger and sequential program debuggers is INQUEST's multi-level debugging features. Multi-tasking programs are naturally hierarchical, being composed of processes containing threads of execution which contain function and procedure calls. The user benefits from the ability to debug in a hierarchical way. The debugger makes it easy for you to move up and down the levels of your program, debugging objects such as processes and threads of execution.

In the debugger, we need to distinguish between two different types of task, so the terminology here is slightly different from that used in the *ANSI C Toolset* and the occam *2.1 Toolset* documentation and elsewhere. In this document, when referring to C programs, we use the term *process* only for configuration-level processes defined in the configuration source code, and the term *thread* is used for program-level tasks defined in the C source code. When referring to occam programs, we use the term *process* for the entire code running on one processor and we use the term *thread* to mean any other sub-process. The functional difference between a process and a thread is that a process is static, defined at build time, while a thread is created dynamically while the program is running.

The debugger has a browser window that enables you to move up and down the hierarchy and select individual processes and threads for debugging. At any one time you can be working on the whole program, a process, a thread or a function or procedure call within a thread.

The debugger behaves appropriately at different levels. For example, clicking on the **Interrupt** button at the top (program) level causes all threads of all processes to be interrupted. At the process level it just causes the threads of the selected process to be interrupted. At the thread level it only interrupts the selected thread, leaving the other threads running.

The following sections summarize the main functions available at each level.

2.1 The program level

This is a top level view of your program. When you start the debugger it begins at the program level. It allows you to see the configuration code and a list of the program's processes. If you are debugging at this level you can:

- move down to the process level by selecting a process;
- examine processor memory;
- find the most recently stopped thread in the program.

When debugging interactively, you can also:

- start the program running;

- interrupt the execution of all running threads;
- continue the execution of all stopped threads.

2.2 The process level

At process level the debugger displays a list of threads belonging to the selected process. The current execution state of each thread (e.g. running or stepping) is displayed beside the name. The displayed source code will generally show whatever you were looking at last time you were looking at this process.

When debugging at this level you can:

- find all threads of the process;
- examine the values of static variables;
- jump down a channel. This changes the context to the thread of execution that is waiting for the channel, which may be the sender or the receiver;
- move up to the program level by clicking on the **Processes** button;
- move down to the thread level by selecting a thread;
- examine processor memory.

When debugging interactively, you can also:

- start all threads of the process running;
- interrupt all the running threads of the process;
- set breakpoints on selected source code lines. The breakpoints affect all of the threads that share that piece of code. Only the threads that hit the breakpoint will be stopped;
- set watchpoints on automatic and static variables so that all threads that write to them stop executing;
- delete breakpoints and watchpoints;
- continue execution of all the stopped threads of the process.

2.3 The thread level

At thread level the debugger displays a list of the threads of the selected process, with the selected thread highlighted. The source code of the selected thread is displayed in the code window.

2.3 The thread level

When debugging at this level you can:

- examine the values of variables;
- jump down a channel. This changes the context to the thread of execution that is waiting on the channel;
- move up to the process level by clicking on the **Deselect** button;
- move up to the program level by clicking on the **Processes** button;
- examine processor memory;
- find all threads;
- view the call stack of the thread.

When debugging interactively, you can also:

- start the selected thread;
- interrupt a thread (if it is running);
- set breakpoints on selected lines. The breakpoints affect only the selected thread. When this selected thread hits the breakpoint it stops. Other threads can execute the same line without stopping;

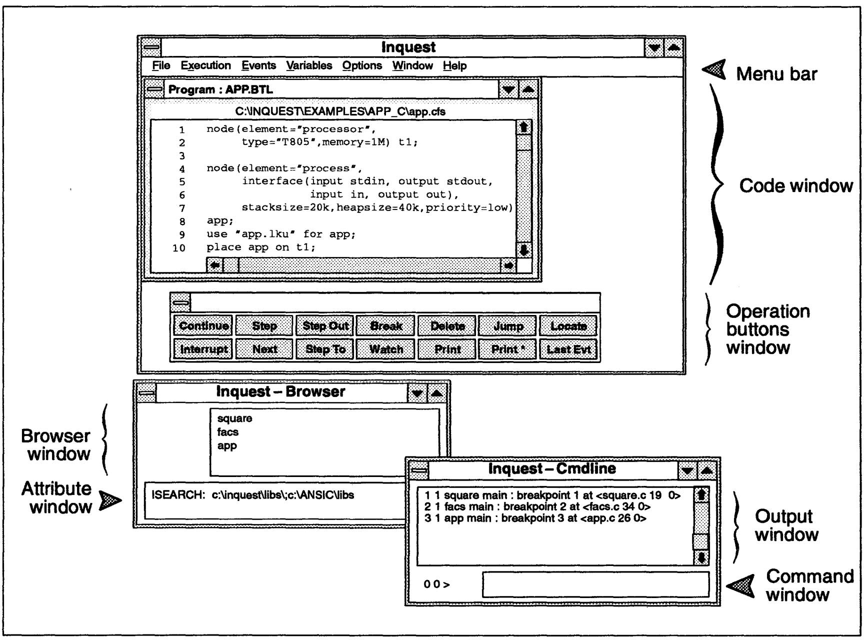

Figure 2.1 The Microsoft Windows debugger display

- set watchpoints on variables. If the thread accesses a variable which has a watchpoint set then it stops. The other threads (if any) will continue running;
- delete breakpoints and watchpoints;
- step through the source code a statement at a time;
- step forward to a specified point in the code.

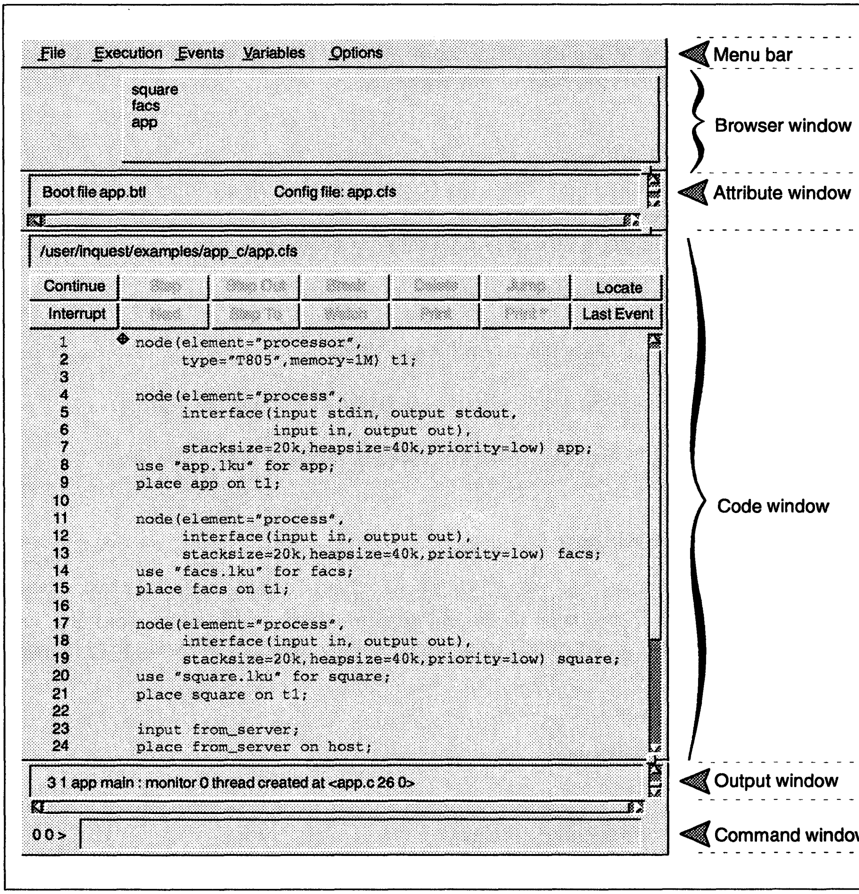

Figure 2.2 X-Windows debugger display

2.4 The debugger display

When you start the debugger, it displays a set of windows that show the configuration code and a list of the processes in the program being debugged. For the example program, on a Microsoft Windows system the thread-level display looks like Figure 2.1. The X-Windows display looks like Figure 2.2. This example is used in the demonstration sessions that follow.

2.4.1 The menu bar

All the debugger functions, called operations, can be accessed through the menus pulled down from the menu bar. The most commonly used functions are also available as buttons, which are at the top of the code window on X-Windows displays or in a separate window on Microsoft Windows displays.

2.4.2 The browser window

The browser is to allow you to select the particular 'object' you want to debug. An 'object' in this sense is a process, a thread or a function or procedure call.

The browser window has three functions:

1 To tell you which object is currently selected.

2 To display brief information about selected and associated objects.

3 To enable you to select a different object or move to a different level.

2.4.3 The attribute window

The attribute window gives you detailed information about the object selected in the browser window. The information displayed depends on the current browser level. This information generally takes more than one line, so the scroll bar or sizing may be used to show the other lines.

2.4.4 The file sub-window

The file sub-window is at the top of the code window. It identifies the pathname of the file that is displayed in the code window.

2.4.5 The code window

When you select an object in the browser window, its source code or disassembled code is automatically displayed in the code window. Each line of code is numbered and can have one or more of four special markers in the left margin. The markers are shown in Figure 2.3.

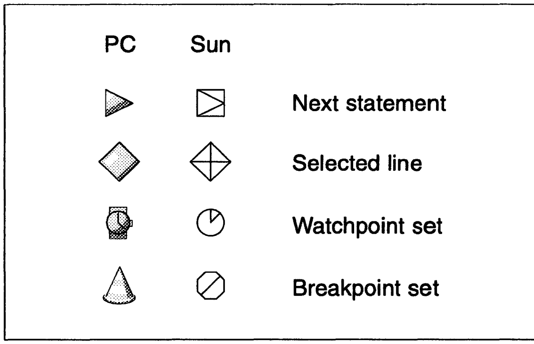

Figure 2.3 Line markers

- The **Next statement** marker shows the line containing the statement or instruction about to be executed by the current thread in its last known state, or, in post-mortem debugging, the line that was being executed when the program stopped.
- The **Selected line** marker indicates the currently selected line. This tells the debugger which line an operation should be applied to. You select lines by clicking on them with the left mouse button. Only one line can be selected at a time.
- The **Watchpoint set** marker appears beside declarations of variables that have been marked as watchpoints.
- The **Breakpoint set** marker appears beside lines that have breakpoints set.

2.4.6 The operation buttons

The operation buttons are at the top of the code window on X-Windows systems and in a separate window on Microsoft Windows systems. They provide quick access to the most frequently used functions. Functions that are not available at a particular level or state appear “greyed out” when that level or state is active.

Continue Continue execution of the current object or objects.

Interrupt Interrupt the execution of the current thread or threads. This is achieved by setting a breakpoint on the next instruction of the thread or threads.

Step Execute a single instruction, statement or part statement.

Next Execute a single instruction, statement or part statement, skipping over function calls.

Step Out Complete execution of the current function or up to the current frame and stop.

Step To	Execute up to the selected line of code.
Break	Set a breakpoint at the selected line for the current object.
Watch	Place a watchpoint on the selected variable for the current object.
Delete	Delete a watchpoint or breakpoint from the selected line.
Print	Print the value of the selected variable in the output window.
Jump	Jump to the thread of execution waiting for communication on the selected channel.
Print *	Print the value of a de-referenced pointer in the output window.
Locate	Display in the code window the code around the next statement marker of the selected thread.
Last Event	Switch context to the thread which stopped most recently.

2.4.7 The output window

The output window displays messages generated by the debugger, including responses to commands, notification of events and error messages.

2.4.8 The command window

The command line window gives access to a powerful C-like command language interpreter that allows you to issue conditional and compound commands.

3 An example ANSI C interactive debugging session

This chapter takes you in detail, step by step, through one example interactive debugging session, to demonstrate the basic features of the INQUEST debugger, using an ANSI C program as the example. A similar occam example is described in chapter 5. Post-mortem debugging is shown in chapter 4.

This chapter shows you how to:

- build the program for debugging;
- start the debugger;
- place a breakpoint;
- start the processes running;
- locate where a breakpoint has occurred;
- examine a variable;
- remove a breakpoint;
- single step through the source code;
- examine a call stack;
- step over function calls;
- interrupt a running process;
- watch communication between two threads;
- set a watchpoint on a variable;
- delete a watchpoint;
- jump down a channel;
- quit from the debugger.

Before starting the session you need to know a little about the example program. This is described in section 3.1 below.

3.1 The example program

The example debugging session uses an example program called `app`, which you will find in the directory `app_c` within the `examples` directory. The directory contains all the source code and makefiles or batch files.

3.1 The example program

This is a simple multi-task program. The tasks are arranged in a pipeline that generates the sum of a series of squares of factorials, as in the following formula:

$$\sum_{i=1}^{n} factorial(i)^2$$

It is not an efficient program, but it provides the structures we need to try out the debugger. The program consists of three processes; `app`, `facs` and `square`. The `app` process generates three threads; `control`, `feed` and `sum`.

Figure 3.1 shows how the processes and threads connect together.

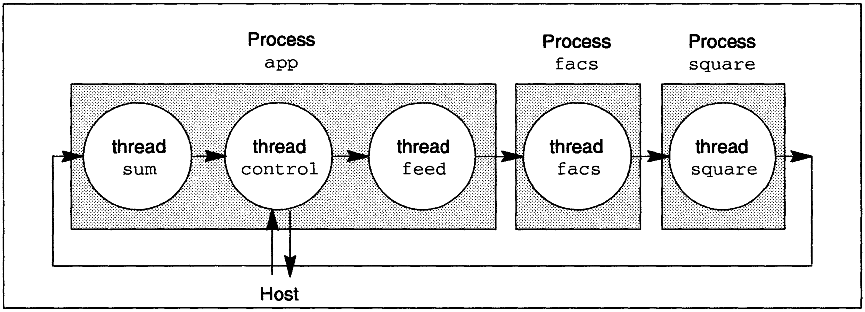

Figure 3.1 The example program

- `control`. This thread prints out the message

  ```
  Sum of the first n (n < 9) squares of factorials
  ```

  and asks you to type in a value for *n*. If *n* is not less than 9, there would be an arithmetic overflow, so the program rejects it. The number you type is sent to `feed`. Zero or a negative number makes the program terminate.

- `feed`. This thread receives the number *n* from `control` and sends all the numbers from 1 to *n* to `facs`, one at a time. It then sends a `next` token to mark the end of this batch of data.

- `facs`. This process generates the factorial of each number it receives from `feed`, and sends the result to `square`. It passes on the `next` batch marker.

- `square`. This process generates the square of each number it receives from `facs` and sends the result to `sum`. It passes on the `next` batch marker.

- `sum`. This thread generates the sum of all the numbers it receives from `square`, until it receives the `next` batch marker. On receiving this, `sum` passes the total on to `control`.

- `control`. This reads the results from `sum` and displays them.

To terminate the program you type zero in response to the '`Please type n :`' prompt issued by `control`. This causes an `end` token to be sent down the pipeline. Each thread terminates after passing on the `end` token.

The channels all use a special protocol which encodes the **next** and **end** tokens as the integers 1 and 2. To distinguish it from these tokens, data is preceded by a **data** tag which is the integer 0. The protocol is handled by the functions **send_data**, **send_next**, **send_end** and **read_chan**. **read_chan** returns the tag it received. These routines are in **comms.c**.

For the purposes of this tutorial, all the processes are configured to run on the same processor. Depending on the version of INQUEST that you have, the configuration should be something like figure 3.2. This refers to the hardware file, which is given in figure 3.3.

```
#include "hardware"

node(element="process",
     interface(input stdin, output stdout,
               input in, output out),
     stacksize=20k,heapsize=40k,priority=low) app;
use "app.lku" for app;
place app on RootNode;

node(element="process",
     interface(input in, output out),
     stacksize=20k,heapsize=40k,priority=low) facs;
use "facs.lku" for facs;
place facs on RootNode;

node(element="process",
     interface(input in, output out),
     stacksize=20k,heapsize=40k,priority=low) square;
use "square.lku" for square;
place square on RootNode;

input from_server;
place from_server on host;
output to_server;
place to_server on host;

connect from_server, app.stdin;
connect to_server, app.stdout;
connect app.out, facs.in;
connect facs.out, square.in;
connect square.out, app.in;
```

Figure 3.2 Configuration file

```
/* This is the T-series hardware description */
T450 (memory = 512K) RootNode;
connect host to RootNode.link[0];
val tx 1;
```

Figure 3.3 Hardware file

Before starting the tutorial you may find it useful to have a listing of the example program source code. It would also be useful to look at figure 2.1 or 2.2 in section 2.4 to show you the names of the parts of the debugger display.

3.2 Building the example program

Any ANSI C program that you want to debug must be built in the following way:

- Compile with the G option to generate debugging symbolic information.
- Link with the startup file `cdebug.lnk` in place of the usual file `cstartup.lnk`, and `cdebugrd.lnk` in place of the file `cstartrd.lnk`. This will include the debugging run-time libraries.
- Configure with the GA option to include the debugging kernel. This option is what causes the program to run with interactive debugging.
- Collect using `icollect`.

On Sun systems, the example program makefile does all this for you and is called `makefile`. On a PC, a batch file called `build.bat` is provided to do this. This generates a bootable file for interactive debugging called `app.btl`.

3.3 Step-by-step tutorial

This section is the step-by-step debugging session to guide you through the main features of interactive debugging with the INQUEST debugger.

Step 1. Move to the `app_c` sub-directory in the `examples` directory which contains the ANSI C example program `app`.

Step 2. Check that the `TRANSPUTER`, `ASERVDB` and `ISEARCH` environment parameters are correctly defined.

Step 3. The root transputer must be an IMS T400, IMS T425, IMS T801, IMS T805, ST20450 (T450) or IMS T9000. The example is configured for a single ST20450 or for a single IMS T9000 depending on the INQUEST version.

If you are using a different type of root transputer then you will need to edit:

- the hardware configuration file, `hardware`;
- for Suns the `make` macro file `tools`;
- for PCs the build batch file `build.bat`.

The changes to make in each case are given in table 3.1.

Target	All users: file `hardware`	Sun users only: file `tools`	PC users only: file `build.bat`
ST20450 / T450	No change	No change	No change
IMS T9000	No change	No change	No change
IMS T400 or T425	Change `T450` to `T425`	Change `450` to `425`	Change `t450` to `t425`
IMS T801 or T805	Change `T450` to `T805`	Change `450` to `805`	Change `t450` to `t805`

Table 3.1 Changes to examples to support different targets

Step 4. Build a bootable file, suitable for debugging. To do this on a Sun, type at an operating system prompt:

```
make
```

On a PC, at a DOS prompt type:

```
build
```

3.3.1 Starting the debugger

Step 5. Start the debugger with the example application, `app.btl`, which is the bootable code for interactive debugging that you created in Step 4.

On PC systems, start up Windows, open the File Manager and double click on the `inquest.exe` program. This will open the **Command line** dialog box. Use the browse button next to the **File** field to find and select the application `app.btl` in the `app_c` sub-directory in the `examples` directory. Click on the `Run` button in the **Command line**.

On Sun systems type:

```
inquest app.btl
```

This loads the debugger onto the host computer and a small kernel of debugger code onto each processor of the target hardware that has processes to be debugged.

The parameters you can give to `inquest` are described fully in the *INQUEST User and Reference Manual.*

Step 6. Wait while a debugging display is created. This will show the configuration file in the code window and a list of the example program's processes in the browser window. This is the program level display. Figure 3.4 shows the PC display; the X-Windows display is shown in figure 3.5.

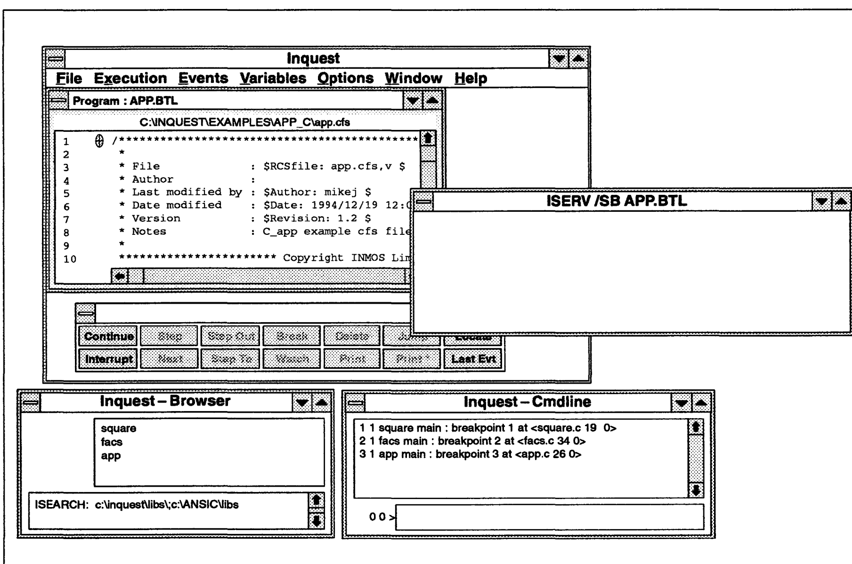

Figure 3.4 The Microsoft Windows start-up display for the example program

On a PC a new input and output window is created for the application, labelled something like `ISERV /SB APP.BTL`. On a Sun, the input and output of the program being debugged will be shown in the window that the debugger was started from. In either case, this will be called the program window in the rest of this document.

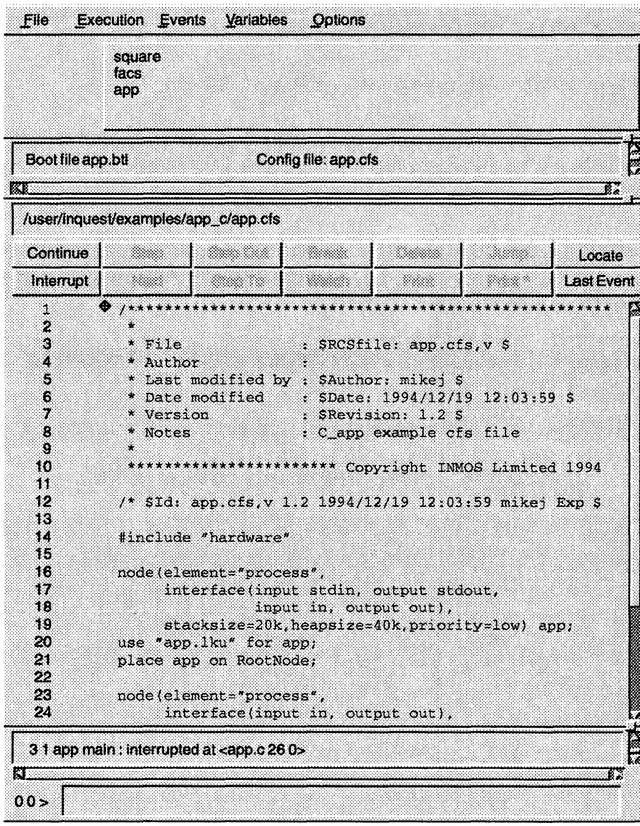

Figure 3.5 The X-Windows initial display for the example program

The factorial program has been halted at the beginning of each process.

3.3.2 Placing a breakpoint

Step 7. You are now going to place a breakpoint in the `facs` process. To do this you must first display the `facs` source code. Move the cursor to the browser window and click on '`facs`'. The browser window changes to process level, displaying a list of the threads belonging to `facs`. There is only one thread, the function `main`. The browser line says:

```
facs main: stopped at <facs.c 34 0>
```

This means the thread `main` in process `facs` is stopped at the zeroth step on line 34 in the `facs.c` source file. The steps on each line are numbered from zero.

Step 8. Click on the thread `main` in the browser window. A source code listing of the `main` thread appears in the code window. This is the thread level display, which should look like figures 3.6 and 3.7.

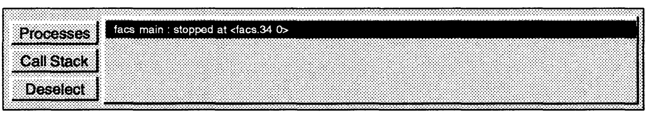

Figure 3.6 Thread level display browser window

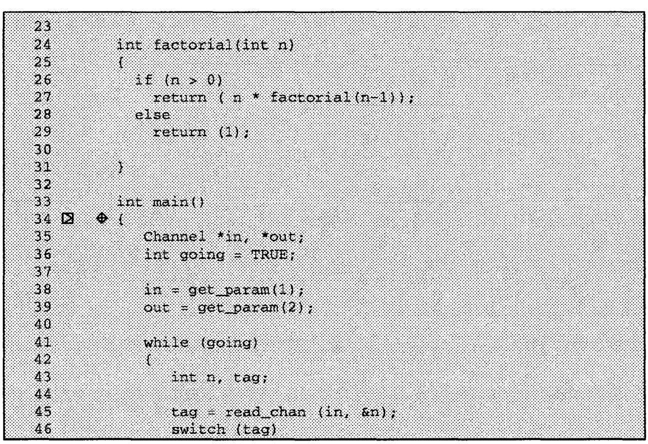

Figure 3.7 Thread level display code window

Step 9. We want to place a breakpoint before line 49, which calculates and sends the factorial. Use the scroll bar at the side of the code window to locate line 49 of the code. This is where you are going to place the breakpoint.

```
31     }
32
33     int main()
34     {
35         Channel *in, *out;
36         int going = TRUE;
37
38         in = get_param(1);
39         out = get_param(2);
40
41         while (going)
42         {
43             int n, tag;
44
45             tag = read_chan (in, &n);
46             switch (tag)
47             {
48                 case DATA: {
49                     send_data (out, factorial(n));
50                     break;
51                 }
52                 case NEXT: {    /* start a new sequence */
53                     send_next (out);
54                     break;
```

Figure 3.8 Source code before line 49 has been selected

Step 10. Select line 49 by clicking somewhere on the '**send_data**' function call. The selection line marker, ⊕ on Suns or ◇ on PCs, appears alongside the statement. Click on the blank space after the statement to ensure that no text is highlighted, or else clicking on **Break** will cause the debugger will try to set a breakpoint on a function of that name.

Step 11. Click on the **Break** button to set a breakpoint at the selected line. A breakpoint marker, ⊘ on Suns or △ on PCs, appears alongside the selected line marker. Notice that the output window displays a message telling you about the breakpoint you have just set:

```
breakpoint 1 at <facs.c 49 0> iptr #8000fbdf of facs ...
```

This means the breakpoint has been labelled event number 1. It has been set at the start of line 49 of the file `facs.c`. The part in chevrons, '`<facs.c 49 0>`', means the zeroth step of line 49 of the file `facs.c`. The breakpoint has been set in the code at address #8000fbdf.

If you scroll the output window up one line, you will find:

```
2 1 > break <facs.c 49 0>
```

This means that a breakpoint has been set in process 2, thread 1. The breakpoint applies only to the currently selected thread, `main`, and will cause execution of the thread to stop when this line is about to be executed. If there were other threads using the same code they would not be affected by this breakpoint.

A breakpoint set at process level would act on all the threads in the currently selected process. It would give a message like:

```
1 0 > break <facs.c 49 0>
```

The '`1 0 >`' means all threads of process 1.

```
}

int main()
{
   Channel *in, *out;
   int going = TRUE;

   in = get_param(1);
   out = get_param(2);

   while (going)
   {
      int n, tag;

      tag = read_chan (in, &n);
      switch (tag)
      {
         case DATA: {
            send_data (out, factorial(n));
            break;
         }
         case NEXT: {   /* start a new sequence */
            send_next (out);
            break;
```

Figure 3.9 Source code after the breakpoint has been set

3.3.3 Starting the example program

Step 12. Having set a breakpoint you are now ready to start the example program. If you were to click on the **Continue** button at this level you would only start thread `main` of the `facs` process running. To ensure that all the processes are set running you must use the browser to return to the program level. To do this, move the cursor to the browser window and click on the **Processes** button to display the list of processes. The browser returns to program level and the browser window changes to look like figure 3.10.

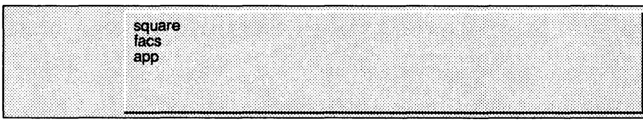

Figure 3.10 Browser window showing the example program's processes

Step 13. At this level, with no processes selected, you can start all the processes by clicking on the **Continue** button. When you do, notice that the output window displays the message:

```
0 0 > continue
```

The '`0 0 >`' means all threads of all processes.

Step 14. As the `app` program runs it displays the following message in the program window:

```
Sum of the first n (n < 9) squares of factorials
Please type n :
```

Move the cursor to the program window and type '`4`' followed by a return.

Step 15. The output window should now display the message:

```
2 1 facs main : breakpoint 1 at <facs.c 49 0>
```

This tells you that breakpoint 1 has been hit in thread `main` of process `facs`. This should come as no surprise as it is where you set the breakpoint in Step 11.

3.3.4 Locating where a breakpoint has occurred

Step 16. To display the code where the breakpoint has occurred, click on the **Last Event** button. The context changes to frame level at the last breakpoint or watchpoint that occurred.

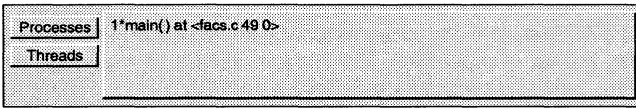

Figure 3.11 Browser window after the breakpoint has been located

```
        in = get_param(1);
        out = get_param(2);

        while (going)
        {
           int n, tag;

           tag = read_chan (in, &n);
           switch (tag)
           {
              case DATA: {
                 send_data (out, factorial(n));
                 break;
              }
              case NEXT: {    /* start a new sequence */
                 send_next (out);
                 break;
              }
              case END: {     /* terminate */
                 going = FALSE;
                 send_end (out);
              }
           }
        }
```

Figure 3.12 Code window after the breakpoint has been located

3.3.5 Removing a breakpoint

Step 17. The breakpoint has served its purpose, so you can now delete it.

With the selection line marker, ⊕ on Suns or ◈ on PCs, on line 49, click on the **Delete** button. The breakpoint marker disappears. The output window should display:

```
2 1 > delete <facs.c 49>;
```

which means that the breakpoint on line 49 of `facs.c` has been deleted.

3.3.6 Examining a variable

Step 18. You are now going to examine the value of a variable.

Place the cursor on the variable '`n`' on line 49, and highlight it by double clicking the left mouse button. This causes the whole word to be selected, which in this case is just the letter `n`.

Step 19. Display the value of the variable by clicking on the **Print** button. The **Print** operation displays the value in the output window; it does not produce a hard copy. The output window should show the message:

```
1
```

This tells you that the `factorial` function is about to compute the factorial of 1.

3.3.7 Single stepping through the source code

Step 20. Since this thread is stopped you can step through the code and follow the execution path. Line 49, the current location, has a call to the **`factorial`** function.

Click on the **Step** button to execute the current statement and move on to the next. The debugger steps into the function **`factorial`** and the current location marker moves to line 25, as shown in figure 3.13.

```
14
15        #define TRUE 1
16        #define FALSE 0
17
18
19                /*
20                 * compute factorial
21                 *
22                 */
23
24        int factorial(int n)
25        {
26          if (n > 0)
27            return ( n * factorial(n-1));
28          else
29            return (1);
30
31        }
32
33        int main()
34        {
35           Channel *in, *out;
36           int going = TRUE;
37
```

Figure 3.13 Single stepping through the **`factorial`** function

Step 21. Click on the **Step** button twice more. The current location marker steps through the **`factorial`** function to the next call of **`factorial`** on line 27.

A single line of source code may have more than one statement on it. The cursor will move to the next line when you have stepped through each of the statements. The statements on a line are numbered from 0.

3.3.8 Examining the call stack

Step 22. You can find out where the factorial function was called by examining the call stack. You do this using the browser window.

The **Last Event** operation moved the browser to the frame level, so the browser window already displays the current list of calls with the most recent at the top. Otherwise you would click on the **Call Stack** button to display the call stack.

Notice that there is one call of factorial at the top of the stack. Click on the **Step** button again to call factorial again. The new call of factorial will appear at the top of the stack, as in figure 3.14.

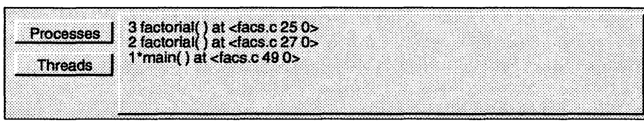

Figure 3.14 Call stack for the factorial function

Step 23. You can display the source code of any of the functions on the stack to see where the call was made.

Move the cursor to the browser window and click on '1*main() at <facs.c 49 0>'. The code window changes to display the code of function main, with the call to factorial marked as the current location. The asterisk (*) in the browser window line means that the thread was created by main.

Step 24. main and each of the factorial calls has a local variable called n. These three variables are different and may have different values. Printing the value of n accesses the n for the current frame.

Double click on n and click on the **Print** button to display the value of n in main, which should be 1.

Select the first call to factorial by clicking on '2 factorial() at <facs.c 27 0>' in the browser window. Select n and print it again. Again it should be 1, although this is a different variable which happens to have the same value.

Select the second call to factorial by clicking on '3 factorial() at <facs.c 25 0>' in the browser window. Print n again. This time it should be 0.

3.3.9 Stepping over and out of function calls

Step 25. The first call of function `factorial` is currently waiting for the second call to finish. The **Step Out** button has been provided so that you can complete a function call in a single operation. It has the effect of continuing execution of the thread back to the current frame or, if no frame is selected, until the current call has returned.

Select the frame '`2 factorial() at <facs.c 27 0>`'. Click on the **Step Out** button to step out to the selected frame. Notice that the current location marker stays on line 27. This is because the function result still has to be multiplied by `n` and returned. Click on **Step Out** again to return from the current call to `main`.

Step 26. You can continue execution up to a selected statement, by using the **Step To** button. This has the same effect as setting a new breakpoint, clicking **Continue**, and then removing the breakpoint with **Delete**.

Select line 53 by clicking on the function name `send_next`. This line will not be executed until the factorials of the numbers 1 to 4 have been calculated and sent.

```
41          while (going)
42          {
43             int n, tag;
44
45             tag = read_chan (in, &n);
46             switch (tag)
47             {
48                case DATA: {
49                   send_data (out, factorial(n));
50                   break;
51                }
52                case NEXT: {    /* start a new sequence */
53                   send_next (out);
54                   break;
55                }
56                case END: {     /* terminate */
57                   going = FALSE;
58                   send_end (out);
```

Figure 3.15 Selecting line 53

Step 27. Now click on the **Step To** button to continue execution until line 53 is reached. Notice that the current line marker now rests on line 53 and the message displayed in the output window is:

```
2 1 facs main : breakpoint 4 at <facs.c 53 0>
```

Step 28. If you step through a function using the **Step** operation you will step into any functions that are called. If you do not want to step into function calls you can use the **Next** operation instead. This steps over function calls, a little like using a combination of **Step** and **Step Out**.

Click on **Next** and the program will step over the call of `send_next`. Continue to click the **Next** button until the current line marker moves to line 41, the start of the `while` loop.

3.3.10 Listing the current threads

Step 29. The next step of the example session is to list the threads of the `app` process. Move the cursor to the browser window and click on the **Processes** button to display the list of processes.

Step 30. Select the `app` process by clicking on '`app`'. The browser window shows a list of the threads that were in `app` the last time the process was stopped. Only one thread, `main`, is listed.

Step 31. To see a list of the current threads, pull down the **Execution** menu and click on **Find Threads**. The cursor will change to a watch while the debugger looks for the threads. Three new threads will appear, which were generated by a `ProcPar` in `main`. The three new threads are `control`, `sum` and `feed`. None of these threads is selected so anything you do at this level will be applied to all the threads.

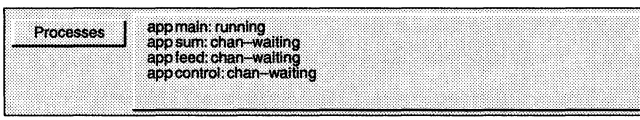

Figure 3.16 The `app` process with all threads listed

3.3.11 Interrupting running threads

Step 32. The next step of the example session is to stop the program by interrupting it. First you must set all the threads running again.

Move to program level by clicking on the **Processes** button in the browse window. Set all the threads running by clicking on the **Continue** button.

Step 33. To stop a process or thread running you use the **Interrupt** operation.

Click on the **Interrupt** button to interrupt all the threads. Notice that the output window tells you that an interrupt has been requested. Select the `app` process and observe that the browser window now shows that threads `control`, `sum`, and `feed` are all waiting for channel communications. This means that the threads have not yet reached their interrupt breakpoints because they are waiting indirectly for the keyboard input.

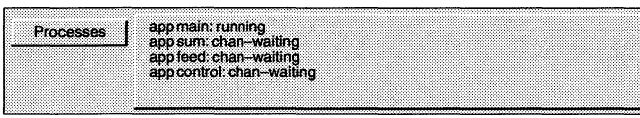

Figure 3.17 The `app` process after being interrupted

Move the cursor to the program window and type '`6`' followed by carriage return. The `control` thread changes to stopped since it has now hit its interrupt.

Step 34. You can now look at where the thread has stopped. Click on the **Last Event** button to show where the program has stopped. The code window displays the code for the `control` thread. This handles the terminal input and output. Notice that it is currently waiting on a '`do ... while (scanf(...)..)`' statement, trying to complete the keyboard input.

Step 35. Click on **Continue** to set the control thread running again. Click on the **Threads** button and observe that the `feed` thread is now stopped. That means it has completed its channel input from `control` and hit its interrupt. Click on **Last Event** to show where the interrupt occurred.

3.3.12 Watching communication between two threads

Step 36. You are now going to watch the communication between `feed` and `facs`.

Select line 13, containing the `ChanOutInt` statement, by clicking on it. Continue execution to line 13 by clicking on the **Step To** button. Click on the **Threads** button to return to thread level.

Step 37. We need to view the source of `feed` and `facs` at the same time. The debugger has an **Open Window** operation to open another display.

Click on the **File** menu at the top of the debug display.

Step 38. Select the **Open Window** option from the **File** menu. This opens another debugging display in the same state. On a Sun, a complete duplicate display is generated, as in figure 3.18.

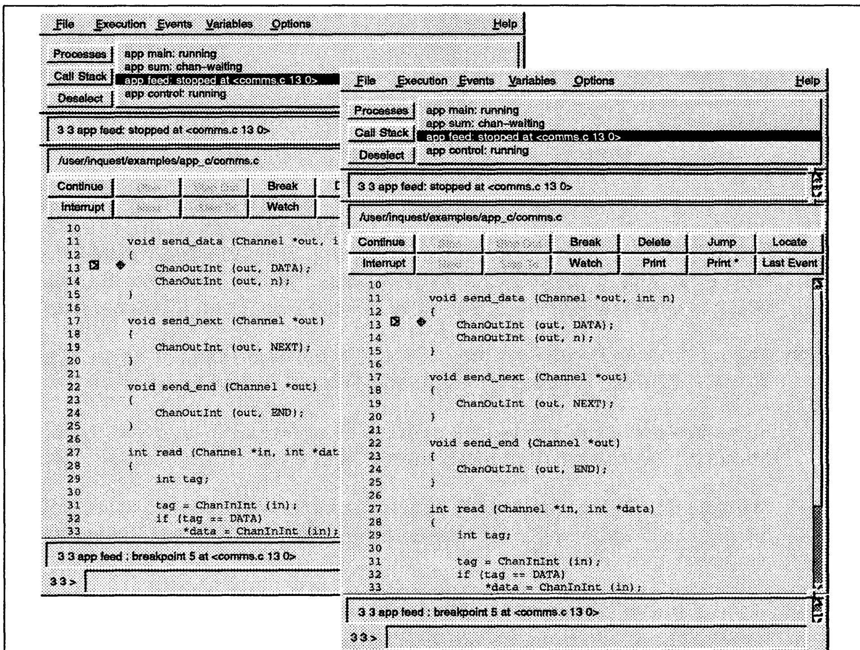

Figure 3.18 Two X-Windows debugging displays open

On the PC, only an extra code window is generated, as shown in figure 3.19. The buttons and other windows apply to the code window that is selected. The **Window** menu provides operations to tile the code windows, pile them up or select one.

Make sure that the second display is positioned so that you can see enough of both code windows.

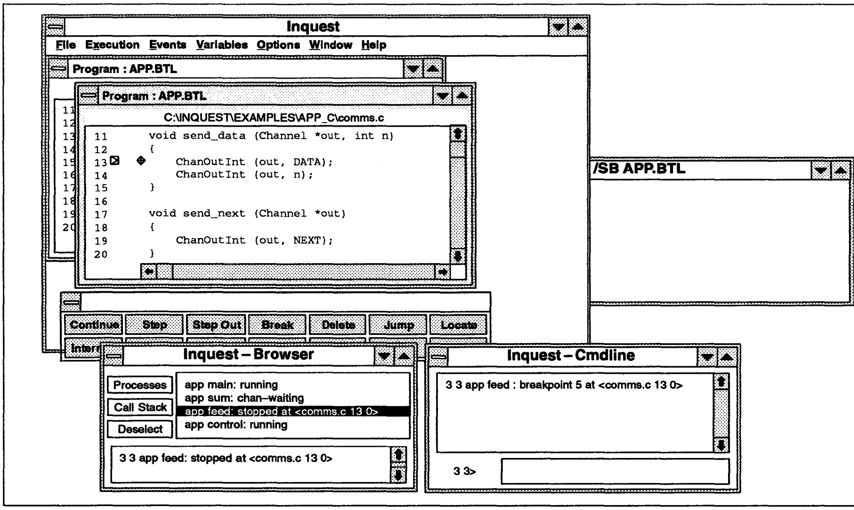

Figure 3.19 Two PC code windows open

Step 39. Change the new display to look at the `facs` thread by clicking on **Processes** and then the `facs` process in the browser window. Click on the `facs main` thread in the browser window to show the `facs` code. We shall call this window the `facs` window and call the other window the `feed` window. You can tell which is which by looking at the browser window or the attributes window.

Step 40. The `out` channel in `feed` is the other end of the same channel as `in` in `facs`. The `feed` thread is about to output and `facs` is waiting to input.

In the `feed` window, single step over the `ChanOutInt` statement by clicking on **Step**. The `facs` thread reads the `DATA` tag, hits its interrupt and stops.

Step 41. In the `feed` window, **Step** again for the second output. `feed` cannot complete the step as it is now waiting for `facs` to input from the channel. In the browser window it is now stepping.

Step 42. In the `facs` window, click on **Step** to read the tag. **Step** again to assign the result to the variable `tag` and again to test its value. It is now about to input from `feed`. **Step** once more to complete the input. `feed` has now also completed its output.

Step 43. You can now get rid of one of the debug displays.

Click on the **File** menu at the top of the `feed` debug display. Select the **Close Window** option from the file menu. The `feed` debug display disappears.

Step 44. Click on **Step Out** to return to the `facs` main code.

3.3.13 Setting a watchpoint on a variable

Step 45. You can set a watchpoint on a variable so that every time the variable is about to be changed the thread stops. You are now going to set a watchpoint on the `n` variable that holds the value whose factorial is being calculated.

Highlight the `n` variable on line 49 of `facs.c` by double clicking on it.

Step 46. Set a watchpoint on the highlighted variable by clicking on the **Watch** button. Notice that the watch symbol appears next to line 43 where `n` is declared and the following message appears in the output window:

```
watchpoint 7 on <facs.c 43 n> address #8000a174 to #8000a178 of facs main frame
```

This tells you that the watchpoint is event 7. The addresses give the location of `n`. If you scroll the output window up one line you will also see:

```
2 1 > watch n -1 <facs.c 49>;
```

Step 47. You now need to set the program running. Click on **Processes** and then **Continue** to restart all the threads. The program will hit the watchpoint and display in the output window:

```
2 1 facs main : watchpoint 7 at <comms.c 33 0>
```

Step 48. You can now examine the value of the variable with the watchpoint.

Click on **Last Event** to see the watchpoint. The local name for `n` is `*data`. Select the variable `data` by double clicking on it and then click on the **Print *** button to print the value of the variable that `data` points to. The output window shows `1`, which is the value before it changes. Make one **Step**, select `data` and **Print *** again to see that it has changed to 2.

Step 49. Click on the **Continue** button to go on to the next watchpoint.

Step 50. If you want to trace the value of the variable as the thread runs you can make the debugger do an automatic **Print** and **Continue** every time the watchpoint occurs by issuing a suitable command in the command window.

Move the cursor to the command window and click the left mouse button to make the window active.

Step 51. Type in the following command, followed by a carriage return:

```
when (7) {w=step; wait(w); fid=1; write n is (print n); continue}
```

This tells the debugger that when event 7 occurs it must:

- step (so that `n` is assigned its new value), saving the event number as `w`,
- wait until the step completes,

- move down the stack, so that `n` is visible, to thread `main` , which has a `fid` of 1,
- output the text '`n is`' together with the value of `n` and
- continue with execution of the program.

Step 52. If you are using a Sun, then the output window is only one line deep. You can change the height of this window so that you can see a history of the most recent messages. You do this by dragging the sizing box.

Move the cursor to the output window sizing box, which is the small box at the top of the scroll bar, as shown in figure 3.20. Notice that the cursor changes to a cross when it is on the box. Click and hold the mouse button down whilst you drag the box up the screen. Keep dragging until you have an output window four lines deep.

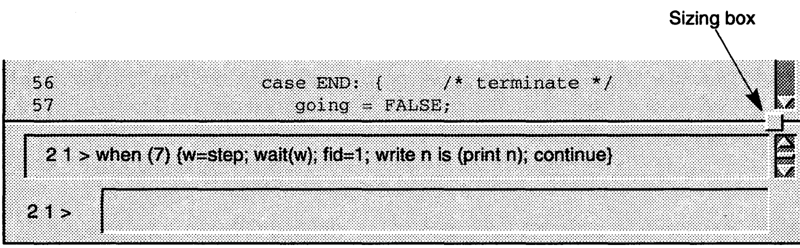

Figure 3.20 The Sun output window before re-sizing

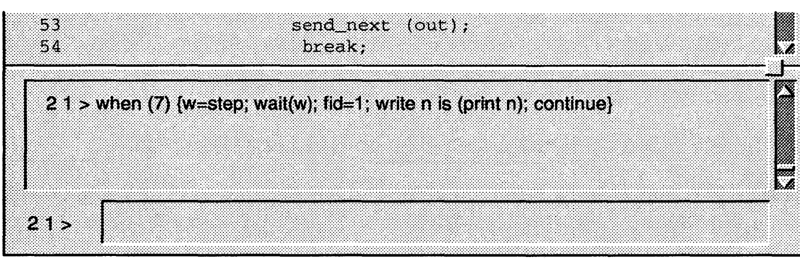

Figure 3.21 The enlarged Sun output window

Step 53. Click on the **Continue** button. The `when` command will keep printing the value of `n` as the `facs` function increments it to the number you entered then sets it back to 0 ready for the next input.

3.3.14 Deleting a watchpoint

Step 54. You can now delete the watchpoint you set on `n`. Click on **Last Event** to locate to the watchpoint again.

Step 55. To set a watchpoint we selected a variable. To delete the watchpoint we select the declaration line. Select line 43 of `facs.c` by clicking on it. This line is the declaration of `n`, which has the watchpoint symbol, ◷ on Suns or on PCs, on it.

```
34    {
35        Channel *in, *out;
36        int going = TRUE;
37
38        in = get_param(1);
39        out = get_param(2);
40
41        while (going)
42        {
43            int n, tag;
44
45            tag = read_chan (in, &n);
46            switch (tag)
47            {
48                case DATA: {
49                    send_data (out, factorial(n));
50                    break;
51                }
52                case NEXT: {   /* start a new sequence */
53                    send_next (out);
54                    break;
```

Figure 3.22 Line 43 with watchpoint set

Step 56. With the declaration line selected, click on the **Delete** button to remove the watchpoint. The watchpoint symbol disappears.

Step 57. Click on the **Continue** button.

Step 58. Type in another number at the prompt '`Please type n :`'.

The program now executes normally since the watchpoint has been removed.

3.3.15 Jumping down a channel

To explore the wider context of a thread we may wish to look at the threads that communicate with it. To find a thread that is waiting for a communication, we can jump down a channel. This means that we are changing context to the thread that is waiting for communication on that channel.

At most one thread can ever be waiting for one channel; when two threads are ready then communication starts. If no thread is waiting then the **Jump** operation will not change the context. If the current thread is waiting then, again, the **Jump** operation will not change the context.

Step 59. Thread `facs` has a `send_data` function call at line 49 which sends data down the channel to the next thread. You are now going to use this statement to jump to the thread that is waiting on the other end of the channel.

First you must interrupt execution of `facs`, as `facs` must be stopped. Set an interrupt on `facs` by clicking on the **Interrupt** button.

Step 60. Enter a value for `n` in the program window. The attribute window should show that the `facs` thread has stopped at the interrupt.

Step 61. The interrupt has left the current line inside the `read_chan` function. Move back down the frame stack to the `main` thread by clicking on '`1*main() at <facs.c 45 0>`. The call to `send_data` should now be visible again on line 49. Double click on the name of the channel `out` to select it.

Step 62. Click on the **Jump** button. The context changes to the process `square`, which is waiting for input. This is the other end of the output channel from `facs`.

3.3.16 Leaving the debugger

Step 63. You have now finished the example interactive debugging session. If you want to exit from the debugger, click on the **File** menu and select the **Exit** or **Quit** option. A pop-up dialog box will appear asking you to confirm the exit request. Select **Yes** and the debugger will close its windows and exit.

On PC systems, you may wish to also close the **ISERV** window.

This concludes the tutorial for interactive debugging with C programs.

4 An example ANSI C post-mortem debugging session

This chapter takes you in detail, step by step, through one example post-mortem debugging session, to demonstrate the features that are available for post-mortem debugging, using an ANSI C program as the example. A similar occam example is described in chapter 6.

This chapter shows you how to:

- build the program for post-mortem debugging;
- start the debugger when the program fails;
- locate where the crash has occurred and the reason for it;
- examine a variable;
- examine a call stack;
- jump down a channel;
- quit from the debugger.

Before starting the session you need to know a little about the example program. This is the same program as is used in chapter 3, so if you have not worked through the interactive debugging session, read section 3.1 before proceeding.

Before starting the tutorial you may find it useful to have a listing of the example program source code. It would also be useful to look at figure 2.1 or figure 2.2 in section 2.4 to show you the names of the parts of the debugger display.

4.1 Post-mortem debugging

Post-mortem debugging means debugging after the program has crashed, terminated or been stopped. It may have crashed during normal running or during an interactive debugging session. In this tutorial, we shall start from normal execution. Starting from an interactive session is easier and is described in section 2.4 of chapter 2 in the *INQUEST User and Reference Manual.*

During post-mortem debugging you can navigate through the code and explore the state of the program exactly as in interactive debugging. However, the program cannot be restarted, so stepping, interrupts, breakpoints and watchpoints do not apply.

4.1.1 Building the code

Post-mortem debugging only requires that the code has been compiled with full debugging data, i.e. with the G option.

On Sun systems, the example program makefile builds the example application for you and is called `makefile`. On a PC, a batch file called `build.bat` is provided to do this. This generates a bootable file for post-mortem debugging called `app_pm.btl`.

For the purposes of this tutorial, all the processes are configured to run on the same processor.

4.1.2 Starting post-mortem debugging

When a program crashes during normal execution then the host is returned to the state before the program was run. Post-mortem debugging can then be started by the `inquest` command.

On a PC this can be done from the File Manager by double clicking on the `inquest.exe` program, which makes a command line box pop up. The browse button is used to select the application `app_pm.btl`. The `-pm` option is added in the **Options** box to request post-mortem debugging.

On a Sun this is done by typing the inquest command at the operating system prompt with the `-pm` option to request post-mortem debugging:

```
inquest app_pm.btl -pm
```

The debugger will *analyze* the target hardware. This means that signals will be sent to the target to halt any threads that may be still running, and then the hardware will be reset without losing any state information. After analyzing the target, the debugger uses a network mapper to explore the state of the target. The debugging display will then appear on the screen.

If the error flag was set, then a message will appear showing where the error occurred. The source line where the error happened is automatically located by the debugger. The process causing the error can then be explored and other processes located using **Jump**.

4.2 Step-by-step tutorial

Here is a step-by-step tutorial to guide you through the main features of INQUEST post-mortem debugging.

Step 1. Move to the `app_c` sub-directory in the `examples` directory which contains the ANSI C example program `app_pm`.

Step 2. Check that the `TRANSPUTER`, `ASERVDB` and `ISEARCH` environment parameters are correctly defined.

Step 3. The root transputer must be an IMS T400, IMS T425, IMS T801, IMS T805, ST20450 (T450) or IMS T9000. The example is configured for a single ST20 or for a single IMS T9000 depending on the INQUEST version.

If you are using a different type of root transputer then you will need to edit:

- the hardware configuration file, `hardware`;
- for Suns the `make` macro file `tools` or for PCs the build batch file `build.bat`.

The changes to make in each case are given in table 4.1.

Target	All users: file `hardware`	Sun users only: file `tools`	PC users only: file `build.bat`
ST20450 / T450	No change	No change	No change
IMS T9000	No change	No change	No change
IMS T400 or T425	Change `T450` to `T425`	Change `450` to `425`	Change `t450` to `t425`
IMS T801 or T805	Change `T450` to `T805`	Change `450` to `805`	Change `t450` to `t805`

Table 4.1 Changes to examples to support different targets

Step 4. Create a bootable file, suitable for debugging. To do this on a Sun, at an operating system prompt type:

```
make
```

On a PC, type:

```
build
```

4.2.1 Starting and crashing the application

Step 5. Start the example application, `app_pm.btl`, which is the bootable code for post-mortem debugging that you created in Step 4.

On Sun systems this is done by typing:

```
irun app_pm.btl
```

On PC systems, start up Windows, open the File Manager and double click on the `inquest.exe` program. This will open the **Command line** dialog box. Use the browse button next to the **File** field to find and select the application `app_pm.btl` in the `app_c` sub-directory in the `examples` directory. Click on the `Run` button in the **Command line** box.

Since the code has not been configured for debugging, this loads the application for normal execution onto the target hardware. The parameters you can give to `irun` are described fully in the `irun` chapter of your *Toolset Reference Manual*.

Step 6. The program will ask for a number with the prompt:

```
Sum of the first n (n < 9) squares of factorials
Please type n :
```

Enter the number 6 to make the program continue.

Step 7. Next time the program asks for a number, enter Control-C. This will halt the server. The application program will continue running until the target hardware is analyzed when the post-mortem debugger is loaded. In this case, the program is waiting for keyboard input, so the state will not change.

4.2.2 Starting the debugger

Step 8. On PC systems, in the File Manager, double click on the `inquest.exe` program. This will open the **Command line** box again. Use the browse button again next to the **File** field to find and select the application `app_pm.btl` in the `app_c` sub-directory in the `examples` directory. This time type `-pm` in the **Options** field to tell INQUEST to do a post-mortem debug. Click on the `Run` button in the **Command line**.

On Sun systems, start the post-mortem debugger with the command:

```
inquest app_pm.btl -pm
```

Step 9. Wait while a debugging display is created. This will show the configuration file in the code window and a list of the example program's processes in the browser window. This is the program level display. Figure 4.2 shows the X-Windows display; the PC display is shown in figure 4.1. The debugger has explored the target hardware and is showing the last known state of the program when it was stopped. If the program had set the error flag, then the debugger would have automatically located to where the error occurred.

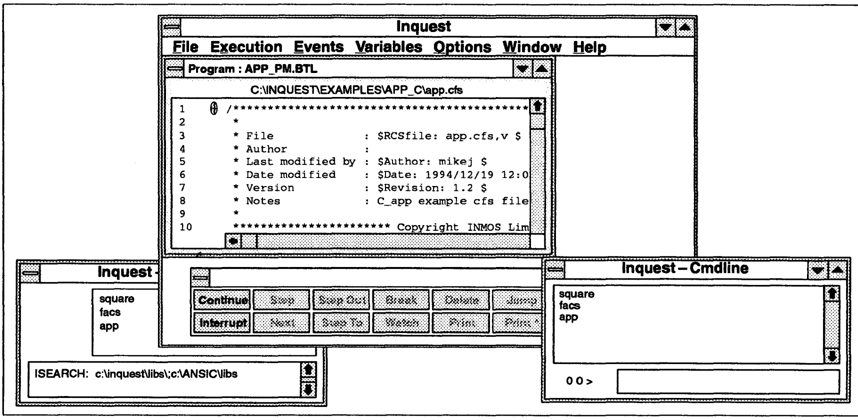

Figure 4.1 The Microsoft Windows start-up display for the example program

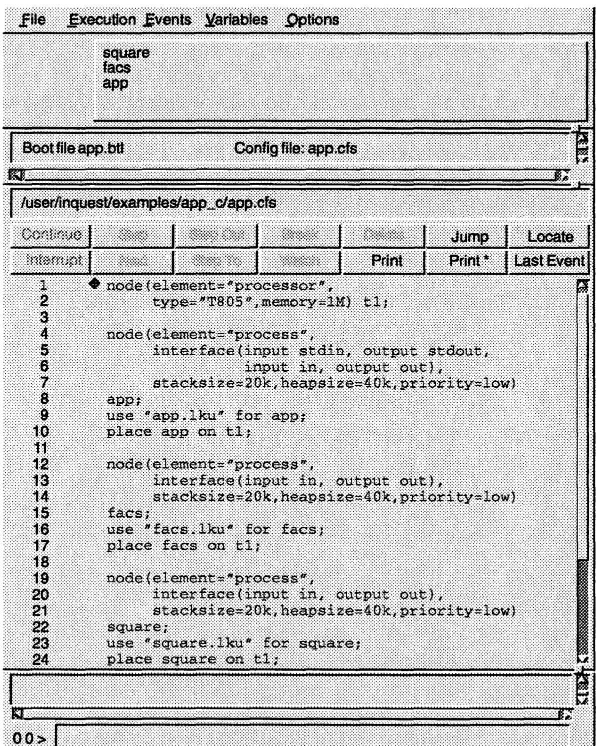

Figure 4.2 The X-Windows initial display for the example program

4.2.3 Using the browser

Step 10. Since the program was halted by stopping the host server, and since the program had not deadlocked, we would expect that the `control` thread would be waiting for a host communication. We will use the browser to check this.

To tell INQUEST to search for the threads, click on **Find Threads** in the **Execution** menu. Move the cursor to the browser window, where the three processes `square`, `facs` and `app` are listed. The control thread is part of the `app` process, so click on `app`. This moves the display to process level, with the browser window showing a list of threads in the process with their states, as shown in figure 4.3. All the threads are waiting for channels. We would expect this because the threads continued to run until they were halted by waiting for a communication with another halted thread. The code window shows `app.c`.

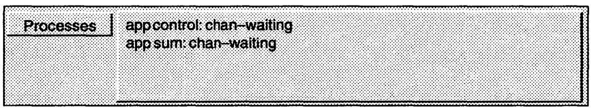

Figure 4.3 Process level browser display

Step 11. To move to the control thread, click on '`app control: chan-waiting`'. The display changes to the thread level display, as shown in figure 4.4. The selected thread is highlighted and the code window shows the source code for the control thread, with the next statement marker, ⊠ on Suns or ⊳ on PCs, on the line that was waiting to complete, as shown in figure 4.5.

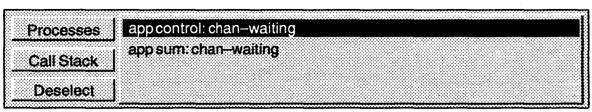

Figure 4.4 Thread level browser display

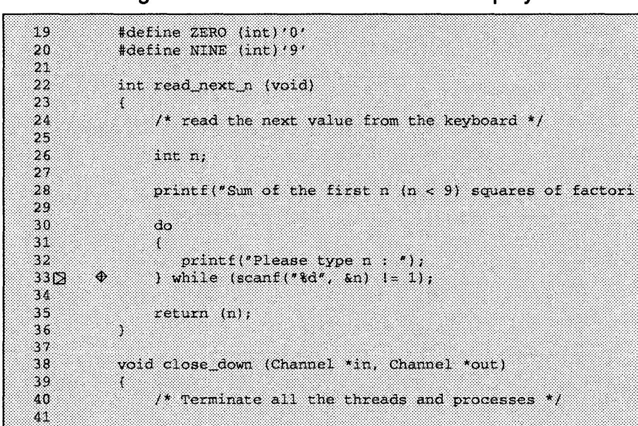

Figure 4.5 `control` thread

4.2.4 Examining the call stack

Step 12. The next statement marker, ⍄ on Suns or ▶ on PCs, is in the function **read_next_n**. You can find out where this function was called by using the call stack.

To see the call stack, move to the frame level by clicking on the **Call Stack** button in the browser window. This changes the browser window to show the call stack.

Each line of the call stack represents a function call in the current thread which has not yet returned. The last few calls are to routines used by **scanf**. **scanf** itself is at line 6, so scroll down to this line, so that the browser window looks like figure 4.6.

The top functions are system functions so they have no debugging data; the debugger only knows that the code is in the library **libc.lib**. The next line of the browser window shows that **scanf** was called by **read_next_n** at line 30 in source file **control.c**. **read_next_n** was called by **control** at line 59. **control** was called by **main** at line 50 of **app.c**, using **ProcPar** and **ProcParList%c**.

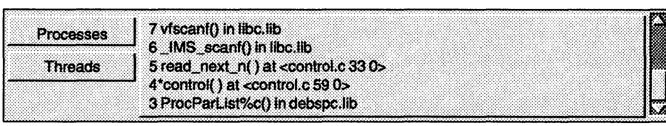

Figure 4.6 Call stack for the **factorial** function

Step 13. You can display the source code of any of the functions on the stack to see where the call was made.

Move the cursor to the browser window and click on '**4*control() at <control.c 59 0>**'. The code window changes to display the code of function **control**, with the call to **read_next_n** marked with the next statement marker, ⍄ on Suns or ▶ on PCs. The asterisk (*) in the browser window line means that the thread was created by **control**.

Step 14. Click on '**1*main() at <app.c 50 0>**' to show the line where control was called; the **ProcPar** statement.

Return to the control frame by clicking on '**4*control() at <control.c 59 0>**' in the browser window.

4.2.5 Examining a variable

Step 15. You are now going to examine the value of a variable.

Place the cursor on the variable '`n`' on line 54, 59, 61, 64 or 67, and highlight it by double clicking the left mouse button. This causes the whole word to be selected, which in this case is just the single letter.

Step 16. Display the value of the variable by clicking on the **Print** button. The **Print** operation displays the value in the output window; it does not produce a hard copy. The output window should show the message:

```
6
```

This tells you that `n` has the value 6, which was the value you last typed in.

Moving up and down the stack, examining the variables at each level, gives you a clear view of what happened to the thread just before it crashed.

4.2.6 Jumping down a channel

To explore the wider context of the current thread we may wish to look at the threads that communicate with it. To find a thread that is waiting for a communication, we can jump down a channel. This means that we are changing the context to the thread that is waiting for communication on that channel.

At most one thread can ever be waiting for one channel; when two threads are ready then communication starts. If no thread is waiting then the **Jump** operation will not change the context.

Step 17. Move back to the `control` function by selecting it on the call stack.

Step 18. Select the channel `in` by clicking twice on a reference to it. Click on the **Jump** operation button to jump down the `in` channel. The `in` channel is empty, i.e. no thread is waiting for it, so nothing happens, except that the message

```
*empty*
```

appears in the output window.

Step 19. Select the channel `out` by clicking twice on it. Click on the **Jump** operation button to jump down the `out` channel. The context will change to the `feed` thread, which is waiting for input on this channel, which is called `in` within the `feed` function. The next statement marker shows exactly which input the thread is waiting for.

Step 20. If we want to jump to the next thread in the pipeline then we need to use the channel `out` in `feed`. This channel is not in scope, because we are in the function `read_chan` which does not use channel `out`. To come out of `read_chan`, click on the Call Stack button in the browser window to show the call stack and select the line below `read_chan`, which says '`4*feed at <feed.c 27 0>`'. The context changes to the `feed` function, with the `out` channel in scope.

Step 21. Select the `out` channel in `feed` and click on the **Jump** operation button again. The context will change again to the `facs` thread, which is waiting for input on this channel, which is called `in` within the `facs` procedure.

4.2.7 Leaving the debugger

Step 22. You have now finished the example post-mortem debugging session. If you want to exit from the debugger, click on the **File** menu and select the **Exit** or **Quit** option. A pop-up dialog box will appear asking you to confirm the exit request. Select **Yes** and the debugger will close its windows and exit.

This concludes the tutorial for C programs on post-mortem debugging.

5 An example occam interactive debugging session

This chapter takes you in detail, step by step, through one example interactive debugging session, to demonstrate the basic features of the INQUEST debugger, using an occam program as the example. A similar ANSI C example is described in chapter 3, and post-mortem debugging is shown in chapter 6.

This chapter shows you how to:

- build the program for debugging;
- start the debugger;
- place a breakpoint;
- start the processes running;
- locate where a breakpoint has occurred;
- examine a variable;
- remove a breakpoint;
- single step through the source code;
- examine a call stack;
- step over function calls;
- interrupt a running process;
- watch communication between two threads;
- set a watchpoint on a variable;
- delete a watchpoint;
- jump down a channel;
- quit from the debugger.

Before starting the session you need to know a little about the example program. This is described in section 5.1 below.

5.1 The example program

The example debugging session uses an example program called `app`, which you will find in the directory `app_occ` within the `examples` directory. The directory contains all the source code and makefiles.

This is a simple multi-process program. The processes are arranged in a pipeline that generates the sum of a series of squares of factorials, as in the following formula:

$$\sum_{i=1}^{n} factorial(i)^2$$

It is not an efficient program, but it provides the structures we need to try out the debugger. The program consists of five configuration-level processes; `control`, `feed`, `facs`, `square` and `sum`.

The debugger uses slightly different terminology from the usual occam terms in order to distinguish between static and dynamic processes. All the processes listed above, `control`, `feed`, `facs`, `square` and `sum`, are all called threads. They are mapped onto the processor `RootNode` in the configuration code. The debugger calls all the code on one processor a process and gives it the name of the processor with a `_p` suffix. The example program has only one process, `RootNode_p`. If you have more than one processor available, you may wish to re-configure the program for three processors so that you can see a program with more than one process.

Figure 5.1 shows how the threads connect together.

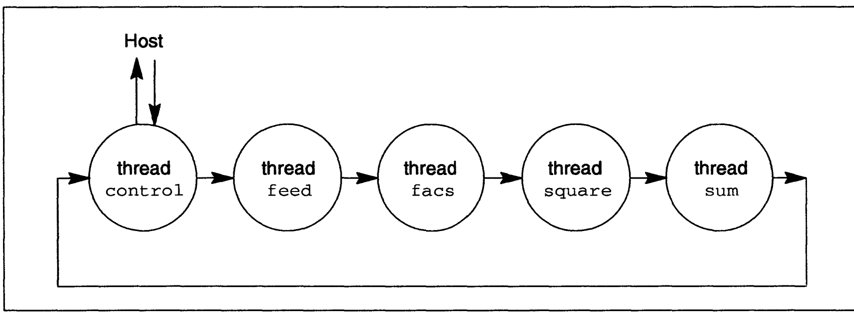

Figure 5.1 The example program

- `control` – displays the message

  ```
  Sum of the first n (n < 9) squares of factorials
  ```

  and asks you to type in a value for `n`. Numbers bigger than 8 would cause an overflow. The number you type is sent to `feed`. Zero or negative numbers terminate the program.

- `feed` – receives the number `n` from `control` and sends all the numbers from 1 to `n` to `facs`, one at a time. It then sends a signal, `next`, to say that this batch of data is complete.

- `facs` – generates the factorial of each number it receives from `feed`, and sends the result to `square`. It passes on the signal `next` to `square`.

- **square** – generates the square of each number it receives from **facs** and sends the result to **sum**. It passes on the signal **next** to **sum**.
- **sum** – generates the sum of all the numbers it receives from **square** until it receives the signal **next**. On receiving **next**, **sum** passes the result on to **control**.
- **control** – displays the result it received from **sum**.

To terminate the program you type zero in response to the '**Please type n :**' message issued by **control**. This causes the signal **end** to be passed down the pipeline. Each thread dies after it has passed on **end**.

Before starting the tutorial you may find it useful to have a listing of the example program source code. It would also be useful to look at figure 2.1 or figure 2.2 in section 2.4 to show you the names of the parts of the debugger display.

5.2 Building the example program

Any occam program that you want to debug must be built in the following way:

- Compiled using **oc** without the **D** option.
- Linked using **ilink**.
- Configured using **occonf** with the **GA** option.
- Collected using **icollect**.

On Sun systems, the example program's makefile does all this for you and is called **makefile**. On a PC, a batch file called **build.bat** is provided to do this. This generates a bootable file for interactive debugging called **app.btl**.

The contents of the configuration file **app.pgm** are shown in figure 5.3. The configuration **#INCLUDE**s the hardware configuration file **hardocc**, shown in figure 5.2.

For the purposes of this tutorial, all the threads are configured to run on the same processor so there is only one process.

```
VAL K IS 1024:
VAL M IS K * K:

NODE RootNode:
ARC HostLink:
NETWORK
  DO
    SET RootNode (type, memsize := "T450", 512K)
    CONNECT RootNode[link][0] TO HOST WITH HostLink
:
```

Figure 5.2 ST20450 hardware configuration **hardocc**

```
#INCLUDE "hardocc"
#INCLUDE "hostio.inc"
#INCLUDE "pipe.inc"

#USE "control.lku"
#USE "feed.lku"
#USE "facs.lku"
#USE "square.lku"
#USE "sum.lku"

CONFIG
  CHAN OF SP fs, ts:
  PLACE fs, ts ON HostLink:
  CHAN OF PIPE control.to.feed, feed.to.facs:
  CHAN OF PIPE facs.to.square, square.to.sum:
  CHAN OF PIPE sum.to.control:

  PROCESSOR RootNode
    PAR
      control (fs, ts, sum.to.control, control.to.feed)
      feed (control.to.feed, feed.to.facs)
      facs (feed.to.facs, facs.to.square)
      square (facs.to.square, square.to.sum)
      sum (square.to.sum, sum.to.control)
:
```

Figure 5.3 Configuration source code `app.pgm`

5.3 Step-by-step tutorial

Here is a step-by-step tutorial to guide you through the main features of the INQUEST debugger used for interactive debugging.

Step 1. Move to the `app_occ` sub-directory in the `examples` directory which contains the occam example program `app`. Run the set-up script to set up the environment, as follows.

On Sun systems, at a prompt type:

```
source setup.csh
```

or

```
. setup.sh
```

On PC systems, at a DOS prompt type:

```
setinq
```

Step 2. Check that the `TRANSPUTER` parameter is correctly defined.

Step 3. The root transputer must be an IMS T400, IMS T425, IMS T801, IMS T805, ST20450 (T450) or IMS T9000. The example is configured for a single ST20 or for a single IMS T9000 depending on the INQUEST version.

If you are using a different type of root transputer then you will need to edit:

- the hardware configuration file, `hardocc`;
- for Suns the `make` macro file `tools` or for PCs the build batch file `build.bat`.

The changes to make in each case are given in table 5.1.

Target	All users: file `hardocc`	Sun users only: file `tools`	PC users only: file `build.bat`
ST20450 / T450	No change	No change	No change
IMS T9000	No change	No change	No change
IMS T400 or T425	Change `T450` to `T425`	Change `450` to `425`	Change `t450` to `t425`
IMS T801 or T805	Change `T450` to `T805`	Change `450` to `805`	Change `t450` to `t805`

Table 5.1 Changes to examples to support different targets

Step 4. Build a bootable file, suitable for debugging. To do this on a Sun, type at an operating system prompt:

```
make
```

On a PC, type:

```
build
```

5.3.1 Starting the debugger

Step 5. Start the debugger with the example program, `app.btl`, which is the bootable code you created in Step 4.

On Sun systems type:

```
inquest app.btl
```

On PC systems, start up Windows, open the File Manager and double click on the `inquest.exe` program. This will open the **Command line** dialog box. Use the browse button next to the **File** field to find and select the application `app.btl` in the `app_occ` sub-directory in the `examples` directory. Click on the `Run` button in the **Command line**.

This loads the debugger onto the host computer and a small kernel of debugger code onto each processor of the transputer network that has processes to be debugged.

The parameters you can give to `inquest` are described fully in the *INQUEST User and Reference Manual.*

Step 6. Wait while a debugging display is created. This will show the configuration file in the code window and a list of the example program's processes in the browser window. This is the program level display. Figure 5.4 shows the PC display; the X-Windows display is shown in figure 5.5.

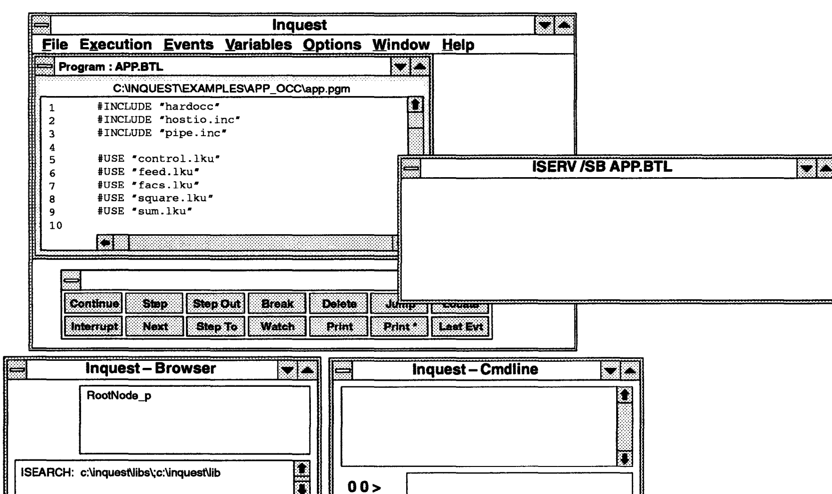

Figure 5.4 The Microsoft Windows start-up display for the example program

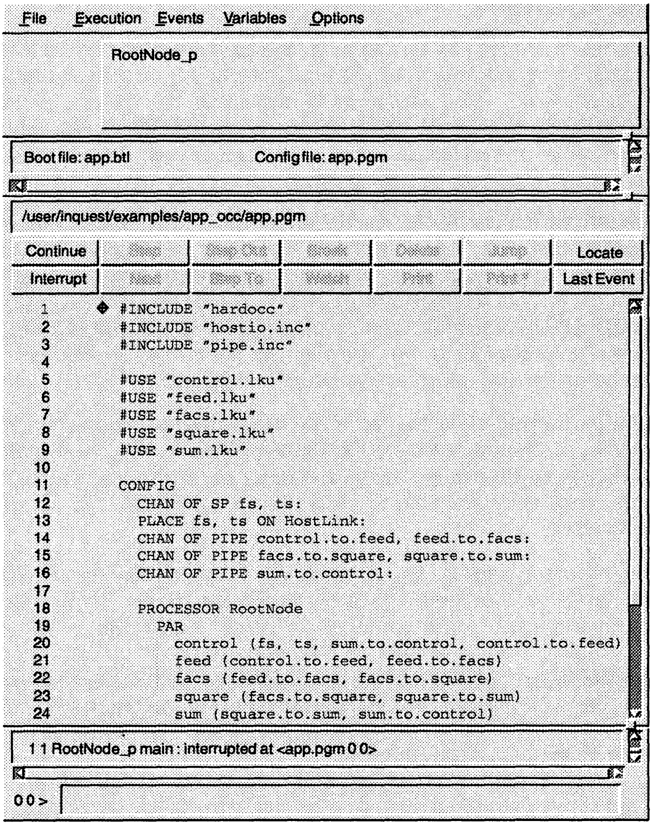

Figure 5.5 The X-Windows initial display for the example program

On a PC a new input and output window is created for the application, labelled something like **ISERV /SB APP.BTL**. On a Sun, the input and output of the program being debugged will be shown in the window that the debugger was started from. In either case, this will be called the program window in the rest of this document.

The transputer program has been halted a few instructions before the beginning of each process.

5.3.2 Single stepping

Step 7. You are now going to step through the configuration code until some more threads are generated. Move the cursor to the browser window and click on the only process '`RootNode_p`'. This changes the browser to process level, giving a list of threads in the browser window. Click on the only thread '`RootNode_p main: stopped at <app.pgm 0 0>`' to select it. This changes the browser to thread level.

Step 8. occam programs are initially stopped several steps before the start of the user code. To start at the `PAR` statement in the configuration code, use the scroll bar on the right of the code window to scroll the window down, and click on the `PAR` on line 19. The selected line marker, ⊕ on Suns or ◇ on PCs, will move alongside the statement. Click on the **Step To** button to step through the initial code up to the selected line. The current line marker, ⊠ on Suns or ⊳ on PCs, moves next to the selected line marker on line 19.

Step 9. Click on **Step** to start the `PAR`. The browser window says that the thread is stepping. It cannot complete the **Step** until the `PAR` has terminated, i.e. until `control`, `feed`, `facs`, `square` and `sum` have all terminated. The **Step** button turns grey, since the thread cannot step again until the current step has completed.

Step 10. The threads `control`, `feed`, `facs`, `square` and `sum` have now been generated, and they are listed in the browser window. The window is not big enough for five threads, so to see the line for thread `RootNode_p main[5]`, scroll down using the scroll bar on the right of the browser window. Thread `RootNode_p main` is the top level process which cannot continue until the `PAR` has terminated. Threads `RootNode_p main[1]` to `[5]` are the processes `control`, `feed`, `facs`, `square` and `sum` respectively, in the same order as they are listed in the `PAR`.

```
 1
 2    -- facs.occ
 3
 4    -- generate factorials
 5
 6    #INCLUDE "pipe.inc"
 7
 8    PROC facs (CHAN OF PIPE from.feed, to.square)
 9
10      -- compute factorial
11
12      INT FUNCTION factorial (VAL INT n)
13        INT result:
14        VALOF
15          SEQ
16            result := 1
17            IF
18              n > 0
19                SEQ i = 1 FOR n
20                  result := result * i
21              TRUE
22                SKIP
23
24          RESULT result
```

Figure 5.6 `facs` procedure code window

Step 11. We can now look at the facs thread, which is stopped. Select it by clicking on 'RootNode_p main[3]: stopped at <app.pgm 37 0>' in the browser window. In the code window, the line markers move to the facs procedure call. Click on the **Step** button to enter the facs procedure and the code window will display the code for the procedure. The browser window should look something like figure 5.7, and the code window like figure 5.6.

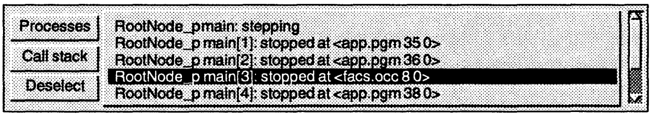

Figure 5.7 facs procedure browser display

5.3.3 Placing a breakpoint

Step 12. You are now going to place a breakpoint in the **facs** thread on line 37, which is the output on channel `to.square`. Use the code window scroll bar to scroll down to line 37. Select line 37 by clicking on the statement. The selection line marker (⊕ on Suns or ◇ on PCs) appears alongside the statement. Click on the blank space after the statement to ensure that no text is highlighted.

Step 13. Click on the **Break** button to set a breakpoint at the selected line. A breakpoint marker (⊘ on Suns or △ on PCs) appears alongside the selected line marker.

```
25      :
26
27      INT n:
28      BOOL going:
29
30      SEQ
31        going := TRUE
32
33        WHILE going
34          SEQ
35            from.feed ? CASE
36              data; n
37                to.squ re ! data; factorial(n)
38              next     -- start a new sequence
39                to.square ! next
40              end      -- terminate
41                SEQ
42                  going := FALSE
43                  to.square ! end
44     :
```

Figure 5.8 Source code after the breakpoint has been set

Notice that the output window at the bottom of the debug screen displays a message telling you about the breakpoint you have just set:

```
breakpoint 2 at <facs.occ 37 0> iptr #8000... of RootNode_p main[3]
```

This means the breakpoint has been labelled event number 2. It has been set at the start of line 37 of the file `facs.occ`. The part in chevrons, '`<facs.occ 37 0>`', means the zeroth step of line 37 of the file `facs.occ`. The breakpoint has been set in the code at address #8000.... If you scroll the output window up one line, you will find:

```
1 4 > break <facs.occ 37 0>
```

A breakpoint has been set in process 1, thread 4, which is `main[3]`. The breakpoint applies only to the currently selected thread, `main[3]`, and will cause execution of the thread to stop when this line is about to be executed. If other threads were using the same code they would not be affected.

A breakpoint set at process level would act on all the threads in the currently selected process. It would give a message like:

```
1 0 > break <facs.occ 37 0>
```

The '`1 0 >`' would mean all threads of process 1.

5.3.4 Starting the example program

Step 14. Having set a breakpoint you are now ready to start the example program. If you were to click on the **Continue** button at this level you would only start the `facs` thread running. To ensure that all the threads are set running you must use the browser to return to the program level.

Move the cursor to the browser window and click on the **Processes** button to display the list of processes. The browser window changes to look like figure 5.9. This is the top level of the browser, the program level.

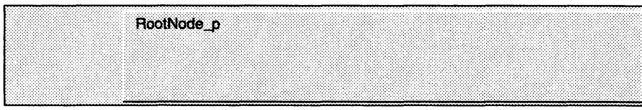

Figure 5.9 Browser window showing the example program's processes

Step 15. At this level you can start all the processes by clicking on the **Continue** button.

Notice that the output window displays the message:

```
0 0 > continue
```

The '`0 0 >`' means all threads of all processes. A non-zero number would refer to a specific process or thread.

Step 16. As the `app` program runs it displays the following message in the program window:

```
Sum of the first n (n < 9) squares of factorials
Please type n :
```

Move the cursor to the program window and type '`4`' followed by return.

The output window should now display the message:

```
1 4 RootNode_p main[3] : breakpoint 2 at <facs.occ 37 0>
```

This tells you that a breakpoint has occurred in `facs.occ` at line 37 for thread 4 of process 1. This should come as no surprise as it is where you set the breakpoint in step 13.

5.3.5 Locating where a breakpoint has occurred

Step 17. To display the code where the breakpoint has occurred, click on the **Last Event** button. This brings you to frame level, as in figures 5.10 and 5.11.

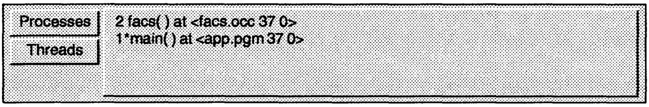

Figure 5.10 Browse window after the breakpoint has been located

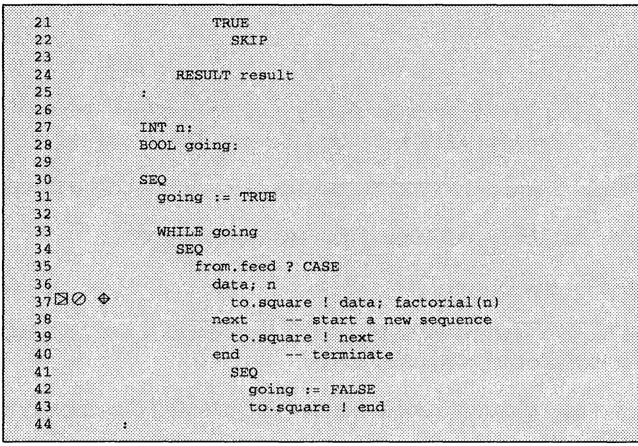

Figure 5.11 Code window after the breakpoint has been located

5.3.6 Removing a breakpoint

Step 18. The breakpoint has served its purpose, so you can now delete it. Check that line 37 is still the current line and click on the **Delete** button. The breakpoint marker disappears.

5.3.7 Examining a variable

Step 19. You are now going to examine the value of a variable.

Click twice on the variable '`n`' on line 27, 36 or 37 to select it. Clicking twice selects the whole of the word where the cursor is.

Step 20. Display the value of the variable by clicking on the **Print** button. The **Print** button displays the value in the output window; it does not produce a hard copy.

The output window should now display the simple message:

```
1
```

This tells you that `n` has the value 1, so the `factorial` function is about to compute the factorial of 1.

5.3.8 Single stepping through the source code

Step 21. Since this thread is stopped you can step through the code and follow the execution path.

Line 37, the current location, includes a call to the `factorial` function. When you **Step** on a call to a function or procedure you will step into the function or procedure.

The debugger treats line 37 as three steps:

- to send the tag `data`,
- to call the `factorial` function and
- to send the result of `factorial`.

Click on the **Step** button to send the `data` tag. **Step** again to step into the function. The code display changes and the current location marker moves to line 12, the beginning of the definition of `factorial`, as shown in figure 5.12.

```
1
2        -- facs.occ
3
4        -- generate factorials
5
6        #INCLUDE "pipe.inc"
7
8        PROC facs (CHAN OF PIPE from.feed, to.square)
9
10         -- compute factorial
11
12         INT FUNCTION factorial (VAL INT n)
13           INT result:
14           VALOF
15             SEQ
16               result := 1
17               IF
18                 n > 0
19                   SEQ i = 1 FOR n
20                     result := result * i
21                 TRUE
22                   SKIP
23
24             RESULT result
```

Figure 5.12 Single stepping through the `factorial` function

5.3.9 Examining the call stack

Step 22. You can find out which line has called the `factorial` function by examining the call stack. You do this using the browser window to go to frame level.

Normally you would move the cursor to the browser window and click on the **Call Stack** button, but **Last Event** left the browser at frame level, so there is no need to do this. At frame level the browser window displays the call stack, which is a list of function and procedure calls, with the most recent call at the top, as in figure 5.13.

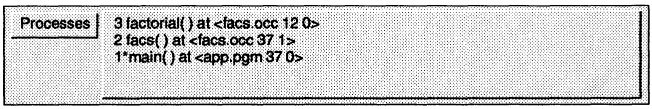

Figure 5.13 Call stack for the `factorial` function

Step 23. You can display the source code of any of the frames to see where the call was made to the next frame in the stack.

Move the cursor to the browser window and click on '`2 facs() at <facs.occ 37 1>`'. Observe that the code window changes to display line 37 of the code of the procedure `facs`, with the output to `to.square` (including the call to `factorial`) marked as the current line, as in figures 5.14 and 5.15.

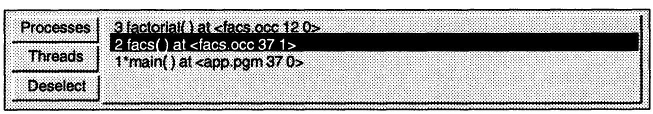

Figure 5.14 Frame `facs()` selected

```
27        INT n:
28        BOOL going:
29
30        SEQ
31          going := TRUE
32
33          WHILE going
34            SEQ
35              from.feed ? CASE
36                data; n
37                  to.square ! data; factorial(n)
38                next     -- start a new sequence
39                  to.square ! next
40                end      -- terminate
41                  SEQ
42                    going := FALSE
43                    to.square ! end
44    :
```

Figure 5.15 Frame `facs()` code

Click on '`1*main() at <app.pgm 37 0>`'. The code window changes again to display the configuration code with the current line marker on the call to `facs`. Click on '`3 factorial() at <facs.occ 12 0>`' to return to the `factorial` function.

5.3.10 Stepping to

Step 24. You may want to continue execution up to a particular statement. This can be done by selecting the statement that you want and using the **Step To** button. This has the same effect as setting a new breakpoint, clicking **Continue** and then removing the breakpoint with **Delete**.

Select line 20 by clicking on the statement on line 20. Now click on the **Step To** button to continue execution until line 20 is reached. The current line marker moves to line 20.

5.3.11 Stepping out of function calls

Step 25. You can step out of this call of `factorial` using the **Step Out** button. This has the effect of continuing until a function or procedure returns or, with a frame selected, **Step Out** will continue until it reaches the current frame. You could of course continue to single step using the **Step** button, but this would be tedious and take longer.

Click on the frame '`2 facs() at <facs.occ 37 1>`' and then click on the **Step Out** button. The current location marker stays on line 37 but the call stack changes. This is because the function has returned but the result still has to be sent down the channel `to.square`.

```
21            TRUE
22              SKIP
23
24        RESULT result
25    :
26
27    INT n:
28    BOOL going:
29
30    SEQ
31      going := TRUE
32
33      WHILE going
34        SEQ
35          from.feed ? CASE
36            data; n
37              to.square ! data; factorial(n)
38            next     -- start a new sequence
39              to.square ! next
40            end      -- terminate
41              SEQ
42                going := FALSE
43                to.square ! end
44  :
```

Figure 5.16 After stepping out

5.3.12 Stepping over function calls

Step 26. If you step using the **Step** operation you will step into any functions or procedures that are called. If you do not want to step into function and procedure calls, then you can use the **Next** operation instead. This has the same effect as **Step** except that function and procedure calls are stepped over, a little like using a combination of **Step** and **Step Out**.

Click the **Next** button until the current line marker moves back to line 37. Click the **Next** button twice more. The first **Next** sends the `data` tag and the second steps over the call to `factorial` and sends the result.

5.3.13 Interrupting running threads

Step 27. The next stage of the example session is to stop all the threads by interrupting them. First you must set the `facs` thread running again.

Move up to thread level by clicking on the browser **Threads** button. Set the `facs` thread running by clicking on the **Continue** button. This should mean all the threads are running except the top level thread which is stepping. The program will wait at the next keyboard input in the state shown in figure 5.17.

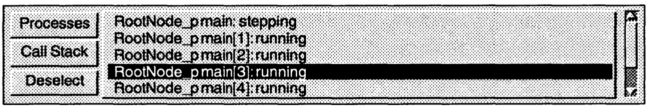

Figure 5.17 The process with all threads running

Step 28. To stop a process or thread running you use the **Interrupt** operation.

Move up to the process level by clicking on the **Deselect** button. Click on the **Interrupt** button to interrupt all the threads. The threads `main[1]` to `main[5]` should all have changed in the browser window from 'running' to 'chan-waiting'. An interrupt breakpoint has been set but 'chan-waiting' means the thread has not reached it because the thread is waiting for channel communications. The output window shows:

```
1 0 > interrupt
```

which means that interrupts have been set on all the threads of process 1.

Step 29. Move the cursor to the program window and type '`6`' followed by return. No echo appears since the echoing thread has been interrupted. The state of the `control` thread, `main[1]`, should now be something like '`stopped at #8000...`'. The other threads are still 'chan-waiting' because they are waiting for channel communications. The output window says:

```
1 2   RootNode_p main[1]: interrupted at #8000...
```

This shows that `main[1]`, the `control` thread, has been interrupted.

Step 30. You can now look at where the program has stopped.

Click on **Last Event** to show where `control` was interrupted. The browser window shows that the interrupt was in the `VIRTUAL.IN()` routine in the `virtual.lib` library module. This library and `hostio.lib` do not have symbolic debugging data so the code cannot be displayed for the frames `7 VIRTUAL.IN()`, `6 sp.getkey`, `5 so.read.echo.line` and `4 so.read.echo.int`. All the debugger can tell us about these routines is which library they are in, although we could look at the disassembled code if we needed to. The debugger therefore shows us the code for the frame which called the code with no debugging data.

```
3 read.next.n() at <control.occ 23 0>
```

We know this frame has full debugging data because the part '`<control 23 0>`' means the debugger has identified the source line as line 23 in the source file `control.occ`. The current line marker shows that it is calling `so.read.echo.int`, which is what we expected from the call stack.

Step 31. Now click on the **Threads** button to return to threads level and set `control` running again by clicking on **Continue**. The browser window should show that `control` is now 'running'. The **Continue** only applies to the `control` thread because that thread is selected in the browser window. This allows `control` to complete its channel communication with `feed`; `feed` then hits its interrupt and becomes '`stopped at #8000...`'.

Step 32. Click on the **Last Event** button to find where this interrupt occurred. The code window shows that it was in `feed` on line 19.

5.3.14 Watching communication between two threads

Step 33. You are now going to watch the communication between `feed` and `facs`. We need to view the sources of `feed` and `facs` at the same time. The debugger provides an **Open Window** operation to allow you open another window to do this.

Click on the **File** menu at the top of the debug display.

Step 34. Select the **Open Window** option from the **File** menu. This opens another debugging display in the same state. On a Sun a complete duplicate display is generated, as in figure 5.18.

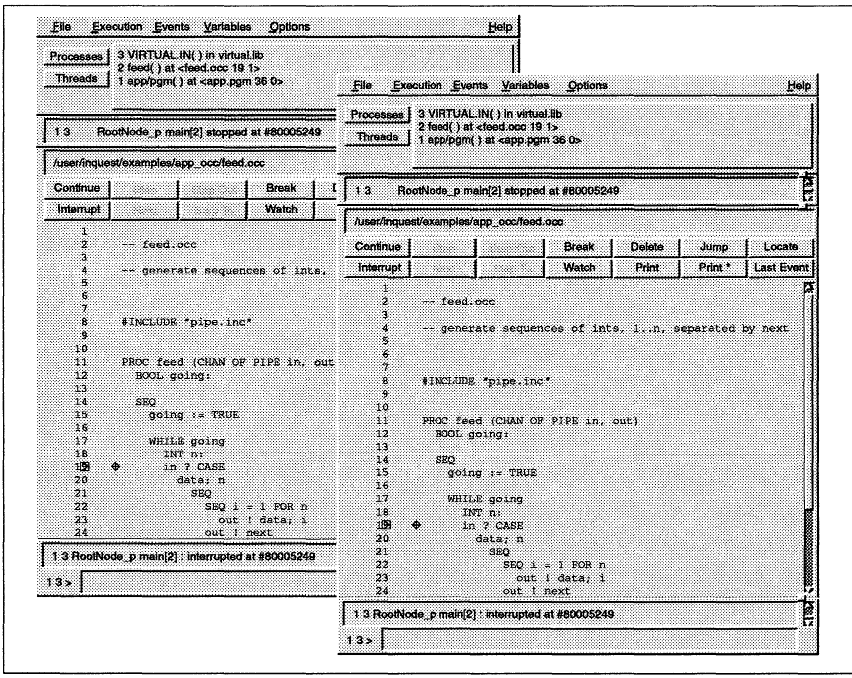

Figure 5.18 Two X-Windows debugging displays open

On the PC, only an extra code window is generated. The buttons and other windows apply to the code window that is selected. The **Window** menu provides operations to tile the code windows, pile them up or select one.

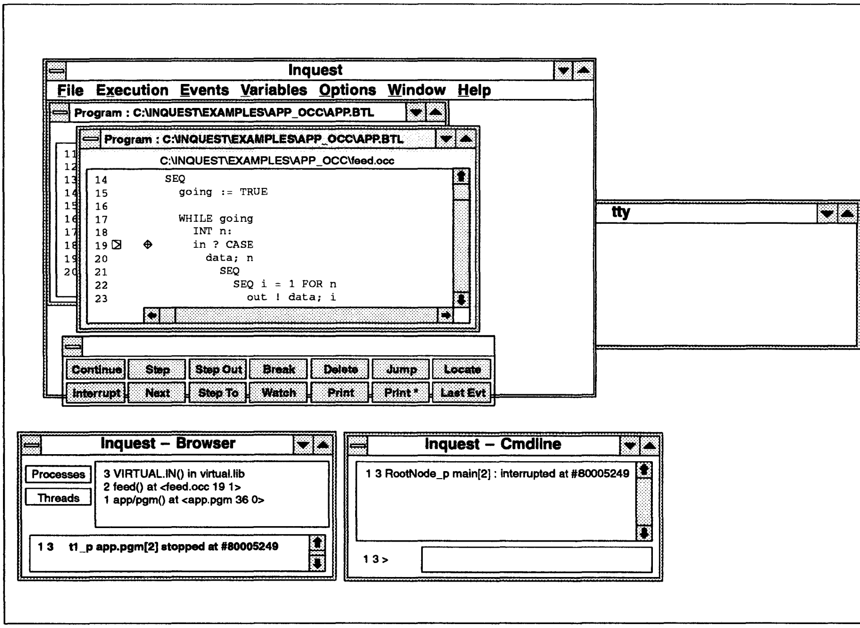

Figure 5.19 Two PC code windows open

Make sure that the second display is positioned so that you can see enough of both code windows.

Step 35. Move the cursor to the second debug window. We want to display the thread `facs`, which should be waiting to read input. Go up to thread level by clicking on the **Threads** button and select '`RootNode_p main[3] chan-waiting`'. It is, of course, waiting an input from `feed`. We will call this window the `facs` window.

Step 36. Move the cursor to the first debug window. We will call this the `feed` window. You can tell which window is which from the file name in the attribute window or the frame stack in the browser window.

Step 37. In the `feed` window, select line 23 by clicking on it. This is the line containing the output statement that sends to `facs`. Continue execution of `feed` up to line 23 by clicking on the **Step To** button. In the attribute window, `feed` is now:

```
1 3 RootNode_p main[2]: stopped at <feed.occ 23 0>
```

`facs` is waiting for input, so in the attribute window of the `facs` window we have:

```
1 4 RootNode_p main[3]: chan-waiting
```

Step 38. In the `feed` window, click on **Step** to send the first part of the output, the tag `data`. `feed` is now on the second step of the statement, so the attribute window now says:

```
1 3 RootNode_p main[2]: stopped at <feed.occ 23 1>
```

Step 39. Executing this step has also had an effect on `facs`. The `facs` thread has now read the tag and hit its interrupt. This can been seen from the `facs` attribute window which has changed to:

```
1 4 RootNode_p main[3]: stopped at #8000...
```

In the `facs` window, click on **Last Event** to see the interrupt. Click on **Step** to complete the call of `VIRTUAL.IN`, which disappears from the call stack and the attribute window changes to:

```
1 4 RootNode_p main[3]: stopped at <facs.occ 35 2>
```

Step 40. Click on **Step** again to test the tag to see whether the communication is `data`, `next` or `end`. The attribute window becomes:

```
1 4 RootNode_p main[3]: stopped at <facs.occ 36 0>
```

Step 41. **Step** again to read in the second part of the data. The thread cannot complete the input and so it has to wait for the sender again before the step can complete. The attribute window says:

```
1 4 RootNode_p main[3]: stepping
```

Step 42. Change to the `feed` window, which is still stopped at `<feed.occ 23 1>` and click on **Step**. This sends the second part of the data and completes the communication. The line marker moves on to line 22 for the next iteration of the loop. The attribute window changes to:

```
1 3 RootNode_p main[2]: stopped at <feed.occ 22 0>
```

Meanwhile, in the `facs` window, the input has completed and the line marker has moved to line 37. The attribute window has changed to:

```
1 4 RootNode_p main[3]: stopped at <facs.occ 37 0>
```

Step 43. You can now close the `facs` window by clicking on the **File** menu at the top of the `facs` window and selecting the **Close Window** option from the **File** menu. The window disappears.

5.3.15 Setting a watchpoint on a variable

Step 44. You can set a watchpoint on a variable so that the thread stops every time the variable is about to be changed. You are now going to set a watchpoint on the `n` variable that holds the input data.

Highlight the `n` variable on line 18, 20 or 22 of `feed` by double clicking on it.

Step 45. Set a watchpoint on the highlighted variable by clicking on the **Watch** button. Notice that a watch symbol, ◷ on Suns or on PCs, appears next to the declaration of `n` and the following messages appear in the output window:

```
1 3 > watch n -1 <feed.occ 18>;
watchpoint 6 on <feed.occ 18 n> address #8000.. to ...
```

Scroll the window up to see the first line. When the watchpoint occurs the thread will stop.

Step 46. You now need to set the program running.

Move the cursor to the browser window and click on the **Processes** button.

Step 47. Click on the **Continue** button to set all the threads running. If you had not clicked on **Processes** before doing a **Continue** you would only have set thread `feed` running. The output window displays the message:

```
0 0 > continue
```

Step 48. The program window displays the result:

```
The result was : 533417
```

Type '`3`' at the prompt '`Please type n :`'

When the program hits the watchpoint, the output window displays the message:

```
1 3 > RootNode_p main[2] : watchpoint 6 at #8000...
```

Step 49. You can now examine the value of the variable with the watchpoint.

Click on **Last Event** to show the code where the watchpoint occurred. Select the variable `n` by double clicking on it and then click on the **Print** button. It will display the value 6, as the value has not changed. **Step** once and **Print** `n` again and it will have changed to 3.

Step 50. Click on the **Threads** button to go up to process level.

Step 51. We can look at the list of current watchpoints by means of the **List Watchpoints** operation. Click on the **Events** menu and select **List Watchpoints**. A window pops up with two watchpoints, both watchpoint 6.

One is set on a thread, so that if another instantiation of `n` were defined then the watchpoint would apply to that `n` too. The other is set on a range of addresses and is the watchpoint on the existing variable `n` in `feed`. We could select a watchpoint by clicking on it to enable, disable or delete it.

Step 52. To close the **List Watchpoints** window, click on **Cancel**.

Step 53. If you want to trace the value of the variable as the thread runs you can make the debugger do an automatic **Print** and **Continue** every time the watchpoint occurs by typing a suitable command in the command window.

Move the cursor to the command window at the bottom of the debug screen and click the left mouse button to make the window active.

Step 54. Type in the following command, followed by a carriage return:

```
when(6) {w=step; wait(w); fid=2; write n is (print n); continue}
```

This tells the debugger to wait until event 6 occurs and then

- step one more instruction to update the variable `n`,
- wait for the step to complete,
- select frame 2,
- output the text '`n is`' together with the value of `n`, and then
- continue execution.

You may remember that event 6 is the watchpoint you set a moment ago. The watchpoint occurs before the variable is changed, so we step past the assignment so that the value of `n` printed will be the new one.

Step 55. If you are using a Sun, then the output window is only one line deep. You can change the height of this window so that you can see a history of the most recent messages. You do this by dragging the sizing box.

Move the cursor to the output window sizing box, which is the small box at the top of the scroll bar, as shown in figure 5.20. Notice that the cursor changes to a cross when it is on the box. Click and hold the mouse button down whilst you drag the box up the screen. Keep dragging until you have an output window four lines deep.

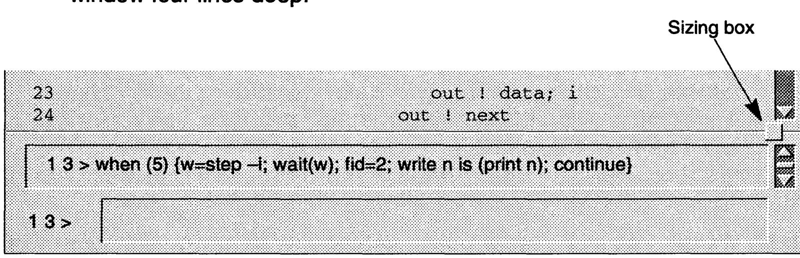

Figure 5.20 The Sun output window before re-sizing

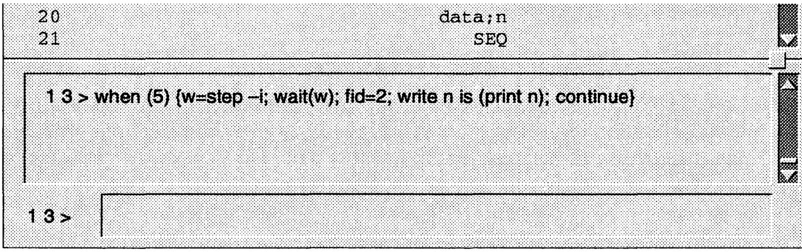

Figure 5.21 The enlarged Sun output window

Step 56. Click on the **Continue** button.

Step 57. Type in another number at the prompt '`Please type n :`'. The new value of `n` is displayed in the output window and the thread continues execution.

5.3.16 Deleting a watchpoint

Step 58. You can now delete the watchpoint you set on **`n`**.

To set a watchpoint we selected a variable. To delete the watchpoint we select the declaration line. Select line 18 of **`feed.occ`** by clicking on it. This line is the declaration of **`n`**, which has the watchpoint symbol on it, ◷ on Suns or ⌚ on PCs.

```
11        PROC feed (CHAN OF PIPE in, out)
12          BOOL going:
13
14          SEQ
15            going := TRUE
16
17            WHILE going
18              INT n:
19              in ? CASE
20        data; n
21          SEQ
22            SEQ i = 1 FOR n
23              out ! data; i
24            out ! next
25
26        end
27          SEQ
```

Figure 5.22 Line 18 with watchpoint set

Step 59. With the declaration line selected, click on the **Delete** button to remove the watchpoint. The watchpoint symbol disappears

Step 60. Type in another number at the prompt '**`Please type n :`**'.

Notice that the program now executes normally since the watchpoint has been removed.

5.3.17 Jumping down a channel

To explore the wider context of a thread we may wish to look at the threads that communicate with it. To find a thread that is waiting for a communication, we can jump down a channel. This means that we are changing context to the thread that is waiting for communication on that channel.

At most one thread can ever be waiting for one channel; when two threads are ready then communication starts. If no thread is waiting then the **Jump** operation will not change the context. If the current thread is waiting then, again, the **Jump** operation will not change the context.

Step 61. Thread `feed` has a channel output statement at line 23. You are now going to use this statement to jump to the thread that is waiting on the channel. First you must interrupt execution of thread 2.

Step 62. Stop the `feed` thread by clicking on the **Interrupt** button. Enter another value for `n` so that the program runs on to the interrupt. Then **Step** so that the thread comes out of the `VIRTUAL.IN` routine to code that has full debugging.

Step 63. Move the cursor to line 23 and select the channel name '`out`' by double clicking on it.

Step 64. Jump to the thread waiting on this channel by clicking on the **Jump** button. Notice that the source code for `facs` is now displayed in the code window with the current line marker resting on the input statement to read from the channel `from.feed`, which is the other end of the `out` channel in `feed`.

5.3.18 Leaving the debugger

Step 65. You have now finished the example interactive debugging session. If you want to exit from the debugger, click on the **File** menu and select the **Exit** or **Quit** option. A pop-up dialog box will appear asking you to confirm the exit request. Select **Yes** and the debugger will close its windows and exit.

On PC systems, you may wish to also close the `ISERV` window.

This concludes the tutorial for interactive debugging with occam programs.

6 An example occam post-mortem debugging session

This chapter takes you in detail, step by step, through one example post-mortem debugging session, to demonstrate the basic features that are available for post-mortem debugging. This chapter uses an occam program as the example. A similar ANSI C example is described in chapter 3. In particular this chapter shows you how to:

- build the program for post-mortem debugging;
- start the debugger when the program fails;
- locate where the crash has occurred and the reason for it;
- examine a variable;
- examine a call stack;
- jump down a channel;
- quit from the debugger.

Before starting the session you need to know a little about the example program. This is the same program as is used in chapter 5, so if you have not worked through the interactive debugging session, read section 5.1 before proceeding.

Before starting the tutorial you may find it useful to have a listing of the example program source code. It would also be useful to look at figure 2.1 or figure 2.2 in section 2.4 to show you the names of the parts of the debugger display.

6.1 Post-mortem debugging

Post-mortem debugging means debugging after the program has crashed, terminated or been stopped. It may have crashed during normal running or during an interactive debugging session. In this tutorial, we shall start from normal execution. Starting from an interactive session is easier and is described in section 2.4 of chapter 2 in the *INQUEST User and Reference Manual.*

During post-mortem debugging you can navigate through the code and explore the state of the program exactly as in interactive debugging. However, the program cannot be restarted, so stepping, interrupts, breakpoints and watchpoints do not apply. The debugger just allows you to inspect the state of the program when it was halted.

6.1.1 Building the code

Post-mortem debugging only requires that the code has been compiled with full debugging data, i.e. without the D option.

On Sun systems, the example program makefile builds the example application for you and is called `makefile`. On a PC, a batch file called `build.bat` is provided to do this. This generates a bootable file for post-mortem debugging called `app_pm.btl`.

The contents of the configuration file are shown in figure 3.2.

For the purposes of this tutorial, all the processes are configured to run on the same processor.

6.1.2 Starting post-mortem debugging

When a program crashes during normal execution then the host is returned to the state before the program was run. Post-mortem debugging can then be started by the `inquest` command.

On a PC this can be done from the File Manager by double clicking on the `inquest.exe` program, which makes a command line box pop up. The browse button is used to select the application `app_pm.btl`. The `-pm` option is added in the **Options** box to request post-mortem debugging.

On a Sun this is done by typing the inquest command at the operating system prompt with the `-pm` option to request post-mortem debugging:

```
inquest app_pm.btl -pm
```

The debugger will *analyze* the transputer network. This means that signals will be sent to the network to halt any threads that may be still running, and then the network will be reset without losing any state information. After analyzing the network, the debugger uses a network mapper to explore the state of the network. The debugging display will appear on the screen.

If the transputer error flag was set, then a message will appear showing where the error occurred. The source line where the error happened is automatically located by the debugger. The process causing the error can then be explored and other processes located using **Jump**.

6.2 Step-by-step tutorial

Here is a step-by-step tutorial to guide you through the main features of INQUEST post-mortem debugging.

Step 1. Move to the `app_occ` sub-directory in the `examples` directory which contains the occam example program `app`. Run the set-up script to set up the environment, as follows.

On Sun systems, at a prompt type:

```
source setup.csh
```

or

```
. setup.sh
```

On PC systems, at a DOS prompt type:

```
setinq
```

Step 2. Check that the `TRANSPUTER` parameter is correctly defined.

Step 3. The root transputer must be an IMS T400, IMS T425, IMS T801, IMS T805, ST20 (T450) or IMS T9000. The example is configured for a single ST20 or for a single IMS T9000 depending on the INQUEST version.

If you are using a different type of root transputer then you will need to edit:

- the hardware configuration file, `hardocc`;
- for Suns the `make` macro file `tools` or for PCs the build batch file `build.bat`.

The changes to make in each case are given in table 6.1.

Target	All users: file `hardocc`	Sun users only: file `tools`	PC users only: file `build.bat`
ST20 / T450	No change	No change	No change
IMS T9000	No change	No change	No change
IMS T400 or T425	Change `T450` to `T425`	Change `450` to `425`	Change `t450` to `t425`
IMS T801 or T805	Change `T450` to `T805`	Change `450` to `805`	Change `t450` to `t805`

Table 6.1 Changes to examples to support different targets

Step 4. Create a bootable file, suitable for debugging. To do this on a Sun, at an operating system prompt type:

```
make
```

On a PC, type:

```
build
```

6.2.1 Starting the debugger

Step 5. Run the example program, `app_pm.btl`, which is the bootable code for post-mortem debugging you created in Step 4.

On PC systems, start up Windows, open the File Manager and double click on the `inquest.exe` program. This will open the **Command line** dialog box. Use the browse button next to the **File** field to find and select the application `app_pm.btl` in the `app_occ` sub-directory in the `examples` directory. Click on the `Run` button in the **Command line** box.

Since the code has not been configured for debugging, this loads the application for normal execution onto the target hardware. The parameters you can give to `irun` are described fully in the `irun` chapter of your *Toolset Reference Manual.*

On Sun systems type:

```
irun app_pm.btl
```

Since the code has not been configured for debugging, this loads the program for normal execution onto the transputer.

The parameters you can give to `irun` are described fully in your host interface software or network interface software user manual.

Step 6. The program will ask for a number with the prompt:

```
Sum of the first n (n < 9) squares of factorials
Please type n :
```

Enter the number 10 to make the arithmetic overflow and the program crash. The host server detects the error flag and displays these messages:

```
Transputer error flag set

Error - iserv - Transputer error flag set
```

Step 7. On PC systems, in the File Manager, double click on the `inquest.exe` program. This will open the **Command line** box again. Use the browse button again next to the **File** field to find and select the application `app_pm.btl` in the `app_occ` sub-directory in the `examples` directory. This time type `-pm` in the **Options** field to tell INQUEST to do a post-mortem debug. Click on the `Run` button in the **Command line** box.

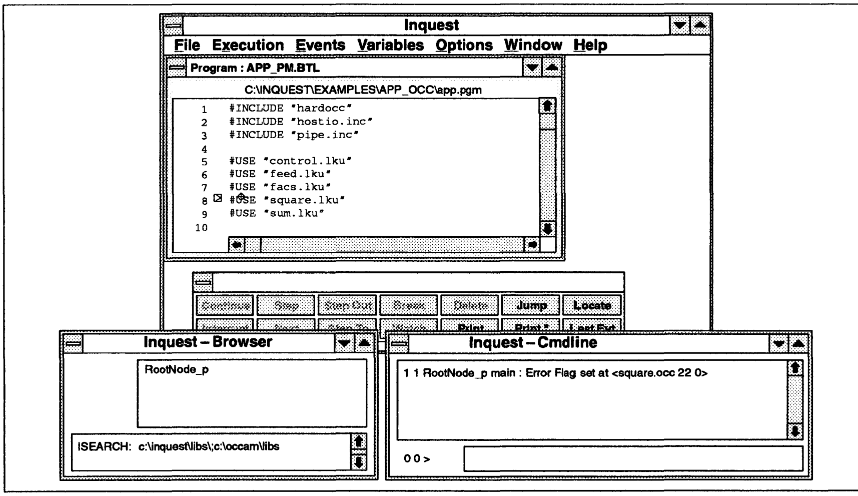

Figure 6.1 The Microsoft Windows start-up display for the example program

On Sun systems, start the post-mortem debugger with the command:

```
inquest app_pm.btl -pm
```

Step 8. Wait while a debugging display is created. The debugger has explored the network and is showing the state of the program when it was stopped.

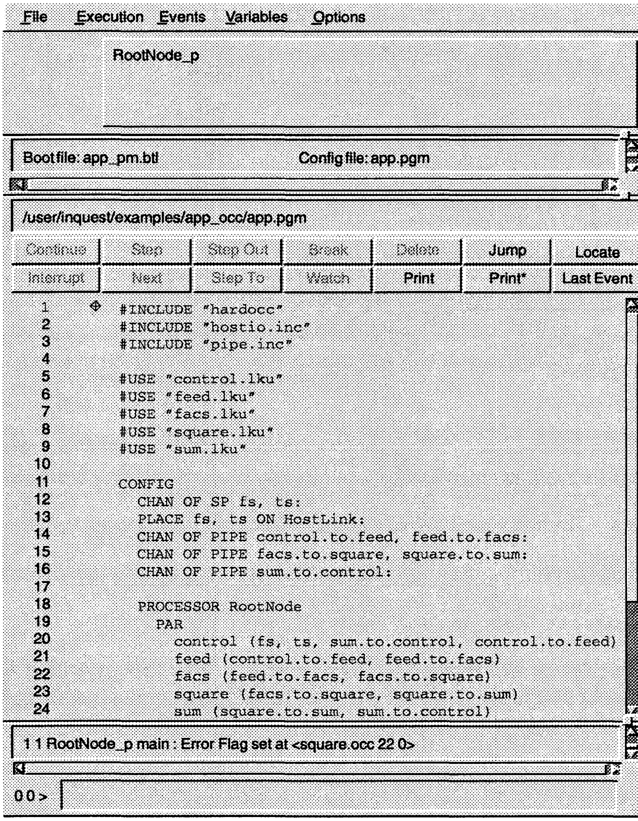

Figure 6.2 The X-Windows initial display for the example program

Step 9. To show where the program set the error, click on the **Last Event** operation button. Figure 6.2 shows the X-Windows display; the PC display is shown in figure 6.1. Source code is shown in the code window with the next statement marker, ⊳ on Suns or ⊳ on PCs, next to the line that crashed, which should be the line:

```
square := n * n
```

If the program had not crashed, it could have been halted by pressing Control-C. Control-C only stops the host server, so the transputer program will generally be waiting at the next point where it interacts with the host. Running the post-mortem debugger will kill the program.

6.2.2 Examining a variable

Step 10. You are now going to examine the value of a variable.

Click twice on the variable 'n' on line 17, 20 or 22 to select it. Clicking twice selects the whole of the word where the cursor is.

Step 11. Display the value of the variable by clicking on the **Print** button. The **Print** button displays the value in the output window; it does not produce a hard copy.

The output window should now display the simple message:

```
362880
```

This tells you that n has the value 362880, and the error flag was set because the square of this number is too big to fit in a 32-bit integer.

6.2.3 Examining the call stack

Step 12. You can find out which line has called the `factorial` function by examining the call stack. You do this using the browser window to go to frame level.

Normally you would move the cursor to the browser window and click on the **Call Stack** button, but the debugger has automatically started at frame level, so there is no need to do this. At frame level the browser window displays the call stack, which is a list of function and procedure calls, with the most recent call at the top, as in figure 6.3.

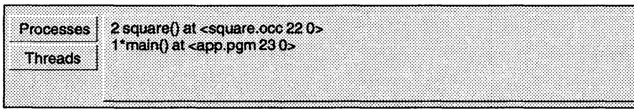

Figure 6.3 Call stack for the initial display

The top line shows that in this thread the function `square` has halted at line 22 in the source code file `square.occ`. The line below shows that `square` was called by `main` at line 23 of the file `app.pgm`. The asterisk (`*`) means that `main` started the thread.

Step 13. You can display the source code of any of the frames to see where the call was made to the next frame up the stack.

Move the cursor to the browser window and click on '`1*main() at <app.pgm 23 0>`'. In the browser window, the selected frame is highlighted, as in figure 6.4. The code window has changed to display line 23 of the configuration code, with the call of `square` marked as the current line with the next statement marker, ⊳ on Suns or ▷ on PCs, as in figure 6.5.

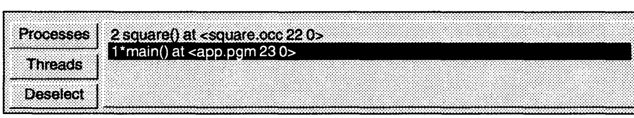

Figure 6.4 Frame `main()` selected

```
#INCLUDE "hardocc"
#INCLUDE "hostio.inc"
#INCLUDE "pipe.inc"

#USE "control.lku"
#USE "feed.lku"
#USE "facs.lku"
#USE "square.lku"
#USE "sum.lku"

CONFIG
  CHAN OF SP fs, ts:
  PLACE fs, ts ON hostlink:
  CHAN OF PIPE control.to.feed, feed.to.facs:
  CHAN OF PIPE facs.to.square, square.to.sum:
  CHAN OF PIPE sum.to.control:

  PROCESSOR RootNode
    PAR
      control (fs, ts, sum.to.control, control.to.feed)
      feed (control.to.feed, feed.to.facs)
      facs (feed.to.facs, facs.to.square)
      square (facs.to.square, square.to.sum)
      sum (square.to.sum, sum.to.control)
```

Figure 6.5 Frame `main()` code

Moving up and down the stack, examining the variables at each level, gives you a clear view of what happened to the thread just before it crashed.

6.2.4 Using the browser

Step 14. We can look at other threads by using the browser. Move to the top, or program, level by clicking on the **Processes** button in the browser window. Tell the debugger to search for the threads of the program by pulling down the **Execution** menu and clicking on **Find Threads**.

The browser shows a list of one process, `RootNode_p`. Select the process `RootNode_p` by clicking on it. This moves the display to process level, with the browser window showing a list of threads in the process with their states, as shown in figure 6.6.

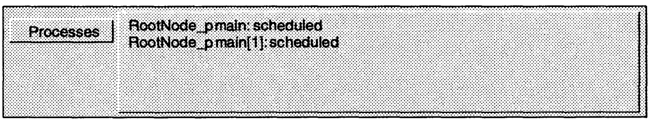

Figure 6.6 Process level browser display

The first thread, shown as `RootNode_p main`, is `square`, which set the error flag. The other listed thread, shown as `RootNode_p main[1]`, is `facs`, which is waiting for processor time. The other threads are all waiting for channels, so the debugger cannot find them except by jumping down the channels.

Step 15. To move to the facs thread, double click on '`RootNode_p main[1]: scheduled`'. The display changes to the frame level display, as shown in figure 6.7. The selected thread is highlighted and the code window shows the source code for the facs thread, with the next statement marker (⊠ on Suns or ⊳ on PCs) on the line that was waiting to complete, as shown in figure 6.8.

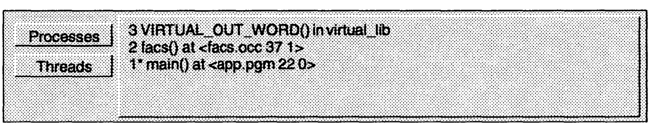

Figure 6.7 Frame level browser display

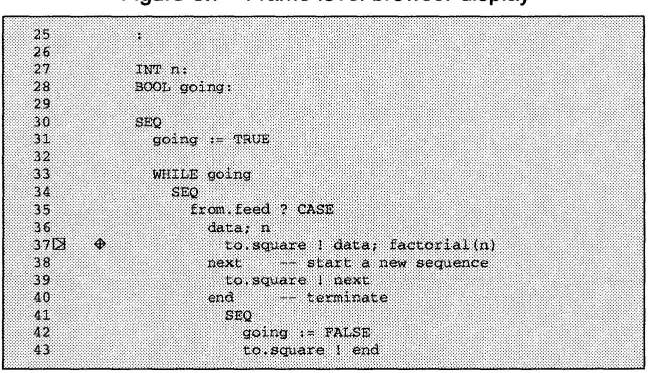

Figure 6.8 The `facs` thread

6.2.5 Jumping down a channel

When the `square` thread crashed, the transputer halted, leaving the other threads queued or waiting for channels or timers. If there were more transputers in the network then they would continue running until they were all waiting for communications with the crashed transputer or they were analyzed by INQUEST.

To explore the wider context of the crashed thread we may wish to look at the threads that communicate with the crashed thread. To find a thread that is waiting for a communication, we can jump down a channel. This means that we are changing the context to the thread that is waiting for communication on that channel.

At most one thread can ever be waiting for one channel; when two threads are ready then communication starts. If no thread is waiting then the **Jump** operation will not change the context.

Step 16. Select the channel `to.square` by clicking twice on a reference to it. Click on the **Jump** operation button to jump down the `to.square` channel. The `to.square` channel is empty, i.e. no thread is waiting for it, so nothing happens, except that the message

```
*empty*
```

appears in the output window.

Step 17. Select the channel `from.feed` by clicking twice on it. Click on the **Jump** operation button to jump down the `out` channel. The context will change to the `feed` thread, which is waiting to send on this channel. The channel is called `out` within the `feed` procedure. The next statement marker shows exactly which statement is waiting to send.

6.2.6 Leaving the debugger

Step 18. You have now finished the example post-mortem debugging session. If you want to exit from the debugger, click on the **File** menu and select the **Exit** or **Quit** option. A pop-up dialog box will appear asking you to confirm the exit request. Select **Yes** and the debugger will close its windows and exit.

This concludes the occam tutorial on post-mortem debugging.

Index

P

S

T

V

W