INQUEST User and Reference Manual

SGS-THOMSON
MICROELECTRONICS

72 TDS 405 05 – 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 405 05

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 performance monitors. This document contains user guides and reference material about the INQUEST tools.

Chapters 2 to 4 describe the INQUEST debugger. Chapter 2 describes the structure of the debugger and how to use the buttons, menus and mouse features. Chapter 3 describes the underlying command language which extends and complements the buttons, menus and mouse features for the advanced user. Scripts can be written in the command language using the programming constructs described in chapter 4.

For a tutorial introduction to the INQUEST debugger, see the *INQUEST Debugger Tutorial.*

Chapter 5 describes the libraries supplied with INQUEST to assist with debugging. Chapter 6 describes the three execution analysis tools. Chapter 7 describes the network analyzer.

1 Introduction

The INQUEST development environment is a set of software tools consisting of a hosted debugger and performance analysis tools.

1.1 Debugging

The debugger consists of a host-based symbolic debugger and target-resident debugging kernels which are added to the application program by the configurer.

The debugger provides the following features:-

- a window-based user interface that displays source code and enables the user to interact with the debugger by means of a mouse, buttons and menus;
- a break point facility that can be used on particular threads of execution;
- a single stepping facility that allows a thread of execution to be single stepped at the source level or at the assembly code level;
- a watch point facility that allows the program to be stopped when selected variables are to be written to or read from;
- a stack trace facility;
- a facility to find the threads of execution of a program and set break points on them;
- a facility to monitor the creation of threads of execution;
- commands to print the values of variables and display memory;
- a simple interpreter to enable structured C and occam variables to be displayed;
- a programmable command language that allows complex break points and watch points to be set and enables debugging scripts to be generated;
- post-mortem debugging on the target or from a dump file.

The mouse, button and menu features are described in chapter 2. The command language commands are described in chapter 3 and the programming language constructs are described in chapter 4. Chapter 5 describes the libraries provided with INQUEST which may be used in an application to aid debugging.

1.2 Execution analysis

The execution analysis tools comprise an execution profiler, a utilization monitor and a test coverage and block profiling tool. With these tools the execution of the user program can be monitored after the user program has completed. The monitoring output is stored locally in the memory of each target processor, so that the profiling tools have little execution overhead on the application. After the program has completed execution, the monitoring data is extracted from the target and processed to provide reports on the program execution.

1.2.1 Execution profiler

The execution profiler tool provides the following information on program execution:-

- the percentage time spent executing each low priority procedure or function per configuration-level process;
- an analysis of time spent in each call of each function or procedure and where it was called from;
- the percentage time spent executing at high priority;
- the percentage idle time of each processor in the network;
- the percentage time spent executing each configuration-level process.

1.2.2 Utilization monitor

The utilization monitor tool provides the following information on program execution:-

- when each target processor was busy and when it was idle over the time of the program execution. This is displayed in the form of an interactive window-based Gantt chart of processor execution against time.

1.2.3 Test coverage and block profiler

The test coverage and block profiling tool provides the following information on program execution:-

- how many times each statement of the application code was executed;
- an analysis of test coverage;
- feedback information for optimization by the compiler;
- accumulation of results from multiple runs.

The execution analysis tools are described in chapter 6.

1.3 Network analysis

The network analyzer is used to determine the configuration of a transputer network and generate part of a configuration description of the hardware. It also resets a transputer network and clears any error flags.

The network analyzer tests the types of transputers in a network and how they are connected together. It recognizes direct connections between transputers and connections via IMS C004 link crossbar switches. When used in this manner `rspy` produces textual output showing the transputer types and their connections.

`rspy` can also be used to generate the hardware description in a form suitable for use in C (`icconf`) and occam (`occonf`) configuration files.

2 Debugging

2.1 Introduction

This chapter describes the INQUEST debugger, which debugs application software running on a target transputer network from a host. It consists of a debugger program and a debugging kernel. It uses a host computer communicating with the target hardware, as shown in figure 2.1. The hardware serial communications link between the host and the target is called the host link.

The debugger can be run either interactively or post-mortem. In both cases, the debugger allows the user to explore the state of the application.

The debugger can be controlled either using a mouse, buttons and menus or using the command language. The command language is described in chapter 3.

2.1.1 Interactive debugging

The debugger runs with the application and controls its execution. The user controls, through the debugger, when the application runs and when it stops.

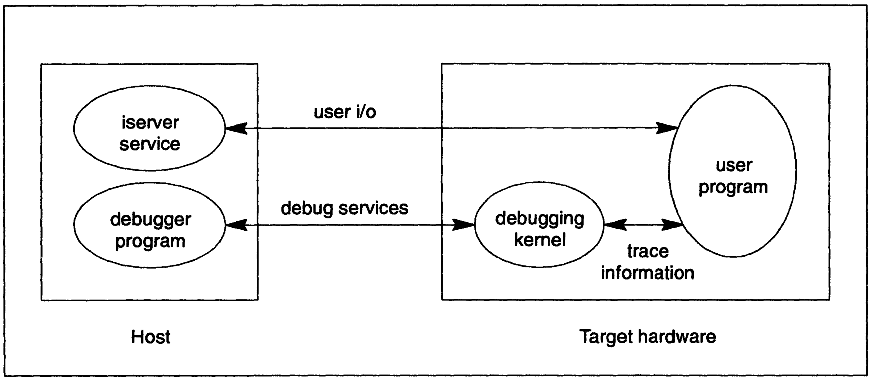


Figure 2.1 Interactive debugging system

The target hardware runs the user program (which is to be debugged) plus a *debugging kernel* which may be distributed over a network. The host runs the host services for the application plus the *debugger program,* which provides the debugger's user interface, displaying the nominal state of the program and passing commands to the debugging kernel as requested by the user.

The host must be able to control and monitor the target hardware. On T2/T4/T8-series transputers, the host must be able to reset and analyze the entire target network. It is preferable that the host can detect errors from all the target processors, so it is not desirable to use a root processor with a subsystem port. On T9000-series transputers, the control network must be connected to the host control port provided by the host interface hardware. If the host cannot detect errors then the full post-mortem capability cannot be used.

The user program may be a parallel program, consisting of many tasks or threads of execution, as described in section 2.5. At any time, some threads of the user program may be running and other threads may be stopped. The debugger keeps a copy of the last stopped state of each thread. The display uses this copy of the state, not the actual state of the target hardware.

A running thread will execute until it is *interrupted*, hits a *breakpoint* or *watchpoint* or the thread terminates. A request from the user to interrupt a running thread will cause a temporary breakpoint to be inserted in the code of that thread which is removed when the breakpoint has been hit. The copy of the state of the thread held by the debugger will then be updated.

Interactive debugging is discussed in more detail in sections 2.2 to 2.3.

2.1.2 Post-mortem debugging

In post-mortem debugging, the debugger is run after the application has terminated or crashed, and will automatically give information about where and why the application halted. INQUEST analyzes the target hardware, copying some target memory to the host, loading a post-mortem kernel and exploring the state of the target.

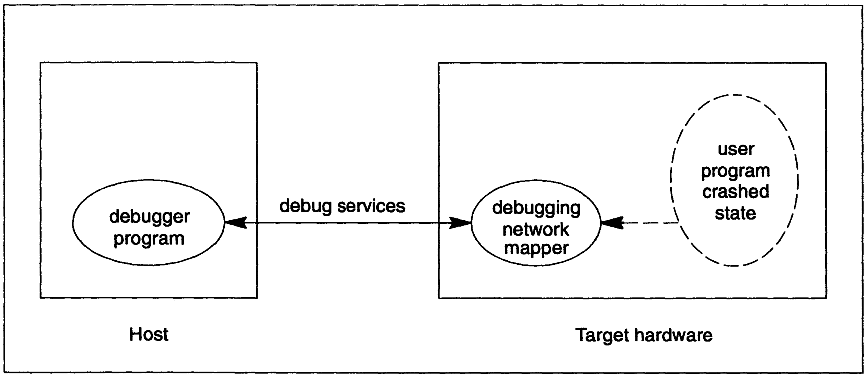


Figure 2.2 Post-mortem debugging system

Post-mortem debugging may be used at the end of an interactive debugging session. The session may end when the processor error flag is set or when an interactive command is timed out. It may also be ended directly by the user. In each of these cases, the debugger offers the user the option of entering a post-mortem debugging session.

Post-mortem debugging may also be used when an application running without the debugger has halted. This may occur when the processor error flag is set or when the program has terminated. The debugger extracts the final state of the application, which can be displayed and explored by the user.

It may also be used after normal execution when the host server is interrupted by the user, for example by the user typing Control-C. In this case only the host server is interrupted, so the application code will continue until it has to wait for communication with the host. Starting the post-mortem debugger will halt all low priority processes at the next deschedule point.

The facilities to explore the program and data state are all available in post-mortem debugging, including inspecting variables, inspecting call stacks, finding threads and jumping down channels. However the application cannot resume execution, so stepping, interrupting, breakpointing and watchpointing do not apply.

Normally, post-mortem debugging is carried out by directly exploring the state of the target hardware after the application has halted. Sometimes it may be preferred to save the post-mortem state of the application in a dump file. The state may then be explored by reading the dump file without using the target.

Post-mortem debugging is discussed in more detail in section 2.4.

2.1.3 Transputer versions

This document describes the behavior of the debugger when used with transputer versions which provide the debugger support instructions, e.g. IMS T225, IMS T400, IMS T425, IMS T426, IMS T801, IMS T805, ST20450 (T450) and IMS T9000. These transputer versions are known as *breakpointing transputers*. Earlier transputers (i.e. IMS T212, IMS M212, IMS T222, IMS T414 and IMS T800), known as *non-breakpointing transputers*, may not be debugged interactively. If code is to be debugged on a mixed network of breakpointing and non-breakpointing transputers then code for the non-breakpointing transputers must be marked in the configuration as not debuggable, as described in section 2.2.

2.2 Preparing a program for interactive debugging

Code which is to be interactively debugged must be compiled for the correct processor, since debugging is performed differently on different processor models. Compiling for the wrong version may cause the wrong debugging kernel to be used which can cause a start-up error. Code for debugging should not be compiled for processor classes.

For example, when using `imakef` to build for an IMS T801 or IMS T805, the configuration code must refer to a `.c9h` or `.t9h` file in the `use` or `#USE` statement. If `imakef` is not being used then code for an IMS T801 or IMS T805 should be compiled with the `T9` or `T805` option.

In order to use the debugger, the code should be prepared as follows:

- ANSI C code for debugging should be compiled with the full debugging data option, `G`, selected on the ANSI C compiler `icc`. occam code should be compiled for debugging using occam compiler `oc`, without the minimal debugging data option, `D`. This instructs the compiler to add symbolic debugging information to the output file. Code compiled with minimal debugging information is capable of providing a stack trace but cannot provide other symbolic information. In particular, source code cannot be displayed in the code window.

- For processes written in ANSI C, link the processes to be debugged with the start-up file `cdebug.lnk` in place of `startup.lnk` and `cdebugrd.lnk` in place of `startrd.lnk`. This links in the debugging run-time libraries. C processes which are not to be debugged should use the normal C run-time start-up files `startup.lnk` and `startrd.lnk`. C code linked with the normal start-up files may cause the debugger to behave strangely unless the process is marked as not debuggable in the configuration file. occam processes should be linked in the normal way. Code for post-mortem debugging only can be linked with the normal libraries.

- Code to be used with the INQUEST debugger must be configured. For interactive debugging, configure with the INQUEST debugging option `GA`, selected on `icconf`, `occonf`, `inconf` or `onconf`, to add in the debugging kernels. The `GD` option places a kernel on each processor being used. The `GA` option places kernel processes only on processors which have processes to be debugged. Processes may be marked as not to be debugged using the `nodebug` configuration attribute.

  Code configured with a debugging option will automatically run under interactive debugging. Code configured without a debugging option cannot be run under interactive debugging.

  When using the `GA` option, the default value for `nodebug` is `FALSE`. Unmarked processes will be debugged and processes not to be debugged must be marked with `nodebug` set to `TRUE`.

  When using the `GD` option, the default value for `nodebug` is `TRUE`. Unmarked processes will not be debugged and processes to be debugged must be marked with `nodebug` set to `FALSE`.

  When using the C configurer, `icconf` or `inconf`, add the following attribute statement to set the `nodebug` attribute to `TRUE`:

  ```
  nodebug = TRUE
  ```

  When using the occam configurer, `occonf`, add the following attribute statement to set the `nodebug` attribute to `TRUE`:

  ```
  set(nodebug := TRUE)
  ```

  Any processes to be run on non-breakpointing transputers (e.g. IMS T222, IMS T414 and IMS T800) must be marked as not debuggable by setting the `nodebug` attribute to `TRUE`.

- Collect using `icollect`.

If `imakef` is used to build code for interactive debugging, the makefile generated must be altered as follows:

- The `CONFOPT` macro should include the `GA` option.

2.3 Running interactive debugging

The debugger is run using the command `inquest`, which loads the application onto the target hardware using `irun`, which then acts as a host server for the application. For details of the `irun` command line and the associated parameters, see the *Toolset Reference Manual*.

On Microsoft Windows systems, `inquest` is run in the same way as other applications. For example, an `inquest` icon can be double clicked in the `inquest` program group in the Program Manager. The command line associated with an icon can be changed by selecting the icon and using **Properties...** in the **File** menu. New icons can be created using **New...** in the File menu.

The command line to start `inquest` has the form:

► `inquest` *{options} bootable_file {options}*

where: *bootable_file* is the path name of a collected bootable application file.

options is a list of one or more options from table 2.1. Other options not listed in the table may not be used with interactive debugging.

Options may be entered in upper or lower case.

Options can be given in any order, but the order of the options can affect the meaning of the command.

Options must be separated by spaces.

If `inquest` is invoked with no options and no bootable file, then help information is displayed, briefly explaining the command line arguments.

Option	Description
`-CMD`	Create command language only interface to the debugger.
`-DS`	Do not stop the program at the beginning of the user code.
`-LOG` *filename*	Log operations to the file *filename*.
`-NR`	Do not assert reset on the target hardware.
`-SC` *filename*	Load the bootable file *filename* onto the target T2/T4/T8-series network.
`-SI`	Switch on the display of extra information.
`-SL` *resource*	Use the target host link connection *resource*, overriding the `TRANSPUTER` parameter.
`-SN` *nif_file*	Initialize the T9000 network with *nif_file*.
`-ZI`*isearch*	Use *isearch* as the search path, overriding `ISEARCH` parameter.

Table 2.1 `inquest` interactive debugging command options

The order of some of the `inquest` options in the command line is significant. Each `inquest` option implies one or more actions by `irun` and the actions are carried in the order described in the `irun` chapter of the *Toolset Reference Manual.*

2.3.1 Example `inquest` command lines

The following command resets the target hardware specified by `TRANSPUTER` and loads it with the bootable file `app.btl`. The interactive debugging session begins with the program stopped at the beginning of the user code. This is the normal command line to debug the program `app.btl` interactively.

```
inquest app.btl
```

The following command resets the target hardware `mytarget`, loads it with the memory configuration file `memconf.btl` and then loads the bootable file `app.btl`. The interactive debugging session begins with the program stopped at the beginning of the user code. This is the normal command line to debug the program `app.btl` interactively.

```
inquest -sl mytarget -sc memconf.btl app.btl
```

The following command loads the skip loader onto a T2/T4/T8-series network then loads `app.btl` onto the network found down link 2 from the root processor and interactively debugs it.

```
inquest -sc skip2.btl app.btl
```

2.3.2 Errors detected by the processor

Target processors will trap certain types of errors which may occur during execution of an application, such as arithmetic overflow in occam or an `abort` function call in ANSI C. If this happens during interactive debugging then the response of the debugger depends on the type of processor being used.

T2/T4/T8-series except T450

On T2/T4/T8-series transputers except the T450, an error will cause the processor to halt and the post-mortem debugger to be initiated. The state of the processor can then be explored. The processor cannot resume, but can only be restarted from the beginning of the code.

ST20450 (T450)

On the ST20450 (T450), if an error occurs the behavior is similar to a breakpoint hit. The thread with the error will halt and the debugger will signal an error event. A message is displayed saying that an error event has occurred and describing the type of error. The thread can be restarted in the same way as after a breakpoint hit, but the subsequent behavior of the application may or may not be meaningful, depending on the nature of the error.

The debugger traps ST20450 errors using the ST20450's error and breakpoint trap handling. For this reason, application code which uses these trap handlers may not be interactively debugged with INQUEST. Applications may safely use a scheduler trap handler, which is not used by INQUEST.

If an ST20450 application sets the trap handler to null then the behavior is similar to a T2/T4/T8-series transputer. A subsequent error will cause the processor to halt and the post-mortem debugger to be initiated. The state of the processor can then be explored. The processor cannot resume, but can only be restarted from the beginning of the code. The ANSI C library function `halt_processor` uses the null trap handler and will cause the processor to halt even if the application is being interactively debugged.

T9000-series

On T9000-series transputers, if an error occurs the behavior is similar to a breakpoint hit. The process with the error will halt and the debugger will signal an error event. A message is displayed saying that an error event has occurred and describing the type of error. The process can be restarted in the same way as after a breakpoint hit, but the subsequent behavior of the application may or may not be meaningful, depending on the nature of the error.

The debugger traps IMS T9000 errors and interrupts using the IMS T9000's trap handling mechanism. For this reason, application code which uses trap handlers may not be interactively debugged with INQUEST. If the user sets a trap handler then breakpoints, watchpoints and interrupts will trap to the user's trap handler instead of the debugger kernel.

If an IMS T9000 application sets the trap handler to null then the behavior is similar to a T2/T4/T8-series transputer. If a subsequent error occurs then the processor will halt and the post-mortem debugger will be initiated. The ANSI C library function `halt_processor` uses the null trap handler and will cause the processor to halt even if the application is being interactively debugged.

2.4 Post-mortem debugging

Post-mortem debugging is used after an application has halted because of an error or because the user has halted it. This may occur during an interactive debugging session or during normal execution of an application. In the latter case, the code must have been compiled with full debugging data, i.e. with the `G` option for ANSI C or without the `D` option for occam.

Normally, post-mortem debugging is carried out by directly exploring the state of the target hardware after the application has halted. Sometimes it may be preferred to save the post-mortem state of the application in a dump file. This facility is provided by the `idump` tool. The state may then be explored by reading the dump file without using the target. Debugging from a dump file is described in section 2.4.5.

2.4.1 Debugging after an interactive session

The interactive debugger can initiate post-mortem debugging in any of three situations:

1 During interactive debugging when the T2/T4/T8-series transputer error flag is set or a T9000-series or ST20450 unmasked error occurs, for example if the null trap-handler is set. In this case a dialog box appears with three options, **Analyse**, **Quit** or **Restart**. **Analyse** will initiate post-mortem debugging; **Quit** will exit from the debugger; **Restart** will reload and start the application from the beginning.

2 When an interactive debugger operation or command is timed-out. The time-out period is set by the environment variable `HKTimeout` and is about 20 seconds by default. Commands and operations will time-out after this time. This allows recovery from deadlocks, loops and catastrophic errors which do not set the error flag. A dialog box appears with three options, **Analyse**, **Quit** or **Restart**. **Analyse** will initiate post-mortem debugging; **Quit** will exit from the debugger; **Restart** will reload and start the application from the beginning.

3 When the **Analyse** operation in the **Execution** menu is selected or the `analyse` command is entered.

If **Analyse** is selected in any of these cases, the target hardware will be analyzed, so all low priority threads will be halted at deschedule points.

2.4.2 Debugging after normal execution

Post-mortem debugging can be initiated by the user from a command line by the command `inquest`. This allows the debugger to be used after normal execution of an application when the application has been halted by setting the error flag or user interruption. The application must have been compiled with the options for full debugging data and the compiled code files (extension `.tco`) and libraries must be on the search path given by `ISEARCH`. On PCs, the `ISEARCH` path may be held in a Windows initialization file and modified using the `ilaunch` tool or the `iset` tool, which are described in the *Toolset Reference Manual*. The application need not have been linked with the debugging libraries.

The debugger will analyze the target hardware, which means that all low priority threads on all processors will be halted at deschedule points.

On Microsoft Windows systems, `inquest` is run in the same way as other applications. For example, an `inquest` icon can be double clicked in the `inquest` program group in the Program Manager. The command line associated with an icon can be changed by selecting the icon and using **Properties...** in the **File** menu. New icons can be created using **New...** in the **File** menu.

The command line to start `inquest` in post-mortem mode has the form:

► `inquest` *bootable_file* `-pm` *{options}*

where: *bootable_file* is the path name of a collected bootable application file.

options is a list of one or more options from table 2.2. The `-pm` option is to indicate post-mortem mode.

> Options may be entered in upper or lower case.
>
> Options can be given in any order, but the order of the options can affect the meaning of the command.
>
> Options must be separated by spaces.

Option	Description
`-DF` *dumpfile*	Debug from *dumpfile* instead of a target (see section 2.4.5).
`-NA`	Do not assert analyse on the T2/T4/T8-series network.
`-PM`	Post-mortem mode.
`-SC` *filename*	Load the bootable file *filename* onto the target T2/T4/T8-series hardware.
`-SI`	Switch on display of extra information.
`-SL` *resource*	Use the target host link connection *resource*, overriding the `TRANSPUTER` parameter.
`-ZI`*isearch*	Use *isearch* as the search path, overriding `ISEARCH` parameter.

Table 2.2 `inquest` post-mortem command options

Running `inquest` with no parameters causes it to display its version number, build date and brief help information.

The order of some of the `inquest` options in the command line is significant. Each `inquest` option implies one or more actions by `irun` and the actions are carried out in the order described in the `irun` chapter of the *Toolset Reference Manual.*

2.4.3 Example `inquest` command lines

The following command analyses the target hardware and post-mortem debugs the application using the bootable file `app.btl`. This is the normal command line to debug the program `app.btl` post-mortem:

```
inquest app.btl -pm
```

The following command loads the skip loader on a T2/T4/T8-series network then post-mortem debugs the network found down link 2 from the root processor, using the bootable file `app.btl`.

```
inquest app.btl -sc skip2.btl -pm
```

2.4.4 Proceeding after analyzing

After analyzing the target hardware, the debugger explores the state of the target hardware. The state of the system can then be examined by the user, but the application cannot continue, so operations such as **Continue**, **Step**, **Break** and **Watch** are disabled.

If the error flag was set, then a message will appear in the output window giving the source line at which the error occurred. The debugger will be located to this line. The process causing the error can then be explored and other threads located using **Jump**.

The displayed state will be the last known state before the post-mortem debugging was started. The type of state of each thread will be one of the states listed in section 2.5.3 or *halted on error flag*. The full state of a thread will be updated to the final analyzed state when the debugger locates or finds that thread.

In post-mortem debugging the **Find Threads** operation will find scheduled threads, threads waiting for timers and threads waiting for channels defined at configuration level. It cannot find threads waiting for unconfigured channels, i.e. channels defined in the ANSI C source code. If necessary, these threads can be located by **Jump**ing down channels.

Information about the processor and the processor queues is displayed in the attribute window and the memory window.

2.4.5 Debugging using a dump file

Sometimes it may be desirable to save the post-mortem state of the application rather than exploring it on the target. Saving the state keeps a record for future inspection or reference. Saving the state also frees the target hardware so that, for example, it may be used by other users while the application is being debugged. It may also be used if the host development system is not available when the application halts or if the target hardware is not available to the host development system.

Creating a dump file

The `idump` tool is provided to save the post-mortem state of the application in a host file. The dump file contains the contents of the memory and the registers used for debugging. The default name for the dump file is `core.dmp`.

The `idump` tool must be used immediately after the application has halted. If `idump` fails or the post-mortem debugger has been used then the state has been lost and cannot be dumped. Similarly, the post-mortem debugger cannot be used on the target once the state has been dumped.

`idump` must have access to the configuration binary file. It expects this file to have the same filename root as the bootable file, but with the extension `.cfb`. For example, if the application bootable is `app.btl` then `idump` will look for a configuration binary named `app.cfb`. This file must be in the current directory or on the `ISEARCH` path.

The command line to run **idump** is:

► **idump** *bootable_file* {*options*}

where: *bootable_file* is the path name of the collected bootable application file.

options is a list of one or more options from table 2.3.

> Options may be entered in upper or lower case.
>
> Options can be given in any order, but the order may be significant.
>
> Options must be separated by spaces.

Option	Description
-SI	Display extra run-time information.
-SL *resource*	Use the target host link connection *resource*, overriding the **TRANSPUTER** parameter.
-SC *filename*	Load the bootable file *filename* onto the target T2/T4/T8-series hardware.
-DF *dumpfile*	Write the output to the dump file *dumpfile.*
-NA	Do not assert analyse on the T2/T4/T8-series network.
-W	Show a busy sign.
-E	Dump only the processors with the error flag set.
-F	Dump free memory as well as allocated memory.
-R	Dump only the register state.

Table 2.3 **idump** command line options

Normally **idump** only saves the memory allocated by the build tools for code and work space. The **-F** option forces **idump** to dump the whole of the physical memory, including the 'free' memory.

Examples:

The following command line dumps the used memory space and registers of the target hardware specified in **app.cfb** that is accessed by the host link given by the **TRANSPUTER** parameter:

```
idump app.btl
```

The following command line dumps all of the memory (including the free memory) and the registers of the target hardware specified in **app.cfs** accessed by the host link connection **mytarget** and displays a busy sign:

```
idump app.btl -sl mytarget -f -w
```

The following command line dumps only the processors that have the error flag set:

```
idump app.btl -e
```

The following command line dumps just the register state of the processor that has the error flag set:

```
        idump app.btl -e -r
```

Using the dump file

The dump file may be read by the debugger in post-mortem mode by using the `inquest` option `-DF`. The debugger uses only the host when debugging from a dump file, so access to the target is not needed.

For example, to debug the application `app.btl` from the dump file `core.dmp`, use the following command line:

```
        inquest -pm -df core.dmp app.btl
```

2.5 Debugging multi-threaded programs

The debugger supports debugging of single thread or multi-threaded programs running on the target hardware. Debugging multi-threaded code is different from debugging single thread code in that, for example, hitting a breakpoint might stop only one thread of execution while the rest of the program may be able to continue.

The debugger is not synchronous with the user code, so the user has full access to the debugger while code is running on the target hardware. The user may browse through the code and inspect variables while some or all of the threads are either running or stepping. For running threads, the state shown is the last known state. The last known state is the state when the thread was last stopped or an interrupt was requested.

Multiple debugging windows may be opened, and each may display a different part of the state of the program. However, the state being inspected by all the windows is the same. For example, if a thread is interrupted from one window then that thread will become stopped in all windows. Similarly, a breakpoint or watchpoint set in one window will be set in all windows.

In the debugger, we need to distinguish between two different types of task, so the terminology in this document is slightly different from that used in the *ANSI C Toolset* and the *occam 2 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 the build time, while a thread is created dynamically while the program is running.

2.5.1 Shared code

When code is shared between threads of execution, the debugger can treat the code of each thread as distinct and distinguish between shared code executed by different threads of execution. For example, a breakpoint may be set in one thread so that only that thread can hit the breakpoint, while another thread executing the same code will not be stopped. Alternatively, a breakpoint may be set so that it stops any thread executing that line of code.

The debugger can also distinguish between instances of automatic variables declared in shared code. An automatic variable is a C variable which is declared inside a function, so multiple executions of the function code give rise to multiple instances of the variable. In occam, all variables behave like C automatic variables. Each thread that executes the shared code will have a different instance of any variable declared in the shared code. Each instance will be in a different memory location and may have a different value. In this case the user must be careful when inspecting the value of a variable that the correct thread is being inspected. A thread may be selected by using the browser. A watchpoint set on such a variable may apply to one thread or all of them.

In C, a function that is called recursively may also cause multiple instances of automatic variables. The debugger is able to distinguish these instances by their frames, i.e. their locations on the stack. The user may inspect or set watchpoints on any instance by selecting a particular frame using the browser.

C static variables have only one instance within a process and are not associated with a particular thread. If a watchpoint is set on a C static variable then any thread accessing that variable will stop.

2.5.2 Processes

A debugging process is an instantiation of a program that is executed on a target processor. An ANSI C process has a single `main` function and so a single entry point. It is compiled and linked into a single linked unit. A C process is declared as a node in the C configuration code and the linked unit is attached to this declaration by a **use** statement in the configuration code. A process consists of a code region and data regions, is statically allocated and is uniquely identified by the process name used in the configuration program or a number called its *process identifier* or *pid.*

The fragment of C configuration code below declares and places the process `facs`:

```
node(element="process",
  interface(input in, output out),
  stacksize=20k,heapsize=40k,priority=low) facs;
use "facs.lku" for facs;
place facs on t3;
```

An occam process is all the code for one processor. It is identified by the name of the processor with the suffix '`_p`' or a number called its process identifier or pid.

The fragment of occam configuration code below declares and places the process `t1_p`:

```
PROCESSOR t1
  PAR
    app (fs, ts, square.to.app, app.to.facs)
    facs (app.to.facs, facs.to.square)
    square (facs.to.square, square.to.app)
```

A process is marked as debuggable if it does not have the configuration attribute `nodebug` set to `TRUE` (see section 2.2). A module of a linked unit may have full or minimal debugging data depending on the compiler options. The source code can only be displayed if the process has been compiled with full debugging data. If a code module has only minimal debugging information then only the stack trace can be seen. A code module will have no debugging data if the user has written assembly code or the source or compiled (`.tco`) code file is not on the search path given by `ISEARCH`. If there is no debugging data then debugging can only be at assembly code level.

2.5.3 Threads

Within a given debugging process there are one or more threads of execution. Each thread of execution is sequential, though it may generate other threads which subsequently run at the same time, using the processor's built-in micro-kernel.

In ANSI C, a thread of execution has its own stack region but can share the static and heap regions of the process in which the thread was created. The creation of threads is dynamic and performed by the ANSI C functions `ProcPar`, `ProcParList`, `ProcPriPar`, `ProcRun`, `ProcRunHigh` and `ProcRunLow`. While a process is running, there is always at least one thread of execution running in the process. When the last thread terminates then the process terminates. The following fragment of ANSI C declares and creates a thread `p_sum`:

```
Process *p_sum;
p_sum = ProcAlloc(sum, 0, 1, p1);
ProcRun(p_sum);
```

In occam, a thread of execution has its own stack space allocated at compile time. The creation of threads is dynamic and performed by the occam `PAR` construct at program or configuration level. While a process is running, there is always at least one thread of execution running. When the last thread of a process terminates then the process terminates. The following fragment of occam declares and creates three new threads. In this example, the thread which contains the `PAR` hangs in a running or stepping state and cannot continue until all three new threads have terminated.

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

When a thread of execution is stopped, the execution state of the thread is saved by the debugger. The execution state includes the contents of all the relevant processor registers when the thread was stopped. This includes the **Iptr**, which is used to indicate (in the code window) where the thread has reached, and the **Wptr**, which is used to calculate the addresses of local variables and the frame stack.

The debugger determines the memory range that the thread is using as its workspace and assigns a unique identity to that workspace range, called a thread identifier or tid. This thread identifier can then be used to denote the thread for the rest of its life. The thread identifier is displayed in the attribute window when the thread is selected using the browser.

The debugger may show a thread as being in one of seven different thread states - stopped, alt-waiting, chan-waiting, timer-waiting, scheduled, stepping and running, as described below. The state of a thread may be shown in the browser window or the attribute window.

In interactive debugging, the state displayed is the thread state when the thread was last sampled. A sample occurs when one of the following occurs:

- an **Interrupt**, **Find Threads**, **Continue**, **Step** or **Next** operation is requested;
- an equivalent command is entered;
- the thread stops because of an interrupt, watchpoint, breakpoint or thread monitor.

In each case, the register state is sampled, which enables the printing of the values of variables and other state information.

Stopped threads

A thread is *stopped* when it has hit a breakpoint or watchpoint or a thread monitor or when an **Interrupt** on the thread has been completed. Initially all debuggable threads are stopped. Watchpoints and breakpoints can be set in a stopped thread and the values of variables can be printed.

In interactive debugging, a thread can resume after it has been stopped. The **Continue** operation will cause it to start running, while **Next** or **Step** operations will cause it to start stepping.

Alt-waiting, chan-waiting, timer-waiting and scheduled threads

Interrupt and **Find Threads** set temporary breakpoints after the current statement or instruction. If a **Find Threads** has been requested or a thread has been **Interrupted** but it has not yet reached the temporary breakpoint then the thread will be in one of the following states:

- *alt-waiting* if the thread is waiting for an alternative.
- *chan-waiting* if the thread is waiting for a channel other than in an alternative.
- *timer-waiting* if the thread is waiting for a timer other than in an alternative.
- *scheduled* if the thread is waiting on a queue for processor time.

The displayed state of the thread is the state when the **Interrupt** or **Find Threads** was requested. If the thread has been **Interrupted** then the values of variables can be printed and the thread will become stopped when it hits the **Interrupt** breakpoint.

The debugger can perform other operations while one or more threads are in any of these states.

Stepping

A *stepping* thread is executing a **Step** or **Next** operation. It may have to wait for a timer or channel communication and will continue to be stepping while it waits. When the step is completed the thread becomes stopped. A stepping thread can also be changed to alt-waiting, chan-waiting, timer-waiting or scheduled by an **Interrupt** operation.

The debugger can perform other operations while a thread is stepping. For example, several threads may be stepping at the same time.

Running

A thread is *running* if it is not prevented from executing by the debugger and it is not stepping, alt-waiting, chan-waiting, timer-waiting or scheduled. A running thread may be executing on the processor or waiting for a timer or channel communication, including host input and output. In the code window display, a running thread is shown in the state at which it was last stopped. A running thread can be stopped by an **Interrupt** operation.

The debugger can perform other operations while a thread is running. For example, several threads may be running at the same time.

2.5.4 Initial process states

The initial state of each process depends on how it was built and the `irun` command line. If the `irun` option `-DS` is used then the application will start and keep running until interrupted or an error or termination occurs.

Without the `-DS` option, a C process which is marked as debuggable and is compiled with full debugging data will stop at the start of its `main` function after the initialization code. A C process which is debuggable but does not have full debugging data cannot find the start of the `main` function. Such a process is initially stopped at the process entry point, i.e. at the start of its initialization code. A C process which is not marked as debuggable executes normally without being stopped by the debugger.

An occam process which is marked as debuggable and is compiled with full debugging data will stop in the initialization code. An occam process which is debuggable but does not have full debugging data is initially stopped at the process entry point, i.e. at the start of its initialization code. An occam process which is not marked as debuggable executes normally without being stopped by the debugger.

2.6 Debugging facilities

2.6.1 Breakpoints

A breakpoint may be set during interactive debugging on a source statement or at a processor instruction address. When a thread hits the breakpoint it will be stopped immediately before executing that statement or instruction.

A breakpoint can be set for a process, in which case any thread of the process that attempts to execute the statement or instruction is stopped. A breakpoint can also be set for a specific thread or frame in which case only that thread or frame will stop when it attempts to execute the statement or instruction. A statement can have more than one breakpoint set on it.

When a breakpoint is set, an *event number* is returned. This number is displayed whenever the breakpoint is hit or listed to identify the event. It may be used as a parameter to the `when` or `wait` command in order to wait for a specific event.

Breakpoints are set by the user with the **Break** operation or `break` command. Hidden breakpoints may also be set by INQUEST during **Step To**, **Step Out**, **Interrupt**, **Next** and **Find Threads** operations. The hidden breakpoints do not appear in the list of breakpoints and are deleted automatically when the operation completes.

2.6.2 Single stepping

During interactive debugging, a stopped thread can be made to execute the next statement or instruction. This facility is called *single stepping.* If a thread single steps at a function or procedure call, it may optionally step through the function or procedure or step over it. *Stepping through* means that the thread executes as far as the first statement or instruction within the function or procedure. *Stepping over* means that the thread executes as far as the return from the function or procedure.

A single step may include a communication or timer wait, in which case the thread may have to wait and the single step will not complete until the communication or timer wait has completed. The debugger can perform operations on other threads while one or more threads are single stepping.

2.6.3 Watchpoints

A watchpoint may be set during interactive debugging on a program variable and may watch for read accesses, write accesses or any accesses. When a thread tries to make an access of the watched type to that variable it will be stopped. Hitting a watchpoint results in the thread being stopped immediately before the access is made.

A watchpoint may be set on a variable for a process. If the variable is a C static then any thread that accesses it will be stopped. If the variable is a C automatic or an occam variable then every future instance of that variable has a watchpoint set on it.

If a watchpoint is set for a thread on a C automatic variable or an occam variable then the watchpoint is on every existing and future instance of the variable in that thread. Only that thread can hit that watchpoint. A variable can have more than one watchpoint set on it.

A temporary watchpoint may also be set which will automatically be removed after it has been hit for the first time.

When a watchpoint is set an event number is returned. This number is displayed whenever the watchpoint is hit or listed to identify the event.

Internal channels and external channels may have watchpoints set on them. Virtual channels cannot have watchpoints set on them.

2.6.4 Interrupting

One or more running threads can be stopped by interrupting during an interactive debugging session. The debugger 'interrupts' the threads by setting temporary breakpoints on them. The breakpoint is inserted before the next statement or instruction. The breakpoints are removed automatically when the thread stops. This facility may be used if, for example, a thread is stuck in a loop.

If an **Interrupt** has been requested on a thread but the thread has not reached the interrupt breakpoint then the thread may be alt-waiting, chan-waiting, timer-waiting or scheduled. The displayed state of the thread is the state when the **Interrupt** was requested, which enables a set of deadlocked threads to be found and explored.

2.6.5 Thread monitors

During interactive debugging, if a running thread creates a new thread then the new thread is not visible to the debugger until it is stopped or a **Find Threads** operation has been completed. Similarly, the debugger does not normally indicate when a thread has terminated. The debugger provides *thread monitors* so that the creation or termination of a thread can be monitored.

A thread monitor can be placed on a process. A *thread birth monitor* stops a thread as soon as it is created. Birth monitors are only available for C threads. A *thread death monitor* informs the user when a thread has terminated.

When a thread monitor is set an event number is returned. This number is displayed whenever the thread monitor is hit or listed to identify the event.

2.6.6 Stack tracing

When a thread has been selected, a *call stack trace* can be displayed which gives details of the nested function and procedure calls of the thread which have not returned. Each function or procedure call of a stack trace is called a *frame* and is allocated a number known as the *frame identifier* or *fid*. The frame identifiers are shown in the stack trace in the browser window. In the stack trace, the frame that created the current thread is marked with an asterisk (*).

Breakpoints and watchpoints can be set that are local to a particular function or procedure call, so allowing recursive programs to be debugged interactively.

2.6.7 Examining variables

The values of C static variables can be displayed at any point in the program execution. The values of C automatic variables and occam variables can be displayed if the thread is stopped or an **Interrupt** or **Find Threads** has been requested. A simple expression syntax is supported to enable the values of structured variables to be displayed, as described in sections 2.9.4 and 3.1.

2.6.8 Jumping down a channel

The value of a channel variable can be used to switch the debugging context to a thread or process waiting for communication on that channel. This is known as jumping down a channel.

Channels are one-to-one and synchronous, so any one channel has only three states:

1 empty – i.e. not being used;

2 pointing to one thread waiting for communication on the channel;

3 communicating between two threads.

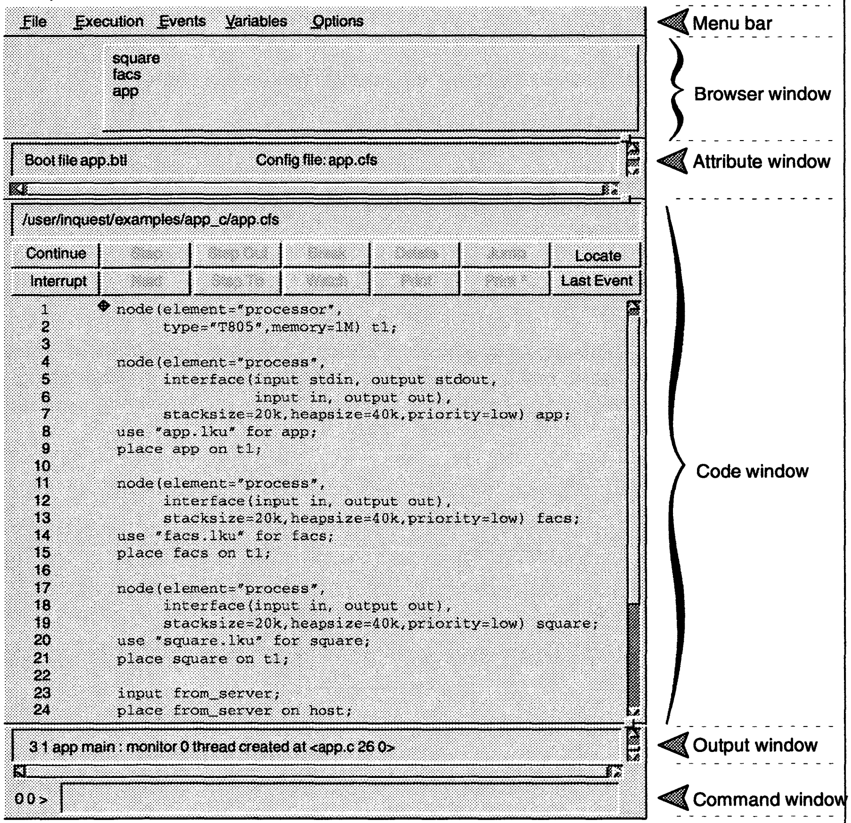
If a channel is in state 2 then the debugger can jump to the thread that is waiting, since the channel points to it. The debugger selects the waiting thread as if it had been selected using the browser.

2.6.9 Low level features

A memory viewing function allows segments of memory to be viewed in various formats. A low level view of a thread can be selected allowing its code to be disassembled, its workspace to be examined and facilitating instruction level breakpointing and single stepping.

2.7 The debugger display

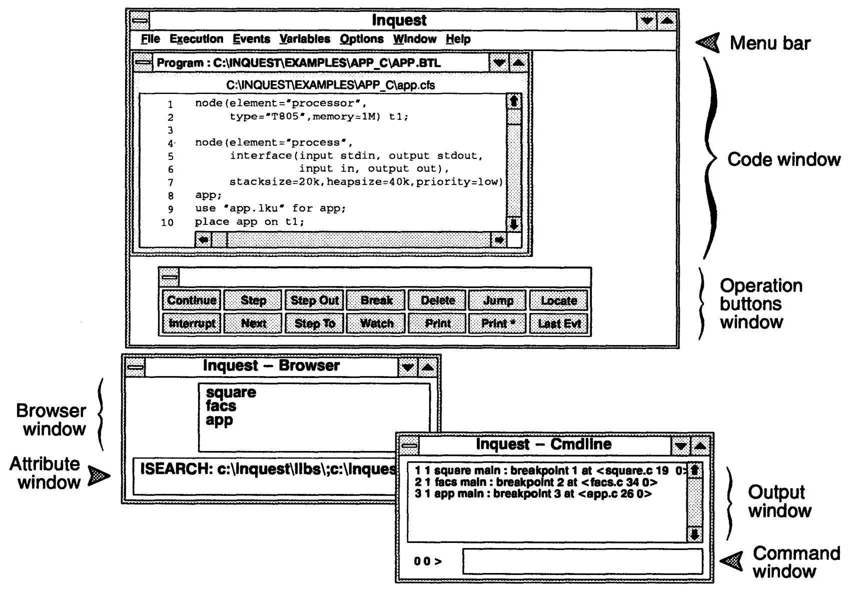
The debugger display consists of a set of windows. On X-Windows systems the display looks like figure 2.3. On Microsoft Windows systems the display looks as shown in figure 2.4. The windows are used for browsing around the program, displaying attributes of the program, displaying the source code, displaying the debugger messages and interacting with the user by means of a mouse, buttons, menus and the command language.



```
node(element="processor",
     type="T805",memory=1M) t1;

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 t1;

node(element="process",
     interface(input in, output out),
     stacksize=20k,heapsize=40k,priority=low) facs;
use "facs.lku" for facs;
place facs on t1;

node(element="process",
     interface(input in, output out),
     stacksize=20k,heapsize=40k,priority=low) square;
use "square.lku" for square;
place square on t1;

input from_server;
place from_server on host;
```

```
3 1 app main : monitor 0 thread created at <app.c 26 0>
```

Figure 2.3 The X-Windows debugger display



```
node(element="processor",
     type="T805",memory=1M) t1;

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 t1;
```

ISEARCH: c:\inquest\libs\;c:\inques

```
1 1 square main : breakpoint 1 at <square.c 19  0>
2 1 facs main : breakpoint 2 at <facs.c 34 0>
3 1 app main : breakpoint 3 at <app.c 26 0>

0 0 >
```

Figure 2.4 The Microsoft Windows debugger display

Instructions can be given to the debugger either in the form of operation requests or commands. Operations are requested by clicking on a button, clicking on a pull-down menu or by using accelerator keys. Commands are typed in the command window or a sequence of commands may be written in a file and used as a script. A command script may be run by a single `load` command or automatically on start-up.

The *browser* enables the user to browse through the program and select an object to be examined, which may be the whole program, a process, a thread or a frame. The *attribute window* gives information on the object that has been selected. The *code window* displays source code or disassembled code that is appropriate to the state of the browser. The *output window* displays the responses of the debugger to user operations and commands.

The debugger display can be duplicated using the **Open Window** operation to provide as many displays as needed. Each display can be set to a different browser state or may display a different section of code as required by the user.

2.7.1 The browser window

The browser window is used to select the context for operations and the display in other windows and to display the state of the threads.

The types of objects which can be debugged in a program are arranged into the hierarchy of program, processes, threads and frames. The browser is used to select a particular level of the hierarchy to enable debugging to be performed at that level and to select a particular object at that level. The browser is described in section 2.8.

The browser window consists of two regions; a button region and a list of objects at the current level. The button region is on the left of the browser window and on the right is the list of objects.

The objects at the current level are represented by a list of names, e.g. process names or thread names. If the list is too long to be displayed in its entirety then a scroll-bar is provided allowing the visible portion to be altered. An object can be selected by clicking on it with the left mouse button.

The button region contains up to three buttons which provide movement to other levels of the hierarchy.

2.7.2 The attribute window

The attribute window displays detailed information about the object selected in the browser window. If there is too much information to be displayed in its entirety then a scroll-bar is provided allowing the visible portion to be altered.

2.7.3 The code window

The code window displays the contents of source files or disassembled code files. It is possible for the user to change the file that is displayed in the code window as described in section 2.8.5. The user has a choice of possible files to view, depending on the current process, thread and frame. The restrictions are intended to limit the files that can be selected to those that are pertinent to the current context.

A sub-window is displayed at the top of the code window which displays the full path-name of the file currently displayed in the code window.

The following markers are shown on Sun systems in the left margin of the code in the code window to indicate various locations in the code:

◈	Selected line
⊳	Next statement or instruction to be executed at the last known state
⊘	Breakpoint set
◷	Watchpoint set

The corresponding markers shown on PC systems in the code window are as follows:

◇	Selected line
▷	Next statement or instruction to be executed at the last known state
△	Breakpoint set
⌚	Watchpoint set

2.7.4 The output window

The output window is used to display messages from the debugger. Some messages are prefixed by the process identifier and thread identifier (see section 3.2) to indicate the context.

There are a number of different types of message that may appear in this window, as follows:

Confirmations

When an operation has been selected, the command language equivalent of the operation is displayed, prefixed by the process and thread identifier context. When a command is entered, the command is echoed in the output window and the output from the command language interpreter will be displayed. Confirmations are displayed in all debugging windows.

Output and error messages

When an operation has been selected or a command entered, there may be output or an error message as a direct consequence of that operation or command. Output and error messages are displayed only in the debugging window where the operation was selected or the command was entered.

Events

When an event occurs, such as a breakpoint or watchpoint hit, a message will be displayed announcing its occurrence prefixed by the process and thread identifier context. The message is displayed in all debugging windows.

2.7.5 The command window

The command window provides access to the debugger command language interpreter. Commands may be entered at any time. The command language is described in Chapter 3. The browser and the command language interpreter maintain the same current context. If the context is changed in one then the other will change with it. The current process identifier and thread identifier are displayed to the left of the command window.

2.7.6 Hexadecimal numbers

Hexadecimal numbers are generally displayed preceded by a hash (#), e.g. `#80000A70`. Hexadecimal numbers may be entered in any of the following formats:

- With a leading `#`, e.g. `#5ab67`
- With a leading `0x`, e.g. `0x5ab67`
- With a leading `%`, indicating that the sign bit is set. For example, `%70` means hexadecimal `80000070` on a 32-bit processor. There must not be a space between the `%` and the number.

2.8 The browser

The browser allows the user to navigate through the hierarchy of executing objects (i.e. processes, threads and frames) in a structured way. It also allows the scope of operations to be changed. The current state of the browser is shown in the browser window. The browser and the command language interpreter maintain the same current context; if the context is changed in one then the other will change with it.

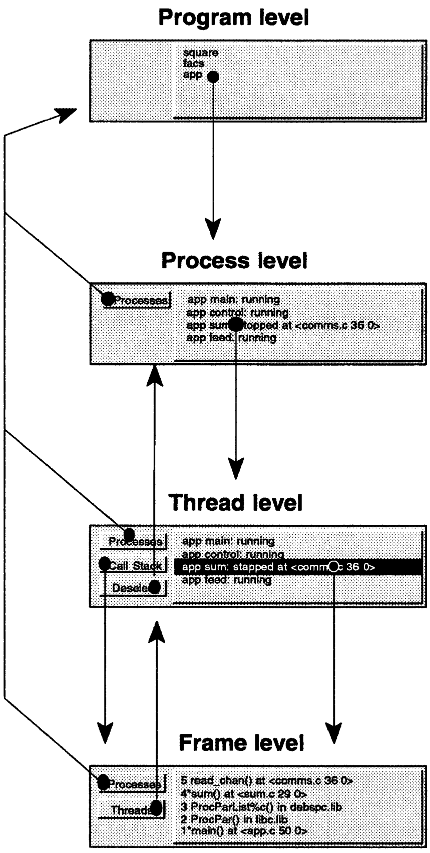
A current process, thread and frame may be selected using the browser. If a frame is selected then the current thread and process are the thread that contains the selected frame and the process that contains that thread. If a thread is selected then the current process is the process that contains the selected thread. The current process identifier and thread identifier are displayed to the left of the command window.

If code is shared between parallel threads of execution then the browser may be used to select one thread. The values of variables in that thread may then be inspected and breakpoints and watchpoints set in that thread.

If a function is called recursively then the browser may be used to select one call of the function, i.e. one frame. The values of variables in that frame may then be inspected and breakpoints and watchpoints set in that frame.

The browser has four levels: program level, process level, thread level and frame level. For each level there is a different browser window display. Figure 2.5 shows the browser window states and the transitions when the user clicks on a button or object name.

The browser level affects which debugger operations can be performed and may change their effect. In general, the level affects the scope of an operation. At program level an operation will generally affect all processes. At process level a process is selected and an operation will generally affect all the threads in the selected process. At thread level, a thread is selected and an operation will generally affect all the frames in the selected thread. At frame level, a frame may be selected in which case an operation will affect only that frame.



```
square
facs
app
```

```
app main: running
app control: running
app sum: stopped at <comms.c 36 0>
app feed: running
```

```
app main: running
app control: running
app sum: stapped at <comms.c 36 0>
app feed: running
```

```
5 read_chan() at <comms.c 36 0>
4*sum() at <sum.c 29 0>
3 ProcParList%c() in debspc.lib
2 ProcPar() in libc.lib
1*main() at <app.c 50 0>
```

Figure 2.5 Browser states and displays

Certain debugger operations cause the debugger to select an object without using the browser. The **Jump** operation makes the debugger select the thread that is waiting for a channel communication. The **Last Event** operation makes the debugger select the thread in which the last event occurred.

2.8.1 Program level

At the program level the list of processes in the application is displayed in the browser window. The attribute window displays the name of the bootable program and the code window displays the configuration source code.

Clicking on a process from the list in the browser window will cause the browser to move to the process level, showing the threads in the selected process.

2.8.2 Process level

At process level, one process has been selected. The browser window display shows a list of threads currently known in the selected process and a **Processes** button. The attribute window displays the attributes of the process, namely:

- the process name, process identifier and starting priority;
- the name of the file that contains the linked unit code of the process and its entry point;
- the identifier, type and memory size of the processor on which the process is executing;
- the locations and sizes in memory of the stack, static and heap.

Clicking on the **Processes** button moves the browser to the program level. Clicking on a thread moves the browser to the thread level, with the selected thread as the current thread.

2.8.3 Thread level

At the thread level, a list of the threads that exist within the current process is displayed in the browser window, with the current thread highlighted. At this level the browser window shows **Processes, Call Stack** and **Deselect** buttons.

When a thread has been selected then the attribute window displays the attributes of the thread, namely:

- the process name, thread name, thread starting point and status;
- the range of the workspace of the thread in memory and the priority of the thread;
- the values of the **Iptr** and **Wptr**;
- the processor identifier;
- the values in the evaluation register stacks.

At thread level, the code window may display the source code of the thread or the disassembled machine code. The **Assembly** operation toggles between these two displays. If the thread is stopped then its current location in the code is displayed. If an interrupt has been requested then the state when the interrupt was requested is displayed. If the thread is running, then the location displayed is where the code last stopped. If the thread has not started then its entry point in the code will be displayed.

All events set at thread level are removed when the thread terminates.

From the thread level, clicking on the **Processes** button moves the browser to the program level and clicking on the **Deselect** button moves the browser to the process level. Clicking on the **Call Stack** button moves the browser to the frame level in the current thread. Clicking on a thread which is not already selected selects that thread. Clicking on a thread which is selected, moves the browser to the frame level in the selected thread.

2.8.4 Frame level

At the frame level, a stack of the frames of the current thread is displayed in the browser window, which shows the current state of a stopped thread or otherwise the most recent stopped state. The first element of the list is the 'current' execution frame of the thread, the next its caller, and so on. The browser window also shows a **Processes** button and a **Threads** button and, if a frame is selected, a **Deselect** button. Clicking on the **Processes** button moves the browser to the program level. Clicking on the **Threads** button moves the browser to the thread level. Clicking on the **Deselect** button deselects the current frame.

All events set at frame level on a function or procedure are removed when the function or procedure returns.

When a frame has been selected then the attribute window displays the attributes of the frame, namely:

- the name of the function or procedure called;
- the thread identifier;
- the line where the function or procedure was called and the current line;
- the values of the **Iptr** and **Wptr** and the processor identifier;
- the actual parameters to the function or procedure call.

2.8.5 Code display

The code display is the section of source or assembly code visible in the code window using the scrolling facilities, plus the next statement marker (⊠ on Suns or ▷ on PCs) and the file pathname in the filename window. A debugger display is said to *locate* when it updates the code display to show the appropriate source code file or section of disassembled code for the currently selected context.

When a thread or frame is selected or the **Locate** operation is performed, the code window locates to the appropriate code. If **Assembly** is off and no frame is selected then the highest frame with full debugging data is used for the display. If **Assembly** is off and a frame is selected which does not have full debugging data then the code display becomes blank. A summary of the effects of locating in various states is shown in table 2.4.

Level	Locate to
Program	Beginning of configuration code
Process	Process entry point
Thread, with **Assembly** off	Highest frame with full debugging data
Thread, with **Assembly** on	Top frame
Frame, deselected, with **Assembly** off	Highest frame with full debugging data
Frame, deselected, with **Assembly** on	Top frame
Frame, selected	Selected frame

Table 2.4 Effect of locating

Normally, when an event occurs or an operation is performed on a selected thread the windows displaying the thread locate. Since this happens automatically, this effect is called *auto-locate.*

Auto-locate is turned off in any window when a **Function**, **Module**, **Enter Include** or **Exit Include** operation is used in that window. While auto-locate is turned off, the code display will not change. This facility may be used to hold a particular code display in one window while browsing and executing in another window. Auto-locate is turned on again by any other execution operation, a browser selection or the **Locate** operation.

2.9 Debugger operations

The debugger operations have been divided into groups of related operations. Each group of operations appears in its own pull-down menu in the menu bar. The menus are pulled down by clicking on the required menu name with the left mouse button. The operation within the menu is selected by clicking the left mouse button again on the name of the operation or by entering the underscored letter at the keyboard. Menus and operations may also be selected by means of accelerator keys, which are keyboard dependent but related to the letter underlined on the menu bar or in the menu.

At a particular browser level, only a subset of the operations may be available. Operations that do not apply to a given browser level are 'greyed-out' and cannot be selected. The effect of some operations differs slightly between levels.

A subset of the debugging operations is available via a set of operation buttons that is displayed in the code window.

2.9.1 File menu

Module

List in a pop-up window all the modules (i.e. compilation units) used. All the modules in the current process are listed. Selecting a module and clicking on the **Apply** button will cause the source code corresponding to the compilation unit to be displayed in the code window. Clicking on the **Cancel** button will close the pop-up window. Clicking on the **OK** button is the same as clicking on **Apply** then **Cancel**. The **Module** operation turns off the auto-locate.

Function

List in a pop-up window all the functions defined in the linked unit that the current process is running. Selecting one of the functions and clicking on the **Apply** button will make the code window locate to the selected function. Clicking on the **Cancel** button will close the pop-up window. Clicking on the **OK** button is the same as clicking on **Apply** then **Cancel**. The **Function** operation turns off the auto-locate.

Enter Include

Display an included file in the code window. An included file is selected by clicking on an `#include` or `#INCLUDE` directive in the source code in the code window before clicking on **Enter Include**. The **Enter Include** operation turns off the auto-locate.

Exit Include

Undo **Enter Include**. The **Exit Include** operation turns off the auto-locate.

Search

Search forward or backward within the source file currently displayed in the code window. If a string has been selected in the code window then it will be used as the search string and the search will be forwards. Otherwise a pop-up dialog box will appear, allowing the string to be searched for to be entered, and forwards or backwards to be selected.

Locate

If text is highlighted then the debugger will interpret this as a function name and try to locate to it. If nothing is highlighted, the debugger will display the code location of the current thread. If the thread is stopped then its current position in the code is displayed. If the thread is executing, the code position at which it last stopped is displayed. If this operation is used at process level, the process entry point is displayed. At program level the start of the configuration file is displayed. The **Locate** operation turns on the auto-locate. This operation is not equivalent to the `locate` command.

Open Window

Create a new debugger display window in the same state as the current debugger display window. Once created the new and old displays may be changed independently of each other and any other displays which may exist. On Microsoft Windows systems, only a new code window appears, and the other windows apply to whichever code window is selected.

Close Window

Remove the current debugger display window, unless there is only one display window in which case it will not be removed.

Exit

Quit from the debugger. A pop-up dialog box will appear asking for confirmation. Clicking on the **Yes** button causes INQUEST to exit. Clicking on the **Cancel** button returns to the debugger.

2.9.2 Execution menu

All the execution menu commands except **Find Threads** switch on auto-locate. Of these operations, only **Find Threads** is available during post-mortem debugging.

Continue

Resume execution of stopped threads. At program level, resume all stopped threads of the program. At process level, resume all stopped threads of the process. At thread level, resume the current thread. **Continue** cancels any pending interrupts.

Interrupt

Interrupt debuggable running threads. At program level, interrupt all debuggable running threads of the program. At process level, interrupt all running threads of the process. At thread level, interrupt the current thread. Interrupting a thread means setting a breakpoint on the next instruction the thread will execute and removing the breakpoint when it is hit.

If a thread is waiting for a timer or a channel then the breakpoint will not be hit until the wait has completed. The debugger considers a thread to be stopping if the breakpoint has not been hit, in which case its state can be examined but not modified. High priority threads can only be interrupted when they are in a waiting state, for example waiting for a channel communication to complete.

In order to interrupt a thread, the debugger must be able to find its descriptor by looking on the scheduling queues, timer queues, or by searching known and identifiable channels. If a thread is not in such a state it cannot be interrupted by the debugger.

Step

Continue execution of the current stopped thread until a single step has been executed. Function and procedure calls are stepped into unless they are compiled with minimal debugging information (e.g. `printf` or `so.write.string`) in which case the function or procedure call is stepped over. With no debugging information, the thread will be repeatedly stepped until it reaches code with debugging information.

In occam, stepping an unreplicated `PAR` statement causes every thread created by the `PAR` to be created and then stop. The parent process remains 'stepping' until all the new processes have been terminated. Stepping a replicated PAR causes all the threads it creates to run until they terminate or hit a breakpoint or watchpoint.

Next

Continue execution of the current stopped thread until the next function, procedure or statement has been executed. Function and procedure calls are stepped over.

In occam, **Next** on a replicated or unreplicated PAR statement causes all the threads it creates to run until they terminate or hit a breakpoint or watchpoint.

Step Out

Continue execution of the current stopped thread until the current function or procedure returns. At frame level, if a frame is selected execution is continued until the selected frame becomes the current execution frame.

Step To

Continue execution of the current stopped thread until the specified source position has been reached.

Find Threads

Find all the threads that exist in the program or current process. The debugger updates its state so that it knows about all of the debuggable threads. This operation is intended for use when the thread monitors have not been set or threads have deadlocked.

Analyse

Put the debugger into analyse mode for *post-mortem* debugging. T2/T4/T8-series transputer target hardware is reset with analyse high and T9000-series transputers are Halted. This halts all low priority threads at the next *deschedule point* and preserves the state. Memory is copied to the host and exploratory code is then loaded onto the hardware so that the state can be interrogated.

Restart

Reboot and restart the program being debugged. A pop-up dialog box will appear asking for confirmation. If the action is confirmed then the hardware will be rebooted and the program being debugged will be reloaded and restarted as if the last `irun` command had been re-entered. All debugger windows being displayed will be kept open but will return to the initial window state.

2.9.3 Events menu

Of these operations, only **Last Event** is available during post-mortem debugging.

Last Event

Switch context to the thread in the program that stopped most recently. A thread will stop executing when an ordinary debug event occurs (i.e. a breakpoint, watchpoint, or thread monitor event) and there is a null or true `when` condition for that event, or when an **Interrupt** or **Step** completes, or Program error or Process exit occurs.

Whenever a thread stops executing an appropriate message is output to each output window and its identity is recorded globally. **Last Event** uses this global value to change the context of the current window to the most recently stopped thread.

Break

Set a breakpoint at the selected source statement or instruction of a stopped process, thread or frame. A source statement can be selected by clicking on a line of source code in the code window or alternatively by highlighting the name of a function that is defined in a source module. An assembly level instruction can be selected by highlighting the address of the instruction.

If the breakpoint is successfully set, a breakpoint marker symbol, ⊘ on Suns or ⚠ on PCs, will appear next to the appropriate statement and the event number of the breakpoint (to be returned when the breakpoint occurs) will be displayed in the output window. If the breakpoint could not be set then an error message will be appear in the output window. A breakpoint set on a thread is deleted when the thread dies. A breakpoint set at frame level is removed when the frame returns.

Breakpoints may be deleted using the **Delete** operation and may be examined, enabled, disabled or deleted using the **List Breakpoints** operation.

Watch

Set a write watchpoint on the selected variable of a stopped process, thread or frame. A variable can be selected by highlighting any reference to it in the source code. If the selected variable is a static then one watchpoint will be set. At process level, each present and future instantiation of an automatic variable will have a watchpoint set on it. At thread level, selecting an instantiated C automatic variable or occam variable will cause a watchpoint to be set on all existing and all future instantiations. At frame level with a frame selected, the watchpoint will be set only on the instantiation in that frame. Selecting a C automatic variable or occam variable which is not instantiated will cause a watchpoint to be set on all future instantiations. All watchpoints set on a thread are deleted when the thread dies. At frame level, the watchpoint will be removed when the frame has finished. A watchpoint set on a C automatic or occam variable is removed when the variable passes out of scope and is removed.

If the watchpoint is successfully set then a watchpoint marker symbol, ◷ on Suns or ⌚ on PCs, will appear next to the definition of the variable and the event number of the watchpoint (to be returned when the watchpoint occurs) will be displayed in the output window. If the watchpoint could not be set then an error message will appear in the output window.

Watchpoints may be deleted using the **Delete** operation and may be enabled, disabled or deleted using the **List Watchpoints** operation. A read or read/write watchpoint may be set using the command language command `watch`.

Internal channels and external channels may have watchpoints set on them. Virtual channels cannot have watchpoints set on them.

Watch Once

Set a temporary write watchpoint on the selected variable of a stopped process, thread or frame, cancelling the watchpoint after the first hit. Otherwise, this operation is the same as **Watch**.

Delete

Cancel a selected breakpoint or watchpoint. The breakpoint or watchpoint is selected by selecting the code line that has the breakpoint or watchpoint marker. Breakpoints and watchpoints may be set using the **Break** and **Watch** operations and may be enabled, disabled or deleted using the **List Breakpoints** and **List Watchpoints** operations.

Thread Birth Monitor

Set monitoring of the generation of new threads. Any new thread in the program or process will stop as soon as it is created. Thread monitors may be deleted, enabled or disabled using **List Thread Monitors**.

Thread birth monitoring is only available for threads written in ANSI C.

Thread Death Monitor

Set monitoring of the termination of threads. Information will be displayed about any thread in the program or process that is terminated. Thread monitors may be deleted, enabled or disabled using **List Thread Monitors**.

List Breakpoints

List in a pop-up dialog box all the breakpoints that have been set in the current process or thread. A breakpoint may be deleted, enabled or disabled by selecting it within the list, selecting the required operation and then clicking on the **Apply** button. Clicking on the **Cancel** button will close the pop-up window. Clicking on the **OK** button is the same as clicking on **Apply** then **Cancel**.

Breakpoints may be set using **Break** and deleted using **Delete**.

List Watchpoints

List in a pop-up dialog box all the watchpoints that have been set in the current process or thread. A watchpoint may be deleted, enabled or disabled by selecting it within the list, selecting the required operation and then clicking on the **Apply** button. Clicking on the **Cancel** button will close the pop-up window. Clicking on the **OK** button is the same as clicking on **Apply** then **Cancel**.

Watchpoints may be set using **Watch** and deleted using **Delete**.

List Thread Monitors

List in a pop-up dialog box all the thread monitors that have been set on the current process or thread. A thread monitor may be deleted, enabled or disabled by selecting it within the list, selecting the required operation and then clicking on the **Apply** button. Clicking on the **Cancel** button will close the pop-up window. Clicking on the **OK** button is the same as clicking on **Apply** then **Cancel**.

2.9.4 Variables menu

Print

Display the value of a variable or simple expression. If a variable or simple expression is highlighted in the code window then the value of that variable or expression is displayed. Highlighted expressions may include any of the following constructs:

- variable, e.g. `abc`
- constant subscript, e.g. `x[0]`
- variable subscript, e.g. `x[i]`
- dereferenced pointer, e.g. `*p`
- address of a variable, e.g. `&y`
- fields of structures and unions, e.g. `z.angle`
- fields of dereferenced structures and unions, e.g. `p->angle`.

If nothing is highlighted in the code window a pop-up window will appear that can be used to type in a variable name or expression. Expressions may be entered in a simple C-like expression language. The expression language is the same as that used to specify a variable in the command language, as described in section 3.1.

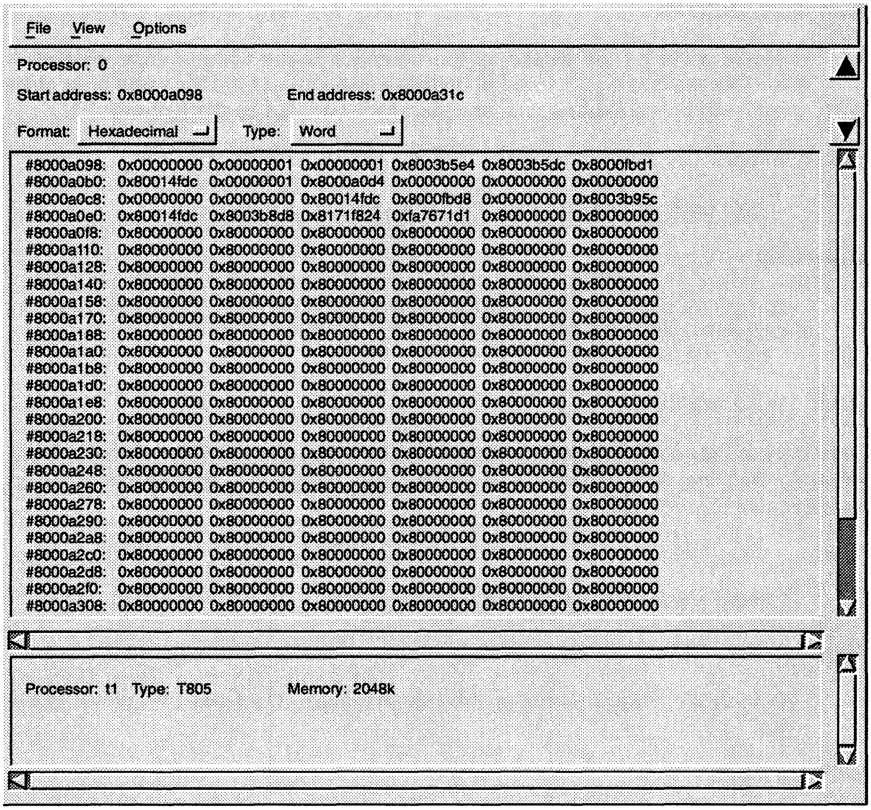
The value of a C process static variable may be displayed at process level or below in any process. The value of a C automatic variable or occam variable can only be displayed in a stopped or stopping thread at thread level. At frame level, the value of a specified C automatic or occam frame variable may be displayed.

In occam code, the index variable of a replicated constructor is not in scope in the line which contains the constructor. If the variable is not used elsewhere, then the **Print** operation cannot be used to display its value. However, the `print` command can be used, as it does not require the variable name to appear in the text.

If an address of a variable is used at assembly level, then decimal numbers (i.e. literals not preceded by #, % or 0x) are treated as word offsets from the Wptr.

Print *

Display the value stored at the location given by a highlighted pointer variable or expression. The pointer variable or expression must be highlighted. Otherwise, **Print *** behaves like **Print**.



Processor: 0

Start address: 0x8000a098   End address: 0x8000a31c

Format: Hexadecimal   Type: Word

```
#8000a098: 0x00000000 0x00000001 0x00000001 0x8003b5e4 0x8003b5dc 0x8000fbd1
#8000a0b0: 0x80014fdc 0x00000001 0x8000a0d4 0x00000000 0x00000000 0x00000000
#8000a0c8: 0x00000000 0x00000000 0x80014fdc 0x8000fbd8 0x00000000 0x8003b95c
#8000a0e0: 0x80014fdc 0x8003b8d8 0x8171f824 0xfa7671d1 0x80000000 0x80000000
#8000a0f8: 0x80000000 0x80000000 0x80000000 0x80000000 0x80000000 0x80000000
#8000a110: 0x80000000 0x80000000 0x80000000 0x80000000 0x80000000 0x80000000
#8000a128: 0x80000000 0x80000000 0x80000000 0x80000000 0x80000000 0x80000000
#8000a140: 0x80000000 0x80000000 0x80000000 0x80000000 0x80000000 0x80000000
#8000a158: 0x80000000 0x80000000 0x80000000 0x80000000 0x80000000 0x80000000
#8000a170: 0x80000000 0x80000000 0x80000000 0x80000000 0x80000000 0x80000000
#8000a188: 0x80000000 0x80000000 0x80000000 0x80000000 0x80000000 0x80000000
#8000a1a0: 0x80000000 0x80000000 0x80000000 0x80000000 0x80000000 0x80000000
#8000a1b8: 0x80000000 0x80000000 0x80000000 0x80000000 0x80000000 0x80000000
#8000a1d0: 0x80000000 0x80000000 0x80000000 0x80000000 0x80000000 0x80000000
#8000a1e8: 0x80000000 0x80000000 0x80000000 0x80000000 0x80000000 0x80000000
#8000a200: 0x80000000 0x80000000 0x80000000 0x80000000 0x80000000 0x80000000
#8000a218: 0x80000000 0x80000000 0x80000000 0x80000000 0x80000000 0x80000000
#8000a230: 0x80000000 0x80000000 0x80000000 0x80000000 0x80000000 0x80000000
#8000a248: 0x80000000 0x80000000 0x80000000 0x80000000 0x80000000 0x80000000
#8000a260: 0x80000000 0x80000000 0x80000000 0x80000000 0x80000000 0x80000000
#8000a278: 0x80000000 0x80000000 0x80000000 0x80000000 0x80000000 0x80000000
#8000a290: 0x80000000 0x80000000 0x80000000 0x80000000 0x80000000 0x80000000
#8000a2a8: 0x80000000 0x80000000 0x80000000 0x80000000 0x80000000 0x80000000
#8000a2c0: 0x80000000 0x80000000 0x80000000 0x80000000 0x80000000 0x80000000
#8000a2d8: 0x80000000 0x80000000 0x80000000 0x80000000 0x80000000 0x80000000
#8000a2f0: 0x80000000 0x80000000 0x80000000 0x80000000 0x80000000 0x80000000
#8000a308: 0x80000000 0x80000000 0x80000000 0x80000000 0x80000000 0x80000000
```

Processor: t1   Type: T805   Memory: 2048k

Figure 2.6 Displaying memory contents using X-Windows

Memory

Display the contents of a segment of the memory of the processor on which the currently selected process is running. A window appears, similar to figure 2.6. Three pull-down menus are given, plus two buttons to allow the user to select the format of the display (e.g. hexadecimal or ASCII characters) and the type of the data (e.g. the size of the data objects).

Jump

Change the browser state to the thread waiting for a specified channel which is in scope. The channel variable must be specified by highlighting the name of the channel variable in the source text.

2.9.5 Options menu

Command Buttons

Toggle on or off the display of the operation buttons in the code window. In X-Windows displays, when the buttons are not displayed the code area of the code window is expanded to use the space vacated by the buttons.

Line Numbers

Toggle on or off the display of line numbers in the code window.

Assembly

Toggle the code window between displaying the source code and displaying assembly code with the workspace of the thread.

2.9.6 Windows menu

This menu is only available on Microsoft Windows and is similar to Windows menus on other applications. Apart from the three operations listed below, the code windows are listed so that any one can be selected.

Tile

Tile the code windows so that they are all fully visible.

Cascade

Overlay the code windows so they appear stacked with the current window on top.

Arrange Icons

Line up the code window icons.

3 Debugger command language

The debugging command language provides an alternative interface to the debugger which complements the button and menu operations. The button and menu operations are described in chapter 2. Command language uses include:

- the definition of complex conditional events;
- the display of specific parts of structured variables;
- the creation of scripts that enable debugging scenarios to be quickly reached;
- the generation of customized debugging commands.

Programming constructs are provided, as described in chapter 4. Commands may be typed directly into the command window or sequences of commands written in a file may be used for customizing the debugger as macros or start-up routines.

The formal definition of the syntax of the command language is given in Appendix A.

3.1 Specifying an object

There are several types of item that need to be specified as arguments to the debugger commands, namely statements, processors, processes, threads, frames, symbols, variables and addresses. Each of these items and the syntax used to denote the item is defined in this section.

3.1.1 Specifying a statement

A *statement reference* is an identification of a unique statement in the source text. A statement reference consists of a file specification, a list of line numbers and, for C programs, an optional statement position within the line, all separated by spaces. A statement reference is enclosed by chevrons (<>).

<*file_specification line_number_list [statement_position]*>

The *line_number_list* is a list of one or more line numbers separated by commas (`,`). The first line number is a line number within the file given by *file_specification*. All but the last line number are the line numbers of `#include` or `#INCLUDE` statements giving the file to which the following line number refers. The *statement_position* is zero for the first statement on the line, 1 for the second statement and so on.

For example, in figure 3.1, `<app.c 13>` specifies line number 13 of the file `app.c`, which states `x[0] = 76;`. Similarly, `<app.c 16 2>` specifies the third statement (statement 2) of line 16, i.e. the statement `x=7`.

Line 12 of `app.c` is a `#include` statement referring to the file `incode.h`. If `incode.h` contains the code shown in figure 3.2. Then `<app.c 12,` means the file `incode.h`, so `<app.c 12,3 1>` specifies the second statement (statement 1) on line 3 of `incode.h`, that is the statement `x=45`.

3.1.2 Specifying a processor, process, thread or frame

A processor is specified by the processor identifier, or `procid`, which is an integer. The current processor is the processor on which the current process is executing. The processor identifier is displayed in the attribute window.

```
 7  void fn() {
 8    int static x[40];
 9    x[0] = 43;
10    {
11      static int x[20];
12      #include "incode.h"
13      x[0] = 76;
14      {
15         static int x;
16         x = 3; x = 5; x = 7; x = 8;
17      }
18
19      {
20         static int x;
21         x = 87;
22      }
23    }
24  }
```

Figure 3.1 Part of file `app.c`

```
1 {
2   int x;
3   x = 24; x=45; x=98;
4 }
```

Figure 3.2 File `incode.h`

A process is specified by the process identifier or `pid`, which is an integer. The pid may be derived from the process name by using the `id` command, so `id app` would return the process identifier of the process `app`. The process name is defined in the configuration code. Using zero as the process identifier specifies all the processes in the program.

A thread is specified by the thread identifier or `tid`, which is an integer. The thread identifier is displayed by the browser and is displayed in the attribute window when the thread is selected. Using zero as the thread identifier specifies all the threads in a process.

A frame is specified by a frame identifier or `fid`, which is an integer. The frame identifier is displayed in the attribute window when the frame is selected. Using zero as the frame identifier specifies all the frames in a thread.

3.1.3 Specifying a symbol

A symbol is a name or identifier, i.e. the name of a variable, channel, function or procedure. A symbol may be specified either by a symbol reference or a nested name. Either method may be used to refer to a unique symbol in the source text.

A *symbol reference* uses source file line numbers. It consists of a statement reference, a symbol name and, optionally, an occurrence within the line. A symbol reference is enclosed by chevrons (<>).

<file_specification line_number_list [statement_position] symbol>

The *file_specification*, *line_number_list* and *statement_position* are the same as for statement references.

For example, in figure 3.1:

`<app.c 8 x>` refers to the symbol **x** declared on line 8 of `app.c`.

`<app.c 11 x>` refers to the symbol **x** declared on line 11 of `app.c`.

`<app.c 12,2 x>` refers to the symbol **x** declared on line 2 of `incode.h`, which is included on line 12 of `app.c`.

`<app.c 15 x>` refers to the symbol **x** declared on line 15 of `app.c`.

`<app.c 20 x>` refers to the symbol **x** declared on line 20 of `app.c`.

A *nested name* is an alternative to a symbol reference. A nested name specifies a symbol by means of a list of nested *scopes* that must be traversed to find a symbol.

A scope is shown in C source code by a pair of braces (`{ }`). In occam source code, a scope is an occam process, i.e. a single statement or a construction. A scope may be named if it is the executable part of a function or procedure, in which case its name is the name of the function or procedure.

The scopes are separated by commas (`,`). The first scope is a file and the last is the symbol. Each unnamed scope is referred to by its position in the scope in which it is nested; the first scope is 1, the second is 2 and so on. A nested name is enclosed by chevrons (<>).

<file_specification, [scope_list,] symbol>

For example, in figure 3.1:

`<app.c,fn,x>` refers to the symbol **x** declared at the beginning of the function `fn`, i.e. the symbol **x** on line 8.

`<app.c,fn,1,x>` refers to the symbol **x** declared at the top of the first set of braces within the function `fn`, i.e. the symbol **x** on line 11.

`<app.c,fn,1,1,x>` refers to the symbol **x** declared at the top of the first set of braces within the first set of braces within the function `fn`, i.e. at the top of the set of braces within `incode.h`, i.e. the symbol **x** on line 2 of `incode.h`.

`<app.c,fn,1,2,x>` refers to the symbol `x` declared at the top of the second set of braces within the first set of braces within the function `fn`, i.e. the symbol `x` on line 15.

`<app.c,fn,1,3,x>` refers to the symbol `x` declared at the top of the third set of braces within the first set of braces within the function `fn`, i.e. the symbol `x` on line 20.

3.1.4 Specifying a variable

A *variable reference* is a means of referring to a simple variable or part of a data structure. It consists of an optional type expression and a reference to a symbol. A symbol can be denoted simply by using the name of the variable or by using a symbol reference or a nested name. A simple symbol may be used to refer to a program variable in the currently selected program process, thread or frame.

[type] symbol

[type] symbol reference

[type] nested name

C-like syntax may be added to dereference pointers and specify the address of a variable, an element of an array or a field of a structure. For example:

`*x` refers to the location pointed to by `x`.

`&y` refers to the address of the variable `y`.

`<app.c 13 x>[5]` refers to element 5 of the subscripted array `x`.

`<app.c 13 x>[i]` refers to element `i` of the subscripted array `x`, where `i` is a program variable in the currently selected program process, thread or frame.

`z.angle.bar` refers to field `bar` of field `angle` of `z`.

`p->angle` refers to the field `angle` of the structure pointed to by `p`.

A segment of an array may also be specified:

`x[5..25]` refers to the segment of the array `x` from element 5 to element 25 inclusive.

`<app.c,fn,x>[i..j]` refers to the array segment from element `i` to element `j` inclusive of the array `<app.c,fn,x>`, where `i` and `j` are program variables in the currently selected program process, thread or frame.

3.1.5 Specifying an address

An address can be specified in any of the following formats:

- A decimal number, e.g. `23`
- An octal number, indicated by a leading zero, e.g. `0765`
- A hexadecimal number indicated by a leading `0x` e.g. `0x5ab67`
- A hexadecimal number with the sign bit set is indicated by `%` at the front, e.g. `%70` means hexadecimal `80000070` on a 32-bit processor. There must not be a space between the `%` and the number.

3.2 Command scope arguments

Many commands have slightly different actions depending on their scope, which may be the entire program, a process, a thread or a frame. These commands have optional process, thread and frame identifier arguments. If one or more of these arguments is supplied then they are said to define the scope of the command. If none of these arguments is given then the scope is defined by the current browser state.

If a command has scope arguments and no arguments are supplied then the browser state is used. If one or more scope arguments are supplied then scope arguments not supplied are assumed to be zero, i.e. to be clear.

3.2.1 Browser state

The state of the browser can be accessed and updated by using and assigning values to the variables given in Table 3.1. See section 4.2 for how to assign values to command language variables. The current values of `pid` and `tid` are shown to the left of the command window.

Variable name	Meaning
pid	the browser process
tid	the browser thread
fid	the browser frame

Table 3.1 Browser state variables

Assigning a positive, non-zero value to `pid` moves the browser to process level. The values of `tid` and `fid` will remain but point to a thread and frame within the process `pid`. Assigning a positive, non-zero value to `tid` moves the browser to thread level. The value of `fid` will remain but point to a frame within the thread `tid`. Assigning a positive, non-zero value to `fid` moves the browser to frame level. When the browser changes levels, `pid`, `tid` or `fid` may be cleared by assigning the value 0. They may be cleared explicitly by a command assigning the value 0.

3.2.2 Effect of command scope

The action of some commands is affected by the scope, either given by the arguments or the browser state. For example, the definition of the `continue` command is `continue` *[pid [tid]]*. `continue` can be used to start all stopped threads of a process or a single stopped thread depending on the scope. If the scope is a process then `continue` will start all stopped threads of the process. If the scope is a thread in a process then `continue` will start that thread.

For example:

> `continue` starts the currently selected browser thread or all the threads in the currently selected browser process.
>
> `pid=4; continue` changes the browser state to process 4 at process level and starts all the threads in process 4.
>
> `pid=4; tid=1; continue` changes the browser state to thread 1 in process 4 at thread level and starts thread 1 in process 4.
>
> `continue 4` starts all the threads of process 4, leaving the browser state unchanged.
>
> `continue 4 1` starts thread 1 of process 4, leaving the browser state unchanged.

3.2.3 Processor identifier

Some commands have the processor identifier as argument. The current processor identifier is displayed in the attribute window. If the processor identifier is not given as an argument then the command uses the value held in the variable `procid`. A value may be assigned to `procid` at any time. The value of `procid` does not affect and is not affected by the browser. The default value of `procid` is zero which is always a valid processor.

3.3 Command descriptions

The command descriptions that follow use `teletype` to define actual values that are to be typed by the user and *italics* to denote values that must be supplied to the command. Options to commands are denoted by a preceding minus sign (-) and may be given in any order. Arguments other than options must be given in the order shown. Commands, arguments and options are all case sensitive.

Some commands return results. If the command is executed as a complete statement then these results are displayed in the output window. If the command returns an integer value, it may be used in an expression, in which case the result is inserted in the expression in place of the command.

3.3.1 Stopping and starting

`quit`

> Exit from the debugger.

`restart` *[-s]*

Reboot and restart the program being debugged. The target hardware will be rebooted and the program being debugged will be reloaded and restarted as if the last `irun` command had been re-run, except that all debug windows being displayed will be kept open but will return to the initial window state. By default, any breakpoints and watchpoints will be deleted. `restart` cannot be called from a script.

`-s` Keep process events (breakpoints, watchpoints and monitors) set on the program.

`analyse`

Put the debugger into analyse mode for *post-mortem* debugging. T2/T4/T8-series transputer target hardware are reset with analyse high and T9000-series transputers are Halted. This halts all low priority threads at the next *deschedule point* and preserves the state. Memory is copied to the host and exploratory code is then loaded onto the hardware so that the state can be interrogated.

3.3.2 Showing and setting state

`threads` *[pid [tid]] [-r]*

Return details of the threads in the scope. The default details are the process identifier, thread identifier, thread origin, processor identifier and process name. If a thread is stopped then the statement reference where the thread is stopped is also returned.

`-r` Return the processor identifier and workspace range of each thread together with the register state if the thread is stopped.

`running` *[pid [tid]] [-r]*

Return details of the running threads in the scope. The default details are the process identifier, thread identifier and the execution position of each thread are returned.

`-r` Return the processor identifier, workspace range and register state of each thread.

`id` *processName*

Return the process identifier number of the named process.

`name` *pid*

Return the process name of the specified process identifier.

`processes` *[pid] [-p]*

Display information about the process.

`-p` Display information about the processor.

3.3.3 Thread control

`continue` *[pid [tid]]*

Continue execution of all stopped threads in the scope.

`step` *[pid tid] [-m] [-t] [-i]*

Start execution of one step of the thread. By default a step is a source statement. Function calls are stepped into unless they are compiled with minimal debugging information (e.g. `printf`) in which case the function call is stepped over. With no debugging information, the thread will continue until it reaches code with debugging information. `Step` returns the event number. `Step` is asynchronous, so in a script, to continue after the step has completed, the script must `wait` until the event has occurred.

In occam, stepping an unreplicated `PAR` statement causes every thread created by the `PAR` to be created and then stop. The parent process remains 'stepping' until all the new processes have been terminated. `stepping` a replicated `PAR` causes all the threads it creates to run until they terminate or hit a breakpoint or watchpoint.

`-m` Step into code that is compiled with minimal debugging information.

`-t` Single step any function that is compiled with minimal debugging until code is encountered that is compiled with full debugging information.

`-i` Step a single machine instruction.

`next` *[pid tid]*

Start execution of one statement of the thread, stepping over function calls. `next` returns the event number. `Next` is asynchronous, so in a script, to continue after the step has completed, the script must `wait` until the event has occurred.

In occam, `next` on a replicated or unreplicated `PAR` statement causes all the threads it creates to run until they terminate or hit a breakpoint or watchpoint.

`stepout` *[pid tid fid]*

Step out to the specified frame. Execution will stop at the first statement encountered of the specified frame. `stepout` returns the event number. `Stepout` is asynchronous, so in a script, to continue after the `stepout` has completed, the script must `wait` until the event has occurred.

`interrupt` *[pid [tid]]*

Interrupt all the debuggable running threads in the scope. Interrupting a low priority thread means that a temporary breakpoint is put before the next instruction to be executed by the thread. High priority threads can only be interrupted when they are in a waiting state, for example waiting for a channel communication to complete. If a thread is waiting to complete an instruction (e.g. a timer wait) when the interrupt is sent then the thread will complete that instruction. When a thread is interrupted while waiting, it is still considered by the debugger to be stopping, so the state of the thread can be examined, but its state cannot be modified until it leaves the waiting state.

3.3.4 Setting, listing and cancelling events

When a breakpoint, watchpoint or thread monitor is set, an event number is returned that can be used to delete or disable the event.

`break` *[pid [tid [fid]]] statement [*`-o`*] [*`-p`*] [*`-s`*]*

Set a breakpoint at the specified statement for all the frames in the scope. *statement* can be a statement reference, the nested name of a function, a symbol reference of a function or an instruction address (see section 3.1). If the breakpoint is successfully set, the event number of the breakpoint is returned.

`-o`	Remove the breakpoint the first time it is hit.
`-p`	Interrupt all debuggable threads of the process.
`-s`	Interrupt all debuggable threads on the processor.

`watch` *[pid [tid [fid]]] varRef [*`-o`*] [*`-r`*] [*`-a`*]*

Set a watchpoint on the specified variable. The default is to set a write watchpoint.

If the specified variable is a C static variable then one watchpoint will be set. If a process identifier only is specified, each present and future instantiation of a C automatic or occam variable in the process will have a watchpoint set on it. If a process identifier and thread identifier are specified, selecting a C automatic or occam variable will cause a watchpoint to be set on al present and future instantiations. At thread level, selecting a C automatic or occam variable that has not been instantiated will cause a watchpoint to be set on each future instantiation of the variable. At frame level, selecting a C automatic or occam variable that will cause a watchpoint to be set on the instantiation of the variable in that frame. At frame level, the watchpoint will be removed when the frame has finished. A watchpoint set on a C automatic or occam variable is removed when the variable passes out of scope and is removed.

Internal channels and external channels may have watchpoints set on them. Virtual channels cannot have watchpoints set on them.

If the watchpoint is successfully set then the event number of the watchpoint is returned.

`-o`	Remove the watchpoint the first time it is hit.
`-r`	Set a read watchpoint.
`-a`	Set an access (i.e. read or write) watchpoint.

`monitor` *[pid] [*`-b`*] [*`-d`*]*

Monitor the specified process for thread terminations and ANSI C thread creations.

`-b`	Monitor only ANSI C thread births.
`-d`	Monitor only thread deaths.

`events` *[pid [tid]] [*`-w`*] [*`-b`*] [*`-m`*]*

List the events set on the all the threads in the scope.

`-w`	Show watchpoint events.
`-b`	Show breakpoint events.
`-m`	Show monitor events.

`delete` *eventNumber*

Delete the event denoted by *eventNumber*, where *eventNumber* has been returned by one of the `events`, `break`, `watch` or `monitor` command.

`disable` *eventNumber*

Disable the event denoted by *eventNumber*.

`enable` *eventNumber*

Enable the event denoted by *eventNumber*.

3.3.5 Examination and update of variables

`print` *[pid [tid [fid]]] varRef [*`-p`*] [*`-u`*] [*`-x`*] [*`-o`*] [*`-c`*] [*`-s`*] [*`-f`*] [*`-a` *val] [*`-j`*]*

Return the value of the variable, channel or data object denoted by *varRef* in the scope.

`-p`	Do not 'pretty print' structured variables.
`-u`	Do not return fields of unions.
`-x`	Display integers in hexadecimal base.
`-o`	Display integers in octal base.
`-c`	Display integers as ASCII characters.
`-s`	Do not display char pointers as null terminated strings.
`-a`	Return no more than the first *val* elements of an array.
`-j`	Set the browser process and thread to the thread waiting for the channel denoted by *varRef*.

`assign` *[pid [tid [fid]]] varRef int*

Update the specified variable reference *varRef* in the scope to the new integer value *int*. See also `modify`.

3.3.6 Stack examination

`where` *[pid tid] [*`-a`*] [*`-r`*] [*`-f` *value]*

Generate a stack trace of the specified thread. A stack trace has for each call a frame number, the name of the function and the position within the function where the call occurred.

`-a`	Return the values of the arguments to the function.
`-r`	Return the values of the **Iptr** and **Wptr**.
`-f`	Return no more than the top *value* frames of the stack.

3.3.7 Low level commands

`alter` *[pid tid] [*`-f` *frame /* `-w` *wptr] [*`-s` *stment /* `-i` *iptr] [*`-a` *areg] [*`-b` *breg] [*`-c` *creg]*

Alter the specified thread to have a new frame, next statement and/or register state.

`-f`	Change the frame identifier to *frame*.
`-w`	Change the workspace pointer to *wptr*.
`-s`	Change the next statement to be executed to the statement reference *stment*.
`-i`	Change the instruction pointer to *iptr*.
`-a`	Change the **Areg** register to *areg*.
`-b`	Change the **Breg** register to *breg*.
`-c`	Change the **Creg** register to *creg*.

If the `f` option is used then the `w` option cannot be used. If the `s` option is used then the `i` option cannot be used. The new value of a register should be an ANSI C integer constant.

`memory` *[processorId] address length [dataType] [format] [*`-l`*]*

Return the memory region from *address* for *length* elements using the specified *dataType* and *format*. The default *dataType* is to return memory as integers that are the same size as the word length of the processor (i.e. 2-byte integers for 16-bit processors and 4-byte integers for 32-bit processors). The default *format* is hexadecimal. The `memory` command returns the contents of the last memory address. If one of the format options `-b`, `-h` or `-w` is used then `memory` may be used in an expression.

`-l`	Display in order of increasing addresses

The supported *dataTypes* are:

`-b`	Bytes
`-h`	2-byte integers
`-w`	4-byte integers
`-g`	8-byte integers
`-f`	4-byte floating point number
`-k`	8-byte floating point number
`-a`	ASCII characters
`-c`	Channels with waiting processes
`-i`	Disassembled instructions

The supported *formats* for displaying numbers are:

`-d` Decimal

`-u` Unsigned decimal

`-o` Octal

`-z` *numcols* Display *numcols* columns

`modify` *[processorId] address [*`-b`*] integer*

Modify the word at the specified address on the processor denoted by *processorId* to the specified integer value. See also `assign`.

`-b` Modify the byte at the specified address to the specified integer value.

`ibreak` *[processorId] address [*`-o`*]*

Set a breakpoint at the specified address on the processor denoted by *processorId.* The address must be an integer. The event number is returned.

`-o` Remove the breakpoint the first time it is hit.

3.3.8 Mapping commands

`statement` *[processorId] address*

Return the statement reference that corresponds to the specified address. See also `addressof`.

`locate` *[processorId] iptr wptr [*`-u`*]*

Return the thread and statement reference that corresponds to the specified **Iptr** and **Wptr** values. This is not the same as the **Locate** operation.

`-u` Update the browser state to the corresponding thread.

`addressof` *[pid] statement*

Give the processor and start address of the specified statement. A *statement* can be a statement reference, a nested name of a function or a symbol reference of a function. The processor identifier is assigned to the variable `procid` and the statement address is returned for use in an expression or assignment. See also **statement**.

4 Command language programming

The debugging command language provides an alternative interface to the debugger which complements the button and menu operations. Single line unconditional commands are described in Chapter 3. This chapter describes the programming constructs.

Commands may be typed directly into the command window or sequences of commands written in a file may be used for customizing the debugger as macros or start-up routines. The formal definition of the syntax of the command language is given in Appendix A.

The debugger command language supports several programming constructs to enable users to create customized debugging functionality. A command language script can be written as ASCII text in a file using a normal text editor and then executed on demand using the `load` procedure typed in the command window.

`load` *filename*

On Sun systems, a script may also be executed automatically on starting up the debugger by placing the script in a file named `.inquestrc` (see Section 4.11).

The formal definition of the syntax of the command language is given in Appendix A.

4.1 Comments

Comments are introduced with two hash signs (##). Anything following the ## up to the end of the line is ignored by the command interpreter.

4.2 Variables

The command language supports the use of integer-valued variables. A variable name consists of a sequence of alphanumeric characters starting with an alphabetic character. A variable is assigned a value using the = operator, e.g. `x = 4` or `y = 16`. The default value of a variable is zero.

The following variables are reserved for use by the debugger: `pid`, `tid`, `fid`, `procid`, `eid`, `etype`.

4.3 Operators

The following "C-like" operations on variables are supported:

- Relational operators: == (is equal to), != (is not equal to), >, >=, <, <=

• Arithmetic operators:	`+`, `-`, `*`, `/`, `%` (remainder), `>>`, `<<`, `++` (increment), `--` (decrement)
• Logical operators:	`&&` (and), `||` (or), `!` (not)
• Bitwise operators:	`&` (and), `|` (or), `^` (exclusive or)

The operands must be separated from the operator by spaces.

4.4 Sequencing

The command language is not free format. Newlines may be inserted only after a command, construct or opening brace (`{`). A set of commands can be sequenced by enclosing the commands in braces. The end of a command is either a newline or a semicolon.

```
{
  command1
  command2
  command3
}
```

and

```
{ command1; command2; command3 }
```

are equivalent.

4.5 Conditional commands

A conditional command can be achieved using an `if` construct. The `if` construct can take an `else` construct.

```
if (condition) command [else command]
```

The condition is true if its value is non-zero.

If a sequence of commands is used then the command in the `if` or `else` construct can start on a new line

```
if (condition) {
  command
  command
} else {
  command
  command
}
```

4.6 Looping commands

Loops can be achieved using a `while` or a `for` construct.

`while` (*condition*) *command*

The `while` construct executes the *command* until the *condition* is zero.

`for` (*command1*; *condition*; *command2*) *command3*

This `for` loop executes *command3* until the *condition* is zero. *command1* is executed once before the *condition* is tested for the first time and *command2* is executed before the *condition* is tested but after *command3* is executed each time around the loop.

4.7 Procedures

Procedures can be defined to enable new commands to be created. The syntax for defining a procedure is:

`proc` *name local_vars* `{` *command1*`;` *command2* `}`

This defines a procedure called *name* which is equivalent to *command1*; *command2*. *local_vars* is a sequence of variable declarations that declare variables that are local to the procedure, so they descope global variables of the same name. For example:

```
proc foobar
   x = 3; y = 2 {
   command1
   command2
}
```

declares `x` and `y` as local variables in the procedure `foobar`.

4.7.1 Arguments and returned values

String type arguments can be passed to a procedure and are referenced using `$1, $2` etc. The number of arguments given to the procedure is available in the pseudo-variable `$#`. All of the arguments can be referenced using the pseudo-variable `$*`. An argument may be specified using the value of a variable; for example if the variable `i` had the value 4, the expression `$(i)` would be equivalent to `$4`.

When an argument is used in an operation that requires an integer (e.g. `x=$4`) the argument is considered to be an integer expression and is evaluated when it is referenced. It is an error to use an argument in an integer operation when the value of the argument is not an integer expression.

Integer values can be returned from procedures by assigning to the pseudo-variable `$$`.

4.7.2 Invoking procedures

A procedure is invoked by the name of the procedure followed by its actual arguments, as follows:

name actualArgument1 actualArgument2 ...

Procedures may be invoked recursively.

Using such calls, a built-in command, such as `break`, can be used in a command language program where a procedure would be used.

Each actual parameter to a procedure is a string. If a string with spaces or tabs is to be passed to a procedure then the string must be enclosed by double quotes ("). If a string is to be evaluated before being passed to a procedure then that string should be bracketed. For example, the value of the variable `k` may be passed to a procedure called `enter` as follows:

```
enter (k)
```

If, however, the string "`k`" is to be passed to `enter` then the command should be:

```
enter k
```

If the string "`(k)`" is to be passed to enter then the command should be:

```
enter "(k)"
```

4.8 Event arrival

A command sequence can be set up to be executed on receipt of an event such as a breakpoint hit. A flag is put on the event to indicate that the command sequence should be executed whenever the event occurs. This is programmed using the `when` construct.

`when` `(`*[eventNumber]*`)` *local* `{` *command1*`;` *command2* `}`

The *eventNumber* is the number returned from setting a breakpoint, watchpoint or monitor event. A `when` construct can be set to be executed on any event by omitting the event number. *local* is a sequence of variable assignments. This declares the variables to be local to the `when` command.

In the command sequence of the `when` construct, the variables `pid`, `tid` and `procid` have values appropriate to the thread that caused the event which caused the `when` construct to be completed. The variable `eid` is set to the event number of the event that has invoked the `when`. The variable `etype` is set to a number that denotes the type of the event, using the encoding shown in Table 4.1.

When an event is deleted, each `when` that is set on the event is removed. A `when` construct returns a number which can be used to remove the `when` but leave the event set.

The `wait` construct causes the debugger to wait until an event occurs.

`wait` *[eventNumber]*

Value of `etype`	Meaning
0	Breakpoint hit
1	Watchpoint hit
2	**Step To** completed
3	About to create a thread
4	Thread has just been created
5	Thread terminated
6	Program debug message
8	Process exit
9	Interrupted
10	Process has just been created
11	Process terminated
12	Error event

Table 4.1 Event type codes

The *eventNumber* is the number returned from setting a breakpoint, watchpoint or monitor event. A `wait` construct can be set to be executed on any event by omitting the event number.

The debugger cannot perform any other actions while a `wait` is outstanding.

4.9 Built-in procedures

The following standard procedures are available.

`pwd`

Display the current working directory.

`cd` *directory*

Change the working directory to *directory.*

`sys` *command*

Send *command* to the shell.

`write` *[*`-n`*] string1 string2 .. stringn*

Display the list of strings, followed by a newline.

`-n` The list of strings is not followed by a newline.

`fwrite` *filename string1 string2 .. stringn*

Write the list of strings to the named file.

`load` *filename*

Execute the script that is in the file *filename.*

`remove` *name*

Remove the variable or procedure *name* from the debugger.

`remove` *[whenNumber]*

Remove the specified when. If no *whenNumber* is specified and the remove is part of a `when` construct then it will remove the enclosing `when` when it has completed execution.

4.10 Example debugging scripts

4.10.1 Example 1

This example creates a synonym, `p`, for the `print` command.

```
proc p {print $*}
```

4.10.2 Example 2

This example creates a command to display the values of several variable expressions for a given process and thread.

```
proc vp
i {
  for (i = 3; i <= $#; i++) {
    print $1 $2 $(i)
  }
}
```

4.10.3 Example 3

Stop thread 1 of process `rndserver` after the tenth hit of a break point at line `<app.c 15>`.

```
p = (id rndserver)
when (break (p) 1 <app.c 15>)
  x = 10 {
    x--
    if (x > 0) {
      continue (p) 1
    } else {
      write process (p) 1 broken
    }
}
```

4.10.4 Example 4

A generalized version of example 3 can be created using a procedure.

```
proc breakafter {
  when (break $1 $2 $3)
    x = $4 {
      x--
      if (x > 0) {
        continue $1 $2
      } else {
        write process $1 $2 broken
        remove
      }
    }
}
```

This can be used as follows:

```
breakafter 5 1 <app.c 15> 10
```

4.10.5 Example 5

Set a watchpoint on a variable until a value has been reached.

```
when (watch 1 1 i) {
  when (step -i) {
    if ((print i) != 8) {
      continue
    } else {
      write i is 8
    }
    remove
  }
}
```

Note the use of instruction stepping, because the watchpoint occurs *before* the variable is updated and the use of `remove` to delete the `step when`.

4.10.6 Example 6

A generalized version of example 5 can be created using a procedure.

```
proc watchfor {
  when (watch $1 $2 $3) {
    when (step -i $1 $2) {
      if ((print $1 $2 $3) != $4) {
        continue
      } else {
        write i is ($4)
      }
      remove
    }
  }
}
```

This can be used as follows:

```
watchfor 1 1 <app.c 87 i> 8
```

4.10.7 Example 7

Count the number of times the procedure `blab` is called by process `rndserver`.

```
when (break (id rndserver) <app.c, blab>) { blabcalls++ }
```

4.10.8 Example 8

Count the number of threads that have been created by a given process.

```
when (monitor -b 6) {
  kids6++
}
```

4.10.9 Example 9

Let a process execute until thread 6 has been created.

```
when (monitor -b 5) {
  if (tid == 6) {
    write "thread 6 is alive"
  } else {
    continue
  }
}
```

Notice that the `continue` refers to the current thread each time that the `when` statement executes.

4.10.10 Example 10

Move up the stack of a process.

```
proc up {
  fid++
}
```

4.10.11 Example 11

Move down the stack of a process.

```
proc down {
  if (fid == 0)
    write can't move down any further
  else
    fid--
}
```

4.10.12 Example 12

Select a current process by name.

```
proc process {
  pid = id $1
}
```

4.11 Start-up scripts

When the debugger starts up on a Sun, the following command language scripts are executed:

1 the system `.inquestrc`,

2 the `.inquestrc` in the home directory and

3 the `.inquestrc` in the working directory.

Command language scripts placed in these files will be executed whenever the debugger starts up.

5 Debugging libraries

INQUEST includes debugging versions of the normal run-time libraries plus some extra ANSI C functions. This chapter describes the use of the extra functions. These functions are provided to:

- assist with debugging (see section 5.1);
- allow interactive debugging of dynamically loaded code (see section 5.2);
- allow interactive debugging of threads started without using the normal processes library (see section 5.3).

All the functions mentioned in this chapter are listed for reference purposes in the appropriate *Language and Libraries Reference Manual*.

5.1 Debugging support library

Three routines are provided as a library to assist with debugging. These provide the functions *stop*, *assert*, and *message*. The routines have different names for each language and are described in more detail in the appropriate *Language and Libraries Reference Manual*. Table 5.1 summarizes the routines.

Function	C function	occam procedure	Description
assert	`debug_assert`	`DEBUG.ASSERT`	If the parameter evaluates to false then stop the process and inform the debugger.
stop	`debug_stop`	`DEBUG.STOP`	Stop the process and inform the debugger.
message	`debug_message`	`DEBUG.MESSAGE`	Insert a debugging message in the program.

Table 5.1 Debug support routines

For ANSI C, these functions are included in the debug libraries `cdebug.lnk` and `cdebugrd.lnk` which are incorporated at link time. The user must `#include` the header file `debug.h` in the source. For occam programs, the library `debug.lib` must be referenced with a `#USE` in the source code and also included as an input to the linker.

The *stop* and *assert* functions are used to stop a process, the latter on the failure to meet a specified condition. Such events are treated by the debugger as if an error had occurred. *assert* and *stop* allow a process to be stopped at any point in the code, where it can then be inspected and possibly resumed. *stop* always stops the process whereas *assert* only stops the process if the parameter evaluates to false or zero.

assert can be used to monitor for unexpected values of variables or to halt a loop at a specific iteration. *stop* can be used to monitor for unexpected branches in the code, such as defaults in switch statements.

The *message* function is used to insert messages that will only be displayed when the program is run under the interactive debugger. Messages are relayed back to the host from any point in the program. It can be used to monitor the activity of outlying processes which are not directly connected to the host.

If the debugger is present then *message* will stop the thread, giving the event type code 6. The command language `when` can be used to resume execution whenever an event type 6 occurs, for example with the command:

```
when () {
   if (etype == 6) {
      continue;
   } else {
      write event occurred ;
   }
}
```

Code stopped by *assert*, *stop* or *message* may be resumed from the line following the call of the debug function using the debugger **Continue** operation.

5.1.1 Examples

The use of the debug support functions in ANSI C is illustrated in the example below:

```
#include <stdio.h>
#include <stdlib.h>
#include <misc.h>
#include <debug.h>

int
main (void)

{
        /* 0 will cause assertion to fail */
         int      x = 0;

         printf ("Program started\n");

         debug_message("A message only within the debugger");

         printf ("Program being halted by debug_assert\n");
         debug_assert (x);

         printf ("Program being halted by debug_stop()\n");
         debug_stop ();

         exit (EXIT_SUCCESS);
}
```

In this example, if `x` is `1` or `TRUE` then the argument to *assert* evaluates to true, so the program continues until it encounters *stop*. If `x` is `0` or `FALSE` then the argument to *assert* evaluates to false and the process stops before it reaches *stop*.

The following is an occam version with a similar structure:

```
#INCLUDE "hostio.inc"
#USE     "hostio.lib"
#USE     "debug.lib"

PROC debug.entry (CHAN OF SP fs, ts, []INT free.memory)
  BOOL x:
  SEQ
    --  FALSE will cause DEBUG.ASSERT to fail assertion test
    x := FALSE

    so.write.string.nl (fs, ts, "Program started")

    DEBUG.MESSAGE ("A message only within the debugger")

    so.write.string.nl (fs, ts, "Program being halted by DEBUG_ASSERT")
    DEBUG_ASSERT (x)

    so.write.string.nl(fs, ts, "Program being halted by DEBUG_STOP()")
    DEBUG.STOP ()

    so.exit (fs, ts, sps.success)
:
```

5.1.2 Action when not debugging

If the application is running without interactive debugging then the debug library procedures have the actions given in table 5.2.

Function	Action
assert	If the parameter evaluates to false then stop the process. Also stops the processor if configured in HALT mode. If true then execution continues.
stop	Stop the process. Also stops the processor if configured in HALT mode.
message	No action.

Table 5.2 Debug routine actions when the debugger is absent

5.2 Dynamic code loading support

An ANSI C process has a single `main` function and so a single entry point and is compiled and linked into a single linked unit. A process consists of a code region and data regions, is normally statically allocated and is uniquely identified by the process name used in the configuration program or a number called its *process identifier* or *pid*. ST20 applications do not normally have more than one process, so a process is generally the same as a program.

Normally INQUEST uses the output from the configuration stage of the build process to find the application processes. If however additional code is dynamically loaded at run-time then that becomes a new process and INQUEST needs to be informed in order to be able to interactively debug it. This section describes the use of the functions `IMSRTL_InformDynamicLoad` and `IMSRTL_InformDynamicUnLoad` so that user applications with dynamically loaded code can be debugged.

A program calling these functions must include the line:

```
#include <dyninf.h>
```

The calling program must be linked with the library `rtlinf.lib`.

5.2.1 Loading processes

For each new process, INQUEST needs to know the start and size of the stack, static, heap and vector space regions, the code entry point, details of any channels and a name for the process. When a dynamic process has been loaded, the function `IMSRTL_InformDynamicLoad` must be used to inform INQUEST. The prototype for this function is:

```
void IMSRTL_InformDynamicLoad (void *code_area_base,
                               int code_area_size,
                               void *stack_area_base,
                               int stack_area_size,
                               void *static_area_base,
                               int static_area_size,
                               void *heap_area_base,
                               int heap_area_size,
                               void *vector_area_base,
                               int vector_area_size,
                               void *iptr,
                               int num_chans,
                               chanDetails *cdetails,
                               char *processname,
                               char *lkuname,
                               int *pidptr);
```

The meanings of the parameters are given in table 5.3. If any of the data regions are not used, a null pointer and a size of zero should be given to the function call.

The details of the channels in the external interface of the process are provided as an array of structures of type `chanDetails` defined by:

```
typedef struct {
  Channel *chan;
  chanDir dir;
  chanType type;
} chanDetails;
```

The channel information for each channel that is passed must define the channel `chan`, the direction of the channel `chanDir` and type of the channel `chanType`. `chanDir` gives the channel direction and whether the channel is a software virtual channel, which is coded by one of the values given in table 5.4.

`chanType` gives the type of channel, which is coded by one of the values given in table 5.5.

Parameter	Description
`void *code_area_base`	The start of the code region
`int code_area_size`	The size of the code region
`void *stack_area_base`	The start of the stack region
`int stack_area_size`	The size of the stack region
`void *static_area_base`	The start of the static region
`int static_area_size`	The size of the static region
`void *heap_area_base`	The start of the heap region
`int heap_area_size`	The size of the heap region
`void *vector_area_base`	The start of the vector space region
`int vector_area_size`	The size of the vector space region
`void *iptr`	The entry point of the code
`int num_chans`	The number of channels in the external interface of the process
`chanDetails *cdetails`	Details of the channels in the external interface of the process
`char *processname`	A name that will be displayed in the INQUEST program browser
`char *lkuname`	The file name of the linked unit that the code region corresponds to
`int *pidptr`	A pointer to an integer where the process identifier will be written

Table 5.3 `IMSRTL_InformDynamicLoad` parameters

Channel direction	Code	Value
Input not on a software virtual channel	`CHANDIR_INPUT`	15
Output not on a software virtual channel	`CHANDIR_OUTPUT`	16
Input on a software virtual channel input	`CHANDIR_VINPUT`	0
Output on a software virtual channel input	`CHANDIR_VOUTPUT`	1

Table 5.4 Channel direction codes

Channel type	Code	Value
Internal channel, implemented by a memory word	`CHANTYPE_SOFT`	0
Hardware link or virtual link channel	`CHANTYPE_HARD`	1
Software virtual channel	`CHANTYPE_VIRTUAL`	2

Table 5.5 Channel type codes

Software implemented virtual channels are channels defined in the configuration between processors that are not IMS T9000 transputers or are not directly connected. They can be identified because they have bit 1 set in the channel word. The channels to and from the host are normally software virtual channels. If a virtual channel is software implemented then its direction must be either `CHANDIR_VINPUT` or `CHANDIR_VOUTPUT`. Other types of channel must have direction either `INPUT` or `OUTPUT`.

5.2.2 Unloading processes

If code is being dynamically loaded, then a process may be overwritten by dynamically loading a new process. In this case, INQUEST must be informed that the old process is to be dynamically 'unloaded', using the function `IMSRTL_InformDynamicUnload`. The prototype of this function is:

```
void IMSRTL_InformDynamicUnLoad (int pid);
```

The process identifier that was returned from the `IMSRTL_InformDynamicLoad` call should be passed to the `IMSRTL_InformDynamicUnload` function as the parameter `pid`.

If only a thread of a dynamically loaded process is being unloaded then the parameter `pid` should be set to `IMSRTL_UnLoadPidMe` to indicate that the thread needs to be unloaded itself. This constant is defined as:

```
#define IMSRTL_UnLoadPidMe (-1)
```

5.2.3 INQUEST behavior with dynamically loaded code

The INQUEST browser assumes that an application has a hierarchical structure, with each thread 'belonging' to a process. The browser identifies which thread belongs to which process by identifying which stack is being used.

This means that if a thread `t` in process `p1` jumps into dynamically loaded code `p2`, then the thread still belongs to process `p1`, even if the code it is executing is part of a different process. Threads created by `t` after jumping into `p2` will be recognized by the browser as part of `p2`. At thread and frame levels, breakpoints, watchpoints and variable printing behave normally.

At process level, breakpoints, watchpoints and variable printing are based on the process to which the code belongs. In the above case, trying to use breakpoints, watchpoints or variable printing from process `p1` on the code in `p2` will result in an error to the effect that the statement does not exist. A breakpoint set at process level in process `p2` can be hit by the thread `t`.

5.3 Dynamic thread creation

The INQUEST debugger needs to be aware of threads of execution in the application program. INQUEST is normally automatically informed of the birth or death of a thread by the library routines `ProcRun`, `ProcPar` etc. This means that there is no need take additional action provided that:

1 the standard thread creation routines (such as `ProcRun`) are used and

2 the user code is linked with a debugging version of the ANSI C run-time library (i.e. `cdebug.lnk` or `cdebugrd.lnk`).

If the application program creates different thread contexts using other mechanisms, then it should explicitly inform the debugger of thread births and deaths. This should be done by calling the functions `IMSRTL_InformThreadBirth` and `IMSRTL_Inform–ThreadDeath`. This section describes these routines and how to use them.

A program calling these functions must include the line:

```
#include <dyninf.h>
```

The calling program must be linked with the library `rtlinf.lib`.

5.3.1 Thread birth

The function `IMSRTL_InformThreadBirth` should be used to inform the INQUEST interactive debugger immediately before a thread is created. If the debugger is not present then no action is taken by this function.

```
void IMSRTL_InformThreadBirth (void *child_iptr,
                               void *parent_wptr,
                               void *lws,
                               void *uws);
```

The parameters comprise the information that needs to be passed to INQUEST before a thread of execution is started.

- `child_iptr` is either the address of the first instruction of the code for the new thread or the null pointer *minint*.

  The `child_iptr` is used by INQUEST for thread monitoring. If `child_iptr` is not the null pointer and thread monitoring has been enabled for the process then INQUEST places a breakpoint in order to stop the thread at its first instruction.

- `parent_wptr` is the **Wptr** of the creating thread.

  `parent_wptr` is used by INQUEST to determine which process that thread belongs to. This **Wptr** must be of a process that is known to INQUEST.

- `lws` is the lowest address of the stack space of the new thread.

- `uws` is the address of the word immediately above the top of the stack space of the new thread.

5.3.2 Death

The function `IMSRTL_InformThreadDeath` should be used to inform the INQUEST interactive debugger immediately before a thread terminates. If the debugger is not present then no action is taken by this function. The prototype for `IMSRTL_InformThreadDeath` is:

```
void IMSRTL_InformThreadDeath (void);
```

5.3.3 Example: creating a thread of execution

In the following example, a thread is created and stopped using assembly inserts. The debugger is informed of the thread creation just before the new thread is started with *runp*. The code of the new thread simply informs the debugger that it is about to terminate and then terminates with *stopp*.

```
{
  int stack[128];
  void *iptr;
  void *wptr;

  __asm {
     ldlabeldiff start - here;
     ldpi iptr;
     here:;
     st iptr;
     ldlp 0;
     st wptr;
  }

IMSRTL_InformThreadBirth(iptr,
                          wptr,
                          stack,
                          (void *)(stack + 128));
  stack[126] = iptr;
  __asm {
     ld stack + 127;
     adc 1;
     runp;
  }

  /* code of main thread */

  main_thread();

  /* code of dynamic thread */

  start:

  IMSRTL_InformThreadDeath();
  __asm {
     stopp;
  }
}
```

5.3.4 Example: installing an ST20450 interrupt handler

In the following example, an interrupt handler is set up using `init_interrupt` and `install_interrupt_handler`. Since the interrupt-handler is implemented as a thread, the debugger must be informed of its creation before it is created. `install_interrupt_handler` does not automatically inform the debugger.

```
{
  void *int2_ws;
  void *wptr;

  init_interrupt();
  int2_ws = malloc(INTERRUPT_WS_SIZE);
  __asm {
     ldlp 0;
     stl wptr;
  }
```

```
    IMSRTL_InformThreadBirth((void *)INT_MIN,
                               wptr,
                               int2_ws,
                              (void *)((int *)int2_ws + INTERRUPT_WS_SIZE));
    install_interrupt_handler(2,
                                interrupt_trigger_mode_any,
                                default_status_register,
                                int2_ws,
                                INTERRUPT_WS_SIZE,
                                interrupt_handler_2);
  }
```

6 Execution analysis

This chapter describes three profiling tools for analyzing the behavior of application programs. The tools provided are the execution profiler `iprof`, the utilization monitor, `imon` and the test coverage and block profiling tool `iline`. The execution profiler estimates the time spent in each function and procedure, the processor idle time and various other statistics. The utilization monitor displays a Gantt chart of the CPU activity of each processor as time progresses. The test coverage and block profiling tool counts how many times each block of code is executed.

The execution profiling and utilization monitoring are performed by profiler kernel processes added to the application program by the configurer. Each tool has a different kernel added by a different configurer option. The monitoring results are stored locally to each target processor, so that the profiling tools have little execution overhead on the application. The presence of the profiler kernel will slow the execution of the program by less than 5%. The profiler kernel processes write the results into the memories of the target processors. When execution is terminated, `imon` or `iprof` is run to extract the results from the target processor memories and display the results of the profiling.

The test coverage and block profiling is performed by code inserted by the compiler. The results are stored in locations in the code locally to each target processor. When execution is terminated, `iline` is run to extract the results from the target processor memories and display the results of the profiling.

For T2/T4/T8-series networks, the host must be able to reset and analyze all the target processors. For T9000-series networks, the control network must be connected to the control port of the host, provided by the interface hardware.

The profiling tools must be used immediately after the application has halted. If the profiling tool fails or the post-mortem debugger has been used then the results have been lost and cannot be extracted. Similarly, the post-mortem debugger cannot be used on the target once the results have been extracted.

6.1 The execution profiler `iprof`

The execution profiler `iprof` provides a post-mortem estimate of how much CPU time has been spent executing each configuration-level process and procedure or function in the program. This is done using sampling techniques.

Since the profiler works using a sampling method, quantization errors are possible, especially if the program does not run for very long. The results should always be treated as estimates. For example, a procedure showing a zero count may have been executed many times but finished too quickly for the profiler to detect. The CPU time does not include time when the process is communicating via a hardware serial link.

The profiler does not profile processes running at high priority, because it is impossible to interrupt a process running at high priority. An estimate is made of the total time taken by all high priority processes. It is good programming practice to keep high priority processes short.

The sampling interval of 1 millisecond cannot be altered by the user.

6.1.1 How it works

The profiler kernel works by interrupting the processor periodically. It then examines the workspace pointer, **Wptr,** and the instruction pointer, **Iptr,** of the interrupted process. It uses the **Wptr** to identify which process was running, and then uses the **Iptr** to find out which section of code was running. The code space of the program is divided into equal blocks of memory, and a counter (a 'bin') is allocated for each block. When the **Iptr** is found to be in a particular block, the counter corresponding to that block is incremented.

The resolution is the size of the code blocks in bytes. In the example in figure 6.1 the resolution is 4, so there is one counter for every 4 bytes of code. A lower resolution means more accurate information but the profiler needs more memory to store the counters. The profiler uses all the free memory area, i.e. the memory left over after the code, static, stack and heap have been allocated. The free memory available determines the number of counters and hence the resolution. The resolution can be increased by decreasing the size of the stack and heap which frees more memory for storing counters.

If there is no process found to be running when the interrupt occurs, then the idle count is incremented.

The above techniques can estimate only low priority process time. High priority processes cannot be interrupted, so if a high priority process is running the profiler must wait. Any high priority processes running or waiting on the scheduling queue ahead of the profiler must complete or be descheduled to wait for a communication or timer before the profiler can run. The profiler records the time it wakes up, i.e. the time at which it is placed on the scheduling queue and would have interrupted if no high priority process had been running. When it starts running on the processor, it compares the current time with the wake time to see if it was delayed by a high priority process. If it was, it then increments the high priority count time by an appropriate amount.

6.1.2 Preparing programs

The program to be profiled should be compiled and linked with `startrd.lnk` or `startup.lnk` in the usual way. The program can be compiled for full or minimal debugging information, i.e. with or without the `G` option for C programs or with or without the `D` option for occam. The profiler will run faster with minimal debugging information.

Programs to be profiled must be configured. The configurer must be invoked with the `-PRE` option.

A process may be marked by the configurer as not to be profiled. This is done by setting the attribute `noprofile` to `TRUE` in the configuration code for the process. When using a C configurer, `icconf` or `inconf`, add the following attribute statement to processes which are not to be profiled:

```
noprofile = TRUE
```

When using the occam configurer, `occonf`, add the following attribute statement to processors which are not to be profiled:

```
set(noprofile := TRUE)
```

The `noprofile` attribute on a process makes the profiler kernel ignore that process. If none of the processes on a processor are to be profiled then a kernel will be placed on that processor but the kernel will not start and there will be no results from that processor. Samples found to be executing `noprofile` processes will be added to the count of **Wptr** misses.

6.1.3 Running `iprof`

To profile an application, the configured and collected bootable program is run as usual. After the program has finished executing or has been interrupted by the user, the program is terminated and the results are extracted and displayed by running `iprof`. `iprof` analyzes the target hardware and must be used immediately after the application has halted.

When `iprof` is run, the compiler output files (`.tco` files) of any code to be profiled should be included on the `ISEARCH` path. On PCs, the `ISEARCH` path may be modified using the Windows launch tool.

On Microsoft Windows systems, `iprof` is a Windows application and is run in the same way as other Windows applications. Two possible methods are:

- Double click on an `iprof` icon in the `inquest` program group in the Program Manager. The command line associated with the icon can be changed by selecting the icon and using **Properties...** in the **File** menu. New icons can be created using **New...** in the **File** menu.
- Select the **Run...** command in the **File** menu of the File Manager and enter the command line.

On all systems, `iprof` can be run with the `-IEX` option to just extract and save the results in a dump file. `iprof` can then be run again with the `-F` option to read the dump file and display the results.

The command line to start `iprof` has the form:

► `iprof` *bootable_file* {*options*}

where: *bootable_file* is the name of the bootable application.

options is a list of one or more options from table 6.1.

> Options may be entered in upper or lower case.
>
> Options can be given in any order.
>
> Options must be separated by spaces.

Option	Description
`-A`	Present the function names in alphabetical order.
`-F` *dumpfile*	Read from the dump file *dumpfile* created using the `-IEX` option.
`-I`	Display information as profiling takes place.
`-IEX` *dumpfile*	Dump the results to the file *dumpfile* to read later using the `-F` option.
`-NA`	Do not assert analyze (Txxx-series transputers only).
`-O` *filename*	Redirect output display to the file *filename.*
`-PZ`	Display functions with zero samples.
`-SC` *filename*	Copy file *filename* to link before extracting. Useful with the skip loaders (Txxx-series transputers only).
`-SL` *resource*	Select the target connection *resource*. Overrides the `TRANSPUTER` parameter.
`-V`	Display information as profiling takes place.

Table 6.1 `iprof` command options

6.1.4 Example `iprof` command line sequences

The following example extracts results from the bootable program `example.btl` and then displays those results. This is the normal command line to profile `example.btl`.

```
iprof example.btl
```

The following example saves the extracted results in the file `profile` and then displays those results.

```
iprof example.btl -iex profile
iprof example.btl -f profile
```

The following example displays the extracted results from `example.btl`, which was running on the network found down link 2 from the root transputer of a Txxx-series network.

```
iprof example.btl -sc skip2.btl
```

6.1.5 Output

This section describes the output from the profiler. An example of profiler output from a C program is shown in figure 6.1.

```
Processor "Root"
Idle time 35.3% (19516)
High time 0.1% (37)
Wptr Misses 0
Iptr Misses 0
Resolution 4

-----------------------------------------------------------------------------
Process "example" (99.9% processor) (35.666s)
Stack 100.0% (35666)    Heap 0.0% (0)    Static 0.0% (0) Vector 0.0% (0)
Function Name                                  | Process  | Processor |Samples
-----------------------------------------------------------------------------
libc.lib/getc                                  |    11.4  |      11.4 |4081
cc/pp.c/pp_rdch0                               |    10.1  |      10.1 |3605
cc/bind.c/globalize_memo                       |     6.9  |       6.9 |2467
cc/pp.c/pp_process                             |     4.3  |       4.3 |1525
cc/pp.c/pp_rdch3                               |     4.2  |       4.2 |1497
cc/pp.c/pp_rdch2                               |     3.9  |       3.9 |1380
cc/pp.c/pp_rdch1                               |     3.8  |       3.8 |1354
cc/pp.c/pp_rdch                                |     3.5  |       3.5 |1252
cc/pp.c/pp_nextchar                            |     3.3  |       3.3 |1189
cc/pp.c/pp_checkid                             |     3.2  |       3.2 |1150
cc/lex.c/next_basic_sym                        |     2.7  |       2.7 |979
libc.lib/strcmp                                |     2.3  |       2.3 |812
libc.lib/DummySemWait                          |     2.2  |       2.2 |784
libc.lib/sub_vfprintf                          |     1.7  |       1.7 |617
```

Figure 6.1 Sample output from the execution profiler

The name of each processor is displayed, together with the following data:

- *Idle time* is defined to be when the processor is not performing any computation, though the processor may be communicating via a link or suspended for some other reason.
- *High time* is an estimate of the amount of time the processor spent in high priority.
- *Wptr Misses* shows the number of samples in which the workspace pointer was pointing outside any of the expected workspace memory areas. This may be an indication that the stack has grown too large or that some processes were marked with the attribute `noprofile` set to `TRUE` in the configuration code.
- *Iptr Misses* shows the number of samples in which the instruction pointer was not pointing to the code segment of the program.
- *Resolution* is the size of the data blocks in bytes, as described in section 6.1.1.

Detailed statistics are then given for each process, i.e. for each C configuration-level process or occam processor. The configuration name of the process is displayed, together with the percentage of CPU time spent on that process and the length of time spent in that process in seconds.

The entries for stack, heap, static and vector space give the relative amount of time that the program was found to be in each area. Threads created in the heap, static and vector space areas will have their time added to the parent.

The statistics are then given for each function or procedure used by the process. The filename is given first, followed by the nested name. In C programs, there will only be one name after the filename. In occam programs there may be any number of nested names as occam supports procedure nesting. This allows each function or procedure to be uniquely identified. The percentage time is given for that function or procedure for that process and for the processor. The final count for that function or procedure is also given as a gauge of the statistical significance of the information.

If no debugging information is available for an object file then the total time spent in that module is displayed. This may happen either because the object file does not include debugging information or because the compiler output file (the `.tco` file) was not found on the `ISEARCH` path. In the case of libraries, if there is a mixture of modules with minimal or full debugging information and modules with no debugging information, then the modules with no debugging information will be gathered together into one entry in the profiling results.

6.2 The utilization monitor `imon`

The utilization monitor, `imon`, provides a post-mortem graphical representation of the utilization of the target processors, showing the blocks of time when each CPU was busy.

The information displayed is derived from sampling the activity of each processor periodically. The sampling period is not controllable by the user. For each sample, the monitor interrupts the CPU and inspects the workspace pointer, **Wptr**, to determine whether the CPU was idle. Any serial links may be busy even when the CPU is idle.

When the application halts, the profiler keeps going, even if all the tasks have terminated. The profiler only halts when the monitor, `imon`, is started, so there may be a large block of idle time at the end of the application. `imon` analyzes the target hardware and must be used immediately after the application has halted.

6.2.1 Preparing programs

The program to be profiled may be compiled with full, minimal, or no debugging information. The utilization monitor can only be used on programs that have been configured. The configurer must be invoked with the `-PRU` option.

If the `noprofile` option is set on *any* process on a processor then the monitor is switched off for that processor.

6.2.2 Running `imon`

The configured and collected bootable program is run as usual. After the program has finished executing or has been interrupted by the user, the program is terminated and the results are extracted and displayed by running `imon`.

On Sun systems, `imon` can be started by a command line.

On PC systems, `imon` is a Windows application and is run in the same way as other Windows applications. Two possible methods are:

- Double click on an `imon` icon in the `inquest` program group in the Program Manager. The command line associated with the icon can be changed by selecting the icon and using **Properties...** in the **File** menu. New icons can be created using **New...** in the **File** menu.
- Select the **Run...** command in the **File** menu of the File Manager and enter the command line.

On all systems, the command line to start `imon` has the form:

► `imon` *bootable_file* {*options*}

where: *bootable_file* is the name of the bootable application.

options is a list of one or more options from table 6.2.

Options may be entered in upper or lower case.

Options can be given in any order.

Options must be separated by spaces.

Option	Description
`-F` *dumpfile*	Read from the dump file *dumpfile* created using the `-IEX` option.
`-IEX` *dumpfile*	Dump the results to the file *dumpfile* to read later using the `-F` option.
`-NA`	Do not assert analyze (Txxx-series transputers only).
`-SC` *filename*	Copy file *filename* to link before extracting. Useful with the skip loaders (Txxx-series transputers only).
`-SL` *resource*	Select target connection *resource*. Overrides the `TRANSPUTER` parameter.
`-V`	Display information as profiling takes place.

Table 6.2 `imon` command options

For X-Windows users, the standard X-toolkit options are supported.

6.2.3 Example `imon` command line sequences

The following example extracts results from the bootable program `example.btl` and then displays those results. This is the normal command line to monitor `example.btl`.

```
imon example.btl
```

The following example saves the extracted results in the file `monfile` and then displays those results.

```
   imon example.btl -iex monfile
   imon example.btl -f monfile
```

The following example extracts results from `example.btl` which was running on the network found down link 2 from the root transputer. It then displays those results.

```
   imon example.btl -sc skip2.btl
```

The following X-Windows example sends the display output to terminal `term`.

```
   imon example.btl -display term:0.0
```

6.2.4 Output

Figure 6.2 shows an example of the output from the utilization monitor.

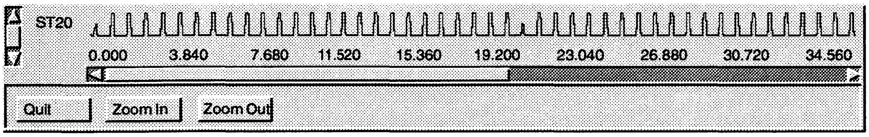


Figure 6.2 Sample output from the utilization monitor

Figure 6.3 shows an example of the output from the utilization monitor for a network of six processors.

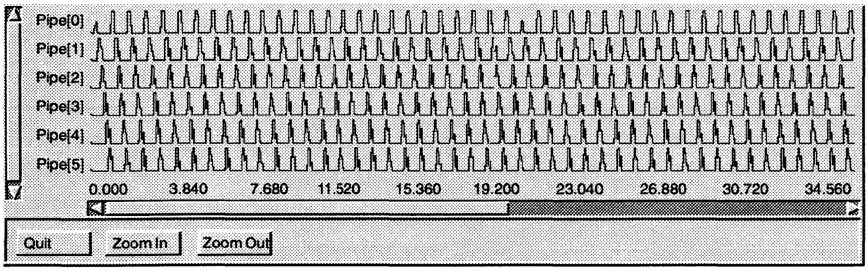


Figure 6.3 Sample output from the utilization monitor

On the vertical axis are the names of the processors. On the horizontal axis is time in seconds. The percentage busy time is represented by the height of the graph. The horizontal scroll bar is used to move along the time line. Clicking on the **Zoom In** button allows more detail to be shown over a shorter time range. The **Zoom Out** button allows less detail to be shown over a longer time range. On X-Windows systems, the user may zoom into a region by holding down any mouse button and dragging a box in the display. The vertical scroll bar scrolls up and down the list of processors.

6.3 The test coverage and block profiling tool iline

The purpose of the `iline` tool is to monitor test coverage and perform block profiling for an application which has been run on target hardware.

This tool is able to:

- provide an overall test coverage report;
- provide per module test coverage reports;
- accumulate a single report from multiple test runs;
- provide a detailed basic block profiling output by creating an annotated program listing;
- provide output that can be fed back into the compiler as a part of its optimization process.

The application program (compiled with the appropriate compiler option) is run and accumulates the counts in the memory of the target processor. The iline tool is used to extract the results and save or display them. The application writes the counts into the code area, so the tool cannot be used with code running from ROM.

6.3.1 Preparing for profiling

Each module that is to be profiled should be compiled with the `-PL` compiler option. Otherwise, the application is built and run on the target hardware in the normal way. The profiling information is stored in the memory of the target hardware as the application runs.

When the application terminates or is halted, `iline` is run to extract and display or save the results. `iline` analyzes the target hardware and must be used immediately after the application has halted.

The `.lku`, `.cfb`, `.btl` files are required by `iline` to be able to produce test coverage and compiler feedback. The directories containing these files must be included on the `ISEARCH` path when `iline` is run. If annotated source output is required then the source files are also required and should be on the `ISEARCH` path.

6.3.2 Command line

► `iline` {*file*} {*options*}

where: *file* is a bootable file or a summation file;

options is a list of one or more options from table 6.3.

Options may be entered in upper or lower case.

Options can be given in any order.

Options must be separated by spaces.

Option	Description
-Q	Suppress information messages.
-NA	Do not assert analyse on the target hardware.
-SL *resource*	Use the target hardware connection *resource*, which is the name of a resource in the AServer database. This overrides the TRANSPUTER parameter.
-SC *bootable_file*	Load *bootable_file* onto the target hardware after any analyzing is done.
-FEED	Produce compiler feedback files.
-TCOV	Produce coverage files.
-MERGE *summation_file*	Merge results into *summation_file*.
-P *processor*	Results from this *processor* only. Multiple -P options may be used.

Table 6.3 iline command line options

6.3.3 ANSI C compiler feedback

When the ANSI C compiler is optimizing, it can make better decisions if it has access to information about which branches are taken most frequently. The -feed option causes iline to create this information, which is written to a file with extension .d. This file can then be given to the compiler on subsequent builds using the -feedback compiler option. For example, for an application app.btl:

```
irun app.btl
iline -feed app.btl
icc -feedback app.d app.c
```

For further details, see the compiler chapter of the *Toolset Reference Manual*.

6.3.4 Test coverage

The -tcov option creates annotated listings of the analyzed modules. For each module one listing, called a *coverage file*, is produced, with the file name extension .v.

For the purposes of this tool, the code is divided into *blocks*. Each block is a section of code with no conditional branches, loops or labels. Every statement in a block is executed the same number of times.

A summary report is written to standard output showing the overall test coverage and the files produced, similar to that shown in figure 6.4. In this report, each module has its coverage shown as a percentage of blocks executed to blocks that exists in the module. An average coverage of all the modules that were appropriately compiled is shown as the last line.

The following are the commands to run the application and then produce a coverage report for each module compiled with -pl, and an annotated listing of each source file:

```
irun app.btl
iline -tcov app.btl
```

```
Writing coverage file "square.v" - 40% coverage
Writing coverage file "comms.v" - 14% coverage
Writing coverage file "app.v" - 75% coverage
Writing coverage file "control.v" - 36% coverage
Writing coverage file "feed.v" - 33% coverage
Writing coverage file "sum.v" - 40% coverage
Total coverage for bootable 39% over 1 run
```

Figure 6.4 Example summary report with the `-tcov` option

If the `-tcov` option is used then the file parameter on the command line may be either a bootable file or a summation file. If the file is a bootable file then new results are extracted from the target and used to generate reports. If the file is a summation file then the results in that file are used to generate reports. Summation files are described in section 6.3.5.

Each coverage file listed in the summary report contains two columns:

1 For each code block a count of the total number of times that block was executed. The counts are shared between all threads that are executing the code.

2 The source code for the block.

At the end of a coverage file is a summary that lists

- information about the module and the test run;
- the 10 most frequently executed blocks;
- the total number of blocks;
- the number of blocks that have not been executed;
- the coverage.

The following is an example of the contents of a coverage file:

```
|/*
| * facs.c
| *
| * generate factorials
| *
| */
|
|#include <stdio.h>
|#include <stdlib.h>
|#include <process.h>
|#include <channel.h>
|#include <misc.h>
|#include "comms.h"
|
                    |#define TRUE 1
                    |#define FALSE 0
                    |
                    |
                    |       /*
                    |        * compute factorial
                    |        *
                    |        */
                    |
                    |int factorial(int n)
96                  |{
                    |  if (n > 0)
74                  |    return ( n * factorial(n-1));
                    |  else
22                  |    return (1);
                    |
                    |}
                    |
                    |int main()
1                   |{
                    |   Channel *in, *out;
                    |   int going = TRUE;
                    |
                    |   in = get_param(1);
                    |   out = get_param(2);
                    |
                    |   while (going)
27                  |   {
                    |      int n, tag;
                    |
                    |      tag = read_chan (in, &n);
                    |      switch (tag)
                    |      {
                    |         case DATA: {
22                  |            send_data (out, factorial(n));
                    |            break;
                    |         }
                    |         case NEXT: {   /* start a new sequence */
4                   |            send_next (out);
                    |            break;
                    |         }
                    |         case END: {    /* terminate */
1                   |            going = FALSE;
                    |            send_end (out);
                    |         }
                    |      }
                    |      }
                    |   }
                    |}
                    |

######################
# Summary of results #
######################
Source file      : facs.c
Number of runs   : 1
Processors       : All
>From linked unit : facs.lku
```

```
Top 10 Blocks!!

Line 25 - 96 times
Line 27 - 74 times
Line 42 - 27 times
Line 29 - 22 times
Line 49 - 22 times
Line 53 - 4 times
Line 34 - 1 time
Line 57 - 1 time

Total number of basic blocks 8
Basic blocks not executed    0
Coverage 100%
```

6.3.5 Summation files

If `iline` is run without the `-tcov` or `-feed` options, then a *summation file* is produced. A summation file contains a binary summary of the results of the application run. The default name for the summation file is the stem of the bootable file name with the extension `.sum`.

For example, the following are the commands to produce a summation file `app.sum`. The application is run, followed by `iline` with the bootable file as parameter:

```
irun app.btl
iline app.btl
```

The results in the summation file may be displayed by running `iline` again using the `-tcov` option with the summation file, for example:

```
iline -tcov app.sum
```

6.3.6 Accumulating results

Summation files may be used to accumulate the results of several runs. If `iline` is run with the `-merge` option then the results will be added to the summation file named in the `-merge` option. If the file does not exist then a new file is created of that name. For example, to add the results of a run to the existing summation file `runtotal.sum` or to save the results into a new file `runtotal.sum`:

```
irun app.btl
iline -merge runtotal.sum app.btl
```

If the `-merge` option is used then the file parameter on the command line may be either a bootable file or a summation file. If the file is a bootable file then new results are extracted from the target. If the file is a summation file then the results in that file are simply accumulated into the summation file given in the `-merge` option.

The following are the commands to use the -merge option to create named summation files run*n*.sum for a sequence of runs, accumulate those results into the summation file runtotal.sum and then display the accumulated results:

```
irun app.btl
iline -merge run1.sum app.btl
irun app.btl
iline -merge run2.sum app.btl
irun app.btl
iline -merge run3.sum app.btl
iline -merge runtotal.sum run1.sum
iline -merge runtotal.sum run2.sum
iline -merge runtotal.sum run3.sum
iline -tcov runtotal.sum
```

6.3.7 Selecting processors

For multi-processor targets, the default is to profile all the processors. The -p option can be used to specify one processor that should be profiled; if several processors are to be profiled then several -p options may be given.

The following are the commands to create annotated coverage for the modules that are placed on the processor t2:

```
irun app.btl
iline -tcov -p t2 app.btl
```

7 Network analyzer

This chapter describes the network analyzer, `rspy`. The network analyzer is used to test the configuration and memory of a transputer network and generate part of a configuration script describing the hardware. It can set connections on an IMS C004 link switch. It also resets a transputer network and clears any error flags. `rspy` is not supplied with IMS T9000 versions of INQUEST.

The network analyzer, `rspy`, is used to test the types of transputers in a network and how they are connected together. It recognizes direct connections between transputers and connections via IMS C004 link switches. When used in this manner `rspy` produces textual output showing the transputer types and their connections.

`rspy` can also be used to generate a hardware description in a form suitable for use in C (`icconf`) and occam (`occonf`) configuration files.

`rspy` can only detect transputers that can be reset by the host.

7.1 Running the network analyzer

7.1.1 Environment variables

An environment variable `ISEARCH` must be set to point to the directory where the `rspy` tool and its associated files are located. This should include a trailing separator ('/' for Suns or '\' for PCs) since the value will be directly pre-pended to the program name.

On Suns, this environment variable is set by running the `setup` script. On PCs it is set by using the `ilaunch` tool from the Program Manager.

7.1.2 Starting `rspy`

On Suns, `rspy` can be started by a command line.

On PC systems, `rspy` is a Windows application and may be run in the same way as other Windows applications. Two possible methods are:

- Double click on an `rspy` icon in a program group in the Program Manager. The command line associated with the icon can be changed by selecting the icon and using **Properties...** in the **File** menu. New icons can be created using **New...** in the **File** menu.
- Select the **Run...** command in the **File** menu of the File Manager and enter the command line.

7.1.3 The `rspy` command line

On all systems, the following command line is used to run the network analyzer:

► `rspy` {*options*}

where: *options* is a list of one or more options from Table 7.1.

> Options may be entered in upper or lower case.
>
> Options can be given in any order.
>
> Options must be separated by spaces.

Option	Description
`-C4`	Give link switch connections in output.
`-CC`	Output `icconf` style hardware description.
`-CL`	Only output link switch connections in switch file format.
`-CO`	Output `occonf` style hardware description.
`-CR`	Reset any link switches that are found.
`-CS` *switch_file*	Set link switches connections from switch file *switch_file.*
`-H`	Display the help page.
`-I`	Display information as `rspy` runs.
`-M`	Do memory sizing.
`-M2`	Memory size IMS T2xx transputers only.
`-M4`	Memory size IMS T4xx and T8xx transputers only.
`-MC`	Do memory sizing including IMS T2xx transputers connected to link switches.
`-ME` *kbytes*	Do memory sizing, stopping after testing *kbytes* of memory per transputer.
`-ML`	Do memory sizing, logging progress of testing.
`-MT` *processor*	Do memory sizing for the processor *processor* only.
`-NR`	Do not reset network at startup.
`-SL` *resourceName*	Use resource *resourceName* rather than that defined in the environment variable `TRANSPUTER`.
`-W`	Show that `rspy` is still working.
`-X` *progname*	Use additional user-supplied `.rsc` code file *progname.*
`-XTO` *seconds*	Set user `.rsc` timeout to *seconds*. Default is 5 seconds.
`-Y`	Do not run part detection.

Table 7.1 `rspy` command line options

The link switch options are described in section 7.3. The memory options (`-M`*) are described in section 7.4.

For X-Windows users, the standard X-toolkit options are supported.

7.2 Network analyzer output

The following sections describe the output from rspy when used without link switch operations or memory testing. The output when memory testing is described in section 7.4.2.

7.2.1 Hardware connection description

To obtain a description of the transputers and their inter-connections use:

```
rspy
```

Figure 7.1 shows an example of the default output produced by running **rspy** on the network in Figure 7.2.

```
# Part-rt   Link0 Link1 Link2 Link3
0 T425-25   HOST   1-1   4-1   4-0
1 T2  -17   C004   0-1   ...  C004
4 T414-20    0-3   0-2   5-1   5-0
5 T2  -20    4-3   4-2   6-1   6-0
6 T800-25    5-3   5-2   7-1   7-0
7 T805-20    6-3   6-2   8-1   8-0
8 T425-25    7-3   7-2   ...   ...
```

Figure 7.1 Sample default output from network analyzing tool

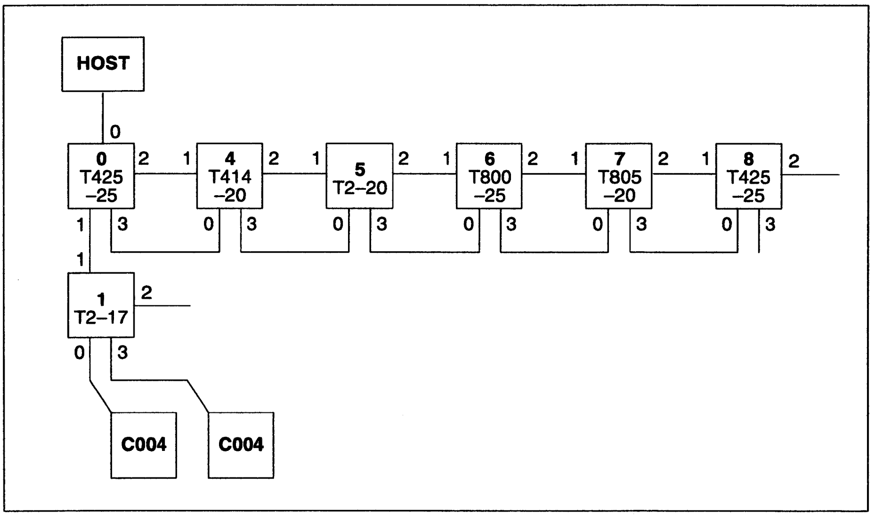


Figure 7.2 Analyzed transputer network

The first line of output contains the following column headings, which are described in Table 7.2:

```
# Part-rt  Link0 Link1 Link2 Link3
```

Each subsequent line of the network description describes a transputer or IMS C004 link switch and its connections to other transputers or IMS C004 link switches or both. Only the configuration links of link switches are listed; a link passing through a link switch is not detectable by the software, although the configuration of the link switch may be listed, as described in section 7.3.

Column Heading	Meaning
#	Node number.
`Part-rt`	INMOS part number and (for transputers) the clock speed in MHz.
`Linkn`	Device connected to Link *n*. See Table 7.3.

Table 7.2 `rspy` output columns

Link description	Meaning
`n-m`	Connected to node *n* link *m*.
`HOST`	Connected to the host.
`C004`	Connected to the configuration link of an IMS C004 link switch.
`...`	Unconnected

Table 7.3 `rspy` output link descriptions

The node numbers are allocated in the order in which the transputers and/or IMS C004 link switches are found. The numbering does not relate to motherboard slots, which are invisible to `rspy`.

The meaning of the link device connections is shown in Table 7.3.

```
/* Hardware description created by rspy */
/* Remember to declare memory sizes */

T425 node0;
T212 node1;
T414 node4;
T212 node5;
T800 node6;
T805 node7;
T425 node8;

connect host to node0.link[0];
connect node0.link[1] to node1.link[1];
connect node0.link[2] to node4.link[1];
connect node0.link[3] to node4.link[0];
connect node4.link[2] to node5.link[1];
connect node4.link[3] to node5.link[0];
connect node5.link[2] to node6.link[1];
connect node5.link[3] to node6.link[0];
connect node6.link[2] to node7.link[1];
connect node6.link[3] to node7.link[0];
connect node7.link[2] to node8.link[1];
connect node7.link[3] to node8.link[0];
```

Figure 7.3 Example C configurer-style output from network analyzer

7.2.2 C configurer-style hardware description

`rspy` can also produce a description of the transputers and their inter-connections in a form suitable for use in an `icconf` configuration hardware description. The output lists the nodes and connections in the correct syntax. The user would need to add memory sizes and define any edges other than the host.

On Suns, the output is written to the standard output, which may be redirected to a file in the usual way. For example, the command line to write an `icconf` script to `net.cfs` would be:-

```
rspy -cc > net.cfs
```

On PCs, the output may be saved by clicking on **Save Buffer...** in the **File** menu. A dialog box will appear, which requests a file name.

Figure 7.3 shows an example of the C configurer-style output.

7.2.3 occam configurer-style hardware description

`rspy` can also produce a description of the transputers and their inter-connections in a form suitable for use in an `occonf` configuration hardware description. The output lists the nodes and connections in the correct syntax. The user would need to add memory sizes and define any edges other than the host.

On Suns, the output is written to the standard output, which may be redirected to a file in the usual way. For example, the command line to write an `icconf` script to `net.cfs` would be:-

```
rspy -cc > net.cfs
```

```
-- Hardware description created by rspy
-- Remember to declare memory sizes

NETWORK
  DO
    SET node0 (type := "T425")
    SET node1 (type := "T212")
    SET node4 (type := "T414")
    SET node5 (type := "T212")
    SET node6 (type := "T800")
    SET node7 (type := "T805")
    SET node8 (type := "T425")

    CONNECT node0[link][0] TO HOST WITH hostlink
    CONNECT node0[link][1] TO node1[link][1]
    CONNECT node0[link][2] TO node4[link][1]
    CONNECT node0[link][3] TO node4[link][0]
    CONNECT node4[link][2] TO node5[link][1]
    CONNECT node4[link][3] TO node5[link][0]
    CONNECT node5[link][2] TO node6[link][1]
    CONNECT node5[link][3] TO node6[link][0]
    CONNECT node6[link][2] TO node7[link][1]
    CONNECT node6[link][3] TO node7[link][0]
    CONNECT node7[link][2] TO node8[link][1]
    CONNECT node7[link][3] TO node8[link][0]
```

Figure 7.4 Sample occam configurer-style output from network analyzer

On PCs, the output may be saved by clicking on **Save Buffer...** in the **File** menu. A dialog box will appear, which requests a file name.

Figure 7.4 shows an example of the occam configurer-style output.

7.3 IMS C004 link switch support

`rspy` can reset link switches and list and set their connections. Link switches may be reset by using the option `CR`. Option `C4` causes `rspy` to list the configured connections of each link switch it finds as part of the normal network display. Option `CL` can be used to write the configured connections of each link switch in the format of a switch file.

`rspy` can set the link connections of one or more IMS C004 link switches in a network. This is requested by the option `-CS` on the `rspy` command line:

```
rspy -cs switch_file
```

The *switch_file* is an ASCII switch file, as described in the sections 7.3.1 and 7.3.2.

7.3.1 Switch file example

This section describes an example of a switch file. The following is an example of a switch file describing the connections for two switches, as shown in figure 7.5:

```
/* Example switch file */

switch
  name    "my_c004_1"
  type    C004
  path    1 0
  connect 1-2 3-4 5-6 7-8 9-10 11-12 13-14 15-16

switch
  name    "my_c004_2"
  type    C004
  path    1 3
  connect 1-2 3-4 5-6 7-8 9-10 11-12 13-14 15-16
```

This switch file relates to the two link switches named `my_c004_1` and `my_c004_2`. Each switch is of type IMS C004. The `path` is the route from the host to the switch configuration link, given by the exit link on each transputer encountered. Thus the configuration link of the switch `board.c004[0]` is connected to link 0 of the transputer connected to link 1 of the root transputer. `connect` lists the switch connections which are required to be set. For each switch in the example, link 1 is to be connected to link 2, link 3 to link 4 and so on, as shown in figure 7.5

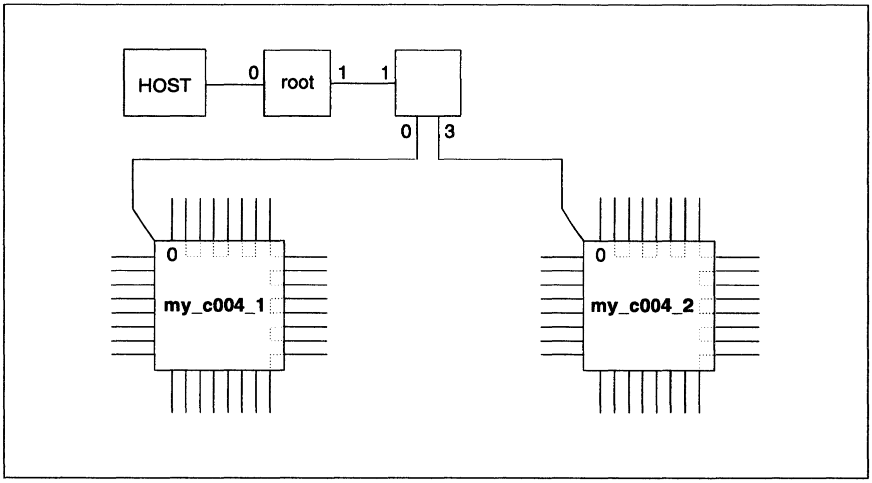


Figure 7.5 Example link switch connections

7.3.2 Switch file syntax

A switch file is an ASCII text file defining one or more switches, where they may be found and the connections set or to be set.

Each separate link switch has a switch declaration block. The first statement of a switch declaration block is a `switch` statement, which has the syntax:

```
switch
```

Each `switch` statement must be followed by a `name` statement, a `type` statement, a `path` statement and a `connect` statement.

`name` statement

`name` *"name"*

The name statement defines a user-defined name (*name*) for the link switch. The name is used in error reporting.

`type` statement

```
type C004
```

The type statement defines the type of the switch. The only type currently defined is `C004`.

`path` statement

`path` *link_number_list*

The path statement tells `rspy` where in the network to find the link switch which is to be programmed. The path *link_number_list* is a list of link numbers which defines a chain of transputers from the host to the configuration link of the link switch. Each link number defines the link from one transputer in the chain to the next, except for the last link number which is the link to the link switch. The first transputer in the chain is the root transputer, which is the first transputer found by the host. Each link number in the list must be in the range 0 to 3. The link numbers must be separated by spaces.

For example, the path `1 0` defines the path from the host to the root transputer, then down link 1 of the root transputer to the second transputer and finally down link 0 to the configuration link of the link switch.

`connect` statement

`connect` *connection_list*

The `connect` statement states the connections to be made by the link switch. The *connection_list* is a list of connections. Each connection is a pair of link numbers separated by a direction. The direction may be either - or >, to specify a connection in both or one direction respectively. Each pair defines a connection between the two links given by the link numbers. For the IMS C004 they must be in the range 0 to 31.

link_number - *link_number* specifies a bi-directional connection, i.e. connecting the input of each of the two links to the output of the other. For example `7-8` connects the input of link 7 to the output of link 8 and the output of link 7 to the input of link 8.

input_link_number > *output_link_number* specifies a connection in one direction only, from the input of the *input_link_number* to the output of the *output_link_number*. For example `7 > 8` connects the input of link 7 to the output of link 8.

Spaces and new lines are not significant in a switch file, and C-style comments style can be used.

7.4 Memory

`rspy` provides memory sizing capabilities for investigating the internal and external memory of the transputers in a network. `rspy` only detects contiguous memory starting at the bottom of the address space. The results of each memory sizing are appended to the default output line for the appropriate transputer. Memory sizing is performed by `rspy` whenever one of the `-M`* options is used, as listed in table 7.1.

Care must be taken when using memory sizing on some boards. This is because on some boards, accessing addresses beyond the top of memory can cause problems, including possibly crashing the host. Use the `-ME` option for such boards to avoid these problems.

The default operation is to size the memory on all transputers that `rspy` has found, except for those T2s that `rspy` has discovered have the control links of IMS C004 link switches connected to their links.

7.4.1 How it works

The memory sizing starts 2 Kbytes above the bottom of the address space (i.e. at #80000800 for 32-bit processors and #8800 for 16-bit processors) and checks whether memory is present every 512 words up the address space. When it finds no memory the sizing stops.

At each test address, `rspy` tests for memory by writing to an address and then reading back and comparing the result with the value written. It also performs a speed test to determine the speed of the memory.

7.4.2 Output

Memory sizing normally adds the information it obtains to the default output from `rspy`. For each transputer that has its memory sized, the results of the sizing are given at the end of the line describing that transputer.

Each memory description lists the blocks of memory followed by a termination character. The descriptions of the memory blocks are separated by plus signs (+) and are listed in the order they are found, starting at the bottom of the memory space. Each block of memory is described by the size of the block (in Kbytes) followed by the letter `K` , a comma, and then the number of memory cycles needed to access that memory.

`rspy` only sizes memory that is contiguous from the bottom of memory.

7.4.3 Termination characters

The following table lists the meanings of the termination characters used at the end of each line of memory description.

Termination character	Meaning
;	Memory wraps around.
|	`ME` option ceiling has been reached.
?	Other error.

Table 7.4 Memory description termination characters

7.4.4 Example memory sizing output

The following is an example of the output from `rspy` when sizing the memory of a network with the `-M` option:

```
# Part rate Mb Bt [ Link0 Link1 Link2 Link3 ] RAM,cycle
0 T800d-25 0.18 0 [  HOST   1:1   3:1   4:0 ] 4K,1+28K,3+2016K,5+4K,3;
1 T2    -20 1.00 1 [  ...   0:1   ...  C004 ]
3 T800c-20 1.75 1 [  ...   0:2   5:1   ... ] 4K,1+1024K,3;
4 T425a-20 1.41 0 [  0:3   5:2   ...   ... ] 4K,1+1024K,3;
5 T800c-17 1.75 1 [  ...   3:2   4:1   ... ] 4K,1+28K,3+2016K,4+4K,3;
```

This shows the description of the memory of processor `0`, a T800d-25, as

```
4K,1+28K,3+2016K,5+4K,3;
```

This indicates that the processor has:

- 4Kbytes of 1 cycle memory (the internal RAM),
- 28Kbytes of 3 cycle memory,
- 2016Kbytes of 5 cycle memory,
- 4Kbytes of 3 cycle memory.

The final semicolon (`;`) indicates that the memory then wraps around.

7.4.5 Command line options

Option `M2` – Size IMS T2xxs only

The `M2` option causes `rspy` to perform memory sizes only on IMS T2xx transputers.

Unless the `MC` option is used, `rspy` does not size the memory of IMS T2xx transputers that have the control links of IMS C004 link switch(s) attached to their links.

Using both the `M2` and `M4` options is allowed and is equivalent to omitting both options. This because the default operation is to size both types.

Option `M4` – Size T4s and T8s only

The `M4` option causes `rspy` to perform memory sizing only on IMS T4xx and IMS T8xx transputers.

Using both the `M2` and `M4` options is allowed and is equivalent to omitting both options. This is because the default operation is to size both types.

Option `MC` – Include T2s with IMS C004 link switches on links

If this option is not used then the memory will not be sized for any IMS T2xxxs that are connected to the configuration links of IMS C004 link switches. The `MC` option forces `rspy` to size the memories of these transputers as well.

Caution: The `MC` option should be used with care, as it may reset any link switches it finds.

Some motherboards have a facility to perform a hardware reset of the IMS C004 link switch or switches on the board; see the board documentation to discover whether the board or boards in use have this facility. On these boards the IMS T2xx that controls the IMS C004 link switch or switches can perform a hardware reset of the IMS C004(s) by writing certain values to any external memory address. Clearly if a memory sizing is performed on such an IMS T2xx then the IMS C004 link switch will be reset thus losing the settings in the IMS C004 link switch. In general this is not desirable.

On the IMS B014 motherboard, using this option will cause the IMS C004 link switches to be reset and so the TRAMs other than slot 0 will no longer be reachable. This may confuse `rspy`.

Option `ME` – Set memory size ceiling

The `ME` option is followed by a decimal integer, which is a ceiling, i.e. the maximum amount of memory, in Kbytes, that will be sized. For each transputer, the memory sizing stops the sizing when the ceiling is reached or the top of memory is found, or an error is detected. If more memory is found then the sizing will be terminated and the pipe symbol (|) will be displayed at the end of the memory description. The default ceiling is 256 Mbytes.

Option `ML` – Log progress

The `ML` option causes `rspy` to log the progress of the memory sizing.

Option `MT` – Sizing a specific transputer

The `MI` option is followed by a decimal number. The number is the number of the transputer, as found by `rspy`, to be sized. Only that transputer is sized. If the `MT` option is used then the `M2`, `M4` and `MC` options are ignored. The `MT` option can be used to size an IMS T2xx that has the configuration links of one or more IMS C004 link switches attached to its links.

7.5 User supplied `.rsc` code

`rspy` can load a user supplied block of code onto each transputer, using the `-x` option. The code might carry out a test and can send a message for display, for example giving the contents of memory locations or the state of a peripheral. The code must be in the form of a `.rsc`-type code file, which is generated by the collector, as described in the *Toolset User Guide* for dynamic code loading. `rspy` looks for a code file with extension `.r2h` to load onto any IMS T2xx transputer or a code file with extension `.rah` to load onto any IMS T4xx or T8xx transputers or both.

For example, the `rspy` command line might be:

```
rspy -x testcode
```

In this case the code in `testcode.r2h` will be loaded onto any IMS T2xxs and the code in file `testcode.tah` will be loaded onto any IMS T4xxs and IMS T8xxs. The appropriate code file is loaded by `rspy` onto each processor and executed. The code is passed, as parameters, the following integer items:

- the number of the parent link, i.e. the link from which the transputer was booted. This link may be used to send a message for display.

- the id number that **rspy** has assigned to the transputer. This number may be used to perform different actions on different transputers.

The code may send a message of up to 512 bytes to the host by writing to the parent link. The message must be preceded by the size of the message as a 16 bit integer. Any message sent down the parent link is routed to the host and displayed in hexadecimal with the **rspy** output at the end of the display line for the transputer that sent it.

The following is an example of message passing code:

```
#INCLUDE "linkaddr.inc"
PROC part(INT parent, my.id)
  [4]CHAN OF INT16::[]BYTE outlink:
  PLACE outlink AT link0.out:

  to.parent IS outlink[parent]:

  VAL buff.size IS 11:
  [buff.size]BYTE buffer:
  SEQ
    buffer :=  [BYTE my.id,1,2,3,4,5,6,7,8,9,10]

    to.parent ! INT16 buff.size :: buffer
:
```

This would produce the following output:

```
rspy
   # Part rate Mb Bt [ Link0 Link1 Link2 Link3 ]
   0 T800d-25 0.18 0 [  HOST   1:1   3:1    ... ] 0 1 2 3 4 5 6 7 8 9 A
   1 T2   -20 1.00 1 [  ...    0:1   ...   C004 ] 1 1 2 3 4 5 6 7 8 9 A
   3 T800c-20 1.75 1 [  ...    0:2   4:1    ... ] 3 1 2 3 4 5 6 7 8 9 A
   4 T800c-17 1.65 1 [  ...    3:2   5:1    ... ] 4 1 2 3 4 5 6 7 8 9 A
   5 T425a-20 1.64 1 [  ...    4:2   ...    ... ] 5 1 2 3 4 5 6 7 8 9 A
```

Appendices

A Debugger command language syntax

The syntax of the debugger command language is defined in this appendix. In the syntactic description terminal symbols are shown in **`Courier bold`** and non-terminals are shown in *italics.* Production rules end with a newline or a comma character (,).

integer ::= a number

string ::= sequence of non blank characters, **"** sequence of characters **"**

variable ::= sequence of alphanumeric characters

terminator ::= **;**, newline

separator ::= a sequence of spaces and tabs

procname ::= a sequence of alphanumeric characters

parameter ::= **$**non-zero positive number
$ (*expr* **)**

result ::= **$$**

args ::= ε, *args separator string,* **(** *expr* **)**

binop ::= **==, =, >, <, >=, <=, +, -, *, /, >>, <<, &&, ||, |, &**

preop ::= **!, -**

postop ::= **++, --**

expr ::= *integer*
variable
parameter
result
(*expr* **)**
expr binop expr
preop expr
expr postop
(*procname args* **)**

vardec ::= ε, *variable* **=** *expr terminator vardec*

cmdlist ::= ε, *cmd terminator cmdlist,* **{** *cmdlist* **}**

cmd ::= *expr*
`proc` *procname vardec cmdlist*

when (*expr*) *vardec cmdlist*
while (*expr*) *cmdlist*
for (*cmd* ; *expr* ; *cmd*) *cmdlist*
if (*expr*) *cmdlist*
if (*expr*) *cmdlist* **else** *cmdlist*
procname args

B Glossary

analyse

Put the debugger into analyse mode for *post-mortem* debugging. T2/T4/T8-series transputer target hardware is reset with analyse high and T9000-series transputers are Halted. This halts all low priority threads at the next *deschedule point* and preserves the state. Memory is copied to the host and exploratory code is then loaded onto the target hardware so that the state can be interrogated.

AServer

A system for portable communication between processes, generally between processes on ST20s or transputers and processes on devices external to the ST20 or transputer network.

attribute window

Debugger: A window of the INQUEST debugger display features of the *process*, *thread* or *frame* currently selected by the INQUEST *browser.*

auto-locate

Debugger: When an *event* occurs or an *operation* is performed on a selected *thread* the windows displaying that thread automatically *locate*.

automatic variable

Debugger: A C variable which is declared inside a function, so multiple executions of the function code give rise to multiple instances of the variable. Any occam variable.

bin

Profiler: A range of code space used by the profiler to identify which part of the program is being executed. A counter of the number of occasions the bin is hit.

block

Profiler: A section of code with no jumps or branches.

block profiling

Profiler: Analyzing how many times each *block* of code has been executed.

breakpoint

Debugger: A point in the code where execution will stop when using the INQUEST debugger. Only the thread which hits the breakpoint will be stopped.

breakpointing transputer

A transputer model with the extra breakpointing instructions, i.e. IMS T225, T400, T425, T426, T450, T801, T805 or later.

browser

Debugger: Part of the INQUEST debugger for navigating through the object hierarchy and selecting a processor, *process*, *thread* or *frame*.

browser window

Debugger: A window of the INQUEST debugger display showing the state of the browser, the state of one or more processes, threads or frames and the browser control buttons.

call stack

Debugger: The stack of current function calls (or *frames*) for the current *thread*.

channel

An unbuffered one-way point-to-point communication path between two tasks or *threads* on the same or different processors.

code window

Debugger: A window of the INQUEST debugger display showing source or disassembled code.

command

Debugger: An instruction to the debugger typed in the command window or executed from a file.

debuggable process

Debugger: Any process not marked as not debuggable in the configuration source, i.e. a process which does not have the configuration attribute `nodebug` set to true.

deschedule point

A point in transputer code where a timeslice or deschedule can occur, such as a loop end, a communication or a timer wait.

dump file

Debugger: A file created by `idump` containing the post-mortem state of an application.

Profiler: A file containing profiling results created by a profiling tool.

dynamic code loading

Loading code during execution of an application, sometimes used to save memory.

event

Debugger: A hit on a *watchpoint*, *breakpoint* or *thread monitor.*

event number

Debugger: The number assigned by the debugger to a future *event* when it is set up which is returned when that event occurs.

execution profiling

See *profiling*.

fid

Debugger: *Frame* identifier. A predefined INQUEST debugger *command* language variable holding the identifier of the current *frame*.

frame

Debugger: The state associated with a function or procedure call.

frame level

Debugger: A *browser* state in which a *thread* is selected. A list of frames in the thread is displayed in the browser window, one of which may be selected. Also displayed are the **Processes** button to go up to the *program level*, the **Threads** button to go up to *thread level* and possibly the **Deselect** button to deselect any selected frame.

free memory

Memory not allocated by the build tools for code or variables, which may be used explicitly by the application.

host

A programmable device capable of resetting, *analyzing* and communicating with the target hardware.

idump

The tool for saving the *post-mortem* state of an application for later debugging.

iline

The *test coverage* and *block profiling* tool.

imon

The *execution monitoring* tool.

INQUEST debugger

A windowing debugger.

inquest

The command to start post–mortem debugging.

interactive debugging

Debugger: Debugging a system while it is running code under the control of the debugger. *See also post-mortem.*

interrupt

Debugger: Stop one or more *threads* after their current instructions.

iprof

The *profiling* tool.

iserver protocol

A protocol for communication between a host and target hardware, used by the i/o libraries.

link

A hardware serial communication port provided on-chip on ST20s and transputers.

AServer: a hardware communication connection between the target and another device using a hardware serial link and any necessary interfacing.

locate

Debugger: To update the code display to show the appropriate source code file or section of disassembled code for the currently selected context.

minint

The lowest possible address, which is #80000000 on 32-bit processors, #8000 on 16-bit processors.

monitoring

See *utilization monitoring.*

See *thread monitoring.*

nested name

Debugger command language: A means of referring to a program identifier or symbol in terms of the program structure.

null pointer

The pointer to *minint* which cannot be a code, process or variable address.

operation

Debugger: An instruction given to the debugger by means of a pull-down menu or one of the operation buttons.

packet

The data sent in a single transmission.

pid

Debugger: *Process* identifier. A predefined INQUEST debugger *command* language variable holding the identifier of the current process.

post-mortem

Debugging: Using the debugger to explore the final state of an application after a fatal error or non-recoverable event. See also *interactive debugging.*

process

A task. A sequential section of code with its own memory and resources running in parallel with the rest of the program.

Debugger and profiler: When debugging or *profiling* C code – a linked unit of code placed on a processor in the configuration. When debugging or *profiling* occam code: all the code on a single transputer.

AServer: A section of code run in parallel with the rest of the program which communicates using the *AServer* protocol.

process level

Debugger: A *browser* state in which a process is selected but not a thread. A list of threads in the process is displayed in the browser window with the **Processes** button to go up to the Program level.

procid

Debugger: Processor identifier. A predefined INQUEST debugger *command* language variable holding the identifier of the current processor.

profiling

Recording how busy each *process*, procedure and function is.

program level

Debugger: A *browser* state in which no *process* is selected. A list of *processes* in the program is displayed in the browser window.

protocol

The format of possible communications.

scope

Debugger: The section of code to which a declaration applies.

service

AServer: A process to which a *client* can open an *AServer connection.*

statement reference

Debugger command language: A means of referring to a program statement in terms of files and line numbers.

stack

Debugger: See *call stack.*

static variable

Debugger: A C variable which is declared as static at *process* level outside any function. It has only one instance within a process and is not associated with a particular *thread.*

step

Debugger: In C or occam, a single statement or part of a statement. In assembly code, a single instruction. To execute a single step.

summation file

Profiler: A file used to hold profiling results in a form suitable for accumulating the results of multiple runs.

symbol reference

Debugger command language: A means of referring to a program identifier or symbol.

test coverage

Profiler: The proportion of blocks of application code executed by the application.

thread

Debugger: A sequential part of a *process* which is running or waiting to run in parallel with the rest of the program.

thread level

Debugger: A *browser* state in which a *thread* is selected but not a *frame*. A list of threads in the process is displayed in the browser window with one selected. The **Processes** button to go up to the *program level*, the **Call Stack** button to go down to *frame level* and the **Deselect** button to go up to the *process level* are also displayed.

thread monitoring

Debugger: The detection by the debugger of the creation or death of threads.

tid

Debugger: *Thread* identifier. A predefined INQUEST debugger *command* language variable holding the identifier of the current thread.

tile

Debugger: To display all the open sub-windows so that they do not overlap.

toolset

A collection of tools for building application programs.

utilization monitoring

Recording when each target processor CPU is busy.

watchpoint

Debugger: A marker on a variable that causes execution to stop when that variable is read or written to. Only the *thread* reading or writing to the variable will be stopped.

Index

SGS-THOMSON MICROELECTRONICS