Technical Report

CMU/SEI-89-TR-032
ESD-TR-89-043

The Durra Application Debugger/Monitor

Dennis L. Doubleday

September 1989

Technical Report

CMU/SEI-89-TR-032
ESD-TR-89-043
September 1989

The Durra Application Debugger/Monitor

Dennis L. Doubleday

Software for Heterogeneous Machines Project

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This report was prepared for the

SEI Joint Program Office
HQ ESC/AXS
5 Eglin Street
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER

(signature on file)

Thomas R. Miller, Lt Col, USAF
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

This work was created in the performance of Federal Government Contract Number F19628-95-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 52.227-7013.

This document is available through SAIC/ASSET: 1350 Earl L. Core Road; PO Box 3305; Morgantown, West Virginia 26505 / Phone: (304) 284-9000 / FAX: (304) 284-9001 / World Wide Web: http://www.asset.com/sei.html / e-mail: webmaster@www.asset.com

Copies of this document are available through the National Technical Information Service (NTIS). For information on ordering, please contact NTIS directly: National Technical Information Service, U.S. Department of Commerce, Springfield, VA 22161. Phone: (703) 487-4600.

This document is also available through the Defense Technical Information Center (DTIC). DTIC provides access to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential contractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center / Attn: BRR / 8725 John J. Kingman Road / Suite 0944 / Ft. Belvoir, VA 22060-6218. Phone: (703) 767-8274 or toll-free in the U.S. — 1-800 225-3842).

Table of Contents

1. Introduction to Durra **1**
1.1. The Durra Language and Method 1
1.2. The Durra Runtime Environment 3

2. The Durra Application Debugger/Monitor **5**

3. Implementation of the Application Debugger/Monitor **7**

4. Application Debugger/Monitor Commands **11**
4.1. Go, Quit, and Ctrl-C Commands 11
4.2. Watch and Break Commands 12
4.3. Kill, Stop, and Resume Commands 14
4.4. Show and Track Commands 15
4.5. Read, Echo, and Silent Commands 18
4.6. Set Commands 19
4.7. Miscellaneous Commands 19

References **21**

Index **23**

List of Figures

Figure 1: Compilation of an Application Description 2
Figure 2: The Durra Runtime Environment 3
Figure 3: The Expanded Durra Runtime Environment 7
Figure 4: Durra Application Example 9

The Durra Application Debugger/Monitor

Abstract. Durra is a language designed to support the construction of distributed applications using concurrent, coarse-grained tasks running on networks of heterogeneous processors. An application written in Durra describes the tasks to be instantiated and executed as concurrent processes, the types of data to be exchanged by the processes, and the intermediate queues required to store the data as they move from producer to consumer processes.

This report describes the Durra application debugger/monitor, a program that works in conjunction with the Durra runtime software to help the developer locate errors and/or performance bottlenecks in a Durra application.

1. Introduction to Durra

1.1. The Durra Language and Method

Durra [3, 1, 4] is a language designed to support the construction of distributed applications using concurrent, coarse-grained tasks running on networks of heterogeneous processors. An application written in Durra selects and reuses *task descriptions* and *type declarations* stored in a library. The application describes the tasks to be instantiated and executed as concurrent processes, the types of data to be exchanged by the processes, and the intermediate queues required to store the data as they move from producer to consumer processes.

Because tasks are the primary building blocks, we refer to Durra as a *task-level description language*. We use the term *description language* rather than *programming language* to emphasize that a Durra application is not translated into object code in an executable (conventional) machine language. Instead, a Durra application is a description of the structure and behavior of a logical machine that is to be synthesized into resource allocation and scheduling directives, which are then interpreted by a combination of software, firmware, and hardware in each of the processors and buffers of a heterogeneous machine. This is the translation process depicted in Figure 1.

There are three distinct phases in the process of developing an application using Durra: the creation of a library of tasks, the creation of an application using library tasks, and the execution of the application.

During the first phase, the developer writes descriptions of the component tasks. A task description specifies some set of properties of the task implementation, including the *ports* through which a task communicates with other tasks, the types of data it produces

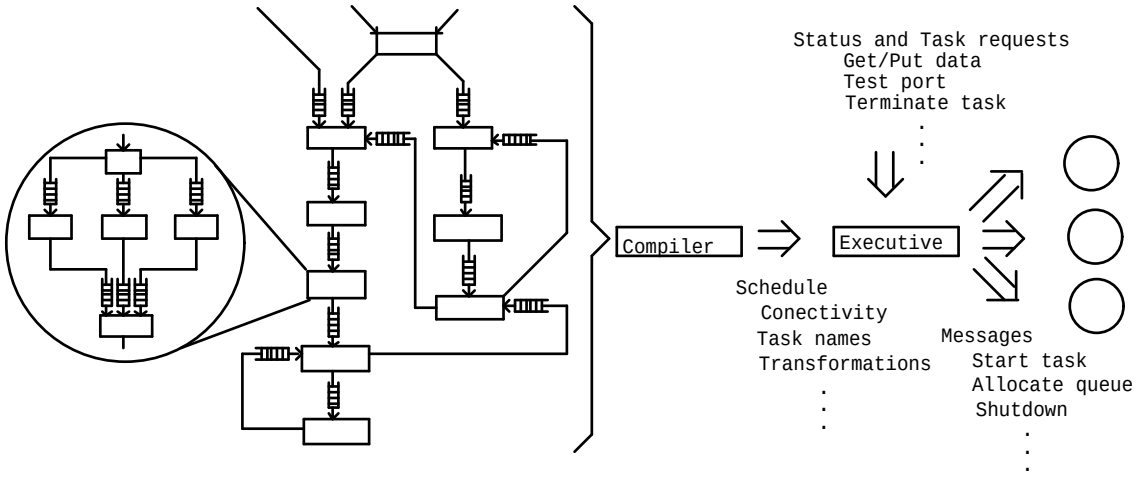


Figure 1: Compilation of an Application Description

or consumes, and other miscellaneous attributes of the implementation. For a given task, there may be many implementations, differing in programming language (e.g., C or assembly language), processor type (e.g., Motorola 68020 or IBM 1401), performance characteristics, or other attributes. For each implementation of a task, a description must be written in Durra, compiled, and entered in the library.

During the second phase, the user writes an application description. Syntactically, an application description is a single task description that can be stored in the library as a new task, allowing for the writing of hierarchical application descriptions. At this level the developer specifies such things as the queues connecting individual ports, required data transformations, and potential reconfigurations (changes in the topology of the application). Compiling the application description generates a set of commands, or instructions, about resource allocation and scheduling to be interpreted by the Durra executive (see Section 1.2).

During the last phase, the executive loads the task implementations (i.e., programs corresponding to the component tasks) onto the processors and issues the appropriate commands to execute the programs.

1.2. The Durra Runtime Environment

This section provides a summary of the Durra runtime environment sufficient for the purpose of understanding the Durra application debugger/monitor's relationship to the other components of the environment. For a more detailed description of the the runtime environment, see [2, 4].

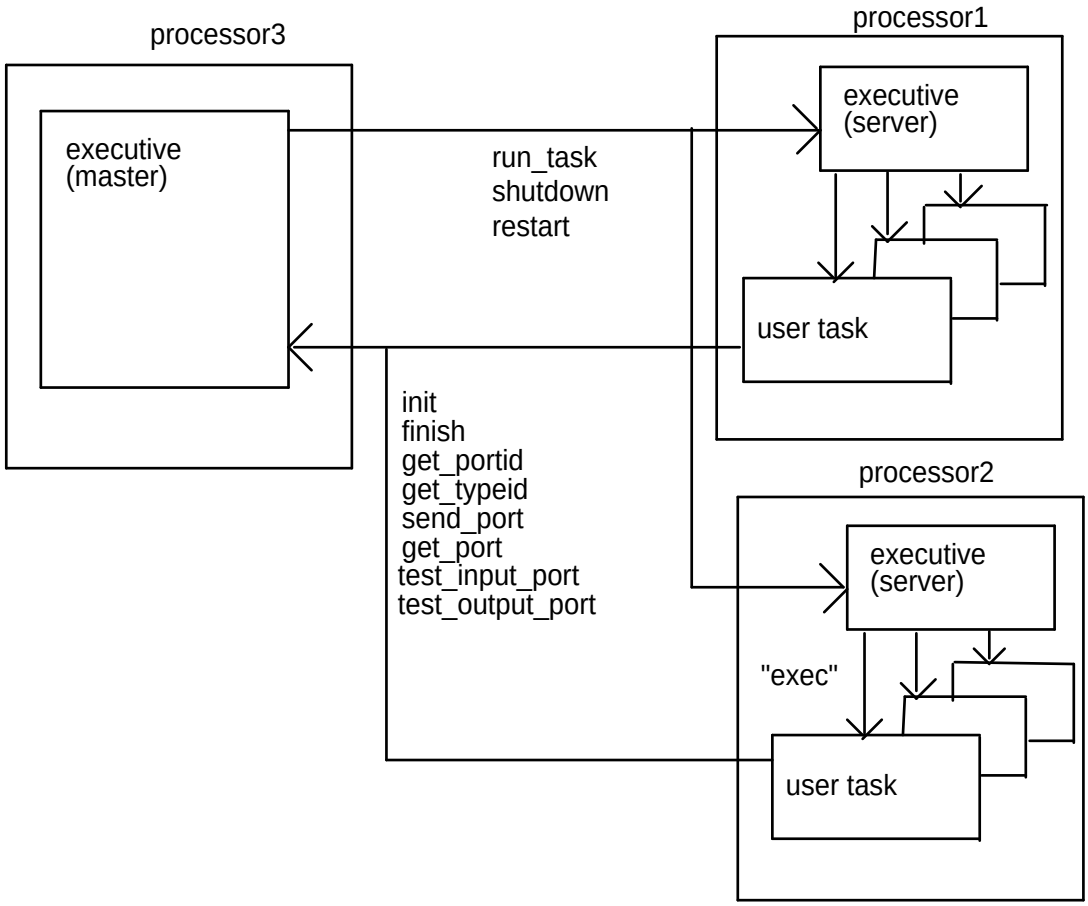


Figure 2: The Durra Runtime Environment

There are two active components of the minimal Durra runtime environment: the application tasks and the Durra executives. Figure 2 shows the relationship between these components. The executives can run in master or server mode. There is one server executive on each processor in the configuration and it is responsible for starting all tasks assigned to that processor. There is one master executive for the entire network and it is responsible for telling the server(s) which tasks to start, establishing communication links, and controlling the execution of the application. The executives implement

the predefined tasks (**broadcast**, **merge**, and **deal**) described in [3]. The executives' specific actions are prescribed by a file containing instructions generated by the Durra compiler.

Once its description has been compiled as described in Section 1.1, an application can be executed by performing the following operations:

1. Load the task implementations into system-defined locations on the processors where they will run.
2. Start an instance of the Durra executive on each processor. All but one of these executives runs in server mode. The remaining executive runs in master mode. The master executive reads instruction file and initiates application execution.

2. The Durra Application Debugger/Monitor

Just as a software developer using a traditional high-level programming language requires a symbolic debugger to efficiently isolate problems in his implementation, so a Durra application developer will need automated assistance to isolate bugs, tune performance, and control the execution of the application. The application debugger/monitor (hereafter referred to as the *monitor*, for brevity) addresses this requirement.

Durra applications are potentially very complex. The Durra developer may face difficulties in debugging and tuning his application that the traditional software developer need not consider, such as distributed concurrent processes, heterogeneous processor architectures, mixed-language programming, dynamic reconfiguration, etc. The fact that Durra hides the details of these complications from the developer is an advantage during development but an impediment during testing. At the testing stage, access to the information about the application state that is encapsulated in the executive, as well as about the state of the individual programs which make up the Durra application, is useful.

The monitor provides this information at two levels, the *application level* and the *source level*. The primary focus of the monitor is at the application level. Here the monitor provides the abstracted Durra view of the world, where the individual tasks are treated as black boxes connected to each other through Durra ports and queues. The user can examine executive-internal data, insert break points on Durra communication interfaces, change task attributes, and do other miscellaneous operations. See Chapter 3 for details.

Obviously, it is also necessary to debug an application at the source level. In a mixed-language, mixed-architecture environment, it is not feasible or desirable for Durra to provide a source-level debugging capability. Instead, the monitor provides a simple mechanism for allowing the developer to use existing source-level debuggers, tools which are widely available and with which the developer is already familiar.

The Durra application developer will also be concerned with tuning the performance of this application. Often it is difficult to isolate a performance bottleneck in one part of the application or another. With this in mind, the monitor provides a tracking facility to monitor the flow of data through the Durra queues. Given this information, the developer may be able to identify the task causing the problem and take corrective actions, such as moving the task to a faster processor or re-implementing the task to improve its efficiency.

Finally, the monitor provides the developer with the means to control the execution of the application, e.g., stepping through the execution, one data transmission at a time, rather than allowing the application to run freely. Such control is especially useful for applications with many specified reconfigurations, because the reconfigurations may be triggered by conditions which are difficult and/or time-consuming to simulate in testing mode. The monitor allows the user to trigger the reconfiguration whether or not the trigger condition has been satisfied.

3. Implementation of the Application Debugger/Monitor

The monitor is an optional component of the Durra runtime environment. The user may invoke it independently at any time during the execution of a Durra application, in which case it can run on any networked processor where its executable image resides. Alternatively, assuming the environment supports the X Window System [6], the monitor may be activated as part of the runtime environment start-up procedure (via an optional argument to the executive command line), in which case it will run on the same processor as the executive. The X support is required in the latter case because a dedicated window must handle the monitor's user interface. *Durra: A Task-Level Description Language User's Manual* [5] (hereafter referred to as the *Durra User's Manual*) describes in detail the start-up process in the Unix environment.

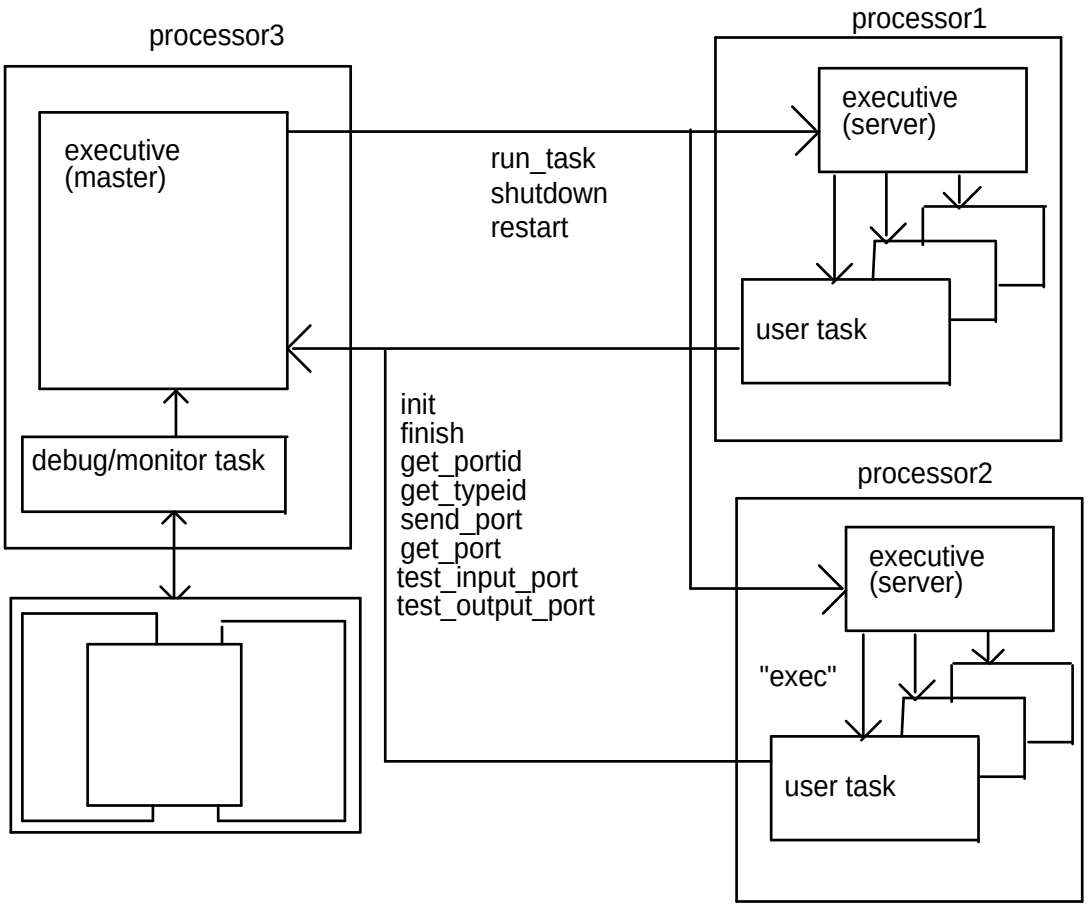


Figure 3: The Expanded Durra Runtime Environment

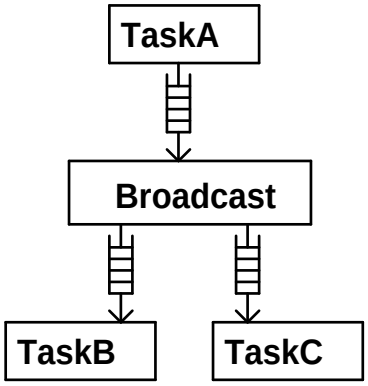
As shown in Figure 3, the monitor exchanges information with the master executive through a remote procedure call (rpc) interface, just as Durra application tasks do. However, the monitor has its own distinct set of rpcs.

The principal processing loop of the Durra executive scans for incoming communications from all Durra tasks, including the monitor if it has been activated. The only monitor rpcs the executive sees at this level are **Init_Monitor** and **Interrupt**; the former call occurs when the monitor starts up, the latter when the user types ctrl-C at the keyboard in order to interrupt execution of the application. On acceptance of either call, the executive transfers control to an alternate processing loop, one in which only calls from the monitor are accepted. This causes all application tasks to block on their ensuing rpcs to the executive. At this point the monitor prompts for commands (see Chapter 4) and processes them, continuing until the command **go** is issued, causing return of control to the main processing loop.

While the executive processes communications from application tasks, the monitor waits for input from either the executive or the keyboard. If the monitor user has requested it, the executive sends the monitor messages indicating the progress of the execution. On an interrupt from the keyboard, control is returned to the monitor as described above. If the interrupt is received while messages from the executive are being processed, then the message processing is completed before the interrupt is recognized.

The **quit** command causes the monitor to terminate. It can be restarted at any time while the executive is active.

To illustrate the use of the monitor, we will use a small Durra application. The example application in Figure 4 consists of two type definitions, a data source task with one output port, two data sink tasks, each with one input port, and an instance of the predefined task **broadcast**, which transmits data received from the source task to each of the sink tasks.



a -- Application Structure

```
type byte is size 8;
type string is array of byte;
```

b -- Type Declarations

```
task taska
    ports
        out1: out string;
    attributes
        processor = vax;
        implementation = "source_task";
end taska;

task taskb
    ports
        in1: in string;
    attributes
        processor = sun;
        implementation = "sink_task";
end taskb;

task taskc
    ports
        in1: in string;
    attributes
        processor = vax;
        implementation = "sink_task";
end taskc;
```

c -- Task Descriptions

```
task main
    structure
        process p1: task taska;
                p2: task taskb;
                p3: task taskc;
                pb: task broadcast
                        ports   in1: in string;
                                out1, out2: out string;
                        end broadcast;
        queues  q1b[10]: p1.out1 >> pb.in1;
                qb2[10]: pb.out1 >> p2.in1;
                qb3[10]: pb.out2 >> p3.in1;
end main;
```

d -- Application Description

Figure 4: Durra Application Example

4. Application Debugger/Monitor Commands

The Durra application debugger/monitor is an interactive process which communicates with the Durra executive at runtime to provide information about and control over the progress of the application.

There are two ways to start the monitor in the Unix environment. To start the monitor when the application begins executing, use an optional flag to the **dexec** command (this requires X Window System support from the environment). To start the monitor after an application has begun executing, use the **dmonitor** command. The details of each method are described in the *Durra User's Manual*.

Commands to the monitor may be entered from the keyboard or from a file (or files) specified by the user. Commands and keywords may be abbreviated to the shortest non-ambiguous initial substring; identifiers must always be complete. This section describes the commands recognized by the monitor.

The following notation is used in the monitor command descriptions:

command or **keyword**
identifier or *literal-value*
''[*a* | *b*]'' means choice of *a* or *b*
''{ *a* }'' means that *a* is an optional argument

The character ''*'' is a wildcard symbol implying all possible values that make sense in the context. In the context of an expected port, queue, task, or type name, the ''*'' expands to all such names in the application currently running. In the context of an expected *rpc-name*, the ''*'' expands to all the Durra interface call names. If an optional argument is omitted, the effect is the same as if the value of the argument were ''*''.

Excerpts from a sample monitoring session on the previously described example Durra application appear throughout the following discussion. In all excerpts, lines beginning with the prompt `monitor>` represent commands and all other lines represent monitor responses.

4.1. Go, Quit, and Ctrl-C Commands

This section describes commands which control the interaction between the monitor and executive.

go { *duration* }

The **go** command tells the executive to continue processing the application for a specified amount of time (in seconds) before prompting the users for more commands. If no duration is specified, the application runs indefinitely. The user can always regain control by typing **ctrl-C** to interrupt the monitor.

quit

The **quit** command ends the monitoring session.

ctrl-C

A **ctrl-C** interrupts the application and prompts the user for a monitor command.

4.2. Watch and Break Commands

This section describes commands which allow the user to follow the flow of data through an application and interrupt the application at any point of communication with the Durra runtime.

dpbreak { [*port-name* **| ‘‘*’’] { [** *rpc-name* **| ‘‘*’’] } }**
dpwatch { [*port-name* **| ‘‘*’’] { [** *rpc-name* **| ‘‘*’’] } }**
dqbreak { [*queue-name* **| ‘‘*’’] { [** *rpc-name* **| ‘‘*’’] } }**
dqwatch { [*queue-name* **| ‘‘*’’] { [** *rpc-name* **| ‘‘*’’] } }**
dtbreak { [*task-name* **| ‘‘*’’] { [** *rpc-name* **| ‘‘*’’] } }**
dtwatch { [*task-name* **| ‘‘*’’] { [** *rpc-name* **| ‘‘*’’] } }**

The above commands are used to delete break points or watch points, which are defined below.

pbreak { [*port-name* **| ‘‘*’’] { [** *rpc-name* **| ‘‘*’’] } }**
pwatch { [*port-name* **| ‘‘*’’] { [** *rpc-name* **| ‘‘*’’] } }**
qbreak { [*queue-name* **| ‘‘*’’] { [** *rpc-name* **| ‘‘*’’] } }**
qwatch { [*queue-name* **| ‘‘*’’] { [** *rpc-name* **| ‘‘*’’] } }**
tbreak { [*task-name* **| ‘‘*’’] { [** *rpc-name* **| ‘‘*’’] } }**
twatch { [*task-name* **| ‘‘*’’] { [** *rpc-name* **| ‘‘*’’] } }**

The above commands are used to set break points and watch points, where a break (or watch) point is defined as a state in the application execution sequence at which a specified Durra object (port, queue, or task) is referenced by one of a specified set of task interface rpcs. When a break point is set and the application reaches the specified interface call on (from) the specified object, the application is interrupted and control passes to the monitor so that the user may issue further commands. When a watch point is set, the monitor informs the user that the watch point has been passed but the application continues to run.

In the following excerpt, the user sets a watch point on the port `main.p1.out1` for all rpcs and then issues the **go** command, causing the application to resume running. The monitor displays a message each time an rpc targeted at `main.p1.out1` occurs. In this case, only `Send_Port` calls are occurring on that port. The messages continue until there is no more activity on that port or until the user interrupts from the keyboard.

```
monitor> pwatch main.p1.out1 *
monitor> go
Watch at port MAIN.P1.OUT1, RPC = SEND_PORT
Watch at port MAIN.P1.OUT1, RPC = SEND_PORT
Watch at port MAIN.P1.OUT1, RPC = SEND_PORT
Watch at port MAIN.P1.OUT1, RPC = SEND_PORT
Watch at port MAIN.P1.OUT1, RPC = SEND_PORT
Watch at port MAIN.P1.OUT1, RPC = SEND_PORT
Watch at port MAIN.P1.OUT1, RPC = SEND_PORT
              :
              :
```

Next, the user removes all port watch points and sets a queue watch point on queue `main.qb2` for all rpcs. The queue has an associated producer port, `main.pb1.out1`, and an associated consumer port, `main.p2.in1`. The occurrence of an rpc on either port, then, causes a message to be displayed by the monitor. (If the port watch point had not been removed, port and queue watch point responses would have been interleaved). When a `Send_Port` or `Get_Port` occurs, the monitor describes the queue state. Below, when the `Send_Port` occurs, the consumer task is already waiting for the data and so the data is immediately transmitted, bypassing the queue. The consumer then issues another `Get_Port` before any more data has been sent and so it is blocked, waiting for data to arrive.

```
monitor> dpwatch * *
monitor> qwatch main.qb2 *
monitor> go
Watch at queue MAIN.QB2, RPC = TEST_OUTPUT_PORT, on port
   MAIN.PB1.OUT1
Watch at queue MAIN.QB2, RPC = SEND_PORT, issued by task MAIN.PB1
   Some task already waiting, data sent immediately
Watch at queue MAIN.QB2, RPC = GET_PORT, issued by task MAIN.P2
   Queue is empty, receiver is blocked
Watch at queue MAIN.QB2, RPC = TEST_OUTPUT_PORT, on port MAIN.PB1.
   OUT1
            :
            :
```

Now the user removes all queue watch points and sets a watch point on the task `main.pb1` for all rpcs. Each time an rpc originates from that task, a message identifying the rpc (and the target port, where appropriate,) is displayed. In this case `main.pb1` is an instance of the predefined **broadcast** task, and so the rpcs are only simulated.

```
monitor> dqwatch * *
monitor> twatch main.pb1 *
monitor> go
Observed task MAIN.PB1 doing TEST_INPUT_PORT at port MAIN.PB1.IN1
Observed task MAIN.PB1 doing TEST_OUTPUT_PORT at port MAIN.PB1.OUT1
Observed task MAIN.PB1 doing TEST_OUTPUT_PORT at port MAIN.PB1.OUT2
Observed task MAIN.PB1 doing GET_PORT at port MAIN.PB1.IN1
Observed task MAIN.PB1 doing SEND_PORT at port MAIN.PB1.OUT1
Observed task MAIN.PB1 doing SEND_PORT at port MAIN.PB1.OUT2
Observed task MAIN.PB1 doing SAFE
            :
            :
```

Break points work exactly like watch points, except that when a break point is reached the application is interrupted and the user has the opportunity to enter more commands. In the following excerpt, the user removes the watch points previously set and then sets a break point on any task doing a `Get_Port`. At each occurrence the user responds with the **go** command and the application continues to the next instance of `Get_Port`. The task break point is then removed and a queue break point set. The effect of the queue break point is analogous to that of the task break point.

```
monitor> dtwatch * *
monitor> tbreak * get_port
monitor> go
Break at task MAIN.PB1 doing GET_PORT at port MAIN.PB1.IN1
monitor> go
Break at task MAIN.P3 doing GET_PORT at port MAIN.P3.IN1
monitor> go
Break at task MAIN.P2 doing GET_PORT at port MAIN.P2.IN1
monitor> dtbreak * *
monitor> qbreak main.q1b *
monitor> go
Break at queue MAIN.Q1B, RPC = SEND_PORT, issued by task MAIN.P1
   Some task already waiting, data sent immediately
monitor> go
Break at queue MAIN.Q1B, RPC = TEST_INPUT_PORT, on port MAIN.PB1.IN1
monitor> go
Break at queue MAIN.Q1B, RPC = GET_PORT, issued by task MAIN.PB1
```

4.3. Kill, Stop, and Resume Commands

This section describes commands used to control the execution state of an application's component tasks.

kill **[** *task-name* **|** **''*''** **]**
stop **[** *task-name* **|** **''*''** **]**
resume **[** *task-name* **|** **''*''** **]**

These commands are used to terminate, pause, or continue execution of a task at the operating system level. As noted previously, break points interrupt a task at the point of some interface call to the executive. It may happen, though, that the user wishes to interrupt an application task that executes for long periods of time without resorting to an interface call; on such occasions the **stop** and **resume** commands are useful. The **kill** command might be used to simulate the occurrence of unexpected task failure for system testing purposes.

4.4. Show and Track Commands

This section describes commands which allow the user to see information about the application currently executing.

show attributes [*task-name* **| ''*'']**

The **show attributes** command displays the attributes of the specified task(s). In the example following we see the attributes of task `main.p2`.

```
monitor> show attributes main.p2
ATTRIBUTES          of task MAIN.P2
   implementation = sink_task
   processor = SUN
   source = taskb.durra.TREE
   xdisplay = :0.0
```

show configuration

The **show configuration** command displays the name of the current configuration and all configurations which can be reached directly from the current level.

show port [*port-name* **| ''*'']**
show queue [*queue-name* **| ''*'']**
show task [*task-name* **| ''*'']**
show type [*type-name* **| ''*'']**

These commands display information the Durra executive knows about the application's Durra ports, queues, tasks, and types. Each of the following fragments demonstrates the results of one of these commands. Additional information is displayed when the situation warrants. For instance, if any break points or watch points have been set on the ports, queues, or tasks in question, then that information will be shown.

```
monitor> show port main.p3.in1
NAME = MAIN.P3.IN1
   ID                  = 2
   CONFIGURATION_LEVEL = MAIN
   DATA_TYPE           = STRING
   ASSOCIATED_QUEUE    = MAIN.QB3
   STATUS              = CONFIGURED
   PORT_DIRECTION      = IN
```

The following example shows that the receiving tasks are waiting for data at queues `main.qb2` and `main.qb3`, but neither sender nor receiver is waiting at queue `main.q1b` (since no ''waiting_client'' field is displayed). Queue `main.q1b` has an upper bound of 10 messages and currently contains three. The results of any active **track** command would also be displayed here.

```
monitor> show queue *
NAME = MAIN.Q1B
   ID                   = 1
   CONFIGURATION_LEVEL  = MAIN
   SOURCE_PORT          = MAIN.P1.OUT1
   DESTINATION_PORT     = MAIN.PB1.IN1
   BOUND                = 10
   ELEMENT_COUNT        = 3
   STATUS               = CONFIGURED
NAME = MAIN.QB2
   ID                   = 2
   CONFIGURATION_LEVEL  = MAIN
   SOURCE_PORT          = MAIN.PB1.OUT1
   DESTINATION_PORT     = MAIN.P2.IN1
   BOUND                = 10
   ELEMENT_COUNT        = 0
   WAITING_CLIENT       = MAIN.P2
   STATUS               = CONFIGURED
NAME = MAIN.QB3
   ID                   = 3
   CONFIGURATION_LEVEL  = MAIN
   SOURCE_PORT          = MAIN.PB1.OUT2
   DESTINATION_PORT     = MAIN.P3.IN1
   BOUND                = 10
   ELEMENT_COUNT        = 0
   WAITING_CLIENT       = MAIN.P3
   STATUS               = CONFIGURED
```

Some explanation of the task fields may be required. The fields `Mailbox` and `Server_mailbox` refer to communications channels from the master executive to the task and from the master executive to the server executive that started the task, respectively. `PID` and `XPID` are the process IDs of the task and any associated *xterm* [6]. `Signal_Pending` indicates whether or not a signal has been received from this task during a reconfiguration. `Time_Out` is the time in seconds that the task has to get to a quiescent state [7] for reconfiguration before it is terminated. `Quiesce_Status` indicates whether or not the task is in a reconfigurable state.

```
monitor> show task main.p1
NAME = MAIN.P1
   KIND                 = USER
   ID                   = 3
   CONFIGURATION_LEVEL  = MAIN
   MAILBOX              = 9
   SERVER_MAILBOX       = 5
   PID                  = 10099
   XPID                 = 0
   PORTS                = MAIN.P1.OUT1
   SIGNAL_PENDING       = FALSE
   TIME_OUT             = 20
   STATUS               = CONFIGURED
   QUIESCE_STATUS       = RUNABLE
```

The `lower_bound` and `upper_bound` in the following type description refer to bounds on the size of the type; since both have a value of `8`, type `byte` is fixed-length, 8 bits.

```
monitor> show type byte
NAME = BYTE
   KIND                = SIZE_TYPE
   ID                  = 2
   LOWER_BOUND         = 8
   UPPER_BOUND         = 8
```

show state

The **show queue** and **show task** commands described above may provide more information than is desired. For example, if at some point during application execution one wished to know how many messages were in each queue, one could display all queue information using the command **show queue *** and then browse through it looking for the relevant numbers. Instead, we provide the **show state** command, the purpose of which is to display a concise picture of the state of the tasks and queues comprising the application. Below is a sample; the user may assume that any task not shown is not blocked currently and any queue not shown is empty.

```
monitor> show state
   Task MAIN.P2 (consumer) blocked at queue MAIN.QB2
   Queue MAIN.Q1B contains 1 messages, bound = 1
```

show track [*queue-name* **| ''*'']**

The **show track** command displays the results of a **track** command on the specified queue(s). The tracking operation records the elapsed time since tracking began, the number of data items that have passed through the queue during that time, the average time a data item spent in the queue, and the number of times the sending and receiving tasks blocked while attempting to write to or read from the queue. In the following excerpt, we see the user request a track operation on all queues. The application starts and runs until interrupted from the keyboard. The results of the tracking operation after 90 seconds show that 120 data elements passed through each queue. The receiving task connected to `main.qb2` had to wait every time; hence the average time a datum spent in the queue was zero, since each was sent directly out when it arrived. In the other two queues, neither sender nor receiver ever blocked, and so the average wait time in the queue is non-zero.

```
monitor> track *
monitor> go
     :
     :
  {keyboard interrupt}
monitor> show track *
NAME = MAIN.Q1B
   ELAPSED_TRACK_TIME  =      90.0 seconds
   DATA_FLOW_COUNT     = 120
   AVG_TIME_IN_QUEUE   =  1.666E-3 seconds
   SENDER_BLOCKED      0 times
   RECEIVER_BLOCKED    0 times
NAME = MAIN.QB2
   ELAPSED_TRACK_TIME  =      90.0 seconds
   DATA_FLOW_COUNT     = 120
   AVG_TIME_IN_QUEUE   =  0.000E+0 seconds
   SENDER_BLOCKED      0 times
   RECEIVER_BLOCKED    120 times
NAME = MAIN.QB3
   ELAPSED_TRACK_TIME  =      90.0 seconds
   DATA_FLOW_COUNT     = 120
   AVG_TIME_IN_QUEUE   =  8.333E-5 seconds
   SENDER_BLOCKED      0 times
   RECEIVER_BLOCKED    0 times
```

track [*queue-name* **| ''*'']**
dtrack [*queue-name* **| ''*'']**

Beginning at the time it is issued, the **track** command initiates the collection of data through the specified queue(s). Partial results of the tracking can be displayed at any time with the **show track** command. Tracking continues until the **dtrack** command is issued.

4.5. Read, Echo, and Silent Commands

This section describes commands which affect the manner in which the monitor communicates with its user.

read *command_file*

The **read** command specifies a command file from which monitor commands should be read. Command files may contain nested **read** commands. When the monitor finishes reading the commands in the file, command processing continues from the scope in which the **read** command was invoked, unless the file contains a **quit** command, in which case the monitor is terminated as usual.

echo

The **echo** command requests that monitor commands be displayed as they are processed. This is the default when commands are entered interactively. If the commands

are being read by the monitor from a file, the default is **silent**, or no display, except when the file is being read via a nested **read** command, in which case the echoing status is inherited from the calling environment.

silent

The **silent** command suppresses the display of monitor commands that are read from command files. This is the default (unless **echo** is inherited from an enclosing file). This command has no effect when issued interactively.

4.6. Set Commands

This section describes the commands used to change certain values maintained by the Durra runtime.

set attribute *task-name attribute-name attribute-value*

The **set attribute** command gives the specified task an arbitrarily-named attribute with an arbitrary string value. Attributes controlling process execution location and display location do not take effect until the next time the process is started. Some attribute names are reserved because they have special meaning to the Durra runtime; see the *Durra User's Manual* for details.

set bound **[***queue-name* **|** **''*''**

positive-integer-value]

The **set bound** command changes the maximum size of the specified queue. The change takes effect immediately. If a task has been blocked attempting to write to the queue, and the new bound is larger than the old bound, the task will be unblocked.

4.7. Miscellaneous Commands

This section describes monitor commands which don't fit into any of the preceding categories.

debug *task-name* **{** *debugger-string* **}**

The **debug** command requests that the specified task be run under control of a source-level debugger. The command must be issued before the task is started or it will have no effect. The optional *debugger-string* provides a way to specify some debugger invocation other than the environment-specified default (e.g., special arguments to the de-

fault debugger or a different debugger altogether). Since a separate terminal interface is required for each debugger activated, this feature is only available when the environment supports the X Window System. Instead of starting the task directly, the Durra executive starts an *xterm* and runs the specified debugger in it, giving it the task name as an argument.

help

The **help** command displays an online help screen which lists the monitor commands.

reconfigure {*configuration_label***}**

The **reconfigure** command causes a reconfiguration to the specified configuration level to occur, regardless of the state of any specified trigger condition. The configuration label is optional if there is only one possible reconfiguration.

The command has no effect when the specified configuration does not exist or is unreachable from the current configuration. The command also fails when another reconfiguration is pending, i.e., the executive has initiated a configuration change but has not yet completed it (for instance, when waiting for a task to get to a quiescent state). Only one reconfiguration may be in progress at any point in time.

References

[1] M.R. Barbacci, C.B. Weinstock, and J.M. Wing.
Programming at the Processor-Memory-Switch Level.
In *Proceedings of the 10th International Conference on Software Engineering*.
Singapore, April, 1988.

[2] M.R. Barbacci, D.L. Doubleday, and C.B. Weinstock.
The Durra Runtime Environment.
Technical Report CMU/SEI-88-TR-18 (DTIC: ADA199480), Software Engineering Institute, Carnegie Mellon University, July, 1988.

[3] M.R. Barbacci and J.M. Wing.
Durra: A Task-Level Description Language Reference Manual (Version 2).
Technical Report CMU/SEI-89-TR-34, Software Engineering Institute, Carnegie Mellon University, September, 1989.

[4] M.R. Barbacci, D.L. Doubleday, C.B. Weinstock, and J.M. Wing.
Developing Applications for Heterogeneous Machine Networks: The Durra Environment.
Computing Systems 2(1), March, 1989.

[5] M.R. Barbacci, D.L. Doubleday, and C.B. Weinstock.
Durra: A Task-Level Description Language User's Manual.
Technical Report CMU/SEI-89-TR-33, Software Engineering Institute, Carnegie Mellon University, September, 1989.

[6] R.W. Scheifler and J. Gettys.
The X Window System.
ACM Transactions on Graphics 5(2):79-109, April, 1986.

[7] C.B. Weinstock.
Performance and Reliability Enhancement of the Durra Runtime Environment.
Technical Report CMU/SEI-89-TR-8 (DTIC: ADA207445), Software Engineering Institute, Carnegie Mellon University, February, 1989.

Index

* 11

Broadcast 13

Ctrl-C 11, 12

Debug 19
Dexec 11
Dmonitor 11
Dpbreak 12
Dpwatch 12
Dqbreak 12
Dqwatch 12
Dtbreak 12
Dtrack 18
Dtwatch 12

Echo 18, 19

Go 8, 11, 12, 14

Help 20

Interrupt 11

Kill 14

Pbreak 12
Pwatch 12

Qbreak 12
Quiescent 16, 20
Quit 8, 12, 18
Qwatch 12

Read 18, 19
Reconfigure 20
Resume 14

Set attribute 19
Set bound 19
Show attributes 15
Show configuration 15
Show port 15
Show queue 15, 17
Show state 17
Show task 15, 17
Show track 17, 18
Show type 15
Silent 19
Stop 14

Tbreak 12
Track 15, 17, 18
Twatch 12

Xterm 16