Technical Report

CMU/SEI-89-TR-033
ESD-TR-89-044

# Durra: A Task-Level Description Language User's Manual

Mario R. Barbacci
Dennis L. Doubleday
Charles B. Weinstock

September 1989

**Technical Report**

**CMU/SEI-89-TR-033**
**ESD-TR-89-044**
**September 1989**

# Durra: A Task-Level Description Language User's Manual

**Mario R. Barbacci**
**Dennis L. Doubleday**
**Charles B. Weinstock**

Software for Heterogeneous Machines Project

**Software Engineering Institute**
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

# Durra: A Task-Level Description Language User's Manual

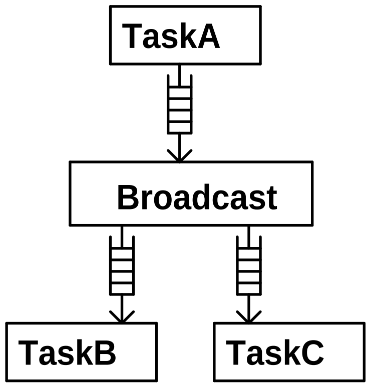
**Abstract**: Durra is a language designed to support the development of large-grained parallel programming applications. This is the manual for users of the Durra compiler, runtime system, and support tools. Additional documents that describe the syntax and semantics of the language and the runtime environment are cited in the bibliography section.

## 1. Introduction

The syntax and semantics of Durra are described in *Durra: A Task-Level Description Language Reference Manual* [3]. This user's manual supplements the language reference manual by describing the facilities available to a Durra user to compile, debug, and execute distributed applications.

All the features described in the Durra language reference manual are implemented with the following two exceptions: (1) Function and timing specifications cannot be used in task selections and the behavioral information part of a task description is treated as commentary information by the compiler. However, timing expressions are used to emulate the behavior of a task by the Durra task emulator [1]. (2) Only sequences of full-item transformations are supported. Array element and record field transformations are not implemented.

To illustrate the various commands and support tools we will use a small Durra application. The example application in Figure 1 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 1:** Durra Application Example

# 2. Compilation Commands

Three Unix commands have been defined to invoke the Durra compiler, the executive-command generator, and the Durra library management utility: **dall**, **dcode** and **dlibrary**.

## 2.1. DLibrary Command

The **dlibrary** command implements a modest library management utility.

**dlibrary { ***options*** } { ***file_names*** }**

The Durra library is a text file containing information about the various compilation units stored in the library and pointers to other libraries containing additional units. The compiler looks first in the current library, then in the libraries referenced in the current library, and so on. The library file is always named ‘‘.DLIBRARY’’ and there can be at most one library file per Unix directory.

**dlibrary -c**

The **-c** option creates a new library (or reinitializes an existing library). This command is normally used when starting the development of a new application.

**dlibrary -a** *directory_name*

The **-a** option extends the library by adding a pointer to a remote directory to be searched for imported task descriptions or type declarations.

**dlibrary -r** *directory_name*

The **-r** option complements the **-a** option and removes a pointer to a remote directory. Task descriptions and type declarations defined in that directory are no longer accessible.

**dlibrary -d** *durra_file_name*

The **-d** option deletes a task description or type declaration from the library (the source files are not disturbed, only the library entry is deleted). Under normal conditions there is no need to delete library entries using this option because the compiler takes care of inserting or deleting task descriptions or type declarations from the library.

## 2.2. DAll Command

The **dall** command invokes the Durra compiler to process a type declaration, a task description, or an application description. A lower-level (i.e., component) task or type must be compiled before a higher-level task or type that uses it.

**dall** *durra_description_file_name*

By convention, a Durra source file name has the extension ‘‘durra’’, although this is not mandatory. The output file (syntax tree stored in the library) is named *durra_description_file_name*.TREE.

## 2.3. DCode Command

The **dcode** command generates the executive instructions for the application.

**dcode** *application_description_file_name*

This command must be issued after all the components and the application descriptions have been compiled with the **dall** command. This command is applied only to the application description file and not to the component task files. The output file (executive program) is named *application_description_file_name*.SCHED

## 2.4. Examples

The following script illustrates the use of these commands to create a library file containing a reference to an external library, to compile several task descriptions and an application description, and finally, to generate the executive commands to execute the application:

```
user > dlibrary -c
user > dlibrary -a /usr/projects/hetsim/durralib
user >
user > dall taska.durra taskb.durra taskc.durra taskmain.durra
taska.durra -> taska.durra.TREE
--Link V4.0 1989/08/14 23:20:25 taska.durra TASK TASKA
Entered in library.
taskb.durra -> taskb.durra.TREE
--Link V4.0 1989/08/14 23:20:34 taskb.durra TASK TASKB
Entered in library.
taskc.durra -> taskc.durra.TREE
--Link V4.0 1989/08/14 23:20:44 taskc.durra TASK TASKC
Entered in library.
taskmain.durra -> taskmain.durra.TREE
--Link V4.0 1989/08/14 23:20:50 taskmain.durra TASK MAIN
Entered in library.
user >
user > dcode taskmain.durra
taskmain.durra.TREE -> taskmain.durra.SCHED
No Errors found. Created taskmain.durra.SCHED
user >
```

After each successful compilation, the library file is augmented with the information about the new unit. This information is used by the compiler to identify the units and to detect changes that might require recompiling dependent units.

# 3. Configuration File

The purpose of the configuration file is to provide information about the hardware configuration to the Durra runtime executive. Figure 2 illustrates the definition of the hardware configuration (i.e., the values for the ‘‘processor’’ attribute).

```
    processor fi.sei.cmu.edu VAX UVAX SEIFI
    processor cu.sei.cmu.edu VAX UVAX SEICU
    processor ag.sei.cmu.edu VAX UVAX SEIAG
    processor e.sei.cmu.edu VAX UVAX SEIE
    processor sei.cmu.edu VAX SEI
    processor cr.sei.cmu.edu VAXVMS SEICR
    processor fh.sei.cmu.edu SUN SEIFH
    processor af.sei.cmu.edu SUN SEIAF
    xdisplay fi.sei.cmu.edu:0.0 SEIFI
    xdisplay ag.sei.cmu.edu:0.0  SEIAG
    xdisplay e.sei.cmu.edu:0.0 SEIE
```

**Figure 2:** Configuration File

Each line of the configuration file identifies a resource available to execute task implementations. The first field of the line, `processor` or `xdisplay` identifies the type of line. The second field specifies the Internet address of the processor (e.g., `fi.sei.cmu.edu`) or the X window display unit (e.g., `ag.sei.cmu.edu:0.0`). The rest of the fields specify values for the ‘‘processor’’ or ‘‘xdisplay’’ attributes used in the task descriptions.

The values of the ‘‘processor’’ attribute are used to specify classes of machines. For example, the first line specifies that processor `fi.sei.cmu.edu` belongs to classes `VAX`, `UVAX`, and `SEIFI`. (The names of the classes are arbitrary although it makes sense to choose mnemonic values.) Class `UVAX` (i.e., microvax) is a subset of class `VAX` and this is different from class `VAXVMS`. (The former run the Ultrix-32 operating system, the latter runs the VMS operating system.) Class `SEIFI` contains exactly one processor, `fi.sei.cmu.edu`, and can be used as the value of the ‘‘processor’’ attribute of a task implementation that can only run on that processor.

The values of the ‘‘xdisplay’’ attribute are used to specify the display for a task under the X Window screen management system. For example, the first `xdisplay` line specifies that `fi.sei.cmu.edu:0.0` will be the X Window display for all tasks that specify `SEIFI` as their `xdisplay` attribute. (The values of the `xdisplay` attribute are arbitrary although it makes sense to choose mnemonic values.)

Note that this configuration file is not written in the task description language. The example shown is simply an illustration of the kinds of information that are likely to be in the file; form and content of the file are implementation-dependent. See [2] for additional information.

# 4. Execution Environment

The runtime executive gains access to the task implementations, the configuration description, and other application or user dependent information through file and directory names defined in the execution environment.

In the Unix implementation, the execution environment is defined by the following shell environment variables.

**DURRA_PATH** is a list of directories containing the task implementations and other runtime programs, including the location of the servers and executive. This variable must be defined before running a Durra application. For example, it could be defined by:

```
    setenv DURRA_PATH ~hetsim/tasklib/:~hetsim/exec/
```

**DURRA_MONITOR** is the name of the monitor task. The executive will start the monitor task if the **-m** switch is used with the **dexec** command (Section 5.1). For example, it could be defined by:

```
    setenv DURRA_MONITOR ~hetsim/mon/monitor
```

**DURRA_MONITOR_WINDOW** defines the window for the monitor task. For example, it could be defined by:

```
    setenv DURRA_MONITOR_WINDOW "-geometry 80x50+150+20"
```

# 5. Execution Commands

Two Unix commands have been defined to invoke the runtime executive and monitor: **dexec** and **dmonitor**.

## 5.1. DExec Command

The **dexec** command starts the executive. This command must be issued on each processor available to the application. The executive can run in two modes: master and server. One of the processors must run the master executive, the others must run server executives.

**dexec -s**

The **-s** options starts a executive in server mode. This command must be issued on each processor that is to execute any of the application tasks. However, if any of these processors is also used to run the master executive then it is not necessary to start a server executive on that processor since the master executive will subsume the server function for that processor.

If the **-s** option is not used, the executive starts in master mode. In master mode the **dexec** command allows additional options and parameters.

**dexec {** *master-executive-options* **}** *executive_program_file_name*

The only parameter required by the master executive is the name of a file containing executive instructions. By convention, these files have names of the form *application_name***.durra.SCHED**. The **dexec** command supplies the appropriate file extensions (‘‘.durra’’ and ‘‘.SCHED’’) if they are missing. For example, ‘‘**dexec** *some_application*’’ invokes the executive with the file ‘‘some_application.durra.SCHED’’.

**dexec -c***configuration_file* **. . .**

The **-c** option specifies the name of a file containing network configuration information (e.g., the names of the available processors on which tasks can execute). If not specified, the executive looks by default for the configuration file ‘‘config.txt’’, first in the current directory and then in each directory specified in the ‘‘DURRA_PATH’’ environment variable (Section 4).

**dexec -m . . .**
**dexec -md***monitor_display* **. . .**
**dexec -mf***monitor_command_file* **. . .**

The **-m** options specify that the monitor task is started by the executive. The user can then use the monitor command language to interact with the executive before the application task start. The monitor task is specified by the **DURRA_MONITOR** environment variable (see Section 4). If **md***monitor_display* is specified, then the X window for the monitor appears on the specified display; otherwise it appears on the default display identified in the environment. If **mf***monitor_command_file* is specified, then the monitor reads its initial commands from the specified file.

**dexec -q***queue_size* **. . .**

The **-q** option specifies the default size (number of elements) for those queues whose size was not provided in the task or application descriptions. If this option is not specified, the default queue size is 1.

## 5.2. DMonitor Command

The **dmonitor** command starts the Durra application debugger/monitor.

**dmonitor {** *optional-parameters* **}**

This command may be issued on any currently configured processor. The Durra executive must be running on some node in the configuration so that the monitor will be able to communicate with it.

**dmonitor -f***monitor_command_file* **. . .**

The **-f** option specifies a file from which the monitor will read an initial set of commands. The default is interactive input.

**dmonitor -h***executive_host_processor_name* **. . .**

The **-h** option specifies the name of a processor on which an executive is currently running. This parameter defaults to the local processor and must be supplied if the monitor is started on a processor on which there is no running executive.

## 5.3. Examples

The following script illustrates the use of these commands to define the environment variables, start servers and executive, and execute the example application:

*Define the environment variables:*
```
user > set hetsim=/usr/projects/hetsim
user > setenv DURRA_PATH "$hetsim/tasklib:$hetsim/exec"
```

*Start the executive with a user-specified configuration file:*
```
user > dexec -cseiag_config.txt broadcast_example.durra
Executive will execute as master out of
                                 /usr/projects/hetsim/exec/executive
Using configuration file: seiag_config.txt
Executive/Server is executing on ag.sei.cmu.edu
```

*Application dependent output:*
```
Opening parameter_file
Message count=500
Message size=20
Process 3(Source_Task ) Message Size= 20 Messages Count= 500
Process 4(Sink_Task   ) Message Size= 20 Messages Count= 500
Process 5(Sink_Task   ) Message Size= 20 Messages Count= 500
        :
        :
```
*Application runs to completion:*
```
user >
```

The invocation of **dexec** in this example illustrates the use of a user-specified configuration file (`seiag_config.txt`) to override the default configuration file (“config.txt”). In this particular example, the configuration file used consisted of a single line:

```
processor ag.sei.cmu.edu UVAX VAX SUN SEIE SEIAG SEIFI
```

The effect of using such a configuration file is to “fold” all the network processors and processor classes into one machine. Although this example is extreme, it illustrates the ability to configure the network and to control the allocation of resources without modifying the task implementations or the task/application descriptions.

# 6. Monitor Commands

The Durra application debugger/monitor [4] is an interactive process which communicates with the Durra executive at runtime to provide a user with 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 is started one includes 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, as described in Section 5.2.

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.

## 6.1. 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.

## 6.2. 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
```

## 6.3. 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.

## 6.4. 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* [5]. `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 [6] 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.

## 6.5. 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.

## 6.6. 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.

## 6.7. 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 default 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.
*MasterTask: The Durra Task Emulator*.
Technical Report CMU/SEI-88-TR-20 (DTIC: ADA199429), Software Engineering Institute, Carnegie Mellon University, July, 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] D.L. Doubleday.
*The Durra Application Debugger/Monitor*.
Technical Report CMU/SEI-89-TR-32, Software Engineering Institute, Carnegie Mellon University, September, 1989.

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

[6] 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

# Table of Contents

## List of Figures