

THE RESOURCE MANAGER

A Guide to the Resource Debugger

User Guide

© COPYRIGHT MCMXCIV EUROTHERM LIMITED

All rights strictly reserved. No part of this document may be stored in a retrieval system, or transmitted, in any form or by any means without prior written permission from Eurotherm Ltd

HA024105C002 2 M Fox



Contents

1	Scope	4
2	Related Documents	4
3	Loading	4
4	Connections	4
4.1	Addressing	4
4.2	Connecting	5
4.3	Ping	5
4.4	Quitting	6
5	Internal	6
5.1	Help	6
5.2	Macros	6
5.3	Delays	7
6	References	7
6.1	Setting a Reference	7
6.2	The Data	8
6.3	Exercising	12
6.4	Waiting	12
7	Inspecting Data	13
7.1	What Is	13
8	ST	14
8.1	Trace	15
8.2	Break	16
9	Redirection	18
9.1	Transcription	19
9.2	Scripts	19
10	Routers	19
11	Debugger Invoke Options	21
12	Debugger Command Summary	22

VERSION HISTORY

Version	Date	Changes
1	March 10, 1994	Initial issue
2	December 20, 1994	Update for Version 2.2

1 Scope

This document describes the Resource debugger (Version 2.2). The debugger is used to conduct a debug session with another **TASK** (possibly on another **RESOURCE**). A debug session may be conducted by an ST programmer to examine and modify the **TASK** data as well tracing or breaking the execution of the ST at desired points. All commands required to conduct a debug session with another **TASK** are described in detail.

2 Related Documents

- [1] HA024105C001 A Guide to Var References
- [2] HA024105C003 A Guide to Tuning the Resource
- [3] HA024105C005 A Guide to Setting Up CMS Networks

3 Loading

A debug session is begun by invoking the resource debugger **resdebug**.

By default the debugger is loaded as the CMS process "ResourceDebugger". If another resource debugger session is current on the same CMS node then another process name must be used (§11).

Once loaded a debug session may be begun. A debug session may either be conducted interactively by typing commands or from a script file. In either case the maximum length of any command is 512 characters.

When the debugger is quit the **TASK** is unloaded.

4 Connections

A debug session may be conducted with any **TASK** whose **RESOURCE** supports debugging. A debug session may be conducted with any such **TASK** on the **RESOURCE** network. Each **TASK** may only be connected to one debugger.

In order to conduct a debug session with another **TASK** it is first necessary to connect to that **TASK**. The debugger may be connected to any number of **TASKs** at any given time. The debugger maintains a list of all **TASKs** to which connections have been issued. It is not necessary to be connected to a **TASK** in order to set up dynamic **VAR REFERENCES** to it §6.

4.1 Addressing

A **TASK** is addressed by using its Application Entity Name which is of the form <Resource>:<TaskName>.

4.2 Connecting

connect

In order to connect to a **TASK** the connect command should be issued. Once the connect has been issued the **TASK** name is added to the list of connected **TASKs**.

This **TASK** will now send *ALL* of its debug messages to this debugger until disconnected. It should be noted that all connected **TASKs** will send their messages to the debugger and it is up to the user to distinguish from which **TASK** they originated. It is therefore advisable to keep the number of connections to a minimum.

A connected **TASK** may run out of buffers to reply in full to a command; this may occur because there are too many active commands (especially trace) to the **TASK** or it has insufficient buffers for the operation. Such an event will be reported in the next message from that **TASK** in the form

```
**** 7 debug messages lost due to lack of buffers ****
```

This may be rectified by retuning the **RESOURCE** for the debug session (see [2]).

The address of this **TASK** will now form the debugger prompt and is known as the *current TASK*. This **TASK** will remain the current **TASK** until it is either disconnected or the debugger is connected to another **TASK**. It may happen during a series of messages from a **TASK** that the prompt “interrupts” the messages, this is perfectly normal.

list connect

The list connect command may be used to list all connected **TASKs**.

disconnect

A disconnect may be issued to any connected **TASK**. If no address is specified in the disconnect command then the current **TASK** is disconnected. When a **TASK** is disconnected then it is removed from the list of connected **TASKs**. If the list of connected **TASKs** is not empty then the **TASK** last connected to becomes the current **TASK**.

When the disconnection command is received by the **TASK** all current trace (§8.1) and break (§8.2) are deleted.

Example

```
NotConnected/Debug: connect "regress2:task1"
regress2:task1/Debug: Connected to TASK "regress2:task1"
regress2:task1/Debug: list connect
List of connections :-
"regress2:task1"
regress2:task1/Debug: disconnect
NotConnected/Debug: list connect
List of connections :-
NotConnected/Debug:
```

4.3 Ping

ping

The ping command may be used to determine if a **TASK** is reachable. Ping simply sends a message and reports any response. A ping may be sent to any **TASK** whether connected or not.

Example

```
NotConnected/Debug: ping "regress2:task2"
NotConnected/Debug: Ping acknowledge from TASK "regress2:task2"
```

4.4 Quitting

`quit`

When the debugger is quit a disconnect is issued to all connected **TASKs**.

5 Internal

This section lists all internal commands, that is commands which have no effect on any other **TASKs**, ie command that do not require any connection to be established.

5.1 Help

`help`

Help is available at 2 levels :

- A list of all commands
- Specific help on any individual command

5.2 Macros

String macros may be defined for commonly used command combinations or any frequently used string combination.

`define`

Is used to define a string macro.

`undefine`

Is used to delete the string macro.

`list macros`

Is used to list all defined macros.

Example

```
NotConnected/Debug: define cnx="connect \"regress2:task1\""  
NotConnected/Debug: define dnx="disconnect"  
NotConnected/Debug: cnx  
regress2:task1/Debug: Connected to TASK "regress2:task1"  
regress2:task1/Debug: list macros  
Macro Name      Macro Definition  
  
dnx disconnect  
cnx connect "regress2:task1"  
regress2:task1/Debug: dnx  
NotConnected/Debug:
```

5.3 Delays

There are 2 debugger commands which introduce delays. These are designed for use with scripts but may be used interactively. Whilst in a delay the debugger will not read any incoming messages.

`wait`

The wait command introduces a delay in seconds between this command and the next. This may also be used to wait for an Outstanding operation on a **VAR REFERENCE** to complete (see §6).

`pause`

The pause command introduces a delay in seconds between consecutive commands. Setting a pause of 0 is equivalent to turning off the pause.

6 References

The references facility can be viewed as creating dynamic, manually controlled **VAR REFERENCES** (see [1] for more detail).

In fact the underlying messaging used is exactly that used by **VAR REFERENCES**.

In order to implement these **VAR REFERENCES** the debugger uses an Outstanding Operation Table (**OOT**), just like an **ST TASK** (11).

It is however possible to create references that are not **TASK** coherent.

6.1 Setting a Reference

`ref`

The first stage in establishing a debugger reference is that of specifying the name of the reference and the associated ref string. The template is then read. There is no template matching stage as there is nothing to match against. The reference string syntax is the same as that for **VAR REFERENCES** (see [1]). Certain ref strings particularly those containing `[]` may need to be enclosed in `""`. The reference string must contain a **RESOURCE** specification unless the debugger the reference is to something on the current **TASK** (ie the **TASK** whose name is given in the prompt).

If a reference is to contain services then the number of services must be specified when the reference is created by appending “with service” or “with <Number> services”. Each service must then have its inputs and outputs specified as a list. This list must not be empty. If the service has no inputs or outputs of interest then the “waiting” output can be specified to ensure the ref string list is not empty. The template is not considered read until all the service templates have also been read.

No read or write operations may be performed on the data until the template has been read. The debugger will create some local data objects of the correct size, these will be uninitialised.

`expand`

Expand a reference to a **FUNCTION_BLOCK** or structured data type. This is a “lazy” way of creating a reference in that it does require knowledge of the interface. This method may result in the inability to re-read a template if the reference string that would be required for the expanded references is longer than 255 characters. In these circumstances an error message will be reported.

`list refs`

List all debugger references.

unref

Any debugger reference once created may be deleted. A debugger reference may be deleted with any number of outstanding operations (including the template read).

Example

```
NotConnected/Debug: ref r1="regress2:prog1.nargs"
NotConnected/Debug: Reference r1 template read
NotConnected/Debug: ref r2="regress2:prog1{error,result}"
NotConnected/Debug: Reference r2 template read
NotConnected/Debug: ref r3="regress2:prog1.args[3]"
NotConnected/Debug: Reference r3 template read
NotConnected/Debug: list refs
Name    VarReference
r3=regress2:prog1.args[3]
r2=regress2:prog1{error,result}
r1=regress2:prog1.nargs
NotConnected/Debug: unref r2
NotConnected/Debug: list refs
Name    VarReference
r3=regress2:prog1.args[3]
r1=regress2:prog1.nargs
NotConnected/Debug: ref r5="regress5:prog1.Local"
NotConnected/Debug: Reference r5 template read
NotConnected/Debug: list ref r5
r5=regress5:prog1.Local
  1:prog1.Local [5.0.1] INTERNAL_FUNCTION_BLOCK=?
NotConnected/Debug: expand r5
NotConnected/Debug: Reference r5 expanded
NotConnected/Debug: Reference r5 template read
NotConnected/Debug: list ref r5
r5=regress5:prog1.Local{WakeUp,Who,Cycle,Woken,WhoFrom,Change,NextCycle,LastCirc
um,LastCycle}
  1:prog1.Local.WakeUp [5.1.1] INPUT BOOL=?
  2:prog1.Local.Who [5.1.2] INPUT STRING=?
  3:prog1.Local.Cycle [5.1.3] INPUT DINT=?
  4:prog1.Local.Woken [5.1.4] OUTPUT BOOL=?
  5:prog1.Local.WhoFrom [5.1.5] OUTPUT STRING=?
  6:prog1.Local.Change [5.1.6] OUTPUT DINT=?
  7:prog1.Local.NextCycle [5.1.7] OUTPUT DINT=?
  8:prog1.Local.LastCircum [5.1.8] INPUT_OUTPUT DINT=?
  9:prog1.Local.LastCycle [5.1.9] INPUT_OUTPUT DINT=?
```

6.2 The Data

store

Once the template has been read it is possible to overwrite any item of data by storing a new value. Each individual item (addressable by name or number), including each element of any array is individually writable.

It should be noted that all enumerated types must be written as a numeric value in the range 0 to 255. No checking can be performed to verify that this is a valid value for the enumeration therefore this feature should be used with care.

read

Once a template has been read the debugger reference data may be read, with the read command. Each read will overwrite all the current values held locally.

Whilst a debugger reference may be made to the **TASK**, **PROGRAM** and **FUNCTION_BLOCK** types it is not possible to perform any operations on the data. It will not be possible to store a value and therefore not possible to issue a write. It will however be possible to issue a read but no data is actually read and the values when listed will all be marked with a ?.

write

Once the template has been read and *ALL* values have been assigned a value either by reading the debugger reference or storing a value locally then the whole of the writable part of the debugger reference may be written. Only **INPUTs**, **IN_OUTs**, **INPUT_OUTPUTs** and **INTERNALs** may be written.

service

Once the template(s) have been read any of the services specified in the reference may be executed. No check is made that all inputs have been written to before the service is executed, as in general this may not be a requirement.

scan

Once the template has been read a scan time may be set up which will cause the data to be read every scan period.

list ref

List an individual debugger reference with all its type, mode, value and GAD information.

It should be noted that all enumerated values are displayed as a numeric value in the range 0-255, it is therefore necessary to know what the enumeration is to interpret this.

Example

```

NotConnected/Debug: ref r="Perform:prog1.in1"
NotConnected/Debug: Reference r template NOT read
NotConnected/Debug: unref r
NotConnected/Debug: ref r="Perform:prog1.In1"
NotConnected/Debug: Reference r template read
NotConnected/Debug: list ref r
r=Perform:prog1.In1
  1:prog1.In1 [4.0.15] INTERNAL DINT=?
NotConnected/Debug: read r
NotConnected/Debug: Reference r read
NotConnected/Debug: list ref r
r=Perform:prog1.In1
  1:prog1.In1 [4.0.15] INTERNAL DINT=0
NotConnected/Debug: store r=5
NotConnected/Debug: list ref r
r=Perform:prog1.In1
  1:prog1.In1 [4.0.15] INTERNAL DINT=5 ( Stored )
NotConnected/Debug: write r
NotConnected/Debug: Reference r written
NotConnected/Debug: ref r2="Perform:ProgU1{Local1,Local2,Local3{In1,In2}}"
NotConnected/Debug: Reference r2 template read
NotConnected/Debug: read r2
NotConnected/Debug: Reference r2 read
NotConnected/Debug: store r2.Local1=8
NotConnected/Debug: store r2.Local2[4]=9
NotConnected/Debug: store r2.2[5]=5
NotConnected/Debug: store r2.3=90
NotConnected/Debug: store r2.In2=67
NotConnected/Debug: list ref r2
r2=Perform:ProgU1{Local1,Local2,Local3{In1,In2}}
  1:ProgU1.Local1 [8.0.13] INTERNAL DINT=8 ( Stored )
  2:ProgU1.Local2 [8.0.14] INTERNAL ARRAY [ 1..32 ] OF DINT 0, 0, 0, 9, 5, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ( Stored )
  3:ProgU1.Local3.In1 [8.13.1] INPUT DINT=90 ( Stored )
  4:ProgU1.Local3.In2 [8.13.2] INPUT DINT=67 ( Stored )
NotConnected/Debug: ref s="regress14:prog1.Sum" with 2 services
NotConnected/Debug: ref s.add=in1,in2,out
NotConnected/Debug: ref s.subtract=in1,in2,out
NotConnected/Debug: Reference s template read
NotConnected/Debug: Reference s.add template read
NotConnected/Debug: Reference s.subtract template read
NotConnected/Debug: store s.add.in1=5
NotConnected/Debug: store s.add.in2=7
NotConnected/Debug: service s.add
NotConnected/Debug: Service s.add Completed
NotConnected/Debug: store s.subtract.in1=3
NotConnected/Debug: store s.subtract.in2=2
NotConnected/Debug: service s.subtract
NotConnected/Debug: Service s.subtract Completed

```

```

NotConnected/Debug: list ref s
s=regress14:prog1.Sum
  1:prog1.Sum [4.0.13] INTERNAL FUNCTION_BLOCK=?
s.add=prog1.Sum.add{in1,in2,out}
  1:prog1.Sum.add.in1 [4.14.1] INPUT DINT=5
  2:prog1.Sum.add.in2 [4.14.2] INPUT DINT=7
  3:prog1.Sum.add.out [4.14.3] OUTPUT DINT=12
s.subtract=prog1.Sum.subtract{in1,in2,out}
  1:prog1.Sum.subtract.in1 [4.15.1] INPUT DINT=3
  2:prog1.Sum.subtract.in2 [4.15.2] INPUT DINT=2
  3:prog1.Sum.subtract.out [4.15.3] OUTPUT DINT=1

```

list properties

List all the properties of the debugger reference as if it were a **ST VAR REFERENCE**.

In addition to the properties the internal state of the reference is displayed. This will have one of the values :

- Uninitialised
- Matching
- MatchingServiceParents
- MatchingServices
- Matched
- Reading
- Writing
- Servicing
- ReadError
- WriteError
- MatchError

There are 2 additional properties listed which are not available from within ST, these are :-

coherent - This indicates if the reference is **TASK** coherent or not.

dataChanged - This is similar to **dataRead** except that it reports if the data values have changed since that last time the property was read.

Example

```
NotConnected/Debug: ref r="regress2:prog1.nargs"
NotConnected/Debug: Reference r template read
NotConnected/Debug: read r
NotConnected/Debug: Reference r read
NotConnected/Debug: write r
NotConnected/Debug: Reference r written
NotConnected/Debug: list properties r
Properties of r :-
ref          'regress2:prog1.nargs'
resolution   Ultimate
scan         T#0s
timeStamp    DT#1993-03-01-13:09:25 QT#0ms
state        Matched
status       Ok
readStatus   Ok
writeStatus  Ok
servStatus   Undefined
newData      1
dataChanged  1
coherent     1
```

6.3 Exercising**exercise**

The exercise command may be used to generate repeated reads or writes. The exercise command reports the time taken to receive all replies. It will also timeout if the responses are not received in sufficient time. An exercise may be used to give a *rough* guide to the throughput of an equivalent ST VAR REFERENCE under comparable conditions.

Whilst an exercise is in progress any debugger references with a scan time set will not be scanned.

Example

```
NotConnected/Debug: exercise 10 read r
Exercise completed in 1 seconds
NotConnected/Debug: exercise 10 write r
Exercise completed in 1 seconds
```

6.4 Waiting

It is possible to wait for an outstanding operation to complete.

Example

```
NotConnected/Debug: read r
NotConnected/Debug: wait r 10
NotConnected/Debug: Reference r read
```

7 Inspecting Data

7.1 What Is

There are 2 what is commands for printing out information about ST items. The form "" is taken as the RESOURCE. Both commands take an optional leading RESOURCE name (in the same form as a reference string), if this is not supplied then the command is issued to the currently connected TASK.

`whatis`

The command prints out information any visible ST item. It prints out the GAD, mode, type, name and array dimensions if found,

`Whatis`

Provides a more detailed description.

Prints out the GAD, mode, type, name and array dimensions of the object, the number of children if complex and the same for each of the objects children.

Example

```
NotConnected/Debug: connect "regress2:task1"
regress2:task1/Debug: Connected to TASK "regress2:task1"
regress2:task1/Debug: whatis ""
regress2:task1/Debug:
[0.0.0] INTERNAL regress2 : RESOURCE ;
regress2:task1/Debug: whatis "prog1.nargs"
regress2:task1/Debug:
[3.0.3] INTERNAL nargs : USINT ;
regress2:task1/Debug: Whatis "prog1"
regress2:task1/Debug:
[3.0.0] INTERNAL prog1 : PROGRAM progreg2 ;
with 10 children
regress2:task1/Debug:
[3.0.1] INTERNAL init : STEP Step ;
[3.0.2] INTERNAL calc : STEP Step ;
[3.0.3] INTERNAL nargs : USINT ;
[3.0.4] INTERNAL args : ARRAY [1..6] OF REAL ;
[3.0.5] INTERNAL operators : ARRAY [1..5] OF STRING ;
[3.0.6] INTERNAL result : REAL ;
[3.0.7] INTERNAL error : STRING ;
[3.0.8] INTERNAL calculation : STRING ;
[3.0.9] INTERNAL docalc : BOOL ;
[3.0.10] INTERNAL i : USINT ;
regress2:task1/Debug: disconnect
NotConnected/Debug: Whatis "regress14:prog1.Sum.add"
NotConnected/Debug: Whatis "regress14:prog1.Sum.add"
[4.13.1] INTERNAL add : SERVICE add ;
with 5 children
NotConnected/Debug:
[4.14.1] INPUT in1 : DINT ;
[4.14.2] INPUT in2 : DINT ;
[4.14.3] OUTPUT out : DINT ;
[4.14.4] OUTPUT waiting : USINT ;
[4.14.5] INTERNAL pending : DINT ;
```

print

Prints the current value of any visible ST item.

set

Attempts to alter the value of any visible item. The previous value of the item is reported when the set is complete.

Example

```
regress2:task1/Debug: print prog1.nargs  
regress2:task1/Debug: 0  
regress2:task1/Debug: set prog1.nargs=65  
regress2:task1/Debug: Previous Value = 0  
regress2:task1/Debug: print prog1.nargs  
regress2:task1/Debug: 65
```

Print and set are fairly simple commands which do not use the **VAR REFERENCE** mechanisms. In general a reference should be set up to either read or write data.

Both print and set treat all items of data to be one of

STRING For all ST **STRING**s

DINT For all integer values eg **INT**, **USINT**, **BOOL**, **EDGE**

LREAL For both **REAL** and **LREAL**

Set recognises the time types **TIME**, **DATE**, **DATE_AND_TIME**, **TIME_OF_DAY** as input (eg **t#10s**) but they are all printed as **DINT**s.

8 ST

If some **FUNCTION_BLOCKS** of a **RESOURCE** have been compiled with the debug option then it is possible to debug those sections of ST. Up to a total of 5 trace or break points may be added at any **TASK**.

A line of ST should only contain one ST statement otherwise multiple trace/break points will be set in between the statements on the line. For the purposes of debugging trace/break points will only have effect on lines of ST that actually execute code.

8.1 Trace

A trace point is a point in the ST which is reported to the debugger whenever the code execution reaches it.

A trace point may be established at any of the following.

- On entry to a `FUNCTION_BLOCK`
- On exit from a `FUNCTION_BLOCK`
- On executing a line of ST

Each arrival at a trace point generates a message to the debugger.

`trace entry` or `trace entry fblock`

Trace entry to a `FUNCTION_BLOCK`.

`trace exit` or `trace exit fblock`

Trace exit from a `FUNCTION_BLOCK`.

`trace either` or `trace either fblock`

Trace both entry and exit from a `FUNCTION_BLOCK`.

`trace entry all` or `trace entry all fblocks`

`trace exit all` or `trace exit all fblocks`

`trace either all` or `trace either all fblocks`

Trace entry and/or exit from all `FUNCTION_BLOCK`s.

Example

```
regress2:task1/Debug: trace entry fblock prog1
regress2:task1/Debug: Trace/Break on block instance prog1 added
regress2:task1/Debug: block type junkprog source line 64 - Entry into
regress2:task1/Debug: block type junkprog source line 64 - Entry into
regress2:task1/Debug: block type junkprog source line 64 - Entry into
```

`trace at`

Trace execution of a particular line of ST in a `FUNCTION_BLOCK`. The trace is issued before the line of ST is executed.

Example

```
NotConnected/Debug: connect "regress2:task1"
regress2:task1/Debug: Connected to task "regress2:task1"
regress2:task1/Debug: trace at prog1 45
regress2:task1/Debug: Trace/Break on block instance prog1 added
regress2:task1/Debug: block type junkprog source line 45 - On Line
regress2:task1/Debug: block type junkprog source line 45 - On Line
```

`list traces`

This lists all trace points.

`delete trace entry` or `delete trace entry fblock`

`delete trace exit` or `delete trace entry fblock`

`delete trace either` or `delete trace either fblock`

Delete trace entry and/or exit from a `FUNCTION_BLOCK`.

`delete trace entry all` or `delete trace entry all fblocks`

`delete trace exit all` or `delete trace exit all fblocks`

`delete trace either all` or `delete trace either all fblock`

Delete trace entry and/or exit from all `FUNCTION_BLOCKS`.

`delete trace all` or `delete trace all fblocks`

Delete all trace points on all `FUNCTION_BLOCKS`.

8.2 Break

A break point is a point in the ST where execution stops until commanded to continue.

A break point may be established at any of the following

- On entry to a `FUNCTION_BLOCK`
- On exit to a `FUNCTION_BLOCK`
- On executing a line of ST

Each arrival at a break point is reported to the debugger.

`break entry` or `break entry fblock`

Break on entry to a **FUNCTION_BLOCK**.

`break exit` or `break exit fblock`

Break on exit from a **FUNCTION_BLOCK**.

`break either` or `break either fblock`

Break on either entry or exit from a **FUNCTION_BLOCK**.

`break entry all` or `break entry all fblocks`

`break exit all` or `break exit all fblocks`

`break either all` or `break either all fblocks`

Break on entry and/or exit from any **FUNCTION_BLOCK**.

`break at`

Break before a particular line of ST in a **FUNCTION_BLOCK**.

`list breaks`

This lists all break points.

`continue`

Continue execution of the ST.

`step`

Single-step through the ST.

`next`

Single-step through the ST without entering any **FUNCTION_BLOCKS** called by this **FUNCTION_BLOCK**.

`display`

Display the ST of this **FUNCTION_BLOCK**.

Example

```

NotConnected/Debug: connect "regress2:task1"
regress2:task1/Debug: Connected to TASK "regress2:task1"
regress2:task1/Debug: break at prog1 45
regress2:task1/Debug: Trace/Break on block instance prog1 added
regress2:task1/Debug: continue
regress2:task1/Debug: block type junkprog source line 45 - On Line
    Print ( str1 := 'Write while reading$N' ) ;
regress2:task1/Debug: step
regress2:task1/Debug: block type junkprog source line 46 - On Line
    Remote := Count ;
regress2:task1/Debug: step
regress2:task1/Debug: block type junkprog source line 47 - On Line
    Reading := 0 ;
regress2:task1/Debug: continue
regress2:task1/Debug: block type junkprog source line 45 - On Line
    Print ( str1 := 'Write while reading$N' ) ;
regress2:task1/Debug: list breaks
regress2:task1/Debug: breaks set :-
Block GAD [5.0.0] Break events (Line 45)
regress2:task1/Debug: delete break all
regress2:task1/Debug: continue
regress2:task1/Debug: display 44,48
    IF ( Remote~status = 1 AND Reading = 1 ) THEN
        Print ( str1 := 'Write while reading$N' ) ;
        Remote := Count ;
        Reading := 0 ;
    END_IF ;
regress2:task1/Debug:

```

delete break entry	or	delete break entry fblock
delete break exit	or	delete break exit fblock
delete break either	or	delete break either fblock

To delete any **FUNCTION_BLOCK** entry or exit break point.

delete break entry all	or	delete break entry all fblocks
delete break exit all	or	delete break exit all fblock
delete break either all	or	delete break either all fblock

To delete any **FUNCTION_BLOCK** entry or exit for all **FUNCTION_BLOCKS**.

delete break all	or	delete break all fblocks
------------------	----	--------------------------

To delete all break points in all **FUNCTION_BLOCKS**.

Deleting a break point(s) does not cause a continue.

9 Redirection

It is possible to redirect both input and output to the debugger. By default the debugger reads all input from the keyboard and sends all output to the screen.

> or <	All input or output can be redirected to a file(s).
--------	---

9.1 Transcription

`transcribe`

The transcribe command can be used to copy all commands to a file.

9.2 Scripts

A debugger script may be supplied on start up with the `-d` option (§11). This will cause all input to be read from this file. No prompts are issued whilst reading from this file.

10 Routers

The debugger provides the facility to send Router messages which provide information about the connections (and potential connections) between different **RESOURCES**. Only Router tasks are capable of responding to these messages, all other tasks will ignore them.

Router messages (especially setting proxies) ought to be performed using the **roumsg** tool [3].

`list nodes`

List all nodes to which this Router has sent messages.

`list media`

List all the media supported by this Router.

`list proxies`

List all node proxies set up on this Router.

`list aes`

List all the application entities identified by this Router.

`list local aes`

List all the application entities on this Router's node.

`list medium`

List the state of the medium.

`set proxy`

Set up a node proxy.

`set logger`

Direct router log messages to the debugger.

Example

```

NotConnected/Debug: router 149.121.128.29 list nodes
NotConnected/Debug:
Router "149.121.128.29"
2/32 Nodes :-
"2.2.2.2" on UDP
"1.1.1.1" on UDP
NotConnected/Debug: router 149.121.128.29 list media
NotConnected/Debug:
Router "149.121.128.29"
1 Media :-
UDP
NotConnected/Debug: router 149.121.128.29 list medium UDP
NotConnected/Debug:
Router "149.121.128.29" UDP Active, Msgs Sent = 6, Received = 6
NotConnected/Debug: router 149.121.128.29 list aes
NotConnected/Debug:
Router "149.121.128.29"
2/64 Application Entities :-
"ROUT" "Router" on 1.1.1.1
"ROUT" "Router" on 2.2.2.2
NotConnected/Debug: router 149.121.128.29 list local aes
NotConnected/Debug:
Router "149.121.128.29"
3/8 Local Application Entities :-
"ROUT" "Router"
"RMP" "regress2:task1"
"RMP" "regress2:"
NotConnected/Debug: router 149.121.128.29 set proxy 3.3.3.3 with 1.1.1.1
NotConnected/Debug:
Router "149.121.128.29" Proxy set
NotConnected/Debug: router 149.121.128.29 list proxies
NotConnected/Debug:
Router "149.121.128.29"
1/8 Proxies :-
"3.3.3.3" with "1.1.1.1"

```

The above example indicates the the on Router node "149.121.128.29" has identified 2 (out of a maximum capacity of 32) other nodes "1.1.1.1" and "2.2.2.2" both on the UDP medium. The Router only supports the UDP medium which is currently "Active" and over which 6 messages have been sent and 6 received. This Router has identified 2 other AEs (both of them Routers). The node "149.121.128.29" has itself 3 AEs (1 Router, and 1 RESOURCE TASK "task1"). Then the Router is instructed to use "1.1.1.1" as proxy for "3.3.3.3", ie all messages for "3.3.3.3" will be sent via "1.1.1.1".

11 Debugger Invoke Options

The following options may be supplied to the debugger task (`resdebug`)

<ProcessName> The name of a CMS process for the debugger

-d <FileName>

-i <CMS buffers> Buffer distribution

-o <OOT size> Outstanding Operation Table

-r <Length> Resource message queue length.

-tot <Timeout> The Read Template timeout in milliseconds

-tor <Timeout> The Read timeout in milliseconds

-tow <Timeout> The Write timeout in milliseconds

-y <Key> Resource IPC key.

12 Debugger Command Summary

Below is a full description of the debugger commands in Bakus-Naur Form.

```

task_address ::= '' [ resource_name ] ':' task_name ''
break_or_trace_args ::= [ 'entry' | 'exit' | 'either' ] 'fblock' fblock_name |
                        [ 'entry' | 'exit' | 'either' ] fblock_name |
                        [ 'entry' | 'exit' | 'either' ] 'all' |
                        [ 'entry' | 'exit' | 'either' ] 'all fblocks' |
                        fblock_name line_number

break_command ::= 'break' break_or_trace_args
connect_command ::= 'connect' task_address
delete_break_command ::= 'delete break' trace_or_break_args |
                        'delete break' all
delete_trace_command ::= 'delete trace' trace_or_break_args |
                        'delete trace' all
disconnect_command ::= 'disconnect' task_address |
                      'disconnect'
exercise_command ::= 'exercise' times [ 'read' | 'write' ] ref_name
help_command ::= 'help' | 'help' command_name
input_command ::= '<' | '<' file_name
list_connect_command ::= 'list connect'
list_properties_command ::= 'list properties' ref_name
list_ref_command ::= 'list ref' ref_name
list_refs_command ::= 'list refs'
list_resources_command ::= 'list resource'
output_command ::= '>' | '>' file_name
pause_command ::= 'pause' seconds
ping_command ::= 'ping' task_address
print_command ::= 'print' hierarchic_name
quit_command ::= 'quit'
read_command ::= 'read' ref_name
router_list_aes_command ::= 'router' node 'list aes'
router_list_local_aes_command ::= 'router' node 'list local aes'
router_list_media_command ::= 'router' node 'list media'
router_list_nodes_command ::= 'router' node 'list nodes'
router_list_medium_command ::= 'router' node 'list medium' medium_name
router_list_proxies_command ::= 'router' node 'list proxies'
router_set_proxy_command ::= 'router' node 'set proxy' node 'with' node
router_set_logger_command ::= 'router' node 'set logger'
ref_command ::= 'ref' ref_name [ '.' service_name ] '=' reference_string
scan_command ::= 'scan' ref_name scan_time
expand_command ::= 'expand' ref_name
service_command ::= 'service' ref_name '.' service_name
store_command ::= 'store' ref_name [ '.' service_name ] [ '.' element ] [ '[' index ']' ]
                '=' constant_value
trace_command ::= 'trace' trace_or_break_args
transcribe_command ::= 'transcribe' file_name | 'transcribe' '''file_name'''
unref_command ::= 'unref' ref_name
wait_command ::= 'wait' seconds
whatis_command ::= 'Whatis' hierarchic_name | 'whatis' hierarchic_name
                | 'Whatis' '''' | 'whatis' ''''
write_command ::= 'write' ref_name

```

