



**EUROTHERM**

---

# **THE RESOURCE MANAGER**

## **A Guide to Var References**


### **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

**HA024105C001 2** J Juer & M Fox



---

## VERSION HISTORY

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

---

## Contents

<b>1</b>	<b>Scope</b>	<b>4</b>
<b>2</b>	<b>Related Documents</b>	<b>4</b>
<b>3</b>	<b>Introduction</b>	<b>4</b>
<b>4</b>	<b>Var Reference</b>	<b>4</b>
4.1	Specifying the Remote Data Objects . . . . .	5
4.1.1	Simple types . . . . .	5
4.1.2	Arrays . . . . .	6
4.1.3	Function Blocks . . . . .	7
4.1.4	Services . . . . .	9
4.1.5	Enumerations . . . . .	9
4.2	Reading the Remote Information . . . . .	10
4.3	Reading Remote Data . . . . .	10
4.4	Writing Remote Data . . . . .	11
4.5	Requesting Services . . . . .	11
4.6	Multiple Outstanding Operations . . . . .	11
4.7	Thruing . . . . .	12
4.8	Messaging Resources . . . . .	12
4.9	Examining the State of a Var Reference . . . . .	13
4.10	Timeouts and Failures . . . . .	13
4.11	Summary of Properties . . . . .	14
<b>5</b>	<b>Diagnostic Function Block</b>	<b>14</b>
<b>6</b>	<b>Performance</b>	<b>17</b>

## 1 Scope

This document provides an ST programmer's guide to the use of **VAR REFERENCES** in the **RESOURCE** ( Version 2.2 ).

## 2 Related Documents

- [1] IEC1131 Programmable Controllers Part 3: Programming Languages First Edition 1993-03
- [2] HA024105C003 A Guide to Tuning the Resource
- [3] HA024105C005 A Guide to Setting Up CMS Networks

## 3 Introduction

**VAR REFERENCES** are a Eurotherm Control extension to the IEC1131-3 standard [1] to provide access to remote data on other Eurotherm Controls products.

## 4 Var Reference

Every **FUNCTION\_BLOCK** or **PROGRAM** can have a reference declaration section. All data specified in this section is remote data, where remote means that it has its defining definition somewhere other than in the current block, though it may well be in the same **RESOURCE** or indeed **TASK**. The reference section is denoted by the keyword **REFERENCE** appearing after the keyword **VAR**. Note therefore that references are internal to the block they are defined in.

For example

```
PROGRAM ex1
VAR
    writeflag: BOOL;
    id: STRING;
END_VAR
VAR REFERENCE
    remflag: BOOL;
    remid: STRING;
END_VAR
```

has a reference section which contains two remote objects **remflag** and **remid**.

Any object may be declared to be a reference, from simple variables of any type, to arrays of any type ( except arrays of **FUNCTION\_BLOCKS** ), to **FUNCTION\_BLOCKS**.

References have a set of "properties" which are predefined built in variables. Properties may be assigned or read or both. Properties cannot be wired to. Properties are used to control and monitor the reading and writing of data via the reference.

The first stage in accessing remote data is specifying where it is and then matching it to the local data. If this operation is successful then the remote data may be read and/or written.

## 4.1 Specifying the Remote Data Objects

A **VAR REFERENCE** has an associated string, the **ref** string. This is set by assigning it from within the **ST PROGRAM**, for example

```
remflag~ref := 'Res1:pid1.in';
```

or at cold start, for example

```
VAR REFERENCE
  remflag: BOOL { ref := 'Res1:pid1.in' };
END_VAR
```

The `~` tells the ST compiler that the next name is a “property” of a **VAR REFERENCE** object. Property names are predefined, and the **ref** property is the reference string. Any ST string or ST string expression may be assigned to it.

A property called **ref** can be used to read the last set reference string.

**VAR REFERENCES** may be categorised as either **SIMPLE** or **COMPLEX**. A **SIMPLE VAR REFERENCE** is one that references either a simple type or an **ARRAY** of simple types, all other **VAR REFERENCES** are complex. This implies that a **FUNCTION\_BLOCK** with a single **INPUT**, and no **INTERNALs**, **IN\_OUTs** ( and **INPUT\_OUTPUTs** ), **OUTPUTs** or **SERVICEs** is **COMPLEX**.

### 4.1.1 Simple types

For a simple **VAR REFERENCE**, that is one which is a simple built in ST type (e.g **DINT**, **BOOL**, **LREAL**) the reference string must specify the full hierarchic path ( or **VAR\_ACCESS** name ) to the object, prefixed by an optional **RESOURCE** name and “:”. The syntax is

```
simple_abs_ref_string ::= [ resource_name ] ':' name { '.' name }
```

(using the usual BNF notation where `[]` means an option, and `{ }` means 0 or more of the enclosed). **name** is any valid ST name.

The **resource\_name** is the name of a remote **RESOURCE**. If omitted the reference is to something in the local **RESOURCE**. The list of names separated by `.` is the full path to the remote object. So in the above example **Res1** is the name of the remote **RESOURCE**, **pid1** is a block in the remote **RESOURCE** which contains a variable **in**.

For local objects which are descendants of the instantiating block a relative path name can be given. The syntax is

```
simple_rel_ref_string ::= '.' name { '.' name }
```

When a reference string is assigned the **RESOURCE** will query the specified remote **RESOURCE** for information about the specified object. The information returned is

- The addressability. It is possible to address this object.
- The type of the remote object (**DINT**, **LREAL** etc.)
- The mode of the remote object (**INPUT**, **OUTPUT** etc.)

- The size of the remote object, which will be 1 for simple types and the total number of elements for an array type.
- A fast address for the remote object.
- The **TASK** that owns the remote object.
- Any write protection.

In order for reads and writes to be performed, the remote data must be addressable. In addition the type and size must match the type, size and mode (i.e **INPUT**, **OUTPUT**, internal, **IN\_OUT**) of the local **VAR REFERENCE**. In fact for simple types (i.e **VAR REFERENCES** that are not blocks) the mode of the remote object must be internal, since the **VAR REFERENCE** is itself internal.

#### 4.1.2 Arrays

For arrays of simple variables, as well as the type and mode matching, the total number of elements in the remote object must match the number in the local object. So, for example a remote 2 by 10 array would match a local 10 by 2 array. In general a remote array with 6 dimensions  $i_1, i_2, i_3, i_4, i_5, i_6$  matches a local one with dimensions  $j_1, j_2, j_3, j_4, j_5, j_6$  provided  $i_1 * i_2 * i_3 * i_4 * i_5 * i_6 = j_1 * j_2 * j_3 * j_4 * j_5 * j_6$ . Local data at position  $x_1, x_2, x_3, x_4, x_5, x_6$  would be the remote data at position  $y_1, y_2, y_3, y_4, y_5, y_6$  if the same position has been specified when the array is “flattened” into a one-dimensional array. Since  $i_6$  and  $j_6$  are the fastest varying dimensions this means

$$x_6 + i_6 * (x_5 - 1 + i_5 * (x_4 - 1 + i_4 * (x_3 - 1 + i_3 * (x_2 - 1 + i_2 * (x_1 - 1))))) = y_6 + j_6 * (y_5 - 1 + j_5 * (y_4 - 1 + j_4 * (y_3 - 1 + j_3 * (y_2 - 1 + j_2 * (y_1 - 1)))))$$

With the constraints for the array indices (e.g  $0 < x_6 < i_6$ ) this gives a unique mapping between one array and another.

It is also possible to match to single elements of an array, or to the whole of a sub-array, in exactly the same way as the Structured Text compiler allows array assignment.

For example given a remote array declared as

```
array: ARRAY[1..10,1..10] OF DINT;
```

and the local declaration

```
VAR REFERENCE
matchall: ARRAY[1..10,1..10] OF DINT;
matchpart: ARRAY [1..10] OF DINT;
matchele: DINT;
```

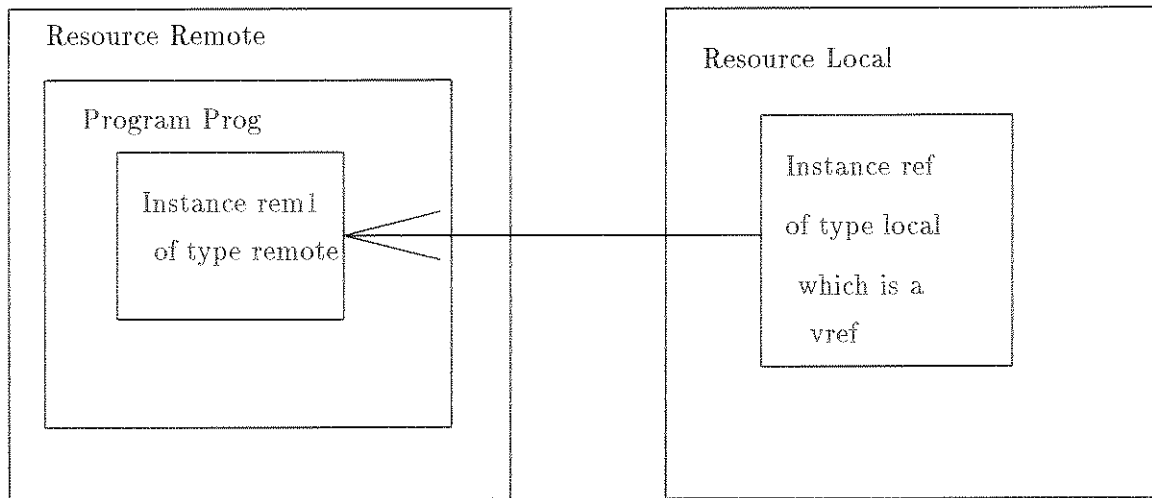
then the following would be valid.

```
matchall^ref := 'Remote:prog.array'
matchpart^ref := 'Remote:prog.array[2]'
matchele^ref := 'Remote:prog.array[1,2]'
```

So it is possible in the reference string to index into remote arrays, and have a successful match provided the dimension of the local object matches.

Note that a local array object can only have a single reference string (not an array of them).

Figure 1: A reference to a remote block



#### 4.1.3 Function Blocks

A **VAR REFERENCE** can also be a **FUNCTION\_BLOCK**. The reference string then specifies one or more remote data objects that are matched to the **INPUTS**, **OUTPUTS** and in-outs of the local object. In the simplest case where only one remote object is specified, the remote object must be a block with parameters which match the local object's parameters in name, mode, type and size except that a remote internal may match a local **INPUT**, **OUTPUT**, in-out or internal. In other words each parameter of the remote object is individually matched to the local object's parameters by name as if it were a simple type. (The remote block may have extra parameters that are not matched). The local block is then an image of (possibly part of) the remote block's data.

The diagram in figure 1 should help understanding of the example shown below. Given a remote block such as

```
FUNCTION_BLOCK remote
VAR_INPUT
    in1:LREAL;
    in2: ARRAY[1..10] OF DINT;
END_VAR
VAR_OUTPUT
    out1: BOOL;
    ignored: BOOL;
END_VAR
```

which was instantiated in a **RESOURCE** called **Remote** in a **PROGRAM** called **prog** as block instance **rem1**, and a local block definition of the form

```
FUNCTION_BLOCK local
VAR_INPUT
    in1:LREAL;
    in2: ARRAY[1..10] OF DINT;
END_VAR
VAR_OUTPUT
    out1: BOOL;
END_VAR
```

then the following VAR REFERENCE

**VAR REFERENCE**

```
    ref:local;  
END_VAR  
  
ref^ref := 'Remote:prog.rem1';
```

would match the in1, in2 and out1 parameters, and would enable data to be exchanged via those parameters.

Of course by instantiating a local instance of `remote` all of its visible parameters may be matched.

It is also possible to specify a list of remote objects that are to be matched to a local block, by using a special syntax in the reference string. Fully hierarchic names can be put in a comma separated lists, or as a shorthand '{' and '}' are used to bracket comma separated lists of names which are then all taken to be relative to the previous hierarchic name. For example the string

```
a{b,x.e,f{j,k,l{m,n}}}
```

expands to the names

```
a.b, a.x.e, a.f.j, a.f.k, a.f.l.m, a.f.l.n
```

A local parameter must be assigned to each name in the resulting expanded list, for example

```
a{ loc1 := b, c { loc2 := d, loc3 := e}}
```

means that the local parameter `loc1` is matched to `a.b`, `loc2` to `a.c.d` and `loc3` to `a.c.e`.

The full syntax of the reference string is

```
ref_string ::= simple_ref_string | complex_ref_string  
  
simple_ref_string ::= simple_abs_ref_string | simple_rel_ref_string  
  
simple_abs_ref_string ::= [ resource_name ] ':' name { '.' name }  
  
simple_rel_ref_string ::= name '.' { '.' name }  
  
complex_ref_string ::= complex_abs_ref_string | complex_rel_ref_string  
  
complex_abs_ref_string ::= [ resource_name ] ':' [ primary_name ]  
                        '{' alternate_names_list '  
  
complex_rel_ref_string ::= '.' primary_name '{' alternate_names_list '  
  
alternate_names_list ::= alternate_names { , alternate_names_list }  
  
alternate_names ::= [ hierarchic_name ] '{' alternate_names_list '}' |  
                    final_name  
  
final_name ::= local_parameter_name ':=' hierarchic_name
```



```
primary_name ::= name { '.' name }  
  
hierarchic_name ::= name { '.' name }  
  
local_parameter_name ::= name  
  
resource_name ::= name
```

The **primary\_name** is the name relative to which all the following list of names is specified. If absent the following list of names is taken relative to the remote **RESOURCE** as a whole, (i.e the list of names must contain the full path to the object). The { and } notation brackets a comma separated list of hierarchic names. Any name may itself contain a list of sub-objects using the { and } notation. At the bottom level the **local\_parameter\_name** specifies the parameter of the local **VAR REFERENCE** that is to be matched to the remote name. Any parameters of the local **VAR REFERENCE** that are not explicitly assigned remote objects are matched to parameters of the same name in the **primary\_name**; if the latter is absent this is an error.

Duplicate **local\_parameter\_names** are not allowed, but duplicate remote names are, so it is possible to match one remote variable to two or more local variables.

For example

```
VAR REFERENCE  
  RP : RemPID;  
END_VAR  
  RemPID~ref := 'R1:a{ Sp := b,c{ Pv := d, Op := e}'
```

means that **R1:a.b** must match **RP.Sp**, **R1:a.c.d** must match **RP.Pv** and **R1:a.c.e** must match **RP.Op**. If **RP** has an extra parameter **X** then it is matched to **R1.a.X**.

#### 4.1.4 Services

A **VAR REFERENCE** to a remote **FUNCTION\_BLOCK** may also reference the **SERVICES** of the remote block.

Any **VAR REFERENCE** which contains **SERVICES** must match the remote **SERVICES** **exactly**, i.e the **SERVICE** names must match, as must the names, types and order of all of the **INPUTs** and **OUTPUTs** to the **SERVICE**.

#### 4.1.5 Enumerations

If a **VAR REFERENCE** includes a reference to an enumerated type then the only matching that is done is that both are enumerations. No attempt is made to validate the match beyond that. This means that if the enumeration at the **VAR REFERENCE** is different from that at the remote end there is the potential to write an illegal value.

## 4.2 Reading the Remote Information

When a reference string is assigned to a **VAR REFERENCE**, the **RESOURCE** will work out the names of the remote objects it needs to match to the local, and send (in one message) these to the remote **RESOURCE**. Various errors may then occur. These may be found by examining the **VAR REFERENCE's status** property, for example

```
IF ref^status > 1 THEN
(* an error of some sort *)
END_IF
```

The errors associated with matching are

- A syntax or other error in the reference string
- The remote **RESOURCE** is not reachable.
- One or more of the remote objects do not exist.
- The remote objects are owned by more than one **TASK**, and so cannot be made into one reference.
- The remote objects do not match the local ones according to the rules specified above.

Once a match has been completed successfully a single read is issued to ensure that the local copy of data contains something valid.

## 4.3 Reading Remote Data

Once a reference has been matched to the remote objects, data may be read. The **scan** property sets the scan rate in milliseconds for the remote data. For example

```
ref^scan := t#10;
```

or else at cold start

```
VAR REFERENCE
  ref : local { scan := t#10 }
END_VAR
```

sets the scan rate for the reference **ref** to be once every 10 milliseconds. This means that every 10 milliseconds the **RESOURCE** will send a read message to the remote **RESOURCE** specified in the reference string to read all of the remote data. When the reply comes back the local image of the remote data is updated. A scan rate of zero means no reads are performed.

If, however, no reply is received by the time the next scan is due, no timeout occurs, and no message is resent. (Timeouts are handled separately with a separate global value). Thus specifying a very fast scan time means that the data will be read as fast as possible, being limited by the rate at which the remote **TASK** responds and the local **TASK** sends messages.

A scan is performed on each **TASK** cycle where "Time now  $\geq$  Time of last scan + Scan time" ( a scan time of  $0 = \infty$  ). Therefore to perform a ~~single-shot read~~, the scan time should be changed from 0 to a value  $\leq$  the **TASK** cycle time and then reset to 0 on the next **TASK** cycle.

A property called **scan** can be used to read the last set scan rate.

For **FUNCTION\_BLOCKS** all data is read (i.e all matched **OUTPUTs**, **INPUTs** and in-outs) and placed in the local **VAR REFERENCE**.

Accessing the data of the **VAR REFERENCE** from **ST** always returns the last data read.

The property **newRead** may be read to determine if any **new data has been read** since the last time this property was read. The reading of the property is a destructive operation.

## 4.4 Writing Remote Data

A write to the remote object is triggered by either assigning to it (if it is a simple variable) or calling it (if it is a block) passing it **INPUT** parameters as usual. A write message is generated, unless a write message is already outstanding. In this case a flag is set to indicate that another write is required when the previous write is acknowledged. In this way the latest local value is always written to the remote object. Note that writes can only occur at the rate at which the remote **TASK** acknowledges them ie every assignment does not guarantee a write.

If the **VAR REFERENCE** is a simple variable or array all the local data is sent (even if only part of an array has been written).

If the **VAR REFERENCE** is a **FUNCTION\_BLOCK** the **INPUTs** and in-outs of the block are sent. Note that the in-outs are not then read back, so the value assigned to the local in-out will be the value written and will not reflect any modifications made by the remote block.

In addition to the above constraints no data that is reported as write protected at the remote **RESOURCE** is sent in a write message.

It is possible to prevent the writing of data by setting the **dontWrite** property. Once set no data will be written to the remote objects until the property is cleared. Once cleared the write will occur at the end of the next **TASK** cycle, this is unlike a normal write where the write is issued at the time of the assignment. While the **dontWrite** property is set no incoming reads will be accepted as they would overwrite the local data.

## 4.5 Requesting Services

A **SERVICE** request is triggered by calling the **SERVICE** as if it was a local **SERVICE**. The **SERVICE** request copies across to the remote **SERVICE** all of its data ( including **OUTPUTs** ) but the response does not include the **INPUTs**.

## 4.6 Multiple Outstanding Operations

It is possible to issue requests for an operation without waiting for a previous operation to complete. The order in which the operations will be performed will be.

- Write the data.
- Issue the **SERVICE** requests. If more than one **SERVICE** on a **VAR REFERENCE** has been requested then the order in which the **SERVICES** are requested is not defined except if the first **SERVICE** is requested while there is no other outstanding operation. There may not be more than one outstanding request on any one **SERVICE**.
- Read the data.

## 4.7 Thruing

If a **VAR REFERENCE** is made to another object that is itself a **VAR REFERENCE** then the nature of the operations is slightly different from the above. Templates are read and matched in the same way except that the resolution of the data is also determined to be one of the following:

**Ultimate** This is the ultimate destination.

**SingleReference** The data resolves to a single **VAR REFERENCE**.

**MultipleReference** The data resolves to more than one **VAR REFERENCE**.

**UltimateAndReference** Some of the data is the ultimate destination and some is by reference.

Only **VAR REFERENCES** which are **Ultimate** or **SingleReference** are legal.

For the operations of writing or issuing **SERVICE** requests the data will be routed via every **VAR REFERENCE** to the ultimate data or **SERVICE**, provided that each **VAR REFERENCE** involved is satisfactorily resolved.

For operations of reading, data will be read from the first **VAR REFERENCE** in the chain whose data is newer than that held by the issuing **VAR REFERENCE** or else the read will be forwarded to the ultimate source of data. This ensures that every read request ( connections permitting ) results in a new more up to date set of data being read, which is never more than twice the `~scan` out of date. If the `~propertyProtect` property of an intermediate **VAR REFERENCE** is set to 0 then its effective scan rate may be adjusted to ensure that it reads data fast enough for all those **VAR REFERENCES** that depend on it for their data.

In all these cases it is possible to have a run-time error where it is not possible to resolve the **VAR REFERENCE** due to an intermediate **VAR REFERENCE** being in a failed state. It also possible that an operation may not be resolved because it is cyclic in nature. Both of these conditions are detected and will result in the setting of the `~status` property and the operation status to unsuccessful. Such a condition is potentially recoverable once the intermediate **VAR REFERENCES** are altered ( usually by changing `~ref` ).

## 4.8 Messaging Resources

**TASKs** in a **RESOURCE** communicate by exchanging messages. Each **TASK** has a configurable number of buffers of different sizes for messaging ( see [2] ). When a message is sent the smallest available buffer that will contain a message is chosen to send it. In addition each message sent requires an entry in an Outstanding Operation Table (OOT).

Therefore in order for a **TASK** to send a message it requires 2 resources ;

- a free buffer large enough to hold the message
- a free OOT entry.

In the event of no resources being available to perform an operation, the operation is retried on the next **TASK** cycle.

Messages to remote **RESOURCES**, however, are routed to a “router” **TASK**. This **TASK** is responsible for finding the route to the remote **RESOURCE**. ( See [3] )

## 4.9 Examining the State of a Var Reference

The **status** property of type **DINT** is available to determine the current state of a **VAR REFERENCE**. The status property has the values and meaning shown in the following table —

State	Value	Meaning
OK	0	Last operation succeeded
InProgress	1	Read, write or read template in progress
ParseFail	2	A reference string had the wrong syntax
ResolveFail	3	Local names in the reference string did not match to the local object, or were duplicated
NoResources	4	The <b>TASK</b> has no buffers left or OOT entries to send messages with, or the expanded reference string is too long
TemplateMismatch	5	The read template did not match the local one
Unreachable	6	The router was not reachable ( loaded )
BadStatus	7	Either a read template specified non-existent objects, or read failed to read the data at the remote end (though the message arrived), or a write failed to write (for example if some block <b>OUTPUTs</b> were being written)
NonUniqueOwner	8	In a read template the remote objects belong to more than one remote <b>TASK</b>
SystemError	9	This should not be seen, if seen there is a internal error in the <b>RESOURCE</b>
Cyclic	10	Operation currently results in a cycle
NotCoherent	11	Operation is currently not coherent

## 4.10 Timeouts and Failures

All read, write and read-template operations have built in timeouts, which may be altered at **TASK** load time ( see [2] ).

On a timeout the **RESOURCE** will automatically try to re-read the template of the remote objects again. This is to ensure consistency if the timeout was because a remote **RESOURCE** was reloaded.

In addition to the above each message contains a checksum for the remote **RESOURCE**. This is stored in the local **VAR REFERENCE**, and if a read or write returns with a different checksum to the local one then the remote **RESOURCE** has been reloaded between the read or write. Again the remote template will be re-read.

The automatic re-read of templates in these circumstances means that a timeout is not visible to the user via the status property, so two other properties are available to examine the success of read or write ( and **SERVICE** ) messages. These are **readStatus**, **writeStatus** and **servStatus** ST **DINTs**. These have the following states

State	Value	Meaning
OK	0	Last operation succeeded
InProgress	1	Read, write in progress
Failed	2	The last read or write failed
Undefined	3	No read or write issued with this ref string
Unsuccessful	4	Operation failed this time, but might work if re-attempted

Read and write requests are essentially asynchronous. To simulate synchronous operations a Sequential Function Chart may be used, which tests the read/write status in a transition to determine when an operation completed.

A synchronous write may be performed by writing the remote data in a step and transitioning out of the step when the **VAR REFERENCE** has **OK writestatus**.

A synchronous read may be performed by setting the **scan** property from 0 to a positive large value in a step (so that only one read will be done), transitioning out of the step when **readStatus** is **OK**, and setting the **scan** property to zero.

Normally, once a template has been matched successfully, there should not be communications errors.

**FUNCTION\_BLOCKS** will be available whose **OUTPUTs** will give information on errors and the state of the various communications interfaces, (see §5).

#### 4.11 Summary of Properties

The following table summaries the properties that a **VAR REFERENCE** has. It should be noted that access to these properties is a function of the compiler used to generate the **VAR REFERENCE** code.

Property	ST Type	Mode	Meaning	Default Value
ref	STRING	IN_OUT	Specifies the object(s) referred to (§4.1)	' '
status	DINT	OUTPUT	Monitor any errors using a <b>VAR REFERENCE</b> (§4.9)	0
writeStatus	DINT	OUTPUT	Monitor success or failure of the last write operation (§4.9)	3
readStatus	DINT	OUTPUT	Monitor success or failure of the last read operation (§4.9)	3
servStatus	DINT	OUTPUT	Monitor success or failure of the last <b>SERVICE</b> operation (§4.9)	3
scan	TIME	IN_OUT	The scan rate (§4.3)	T#0ms
dontWrite	BOOL	INPUT	Prevent writing (§4.4)	0
newRead	BOOL	OUTPUT	New data has been read (§4.3)	0
timeStamp	DATE_AND_TIME	OUTPUT	The date/time that the last operation completed	DT#1970-01-01-00:00:00
qTimeStamp	QTIME	OUTPUT	The fraction of a second over $\sim$ timeStamp	QT#0ms
propertyProtect	BOOL	IN_OUT	Prevent changing of effective scan rate (§4.7)	0

## 5 Diagnostic Function Block

There is a **FUNCTION\_BLOCK** provided which provides diagnostic statistics about all the **VAR REFERENCES** in a **TASK**.

The **FUNCTION\_BLOCK** will consist of counts of events ( possibly fleeting ) that are visible from ST and others which are not.

The **FUNCTION\_BLOCK** ( Figure 5 ) will be of the form :

**INPUTs :**

**OutputMode** Change the mode of the **OUTPUTs**. Valid modes are :

- 0 Display running total
- 1 Display running total since last **ZeroRelative**
- 2 Display total in last completed **CountPeriod**

**ZeroAbsolute** Set all values to zero. This has a global effect on all instances of this block.

**ZeroRelative** For mode 1, **OUTPUTs** are now differences from now.

**CountPeriod** Period for mode 2.

**OUTPUTs** :

**Requests** Number of requests issued

**ReqMatch** Number of read template requests issued

**ReqRead** Number of read requests issued

**ReqWrite** Number of write requests issued

**ReqServ** Number of **SERVICE** requests issued

**Responses** Number of successful responses received

**ResMatch** Number of read template responses received

**ResRead** Number of read responses received

**ResWrite** Number of write responses received

**ResServ** Number of **SERVICE** responses received

**State errors** Number of times a **VarRef** falls into an errored state.

**MatchError** Number of times a **VAR REFERENCE** falls into a error state after a read template.

**ReadError** Number of times **VAR REFERENCE** falls into error state after a read.

**WriteError** Number of times **VAR REFERENCE** falls into an error state after a write.

**ServError** Number of times **VAR REFERENCE** falls into an error state after a **SERVICE**.

**Timeouts** Request timeouts

**MatchTimeout** Number of times a read template has not received a response in the timeout period.

**ReadTimeout** Number of times a read has not received a response in the timeout period.

**WriteTimeout** Number of times a write has not received a response in the timeout period.

**ServTimeout** Number of times a **SERVICE** has not received a response in the timeout period.

**Cancelled operations** Number of times an operation was cancelled and another operation performed ( this is usually when **^ref** is set.

**MatchCancel** Number of time a read template request was cancelled.

**ReadCancel** Number of time a read request was cancelled.

**WriteCancel** Number of time a write request was cancelled.

**ServCancel** Number of time a **SERVICE** request was cancelled.

**Unresolved operations** Number of times an operation cannot be resolved ( **VAR REFERENCE** to another **VAR REFERENCE** ).

**ReadUnsuc** Number of times a read was unsuccessfully resolved.

**WriteUnsuc** Number of times a write was unsuccessfully resolved.

**ServUnsuc** Number of times a **SERVICE** was unsuccessfully resolved.

**Operation errors** Var Ref operation errors. These are errors that prevent the request actually being sent.

**OpInProgress** Current operation still in progress on a **VAR REFERENCE** when another operation was requested

**BadState** Vref was in the wrong state to perform the requested operation.

**OOTFull** The Outstanding Operation Table (OOT) was full, and therefore the request was rejected. Each request requires a free entry in an internal table, the OOT, until the response is received or the request terminates in an error or timeout.

**NoBuffers** No CMS buffers were available for the request.

**ParseFail** Syntax error in ref string. ( The number of occurrences of the status ParseFail ).

**ResolveFail** Error in contents of ref string. ( The number of occurrences of the status ResolveFail ).

**Status errors** ( Section 4.9 )

**Mismatch** Template mismatch. ( The number of occurrences of the status TemplateMismatch ).

**Unreachable** Router is unreachable. ( The number of occurrences of the status Unreachable ).

**BadStatus** Status not OK in a received message. ( The number of occurrences of the status BadStatus ).

**ManyOwners** A **VAR REFERENCE** resolves to items held owned by different **TASKs**. ( The number of occurrences of the status NonUniqueOwner ).

**System errors** ( Should not happen ).

**SystemError** ( The number of occurrences of the status SystemError ).

**Other counters** **ScanOverRun** The number of times a scan(s) is skipped as the scan rate could not be met.

Some status's are not included because they may be implied from others. These are :

**Ok** This is  $1 + \text{ResMatch} + \text{ResRead} + \text{ResWrite}$

**InProgress** This is a fleeting condition that occurs once a request is issued. This is  $\text{ReqMatch} + \text{ReqRead} + \text{ReqWrite}$

**NoResources** This is  $\text{NoBuffers} + \text{OOTFail}$ .

For every request that is issued one of the following will result :

- A valid response.
- An operation error. The request has not been sent.
- A timeout. No response was received to a request.
- The operation is cancelled.
- The operation is unsuccessfully resolved.
- A state error + status error. A response was received but the response actions were not completed due to an error.
- A state error + system error

Therefore :



---

$\text{Completed Requests} = \text{Requests} - \text{Uncompleted Requests}$   
 $\text{Completed Requests} = \text{Responses} + \text{Operation Errors} + \text{State Errors} + \text{Timeouts} +$   
 $\text{Cancellations} + \text{Unresolveds}$   
 $\text{State Errors} = \text{Status Errors} + \text{System Errors}$

Uncompleted requests consist of :

- A single outstanding request per VarRef
- Requests aborted by changing the VarRef Ref string.

## 6 Performance

The performance of **VAR REFERENCES** ( the throughput of data ) is dependent on a number of aspects :

- The quantity of data in the **VAR REFERENCE**.
- The structure of the **VAR REFERENCE**.
- The operation ( read/write ) performed.
- The availability of **TASK** messaging resources.
- The route taken to reach the remote **RESOURCE**.

In general a greater throughput of data is achieved by having fewer **VAR REFERENCES** with large amounts of data in preference in large numbers of **VAR REFERENCES** with only a small amount of data. Also operations on **ARRAYs** of data are more efficient than on the same number of individual items. **FUNCTION\_BLOCK** writes may be faster than reads as write protected data (§4.4 ) is not sent. For references to remote data which is actually contained in the same **TASK** no messaging resources are required as no message is sent, the read or write is performed directly. It is possible to tune a **RESOURCE** for its individual requirements ( see [2] ).



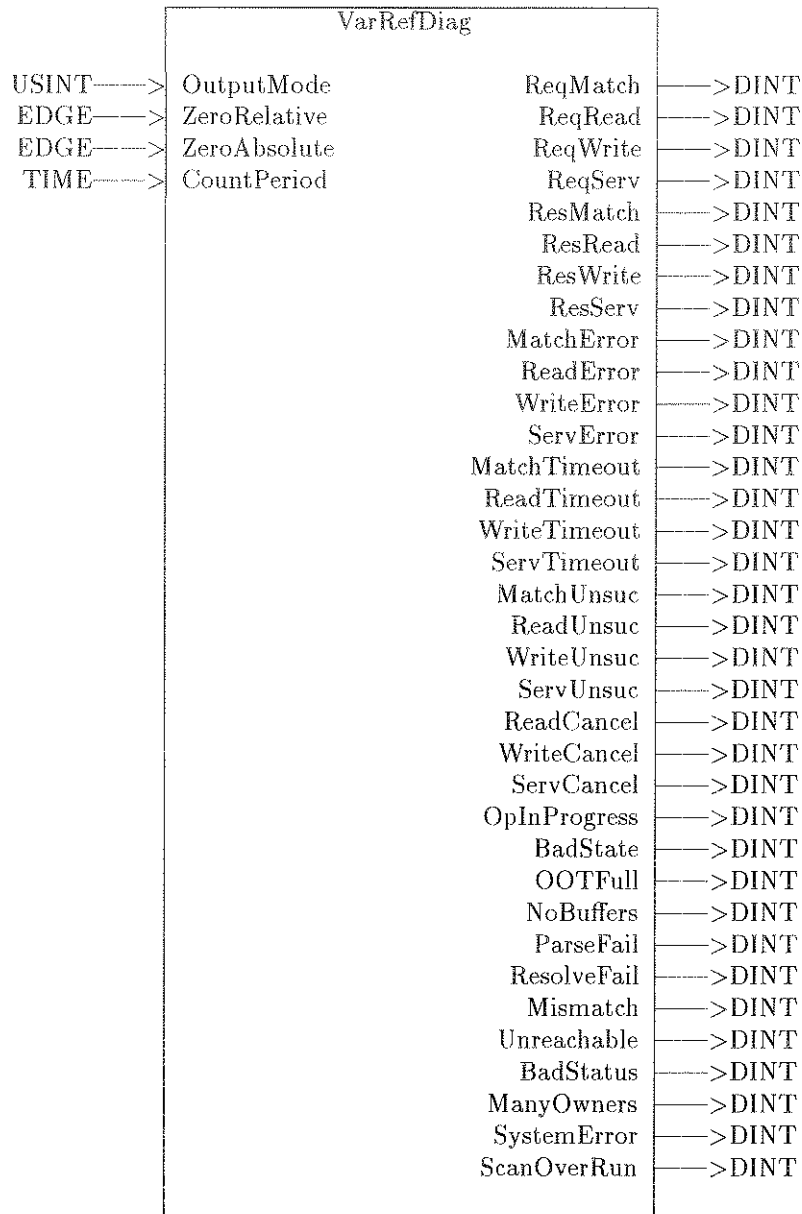


Figure 5 Var References Diagnostics