
EUROTHERM

The Resource Manager

Implementation Guide

Revision History

Revision	Date	Changes
A	February 1995	Draft (Incomplete)

Contents

TABLE OF FIGURES	6
1. INTRODUCTION	7
2. RELATED DOCUMENTS	8
3. DATABASE	9
3.1 Creation	9
3.2 Templates	10
3.2.1 Bounds	11
3.3 The Resource	11
3.3.1 Checksum	12
3.3.2 VAR_ACCESS	12
3.4 Resource Level Objects	12
3.5 Instance Data	13
3.6 GAD Tables	13
3.7 Packed ARRAYs	14
3.8 IEC1131-3 Languages	14
4. GENERAL	14
4.1 Types	14
4.2 Instance Data	15
4.3 Shape	15
4.4 Fast GADs	15
4.5 STRINGs	16
4.6 Value Descriptors	16
4.7 Doubly Linked Lists	16
4.8 Sets	16
5. GENERIC ADDRESS DESCRIPTORS	17
5.1 GAD Element Types	18

5.1.1 Numeric	18
5.1.2 String	18
5.1.3 Control	18
5.1.4 RLO_Inst	18
5.2 Fast GADs	19
6. NAVIGATION	19
6.1 The Refer Class	19
6.1.1 Refer Sets	20
6.1.2 Refer Stacks	20
6.2 The Navigation Class	20
7. THE DATABASE LOADER	20
7.1 Compiled Object to Run-Time Database Linkages	21
7.2 Tracing the Load	21
7.3 Linking Cross References	21
7.4 The Use of Files	22
8. TASKS	22
8.1 State Machine	22
8.1.1 Load State	23
8.1.2 Run State	23
8.1.3 Data State	24
8.2 The TASK Block	24
8.3 Function Lists	24
8.3.1 Order	25
8.3.2 Uniqueness	25
8.3.3 Condition	25
8.4 Debug	25
9. RESOURCE MESSAGING PROTOCOL	25
9.1 Message Objects	26
9.2 Message Types	27
9.2.1 Read Template	27
9.2.2 Simple Read	28
9.2.3 Simple Write	28
9.2.4 WriteRead	28
9.2.5 Service	30
9.2.6 Service User	30
9.2.7 Read Description	31
9.2.8 Debug Command	31
9.2.9 Debug Output	32

10. RESOURCE MESSAGE QUEUE	33
11. VAR REFERENCES	34
11.1 Outstanding Operation Table	34
11.2 Pending Service Table	35
12. EMBEDDED DEBUG	36
12.1 Debug Output	36
12.2 Debug Session	36
12.3 Ping	36
12.4 Trace and Break	36
12.4.1 Resource Manager Class Debug	36
12.4.2 RMP Messages	37
12.4.3 Source Code Debug	37
12.5 Resource Database Interrogation	37
13. ST COMPILER INTERFACES	38
13.1 Var References	38
13.2 Source Code Debug	38
13.3 SERVICES	39
14. THE SOURCE CODE	39
14.1 Where to Find It	39
14.2 Order of compilation	41
14.3 Pre-processor	41
14.3.1 Target	41
14.3.2 Options	41
15. GLOSSARY OF TERMS	43
INDEX	44

Table of Figures

FIGURE 1 - DATABASE CREATION PROCESS	10
FIGURE 2 - TEMPLATE CLASS HIERARCHY	11
FIGURE 3 - EXAMPLE OF GADs IN A RESOURCE	17
FIGURE 4 - TASK CLASS(ES) HIERARCHY	22
FIGURE 5 - TASK STATE MACHINE	23
FIGURE 6 - RMP WRITE READ REQUEST CLASS HIERARCHY	29
FIGURE 7 - RMP WRITE READ RESPONSE CLASS HIERARCHY	29
FIGURE 8 - RESOURCE MESSAGE QUEUE ENTRY CLASS HIERARCHY	33
FIGURE 9 - RESOURCE MESSAGE QUEUE CLASS HIERARCHY	33
FIGURE 10 - VAR REFERENCE CLASS HIERARCHY	34
FIGURE 11 - VAR REFERENCE SERVICE CLASSES	34
FIGURE 12 - OOT CLASS HIERARCHY	35

1. Introduction

This document contains notes on how the Resource Manager software is organised and used. It contains descriptions of the data structures and classes used.

This document is intended as a supplement to the design documentation and not to be complete in itself. The primary audience for this document is for software engineers intending to port, modify or just simply understand the code of the Resource Manager.

It is assumed throughout that the reader is familiar with the IEC1131-3 standard and the basic principles of the Resource Manager.

The Resource Manager is implemented in C++ with C interfaces for the ST compiler.

The document contains diagrammatic representations of the more complex class hierarchies and lists the source code modules related to each subject. It also includes an index which includes all class and structure definitions referenced in the document. This document is not intended to be exhaustive in its coverage of the Resource Manager code but merely to give an overview and some “pointers” into the code.

This document refers to the source code that forms the Resource Manager Version 2.2 only.

2. Related Documents

HP024105	Resource Manager	Release 1 Specification
HP024105C300	Resource Manager	Design and Architecture
HP024105C301	Resource Manager	Communications Messaging Services
HP024105C304	Resource Manager	Var References
HP024105C305	Resource Manager	GAD Format
HP024105C306	Resource Manager	Enhancements to Var References
HP024105C307	Resource Manager	Resource Services
HP024105C308	Resource Manager	Resource Transactions
HP024105C309	Resource Manager	Enhanced Resource Level Functionality
HP024105C322	Resource Manager	Communications Access
HP024105C323	Resource Manager	Interpreted Resource Levels
HA024105C001	Resource Manager	A Guide to Var References
HA024105C002	Resource Manager	A Guide to the Resource Debugger
HA024105C003	Resource Manager	A Guide to Tuning the Resource
HA024105C007	Resource Manager	A Guide to the Resource Loader
HA024105C008	Resource Manager	A Guide to the Resource Task
IEC1131-3	Programmable Controllers	Programming Languages

3. Database

The Resource database is largest and most important part of the Resource Manager.

3.1 Creation

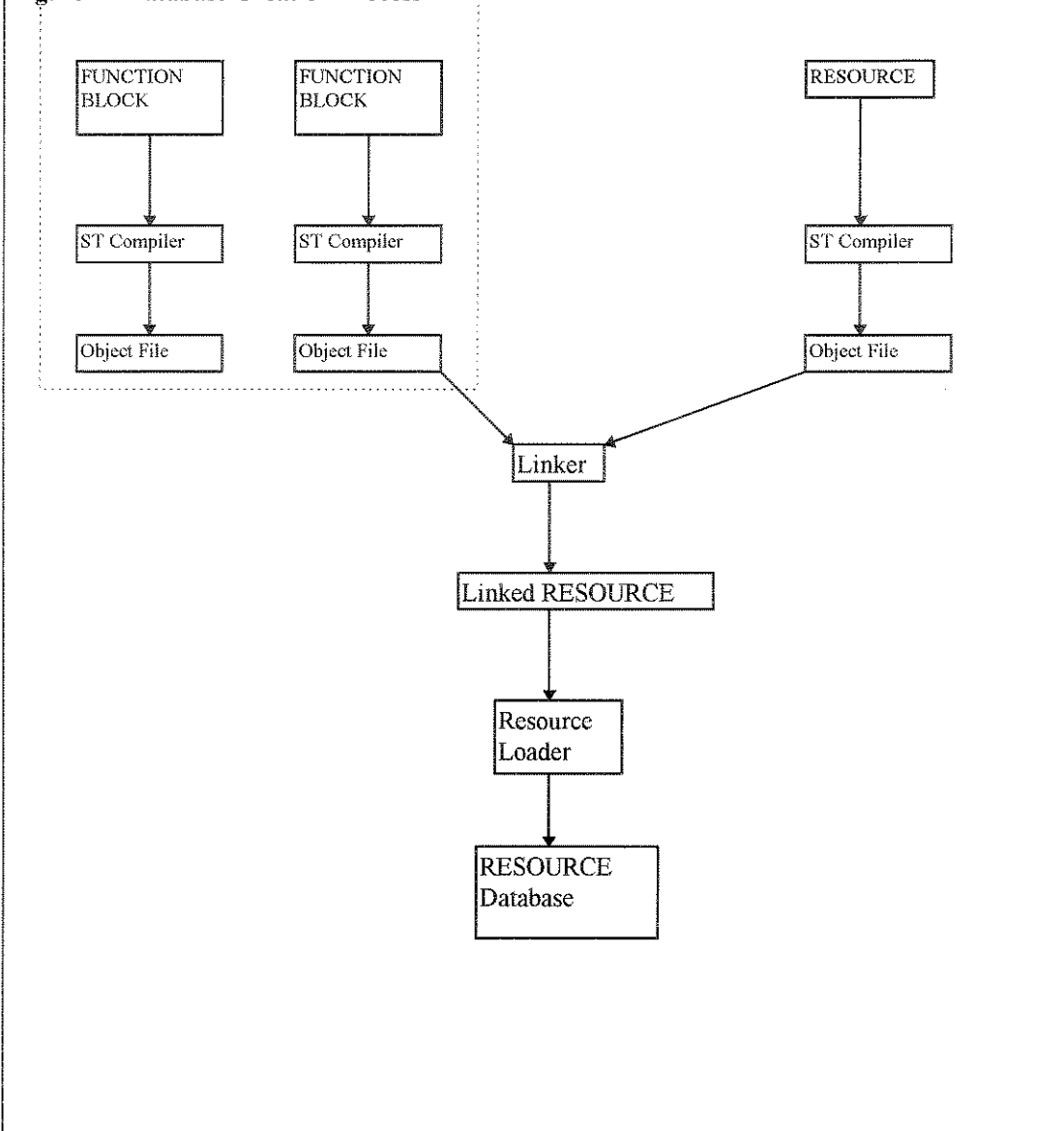
The run-time database is generated from the output of the ST compiler / GCT from a number of FUNCTION_BLOCKS to form a RESOURCE. This procedure produces a single object module containing C templates, GAD tables and the block bodies. This output is converted by the Resource Manager loader into the run-time database. This process consists of the following :-

- Generate the C++ objects in the run-time database from the ST compiler output.
- Create the instance data for the RESOURCE.
- Create all the additional run-time structures required for the RESOURCE.
 - VAR REFERENCES
 - Outstanding Operation Tables
 - Pending Service Tables
 - Resource Message Queues
 - Additional information for TASKs
 - VAR_ACCESS Tables
- Create RLO objects with method pointers back into the original object.

1 Guide to Resource Executive
2 - 68K XEC
- REY sits on top of XEC

* Supervisor
* eelin - DEAD
* timex
* xall Tasks
*

Figure 1 - Database Creation Process



3.2 Templates

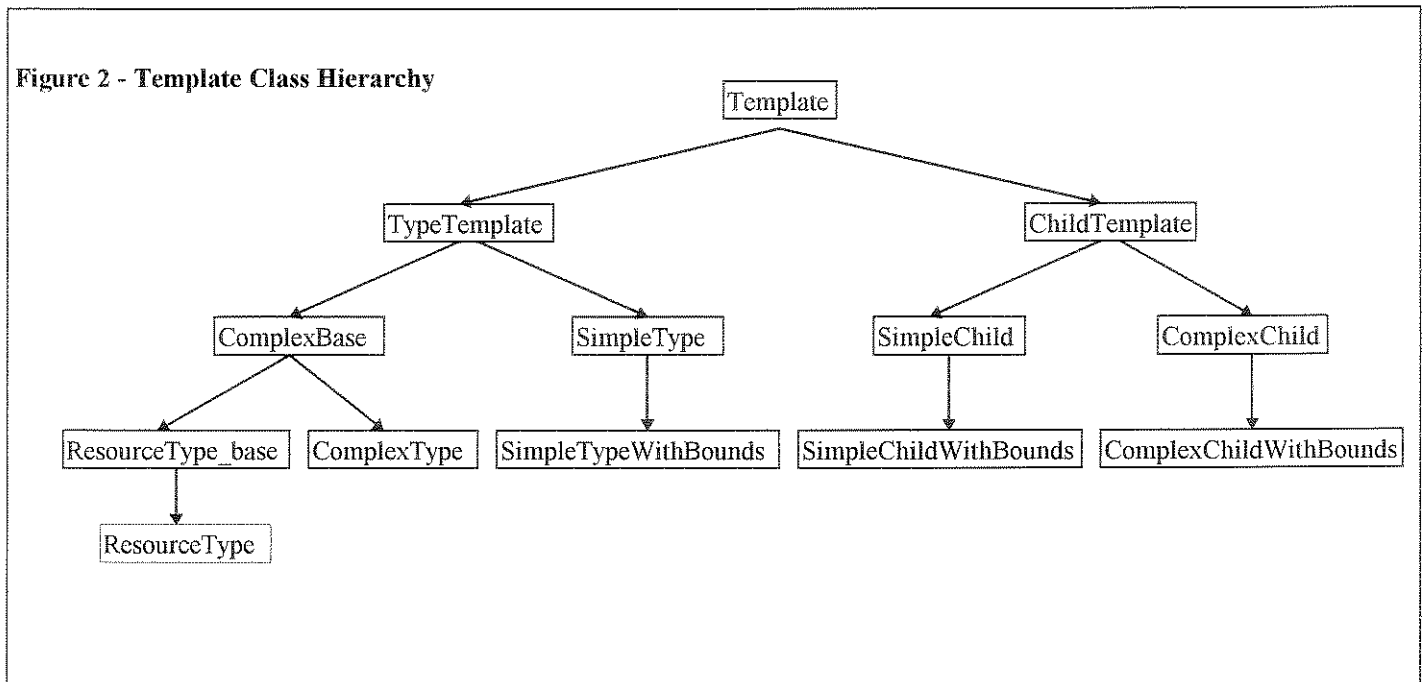
The **RESOURCE** database templates may be divided into a number of classes.

The first distinction that needs to be made is between type templates (class **TypeTemplate**) and child templates (class **ChildTemplate**). A type template describes the type and not any instance of it. The child template describes an instance of a type. It is called a child template as it must be the child of something in order for it to have been instantiated.

The second distinction that needs to be made is between simple and complex templates. A simple template is one for a base IEC1131-3 type such as **DINT** which does not encapsulate any other types or instances. A complex template is for something that contains (or could contain) other types, such as **FUNCTION_BLOCKS**.

The final component of a template that needs highlighting is bounds. A template has bounds if what it is referring to has some shape or dimensions, i.e. **ARRAYs** or **STRINGs**.

Figure 2 - Template Class Hierarchy



3.2.1 Bounds

All **ARRAYs** and **STRINGs** have their “shape” defined by a **Bounds** class. *The terminology here is a little confusing.* The rank (a maximum of 6) of a bounded item is the number of dimensions (class **Dimension**) it has. Each dimension is described in terms of a left and right-hand index.

e.g. **ARRAY [l1..r1,l2..r2]** has a rank of 2 with an overall shape of $(r1-l1)*(r2-l2)$ with dimensions $(l1,r1)$ and $(l2,r2)$.

For **STRINGs** it should be noted that they can have a maximum of 5 dimensions (i.e. maximum rank 5) as one dimension describes string size (where $l1 = 1, l2 \leq 255$).

3.3 The Resource

The whole of the database is navigated from the top from the instance of the **ResourceType** class pointed to by the global pointer **TheResource**. The shared memory area into which the whole of the database is constructed has as its first location a pointer to **TheResource**.

The **ResourceType** contains :-

- an array of RLOs (class **RLObj**).
- the **VAR_ACCESS** definitions.
- a checksum (and partial checksum calculation)

3.3.1 Checksum

The **RESOURCE** contains a checksum which is a checksum of some of the invariant data in the **RESOURCE**. This comprises all the templates.

At run time one **TASK** may verify the checksum, this is done in stages (i.e. one RLO at a time) and a partial checksum is kept, i.e. the full checksum is checked every number of RLOs * **TASK** cycle. If the checksum fails then the **TASK** unloads all other **TASK**s.

3.3.2 VAR_ACCESS

The Resource manager provides the IEC1131-3 **VAR_ACCESS** construct at the **RESOURCE** level, unlike the IEC1131-3 which provides this at **CONFIGURATION** (which is not currently supported) and **RLB** levels.

A **VAR_ACCESS** is implemented as a character string of up to 20 characters in length.

For each **VAR_ACCESS** (class **RdbAccessPath**) created the Resource Manager creates a database reference (class **Refer** , see 6.1). The implication of this is that data access via a **VAR_ACCESS** is “faster” as the **Refer** does not get re-created each time. This provides faster access across RMP.

The set of **VAR_ACCESS** (class **RdbSetOfAccessPath**) is attached to the **ResourceType**.

For each one of them a fast GAD is allocated with the RLO number = 0, and the Instance number an offset (from 1) into the **RdbSetOfAccessPath**.

VAR_ACCESS are built on top of base (class **RdbBaseAccessPath**) which it is intended can be used as the base for other types of look ups into the resource database. Each of these in turns references a instance of class **RdbAccessNode** which is just a re-badged **Refer**. To cater for the case where we have many routes into the database through different **RdbBaseAccessPaths** it does itself contain the **RdbAccessNode** but these are referenced from a poll (class **RdbSetOfAccessNode**) which are attached to the **ResourceType**. In this way only one **Refer** is instanced even if there are many “routes” to it. This in fact applies to **VAR_ACCESS** as well, if more than one **VAR_ACCESS** references the same item only one **RdbAccessNode** is created. All this is because a **Refer** is a non-trivial class and it is desired to keep down memory usage.

3.4 Resource Level Objects

Resource Level Objects (RLO) may be divided into a number of different types :-

- Resource Level Blocks (RLB), i.e. **FUNCTION_BLOCKS** or **PROGRAMS** which are instantiated in a **RESOURCE** declaration.
- **TASKs**
- **VAR_GLOBAL** declarations

A RLO is to be defined to be the unit of reloadability in the Resource Manager when this feature is implemented.

3.5 Instance Data

All the instance data for each RLO is allocated as a single contiguous block of data which reflect the data structure generated by the ST Compiler / GCT for the RLO.

The instance data for a block is divided up into the following 4 sections (in this order) :-

- An optional pointer to an array of **VAR REFERENCE** pointer for **VAR REFERENCES** that have been instantiated in this block. This pointer is assigned and the array allocated and initialised during the loading of the database.
- The SFC data. This consists of SFC step variant data followed by the standard SFC information.
- The main (and user visible) component containing the instance data for all the declared IEC1131-3 data.
- Any hidden instance data, usually only present on blocks which have bodies that have not been written in an IEC1131-3 language (i.e. C).

3.6 GAD Tables

Each RLO has a GAD table, and each complex child of the RLO or its children has an entry in the GAD table. The GAD table is an array of GAD table entries. A GAD table entry (class **GADTableEntry**) consists of the following :-

- A type template pointer
- An instance data offset, i.e. a pointer relative to the start of the instance data for the RLO.

The GAD table reflects a “flattened” view of the database. The database is flattened in such a way that it reflects how the hierarchy would be walked starting from the top and walking as far down as possible at each stage, i.e. depth, then breadth.

This is illustrated by comparing numerically a hierarchical view with a “flattened” view such as that show below :-

1	1.0
1.1	1.1
1.2	1.2
1.2.1	1.3
1.2.2	1.4
1.2.2.3	1.5
1.2.2.4	1.6
1.2.3	1.7
1.3	1.8
1.4	1.9
1.4.1	1.10
1.5	1.11

The above would also reflect full GADs against fast GADs

3.7 Packed ARRAYS

The ST compiler / GCT optimises structures of 8, 16, and 32 **BOOLs** to 1, 2 and 4 byte values respectively. In order to handle these optimisations correctly the Resource Manager includes some structures (**CDL_struct_8**, **CDL_struct_16**, **CDL_struct_32**) so that they can be manipulated in a target independent way.

3.8 IEC1131-3 Languages

Once the database is created there is no explicit reference to which of the IEC1131-3 languages (ST, SFC, FBD, LD, IL) was used to generate a block. As everything (ST, FBD, SFC) is reduced to ST by GCT it all referred to as ST. It is possible to identify a block as being an SFC by virtue of having STEP blocks inside but otherwise no information is available.

4. General

This section describes some general constructs used throughout the Resource Manager.

4.1 Types

The following basic types are used throughout the Resource Manager.

VarType - This is the type of an instance data item. This takes values of the form **tagDint**, **tagUsint**, **tagFbblock** etc.

VarMode - This defines the mode of an item. This contains a number of fields :-

MODE - This defines the IEC mode of the data and takes one of the following values :-

modeGLOBAL - An item defined in a **VAR_GLOBAL** section

modeINPUT - An item defined in a **VAR_INPUT** section

modeOUTPUT - An item defined in a **VAR_OUTPUT** section

modeINOUT - An item defined in a **VAR_IN_OUT** section

modeINTERNAL - An item defined in a **VAR** section

modeINOUTP - An item defined in a **VAR_INPUT_OUTPUT** section, a Eurotherm extension.

modePARENT - An item whose mode is defined by its parent or first non **modePARENT** ancestor.

modeEXTERNAL - An item defined in a **VAR_EXTERNAL** section

REF - This defines if the item is a **VAR REFERENCE**

WriteProtect - This defines if the **writeProtect** attribute was assigned from the programming tool.

Rendezvous - If the item is a **SERVICE** then this defines if that **SERVICE** is a Rendezvous, i.e. the **ACCEPT** keyword has been used.

Parent Is REF - This is a run-time bit that is only set in response to a RMP read template request if an ancestor of the data item is a **VAR REFERENCE**

RETAIN - This bit is set if the item is specified using the **RETAIN** keyword.

Bits	7	6	5	4	3	2	1	0
Usage	RETAIN	Parent Is REF	Rendezvous	Write Protect	REF	MODE		

4.2 Instance Data

There are 2 basic general purpose structures for holding instance data, both of which are unions of all the basic IEC1131-3 types. These are the **CDL_Any_P** and the **CDL_Any_S** unions, each of which holds a single value. There is only one difference between the **CDL_Any_P** and the **CDL_Any_S** is the way in which STRINGS are handled. In an **CDL_Any_P** the union holds a pointer (hence **_P**) to the string and the **CDL_Any_S** actually holds the value in a string buffer in the union.

In addition to the **CDL_Any_P** and **CDL_Any_S** structures there are **Value_P** and **Value_S** structures which are encapsulations of the **CDL_Any_** structures with the type (**VarType**) of the data element as well.

4.3 Shape

The type **VarShape** (a unsigned 16 bit integer) defines how many data items are being referenced. This is most typically used to indicate the number of elements in an **ARRAY**, or else is 1 when the item is not an **ARRAY**.

Examples :-

ST	Shape
ARRAY[1..4] OF DINT	4
ARRAY[1..4,1..2] OF DINT	8
DINT	1
STRING	1

4.4 Fast GADs

A Fast GAD (**FastGad_S**) is a quick way of referring to a database item by exploiting the “flattened” nature of the database. The fields of a fast GAD are :-

Field	Type	Description
RLO_Inst	RLO_InstIx	RLO number and “flattened” child instance number
Elem	ElementIx	Child number
ElOrIx	FlatIndex	A flattened array index

For a full description of fast GADs see 5.2.

4.5 STRINGS

IEC 1131-3 **STRINGS** are stored as an array of bytes where the first byte defines the length of the string, the subsequent bytes are the string contents and then finally a **NULL** byte is appended. This last **NULL** byte is appended so that in contexts where it is known that the string contains a text string then normal C string handling can be applied.

4.6 Value Descriptors

A value descriptor (**ValDesc_S**) is a concise form of the template and addressing information for a database item. This contains the following information :-

Field	Type	Description
vtype	VarType	Data type
vmode	VarMode	Data mode
shape	VarShape	Number of data items referred to
Fastgad	Fastgad_S	How to get to it quickly

4.7 Doubly Linked Lists

One of the few general purpose classes employed in the Resource manager is a doubly linked list and an associated iterator.

The doubly linked list (class **static_dlist**) is composed of a number of links (class **dlink**). This is a fairly standard pair of classes except that the **dlink** also contains an entry pointer which allows the link to "point" to something other than itself.

An iterator class (**dlist_iterator**) is also provided to iterate around the linked list.

4.8 Sets

There is a simple base class for implementing sets of objects (class **SetOf**) with a maximum capacity of 64k items. This is primarily used for generating sets of values (as **Value_P**) (class **SetOfVals**) with an associated iterator (class **SetOfValsIterator**) and sets of value descriptors (class **SetOfValDescs**).

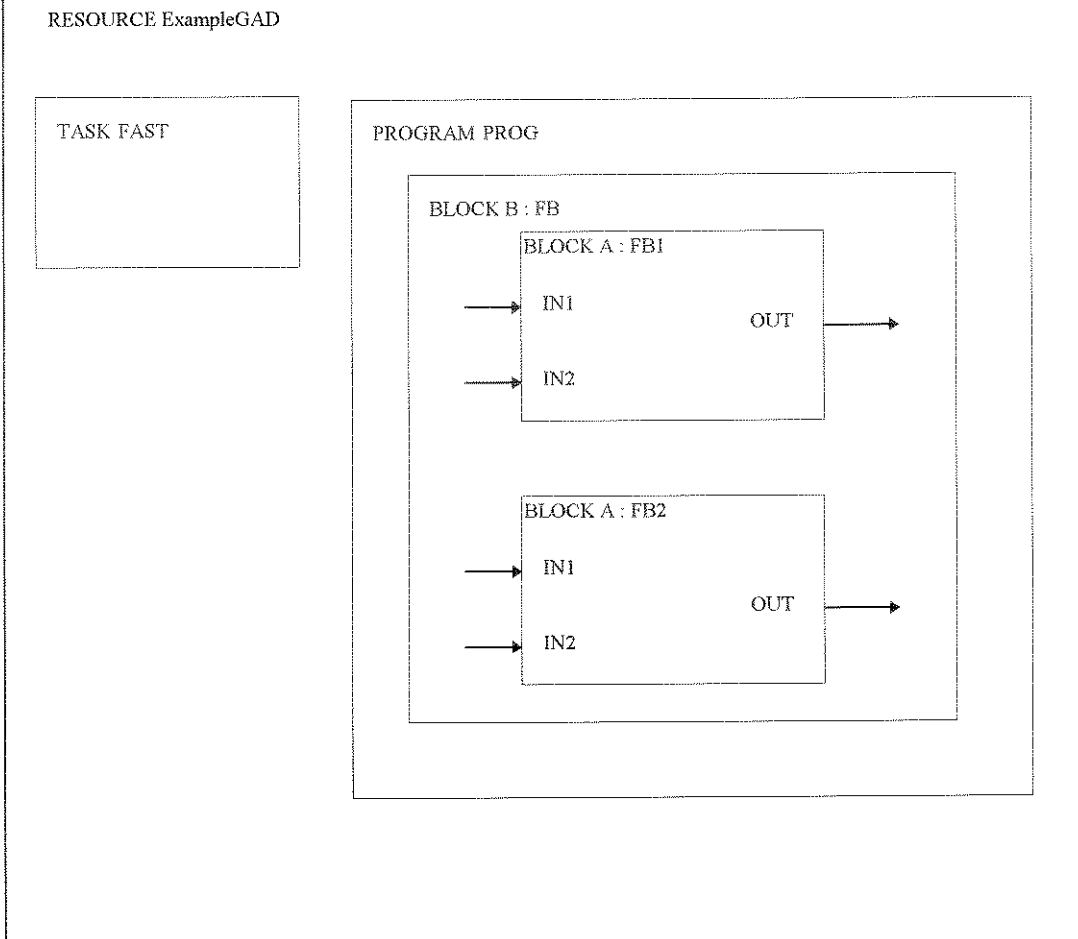
The primary uses of these sets is for RMP messages.

5. Generic Address Descriptors

Every IEC1131-3 item in a **RESOURCE** database may be referenced using a generic address descriptor (GAD). A GAD is a sequence of strings and numbers which together form a path from the top of the database (the **RESOURCE**) to the item.

To illustrate the ways GADs can be used consider the **RESOURCE ExampleGAD**. This contains a single **PROGRAM** with a single instance of **FUNCTION_BLOCK** B which itself has 2 instances of **FUNCTION_BLOCK** A, names **FB1** and **FB2**. The output from the second block could be referred to as **PROG.FB.FB2.OUT** or else as **2.1.2.3** as **PROG** is the second RLO of the **RESOURCE**, **FB2** is the second block in **PROG** and **OUT** is the third child of the block type **A**.

Figure 3 - Example of GADs in a RESOURCE



The **GAD** class is created into another class a **SetOfAplGAD** which is defined as an open array of GAD elements (class **AplGADEI**). The GAD may be formed either by directly using the information already present in the **SetOfAplGAD** or else by adding individual GAD elements or else from a value descriptor (**ValDesc_S**).

5.1 GAD Element Types

A GAD is composed of a sequence of a number of different GAD elements (class **AplGADEI**, a union of all the element types). The GAD element types are :-

- Numeric - Child number from current block.
- String - Part of a hierarchical name
- Control - For indexing into arrays
- Combined RLO and Instance

5.1.1 Numeric

A numeric GAD element is simply the (16 bit) child number of the parent block, or the RLO number if it is the first element (i.e. the **RESOURCE** is the parent).

5.1.2 String

A string GAD element is a C string hierarchical name in standard form (e.g. **a.b.c.d**). This is the one GAD element type that form a complete reference to any data item in its own right.

5.1.3 Control

There are a number of different control GAD types defined but only 2 are used, these are :-

- Flat index - Flattened array index. This requires 2 numeric values, one for the index, and the second for the number of elements referenced.
- Index - Normal array index. This is a single numeric.

A control GAD element is always contains more than one value. It is formed with a control GAD element type followed by a number of numeric (16 bit) arguments.

5.1.4 RLO_Inst

This is the first element in a fast GAD, although can be used as the first element in a full GAD. It *must* be the *first* element. This 16/32 bit GAD element can be in one of 4 flavours :-

Bits	15-14	13-10	9-0
Usage	0	RLB	Instance

Bits	15-14	13-7	6-0
Usage	1	RLB	Instance

Bits	15-14	13-4	3-0
Usage	2	RLB	Instance

Bits	31-16	15-14	13-0
Usage	Instance	3	RLB

This element can be composed and decomposed using the **ApInstGAD** class. This has 2 constructors one for the single 16 bit value and the other for the separate RLB number and instance numbers.

5.2 Fast GADs

A fast GAD is an index into the flattened form of the database and consists of a block reference (consisting of an RLO number and a flattened index) followed by the element number and optional index number (for **ARRAYs**).

In the above example **PROG.FB.FB2.OUT** would become 2.3.3.

6. Navigation

This section describes the facilities available to navigate the database.

6.1 The Refer Class

The Refer class is the primary means by which access to the database is made. Making a reference to an item in the database is made by constructing a Refer from a set of GADs. Once the construction is complete the Refer will effectively point to the database item and all the associated information, such as :-

- The instance data
- The type template
- The child template
- The GAD table entry
- The TASK association

The Refer class is designed to be constructed once and then re-used to obtain several items of information or the same information again (i.e. re-reading instance data).

Construction of a **Refer** is strictly “top down” i.e. there are no methods to migrate to siblings or ancestors of the current reference.

6.1.1 Refer Sets

Certain features require operations on or to deal with more than one reference. In order to satisfy these requirements there is a **SetOfRefer** class. This class is primarily used in the **VarRef** class to refer to local values and also in the message processing of a read template request. In particular it is able to determine if data reference by a set of **Refer** is in fact **TASK** coherent.

6.1.2 Refer Stacks

In order to process read template requests the class **ReferStack** was developed. This essentially parses the string in a read template request (at least one item referenced by the string). The “stack” is stacking the “{“ and “}” in the string.

6.2 The Navigation Class

The navigation class (**ResourceNavigObj**) is effectively a wrapper around the **Refer** class. The **ResourceNavigObj** provides a “higher level” interface to the same functionality including textual representations. It also includes navigation both “sideways” to siblings and “upwards” to ancestors of the current context.

7. The Database Loader

The tool that creates the run-time database from the compiled output (see 3.1) from the ST compiler / GCT is known as the “Resource Loader”. This creates the database into an area of “shared memory” (on some targets the compiled output would not be generally accessible).

The process of creating the database (as performed by the class **RldObj**) consists of walking the compiled (C) structures and then creating the equivalent C++ structures. It then requires all the additional structures that are not generated by the compiler to be added into the database. The actual sequence of creation is the following :-

- Allocate the shared memory area
- Pointer to the **RESOURCE**
- Array of **RLObj** pointers
- **RLObj**, all the RLO contents :-
 - Templates,
 - Instance Data
 - GAD Table
- **ResourceType**
- Resource Level Wiring
- **VAR REFERENCE** parser (one per **TASK** that has **VAR REFERENCES**)
- Create and initialise the **VAR REFERENCE** (including pointers to arrays of **VAR REFERENCE** pointers held in the instantiating blocks instance data).
- **VAR_ACCESS**
- **OOT** - Optional (may be created by the **TASK**) - default size is the number of **VAR REFERENCES** plus number of **VAR REFERENCES** to **SERVICES**
- **PST** - Optional (may be created by the **TASK**) - default size is the number of **SERVICES**

7.1 Compiled Object to Run-Time Database Linkages

The process of creating the database from the ST compiler / GCT generated compiled object does not completely de-couple the 2 from each other. There remain some mandatory linkages and some optional ones. The mandatory linkages are :-

- The block bodies (i.e. methods) are referenced from the run-time database.

Optional linkages are dependent (selection of which linkages is a Resource loader option , the default being the maximum safely permissible on the target - to optimise memory usage) on the target itself and are restricted (normally) by memory protection (particularly on targets such as unix). These linkages are :-

- Template names. Where possible the run-time database makes references to the text strings in the compiled output rather than generate these strings again. The one exception to this being the **ResourceType** which always has its name created anew.
- Templates. Once loaded the compiled template become obsolete unless a reload from the same object is required. The memory space once occupied by the C template can be reused by the C++ class in most instances (as they are the same size).

7.2 Tracing the Load

The loader has the facility to trace the creation of objects by name/type and/or memory location. this facility is only intended as a loader debug facility which may be compiled out (by **#defining NO_TEXT**).

7.3 Linking Cross References

When the ST compiler / GCT compiles the **RESOURCE** it only generates one copy of each type template and GAD table and forms the linkage through the linker. The loader is unable to do this as it walks the compiled output but an equivalent mechanism is required to eliminate the duplication of

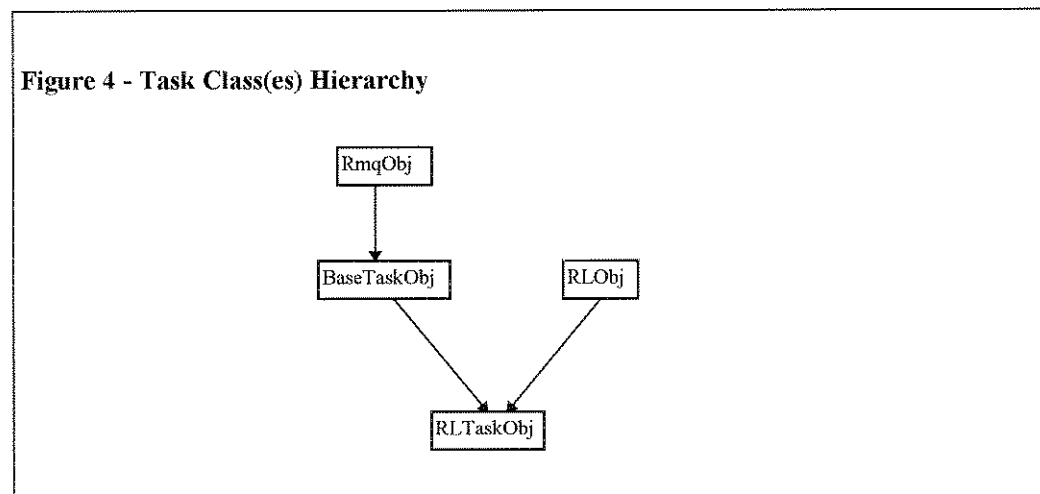
data structures. In order to achieve this the loader maintains a cross-reference table of all complex type templates and GAD tables. This cross-reference is created at the “end” of the shared memory section allocated for the database and grows downwards towards the shared memory that is actually being used for the database. For virtually every application enough memory is required after this process in the loading is complete that the 2 never overlap (checking is put in to ensure that the load aborts should this condition occur). In the very few (very small in size, and usefulness) applications where an overlap could occur some additional shared memory would be required for the cross-references¹.

7.4 The Use of Files

It is possible to create a **RESOURCE** database out of a library² of RLOs with the instantiation of those RLOs coming from a file in the form of an **ST RESOURCE** declaration. All this processing is handled by another loader class (**RldFileOfST**).

8. TASKs

The term “task” can refer to 2 (related) classes of task, the first is the IEC1131-3 **TASK**, which is the normal **TASK** that we refer to (this is implemented as the **RLTaskObj** class), the second is a base task from which this is derived (class **BaseTaskObj**). This is used for tasks that are required to support the facility of RMP but are not implemented as IEC1131-3 **TASKs** (e.g. the Resource Debugger).



Each **TASK** can locate its **RLTaskObj** from the global pointer **MyTaskObj**, and its **BaseTaskObj** from the global pointer **MyBaseTaskObj**

8.1 State Machine

The state machine for a task has 3 components which are :-

- Run State
- Load State
- Data State

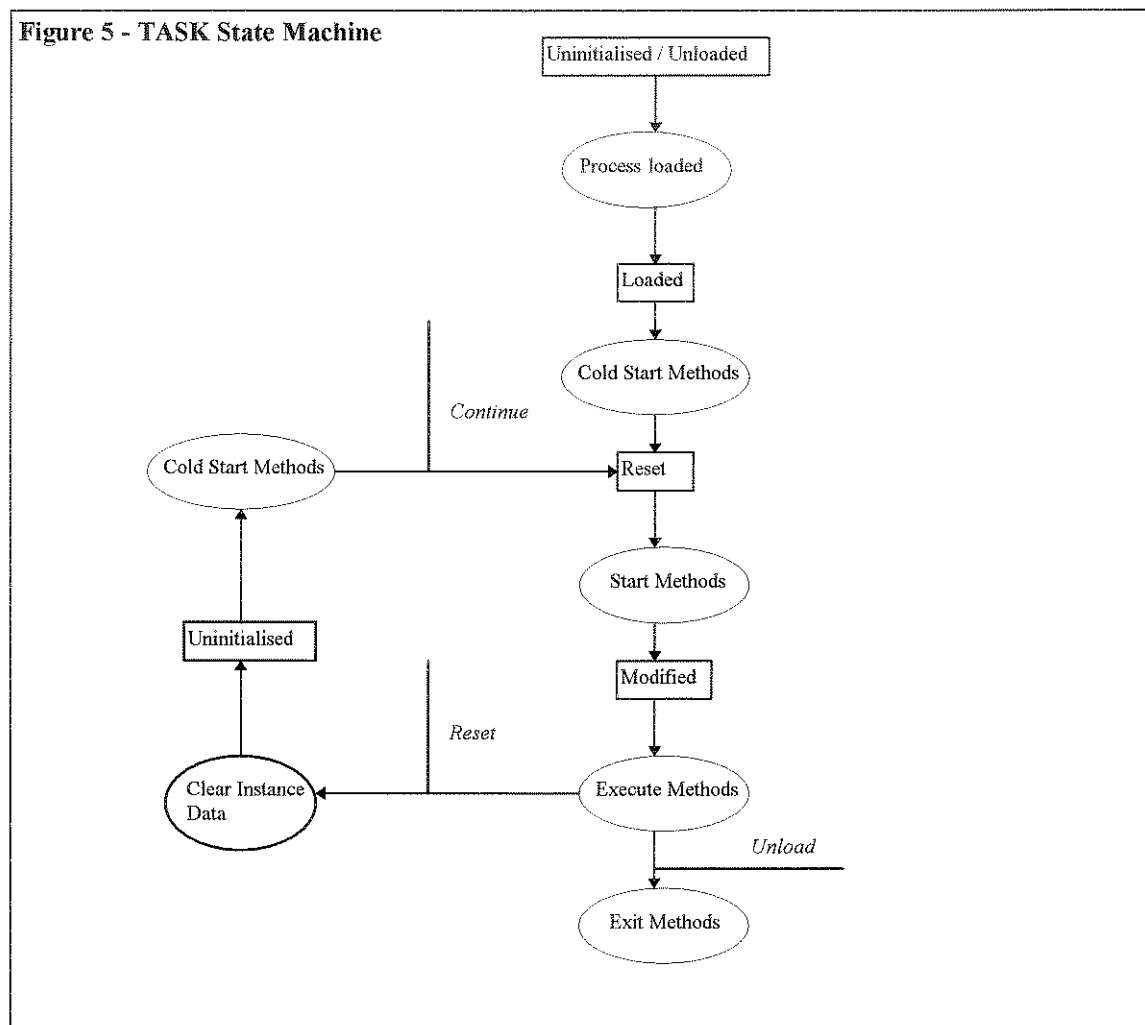
There are a number of key transitions that affect the state machine. These are :-

¹ No such application has yet been created as it would have to be trivial.

² The actual concept of a library is as yet still immature, but it is essential another compiled **RESOURCE**. How libraries are used is very target dependent.

- **Stop** - Stop the **TASK** execution.
- **Start** - Restart the **TASK** execution
- **Suspend** - Stop the block executions
- **Resume** - Restart block executions, this will only happen if the **TASK** is not also stopped
- **Reset** - Start the reset process. This resets all instance data to the cold start values.
- **Continue** - Continue the reset process started by Reset. This allows for external setting of instance data after Reset.
- **Unload** - Unload and terminate the **TASK**

All of the above transitions are executed by the **TASK** state machine and are **TASK** coherent, i.e. they only take effect at the start of a **TASK** execution before any blocks are executed.



8.1.1 Load State

The load state (class **TaskLoadStateObj**) is the simplest component of the state machine and simply indicates if the task process is actually running, i.e. it has 2 values **Loaded** and **Unloaded**

8.1.2 Run State

The run state (class **TaskExecuteStateObj**) is the most important state as this defines how the **TASK** is (or will once loaded be) executing. This has 3 values :-

- Running - This implies that all the functionality of the **TASK** is executed.
- Suspended - This implies that the **FUNCTION_BLOCKS** of the **TASK** do not execute but all the other system functions of the **TASK** e.g. **VAR REFERENCES**, **RMP** still run.
- Stopped - This implies that none of the functionality of the **TASK** is executed, it simply waits for a request to change state, i.e. the state machine engine still runs.

8.1.3 Data State

The data state (class **TaskDataStateObj**) (which is very much a function of the run state) has 3 values :-

- Uninitialised - This is the state after the database has been loaded but before any cold starts have been applied, i.e. bytes of instance data are 0.
- Reset - This is the state once cold start has completed.
- Modified - This is the state once **FUNCTION_BLOCK** execution, or **RMP** has begun.

8.2 The TASK Block

For each target supported by the Resource Manager a **TASK** block is required. This implements the **TASK**. There may only be *one* type of **TASK** in any **RESOURCE**. A **TASK** is normally expected to be a super-set of the IEC1131-3 **TASK** model.

The fixed methods for the **TASK** are :-

TASK_cold_start (p)

This is called at cold start time with the argument **p** being the instance data for the **TASK** block.

TASK_execute (p)

This is called every **TASK** execution with the argument **p** being the instance data for the **TASK** block. This will normally implement the **INTERVAL** and **SINGLE** block inputs.

TASK_exit ()

This is called when the **TASK** terminates.

8.3 Function Lists

Associated with each **TASK** are a number of lists of functions that are to be executed at fixed points. Each list contains an ordered list of functions (and single arguments to those functions). The fixed points at which lists are executed are :-

- **Cold Start** - after function block cold start
- **Start** - after function block cold start and before the first execution of the **TASK**
- **Execute** - on every **TASK** execution.
- **Exit** - on **TASK** termination and exit.

There is by default a small (10) pool of list entries (structure **tfl_entry**) allocated to each **TASK** of which all but one are available to any “user” but in general it expected that each “user” allocated their own structure and adds it in to the free list.

Each **tfl_entry** contains the following :-

- An ordering (see 8.3.1)
- An function pointer
- An argument to the function
- A statement of conditionally (see 8.3.3)

8.3.1 Order

A function (and argument) may be placed in a list in order. Each function that is placed in a list is given an ordering (a number from -128 to 127) where the lowest numbered functions are executed first. In the case of the execute list 0 represents the point in the execution cycle where the function blocks associated with that **TASK** are executed. If more than one function is placed into a list with the same ordering then the first one entered is executed first.

8.3.2 Uniqueness

When a function is placed into a list (or removed) a definition of the “uniqueness” of the entry is used. There are 3 types of uniqueness :-

- A unique combination of function address and argument.
- A unique function address - this implies that the function can only be entered once in the list.
- No uniqueness - this implies the entry will always succeed.

8.3.3 Condition

When a function is placed in a list it is also specified whether or not is executed if a specific condition is true. The condition is “is the **TASK** running, i.e. not stopped or suspended” in the case of all the lists.

8.4 Debug

The **RLTaskObj** also contains classes to implement the RMP debug (see 12) with contained classes for trace (class **Trace**), break points (class **BreakPointClass**) and Resource Manager class debug (class **ClassDebug**).

9. Resource Messaging Protocol

The Resource Messaging Protocol (RMP, also known as RDP) is a CMS protocol that provides the following facilities :-

- Remote data access (e.g. from **VAR REFERENCES**, see 11)
- Debugging facilities (see 12)

-
- Database structure interrogation

The use of RMP can be provided by instantiating a **RmpObj** class (this is in fact done in the **RmqObj** class, see 10). The **RmpObj** is a simple class that contains encode and decode methods for all the RMP message types. By changing an encode or decode method to a null method that capability is effectively disabled. The **RmpObj** constructor automatically fills in encode and decode methods for all RMP message types except the debug types (command and output).

Each RMP message is constructed into a CMS buffer of the required size using a message type specific constructor. Each message structure and message type object is described in the following sections.

9.1 Message Objects

This section describes some objects and types that are used by more than one message type.

Time stamps are defined (structure **RmpTimeStamp**) as an IEC1131-3 **DATE_AND_TIME** (in seconds) together with a **QTIME** (fraction of a second ³).

All data is sent/received as a sequence of data items (an **ARRAY** is several items) (in a class **RmpFlatDataObj**).

Sets of fast GADs (for read template only) are sent as class **RmpFastGadSetObj**.

Sets of GADs are (mostly) sent as instances of class **RmpGADSetObj** which is an encapsulation of a **SetOfAplGAD**.

Status is a **RmpStatus** which takes values **RmpStatusOk** and **RmpStatusFailed**. In message types that requires multiple status these are sent in an instance of class **RmpStatusSet**.

Value descriptors (for read template only) are sent in sets as class **RmpValDescSetObj**.

RESOURCE database checksums are sent as **RmpResourceChecksum** types.

When the facility to forward messages on (this is instances when a **VAR REFERENCE** is to another **VAR REFERENCE**, ...) is required each step in the chain of references is contained in a **RmpThruRecord** which contains the fast GADs of the **VAR REFERENCES** and the **RESOURCE** name. These are grouped in a set (class **RmpThruRecordSetObj**).

In order to determine if a message required forwarding a “resolution” of a reference is used. This is modified at each step of the chain from the **VAR REFERENCE** to the final source of the data. This has the following values :-

- **RmpTemplateUltimate** - No forwarding required (this is the normal case).
- **RmpTemplateSingle** - The reference is to another **VAR REFERENCE**.
- **RmpTemplateMultipleReference** - The reference is to data contained in more than one **VAR REFERENCE**. (This case is not currently supported).
- **RmpTemplateUltimateAndReference** - The reference is to data contained in one or more **VAR REFERENCES** and also to the actual data itself. This case is not currently supported).

³ This is always 0.

Whenever the possibility of forwarding may occur it is possible that any message not be able (temporarily) be able to complete and therefore such message carry an **RmpOperationStatus** to indicate if a problem has occurred. This takes one of the following values :-

- **RmpOperationOk** - No problem encountered
- **RmpOperationNotCoherent** - one of the **VAR REFERENCES** in the chain may currently not be resolved and the message will (possibly temporarily) fail .
- **RmpOperationCyclic** - the **VAR REFERENCES** may wrap back on themselves

9.2 Message Types

All messages have a common header which comprises 3 components :-

- **RmpReqId** - Request Id to uniquely identify the message. This is supplied in the constructor to all requests and should be returned in all responses.
- **RmpMsgType** - The type number of the message
- **RmpMsgVersion** - The version number of the message. All messages are generated with the **RmpCurrentVersion** (0). This is to allow modification of the RMP and to provide backwards compatibility if the need arises.

9.2.1 Read Template

The read template message is designed for use by **VAR REFERENCES**. It returns descriptions of remote objects (as **ValDesc_S**, value descriptors). This provides sufficient information for template matching and provides a fast GAD for future use.

The read template request (class **RmpReadTemplRequestObj**) can be constructed in of 2 ways :-

- From a character string, for normal template requests.
- From another template request when a template request is forward to the proxy **TASK** on another processor within the **RESOURCE** for processing.

The request itself just contains a single C string describing the template to be read.

The read template response (class **RmpReadTemplResponseObj**) is generated by the proxy task and contains the following :-

- A checksum used for validation of future requests and responses.
- The task id of the **TASK** that owns all the items in the template request, this is 0 if the data is not all owned by a single **TASK**, this implies that the data is not coherent.
- The address to which future requests for this data should be addressed. This will be the owning **TASK** if coherence can be maintained or (normally) the proxy if not.
- A status indicating if all the information in the template could be located or not.
- Value descriptors (**SetOfValDescs**) for all items in the template.

9.2.2 Simple Read

The simple read message is designed for reading a single data item (including an **ARRAY**).

The read request (class **RmpSimpleReadRequestObj**) may be constructed in one of 2 ways :-

- From a single character string (which converts to a string GAD).
- From the fast GAD contained in a value descriptor.

The read request simple contains a set of GADs (as a **SetOfAplGAD**).

The read response (class **RmpSimpleReadResponseObj**) contains the following :-

- A checksum, which should match that returned in the initial read template response.
- A time stamp.
- The data (as a **RmpFlatDataObj**).
- A status indicating if the operation was successful

9.2.3 Simple Write

The simple write message is designed for writing a single data item (including an **ARRAY**).

The write request (class **RmpSimpleWriteRequestObj**) may be constructed in one of 2 ways :-

- From a single character string (which converts to a string GAD).
- From the fast GAD contained in a value descriptor.

The write request simple contains :-

- A set of GADs (as a **SetOfAplGAD**).
- The data (as a **RmpFlatDataObj**).
- A checksum, which should match that returned in the initial read template response

The write response (class **RmpSimpleWriteResponseObj**) contains the following :-

- A status indicating if the operation was successful
- A time stamp.

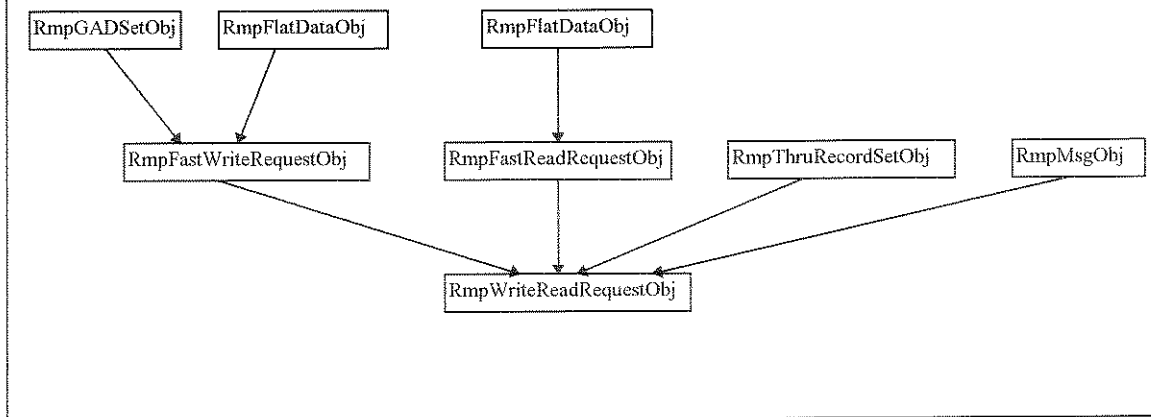
9.2.4 WriteRead

The write then read messages is designed for complex operations that require reading and/or writing of a number of data items. It also incorporates some additional information not provided in the simple messages and so may be used for single items as well.

The write read request (class **RmpWriteReadRequestObj**) contains :-

- A checksum, which should match that returned in the initial read template response
- A **RmpThruRecordSet** for forwarding.
- A read set (class **RmpFastReadReqObj**) which contains :-
 - A set of GADs (as a **RmpGADSetObj**).
- A write set (class **RmpFastWriteReqObj**) which contains :-
 - A set of GADs (as a **RmpGADSetObj**).
 - A set of data (as a **RmpFlatDataObj**)
- A time stamp (for use in forwarding reads).
- The “resolution” (**RmpTemplResolution**) of the reference

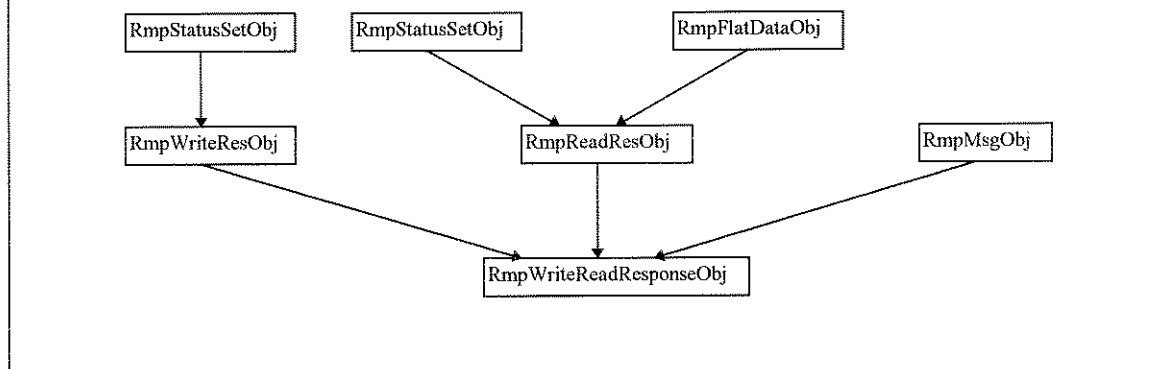
Figure 6 - RMP Write Read Request Class Hierarchy



The write read response (class **RmpWriteReadResponseObj**) contains :-

- A time stamp
- A read set (class **RmpFastReadReqObj**) which contains :-
 - A set of data (as a **RmpFlatDataObj**)
 - A set of status (one for each GAD) (as a **RmpStatusSetObj**)
- A write set (class **RmpFastWriteReqObj**) which contains :-
 - A set of status (one for each GAD) (as a **RmpStatusSetObj**)

Figure 7 - RMP Write Read Response Class Hierarchy



9.2.5 Service

The service message is designed for requesting a **SERVICE** from a **SERVICE** provider.

The service request (class **RmpServiceRequestObj**) contains :-

- The checksum returned from a previous read template request.
- The “resolution” (**RmpTemplResolution**) of the **SERVICE**.

-
- A **RmpThruRecordSet** for forwarding.
 - An set of GADs for *all* **INPUTs** of the **SERVICE**.
 - A set of data (**RmpFlatDataObj**) for the **INPUTs**.
 - The GAD (**RmpServiceGAD**) of the **SERVICE**.
 - The GAD (**RmpServiceGAD**) of the parent of the **SERVICE** (this is needed to locate the instance data).
 - The number of **OUTPUTs**, the number of data items for all **OUTPUTs** and the maximum **STRING** size of all **OUTPUTs** (to size the response).

The service response (class **RmpServiceResponseObj**) contains :-

- The checksum
- The operation status (**RmpOperationStatus**).
- The status of the **SERVICE** (**RmpServiceStatus**). Which is one of :-
 - **RmpServiceOk** - **SERVICE** completed
 - **RmpServicePending** - **SERVICE** requested accepted (rendezvous)
 - **RmpServiceRejected** - **SERVICE** request already queued (rendezvous)
 - **RmpServiceError** - **SERVICE** request invalid.
- The data for the **OUTPUTs** (**RmpFlatDataObj**)

9.2.6 Service User

The service user message is designed to “keep alive” a **SERVICE** request that has been queued.

The service user request (class **RmpServiceUserRequestObj**) contains :-

- The checksum returned from a previous read template request.
- The “resolution” (**RmpTemplResolution**) of the **SERVICE**.
- A **RmpThruRecordSet** for forwarding.
- The GAD (**RmpServiceGAD**) of the **SERVICE**.
- The GAD (**RmpServiceGAD**) of the parent of the **SERVICE** (this is needed to locate the instance data).

The service user response (class **RmpServiceUserResponseObj**) contains :-

- The checksum
- The operation status (**RmpOperationStatus**).
- The CMS address of the current user of the service (if there is one).
- The request id (**RmpReqId**) of the user.

9.2.7 Read Description

The read description message is designed to provide “full” database information about a number of items. This provides more information than that provided by a read template message.

The read description request (class **RmpReadDescriptionRequestObj**) is constructed from a set of GADs (class **RmpGADSetObj**).

The read description response (class **RmpReadDescriptionResponseObj**) contains a set of descriptions (**RmpDescription**, set **RmpDescriptionSet**). Each description contains :-

- A value descriptor (**ValDesc_S**).
- The instance name (truncated to a maximum of 12 characters).

- The type name (truncated to a maximum of 12 characters), for complex types only.
- The number of children (always 0 for simple types).

9.2.8 Debug Command

The debug command message is an “unconfirmed request” type for issuing debug messages (form the Resource debugger) to a **RESOURCE** as part of debug session.

The debug command message (structure **RemDbgCmdRequest**) simply contains a union of all debug command and their arguments (structure **dbg_cmd**). The union consists of 2 types of structures, those that are RMP debug commands and those that are internal debugger commands. The following table identifies the commands and which type they are :-

Command	Type
List	RMP and Internal
Continue	RMP
Help	Internal
Quit	Internal
Trace	RMP
Break	RMP
Delete	RMP
Print	RMP
Define	Internal
Set	RMP
WhatIs	Internal
Transcribe	Internal
RedirectIn/Out	Internal
AppendOut	Internal
Ping	RMP
Undefine	Internal
Connect	RMP
Disconnect	RMP
Ref	Internal
Unref	Internal
Read	Internal
Write	Internal
Service	Internal
Store	Internal
Exercise	Internal
Pause	Internal
Wait	Internal
Step	RMP
Disp	RMP
Next	RMP
Scan	Internal
Router	Internal
No	Internal
DeleteTrace	RMP
DeleteBreak	Internal
Expand	Internal

9.2.9 Debug Output

The command output message is an “unconfirmed request” type for reporting results of debug commands that have previously been issued. A single debug command may result in any number of command output messages at any time.

The debug output message (class **RemDbgOutputObj**) contains the following :-

- A count of the number of previous debug messages lost (since the last successful debug output message sent). This is given as the embedded debug may lose messages (due to a lack of CMS buffers) when substantial output is required (e.g. during trace).
- A debug message number, this is an index into a format string on the debugger to format the output.
- A set of data, the arguments to the format string above. This data is a set of **DINT**, **LREAL** and **STRING** only.

10. Resource Message Queue

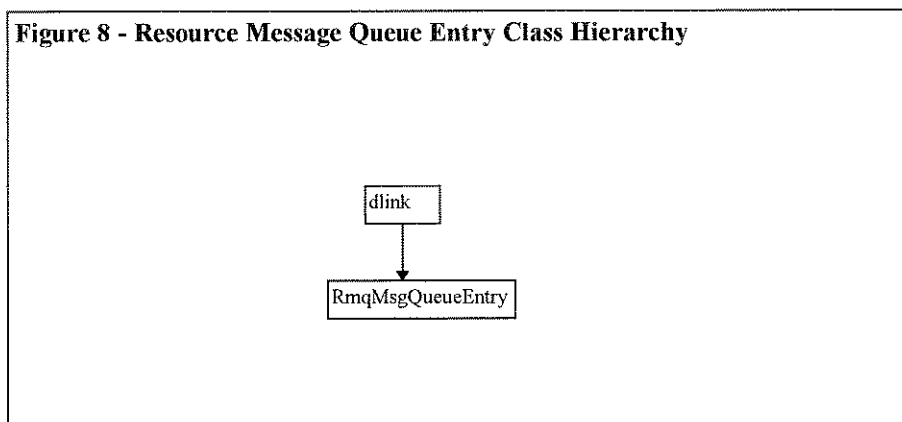
Each **TASK** (supporting **RMP**) has a message queue for **RMP** messages. The queue exists in order to de-queue messages from CMS so that they can be processed at the correct point in the **TASK** execution cycle.

The message queue is implemented by a message queue object (class **RmqObj**) with a (load time) fixed number of queue entries (class **RmqMsgQueueEntry**). The queue entries are split between 3 queues :-

- The free queue (class **RmqFreeMsgQueue**) which holds all currently unused queue entries.
- The incoming queue (class **RmqIncomingMsgQueue**) which hold all as yet unprocessed queue entries.
- The deferred queue (class **RmqDeferredMsgQueue**) which holds messages which have had their processing deferred until a later time.

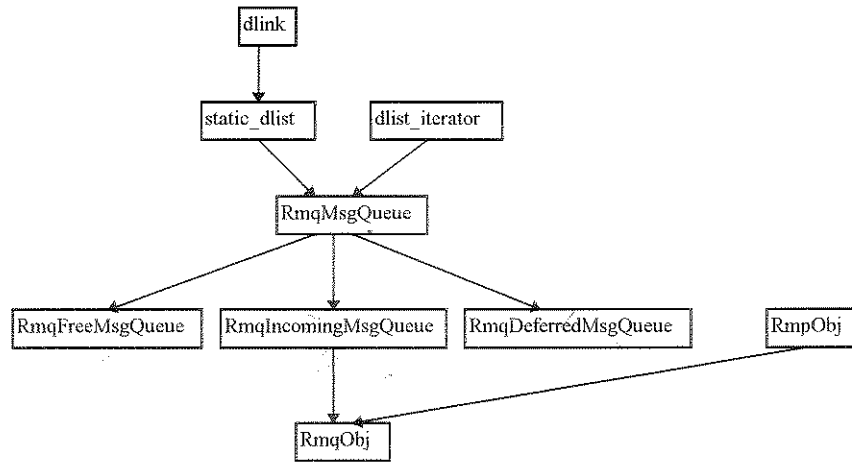
Messages are normally processed in-between each RLB execution and at the start of the execution of all RLBs. When a break point has been set using the debug features of the RMP protocol then only debug messages will be processed until the break point is cleared and the normal messages processing

Figure 8 - Resource Message Queue Entry Class Hierarchy



conditions prevail.

Figure 9 - Resource Message Queue Class Hierarchy



The **RmqObj** contains “action function pointers” for all RMP message types. These action pointers may be selectively filled according the **RmqObj** instantiators intended use of RMP. If no action function is provided the for a message type then all messages of that type will be discarded without processing.

11. Var References

The Resource Manager provides access to the database from an IEC1131-3 extension called a VAR REFERENCE.

Figure 10 - Var Reference Class Hierarchy

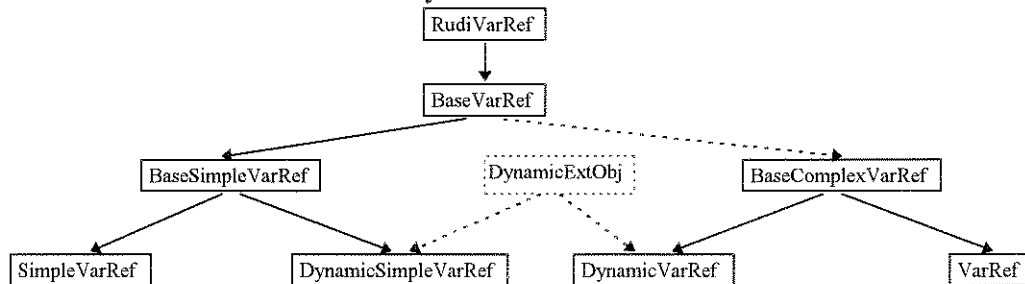
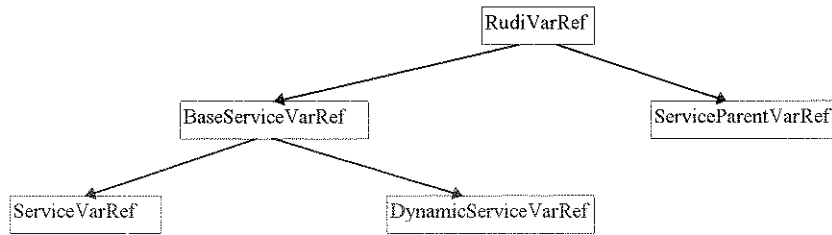


Figure 11 - Var Reference Service Classes



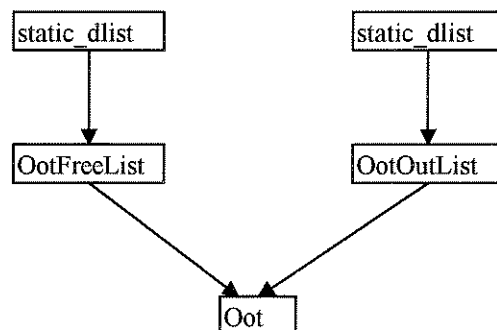
11.1 Outstanding Operation Table

The outstanding operation table (class **Oot**) is used to retain an entry (class **OotEntry**) for outstanding operation on a Var Reference. An entry is removed from a free list (class **OotFreeList**) and placed on the outstanding list (class **OotOutList**), the size of the OOT i.e. the number of entries available is fixed at load time. The entry is created on a request and remains in the table until either the response is received or else it is timed out. The one exception to this is for **SERVICE** requests which are not explicitly timed out as they may take an indefinite period of time to complete (in fact may never complete - for rendezvous the rendezvous may never be met) but may be removed on a **SERVICE** user request time-out.

The OOT has built in time-outs for 3 classes of Var Reference operation :-

- Read Template and Service User (default 60 seconds)
- Read (default 5 seconds)
- Write (default 5 seconds)

Figure 12 - OOT Class Hierarchy



The OOT is polled regularly by the **TASK** to which it is attached and if an entry times out then a time-out function is called, this is either the **Timeout** virtual method on the Var Reference class or else a bespoke time-out function (only used by the Resource Debugger) that was specified at time of entry into the OOT.

11.2 Pending Service Table

The pending **SERVICE** table (class **Pst**) comprises a number of entries (class **PstEntry**) in one of 2 lists (class **PstTable**). These lists are the free list and the outstanding list.

For each **SERVICE** that is “outstanding”, i.e. started (as a rendezvous) or else under execution (remotely) an entry is required in the PST. Each **PstEntry** contains the following :-

- The request id (**RmpReqId**) of the requester.
- The CMS address of the requester.
- The amount of data **OUTPUT** expected (this is expressed as the number of GADs, number of data items, and string size of all items) - this is required to size the response message.
- The GAD of the **SERVICE** itself.

Entries in the PST are identified by an ID which is a mask of offset into the array of **PstEntry** and the instance of that entry. The mask being dynamically sized according to the size of the PST to gain the maximum uniqueness of the ID. The user of the **SERVICE** also plants a user key **PstKey** to assist in identifying their own PST entry (this is the **ChildTemplate** pointer for remote access), this value is held in the **pending** hidden **DINT** of the **SERVICE** and should not be ≤ 0 for remote access.

12. Embedded Debug

The embedded debug facilities are available using RMP from the resource debugger⁴. These are all implemented using 2 RMP messages (the command set request the facility, and output to generate the result(s) - zero or more outputs may result from a single command).

All features of the embedded debug are available over RMP as services of a single RMP message type (the debug command).

12.1 Debug Output

Debug output (the RMP debug output message) type is effected by doing “printf” from a debug object (class **DbgIO**). Each **TASK** has a “super” debug class (class **SDBGIO**) derived from **DbgIO** which is attached to the **RLTaskObj** and located by the global pointer **DebugIO** (there is also a “global” **SDebugIO** which is in fact the same as **DebugIO**).

The **SDBGIO** class only permits a debug “session” with a single remote debugger.

12.2 Debug Session

In order to use the debug services with a **TASK** a debug “session” must be established, this requires the connect service to be issued. This will only be accepted (by the **TASK**) if no other debug session is currently in progress. The debug session is ended by terminating with a disconnect service. The disconnect terminates the session even it is issued by someone other than that conducting the debug session.

⁴ It is hoped that GCT will also provide some of these facilities in the future.

12.3 Ping

This is a simple service which echoes back the **TASK** to which the debugger is connected.

This is the only service that does not require a debug session.

12.4 Trace and Break

This section describes those objects to which trace and break can be applied.

12.4.1 Resource Manager Class Debug

The facility to debug the primary Resource Manager classes (by setting break and trace points) is provided. The following classes are debuggable in this manner :-

- Refer (class **Refer**)
- Var Reference (class **RudiVarRef**)
- GAD (class **GAD**)
- Template (class **Template**)
- GAD Table (class **GADTableEntry**)
- RLO (class **RLObj**)

Each of the above classes is derived from a base debug class (**idebug**).

It should be noted that not all methods of the above classes are debuggable (i.e. break/trace points can not necessarily be set on entry/exit to these methods - this is so that maximum use of in-line methods can be made.)

It is not intended that any product will have this feature enabled in shippable code due to the run-time over-head.

12.4.2 RMP Messages

It is also possible to trace or break on an RMP debug message type being received (this is implemented in the Resource message queue (**RmqObj**). This facility should be used with caution.

It is not intended that any product will have this feature enabled in shippable code due to the run-time over-head.

12.4.3 Source Code Debug

It is possible to trace or break **FUNCTION_BLOCKS** (and **PROGRAMs**) at entry, exit or specified line numbers. Blocks that are to be debugged must have the **-DEBUG** switch applied to the ST compiler which then plants calls for each trace/breakable point in the block. Blocks with debug on will run slower than without (i.e. debug is intrusive).

12.5 Resource Database Interrogation

Database interrogation (and modification) can be effected using the set and print services of the debug command message type. These are however very crude and reduce all items to one of 3 types :-

-
- **DINT**
 - **STRING**
 - **LREAL**

The set service is completely unprotected and over-rides the normal protection features of the database.

This feature is only intended as debug facility for system programmers and not for customer use.

13. ST Compiler Interfaces

This section lists all calls planted by the ST compiler into the Resource manager code.

13.1 Var References

The bulk of the ST compiler plants are for **VAR REFERENCES**.

In all cases these function begin with the same 2 arguments which identify a pointer to the **VAR REFERENCE** class. These are :- **Refp**, pointer to the head of the instance data which contains a pointer to an array of **VAR REFERENCE** pointers for the block, **offset** the offset into this array for the **VAR REFERENCE** to which it applies. A full listing of all functions is given below, where the ellipsis ... refers to the implicit arguments **Refp** and **offset**.

```
void refVREFMETHOD(..., CDL_string* ref )
CDL_string* getRefVREFMETHOD(...)
void scanVREFMETHOD(..., CDL_dint scanrate)
CDL_time getScanVREFMETHOD(...)
int statusVREFMETHOD(...)
int readStatusVREFMETHOD(...)
int writeStatusVREFMETHOD(...)
int servStatusVREFMETHOD(...)
int writeVREFMETHOD(...)
void setDontWriteVREFMETHOD(..., bool dontwrite)
CDL_bool getDontWriteVREFMETHOD(...)
int serviceVREFMETHOD(..., char* name)
void setPropertyProtectVREFMETHOD(..., bool Protect)
CDL_bool getPropertyProtectVREFMETHOD(...)
CDL_bool* getWriteMarkersVREFMETHOD(...)
int getResolutionVREFMETHOD(...)
CDL_date_and_time getTimeStampVREFMETHOD(...)
CDL_qtime getQTimeStampVREFMETHOD(...)
```

13.2 Source Code Debug

In order to provide source code debug the ST compiler plants calls to the embedded debug software with the function :-

```
TraceFblock ( char* InstanceData, int lineNumber, int entry, char* fname ) ;
```

where :-

- InstanceData - Pointer to the instance data for the block
- lineNumber - Current line number
- entry - one of :-
 - 0 - block entry
 - 1 - block exit
 - 2 - line number
- fname - **FUNCTION_BLOCK** name

13.3 SERVICES

A single call is planted by the ST compiler when a **SERVICE** completes.

SERVICECOMPLETE (int OstEntry)

The OstEntry is in fact the value of the **pending OUTPUT** (see 11.2) of the **SERVICE**.

14. The Source Code

14.1 Where to Find It

The following table identifies what is defined in what source module

Source File(s)	Classes, Types and Globals
apl_gads.[ch]xx	AplGADEI, SetOfAplGAD, GAD, AplInstGAD
cif_cxx.cxx cif_rsrc.h	<i>ST Compiler Interfaces for SERVICECOMPLETE and TraceFblock</i>
cif_rdb.h, .cxx	<i>Various C Resource database interfaces for function blocks</i>
cif_tfl.h, .cxx	<i>C Interfaces to TASK function lists</i>
cif_vref.h, .cxx	<i>ST Compiler VAR REFERENCE Interfaces</i>
dbe_cmnd.h	dbg_cmnd
dbe_ebed.cxx	<i>Implementation of debug commands</i>
dbe_io.[ch]xx	DebugIO, SDebugIO, SDbgIO
dbe_outp.cxx	
dbe_rem.[ch]xx	RemDbgCmndRequest, RemDbgOutputRequest, RemDbgOutputObj
dbe_util.[ch]xx	
def_defs.h	VarMode, VarShape, VarType
def_diom.h	<i>Format strings for debug output</i>
def_fbt.h	CDL_Any_P, CDL_Any_S, CDL_struct_8, CDL_struct_16, CDL_struct_32
def_idbg.hxx	idebug, DbgIO
def_vals.h	FastGad_S, ValDesc_S, Value_P, Value_S
mai_load.cxx	<i>The Resource loader main</i>
mai_task.cxx	<i>The Resource (IEC) task main</i>
mai_ldtk.cxx	<i>A combined main for the loader and the task</i>
rdb_acc.[ch]xx	RdbAccessNode, RdbBaseAccessPath, RdbSetOfAccessNode
rdb_bnds.[ch]xx	Bounds, Dimension
rdb_btisk.[ch]xx	BaseTaskObj, MyBaseTaskObj
rdb_cver.hxx	<i>This defines the current version number of the Resource Manager</i>
rdb_gadt.[ch]xx	GADTableEntry
rdb_navi.[ch]xx	ResourceNavigObj

rdb_rlo.[ch]xx	RLObj, RLBObj
rdb_rsrc.[ch]xx	ResourceType_base, ResourceType
rdb_task.[ch]xx	BreakPointClass, ClassDebug, MyBaseTaskObj, RLTaskObj, TaskDataStateObj, TaskExecuteStateObj, TaskLoadStateObj, Trace
rdb_tmpl.[ch]xx	ChildTemplate, ComplexBase, ComplexChild, ComplexChildWithBounds, ComplexType, SimpleChild, SimpleChildWithBounds, SimpleType, SimpleTypeWithBounds, Template, TypeTemplate
rdb_vacc.[ch]xx	RdbAccessPath, RdbSetOfAccessPath
rdb_who.[ch]xx	<i>Functions for database location by instance data pointer</i>
rem_pst.[ch]xx	Pst, PstEntry, PstKey, PsTable
rem_rdsc.cxx	<i>Action function for RMP read description requests</i>
rem_read.cxx	<i>Action function for RMP simple read requests</i>
rem_serv.cxx	<i>Action functions for RMP server and service user requests</i>
rem_tmpl.cxx	<i>Action function for read template request</i>
rem_wr.cxx	<i>Action function for write read requests</i>
rem_writ.cxx	<i>Action function for simple write requests</i>
ref_ref.[ch]xx, ref_writ.cxx	Refer
ref_set.[ch]xx	SetOfRefer
ref_stck.[ch]xx	
ref_str.[ch]xx	
rld_atc.[ch]xx	TheResource
rld_fost.[ch]xx	RldFileOfST
rld_ldsh.[ch]xx	RldObj
rmp_cms.hxx	
rmp_cod.[ch]xx	<i>Encode and decode of RMP messages.</i>
rmp_defs.hxx	RmpMsgType, RmpMsgVersion, RmpOperationStatus, RmpReqId, RmpResourceChecksum, RmpServiceStatus, RmpStatus, RmpTemplResolution,
rmp_obj.[ch]xx	RmpFastGadSetObj, RmpFlatDataObj, RmpGADSetObj, RmpStatusSetObj, RmpTimeStamp, RmpThruRecordSetObj, RmpValDescSetObj
rmp_rmp.[ch]xx	RmpObj
rmp_rdsc.[ch]xx	RmpDescription, RmpDescriptionSet, RmpReadDescriptionRequestObj, RmpReadDescriptionResponseObj
rmp_read.[ch]xx	RmpSimpleReadRequestObj, RmpSimpleReadResponseObj
rmp_serv.[ch]xx	RmpServiceGAD, RmpServiceRequestObj, RmpServiceResponseObj, RmpServiceUserRequestObj, RmpServiceResponseObj
rmp_size.[ch]xx	<i>Functions for determining the size of RMP messages</i>
rmp_tmpl.[ch]xx	RmpReadTemplRequestObj, RmpReadTemplResponseObj
rmp_ver.[ch]xx	RmpCurrentVersion
rmp_wr.[ch]xx	RmpWriteReadRequestObj, RmpWriteReadResponseObj
rmp_writ.[ch]xx	RmpSimpleWriteRequestObj, RmpSimpleWriteResponseObj
rmq_msgq.[ch]xx	RmqDeferredMsgQueue, RmqFreeMsgQueue, RmqIncomingMsgQueue, RmqObj
rut_dlnk.[ch]xx	static_dlist, dlink, dlist_iterator
rut_tfl.[ch]	tfl_entry
rut_time.[ch]	
rut_vals.[ch]xx	
rut_vdsc.[ch]xx	SetOf, SetOfVals, SetOfValDescs, SetOfValsIterator
sio_io.hxx	
vrf_base.[ch]xx	BaseComplexVarRef, BaseServiceVarRef, BaseSimpleVarRef, BaseVarRef, RudiVarRef, ServiceParentVarRef
vrf_diag.h	
vrf_diag.hxx	
vrf_dyn.[ch]xx	DynamicExtObj, DynamicSimpleVarRef, DynamicServiceVarRef, DynamicVarRef
vrf_new.cxx	
vrf_oof.[ch]xx	Oot, OotEntry, OotFreeList, OotOutList

vrf_pars.[ch]xx	
vrf_rem.cxx	Action functions for RMP responses for Var Reference generated requests
vrf_vref.[ch]xx	SimpleVarRef, ServiceVarRef, VarRef
vt_vtbl.cxx	This contains headers to generate a single virtual function table some targets

14.2 Order of compilation

The order of compilation of the Resource manager modules must be :-

1. rut
2. sio
3. apl
4. rmp
5. rmq
6. rdb (rdb, ref, rem, vrf)⁵
7. rmi
8. dbe
9. ras
10. rld
11. cif
12. vt⁶
13. mains (loader, task)

The above ordering assumes that all modules and features of the Resource Manager are to be used.

14.3 Pre-processor

This section identifies those C/C++ pre-processor symbols that affect compilation of the Resource Manager.

14.3.1 Target

This section identifies those symbols that are dependent on the Resource Manager target.

Symbol	Relevant Targets	Description
SHARELIB	unix	Is the Resource Manager to be built as a shared library
FloatIEEE	all	Are floating point numbers in IEEE format
NativeIsUni	all	Is byte ordering the same as the universal format (Motorola)

14.3.2 Options

This section identifies those symbols that enable or disable features of the Resource Manager.

Symbol	Description
ATTRIBUTE	Enables draft attribute implementation. This currently achieves nothing useful.
debug_mode	Defines what debug facilities are available

⁵ There are cyclic dependencies between these modules and so must be compiled as a set with all their header previously exported

⁶ Separately compiled virtual function tables are only required for some targets.

NO_TEXT	Loader trace facilities disabled
TargetSupport_CD_LREAL	Defines if IEC1131-3 LREALs are supported. No current targets support LREALs, although it is required for GCT build.
TargetSupport_DynamicVarRef	Are dynamic var references supported.
TargetSupport_SERVICE	Are SERVICES supported
TargetSupport_VAR_ACCESS	Is VAR_ACCESS supported
TargetSupport_VAR_REFERENCE	Are VAR REFERENCES supported

15. Glossary of Terms

CMS - Communications Messaging Services

GAD - Generic Address Descriptor

OOT - Outstanding Operation Table

PST - Pending Service Table

RLB - Resource Level Block

RLO - Resource Level Object

RMP - Resource Messaging Protocol

SERVICE - A Eurotherm extension to the IEC1131-3 standard to provide remote procedure calls.

ST - Structured Text, used to refer to the output of the compiler in any of the IEC languages

VAR REFERENCE - A Eurotherm extension to the IEC1131-3 standard to provide access to remote data.

Index

A

ACCEPT, 14
AplGADEI, 17
AplInstGAD, 18

B

BaseTaskObj, 21
Bounds, 11, 37
break, 30, 35
BreakPointClass, 24

C

CDL_Any_P, 14, 15
CDL_Any_S, 14, 15
CDL_struct_32, 13
checksum, 12
ChildTemplate, 10, 34
ClassDebug, 24
CMS, 24, 25, 29, 31, 34, 41

D

dbg_cmdnd, 30
DbgIO, 34
debug, 24, 30, 31, 34, 35, 36
debugging, 24, 30, 31, 34, 35, 36
DebugIO, 34
Dimension, 11, 37
dlink, 16
dlist_iterator, 16

F

FastGad_S, 15
FUNCTION_BLOCK, 9, 10, 12, 16, 23, 35, 37

G

GAD, 8, 9, 12, 13, 15, 16, 17, 18, 19, 20, 26, 27, 28, 29, 34, 35, 37, 41
GADTableEntry, 13, 35
GCT, 9, 13, 19, 20, 34, 40

I

idebug, 35

O

Oot, 33
OotEntry, 33
OotFreeList, 33
OotOutList, 33
Outstanding Operation Table, 41

P

PROGRAM, 16
PST, 9, 34, 41
PsTable, 34
PstEntry, 34
PstKey, 34

R

RdbAccessNode, 12
RdbAccessPath, 12
RdbBaseAccessPath, 12
RdbSetOfAccessPath, 12
Refer, 12, 19, 35, 38
ReferStack, 19
RemDbgCmdndRequest, 30
RemDbgOutputObj, 31
Rendezvous, 14
ResNavigObj, 19
RESOURCE, 9, 10, 12, 16, 17, 18, 20, 21, 23, 25, 26, 30
Resource Level Blocks, 12, 18, 31, 41
Resource Level Objects, 9, 12, 13, 15, 16, 17, 18, 41
ResourceNavigObj, 19
RETAIN, 14
RLB, 12, 41
RldObj, 20
RLO, 12, 41
RLTaskObj, 21, 24, 34
RMP, 12, 14, 16, 21, 23, 24, 25, 26, 28, 30, 31, 32, 34, 35, 38, 39, 41
RmpCurrentVersion, 26
RmpDescriptionSet, 29
RmpFlatDataObj, 25, 27, 28, 29
RmpGADSet, 25
RmpGADSetObj, 25, 27, 29
RmpMsgType, 26
RmpMsgVersion, 26
RmpObj, 25
RmpOperationStatus, 26, 29
RmpReadDescriptionRequestObj, 29
RmpReadDescriptionResponseObj, 29
RmpReadTemplRequestObj, 26
RmpReadTemplResponseObj, 26
RmpReqId, 26, 29, 34
RmpResourceChecksum, 25

RmpServiceGAD, 29
 RmpServiceRequestObj, 28
 RmpServiceResponseObj, 29
 RmpServiceStatus, 29
 RmpServiceUserRequestObj, 29
 RmpServiceUserResponseObj, 29
 RmpSimpleReadRequestObj, 26
 RmpSimpleReadResponseObj, 27
 RmpSimpleWriteRequestObj, 27
 RmpSimpleWriteResponseObj, 27
 RmpStatus, 25
 RmpStatusSetObj, 28
 RmpTemplResolution, 27, 28, 29
 RmpThruRecordSetObj, 25
 RmpTimeStamp, 25
 RmpWriteReadRequestObj, 27
 RmpWriteReadResponseObj, 28
 RmqDeferredMsgQueue, 31
 RmqFreeMsgQueue, 31
 RmqIncomingMsgQueue, 31
 RmqMsgQueueEntry, 31
 RmqObj, 25, 31, 32, 35
 RudiVarRef, 35

S

SDBGIO, 34
 SDebugIO, 34
 SERVICE, 14, 28, 29, 34, 37, 40, 41
 SetOf, 16
 SetOfAplGAD, 17, 27
 SetOfRefer, 19
 SetOfValDescs, 16, 26
 SetOfVals, 16
 SetOfValsIterator, 16
 SFC, 13
 ST compiler, 13, 36, 37
 static_dlist, 16

STRING, 15, 29, 31, 36
 STRINGs, 15, 29, 31, 36

T

TASK, 12, 19, 20, 21, 22, 23, 24, 26, 31, 34, 35, 37
 TaskDataStateObj, 23
 TaskExecuteStateObj, 23
 TaskLoadStateObj, 22
 Template, 6, 11, 20, 33, 35
 tfl_entry, 24
 trace, 24, 30, 35, 38
 TypeTemplate, 10

U

unix, 20, 39

V

ValDesc_S, 15, 17, 29
 Value_P, 15, 16
 Value_S, 15
 VAR_REFERENCE, 9, 13, 14, 20, 23, 24, 25, 26, 32, 36, 37, 40, 41
 VAR_ACCESS, 9, 11, 12, 20, 40
 VAR_EXTERNAL, 14
 VAR_GLOBAL, 12, 14
 VarMode, 14
 VarRef, 19
 VarShape, 15
 VarType, 14, 15