# Preface

By Professor Brian Warboys, University of Manchester

It is now some 20 years since the foundations of the VME architecture were laid. The original objectives were to produce an architecture which would be flexible enough to allow for the many changes that were inevitable in the market, in the enabling technologies and in the end-user demands for IT systems. At the same time it was imperative that the architecture should be constraining enough to control the very long development process that long life implies.

The result was essentially to define a simple but powerful approach which has basically remained unchanged during its long life. This is not a tribute to the remarkable foresight of the many people who contributed to that design but rather confirmation of the long known but often forgotten maxim that the best solution to mastering complexity is by the formulation of simple rules.

The basic rules, which are enshrined in the architecture, have not had to be reformulated but merely extended in order to produce an architecture which meets the modern demands of Open Client-server systems. The essential architectural simplicity which guided the development of a complex Operating System throughout the 70's and 80's is as applicable in the 90's. Moreover this approach has been thoroughly tested over the last twenty years and there should be no doubt in anybody's mind about the capability of the OpenVME architecture to manage the complex design issues which confront modern Client-server systems.

A splendid side effect of this text is that, for the first time, it enumerates, in a readily digestible form, the thinking behind the VME architecture. This has been long overdue and has been for a very long time an unsatisfied objective of mine. I should like to congratulate the author on relieving me of this burden. His text is far superior to any that I would have produced and I am sure that readers will fully appreciate his efforts.

## About the Author

Nic Holt is a member of the Systems Architecture group of ICL Corporate Systems and is responsible for the technical strategy for OpenVME. He joined ICL in 1972 having gained an honours degree in mathematics at Cambridge. His subsequent experience with ICL has encompassed system and software design, distributed processing architectures and formal design methodologies. In 1993 he became an ICL Distinguished Engineer.

He has worked with several external organisations on collaborative research projects and is a visiting Professor at Glasgow University. He is a Member of the BCS and on the editorial panel for the BCS Practitioner Series.


## Acknowledgements

Very many people have contributed to the continued evolution of VME since its inception. This text is founded upon their work.

Particular thanks are due to Brian Warboys, Roger Poole, Eileen Vaughan, Paul Coates, members of Systems Architecture and OPENframework, and other colleagues and friends for their help and encouragement.


## Product Information

Where information about specific products is given in this document, this is intended solely to illustrate the technical principles described in the architecture, and not to assist readers in evaluating or using the products concerned. Products undergo continuous development and published details can quickly become out of date. No liability can be accepted for error or omission. For definitive information, the reader should consult the supplier of the product in question.



# Contents

# Chapter 1
# Aims of this document

## Document scope

The scope of this document is the architecture of ICL's OpenVME system. It describes the architecture in detail; the benefits that OpenVME brings to corporate systems, Client-server and distributed systems; and how OpenVME is able evolve to match the trends in open corporate systems. The architecture is a specialisation of the OPEN*framework* systems architecture.

The OpenVME System combines the technical and architectural strengths essential for Corporate Systems with the Open Systems interfaces and interworking capabilities required to support the current and future standards for portable applications and distributed computing. OpenVME thus offers excellent support for open applications within a Client-server or co-operative (distributed) processing environment, especially where considerations such as scale, manageability, security and reliability are critical.

## Purpose of the Architecture

This architecture provides a framework for understanding the fundamental design principles, the key attributes, the overall structure and the relationships between the major components of OpenVME. The architecture can thus be used to guide the development and exploitation of the features of OpenVME which enable it to provide exceptional support for applications and services within a corporate system.

## Readers

This document is intended for everyone who wishes to understand the architecture of ICL's OpenVME system. The OPEN*framework* perspectives from which the OpenVME architecture is described are primarily those of the Service Provider, the Application Developer and, to a lesser extent, the User. Readers are expected to be technical consultants and technicians.

## Document Structure

| | |
|---|---|
| Chapter 2 | outlines the aims and principles of the OpenVME architecture and discusses them in terms of the OPEN*framework* Qualities and Perspectives. |
| Chapter 3 | describes the OpenVME System Architecture, providing an overview of OpenVME functionality and the open interfaces and services offered by each element. |
| Chapter 4 | introduces the fundamental architectural concepts and principles on which the OpenVME architecture is based. |
| Chapter 5 | identifies the structuring principles of the OpenVME architecture and uses them to provide a description of the major components of the core OpenVME system. |
| Chapters 6-14 | describe the architecture of the major functional elements of the OpenVME system in detail, showing how they build upon and exploit the features of the fundamental architecture. The chapters correspond closely to the OPEN*framework* elements. |
| Chapter 15 | analyses the OpenVME architecture in terms of the OPEN*framework* qualities, concentrating on those with particular relevance for corporate systems. |
| Appendices | include a glossary, a bibliography, a list of relevant standards and other OpenVME reference material. |

# Chapter 2
# Aims of the Architecture

**Form**: style and arrangement; structural unity [Chambers English Dictionary]

**Architecture**: structure; the overall design [Chambers English Dictionary]

## Overall aim

The overall aim of the architecture is to describe the technical concepts and principles upon which the design of OpenVME is based. The OpenVME system provides an environment within which open applications may be easily developed or ported, executed and managed, and interwork with other open applications. Particular emphasis is placed upon the ability to support services on a corporate scale within Client-server systems. In this context, the term "Client-server" embraces both hierarchic (workstation/server) and peer-to-peer (co-operative processing) dimensions of distributed computing.

The architecture provides a framework for analysing and designing Client-server systems in which OpenVME has a role providing an environment for corporate server applications, and for identifying the OpenVME strengths and how they can best be exploited in a particular context. A subsidiary aim is to identify those features and capabilities of OpenVME which are of differential benefit compared with alternative environments.

## Major Themes

The major themes of the OpenVME architecture are:

- Open Application Environments & Services
- Open Client-server & Distributed Computing
- Support for 'Corporate' qualities and scale

# Principles

## Openness

The architecture is an *open* architecture:

- The architecture provides for application portability by provision of Application Programming Interfaces conforming to open standards. Applications written to such standards can then be readily ported to OpenVME; equally, this approach enables applications developed for OpenVME to be readily ported to other open environments.
- The architecture provides for an OpenVME system supporting server applications within a distributed or Client-server computing environment, using open interworking standards across heterogeneous platforms.
- The architecture provides application development tool interworking to open standards. This is particularly crucial for distributed systems whose component applications may be developed and/or executed in heterogeneous environments.

## Conformance to Standards

The ICL OPEN*framework* architecture is used as a basis for the structure of this document and OPEN*framework* terminology is used where applicable.

There are many and diverse standards (international, industry, de facto) that the OpenVME system supports. As a general statement, conformance to relevant X/Open standards is supported where appropriate; these X/Open standards may evolve with time (*e.g.* XPG3, XPG4, etc.). The primary networking and interworking standards are those defined by ISO as part of the Open Systems Interconnection architecture, together with the Internet set of protocols. The OpenVME architecture allows support of all networking standards, by use of gateways where appropriate.

## Qualities

### *General*

OpenVME provides the high levels of integrity, availability, performance, security, ease of use, management and adaptability which are required for supporting mission-critical corporate applications and services.

### *Potential for Change*

IT technology is continuously evolving at an ever increasing rate. New hardware, operating systems, applications, services and networking facilities may offer major opportunities for more effective use of IT provided that they can be readily incorporated within the system. At the same time, business methods are also evolving, necessarily adapting to external competition and pressures as well as increased use and integration of IT into business processes. OpenVME is therefore designed actively to support the interception of these evolutionary trends.

There is considerable user investment in existing IT systems and the information they contain. This is particularly true for large organisations in which the use of IT has become widespread. OpenVME makes it possible to exploit new IT capabilities and opportunities without significantly diminishing the value or use of existing IT assets created by this historical investment.

## Perspectives

### *The Service Providers' Perspective*

Numerous business and technology trends and influences have resulted in a massive and continuing growth in the use of various distributed computing techniques. OpenVME provides excellent support for the interfaces and interworking capabilities, and the system management facilities necessary to support the provision of large-scale services within corporate distributed computing environments.

### *The Application Developers' Perspective*

The support of open standards greatly simplifies the task of the application developer and allows the use of standard application development tools. The OpenVME application development facilities can be used, optionally in conjunction with standard CASE tools, to generate open applications for a wide range of heterogeneous, distributed computing environments.

### *The Users' Perspective*

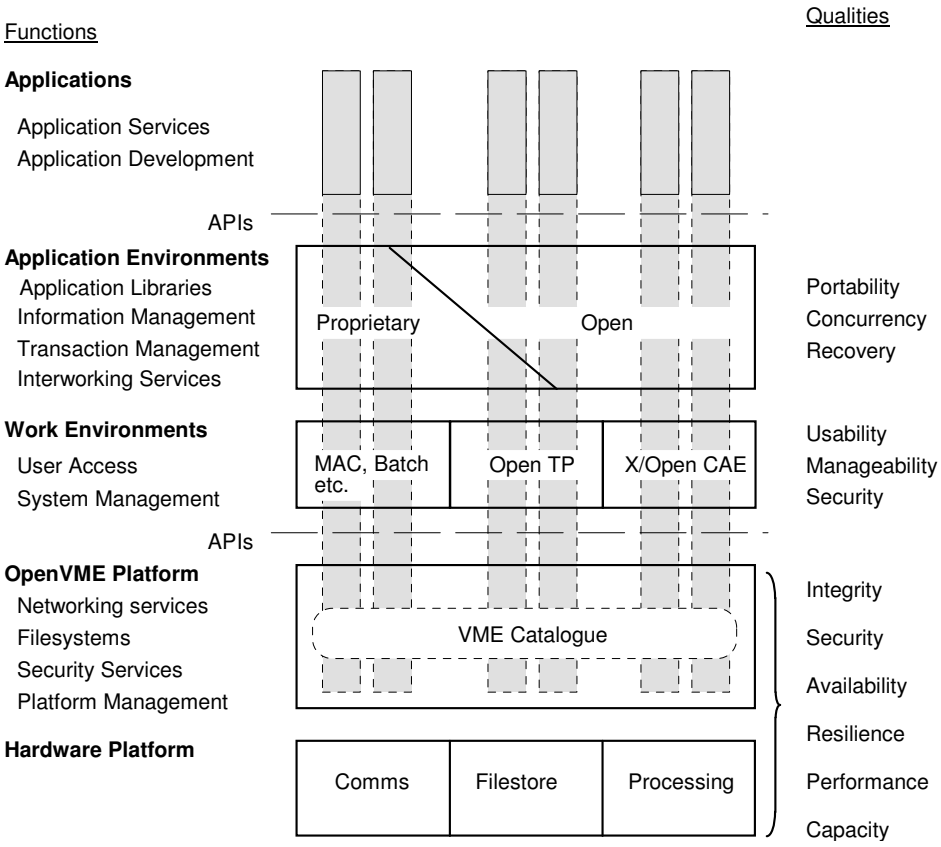
The OpenVME architecture is designed to provide support for large-scale server applications within a Client-server environment. It is expected that users will increasingly exploit the usability and functionality provided by personal workstations. Applications executed on workstations can act as clients for common services executed on one or more OpenVME systems. The users thereby gain the benefits of advanced user interfaces and personal applications whilst their roles within the organisation are supported and enhanced by shared services and information.

# Chapter 3
# The OpenVME System Architecture

## Introduction

The functionality of OpenVME is grouped into several major areas:

- Application Environments
- Transaction Management
- Information Management
- Interworking Services
- Networking Services
- Application Development
- User Access
- Systems Management



These areas are closely aligned with the elements of the OPEN*framework* Systems Architecture. This chapter provides an overview of the elements of the OpenVME System Architecture, identifying the open Application Programming Interfaces and services provided by each element.

## Application Environments

OpenVME provides an environment for the support of multiple concurrent *Application Service*s. An Application Service is implemented as the co-operative interaction of one or more applications, each executed in the context of its own Virtual Machine. Each Virtual Machine can be dynamically configured to provide a tailored application environment with appropriate underlying services and Application Programming Interfaces.

The Application Programming Interfaces used to access OpenVME system services conform to open standards where applicable - notably those defined in the X/Open Portability Guides (XPG). Interfaces to new services are designed to intercept emerging standards, providing a simple route to early adoption of open standards. Use of open application interfaces ensures that software can be readily ported to run on OpenVME, and also that software developed on the OpenVME system can be ported to any other platform supporting open application interfaces.

## Transaction Management

Transaction management is a major contributor to the support of reliable Client-server computing. Distributed transaction management is an integral feature of OpenVME. This is exploited by TPMSX and the Information Management services to provide an open application environment for distributed TP applications.

## Information Management

OpenVME supports Relational databases, a high performance Codasyl database (IDMSX) and various file organisations (including sequential and indexed).

## Interworking Services

OpenVME Interworking Services build upon lower level networking Services to establish a framework for linking separate applications together to form coherent large-scale systems in distributed, heterogeneous environments. A key aim of the OpenVME Distributed Application Services is to hide the differences between underlying interfaces from the application and the application programmer. This is achieved through a combination of Application Development tools and run-time services.

## Networking Services

OpenVME Networking Services provide a comprehensive set of communications capabilities to support Client-server and co-operative processing. This is based on a fully integrated OpenVME implementation of the International Standards Organisation OSI architecture, supplemented with the TCP/IP protocol suite commonly used on UNIX systems. Full interworking with other open systems is supported.

OpenVME security features control access to networking services and comprehensive management of the communications system is provided. Applications access is via the X/Open Transport Interface (XTI) over both OSI and TCP/IP networks.

Higher level networking is provided for Terminal Access, File Transfer, Mail and other services, where proprietary protocols are being replaced by the corresponding OSI standards - VTP, FTAM, X400; in addition, key UNIX standards are supported (*e.g.* NFS, ftp and telnet).

## Application Development

OpenVME Application Development is based around the OpenVME Data Dictionary System (DDS). This is an open central repository for corporate application information. Open interfaces to the data dictionary are provided so that it can interwork with leading Upper CASE tools. The advanced ICL QuickBuild application development system generates fully portable Client-Server applications. Language support is maintained to conform to prevailing open standards.

## User Access & User Interface

The OpenVME User Interface element is optimised to provide access to OpenVME from standard PC and intelligent workstations, using industry standard Human-Computer Interfaces - in particular Windows on PCs. PC interconnection is supported for industry standard PC LAN systems. Advanced user interfaces are available to users of these systems through the use of OpenVME Client-server networking facilities, which allow PC based applications and user interface handling facilities to be used and integrated with OpenVME based servers. Facilities are also provided to support the connection of basic terminals.

## Systems Management

OpenVME Systems Management combines the comprehensive OpenVME-specific systems management facilities with the ability to access these facilities over open systems management interfaces and protocols, allowing an OpenVME system to be managed as part of a network of open systems. Facilities are provided to allow both the system and critical services such as TPMSX services to be remotely managed in a distributed network. OpenVME based distributed management facilities include centralised operations control and fault management facilities and the archiving of distributed Unix and PC based systems.

## The Series 39 Hardware Platform

The underlying platform is provided by the OpenVME primitive architecture supported by Series 39 hardware. Extensive features are incorporated into the OpenVME Kernel and Series 39 hardware to achieve exceptional levels of security, availability, resilience and integrity.

The Series 39 Nodal Architecture enables integrated processing nodes to be combined into *multi-node* systems in which shareable resources are uniformly accessible, but unshared resources are handled efficiently within a node, thus eliminating inefficiencies often associated with more closely coupled multi-processing architectures.

# Chapter 4
# The Fundamental Architecture of OpenVME

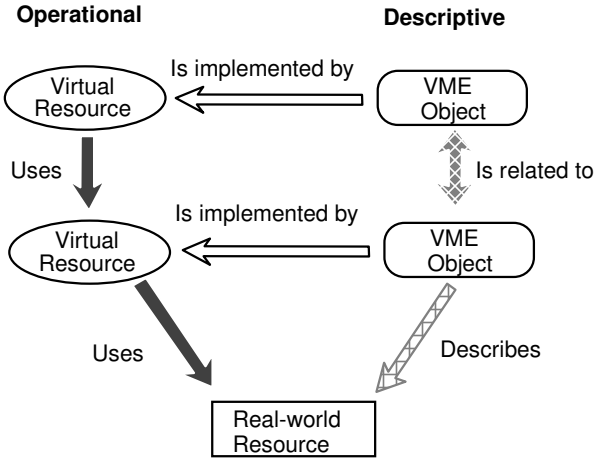
## Introduction

The OpenVME Architecture is founded upon a small number of basic concepts. These are used to describe the structure and operation of the OpenVME system at all levels. This chapter introduces those ideas and the relationships between them, establishing a fundamental set of conceptual building blocks from which a description of the whole Architecture can be constructed.

## Summary

VME is an acronym for **V**irtual **M**achine **E**nvironment. In the OpenVME architecture, computation proceeds by operations on, and interactions between *VME Objects*. A VME object may correspond to some real resource or may be entirely conceptual, possibly providing the means of accessing other objects.

Fundamental Concepts of the OpenVME Architecture




A *Virtual Resource* is a scoped, executable representation of a VME object. It comprises data and a set of associated *procedures* implementing the operations of the resource. A *Virtual Machine* (VM) is a logical container for a set of Virtual Resources. The *Virtual Machine Environment* comprises the totality of Virtual Resources within a Virtual Machine.

Object usage is considered in two phases:

- The *declarative* phase during which an object is selected and made available to the VM as a Virtual Resource;
- The *imperative* phase during which the Virtual Resource is used.

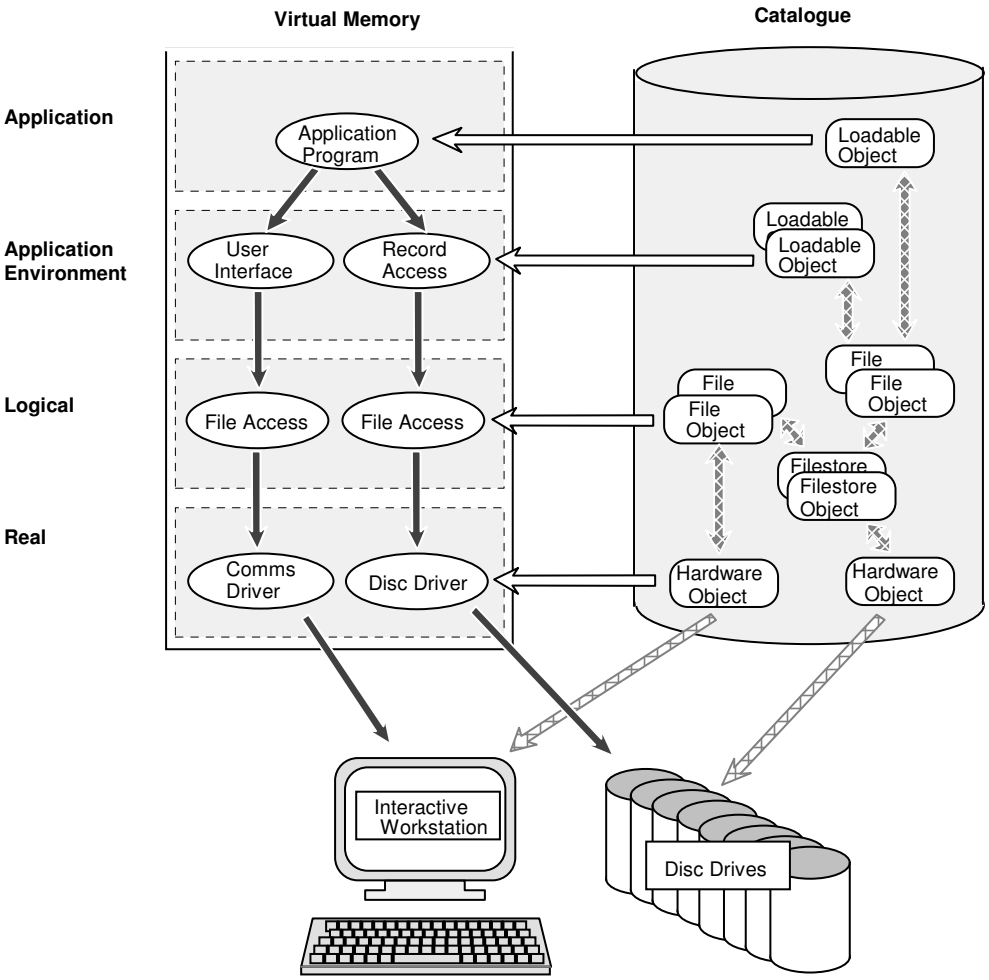
Within each VM, the system provides controlled means to acquire and use the real resources of the system. At the lowest level these include memory, the processor and external devices. At higher levels these are represented by Virtual Resources in such a way that each VM appears to have available to it exclusive use of the resources it requires even though the real resources may be shared, by the system, between several VMs. Thus the description "virtual" refers to the temporary, scoped *representation* of a VME object or resource within a VM.

The VM provides a dynamic environment in which work is performed by the execution of a *process* threading through user and system *procedures*. A stack is provided for automatic allocation and de-allocation of workspace and to preserve links between procedures. Each procedure is executed in a particular context: a *process context*, related to the dynamic state of the process; and a *static context,* associated with a specific Virtual Resource.

An operational OpenVME system supports many applications or services implemented as sets of related VMs. The architecture provides communication and synchronisation features which enable VMs to co-operate efficiently with one another. The fundamental objects which support these features are *shared memory areas* and *events* both of which may be shared between two or more VMs.

Most applications have some requirement to communicate with external devices - whether explicitly (*e.g.* communicating with another system or outputting to a terminal) or implicitly (*e.g.* accessing a data file or loading an application). The OpenVME Input/Output architecture allows each VM to initiate I/O requests directly, transferring data between the I/O device and a memory area within the VM; transfer termination is notified, via an *interrupt,* to the requesting VM.

The Fundamental Concepts of OpenVME - an Illustrated Example



# The Declarative Architecture: VME Objects

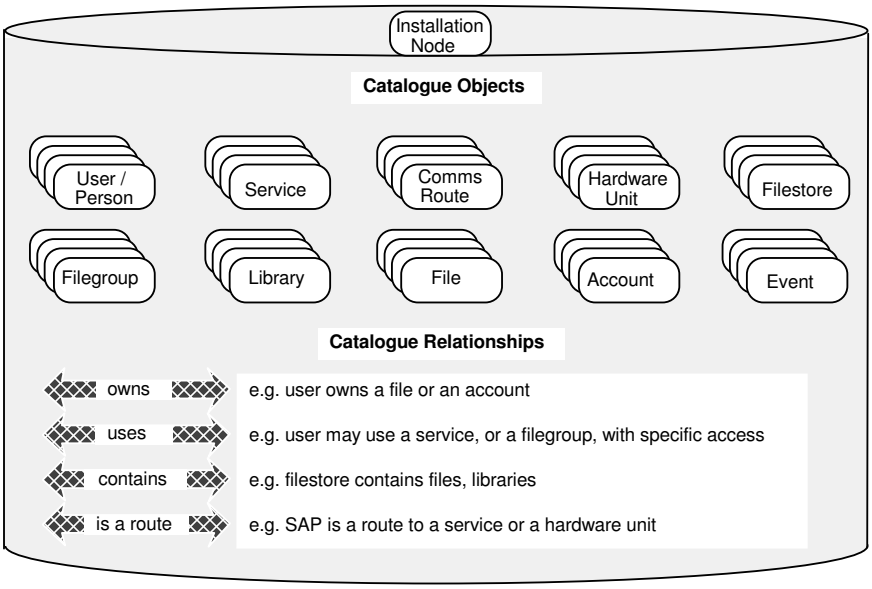
## Introduction to VME Objects

*VME Objects* play a key role in the OpenVME architecture. A VME object is an instance of some class of similar objects with defined attributes and behaviour. An object may an abstract representation of some real resource, providing a standard, controlled means of accessing that resource (*e.g.* a hardware device or physical memory); alternatively, an object may be entirely conceptual (*e.g.* a file description), possibly providing the means of accessing or using other objects. The VME Catalogue records each object known to an OpenVME system and the relationships between them.

## The Catalogue

### *Catalogued Objects*

The Catalogue is a system database which records the complete set of objects known to an OpenVME system. Users, data files, services, communications network structure and hardware devices are all examples of catalogued VME objects. The information recorded in the Catalogue for each object includes values for some or all of the attributes defined for all similar objects (*e.g.* a file's creation date or a hardware device's address). In addition, the catalogue records *relationships* between objects; such relationships include relative naming, ownership and privacy information .



For a *data file* object, for example, the placement of the file data on physical storage and its file organisation are recorded, enabling a copy of the relevant file access code to be loaded and particularised for the file. The Catalogue objects representing the communications network structure (including routing and addressing) are used by the OpenVME system to determine the protocol handlers required to support end-to-end communications (*e.g.* a virtual circuit, or transport connection). A *file description* object is a generic template which, suitably supplemented with the attributes of a particular file, enables a file object of that description to be created.

Some catalogue objects represent collections of objects with common attributes so that the objects in the collection need not be individually recorded in the Catalogue. For example, a *Library* object represents a collection of several files all of which inherit the file attributes associated with the Library in the Catalogue. This technique is highly efficient and also allows new types of collection (*e.g.* alternative filesystems) to be incorporated into the system.

Objects are, in general, identified relative to other objects; this results in a directed graph whose nodes are objects and whose arcs are the relationships between them; the graph is rooted at the *installation node*. Each relationship is labelled with a *selector,* an identifier which distinguishes that relationship from other relationships - *e.g.* the name of a file relative to a filegroup. Thus any object can be identified by specifying an initial node and a sequence of selectors identifying the relationships to be traversed to reach the required object; the initial node of this sequence may be an element of explicit or implicit *context*, or the installation node. Each relationship may have *security attributes*, constraining the manner in which one object may be accessed relative to another. Each object may also possess security attributes. The security attributes associated with objects and their relationships are the basis of the OpenVME security architecture.

## Object selection & currencies

The sequence of selectors used to specify an object selection is known as a *hierarchic name*. Although in many cases such a name uniquely defines a catalogued object two successive selections with the same hierarchic name *may* select different catalogue objects - if, for example, a new version of an object has been created between selections.

The OpenVME system provides a means of establishing an efficient reference to a particular object, a process known as *selection*. Such a reference is known as a *currency* and represents not merely the selection of an object but also the context in which it was selected, including the security attributes associated with that selection. A currency, for its lifetime, always represents a localised, temporary reference to the same catalogued object; once established, it may also be used as a starting point for further object selection.

Any currency may be used as the starting point for selection of other objects reachable, via catalogue relationships, from the object represented by that currency. In addition, the system maintains named *contexts* for many object types; a context is a list of objects which may be used as a starting point for selection; a null context is used as a starting point when none is otherwise specified. These mechanisms allow the context of object selection to be refined so that object selection may be performed efficiently; the first provides the means of establishing an efficient starting point; the second enables the establishment of a limited (but changeable) set of alternative starting points as a context.

It is also possible to select, in sequence, all the objects of a given type reachable from a given starting object without knowing in advance the selector of each object. This is achieved by selecting *along* a set of objects, each selection being relative to (the currency of) its predecessor.

## Naming and Binding

### *Naming*

As described above, catalogued objects are *named* by a hierarchic name, a sequence of selectors, relative to some starting point. Such a name may not uniquely identify a particular object for all time. For example, filenames may be qualified with *file generation numbers (fgn)* and if none is specified the latest generation is selected; if a new generation of a file is created between one selection of a named file and the next, different files are selected.

An important naming domain is that of *loadable objects* which are aggregates of code and data used by the VME Loader to construct executable representations of VME objects within a VM. Each loadable object may contain several named objects as well as references, by name, to external objects. These names are resolved in the *loading environment* of each VM, a dynamic set of libraries which is searched to locate a loadable object by name.

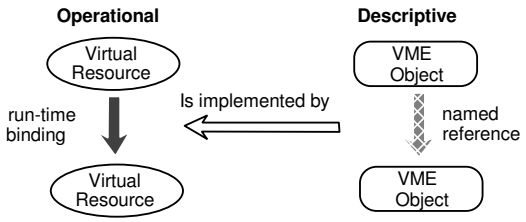
### *Binding*

A name represents a potential reference to a value. When interpreted in a particular environment or context, the name may refer to a specific value, which is said to be *bound* to the name, or be *unbound*. The association of a name, in a particular context, with a specific value is termed a *binding*.

For example, the resolution, via the loading environment, of an external reference to a named object creates a binding of the object to the name. Similarly, the selection of an object and acquiring a currency for it is an example of a binding of the object to the currency.

Binding may take place, in various forms, within the application source, when the application is compiled, when execution of the application commences or

immediately prior to usage of the resource. In general, earlier binding results in greater efficiency and later binding in greater flexibility.



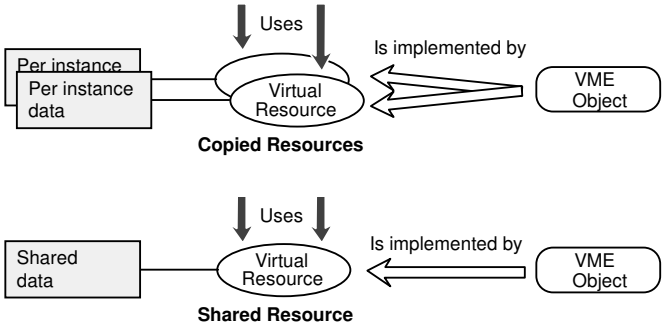

## Virtual Resources

A *Virtual Resource* is a scoped, executable representation of a VME object. It comprises data and a set of associated *procedures* implementing the operations of the resource. For each object, the Catalogue contains (or indirectly refers to) the information required by the OpenVME system to create an executable instance of the object: a Virtual Resource representing the object. When an object is actively shared by several users, a Virtual Resource may be created for each usage.

### *Virtual Resource Creation*

Creation of an individual Virtual Resource requires the loading of the executable code required to perform the operations supported by that resource, constant data, and a dedicated copy of data which may differ between instances. Resources specific to an application are generally instantiated in the VM of use; code is loaded into the VM and links established to supporting resources and memory is allocated and initialised for the per-instance virtual resource data.

A class of similar system resources is often supported by a *subsystem* whose code and constant data is pre-loaded, and which maintains per-instance virtual resource data in pre-allocated tables.



### *Copying and Sharing*

When an object is to be used, it is first selected and then a Virtual Resource is created in the appropriate VM. It is possible for there to be more than one concurrent usage of the same object, either in the same VM or in different VMs. A separate Virtual Resource may be created for each usage (copying); in some cases a single Virtual Resource may be shared between some or all usages (sharing). Since a Virtual Resource may, in general, make use of other Virtual Resources there may be several points at which the issue of copying *vs.* sharing may arise.

### *The VM as a Context: the OpenVME In-process Architecture*

As already described, object selection, the generation of currencies and the instantiation of Virtual Resources occurs within a VM context. This approach minimises interactions between VMs and allows direct bindings between Virtual Resources to be established solely within the VM context. Thus the set of virtual and real resources associated with each VM, including shared resources, can be simply accessed and operated upon within the VM. This fundamental architectural principle results in the *VME in-process architecture*.

## Block-structured Resource Allocation & Control

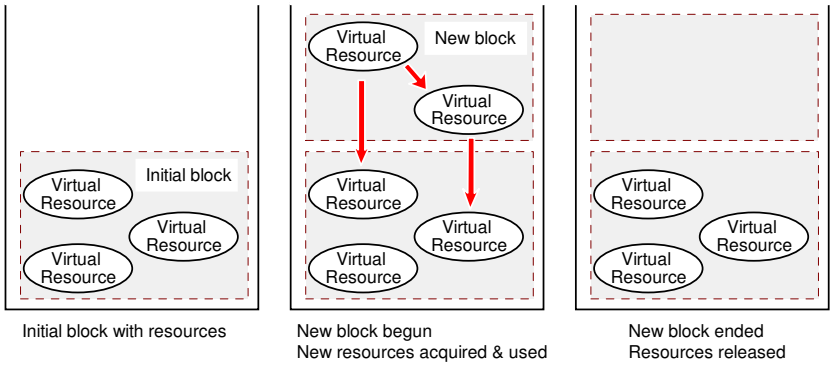
The previous section described the selection of particular objects, binding them into the environment and the creation of the corresponding Virtual Resources within a VM. A VM will often need to establish a new execution context with associated bindings and Virtual Resources while some activity is undertaken; when the activity has completed, this context may no longer be required and a new context may need to be established for a subsequent activity.

Most Virtual Resources are associated, explicitly or implicitly, with real resources. In any multi-programming environment there may well be contention for real resources which can lead to deadlock. The system therefore provides a resource scheduling mechanism which allows an application to allocate the resources it requires for some activity as a set.

Clearly, some mechanism is required for destroying bindings and de-allocating resources when they are no longer required. The model adopted by OpenVME is of nested *blocks* (or *resource allocation blocks*): a block is entered when previously requested resources are available; further resources may be acquired during the existence of the block; when the block is ended all resources associated with that block are de-allocated. Resources associated with enclosing blocks (*i.e.* blocks previously entered) are unaffected by entering and ending new blocks. This model provides a means of temporally scoping the allocation of real resources and the lifetime of bindings and virtual resources.

The formal association of resources with particular blocks has major advantages. If execution of an application commences in the context of a new block, *all* the

resources associated with that application can be simply de-allocated by ending the block. In particular, even if the application fails voluntarily to relinquish resources it has acquired (*e.g.* as the result of an error) they can be forcibly released. The orderly management of system resources is therefore not critically dependent on co-operative or well-behaved applications.




The currencies and Virtual Resources associated with some object types may be explicitly and individually de-allocated. This is essential for certain multi-threading applications in which there is never generally a point at which *all* resources in the current block can be conveniently de-allocated and thus a block can never be ended.

## Object Privacy & Security

Each object and each relationship between objects may possess security attributes. At any instant, a VM is associated with a single Workgroup or Person (a User). Traversal of a relationship between objects may be restricted, by the security attributes associated with that relationship, to certain users, inclusively or exclusively, and by type of access or usage (which must be specified during selection). These attributes may be used as the basis of a static discretionary access control policy.

When an object is selected, there is a potential flow of information between that object and the VM which is performing the selection; this may, in turn, lead to information flows to or from other objects. The security attributes of an object may include properties such as integrities, security level, caveats and codewords which are checked for compatibility with the security status of the VM. Making a selection may cause the security status of the VM to be modified. These *dynamic* features are used to prevent the possibility of unauthorised flows of information - *e.g.* from a high sensitivity object to a low security object. The static and dynamic security features may be used as the basis of a mandatory access control policy.

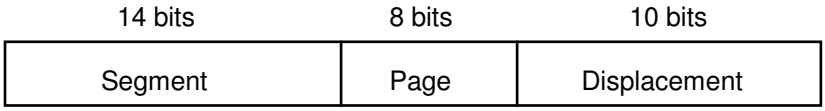
# The Containment Architecture: Virtual Machines

## Introduction

The Virtual Machine is a secure container for a set of virtual resources. This section describes the architecture of a Virtual Machine, including the Virtual Memory and Process models, and shows how these models provide a protected environment for the execution of processes.

## The Virtual Memory Model

The instantiation of a virtual resource within a VM is achieved by loading or creating, in memory, the areas of code and data which constitute an executable representation of that resource. In the OpenVME architecture each VM is provided with a V*irtual Memory*, an address space into which the areas representing the virtual resources of the VM are mapped. Each VM has a virtual address space of $2^{32}$ bytes. A *virtual address* has the following structure:




This address space of a VM is structured into *segments*. Segment numbers are divided into two ranges: *Public* segments (8192 - 16383) and *Local* segments (0 - 8191). Public segments are common to all VMs whereas local segments are allocated and used independently in each VM. Various fundamental properties associated with an area of virtual memory are enforced by the architecture at the level of the segment containing the area. Each segment is further divided into up to 256 *pages* to facilitate memory allocation and management and to reduce memory fragmentation. A segment may be of fixed or variable length.

Segments are the fundamental units of memory allocation and a logical area is usually mapped within a single segment. A *Super-segment* is a set of contiguously addressed segments all of which, other than the last, are of maximum size. A logical area whose size is greater than the largest permissible segment can be mapped, transparently, onto a super-segment.

### *Public Segments*

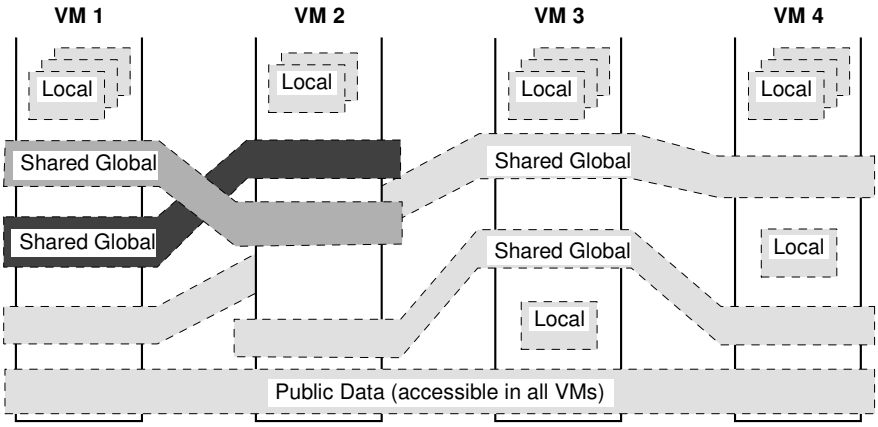
Public segments are used by the OpenVME system to support system resources. Since these may be used (either concurrently or sequentially) by several VMs, instantiating them in Public segments ensures that they are uniformly accessible in any VM which needs to use them. Public segments thus provide space for the instantiation of system resources commonly accessible by several (or all) VMs.

### *Local Segments*

Local segments are used to support virtual resources particular to a VM. They are only accessible *locally* to the VM in which they have been allocated. Local segments thus provide space for the instantiation of the virtual resources required by a VM.

### *Global Segments*

Some applications may be implemented by the active co-operation of several VMs. There are two fundamental models for achieving this co-operation: *message passing* and *shared memory*. Global segments provide shared memory areas across a set of VMs. A shared memory area may made be accessible as one (or more) contiguously addressed local segments in each VM sharing the area. The area is not necessarily addressed by the same local segment numbers in each VM. Shared memory areas are loadable, catalogued objects and access to them is controlled by catalogue security attributes. Global segments provide a more selective and flexible way of sharing memory areas than Public segments.




### *Segment & Page Properties*

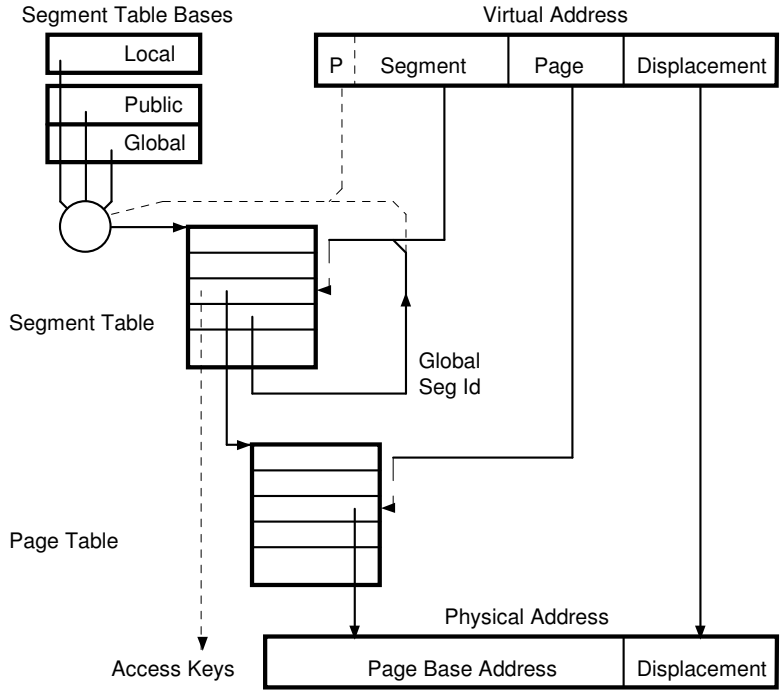
Various properties are associated with an area of virtual memory on the basis of the segment(s) in which it is located. Segment properties include the size of the segment, access keys which control the ability to read, write or execute the contents of the area, and attributes which define paging policy for the segment.

Any segment or page of virtual memory may be present in physical memory, absent (*e.g.* when on secondary storage) or unallocated; this status is indicated in the appropriate segment or page table entry. Other low level properties of virtual memory are generally defined at segment level - in particular, *Access Permission Field*, containing the *Access Keys* of the segment for *execute, read* or *write access* to a process executing at a particular *access level*. This protection mechanism is described in more detail below.

### *Mapping Virtual to Real Memory*

The OpenVME system maps this virtual memory onto physical memory. The mapping process is known as *address translation.*

Address translation is logically a three stage process. First the segment number is used to index an entry in a *Segment Table*; this points to a *Page Table* for the pages in the segment. The page number is used to index an entry in the Page Table; this defines the address, in physical memory, of the start of the page. Finally the displacement is concatenated with this page start address to yield a physical address.




Provision is made for areas to be shared between VMs by using *Indirect Segment*s which are mapped onto *Global Segment*s. The Segment Table entry for an indirect segment is marked to indicate that, rather than pointing to a Page Table, it contains a Global segment number; this is used to index a new Segment Table entry in a Global Segment Table, pointing to the required Page Table.

The virtual memory particular to a VM is that addressed by local segment numbers (0 - 8191) and hence via segment and page tables rooted at the Local Segment Table Base (LSTB). A value of LSTB therefore fully defines a VM's local virtual memory.

## The Process Model

The execution of code associated with a set of virtual resources within a VM is termed a *Process*; the set itself is termed a *Process Image* and the state of execution, defined by values of processor registers, is termed the *Process State.*

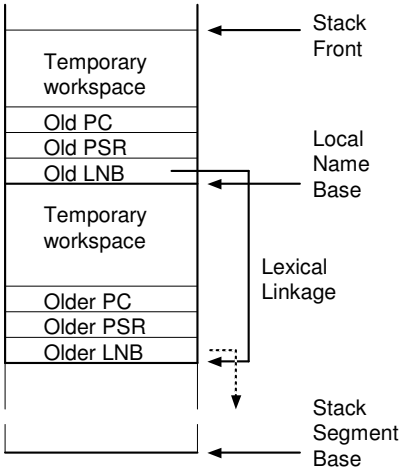
### *The Process State*

The Process State comprises:

- a Local Segment Table Base (LSTB), defining local virtual memory
- a Program Counter (PC), pointing to the current instruction
- a Process Status Register (PSR)
- Stack Pointer registers, defining the current stack frame (LNB, SF)
- Address Base registers, used to address simple operands (XNB, CTB)
- an item Descriptor register, used to address complex operands (DR)
- an Index register, used as an operand address modifier (B)
- a variable length Accumulator, holding computed values (ACC)
- an Interval Timer (IT) and an Instruction Counter (IC)

### *The Stack*

Each process is provided with a *Stack*, an area of Local virtual memory used for automatic allocation and de-allocation of temporary working space, for expression evaluation, and for storing dynamic linkage information. A process automatically has full read and write access permissions to its current stack. The diagram below shows the major structural concepts associated with the stack. The stack is divided into regions termed *stack frames*; the current stack frame is defined by the LNB and SF registers. At the start of each stack frame linkage information points to its predecessor; the remainder of the stack frame is used as temporary, dynamically allocated workspace for the code with which it is associated. The usage of stack is described in more detail below.



### *Operand Addressing*

Operands are normally addressed by specifying an Address Base register (containing a Virtual Address) and a literal displacement; this is termed *direct addressing*. For on-stack items, the LNB and SF registers provide the means of addressing items in the current stack frame. Addressing of other data items can be achieved using XNB or CTB, or relative to the current PC.

### *Indirect Addressing: Descriptors*

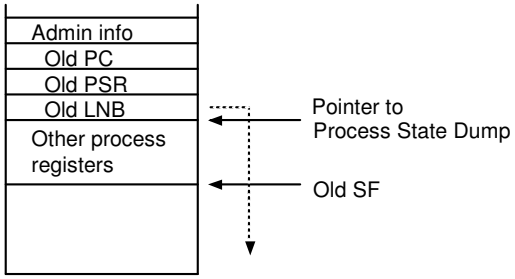
A more powerful means of addressing data items is via a *descriptor* in DR or in memory, possibly indexed by a modifier in B or in memory; this is termed *indirect addressing*. A data descriptor is a typed reference to a structured data item and comprises a virtual address pointing to the start of the item together with information defining the type and size of the item. Descriptors may be used to refer to scalar items; vectors of scalar items; byte strings; or other descriptors. In addition various special descriptors are used to mediate entry to or exit from procedures etc.

### *Processes & sub-processes*

A VM may have multiple processes each of which has its own stack. No more than one process may be active in a VM at a given instant. New processes are created by the system when it is necessary for execution to commence on a new stack. They may also be created by user software, in which case they are termed *sub-processes.* Sub-processes provide a facility similar to *threads* in other systems.

### *Process activation & suspension*

When a process is inactive, the Process State is stored by the processor on the top of the process stack. This creates a special stack frame termed a *Process State Dump.* The processor provides an *activate* mechanism which, given a pointer to a process state dump, causes the processor to resume execution of the process from the state indicated in the process state dump.



Information about the state of each process known to the system is recorded in a protected data structure - the *Process Management Table*. The entry for each process includes:

- the Local Segment Table Base (LSTB);
- a pointer to the on-stack process state dump;
- the Instruction Count and Interval Time;
- a pointer to the System Call Index used to decode System Calls;
- the identity of the System Call Table used for in-process interrupts;
- data related to event handling and process suspension.

## Protection & Privilege

### *Virtual Machines*

The local virtual memory of one VM cannot be accessed from any other VM except for segments shared between VMs. This can only occur as a result of explicit action by *both* VMs. The access permissions for the shared (Global) segment can be set independently for each local (Indirect) segment mapped to the shared segment.

### *Access Levels*

One property of a process, defined in the Process Status Register (PSR) of the process state, is the *Access Level* at which it is currently being executed. The lower the Access Level, the more privileged the process. It is an important principle of the OpenVME architecture that use, by a process executing with a certain level of privilege, of more privileged resources should be strictly controlled.

Many resources are protected by an *Access Key*. When a process references such a resource, the process Access Level is compared with the appropriate Access Key: if AL ≤ AK then access is permitted. Examples of such resources are:

- Virtual Memory, protected by the segment APF
- Privileged Virtual Resources, protected by the System Call Access Key
- Software Events, protected by Cause, Entry & Notify Access Keys
- Attributes of catalogued objects, protected by read & write Access Keys

In the case of virtual memory accesses and System Calls, the enforcement of access checks is supported by hardware mechanisms.

The means by which the Access Level of a process may be changed is strictly controlled (by the *System Call* mechanism, described later). Access Levels are thus able to provide the basis for protection between virtual resources associated with different levels of privilege within a VM.

### *Memory Protection*

Each segment of virtual memory has an Access Permission Field (APF), defined in its Segment Table entry, comprising an *Execute Permission Bit* (EPB) and *Read* and *Write Access Keys* (RAK & WAK). Execution of code is only possible from a segment with EPB set.

### *Address Validation*

Arbitrary virtual addresses may be formed by a process and subsequently passed to more privileged software. The processor provides an instruction (VALidate) which validates whether, and in what mode(s) of access, an area of virtual memory is accessible to a specified Access Level.

# The Imperative Architecture: Procedures & Data

This section describes how Virtual Resources are implemented within the OpenVME architecture.

## Procedures and Procedure Linkage

### *Procedures*

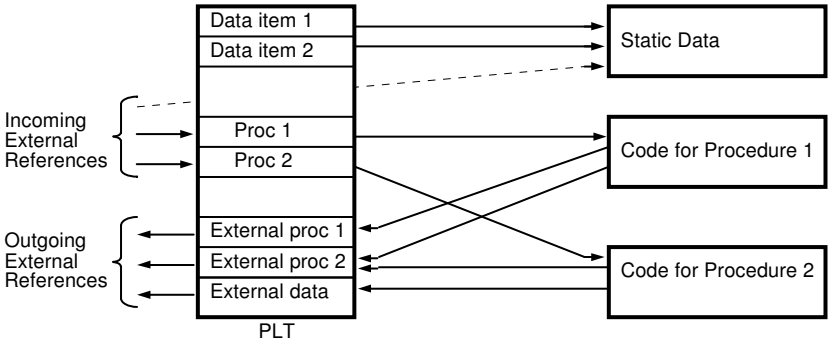
The basic unit of execution is a *procedure*. Formally, a procedure is an encoding of some algorithm which operates on a set of *parameters* and may produce a *return value.* A procedure is executed in a context comprising two components: the *Static Context* and the *Dynamic Context*.

### *The Static Context of a Procedure*

A procedure may have *static data* areas uniquely associated with it - *i.e.* not, by default, directly accessible from other procedures. It may also reference *external* data areas and procedures. These elements of the *static context* of a procedure are generally accessed via a *Procedure Linkage Table* (PLT). Usually several related procedures and their associated static data areas (some of which may be shared between several procedures) are combined into a *module* and share a PLT. The PLT also contains pointers to procedures within the module which are accessible, externally, by other procedures or modules.

### *Instantiation of Virtual Resources*

A simple Virtual Resource can be represented as a module with a copy of its associated static data areas and PLT. Additional instances of this resource only require further copies of the static data and PLT: the code areas of a module are sharable between several instances.




The diagram above shows an example of a Virtual Resource with two procedure entry points - *e.g.* a queue, with procedures for adding and removing a queue entry. The queue itself would be represented in the Static Data. An additional queue can be instantiated by creating further copies of the PLT and the Static Data, unique to that instance.

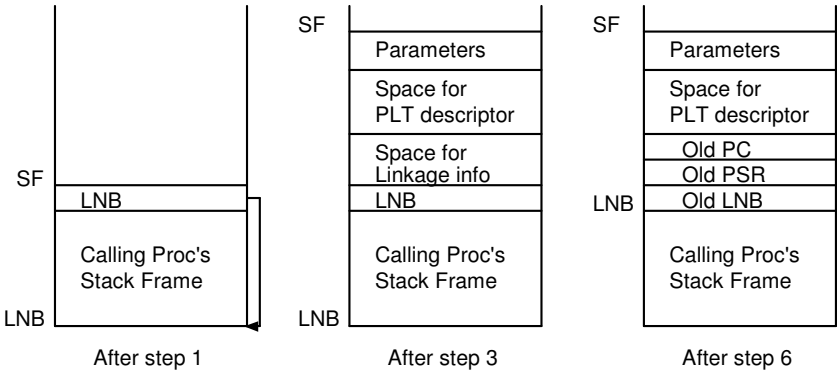
### *The Dynamic Context of a Procedure*

The stack is used to hold the *dynamic context* of a procedure. Parameters (and dynamic linkage information) supplied to a procedure when it is invoked and additional workspace, temporarily allocated until completion of the procedure, are stored in a stack frame created for that purpose.

### *The Procedure Call Mechanism*

The *Procedure Call Mechanism* is the means by which procedures are invoked or *entered*. A corresponding mechanism, the *Exit Mechanism* is used to return or *exit* from a procedure when it has completed. There are several different types of Call but they all have the following outline sequence of steps in common:

1. Create lexical linkage by storing current LNB on top of stack;
2. Create space on top of stack for linkage & PLT info., by raising SF;
3. Store parameters on the top of stack;
4. Establish a new stack frame for the procedure to be entered, by raising LNB;
5. Store dynamic linkage information at the start of the new stack frame;
6. Enter the called procedure (usually via a descriptor);
7. On entry, the called procedure allocates temporary workspace on the stack.




### *Procedure Exit*

The Exit mechanism also has several variants (depending on the linkage information) but they also have a common outline sequence:

1. The returning procedure may place a return value in the Accumulator (ACC);
2. The current stack frame is collapsed, re-establishing the stack frame of the calling procedure, by setting SF to LNB and restoring LNB from the link;
3. Resume execution of the calling procedure immediately after the Call, using Old PC.

### *Passing Parameters & Return Values*

A parameter may be a *value* (*e.g.* integer, character, procedure etc.) or a *reference* to a value (or to another reference). A Return Value may be a value or a reference to a data item which will remain in scope after the return.

### *Parameter Validation*

Data referenced by a reference parameter may be changed by a procedure. It is therefore essential to ensure that the calling procedure is permitted to access the area of virtual memory specified by such a parameter. The VALidate instruction enables a called procedure to ensure that the calling procedure is permitted to access any reference parameters it passes.

### *Types of Call & Return*

There are several different types of call which are distinguished by the type of the descriptor used to reference the called procedure. The type of descriptor is usually determined by the Loader, when fixing up references between modules instantiating Virtual Resources, and is dictated by the relative privileges (Access Levels) associated with the calling and called procedures; a more privileged procedure is usually an operation associated with a more privileged Virtual Resource. Each *call* type has a corresponding *return* type which restores the Access Level of the calling procedure; the call types are:

1. Normal Call to a procedure of equal privilege (Access Level unchanged);
2. Inward Call, to a more privileged procedure (decrease of Access Level), and Outward Return. This is a *System Call* and is made via a *System Call Descriptor*. The called procedure is entered on the current stack and a normal link is stored, recording the Access Level of the calling procedure;
3. Outward Call, to a less privileged procedure (increase of Access Level), and Inward Return. This is also a System Call but in this case the called procedure must be executed on a new stack since otherwise it might corrupt more privileged data associated with the calling procedure. A special System Call link is stored in the stack frame on the new stack;
4. Sub-process Call, used to resume execution of a sub-process.

Call types 1 & 2 are normally performed automatically by the processor; types 3 & 4 need system software intervention to effect the switching of stacks.

### *System Calls*

*System Calls* provide the fundamental mechanism for controlling entry to the procedural interfaces of more privileged Virtual Resources. A System Call is performed by executing a procedure call via a *System Call Descriptor*. A System Call Descriptor contains values used to identify an entry in a *System Call Table*, which is administered by privileged system software. This entry defines:

- the *System Call Access Key* (SCAK), the maximum permitted Access Level of the calling procedure;
- the called procedure, defined by a PLT or code descriptor;
- the PSR, including Access Level, to be adopted by the process during execution of the called procedure.

The combination of SCAK and PSR in the System Call Table entry define a specific capability for a process executing at one Access Level to enter a specific procedure at another Access Level. In particular, the SCAK ensures that entry to privileged procedures is only permitted to processes which are already sufficiently privileged.

### *Forced Procedure Calls*

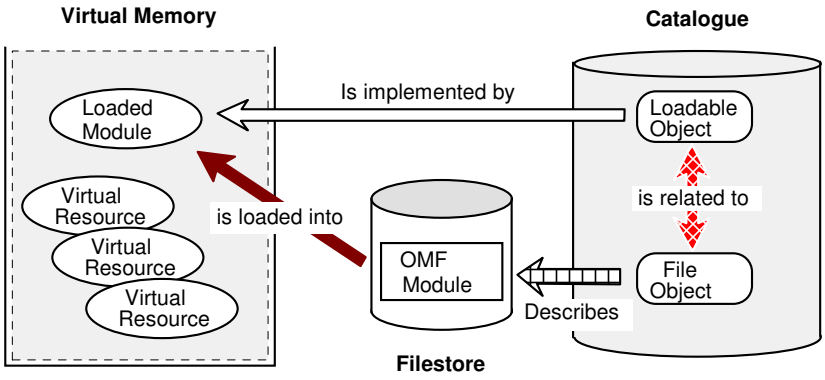
Various circumstances require the normal execution of a process to be interrupted by the execution of some procedure not directly invoked by the process itself. This may be the result of an asynchronous event external to the VM; it may be the result of a synchronously detected exception condition within the VM. In all such cases, generally known as *interrupts,* a *forced procedure call* is made to the required procedure, by storing a process state dump on the current process stack and entering the procedure via a system call. The parameters for the procedure are usually supplied by the source of the interrupt which may be hardware or software. When the interrupt procedure exits, the interrupted process is resumed at the point at which it was interrupted, by using the process state dump to restore the process state.

## Object Modules and Loading

### *Object Module Format (OMF) Modules*

The OpenVME model of execution is based on the interaction of Virtual Resources. Some of these resources are supported by OpenVME subsystems directly; others are created by the loading of *OMF Modules* into virtual memory and linking them, via *names*, to other resources. Most modules are loaded into the Local virtual memory of a VM but a module can be loaded into Public or Global virtual memory if the resource is to be shared throughout the system or between specific VMs.

An OMF module contains the information required by *Loader* to construct the corresponding Virtual Resource(s) in the execution environment, including definition and initialisation information for *code* and *data* areas; object *names*; area, object and name *properties*; references to *external* names; and *diagnostic data*.



### *OMF Areas*

The components of a module required to construct a Virtual Resource are termed *areas*. Each area occupies a contiguous region of virtual memory having particular properties. The OMF representation of an area defines various properties including its length and may also define initial values for the area when it is first created in virtual memory.

### *OMF Objects*

Areas are merely convenient containers for code and data generated by a compiler. They contain executable representations of entities such as procedures and data arrays. Such entities are termed OMF *Objects*.

OMF objects have properties which define the ways in which they can be used. For example, every OMF object has a *named point* which, for a procedure, defines the first instruction to be executed and for a data array defines the start of the array. A procedure may have properties defining the Access Levels at which it should be executed and from which it may be called, and indicating that a System Call is required to enter it. In general the properties of an OMF object in an area are independent of the properties of other OMF objects in the same area.

OMF objects may have *names*. An object must have a name if it is to be referred to from outside the module. OMF object names have properties some of which affect the way in which the *name* may be used ("*bound")*, and others which affect the way in which the *named object* itself may be used once bound.

Some OMF object names may be designated *module keys* of the OMF module. Such names are the means by which the module may be identified for the purpose of loading. This may occur as the result of an explicit request to load the module or when an already loaded, executing resource refers to an OMF object in the module via its keyed name.

### *Templates*

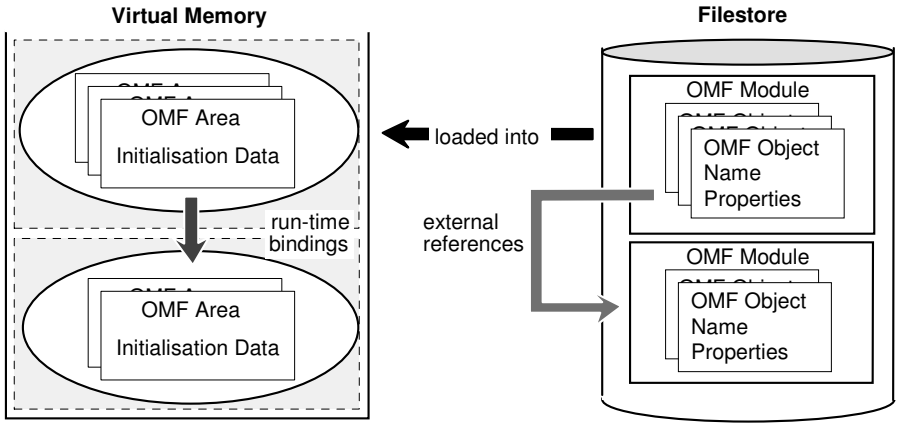
An OMF procedure object which is intended to be directly callable by an interactive user or from an SCL procedure may have a *template* which defines the formal parameters to the procedure and associates them with *keywords*. Each parameter may have a default value defined as part of the template. This facility is used extensively for SCL commands both to help interactive users supply the correct parameters and to allow parameters to be specified by keyword rather than position.

### *External References*

Objects in OMF modules may require access to OMF objects in other modules, either to call procedure objects or to access data objects. These requirements are expressed as *external references* from the module. External reference *properties*

determine whether loader attempts to satisfy such a requirement (by loading a module with a module key corresponding to the required name) when the module is loaded, or whether resolution of the reference is *delayed* until it is used. The mechanism by which external references in module can cause another to be loaded is known as *cascade loading* and continues until all references are satisfied, or have the delayed property, or cannot be satisfied in the current loading environment. External references may be qualified by a library name which restricts loader's search to libraries of that name.

Some of the external references of a module may not be satisfied when it is loaded. In such cases the reference is temporarily satisfied by a special type of descriptor, known as an *Escape Descriptor*. If an instruction attempts a reference via an Escape Descriptor, hardware forces entry to an *Escape Routine* whose address is contained in the descriptor. The normal action of the routine is to re-attempt to satisfy the reference (for example, to an object in a subsequently loaded module) and the instruction is then restarted.




### *The Loading Environment*

OMF modules are stored in filestore, in libraries. Each VM has a *loading environment* which defines the libraries to be searched to find a named OMF object. The loading environment comprises:

- the Level List for modules executed at the current Access Level;
- the Context, for modules executed at a more privileged Access Level;
- the set of resources already loaded in the VM.

The Level List and the Context are each lists of libraries to be searched for a module containing the requested OMF object. Objects loaded from the Level List are loaded into the VM as Virtual Resources associated with the current Access Level. Each library in the Context has a defined Access Level; Objects loaded

from the Context are loaded into the VM as Virtual Resources associated with the Access Level of the library from which they are loaded.

The names of OMF objects are recorded as they are loaded and subsequent references to those names are, by default, satisfied by the already loaded objects. Such objects are therefore part of the loading environment. It is possible to modify this action by associating a property with an OMF object name; each time an OMF object named with this property is loaded, a new instance of the resource is created.

### *Privileged Interfaces and System Calls*

Where required, loader establishes System Calls to enable a procedure executed at one Access Level to enter a procedure executed at a more privileged Access Level. This happens when a module loaded at one Access Level has an external reference to a more privileged interface in another module.

Most privileged system interfaces to Kernel and Director are entered by System Calls which are established during system load. There is a special file, the *steering file*, which defines the Access Level from which such interfaces may be called and at which the corresponding procedures are executed. The steering file defines a *command class* for each interface; each library has a *command mask* which restricts the command classes that software loaded from the library may use. In addition, the steering file may restrict the ability to invoke privileged interfaces to VMs having particular libraries in their loading environment.

# Inter-Process Communication & Synchronisation

## Events & Interrupts

### *Events*

The *Event System* provides the fundamental means of enabling processes, or different portions of one process, to be synchronised. An *Event* is a synchronisation channel on which only two basic operations can be performed:

- *Causing* an event: putting an event message into the channel;
- *Notifying* an event: removal of an event message from the channel.

Each pair of operations, cause and notify, is termed an *Event Occurrence*.

There are two types of event, characterised by the way in which the event message can be notified:

- A *Flag Event*, which holds an event message until a process indicates that it is ready to be notified by *waiting* for the event or *reading* it;
- An *Interrupt Event*, which holds the message until either it is permitted to be notified to the process by the transfer of control to a nominated *interrupt procedure* (which receives the event message as a parameter); or it is explicitly read or notified as a flag event before transfer of control is permitted. When the interrupt procedure exits, the process resumes at the point and Access Level at which it was interrupted.

Events may be global or local.

- A *Global Event* is a catalogued object which may be caused or notified in any VM which has selected the event and connected to it.
- A *Local Event* is caused and notified in the same VM and can be created dynamically without a corresponding catalogued object.

Before a process may cause or be notified of an event, it must first establish its right to do so by *connecting* to the event; in the case of an Interrupt Event a descriptor to the interrupt procedure must be supplied. Certain properties associated with the event connection may also be specified; these are:

- The *Cause Access Key* (CAK): the Access Level at or below which the event may be caused;
- The *Notify Access Key* (NAK): the Access Level at or below which the event may be notified;
- The *Entry Access Key* (EAK): the Access Level at or above which a process must be executing before entry to the nominated interrupt procedure is permitted.

By default, CAK, NAK & EAK are set to the current Access Level of the process. Setting EAK greater than NAK (known as an *unequal entry condition*) ensures that interrupt procedure cannot be entered recursively.

### *Suspension & Notification of Flag Events*

A process may *wait* for an event (or a list of events) and, if no notification is already outstanding, the process is then *suspended* until a notification is received from (any one of) the event(s). When the notification occurs, the event message (and, in the case of a list of events, the event currency) are returned to the process. It is also possible to *read* an event; in this case, if there is no outstanding notification the process is not suspended.

When a process waiting for a Global Event is to be suspended, it is added to a queue of processes waiting for that event; this queue determines which process receives the next notification for each particular event. When a process waiting for a list of events has its suspension lifted, any other queue positions associated with the same wait request are discarded; having dealt with the notification, the process may *wait* again to rejoin the queue(s).

### *Suspension & Interrupt Events*

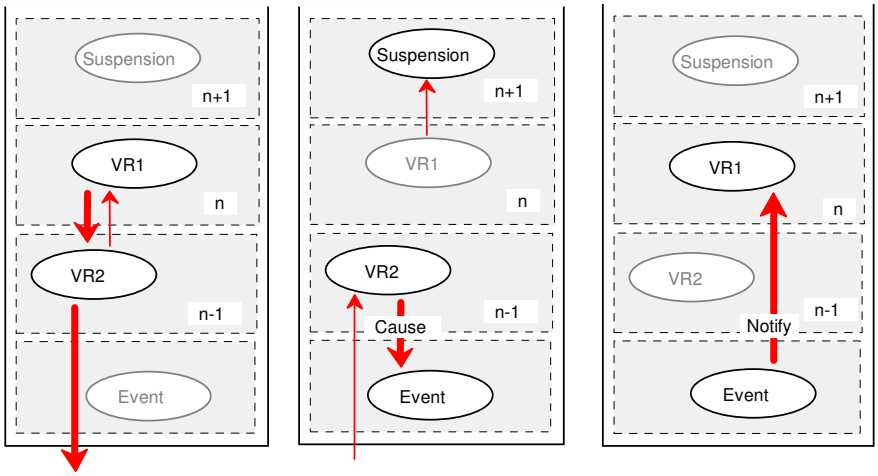
A process may be interrupted to execute an interrupt procedure provided that it is not currently executing, or suspended at a lower Access Level than the EAK.

### *Use of Interrupt Events*

Unequal Entry Interrupt Events provide a powerful technique for enabling multi-level structures of Virtual Resources to co-operate asynchronously within a VM and this is used extensively within the OpenVME system.

As an example, consider a Virtual Resource VR1 which requires the services of a lower level (and more privileged) Virtual Resource, VR2. Suppose that some operation supported by VR2 may operate asynchronously. VR1 can pass to VR2 an event currency for an interrupt event which VR2 may cause when the operation terminates; if VR1 executes at Access Level n, then the CAK must be $\geq n$ (allowing it to be caused by VR2), the EAK might be n+1 and the NAK would be n.

Suppose that VR1 calls a procedure in VR2 to initiate the operation after which control is returned to VR1. VR1 may then initiate further operations, voluntarily suspend itself, or return control to a resource at a higher Access Level. When the operation terminates, VR2 causes the specified event and eventually the interrupt procedure in VR1 will be entered at Access Level n. Handling of the event notification in VR1 cannot be interrupted by a further notification of the same event until the process voluntarily suspends or returns control to a higher Access Level. The event system acts as a queuing mechanism allowing virtual resources at each Access Level to complete the handling of one event before being notified of the next.




VR1 calls VR2
VR2 initiates asynch operation

VR1 suspends at AL n+1
Asynch operation terminates
VR2 causes event

Event notified to VR1, EAK = n+1
VR1 enters interrupt proc, NAK = n

In the example above, the asynchronous operation invoked by VR2 might be an operation on another Virtual resource, whose termination would be notified to VR2 by an event notification. Recursive use of event in this way allows large numbers of asynchronous operations to be in progress concurrently, at various Access Levels, within a VM. Ultimately the lowest level sources of asynchrony are events caused in other VMs or those associated with hardware interactions.

### *Interrupts*

An *Interrupt* is logically an interrupt event caused by hardware which is used to notify the occurrence of some hardware detected condition. Such conditions include:

- Hardware detected system exception conditions
- Hardware detected program exception conditions
- Virtual Memory exceptions (*e.g.* a page fault)
- Real-Time Clock, process Interval Timer & Instruction Counter conditions
- Hardware supported inter-VM interrupt
- External interrupts (*e.g.* from input/output devices)
- Software generated interrupts (*e.g.* some special System Calls)

Interrupts are notified to the most appropriate VM. In the case of synchronously detected process conditions or errors, they are notified in the current VM. The interrupt procedure, an associated value of PSR, and the process in which it should be executed are always defined by a System Call Table Entry whose identity depends on the nature of the interrupt. External interrupts are directed to the appropriate VM and process via a Kernel data structure known as the *Unit Table* which also identifies the System Call. In-process interrupts are handled within the process by a System Call identified by the process's Process Management Table entry.

In most cases a process state dump is created on the stack of the currently executing process; all other (suspended) processes will already have such a dump at the top of their stacks. A forced procedure call is then made to the interrupt procedure in the process to be interrupted, using the stack frame established by the process state dump.

Sometimes software at the lowest levels of the system needs to execute a procedure without being interrupted. Whilst Access Levels can be used to ensure non-interruptibility within a VM, they have no effect on scheduling between VMs. A mechanism is therefore provided to allow a suitably privileged process to become *pre-emptive* (indicated by the PEP status bit in the process PSR). A pre-emptive process may not be interrupted; if an asynchronous interrupt condition arises while the current process is pre-emptive the hardware queues the interrupt until the process is no longer pre-emptive.

During an execution of an interrupt procedure, software may identify a different process to be executed on completion of the procedure from that which was interrupted. A mechanism is provided which allows suitably privileged software to nominate which process is to be executed on exit from the interrupt. When the interrupt procedure exits, the hardware resumes the required process by restoring its state from the process state dump at the top of its stack.

## VM & Process Scheduling

Events and interrupts provide basic mechanisms for scheduling execution within a VM. Between VMs, the system operates a priority scheduling system. Each VM has a *priority* which can change dynamically. The system schedules the use of real resources, including processing time and Input-Output requests, between VMs on the basis of priority algorithms, with fair allocation between VMs of equal priority. This is to enable high priority VMs to have access, as soon as possible, to shared resources while allowing lower priority VMs to use the resources.

The containment of logically related Virtual Resources within a VM ensures that priority scheduling applies to all activities associated with the work executing in the VM. This is one of the major benefits of the VME in-process architecture; it would be much harder to ensure that all execution took place at an appropriate priority if major functions were performed out-of-process.

## Shared Memory, Semaphores and Events

### *Shared Virtual Resources*

In some cases, a Virtual Resource is naturally useful only to the VM in which it was created. In other cases, especially within the OpenVME system itself, it is necessary to *share* a Virtual Resource between several VMs so that the *same* Virtual Resource can be used by those VMs. Such sharing is achieved by arranging that the data area(s) associated with the resource are loaded into Public or Global segments commonly accessible to the VMs; these are termed *shared* data areas.

A shared data area is (part of) a loadable module which is a catalogued object. Access to a shared data area is therefore controlled by the normal privacy and security mechanisms controlling selection and use of the catalogued object.

### *Synchronisation*

Use of a shared resource has several potential pitfalls. Consider a shared Virtual Resource representing a numerical counter; the memory location containing its current value is in a shared data area. The only operation supported by the counter is to increment its value and this is achieved, at the instruction level, by reading its current value, adding one and storing the new value. Two processes may have their execution of this operation interleaved in such a way that they *both* read the current value before *either* stores a new value; they will both read the *same* value, increment it, and store the *same* new value - only incremented once - which is clearly not what is intended. To eliminate this problem each process must be provided with a means of ensuring that its execution of the read-increment-store sequence *cannot* be interleaved with any other process attempting to execute the same sequence concurrently. Such a sequence is known as a *critical region* of code.

### *Semaphores*

The classic method of protecting critical regions (due to Djikstra) is by the use of a *flag*. A flag is a shared data item which may have one of two states: *on* or *off*. Two flag operations are supported: *set* which turns a flag on if it is currently off; and *unset* which turns a flag off if it is currently on. In either case, the operation is deemed successful if the flag was in the appropriate state before the operation and unsuccessful otherwise. A critical region can be protected by performing a successful set operation on a shared flag before entering the critical region and an unset operation on the flag when leaving the critical region.

When an attempt by a process to set a flag is unsuccessful, it will usually need to re-attempt the operation. At some later time the process which previously succeeded in setting the flag will complete its critical region and unset flag. However there is no obvious way in which the unsuccessful process can know when this occurs and hence when it is appropriate to re-attempt the operation. For this reason flags and their associated operations are difficult to use for many applications.

A more sophisticated approach is to use a *semaphore* which is a shared Virtual Resource having similar states (*reserved* and *free*) and operations (*reserve* and *release*) to a flag. However a semaphore reserve operation waits until the semaphore is free, rather than failing, if it has already been reserved. When the process which has reserved the semaphore releases it, the waiting process completes the reserve operation successfully. This scheme can be generalised to allow several processes to queue waiting to reserve a semaphore.

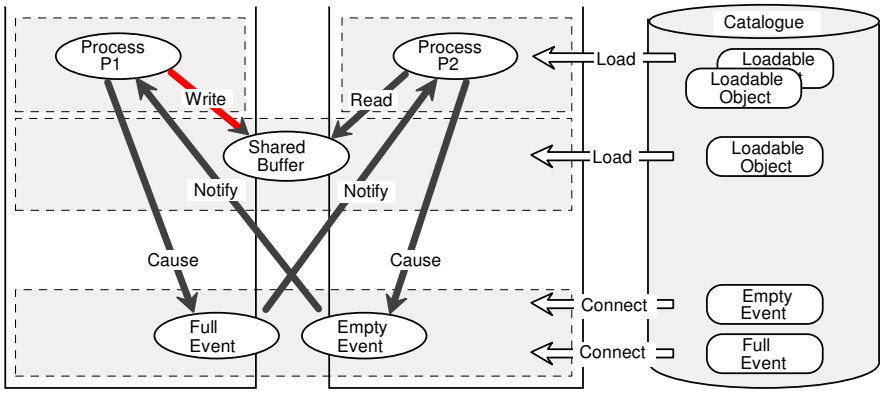
The OpenVME system supports a *semaphore* scheme similar to that outlined above; it interacts with priority scheduling mechanisms to ensure that processes which have reserved semaphores for which other processes are waiting are optimally scheduled. The scheme uses events to communicate between processes sharing the semaphore. When a process attempts unsuccessfully to reserve a semaphore, an event is created for which it waits. When the semaphore is released the event is caused, notifying the waiting process that it is now deemed to have reserved the semaphore.

### *Synchronisation with events*

Semaphores are an appropriate method of co-ordination when the patterns of synchronisation between processes are unpredictable. A common example is when a Virtual Resource is arbitrarily shared between several otherwise uncoordinated processes. In many cases a more disciplined pattern of synchronisation exists and can be more optimally implemented using events. A particular example of co-ordination by events is *message passing*.

Consider a Virtual Resource representing a shared message buffer, into which a process P1 may store a message for subsequent retrieval by some other process P2. It would be possible to protect the critical code regions which store and

retrieve the buffered message by a common semaphore. A more satisfactory means of co-ordination would be to create two events: one would allow process P1 to notify process P2 that the buffer was full; the second would allow process P2 to notify process P1 that it was empty. Neither process would then attempt to perform an operation on the buffer unless or until it had been specifically notified that the buffer was in a suitable state.




The OpenVME system provides several message-passing schemes, each optimised for a particular style of usage. Wherever possible standard open APIs are used to drive the message passing facilities; however, some interfaces are specifically designed for use solely within the OpenVME system. The message passing APIs include:

- Communications APIs such as the X/Open XTI;
- Inter-process communications APIs such as X/Open Message Queues;
- Proprietary APIs provided for specific internal usage.

## Timer Facilities

An important usage of events is for *timer channels*. An event is associated with each timer channel and the system provides facilities for requesting that the event is caused either at an absolute time or after a specified time interval; in the latter case, the request may specify that the event should be caused once only or periodically, each time the time interval elapses. The notification of the event is termed a *timer notification*. Only one timer request may be established for a timer channel at a time. The system provides a means of *cancelling* any extant timer channel request thus allowing a new request to be established.

In addition to real-time notifications, facilities are provided to request notification after a specified amount of processing time has elapsed.

# Input-Output

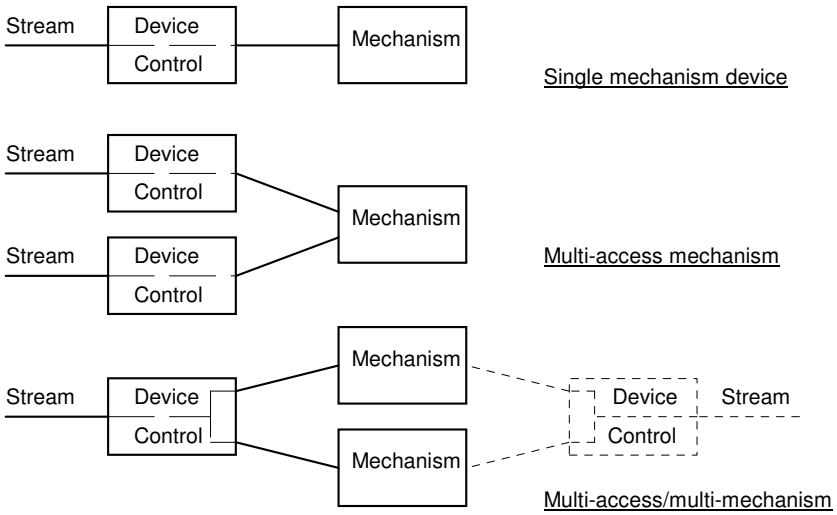
## Basic Concepts

### *Streams, Devices & Mechanisms*

The fundamental architecture has, thus far, only described activity within, and interactions between OpenVME processes. More generally there is a requirement to communicate with other processes, possibly executing in a very different environment. A particularly important example of this is performing *Input-output* (IO) operations on external peripheral devices. In many respects, requirement is similar to that for message passing within the OpenVME system and the fundamental Virtual Resource used to provide this is the *stream*.

A stream represents a specialised message passing channel between two processes. In the case of IO, one process is that performing the IO operation and the other is (a part of) the external logical *device* supporting that operation. A device is not necessarily the same as a single peripheral *mechanism* (*e.g.* a disc drive or a printer). There are several possible arrangements of IO devices with respect to mechanisms:

- A single mechanism device;
- A multi-access mechanism, shared between multiple devices;
- A multi-mechanism device;
- Multiple mechanisms shared between multiple devices.



### *Stream Operations*

For each Stream, there is an entry in the Unit Table which links the process initiating an IO operation with the stream. *Activation* of a stream is achieved by means of a System Call which enters the stream activation procedure via a PLT descriptor which points at the relevant Unit Table entry. The Unit Table entry also contains data identifying the process using the stream together with a value identifying the System Call Table to be used for notifying interrupts to the process, when required.

The stream activation procedure has two parameters:

- a unit-dependent value defining the operation to be performed;
- a pointer to a *Transfer Control Block*.

The Transfer Control Block (TCB) is a data structure containing additional parameters for the stream operation; for a data transfer to an IO device, this includes:

- a device command and optional qualifiers;
- a descriptor to the data area (& optional pre- & post-amble areas);
- a response area in which the success (or otherwise) of the transfer is recorded;
- an optional *chain* pointer to another TCB.

The stream activation procedure generally performs some checks to ensure that the caller has sufficient privilege to use the stream and then initiate some asynchronous activity on the stream. When the activity is completed, the termination data from the stream is used, by hardware, to generate an interrupt which is routed, via the Unit Table and a System Call Table, to the correct process.

### *Mechanism Sharing*

Many mechanisms are implicitly shared, in a time-sharing manner, by several processes. This sharing can take place at several levels:

1. at the mechanism level by having multi-access mechanism, with each stream Virtual Resource exclusively allocated to a single process;
2. at the device level, with the stream Virtual Resource shared dynamically between several processes;
3. at the device level, with the stream Virtual Resource allocated by the system, in turn, to each process for its exclusive use.

Disc drives are usually driven as single (or multiple) device mechanisms, each device shared dynamically at the device level. Line printers are usually driven as single device mechanisms shared statically at the device level.

### *Shared Devices*

In practice, most IO transfers involve shared devices. In particular, discs and communications devices are invariably shared between several VMs. It is important to note that most IO operations involve several layers of software, with only the higher layers visible to applications. Some layers *multiplex* several concurrent activities onto a single activity in the layer below. Thus what may appear to in a VM as an unshared Virtual Resource representing the endpoint of a single, independent connection to an external resource, may, in a lower layer, be supported by a shared Virtual Resource in common with similar connections in other VMs.

For example, two VMs accessing distinct files will contain individual virtual resources for each file but, at lower levels of the system, will share the virtual resource which performs an IO operation on a stream.

Similarly most networking methods involve several layers of protocol handling, with only the higher layers visible to applications. OpenVME support for networking therefore comprises both public resources (which handle the low level, shared communications devices) and local resources (which handle the higher level, unshared communications channels). The system provides a set of public buffers, allowing messages to be passed between the public and local resources.

# Chapter 5
# The Structure of OpenVME

## Introduction

OpenVME is designed and constructed according to a set of structuring principles. These ensure excellent modularity and encapsulation of system components as well as a high degree of protection between them. This chapter describes the structure of OpenVME system in terms of these principles, identifies the major components of the system and describes their function.

## Principles of OpenVME Structuring

### The Virtual Machine Structure

The concept of a Virtual Machine as a secure container of Virtual Resources and the consequent structuring of the system into disjoint sets of Virtual Resources local to a VM was described in detail in the previous chapter.

### The Layered Structure

#### *Levels of Abstraction*

The OpenVME system is structured into a number of *layers*, each a container for a subset of the Virtual Resources supported by the system. This containment structure is orthogonal to the VM structure. Higher layers provide increasing *levels of abstraction* from low level resources. In general, Virtual Resources in higher layers make use Virtual Resources in lower layers. An operational OpenVME system is a layered hierarchy of Virtual Resources with the lowest layer (the *kernel*) containing the Virtual Resources which correspond most closely to real hardware resources.

#### *The Role of Access Levels*

It is an important principle of the OpenVME architecture that use, by a process executing with a certain level of privilege, of more privileged resources should be strictly controlled. Access Levels are used to enforce this principle: the Virtual Resources of each layer are associated with a particular *Access Level*. Lower layers of the system and the Virtual Resources they contain are associated with more privileged (lower) Access Levels.

Access to a resource is protected by an *Access Key*. A process performing an operation on a resource in a layer must be executed at the Access Level with which the resource is associated. When a process attempts to operate on a resource, its Access Level is compared with the appropriate Access Key: if Access Level $\leq$ Access Key then the operation is permitted. The System Call mechanism is used to lower the Access Level of the process during execution of an operation on a more privileged resource. An Access Key is associated with the System Call itself, representing the Access Key of the resource.

Use of the System Call mechanism also ensures that the particular operations which may be performed on a privileged resource are restricted to those explicitly made available via System Calls. As an architectural principle, resources which support operations which are capable of being invoked from less privileged Access Levels must validate reference parameters (see previous chapter).

The association of layers with Access Levels has further ramifications. In particular, scheduling of processes within a VM and the notification of interrupt events are closely tied to Access Levels. Within a VM, lower (more privileged) Access Levels cannot be interrupted by higher (less privileged) Access Levels.

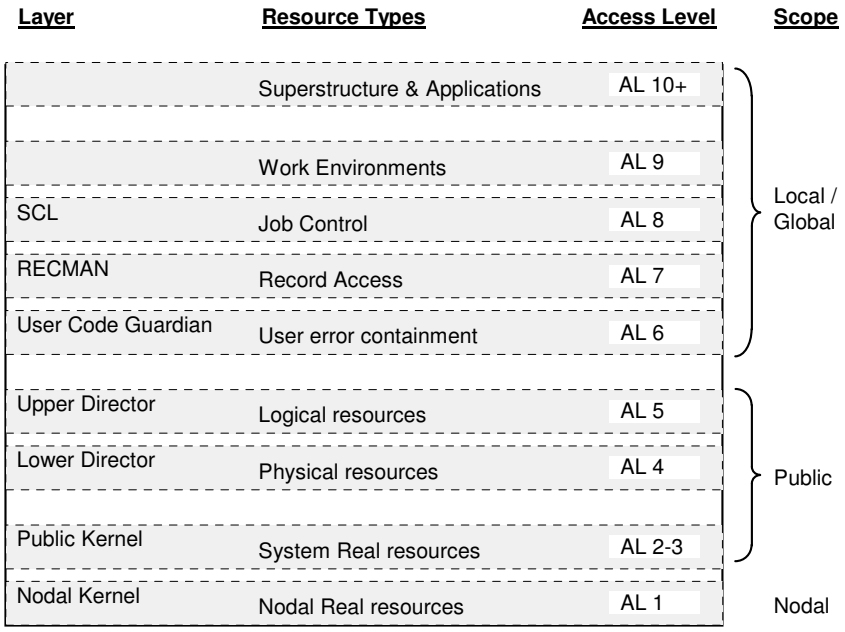
Although these two roles of Access Levels - for protection and for scheduling - may not immediately seem compatible, both uses are, in fact, a natural consequence of the structure of OpenVME reflecting different levels of abstraction. Most importantly the use of Access Levels means that, within a VM, operations on a Virtual Resource are executed *atomically* with respect to the execution of operations on the same or other resources in the same or higher layers. This greatly simplifies the design and implementation of Virtual Resources and is a foundation of the integrity and reliability characteristics of the OpenVME system.

### *Error Containment*

A further benefit of the layered structure of OpenVME is that it ensures that the effects of errors are prevented from propagating throughout the system. The layered protection mechanisms prevent corruption of more privileged data by less privileged software; in addition, the VM structure prevents corruption of data local to other VMs. Thus the effects of any error or failure is confined to a well defined and limited part of the system. In most cases a specific *Error Manager* is notified when an error occurs enabling it to take appropriate recovery action.

### *The Layers of OpenVME*

The major layers of OpenVME are illustrated in the diagram below. These layers and the functions supported within each layer are described in later sections of this chapter.




### *Use of Resources in Lower Layers*

Although the system is structured into many layers, a request by a process in one layer to perform an operation on a resource in a lower layer can, subject to privilege checks, invoke the operation directly, without passing through intermediate layers. Typically, during the declarative phase of resource usage, software in several layers may be involved in establishing a direct route from the requesting layer to the target resource; subsequently, during the imperative phase, this route allows operations on the target resource to be invoked very efficiently.

## The Subsystem Structure

An OpenVME *Subsystem* is a collection of procedures and data which support a set of related Virtual Resources. Each subsystem is encapsulated in a manner which hides internal representation and implementation, and the only means of accessing the Virtual Resources supported by a Subsystem is via a defined set of explicitly named procedural interfaces. The concept of a Subsystem is recursive in that any Subsystem may be constructed from a number of component Subsystems, although only a limited subset of the total set of interfaces may be made externally accessible.

In general, a Subsystem is responsible for the creation, operational support and eventual deletion of the Virtual Resources which it supports. The way in which the Virtual Resources supported by a Subsystem are represented is an internal concern of that Subsystem. Some Subsystems may represent the state of a resource as an entry in a (Public or Global) shared area, allowing the possibility of the resource itself being shared across VMs; others may represent the state of a resource as a Local data area within the VM using that resource (assuming it to be unshared).

Considering Virtual Resources as *objects*, the Subsystem supporting any particular resource can be considered as providing all the functionality associated with the *class* to which the object (resource) belongs.

### *Encapsulation & Modularity*

The Subsystems of OpenVME itself are usually constrained to be associated with a single Access Level within the layered structure of the system described above. Subsystems therefore provide an additional, modular, structuring of the functionality within a layer of OpenVME. Within a layer, the OpenVME development route itself constrains access to a Subsystem to be solely via named procedural interfaces, and from higher layers the Access Keys are used to enforce this restriction.

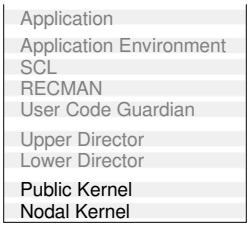
The encapsulation of functionality which is implied by the Subsystem structure has important consequences:

- it allows Subsystems to be developed (and evolved) independently, supporting a stable, defined set of external interfaces;
- it ensures that data associated exclusively with a Subsystem cannot be accessed except via the defined interfaces; this contributes significantly to the robustness of the OpenVME system.

## The Nodal Structure

The hardware of Series 39 systems comprises one or more *processing nodes*, each containing one or more processors and memory. The nodes are linked by an *Input-Output* network and an *Inter-Node* network. There is no physically common memory, though the Inter-Node network is used to provide a Virtual Shared Memory. This *Nodal* hardware architecture has certain implications for the structure of Kernel (which manages hardware resources). The nodal nature of the hardware architecture is only visible to Kernel which is responsible for ensuring that all higher layers are presented with a uniform view of all the resources of the system. The Nodal Architecture and its implications on Kernel structure are described in detail in Chapter 14 (Platforms).

## The OpenVME Kernel




The OpenVME Kernel comprises those subsystems which directly manage real hardware resources - in particular: memory, processing resources and routes to external resources such as Input-Output devices. A major function of Kernel is to abstract the complexity of the real hardware resources and their organisation into an interface which is independent of those aspects of hardware which may differ between implementations. Kernel presents a uniform procedural interface - the *Kernel Interface* - to all other system software.

Kernel subsystems operate on Nodal, Public and Local data. They execute at Access levels 1 (Nodal resources), 2 (locked Public resources) and 3 (other resources).

### Memory Management

#### *Virtual Store Manager (VSM)*

Virtual Store Manager uses real memory and hardware mechanisms to create virtual memory. Virtual memory may be:

- *Public* to all VMs in a node, or to all VMs in the entire system;
- *Local* to a single VM;
- *Global*, shared between several VMs.

Virtual memory is created in *Segments*, contiguous regions of virtual address space in which all locations have similar properties, such as Access Keys.

VSM is responsible for mapping virtual memory onto real memory. This mapping is in terms of sub-divisions of a segment, *pages*, to reduce fragmentation of real memory. VSM manipulates the page and segment tables which enable hardware to translate virtual addresses to real addresses.

The virtual memory requirements of the system can exceed the amount of real memory available. Pages of virtual memory which are not currently mapped onto real memory are stored on *Secondary Storage* discs. Hardware *Use* and *Written* bits associated with each page enable VSM to assess page usage. When there is contention for real memory VSM can *discard* pages, particularly those which have been rarely or never accessed, from real memory. Each VM has a guaranteed minimum quota of real memory and, when necessary, VSM may reduce its usage of real memory to the quota. If a page to be discarded has been updated since the last time it was written to secondary storage, the new contents of the page are written to secondary storage.

### *Virtual Machine Manager (VMM)*

Virtual Machine Manager is responsible for the basic management of Virtual Machines including their creation, deletion and low-level & high-level scheduling.

A Virtual Machine comprises virtual memory, defined by Segment Table Base Registers, and one or more processes, each defined by a Register Set (process state dump). Only one process within the VM may be active at a time. VMM maintains a Process Management Table, which has an entry for each process in the system, containing:

- the STBR for the process, hence defining the Virtual Machine in which it is executed;
- a Register Set for the process;
- meters and statistics for the process including elapsed process time and instruction counts;
- information concerning the scheduling state and suspension Access Level of the process.

VMM schedules processes eligible for execution according to a multi-level priority scheme. Time-slicing is used to ensure fair scheduling within a priority of compute-intensive processes.

VMM schedules VMs to real memory on the basis of their memory quota and priority and the amount of memory available. If the demand for real memory significantly exceeds that available then VMM may relocate entire VMs onto secondary storage to make space available. This action and its inverse, moving a VM into real memory from secondary storage, are termed *roll-out* and *roll-in* respectively.

VMM is responsible for scheduling VMs to nodes in a multi-node system. A VM may normally be executed on any node but it is possible to allocate a VM to a specific node. VMM attempts to balance the demands for memory and processor time between the nodes by relocating VMs from one node to another. This is achieved by rolling the VM out from one node and into another.

VMM handles various interrupts that may occur while a process in a VM is being executed. These include exceptions (such as Program Errors), Interval Timer and Instruction Counter interrupts (used to measure process time and instruction counts, and to enforce time-slicing) and System Call interrupts (see the description of SC, below).

VMM provides interfaces to allow communication between the instances of Nodal Kernel in a multi-node system. These operations are:

- the Broadcast Call which causes a nominated procedure to be executed on all nodes;
- the Directed Call which is executed within a nominated VM on another node.

### *System Call Handler (SC), part of VMM*

System Call Handler manages the mechanisms associated with System Calls. It creates and maintains System Call Tables. Entries in these tables are used to decode System Calls when they occur. By default, System Calls are decoded by hardware though under some circumstances SC itself is entered, via an interrupt, to perform the decode.

System Call Handler provides interfaces enabling software to create System Calls, establishing the necessary entries in the System Call Tables. Since a System Call is the primary means by which a process can change its Access Level to acquire greater privilege, SC enforces checks to ensure that the subsystem creating a System Call has sufficient privilege to do so.

SC also provides facilities for the creation of *sub-processes* within a VM. A sub-process has its own stack and process state. A sub-process may be created by a (sub-)process at any Access Level provided that the Access Level at which the sub-process itself is initially executed is at least that of the creator. When a sub-process is created, SC returns a System Call descriptor to the caller. Control can then be passed to the sub-process by invoking the corresponding System Call which causes an entry to SC to schedule it for execution.

### *Kernel Event Manager (KEVM)*

Events are the fundamental means of enabling processes, or different portions of one process, to be synchronised. Kernel Event Manager provides all the low-level functions associated with events, including co-operating with VMM when a process is to be suspended waiting for an event or rescheduled for notification of an event. Public, global and local events are supported.

Kernel Event Manager provides facilities to:

- create and delete an event;
- connect to an existing event;
- wait for an event;
- cause an event;
- receive notification of an event occurrence.

### *Kernel Semaphore Handler (KSH)*

Semaphores provide a means of co-ordination between processes to ensure that operations on shared resources can be performed safely, without interference

from another process attempting to operate on the same resource concurrently. When one process attempting to reserve a semaphore discovers that it has already been reserved by another process a *clash* occurs.

Kernel Semaphore Handler provides a *clash handler* which is entered by the process which detects the clash condition, registering its interest in reserving the semaphore. KSH allocates an event on which the process waits at the Access Level of the caller. If the process holding the semaphore has a lower priority it is given the priority of the waiting process (a technique known as *priority inversion*). When the semaphore is subsequently released, the process which had reserved it detects that another process is waiting to reserve it and enters KSH. KSH then causes the event, notifying the waiting process that it has now reserved the semaphore. Where several processes are waiting for the same semaphore, that with the highest priority is notified.

## Timer Management

### *Timer Manager (TIM)*

Timer Manager provides facilities for a VM to request notification at a specified real time or periodically. The operation of Timer Manager is based on a type of Virtual Resource known as a *Timer Channel*.

Operations are provided to create and delete Timer Channels. When a Timer Channel is created it is associated with a specific event by which all timer notifications are communicated. Timer Manager provides facilities for requesting notification of the specified event either at an absolute time or after a specified time interval; in the latter case, the request may specify that the event should be caused once only or periodically, each time the time interval elapses. When a timer notification is requested, an event message is also specified. Only one timer notification may be established for a timer channel at a time. Timer Manager provides a means of *cancelling* any extant timer channel notification thus allowing a new notification to be established.

Timer Manager also provides facilities for handling *process time* - the time for which a process has been being executed. Operations are provided to read the current process time, and to request a single or periodic notification after a specified amount of process time has been used. These facilities can be used for various performance monitoring purposes.

## Hardware Resource Management

### *Kernel Reconfiguration Manager (KRM)*

Kernel Reconfiguration Manager maintains details of all permanent hardware units and the routes linking them, including Nodes, Controllers, IO devices, inter-node and IO networks. During initialisation, when units are configured in or off,

or when routes are changed, KRM orchestrates other subsystems to take the necessary actions - *e.g.* loading a microprogram, evacuating all VMs from a node or changing routing tables.

Various subsystems handle different aspects of unit configuration management, including:

- OCP Manager (OCPM)
- Multi-Node Manager (MNM)
- Public Write Manager (PWM)
- Public Write LAN Manager (PLAN)
- Node Support Computer Manager (NSCM)

### *Peripheral Manager (PERM)*

Peripheral Manager handles the low-level driving of IO devices. It has several associated subsystems which operate at two levels:

- Controller Manager (CM) and Device Reconfiguration Manager (DRM) which manage the peripheral controllers themselves and the streams which they support;
- Device Access Managers (RDAM, DDAM) which manage access to the devices supported via streams. The DAMs control the initiation and termination of IO requests as well as unsolicited reports of status changes. Different DAMs support serial and random-access devices as well as low-level access to devices for engineering or other special purposes.

### *Stream Manager (STM)*

Stream Manager is responsible for mapping IO requests onto the underlying IO Network. A logical IO request, known as a *stream transfer*, comprises:

- a command and initialisation information;
- optional data, which may flow to or from the IO device;
- status and termination information.

The IO Network is a set of packet-switched LANs. It uses a full transport service carrying stream transfer requests between the processing nodes and the peripheral controllers. STM handles the processing node end of this transport service driving the LANs via the ETH, MLAN & ELAN communications subsystems (see below).

### *Communications*

The Communications Subsystems of Kernel provide the low-level protocol handling capabilities of the system. The most important are:

- Network Connection Manager (NCM)
- Network File Handler (NFH)
- Kernel Comms (KC)

- Communications Processor Manager (CPM)
- Transport Service Manager (TSM)
- Ethernet LAN Manager (ELAN)
- Macrolan LAN Manager (MLAN)
- ECMA Transport Handler (ETH)
- X25 Handler (X25H) etc..

They are described in detail in Chapter 9.

## Miscellaneous Kernel Subsystems

### *Meters and Statistics (MCM)*

MCM provides interfaces to VM and system monitoring facilities operating in Kernel and above.

### *Kernel Error Manager (KEM)*

Kernel Error Manager handles exception conditions which occur in Kernel. Some subsystems have recovery mechanisms and if an error occurs in one of these, KEM enters the appropriate recovery procedure. Otherwise a system dump is taken and the system is restarted.

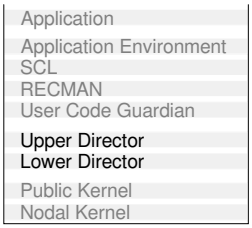
### *Supervisor Loader (SVL)*

Supervisor Loader loads Director and many above-director subsystems from system filestore. Its operation is controlled by a *steering file* which indicates which modules are to be loaded, which interfaces are to be made available to higher level software (and the privilege checks to be applied to use of those interfaces) and various initialisation options.

Amendments to the software, in the form of *repairs*, may be included in the steering file, allowing software corrections to be made without re-issuing the whole system.

When the System Load is complete, Supervisor Loader passes to the VME loader information about the system interfaces provided by both Kernel and Director which are identified in the steering file. These interfaces become part of the loading environment for subsequent load operations and are, in general, made accessible to higher level software via System Calls.

## The OpenVME Director




The OpenVME Director comprises a set of subsystems which, in terms of the layered structure, are immediately above Kernel. It controls the establishment of Virtual Machines and supervises the allocation of resources between different VMs, as well as providing loading facilities, catalogue management, physical file management, operator communications and journal facilities. The procedural interface presented by Director is known as the *Director Interface*.

Director subsystems are the highest layer subsystems which operate directly on Public data. Subsystems above Director generally operate either on Local data or, if sharing data between VMs, on Global data. However, a facility is provided to load the code and read-only data of a subsystem into public segments to obviate the need to load them separately into local segments in each VM; this mechanism is exploited by commonly used above-director subsystems. Many "Director Subsystems" have components above Director. In general, the component within Director operates on Public resources whilst the above-Director component provides the Local functionality and maintains the Local context for the subsystem.

Director is divided into two layers: *Lower Director* and *Upper Director*.

### *Lower Director*

Lower Director is mainly concerned with the management of physical resources accessed via Kernel. The resources are handled by subsystems known as *Physical File Managers*. Using knowledge of the physical characteristics of the resources they provide basic facilities for controlling the resources via *files*. The two main groups of resources are:

- magnetic tape and disc physical files;
- communications and slow devices (*e.g.* Line Printers).

Lower Director is executed at Access Level 4.

### *Upper Director*

Upper Director is concerned with the management of logical resources which are declaratively referenced by name. The main functions of Upper Director are:

- Selection of Objects referenced by name via the Catalogue and associated contexts, including enforcement of security;
- Creation of Virtual Resources, within Director subsystems, corresponding to catalogued objects such as data files;
- Creation of Virtual Resources by loading of OMF modules referenced by name, and resolution of named references between modules;
- Scheduling of resources to VMs and block-structured resource control;
- Co-ordination of Virtual Machine initialisation in Director and Kernel.

Upper Director is executed at Access level 5.

## Physical File Management

### *Magnetic Physical File Manager (MAMPHY)*

The Magnetic Physical File Manager uses knowledge of physical characteristics of a device to provide the basic facilities for controlling magnetic tape and disc physical files.

MAMPHY supports transfers directly between user level buffers and the physical device, eliminating unnecessary movements of data via intermediate buffers. In each VM MAMPHY supports multiple *transfer slots* each of which supports a single transfer request. By using several transfer slots, a VM may arrange for several transfers to be in progress concurrently.

MAMPHY provides to allow multiple VMs to share update access to a physical file. Data consistency can be assured by using MAMPHY's block-level locking facilities or alternative, application-specific mechanisms.

MAMPHY supports *plexing* of disc partitions allowing several physical copies of the partition, each known as a *plex*, to be maintained in step with each other. Facilities are provided to add and remove plexes dynamically, with newly added plexes being automatically brought up to date. Plexing has several benefits:

- It provides resilience against failure of a single magnetic device;
- It improves read latency by allowing read accesses to be performed from any of the plexes;
- An active plex can be removed from active use and is then a precise copy of the plexed file at the instant at which it was removed; this method can be used to take "instant archives".

### *Communications and Slow Device Manager (COSMAN)*

COSMAN is a collective term covering a group of subsystems which provide physical file management facilities for locally connected slow devices (such as line-printers) and communications links. COSMAN is described in more detail in Chapter 9.

## Catalogue and Security

### *Catalogue Handler (CATHAN)*

The Catalogue Handler provides powerful data management capabilities for storage and retrieval of information in the Catalogue.

### *Privacy Handler (PV)*

Privacy Handler provides secure interfaces to access information in the Catalogue, ensuring that all accesses are permitted by the Discretionary Security Policy. It provides facilities for selecting catalogued objects absolutely, using *Full Hierarchic Names* or, relative to other objects, using *Selectors*. It manages *Currencies*, efficient references to objects, once they have been initially selected. Privacy Handler also manages *contexts* for most types of catalogued object which, within a VM, define default starting points for relative selections of such objects. PV has above-Director components.

### *Security Handler (SCHH)*

If the OpenVME High Security Option is installed, Security Handler provides the basis for maintaining and acting upon dynamic security information. Additional information is stored in the catalogue for each object defining its security attributes. When such an object is accessed, SCHH checks that the access is permitted by the Mandatory Security Policy and adjusts the dynamic security classification of the VM according to the nature of the access to the object.

### *Location Manager (LOCM)*

OpenVME supports the concept of *Locations*. Locations are used to group resources which have some association - most usually they are geographically co-located. A location is a catalogued object associated with which is a default *language*, a *civil time displacement* (relative to the system clock), and operational subsetting data. Any hardware unit, SAP, or service may be associated with a specific location and each job in the system has a default location. LOCM controls the use of locations and their associated data in the catalogue.

### *Other Catalogued Object Managers (UM, NM, UOM)*

Various minor subsystems - User Manager (UM), Node Manager (NM) and User Object Manager (UOM) - provide interfaces to specific types of catalogued object.

### *Export / Import (EXIM)*

The Export / Import subsystem provides facilities to enable the migration of catalogued objects from one OpenVME system to another.

## Resource Scheduling and Management

### *Resource Scheduler (RS)*

Resource Scheduler is the subsystem responsible for co-ordinating the allocation of real resources amongst VMs in the system. Examples of resources co-ordinated by Resource Scheduler include a hardware peripheral device, a magnetic tape or disc volume, a communications connection or a printer with specific media. Resource Scheduler interacts with the subsystems directly responsible for managing specific resources to acquire them, including Hardware Manager, Network Controller, Volume Manager, Media Handler, etc..

Resources are requested by type, specifying in addition any specific attributes such as media, location, volume name. This allows Resource Scheduler to satisfy the request with any resource which conforms to the specified requirements.

In order to resolve potential conflicts of resource requirements between VMs, resources are allocated in sets. When a VM requires resources it registers its requirements with Resource Scheduler. Eventually it requests Resource Scheduler to perform a resource *reschedule* on its behalf. Resource Scheduler then attempts to acquire all requested resources and the reschedule is only completed when they are all available. This technique precludes the possibility of, for example, two VMs with overlapping resource requirements each acquiring only some of the required resources and being unable to proceed because the remaining resources are held by the other VM.

Resources are allocated within logical *Resource Allocation Blocks*. A VM may request a set of resources before entry to a block; when it then *BEGINs* a block, a resource reschedule is performed. The VM then uses the resources and, when it no longer requires them, it *ENDs* the block. This causes the resources allocated to the VM for that block to be de-allocated and made available for use by other VMs.

### *Tag Manager (UTM)*

Tag Manager provides for recording the usage of resources by a VM with each Resource Allocation Block. As each subsystem allocates a resource within a VM it notifies Tag Manager, identifying the type and identity of the resource. When a Resource Allocation Block is ended, Tag Manager notifies those subsystems which have recorded the allocation of a resource with the block so that the resource can be de-allocated. This method ensures that all resources are correctly de-allocated at the end of a Resource Allocation Block and that only those subsystems which had allocated resources need be invoked to perform end-of-block housekeeping.

## Hardware Management

### *Hardware Manager (HM)*

Hardware Manager provides logical interfaces for the management of hardware resources including central hardware units, local interconnection, peripheral controllers and devices. HM maintains in the catalogue a definition of the permanent hardware configuration and its structure. The configuration is defined in terms of a hierarchy of named *hardware units* in which each unit is either *owned* by another unit (and has an address relative to that unit), or is part of the top-level *installation*. Hardware Manager also manages hardware *unit descriptions* for various classes of hardware unit, which define the generic properties and attributes of each class of unit

Hardware Manager also provides facilities for the dynamic declaration of hardware units. This capability is used particularly by communications subsystems such as Network Controller to create the unit hierarchies corresponding to dynamically established communications links.

Hardware Manager handles unexpected changes in state of units, notifying the subsystem responsible for managing that unit. Error conditions are notified to the *Error Logging VM (ELVM)* which records periodic statistics as well as diagnostic dumps and error logs. In some cases, the operator is informed, for example, if a line-printer runs out of paper or a hardware unit failure occurs. The operator is provided with facilities to alter the configured state of a unit so that it can be made available or unavailable for use.

## Module Loading

### *VME Module Loader (BL)*

The VME Module Loader is the subsystem responsible for loading OMF modules into virtual memory. It is invoked either by a direct request to load a module (by reference to one of its module keys) or as a result of a dynamic reference to an external named object during execution of a previously loaded module.

Loader maintains data recording the names of loaded modules and the names of visible OMF objects within the modules; the names of externally accessible Director and Kernel interfaces are also recorded (during system load). Loader uses this data to satisfy dynamic references to already loaded objects, which can include resources to be shared between VMs. Where required, Loader establishes System Calls to enable a procedure executed at one Access Level to enter a procedure executed at a more privileged Access Level.

Loader also provides facilities for dynamically creating temporary OMF objects and areas which are subsequently treated as though they had been loaded from OMF modules.

## Logical File management

### *OpenVME Logical Filestore Organisation*

Filestore is regarded, logically, as an organised set of files, each identified via the Catalogue. Files may be individually catalogued. Each file has a *file description* describing the properties of a file including information about the organisation of data within the files. A *Library* is a collection of similar files with properties inherited from the Library description in the catalogue. Files and Libraries may be grouped together under a *Filegroup* node in the catalogue.

A file is normally viewed as a logical collection of *records*. An application manipulates records via a Virtual Resource known as *Record Access Mechanism* (RAM) which hides all details about how the file is stored or handled as a physical entity. The level of abstraction provided by RAMs has important consequences allowing, for example, an application to access a serial file independently of whether it is based on disc, magnetic tape or a communications connection.

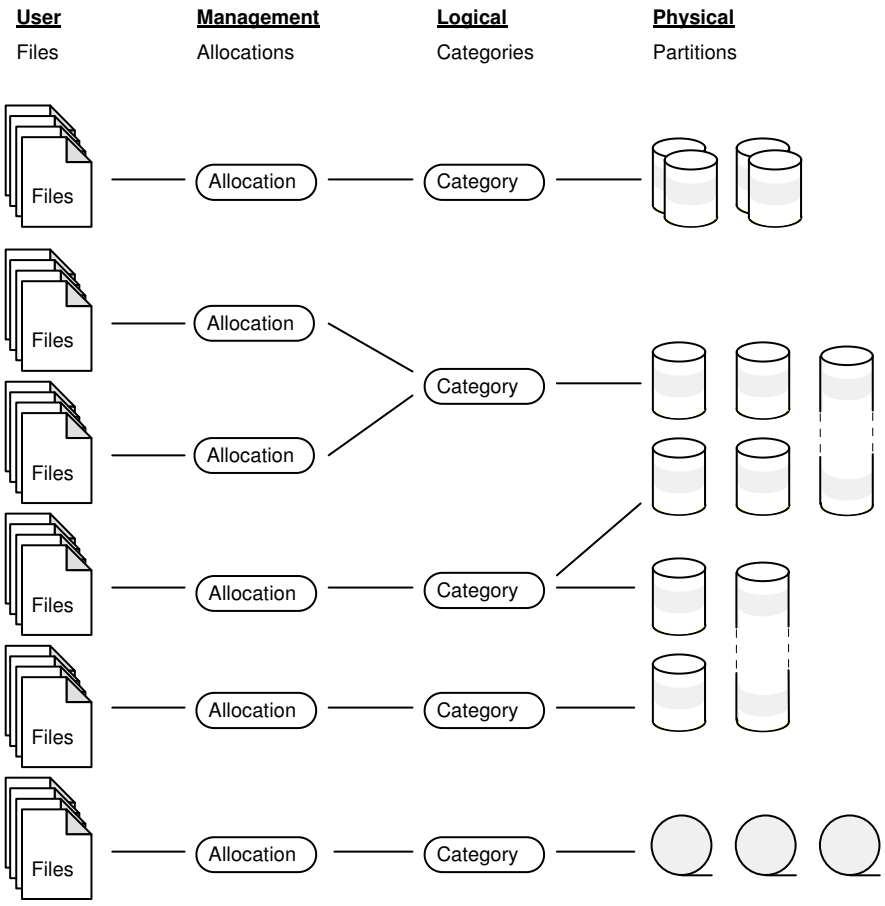
Several types of file are provided, to carry out different functions - for example:

- Serial and Random data files;
- Program files holding OMF (program) modules;
- Journals recording information about individual jobs or system behaviour;

All filestore is associated with some *category* which defines the manner in which the physical filestore is managed including, for example, block size, allocation

unit size, backup and multi-plexing requirements. Each category may have one or more *partitions*, each a discrete area of a disc or magnetic tape volume. Filestore is allocated to users by means of *allocations*; each allocation specifies an amount of filestore within a single specified category.

Each file is placed in a single allocation and comprises one or more *file sections*, each comprising one or more *extents*. Each extent is a set of one or more contiguous *allocation units* from the same partition within a category.




### *Volume Manager (VOLM)*

Volume Manager is responsible for the management of Volumes and Volume Descriptions. It maintains dynamic information about Categories and Partitions and how they are mapped to volumes.

### *File Controller (FC)*

File Controller manages the catalogued information about the physical placement and format of real files, primitive files and file sections. It provides the declarative interface to files, allowing them to be created (based on file descriptions), saved, deleted, assigned, released etc.. FC has above-Director components.

### *Library Controller (LC)*

Library Controller manages libraries and the libraryfiles they contain. It provides facilities for the description, creation and deletion of libraries. It also provides the declarative interface to libraryfiles themselves, allowing them to be created (based on the library description), saved, deleted, assigned, released etc. Library Controller is invoked, for example, by File Controller when it encounters a library when attempting to select a file. LC has above-Director components.

### *Embedded Filestore Manager (EFSM, AL7)*

Library Controller provides a mechanism whereby an externally implemented filestore can be made to appear as an extension of the VME filestore structure. This is a general mechanism allowing new subsystems to be introduced to access additional external filestores. EFSM is one such subsystem, enabling the filestore of a VME-X service to be accessed by VME applications. EFSM establishes a connection to the VME-X Service VM (subject to the user being allowed to access the VME-X service) and uses this connection to exchange file access protocol with the Service VM. EFSM operates in conjunction with the above-Director components of LC at Access Level 7.

## Task, Service & Job management

### *Jobs*

A *job* is a self contained unit of work. An interactive user session, a database service run, a request to list or transfer a file, and a batch program run are all examples of jobs.

### *Tasks*

A *Task* is a catalogued object which provides a specific autonomous function. An active instance of a task is established by creating an appropriate collection of Virtual Resources, defined in the catalogue, within a VM. A task may be *unique* in which case only a single active instance of the task can exist; or *generic* in which case multiple active instances can exist. Generic tasks are used as *support tasks* for common work environments such as MAC, BATCH, Spooler etc. Active support tasks are organised into *task pools*, sets of similar tasks managed by a *scheduler task*.

External communication with a task is achieved by sending *task messages* via *Task Channels*. Task messages may be sent even when the destination task is not currently active, in which case an active instance of the task is created automatically.

### *Services*

A Catalogued *Service* is an object in the Catalogue describing some functional capability independently of the means by which it is provided. All catalogued services are named and referenced in a standard manner although the functionality of each is used via an interface whose nature is dependent on the particular service provided. Examples of catalogued services are:

- a MAC, VME-X, BATCH or TP work management service;
- an application service provided by an application server;
- a communications service;
- a spooling or file transfer service;
- a database management service;
- a user session management service.

Each catalogued service has a fixed *management name* and optionally several *service selectors*, additional names referring to the service which may be changed. Various standard attributes apply to all services, whilst others are specific to a particular service or to a class of services sharing a common *service description*. Amongst the common attributes are:

- the service selectors by which the service can be named;
- the mechanism by which use of the service is requested;
- the security mechanisms controlling permission to use the service;
- the communications routes through which the service can be accessed;
- the task(s) responsible for providing the service, including scheduler & support tasks.

### *Task Controller (TC)*

Task Controller (TC) is the subsystem which manages all aspects of tasks including cataloguing, starting and stopping tasks, task pools and task messages. TC also manages *operator tags*, unique identifiers which are assigned to jobs as they are requested. TC provides facilities to allow an operator tag to be associated with the task currently responsible for the job, enabling other tasks to identify the job and "communicate" with it.

### *Service Manager (SVM)*

Service Manager is the subsystem which manages all aspects of services and service descriptions including cataloguing, management, starting, stopping and suspension of services. SVM has above-Director components.

### *Virtual Machine Initialiser (VMI) & VM Description Handler*

VDH manages VM description data. VM descriptions are held in virtual memory rather than the catalogue because they need to be accessed by VMI prior to connection to the catalogue. A VM description comprises two lists of procedures to be entered at each Access Level in order to initialise and collapse the VM, respectively.

Virtual Machine Initialiser initialises the standard components of every VM. This involves calling the initialisation procedures for each Access Level as defined by the VM description. VMI provides defaults for these procedures which call the initialisation procedures for each Kernel or Director subsystem which operates at Access Levels 3 to 5. A subsystem initialisation procedure creates and initialises a copy of its in-process data within the VM and amends any public data accordingly. In some cases subsystem initialisation is dependent on job-related information which is supplied by subsystems operating at Access Level 9; in these cases the subsystem is initialised by a subsequent inward call from Access Level 9 rather than by VMI.

### *Task Creation & Initialisation*

Almost all VMs are created at the instigation of Task Controller according to a specified task description defined in the Catalogue node for the Task. The initial creation of a VM is performed by Kernel acting at the request of Task Controller. Subsequently a set of standard Kernel and Director Virtual Resources are created in the VM under the control of Virtual Machine Manager (VMM) and Virtual Machine Initialiser (VMI) subsystems respectively.

Once Kernel and Director initialisation are complete, Task Controller effects the higher level initialisation of the VM by causing *Task Related Initialisation Procedures* (TRIPs), defined in the task description, to be executed within the VM. These TRIPs create and initialise the Virtual Resources supported by Above-Director subsystems. Thus the set of Virtual Resources created in a VM depends on the TRIPs executed and an application environment can therefore be tailored to particular requirements by specifying suitable TRIPs to create that environment.

## Operator Communications and Journals

### *Operator Communications Handler (ECFH, OFM and PT))*

OpenVME makes provision for *operators* to monitor and control the state of the system as a whole and of particular aspects of the system - *e.g.* the resources at a particular location. *Operator Communication Files* (OCFs) are virtual store files which can be used by subsystems within Director and above to record the system state. *Prompts* are messages to an operator notifying some system state or condition which requires operator action; when appropriate, the operator can respond to the prompt informing the system what action has been taken.

An operator interacts with the system from within an interactive *OPER* job which is associated with particular properties such as location; only that subset of the system corresponding to those properties is visible from the OPER job. The operator can display OCFs, journals and prompts on the terminal and can perform system management commands.

A *journal* is a file which records events occurring within the system, an application server or a job. System journals record job initiation & termination, system exceptions, security audit and accounting information. A job journal records progress of an individual job or task. Many subsystems can be requested to write information about their usage or diagnostic information to journals.

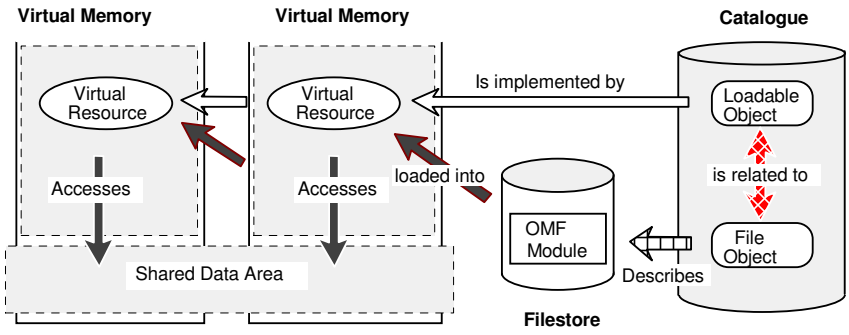
ECFH provides facilities for journal creation, opening, closing, writing, reading etc., and controls the mapping of messages according to their type into appropriate journals. It also provides support for the creation, updating and reading of OCFs and for the handling of prompts.

Journal messages are stored in parametric form and translated, when required, into natural language via *Message Libraries*. This translation is handled by the Operator File Manager (OFM) and Parametric Translation (PT) subsystems.

## Shared Memory & Message Passing

### *Shared Data Areas*

A Virtual Resource can be shared between several VMs by loading the data area(s) associated with the resource into Public or Global segments commonly accessible to the VMs. This is achieved by loading the same OMF module in each VM to create the VM's view of the shared resource. In the OMF module one or more OMF objects represent shared data areas, with properties indicating that they are to be loaded into shared segments. The shared segments are created when the module is first loaded. A subsequent load operation in another VM causes the segments to be made accessible in that VM and loader's tables to be updated to record the VM's local view of the loaded module.



### *Address-Space and Region Manager (ASR)*

ASR provides the basic mechanisms required to support X/Open Inter-Process Communications facilities. These are:

- Shared Memory Areas: regions of contiguous shared memory which can be mapped into the address space of one or more VMs;
- Message Queues: objects which allow messages to be passed between processes in one or more VMs, notifying interested VMs when messages arrive;
- Semaphores: objects which allow processes in one or more VMs to synchronise with each other. Semaphores can be grouped into *Semaphore Sets* and a process can perform an *operation sequence* on the semaphores in a set, the sequence only terminating when the final operation is complete.

The shared memory facilities provided by ASR are lower level than those provided by module loader. ASR facilities are based on the explicit creation of address spaces and mapping of regions whereas the sharing of data areas loaded from OMF modules is an entirely implicit mechanism.

ASR supports a public object type known as a *Shared Area Context* (SAC). A SAC is owned by the VM which created it; other VMs can gain access to the objects within a SAC by establishing an association with the SAC using an interface provided for this purpose by QISH.

A SAC may contain public objects of three types: *Shared Areas, Message Queues* and S*emaphore Sets*. A shared area can only be created by the VM which owns the SAC but message queues and semaphore sets can be created by any VM gaining an association with the SAC. Within each VM associated with a SAC, local objects are created corresponding to each public object to which the VM requests access, as shown in the table below.

| Public Object | Local Object |
|---|---|
| Shared Address Context | Group (objects owned by a VM) |
| | Subgroup (objects owned by a process) |
| | Address Space |
| Shared Area | Region |
| Message Queue | Message Queue Connection |
| Semaphore Set | Semaphore Set Connection |

ASR provides the interfaces required to create, modify, delete and establish local connections to Address Spaces and Regions, Message Queues and Semaphore Sets; it also provides interfaces for performing operations on these objects. The interfaces correspond closely to those defined by X/Open for IPC, including provision for permission checking; however, operations which are defined to suspend synchronously in the X/Open definition instead return control to the caller, with a suitable result code.

### *Quick Intra-System Handler (QISH) and Quick Message Handler (QMH)*

QISH and QMH are two related subsystems providing fast message-passing between VMs using shared memory and events. QMH provides the basic message passing facilities between VMs whilst QISH makes individual connections available to higher level subsystems and applications. QISH provides a mechanism which allows connections to be established and used in the same way as general communications connections; this is based on intercepting relevant RS and COSMAN interfaces.

QISH has a minor but important subsidiary role in providing a secure means by which a VM can perform operations on objects supported by ASR. Above-Director subsystems and applications cannot do so directly but QISH provides a secure means for a VM to gain such a capability and create local IPC objects, allowing it to call ASR directly to operate on those objects.

## Transaction Management

The OpenVME transaction management facilities are described in detail in Chapter 7. This section summarises the role of the Director subsystems supporting transaction management. These facilities only use global virtual memory for areas shared between VMs..

### *Commitment Co-ordinator (COCO)*

*Commitment Co-ordinator* (COCO) links the activities of a set of Resource Managers (RMs) within a VM so that they can collectively perform *transactions* which have the required transactional properties. In terms of the X/Open architecture, it performs the role of a *Transaction Manager*.

### *Application Dialogue Handler (ADH)*

*Application Dialogue Handler* (ADH) provides an efficient method of passing messages between VMs within a single OpenVME system. ADH uses supporting interfaces provided by QMH.

### *Work-In-Progress Store Manager (WSM)*

WIP Store Manager handles the data used to roll-back or recover transactions, sessions, databases etc. In effect, it handles the non-volatile data except for that managed directly by the Resource Managers themselves.

## Communications support

### *Network Controller, Upper Director (NCUD)*

NCUD operates in conjunction with the Network Controller Task (NETCON, described in Chapter 9). Its main function is to support declarative operations associated with network connections. These operations include:

- the initiation of outward network connection requests by user VMs;
- the initiation of network disconnection requests by user VMs;
- the notification of inward network connections to an appropriate listening VM;
- the notification of network disconnections to the VM currently using the connection;
- the buffering of information associated with connection & disconnection handling.

NCUD provides the interfaces by which Resource Scheduler handles network connection resources, passing information to or from NETCON as required, via task messages.

## System Loading, Initialisation & Checkpointing

System Load commences with a hardware-dependent mechanism which places a primitive Kernel in memory and enters a pre-created VM. Kernel then initialises itself, establishes the hardware configuration. It then enters Supervisor Loader which loads Director as indicated by the steering file. A new VM, the Director Initialisation VM (DIVM), is then initialised by VMI, establishing Director subsystems within the VM. The System Task Watchdog VM is started and it establishes VMs containing the required set of System Tasks and schedulers; these in turn create VMs for any required support tasks.

A *Full Load* can be a very lengthy operation and therefore a *checkpoint* mechanism is provided which allows a system which has been almost completely loaded and initialised, and which has been specialised for the requirements of a particular installation, to be saved on disc. A subsequent *Restart Load* can load the system from the saved checkpoint, considerably reducing the time taken to establish an operational system. The checkpoint can only contain information which does not vary between system sessions. Therefore if a file was in use at the time of the checkpoint and its contents are later changed, or the configuration is changed, the checkpoint becomes invalid.

### *Checkpoint Restart Controller (CRC)*

Checkpoint Restart Controller controls the functions involved in creating a *restart site* on disc and restarting the system from it.

## Director Meters & Statistics

### *Standard Monitoring Facility (SMF)*

The *Standard Monitoring Facility* (SMF) provides standardised support for collection and dissemination of monitoring information from applications. SMF has components at Kernel, Director and User Access Levels. It is based on the Managed Object model and supports named *monitoring objects* which may be public, local or global and which contain:

- *counter meters* (containing values which increase in discrete steps);
- *gauge meters* (containing values that may increase or decrease);
- *compound meters* (containing arbitrarily typed values);
- further objects (allowing the creation of a hierarchy of objects).

Counters and gauges may have *upper* and *lower thresholds* specified.

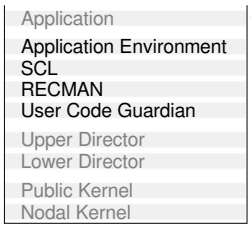
Applications using SMF are grouped into two classes: *Providers* and *Consumers*. Providers supply information to update monitoring objects and may cause *monitoring events* under certain circumstances including changes in status, alerts and the crossing of a gauge or counter threshold. Consumers can read meters and elect to be informed of the occurrence of specified monitoring events. RMSV and VCMS are examples of consumer applications whilst the subsystems providing transaction management (COCO, WSM) are examples of producers.

## Director Error Management

### *Director Error Manager (DEM)*

Director Error Manager handles exception conditions occurring in Director. Exceptions detected by Director Subsystems are logged to the System Journal as *Software Incidents*. Exceptions detected by hardware or Kernel are notified to Director as contingencies. Some Director subsystems have contingency handlers which are notified in these circumstances so that recovery from the contingency may be attempted. Otherwise a diagnostic dump is performed and system crash is usually forced as the integrity of public Director data is no longer assured.

## Above Director Software




Above-Director software is divided into 4 layers:

- User Code Guardian, at Access Level 6
- Record Access Manager (RECMAN), at Access Level 7
- File management utilities, at Access Level 7 and above
- The System Control Language environment (SCLS), at Access Level 8
- Various Job and Application Environment subsystems, at Access Level 9

### User Code Guardian

*User Code Guardian (UCG)*

User Code Guardian has three main functions:

- Contingency handling
- VM dumping
- Budgeting & Accounting (BUC, ACC)

A *contingency* is an unanticipated exception condition. Contingencies occurring at Access Level 6 or above are notified, in the first instance, to UCG. Software executing at Access Level 6 or above can elect to handle certain contingencies and UCG provides interfaces to allow an event to be nominated to be caused by UCG when such a contingency occurs; software handling the event may then attempt recovery action. Otherwise, UCG has a range of default actions depending on the nature of the contingency. If no recovery is possible, it performs a diagnostic dump of the VM to a journal and, depending on the work environment, may abandon the current application, job or even the whole VM.

### Record Access

*Record Access Manager (RECMAN)*

Record Access Manager is a collection of subsystems which provide record-level access to file data, independently of the nature of the underlying physical file or medium. Thus an application may read a sequence of serially accessed records from a file held on disc, magnetic tape, a communications connection or virtual store without needing to be aware of any differences.

RECMAN subsystems can be divided into two main groups:

- those implementing record access: the *Record Access Mechanisms* (RAMs);
- those establishing and, subsequently, providing common file services to the RAMs.

A RAM is a Virtual Resource created by RECMAN for accessing a file. The operations supported by a RAM may include: selecting a record by absolute or relative position, or by key; reading, modifying, locking, unlocking, adding or destroying a record; maintaining and sorting indexes; actions specific to interactive devices such as terminals.

Record Access Mechanisms are provided for the following file organisations:

- Serial (RC), providing record level access to serial, direct serial and ordered serial file organisations;
- Indexed Sequential (IS), providing record access level access to files of indexed sequential, hashed random and COBOL relative file organisations;
- Slow Device (RF), providing record level access to interactive devices, control programs and bulk input-output devices;
- Block Access (BA), providing access to whole physical blocks on magnetic media.
- Alternate Key (AK), allowing an Indexed Sequential file to be accessed by one or more Alternative Keys as well as the primary key;
- Record Transformation (TR), allowing user defined record transformations.

Supporting services are provided by the following subsystems:

- File Services (FL), which is responsible for reading file descriptions form the Catalogue, and loading and establishing the required RAM.
- Resource Manager (RM), which allocates and releases resources required by the RAMs. This includes instances of RAMs (so that they can be serially re-used) and memory for buffers, file checkpointing and recovery information.
- Record Access Tag Manager (RT), which tags resources associated with a particular RAM so that they can released when the RAM is de-allocated or at the end of the SCL Resource Allocation block in which they were requested.
- Record Access Error Manager (ER), which deals with unrecoverable errors detected by the RAMs.
- File Recovery (RR), which provides file checkpointing and recovery facilities.
- Section Manager (SM), which provides function common for all RAMs to attach and detach file sections as required.
- CAFS Generator (CG), which generates code for the CAFS search accelerator, which can be invoked via the Indexed Sequential RAM or IDMSX.

- Direct CAFS Interface (DCI), which provides high level access to CAFS capabilities, allowing retrieval of records from CAFS-based files according to specified search criteria. DCI Interfaces can be called directly from SCL.
- FTAM Virtual Filestore (VFS) supports the Virtual Filestore defined for FTAM. It provides the basic operations of the Virtual Filestore, and maps them onto corresponding operations on VME filestore.

## File Management Utilities

### *File Description Handler (FD)*

File Description Handler provides facilities for creating and modifying file descriptions. Descriptions are generally based on standard descriptions but can have existing attributes modified and new attributes added according to the supplied attributes. Each description is stored in the catalogue and can be used as the basis for creating further descriptions.

### *Resource Interface Manager (RI)*

Resource Interface Manager provides a user interface to Volume Manager, File Controller and Library Controller. e.g. Volume, Allocation, Category, Partition, File & Library interfaces and commands.

## System & Job Control

Interactive control of an operational system, individual jobs and application servers is performed using statements expressed in the OpenVME *System Control Language* (SCL). SCL statements allow the user to call interfaces made available by applications and by *command* procedures. Commands are provided for various classes of usage, including:

- System management;
- Job management;
- File creation, editing, management and deletion;
- Executing applications, including application development tools.

The set of standard commands supplied with the system are termed the *General System Interface* (GSI) Commands.

### *The System Control Language (SCL)*

SCL combines the functions of a job control language and a programming language. It is a procedural language designed for both interactive and batch usage. SCL has facilities for calling system-supplied and user-written procedures, including the interfaces used to manage the system. SCL procedures are callable in the same way as any other procedural interfaces in the system.

An SCL procedure is structured as a single procedure whose interface is specified in a *template*, which defines the parameters of the procedure, their names, their data types, keywords which may be used to identify them and default values. Data declarations in the procedure allow variables and constants local to the procedure to be defined. Data types include integers, booleans, strings and arrays of each of these types. The body of the procedure is written using conventional structured programming facilities such as *if then else, case, while* and *do* loops etc.. SCL BEGIN and END statements create and destroy resource allocation blocks enabling control over the scheduling of resources from SCL.

SCL procedures can be executed as source files or compiled into a much more efficient form known as *SCL Intermediate Code*. Compiled SCL procedures are stored in OMF modules and can be manipulated and loaded in exactly the same manner as OMF modules generated by any other compiler. In addition, an SCL template can be inserted into any OMF module for a procedural interface in that module, allowing the interface to be called as an SCL command.

It is often desirable for data values to persist between different procedures or applications. For this purpose *SCL Jobspace* is provided. Variables can be declared which persist in Jobspace and which are accessible as external data to any procedure or program. They are frequently used to record information specific to a particular job - *e.g.* a user name, or environment variables.

### *Interactive SCL*

An interactive user can enter SCL statements, including calls to templated procedures, directly at the terminal. Parameters to procedures can be specified:

- positionally, *e.g.* Command( parameter_1, parameter_2, . . parameter_n )
- by keyword, *e.g.* Command( key_1 = value_1, key_2 = value_2, . . key_n = value_n )
- interactively, by entering parameters into a form.

The interactive style of calling a procedure is known as *Prompting*. It is invoked by entering "Command?" (where Command represents a named procedural interface). Similarly, a single "?" can be entered instead of the value of any parameter. This causes the SCLS *Interactive Help* system to display help information about the relevant parameter. The Help system can also be invoked to display information about a command or various aspects of the system by entering "Help(Command_Name_or_Topic)". Alternative or additional help files can be created by the user to accommodate particular requirements.

### *SCL Programs and Batch Control Files*

An *SCL Program* is a file containing SCL statements with all the facilities described above though not structured as an SCL procedure; it does not, therefore, have parameters. SCL programs are interpreted directly from SCL source files.

### *Control Programs*

Each job has a notional source of SCL statements used to describe the actions to be performed by the job. This source is handled as a standard file and is termed the *Control Program.* Output from interactive jobs is returned to the terminal; for interactive or batch jobs, output can (also) be written to a job journal.

For a batch job the control program is taken from a standard serial file. The file contains SCL statements and is read sequentially. Provision is made for *embedding* data in the file for applications which read data from the standard control program source.

In the case of an interactive job (*e.g.* in a MAC work environment) the control program is taken from the interactive terminal by means of an interactive dialogue cycle with the user:

- The user is prompted to enter an SCL statement;
- The user enters an SCL statement which might call an application or command;
- The SCL statement is executed and any output returned to the interactive terminal.

When an application or command is being executed in an interactive job, the user may wish to temporarily suspend execution in order, for example, to examine the contents of a file or the status of a queue. The user can perform a *break-in* from the terminal with a special key combination. Execution of the current SCL statement is then suspended and a new interactive dialogue cycle initiated. Within this new dialogue cycle, the user can choose to *continue* or *quit* the suspended SCL statement (thus leaving the new cycle), or to enter a new SCL statement. Break-ins are handled recursively so that a *stack* of *nested break-ins* can be created up to a system-defined maximum depth.

Execution of an SCL statement in a new dialogue cycle does not directly affect the state of the resources associated with the suspended SCL statement; however in some circumstances, resources might be shared between them (*e.g.* files) so that interactions could occur. This can be exploited very effectively in some applications: a single Virtual Resource might provide several templated interfaces to operations which allow its behaviour to be modified or its internal status inspected. Execution of an application embodying the resource could then be suspended by break-in and the additional interfaces invoked from the break-in dialogue cycle.

### *System Control Language System (SCLS)*

SCLS is a multiple Access Level subsystem having two main components, referred to as the Control Level and the Service Level. The Control Level, executed at the current job Base Access Level, is the level at which SCL is executed and the user's Jobspace is manipulated. The Service Level, executed at Access Level 8, carries out the more privileged functions such as control stream access, manipulation of data which must be protected from direct access by the user, SCL syntax analysis and Intermediate Code generation, .

The main functions of SCLS are:

- provision of the SCL environment - jobspace;
- execution of SCL statements, programs and procedures;
- support for block-structured resource allocation and de-allocation;
- support of prompting and help for interactive users;
- support for interactive features such as *break-in*.

## Work Management & Scheduling

OpenVME is designed to handle a wide variety of workloads ranging from a system dedicated to running a single massive OLTP service to a system running a mixed workload of OLTP, Batch, MAC and other work types. Work is managed in terms of *jobs* and for each type of work there is a *Scheduler* which manages the allocation of jobs to user VMs which actually execute the jobs.

Schedulers for the common work types are a standard part of the system. Additional schedulers may be user-written or can be created by using the customisation features of one of the standard schedulers. The standard schedulers are:

- The Batch Scheduler (JXS);
- The MAC Scheduler (MXS);
- The OPER Scheduler (OXS);
- The Transaction Processing Scheduler (TPXS);
- The Output Scheduler (OPS), which handles spooling and listfile requests;
- The File Transfer Scheduler (FTS), which handles file transfer requests;
- The Filestore Management Scheme Scheduler (FMS);

OpenVME provides comprehensive facilities for defining the combination(s) of work which may be running concurrently at any time, known as the *workmix*. A workmix definition specifies, for each main type of work, parameters constraining the behaviour of the relevant scheduler:

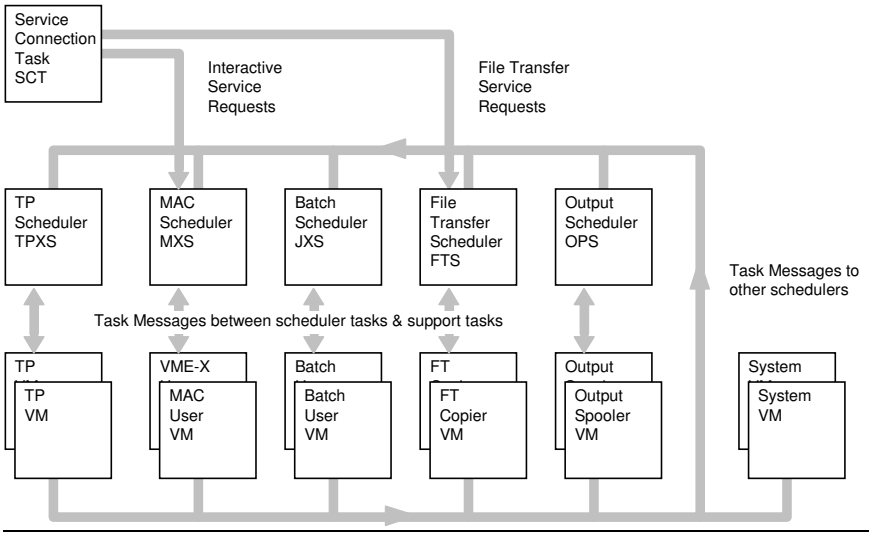
- the total concurrency of jobs of that type;
- individual minimum and maximum concurrency levels for each type of Support Task capable of executing jobs of that type;

At a lower level every task is assigned to one of several *policies* which define low-level scheduling characteristics such as time-slicing, range of priorities and memory requirements.

New jobs are submitted by sending a request to the appropriate scheduler via a task message. If the system is unable to execute the job immediately because of insufficient resources or there is no support task currently available, the job is queued for subsequent execution. Queues are held on permanent storage so that, in the event of a system failure, queued requests can be recovered automatically. Various queuing disciplines are supported including first-come-first-serve, relative-priority and absolute-priority.

When a job is started the appropriate scheduler task allocates a support task to it, if necessary creating a VM in which to run a new instance of the support task, and then passes details of the job to the allocated task. The support task then invokes work initialisation procedures to establish the required work environment for the job. Whilst the job is active the scheduler continues to control certain aspects of its execution by exchanging task messages with the support task. When the job is complete, the scheduler is notified by a task message, and the support task returned to the task pool.

Facilities are provided for a user or operator to communicate with any job in the system whether it is queued or active. Each job is allocated a unique *operator tag*, by which it can always be identified. As responsibility for a job passes between one task and another (*e.g.* from a scheduler to a user task), the operator tag becomes associated with each task in turn. A command or message to a job identifies the job by its operator tag and is handled by the task with which the operator tag is currently associated. Standard commands include the ability to abandon a job or to change its scheduling characteristics.




*Work Management interactions between scheduler & support tasks*

### *Work Management (WM, WMX, WMT)*

Work Management (WM) is a multi-Access Level subsystem providing various common support facilities for scheduler tasks. It has components at Access Levels 5, 8 and 9, and user level. Its most important functions are:

- providing non-volatile queue support for JXS, OPS & FTS;
- providing a means of defining, changing and interrogating permanent scheduler data;
- providing facilities such as standard tracing.

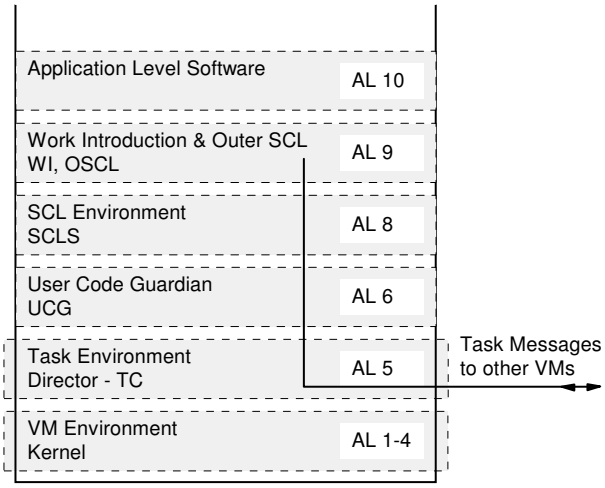
Most of the active functions of managing the workload to conform to the required workmix is performed in the Workmix Task (WMT). Workmix Task runs the *workmix algorithm* and communicates with the schedulers, via task messages, to inform them of relevant changes to the workmix. Workmix (WMX) provides in-process interfaces to allow a suitably privileged user to define, change or interrogate the current workmix. Where necessary, WMX procedures communicate with WMT via task messages.

### *Work Introduction (WI)*

Work Introduction provides interfaces which enable a VM to communicate, in a standard manner, with a scheduler VM in order to submit a new job, or with a job (via its operator tag) to modify or interrogate some aspect of its handling. Standard routes to the Output and File Transfer schedulers are provided to allow file listing and transfer requests to be submitted.

### *Above-Director Subsystem Layering*

The diagram below illustrates the layering of Above-Director subsystems within a VM, and shows the use of task messages for communication between scheduler and support tasks.




### *Outer SCL (OSCL)*

Outer SCL is the subsystem responsible for establishing the Work Environment within the support task in a user VM. A user VM is created running a support task whose initialisation includes execution of standard OSCL procedures which create the initial OSCL environment. The OSCL environment includes the mechanisms required to communicate with the scheduler task responsible for the work type with which the user VM is associated. Default handlers for contingencies and operator messages (directed to a job tag) are also provided.

When a scheduler allocates a user VM to execute a job, it sends a task message to the support task, which is handled by OSCL procedures. After initialisation OSCL processing comprises a cycle, the *OSCL processor loop*, whose main steps are:

- wait for and receive a job request from the scheduler via a task message;
- begin a new SCL Resource Allocation Block;
- perform any standard job initialisation including journal creation; establishment of spool and listfile batches in conjunction with the Output Scheduler; switching to the required User and Account contexts; and establishing standard file, library and loading contexts;
- execute any standard user-specific job initialisation (USERJOINIT);
- establish the control program - either a magnetic file or an interactive file associated with a terminal device, as specified in the job request;
- execute the control program for the job itself; this may further extend the Work Environment by loading and initialising additional Virtual Resources within the VM before commencing the main part of the job;
- execute any standard user-specific job completion actions (USERJOFINAL);
- perform any standard job completion actions including requesting journal listing, completing spool and listfile batches;
- release any remaining resources acquired by the job by ending the Resource Allocation Block.

## Language Support

### *Language Run-time Libraries*

OpenVME language systems generate compiled OMF modules. In most cases the OMF contains compiler generated references to language-specific routines which support standard or optional features of the language -*e.g.* print formatting routines or mathematical functions. These routines are collected into one or more additional OMF modules, with named entry points for each routine. The modules constitute the *run-time library* for the language and are considered as part of the language system.

### *Testing & Diagnostics*

The OMF definition provides for detailed diagnostic information to be contained in an OMF module to enable symbolic debugging and tracing. When a source module is compiled, the programmer can specify the level of diagnostic information included in the OMF module. At run-time, part of the standard Application Programming Support environment is the *Object Program Error Handler* (OPEH).

The user can specify the level of diagnostic and tracing facilities required, provided these are compatible with the diagnostic information in the OMF module. When the application is first entered, compiler generated code invokes OPEH which then requests UCG to notify contingencies such as Program Errors to a nominated OPEH procedure. When a contingency arises, OPEH uses the OMF diagnostic information to generate an error report which can be displayed to an interactive user or written to a job journal. Similarly, if run-time tracing is required, the trace can be displayed on a terminal or written to a journal. In either case, if suitable compile-time and run-time options have been specified, information is produced in terms of source procedure and variable names, line numbers etc.

### *Profiling*

The Program Activity Sampler (PAS) can be used to produce a *profile* an application's execution. PAS works by periodically interrupting the application and saving a snapshot of the process state, including the Program Counter and addresses of operands. When sampling ceases or the application is terminated, an analysis phase generates a profile of the activity during the sampling period. PAS uses loader and OMF diagnostic data to report profile information in terms of source procedure names and data areas.

## Protocol Handling

### *Transport Interface (TI)*

The Transport Interface subsystem supports the X/Open Transport Interface (XTI) which provides access to the OSI and TCP/IP & UDP/IP transport providers. Libraries are provided in both the X/Open Application Environment and within the VME C language to map calls in C on the XTI to equivalent calls on OpenVME Transport Service interfaces.

### *OSI Application Layer APIs*

X/Open conformant APIs are provided to various application layer services, most notably:

- The X400 Message Access service (MA);
- The X500 Directory service (DIR);
- The OSI Remote Operations Service (ROSE);
- The OSI Object Manager (OM).

# Out-of-process Subsystems

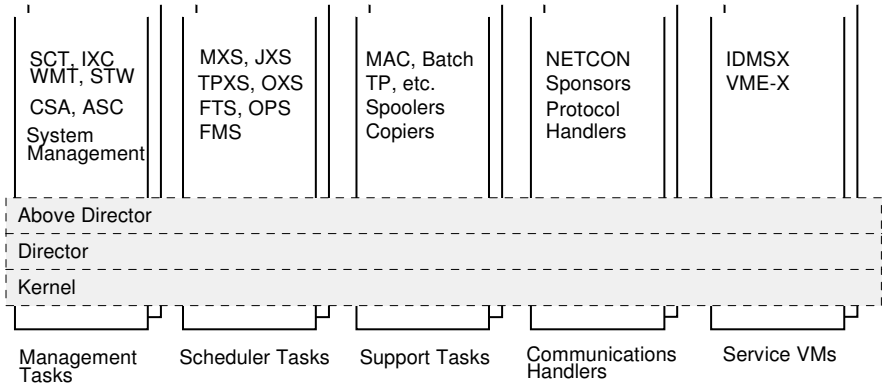
## Introduction

Several major areas of OpenVME functionality provide common services for the whole system but are not directly part of the in-process application environment. The subsystems which support these functions are not executed within the calling VM (or process) but are implemented in separate VMs. This allows the subsystems to exhibit autonomous behaviour. Such subsystems are termed *out-of-process* and are usually executed in dedicated VMs as *System Tasks*.

Normal inter-VM communications mechanisms, most commonly task messages, are used by other VMs to request the functions offered by out-of-process subsystems. Typically a component of the application environment provides an API which is executed in-process and, invisibly to the calling application, passes the request to the VM in which the target subsystem is being executed.

The most important classes of functions provided in this way are:

- Schedulers
- Work management tasks
- Spoolers & Copiers
- Communications Protocol Handlers
- Service tasks



## Schedulers

Scheduler tasks are responsible for allocating work to appropriate support tasks. Each scheduler is maintains, according to the current workmix definition, a pool of support tasks capable of executing the types of work for which it is responsible. Typically a scheduler receives notification of an incoming request for a service and allocates an appropriate support task to provide the service. If the number of requests exceeds the resources available, the scheduler is responsible for queuing excess requests until resources become available. A scheduler provides operator interfaces to control its operation and to allow manipulation of its queues and task pools.

### *Execution Schedulers*

Batch execution scheduler (JXS)
TP execution scheduler (TPXS)
MAC execution scheduler (MXS)
OPER execution scheduler (OXS)

### *Spooler & Copier Schedulers*

Output scheduler (OPS)
File Transfer scheduler (FTS)
Filestore Management Scheme scheduler (FMS)

## Work Management Tasks

Service Connection Task handles interactive service requests. It handles user authentication, service identification and soliciting for any relevant service options. It then passes the interactive connection to the appropriate scheduler or VM for execution. An associated task, Index Constructor (IXC), assists SCT in building a lists of all services accessible to the (possibly authenticated) user. SCT is also the *default listener* for all incoming communications connection requests, and is responsible for passing them to the appropriate scheduler.

### *Service Connection*

Service Connection Task (SCT)
Index Constructor (IXC)

It is often convenient to run multiple interactive sessions from the same terminal, switching between them from time to time. Concurrent Session Access (CSA) and Advanced Session Control (ASC) provide such facilities.

### *Session Management*

Concurrent Session Access (CSA)
Advanced Session Control (ASC)

Workmix Task (WMT) is responsible for evaluating the current system load and co-operating with the schedulers to ensure that concurrencies of different types of workload match the available system resources. System Task Watchdog (STW) monitors the availability of specified System Tasks, starting them or, in the event of a failure, restarting them as required. System Task Watchdog , is responsible for starting other System Tasks and performing a watchdog role over them. If a System Task which STW has been requested to oversee fails, STW attempts to start a new instance of the task to take over the functions of the failed task.

Workmix Task (WMT)
System Task Watchdog (STW)

## Spoolers & Copiers

File Transfer copier (NIFTP & FTAM)
Filestore Management Scheme (FMS) copier
File Transfer And Manipulation (FTAM) copier
Output Spooler (OSP)
Page Spooler (PSP)

## System Management Tasks

The Programmable Operator Facility infrastructure provides a System Task environment in which customised prompt handling and periodic house-keeping procedures are run. A POF task is initialised at Access Level 9, declaring the existence of a programmable operator to the Director subsystem EFCH. ECFH routes prompts to the appropriate VM, either an interactive OPER VM or a POF VM, depending on the characteristics of the prompt and the subset of those in which any potential recipient VM has declared an interest. In the case of Automated System Operation, the ASO task is based on POF, customised with procedures of the *Automated Operator Facility* (AOF).

Other standard system management tasks are described in Chapter 13.

Automated System Operation (ASO)
Error Logging VM (ELVM)
Remote Management System (RMSV)
Remote Management System for VME-X (RMS)
Support and Maintenance (SAM)
Total System Teleservice (TST)

## Miscellaneous System Tasks

Certain services which are available as part of the application environment in each user VM have state which must be co-ordinated across all VMs using the service. The state may also be required to persist even when the service is not currently in use by any VM. A common way of satisfying these requirements is for the service to have a *Service VM* which is permanently active and which maintains global data, accessible to any VM using the service. Examples of services with Service VMs are IDMSX (and other data managers accessed in-process) and VME-X.

IDMSX Service VM
VME-X Service VM

## Sponsors & Communications Protocol Handlers

The OpenVME Networking Services architecture is described in detail in Chapter 9. Below is a summary of the major communications services provided by out-of-process protocol handlers.

Communications Network Controller (CNC, NETCON)
TCP/IP (XTI), Streams (STR)
Layer protocol handlers (*e.g.* Yellow Book Transport Service)
OSI TP Gateway (OTP)
X400 Message Transfer Agent (MTA)
X500 Directory Service Agent (DSA)
Remote Session Access (RGT)
Asynchronous Sponsor Service (ASS)
Asynchronous Terminal Handler (ATH)
Virtual Terminal Sponsor (VTP)
Transport Relay (RLY)
X400 Remote Sponsor (X4SP)

## Commands & Utilities

OpenVME provides a large number of commands for performing various routine operations. Some of these are built in to the OpenVME system; others are provided as free-standing, loadable modules; in either case, the commands can be called in similar ways, whether interactively or by program, A selection of important commands is identified below to illustrate their range and scope.

### *Volume & Partition Utilities (VPU)*

Volume, Category and Partition management, archiving & recovery

### *File Utilities (FC, FU, FMSUI etc.)*

File creation, copying, deletion, archiving & recovery etc.
File listing, file transfer etc.
Simple File Access (from SCL)

### *OMF Utilities*

Module Amender (MA)
Collector (COLL)
Module Lister (LM)
SCL Compiler

### *Basic (record) Utilities (BUS)*

List Records (LR)
Append Records (AR)
Copy Records (CR)
Match Records (MA)
Merge Records (ME)
Sort Records (SR)

### *Text File Utilities*

Edit (ED) & Screen Edit (SD)
Browse File (BRF)

### *Catalogue Utilities*

Introduce_x
Change_x
Display_x_details
Change_x_permisssions

### *System & Work Management Utilities*

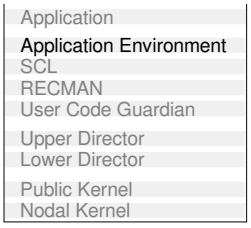
Commands to jobs, schedulers, workmix etc.
Tracing and diagnostic control
Performance monitoring
Loadset, checkpoint, catalogue utilities
Hardware management commands
Network Control commands

### *X/Open Commands & Utilities*

Within the X/Open work environment the standard set of Commands and Utilities defined in the X/Open Portability Guide is provided. Additional commands and utilities are provided where appropriate - *e.g.* associated with optional features beyond the X/Open base definition.

# Chapter 6
# Application Environments

## Introduction




An *Application* is an active (software) component of system providing functionality or services specific to a user's requirements. An application is executed in a context, the *Application Environment*, which is specific to the application's particular usage and which provides supporting services to the application.

This chapter describes the means by which application environments, including interworking services, are established within a VM. The facilities for supporting co-operative processing and Client-server relationships between applications are identified.

## Basic Concepts

An *Application Environment* comprises the Virtual Resources which an application uses in providing *Application Service*(s). The Virtual Resources provide supporting services, invoked via *Application Programming Interfaces* (APIs). For example, an application environment supporting transaction management services creates a Virtual Resource which provides the X/Open transaction management services, via the TX API.

Application environments are specifically tailored for particular workloads. An application environment within a VM is established by creating and initialising the Virtual Resources required to support that environment. Virtual Resources can be created at the time of VM creation, when an application is first started, or dynamically during execution of an application.

Many IT Business Solutions require the co-operation of several applications, invariably making use of underlying *interworking services*. OpenVME provides such services at various levels, from simple *Networking Services* (which the applications have to use explicitly) to high level *Distributed Application Services* (with which the distribution is invisible to the applications). These underlying interworking services may be provided as part of the application environment within a VM, as described above.

In any case, co-operation between particular applications may be established in various ways, notably:

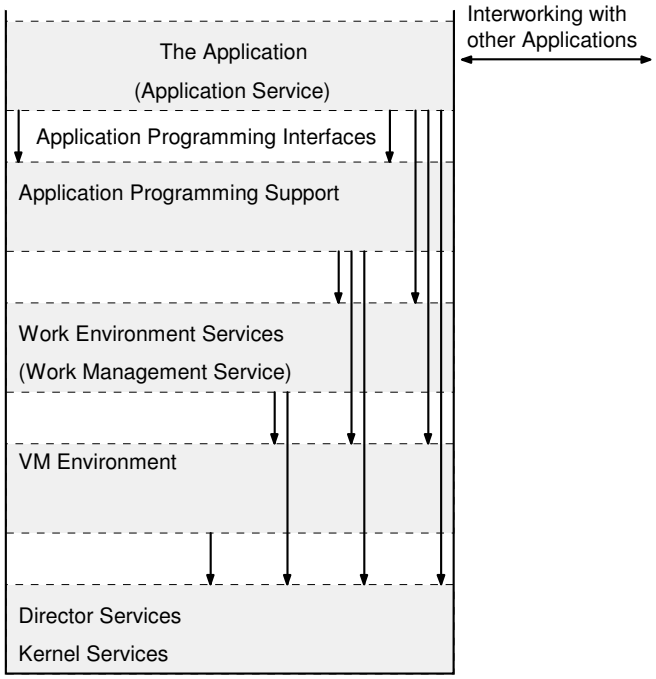
- statically, during application development;

- statically, during application installation or initialisation;
- dynamically, during application execution;
- for each interaction.

In the most general case, the concept of an isolated application becomes less meaningful as each application may co-operate, dynamically, with other applications in providing the particular service for which it is responsible. This is the model of *distributed computing* or *co-operative processing*, in which applications co-operate by using services provided by other applications. In this model, an application acting in the role of a *client* is said to use the services of an application acting in the role of a *server*, thus establishing a *client-server* role relationship between the applications.

## A Structured View of Application Environments

The environment in which an application is executed can be considered as comprising a set of Virtual Resources. The Virtual Resources are organised into several layers. Each layer is concerned with a different aspect (or level of abstraction ) of the environment in which the application is executed.




The lower layers are concerned with providing a rich initial VM environment which is broadly independent of any particular application or application type. This layer includes the ability to enhance the environment dynamically by

"loading" further software modules from a *loading environment* into the VM, thus creating additional Virtual Resources.

The next layer provides a *Work Environment* for the application, controlling the introduction and subsequent organisation of work within the VM; the facilities provided by the Work Environment depend on, for example, whether the application is interactively controlled or not.

The layer closest to the application, Application Programming Support, provides most of the services associated with open application environments. Together, *all* of these layers provide services which may be used directly or indirectly by the application. The services are invoked via Application Programming Interfaces (APIs). Wherever appropriate, the interfaces conform to the relevant open standards.

The concept of an application environment is contextual. An "application" may, itself, provide elements of (or a complete) application environment to a higher level "application". This recursive structure extends downwards as well and is a natural consequence of the generalised manner in which OpenVME allows Virtual Resources to be dynamically created in a VM, using the services provided by other, previously created Virtual Resources. Each layer of the recursive structure, in providing its own services to higher layers, encapsulates services provided by lower layers.

## OpenVME Work Environments

A *Work Environment* is that part of an application environment which relates to how work is introduced and subsequently organised. This is not a precise distinction: rather it is convenient to group applications whose application environments have these characteristics in common and to support them in a standard manner. Any particular job, session or application is executed in a work environment.

OpenVME provides several standard work environments, each designed for a particular class of work. Additional Work Environments can be constructed by combining components used to construct the standard Work Environments.

The following list identifies most of the standard Work Environments. It is not exhaustive and, because of the flexible manner in which Work Environments can be created, the items are not necessarily exclusive. For example, the Spooler and Pseudo-job are specialisations of the System Task environment.

### *Single-threading Work Environments*

- The *MAC* (Multi-Access Computing) work environment provides facilities for interactive use of VME. It is always associated with an interactive terminal or workstation. The user can issue SCL commands,

invoke applications (which may be interactive) or initiate jobs in other work environments.

- The *VME-X* work environment provides an X/Open branded Common Application Environment for interactive sessions and free-standing (*daemon*) processes.
- The *Batch* work environment provides facilities for work which is not directly dependent on an interactive terminal. The work is controlled by a file of SCL commands.
- The *Spooler* environment provides facilities for input and output spoolers, and file-transfer copiers, under the control of an appropriate scheduler.
- The *TP AVM* (Transaction Processing Application VM) work environment provides transaction management facilities for the support of TP applications. The AVM is serially re-used, performing one complete transaction (allocated by the CVM) before the next.

### *Multi-threading Work Environments*

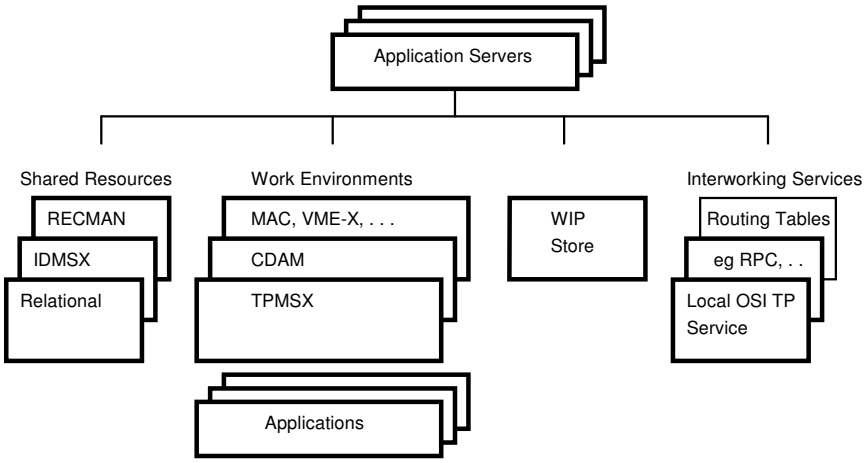
- The *TP CVM* (Transaction Processing Control VM) work environment provides scheduling for the support of TP services which interact with large numbers of terminals or workstations, other (local or remote) applications, or other transaction sources.
- The *CDAM* (Co-ordinated & Distributed Application Manager) work environment provides an facilities for non-TP applications which nonetheless require transaction management services to co-ordinate their operation.
- The *System Task* environment provides an environment in which OpenVME system services, such as schedulers, system management sponsors, communications protocol handlers, Service VMs, etc. may be executed within VMs.
- The *Pseudo-job* environment (PJE) provides facilities for user-written System Tasks.
- The *Communications Service Infrastructure* (CSI) environment provides facilities for complex communications-based services.

## OpenVME Application Servers

An OpenVME *Application Server* is formally described in the catalogue and is thereby uniquely associated with the provision of certain *Application Services*. It is an autonomous unit, operating independently of other servers, and is a unit of portability between systems. It is responsible for the consistency and availability of all its resources, notably data. An important consequence of this is that an individual resource may only be accessed under the control of a single application server.

An application server comprises:

- Applications providing the functionality of the services offered;
- A set of *Resource Managers*: databases or organised file services;
- A *work-in-progress store*;
- The means to interwork with other servers, and the associated routing tables;
- The dedicated work environments required to support the services offered.



The concept of an application server provides a convenient encapsulation of a logical group of functionality into a manageable unit with well defined interfaces. Application servers conform to certain rules:

- An individual application server exists only on a single OpenVME system but can be relocated, as a single unit, between systems;
- An application server, its components and structure are formally defined in the catalogue;
- Each application server is unique in terms of the combination of application services it offers, their names, the resources it owns and the data it holds;
- Access to each application service supported by an application server is through well defined interfaces which allow the server to police the clients which access it and the actions they perform;
- Access by an application to application services provided by other servers is through well-defined interfaces;
- An application server is uniquely addressed by its application services; thus a client need not know where it is actually located.
- An application server is autonomous. It can continue to offer its services even if other servers in the same system fail;

*Server Manager* (SMAN) provides facilities for managing an application server. Application servers are managed in a uniform manner, independently of the services supported by the server. The server management facilities provided by SMAN include:

- the means of cataloguing the components and structure of an application server;
- interrogation of the structure and status of components of a Server (such as Resource Managers and Work Managers);
- run-time control of a Server, including facility to *start* and *close* Servers.

### Applications, Application Services & Application Servers

An application is executed within the work environment defined by a particular catalogued service; this service is thus a generic work management service *used* by the application. The application may subsequently *bind* itself to a catalogued application service (which may be associated with an application server); the application thus undertakes to *provide* the application service. The manner in which an application binds itself to an application service depends upon the work environment in which it is being executed, the nature of the service it is providing and the means by which that service may be externally accessed (*i.e.* specific interworking mechanisms).

As already described, an application server is uniquely associated with the provision of certain application services; the essential property that associates those services is that they access *shared resources* under the *exclusive* control of the application server, which is responsible for maintaining the consistency of those resources. Furthermore, an application server also provides the means for applications to access application services provided by applications associated with other application servers and *vice versa*.

Moreover, when required and when using suitable interworking methods, the application server provides the means for applications to *co-ordinate* with other applications (which may be associated with other application servers). This allows co-operating applications to maintain consistency of resources under the control of multiple application servers.

## Establishing Application Environments

An application environment is established by the creation of Virtual Resources to provide the required services. They are created at various times:

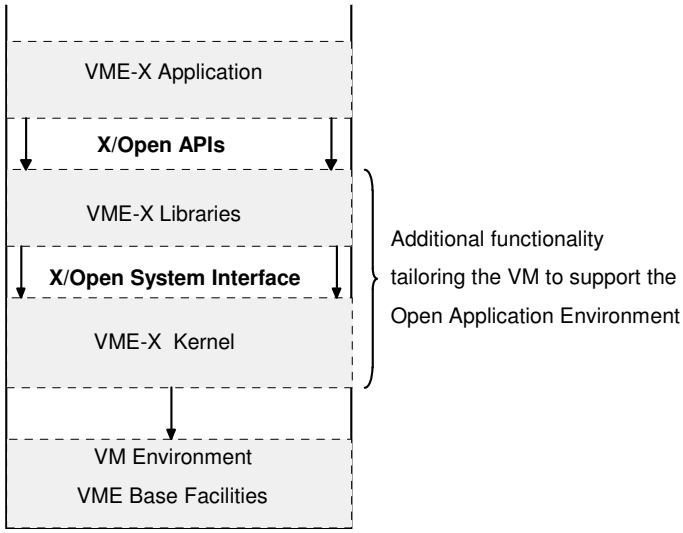
- When a VM is created it is endowed with various scheduling attributes and capabilities. When it is initialised, it is associated with a particular work environment and the appropriate Virtual Resources to support the environment are created.
- When a job or session is first allocated to a specific VM (*e.g.* by a work management scheduler), the work environment is initialised for the specified service and particularised for the job or session.
- When a job or session is active, an application is started and the Virtual Resources supporting any required APIs are dynamically created. These resources constitute the Application Programming Support environment.

The block-structured scoping of resources within the OpenVME system enables resources to be deleted in a manner complementary to their creation. Thus within a session, the resources created for a particular application are relinquished when the application is terminated.

## Open Application Environments

### The X/Open Common Application Environment: VME-X

The VME-X application environment is fully X/Open conformant. It provides the full base set of X/Open System Interfaces (XSI) and libraries together with the standard Commands and Utilities. ISO standard C and the associated libraries are provided. VME-X therefore provides an environment into which applications written to X/Open standards can be simply ported.




The VME-X Application Environment comprises:

- A *VME-X Service*, which is a work management service defining various aspects of environment specific to that service such as the permitted service users, filestore, networking services etc.
- A work environment, provided by the *VME-X Kernel*, which supports the X/Open System Interfaces. This VME-X Kernel supports multiple concurrent VME-X *processes* within a single VM and manages the sharing of the VM resources between them. Each process executes a single *program*.
- VME-X libraries which support the standard X/Open APIs. These libraries can be bound in to an application program when it is being constructed during the phase known as *link*ing; alternatively, commonly used libraries can be *shared* in which case linking takes place when the application program is loaded.
- The services of a specific VME-X *Service VM*. The VME-X Service VM, via the VME-X kernel, provides overall co-ordination for all VMs associated with a particular VME-X service.

A comprehensive set of features is provided in addition to the mandatory base facilities. These include: C-ISAM & SQL (Informix), COBOL 85, TCP/IP & UDP/IP (via sockets & XTI), uucp, mail, ftp, telnet and several Berkeley Unix and Unix SVR4 features to facilitate practical application portability.

## The VME-X Architecture

### *Basic Concepts*

The definition of the X/Open CAE is derived from that of the UNIX operating system. In the UNIX environment, work proceeds by the execution of *processes*. The process is the fundamental unit of software composition in UNIX. Whereas in other operating systems large applications can be constructed by software components invoking each other directly, by procedure call, in UNIX this is often achieved by creating a new process in which the called software component is executed. A process is defined as:

- an *address space* in which the process state is represented; this comprises *text* (executable code), *stack* and *data* (statically or dynamically allocated) areas;
- a single thread of control that executes within that address space;
- the system resources required by the process.

A process (the *parent*) can perform a *fork* which creates a new process (the *child*) which is an almost exact copy of the parent process. After a fork:

- a new address space is created, an exact copy of the parent process's address space;
- a new thread of control is established whose point of execution, at the time of forking, is exactly that of the parent process;
- system resources available to the parent process (*e.g.* open files) are made available to the child process;
- each process can determine whether it is the parent or the child.

Within each process a single *program* is executed. A process can perform an *exec* which causes a process to enter a new program within a re-initialised address space. After a *fork*, the child process frequently *exec*s a new program.

### *VME-X Services*

A VME-X service is a work management service. An OpenVME system can support several VME-X services concurrently. Each service resembles a multi-user UNIX system in its own right, and behaves as a separate open system with its own resources - in particular, an independent X/Open-compliant filestore. VME-X services can be protected from each other using VME's security facilities. Each VME-X service recognises a subset of the people known to the OpenVME system, and only recognised users are allowed to log into the service.

### *The VME-X Work Environment*

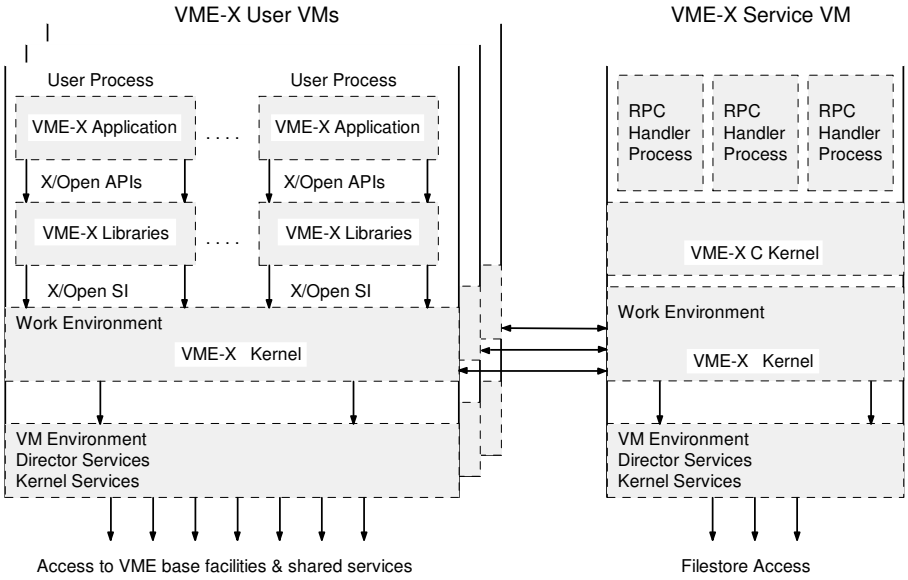
The VME-X Kernel runs mainly at access level 10, partly at 9. In each VM, there is a separate instance of the VME-X kernel's data, which is all local. The VME-X kernel has two basic functions:

- it has the ability to load a program in standard UNIX executable file format (COFF) from VME-X filestore and enter it; the program is loaded into an address space within the VM and is executed as a VME sub-process.
- it provides a set of entry points to which the program's kernel calls can be fixed up. The System Interfaces (the XSI) which the program calls are all implemented as library routines, but some of these routines issue calls to the VME-X kernel.

All VME-X application programs execute at Access Level 11. When they call the kernel, the calls are translated into non-stack-switching system calls to the VME-X kernel. The process's Access Level drops to 10 while executing in the VME-X kernel, and perhaps lower still if the VME-X kernel makes inward calls into the VME operating system. On exit from the VME-X kernel, the Access Level reverts to 11. The read and write Access Keys of the VME-X kernel's data are 10, so it cannot be corrupted or read by applications. The VME-X kernel makes extensive use of interrupt events to detect I/O conditions, timer requests made by user programs, hardware-detected faults in user programs, and messages from the VME-X Service VM. The execution Access Key of all these events is 10, so the VME-X kernel can take interrupt events, no matter what the user-level code is doing. On the other hand, these events do not interrupt the VME-X kernel itself.

The VME-X kernel implements many system calls directly. In some other cases the VME-X kernel calls interfaces provided in the underlying VME application environment. In the remaining cases, the VME-X kernel passes the request to the VME-X Service VM (see below). Code running in a VME-X environment cannot call VME system interfaces directly. The usual technique for making a set of related VME functions accessible is to use a device driver, so that the VME-X application can use a special file and make stylised calls on standard interfaces such as write() and ioctl().

When the VME-X Work Environment is first established an initial program is automatically executed. For interactive VMs, this is /etc/init and for daemons, /etc/initadi. When initialisation is complete, a new process is forked to execute the initial program specified for the user in the /etc/passwd file.




### *The VME-X Service VM*

There are many aspects of a VME-X service which require co-ordination across all the VMs associated with that service. For example, access to files within a filesystem owned by the service must be properly sequenced to ensure the integrity of filestore data. In many cases, the VME-X kernel can achieve this by suitable exploitation of underlying VME facilities. This is true, for instance, of all non-disc I/O, pipe handling, most signal processing, and scheduling of processes within a VM. Some functions which a UNIX kernel normally has to handle - *e.g.* virtual store management - can be left entirely to VME. Many others are supported by specific VME subsystems - *e.g.* address spaces and inter-process communication (ASR and QISH).

In other cases, VME alone does not provide adequate co-ordination. In these cases, the VME-X kernel passes the request to the VME-X *Service VM* using VME message-passing facilities. Since there is only one Service VM per service, inter-VM interference cannot occur. Many of the less critical, more complex kernel calls are processed in the Service VM, as are certain disc I/Os.

Process scheduling takes place at two levels. Each VM is scheduled by VME according to its priority. Within each VME-X VM, whether user VM or Service VM, the VME-X kernel schedules the processes using the sub-process call mechanism. When a process issues a request to the Service VM, the VME-X kernel will not allow it to run again until the Service VM replies. In the meantime it will schedule other processes in the VM if possible. In theory, a terminal user can have two VMs on two different processors, working simultaneously. The VME-X kernel also reschedules processes if the current process exceeds its timeslice or if it sleeps.

### *The C Kernel*

The Service VM has a VME-X kernel, but does not obey /etc/init or /etc/initadi. Instead it loads a COFF program called the *C Kernel*. Each request from another VM in the service is passed to the C kernel for action, and the VME-X kernel issues a reply when the action is complete. Much of the source of the C Kernel is derived directly from UNIX System V source code obtained under licence from Unix System Laboratories Inc. (USL).The advantages of using USL source are lower development costs and accurate conformance to the System Interface definition as well as any unwritten folklore that applications ported from UNIX may rely on.

The C kernel's interface to the VME-X kernel is quite different from the UNIX-like interface of user VMs. Processes are also handled differently; there is a pool of processes, allocated on demand to handle each incoming request from a user VM. For instance, if a user VM issues a file read request, this is passed to the Service VM, which allocates a process. Several physical transfers may be needed, and the process remains allocated until the last of these is complete. Each process in the Service VM has its own stack but, in contrast with user VMs, there is only one copy of the static data. This reflects the fact that the C kernel is modelled on parts of the UNIX System V kernel, which expects to operate on only one instance of most of its data.

### *Device Drivers*

A UNIX System V kernel has standard internal interfaces known as the DDI & DKI into which Device Drivers, subsystems designed to drive particular types of hardware devices, can be inserted. The UNIX kernel expects all drivers to present it with a common interface; this is the DDI. The drivers themselves may use the facilities of the generic kernel by calling functions in the DKI. The VME-X kernel in VME-X also supports device drivers. Its device driver interface resembles the system V DDI, but is slightly different in form. Device drivers are compiled and constructed as ordinary VME programs, not as VME-X COFF files, and they have the entire VME application environment directly available to them. Each driver is a distinct executable file separate from the VME-X kernel, and linked from it using VME loading mechanisms when the user logs in.

### *Libraries, Commands & Utilities*

Standard libraries (*e.g.* for C, standard IO etc.) are provided together with corresponding C header files. Library routines can be incorporated by statically linking them into the application program. Commonly accessed libraries are also supplied as *Shared Libraries* in which case the application is linked to a shared copy of the library when it is loaded; this greatly reduces the size of the application program file and hence the time taken to load it.

Standard Commands and Utilities are provided, based closely on the UNIX System V source.

### *VME-X Filestore*

The VME-X filestore is closely modelled on the UNIX System V filestore. Each VME-X service can access up to 100 *filesystems*, each of which is implemented as a single VME file with permissions to the username which owns the VME-X service (and under which the Service VM runs). Permissions to individual UNIX files are enforced by the VME-X system itself. Several VME-X services can share access to a single filesystem (subject to permission checks) provided only one has write access.

### *VME-X Access to VME Resources*

VME SCL commands can be invoked by interactive users, and they can be used in pipelines just like any other command. Calls can also be made to other services, such as TPMSX services, but in this case the output cannot be redirected or piped.

VME files can be accessed, using the syntax "/dev/vme/vme-filename". The file is assumed to be a VME serial character file in the usual VME format and character encoding. A binary view of the same file can be obtained by using /dev/rvme instead of /dev/vme. The user's access rights to a VME file are evaluated just as if the file had been accessed from a VME service.

### *VME Access to VME-X Resources*

A user of an interactive OpenVME service such as MAC can issue VME-X commands on VME-X services to which the user has suitable access permissions. The VME-X commands are issued as input to whichever program is registered as the user's initial shell. Input and output can easily be directed into native VME files.

Similarly, an OpenVME user or application using a non-VME-X service can specify a VME-X file in any context where it could have specified a native VME file. The syntax is:

"other-service!/usr/lib/..." or "other-service!my_directory/my_file".

In the latter example, the user is accessing my_directory/my_file in the user's home directory. If specially requested, a binary view of the file will be given;

otherwise the file will be assumed to contain character data and it will be automatically presented to the VME application in ICL EBCDIC record format.

### *Administration*

Each VME-X service can have a separate administrator, so different departments within an organisation can each have their own service. The administrator is provided with a menu-driven package for performing routine tasks, including the recording of users, the allocation of users to file systems, mounting and unmounting of file systems, daemon control, setting of service parameters, and control of the uucp suite and NFS. The administrator's job is simpler than the equivalent job on most UNIX systems, as only a software service is being controlled, not a physical machine.

Some administrative tasks have to be carried out by the VME system manager. This includes the allocation of disc space for file systems, and the management of IP addresses, and OSI network and transport addresses, installing VME-X releases, and rationing the amount of processing resource each service may use.

The automatic file security systems of OpenVME can be used to maintain backups. With these systems, the backup tapes are controlled solely by the VME system manager, and the users never need to know about them, even when a file is being retrieved. The VME-X service administrator periodically creates disc archives containing files needing to be secured, and the OpenVME system automatically secures them. If preferred, an entire file system can be backed up, and this is particularly useful for the root file system. If the system is part of a network containing UNIX machines VME-X can manage backups for them too.

## The X/Open TP Application Environment

The Open TP application environment provides support for the X/Open model of transaction management via the appropriate X/Open APIs. The Open TP environment therefore provides an environment into which applications written to X/Open TP standards may be simply ported. Equally, applications developed to run under Open TP can be ported to X/Open conformant TP environments on other systems.

The Open TP application environment comprises:

- A *TPMSX Service* which is a work management service defining the major characteristics of the service(s) supported by the application;
- A work environment provided by TPMSX which establishes and manages the VMs required to support the TP service and provides the basic transaction management facilities;
- the TPMSX Application Programming Support environment which provides Open APIs for TP applications;
- the services of shared resource managers such as databases;
- the means to interwork with other applications.

Applications are written in standard C or COBOL using open transaction management APIs. Support for the APIs is provided by the VME Open TP application environment. Distributed applications are supported using the X/Open APIs and standard transaction co-ordination facilities allow such applications to perform distributed transactions.

A detailed description of the Open TP application environment is given in Chapter 7 (Transaction Management).

# Chapter 7
# Transaction Management

## Introduction

This chapter describes the facilities provided by OpenVME to support Transaction Management, Transaction Processing and Distributed Transaction Processing.

### Transactions

A *transaction* is an application operation or unit of work which is executed according to certain principles which ensure that, under *any* circumstances, the results of transaction are predictable. Execution of a transaction usually entails:

- receiving a *message* from a transaction source;
- processing the message, including any required database updates;
- returning a *reply* to the transaction source.

The principles, known as the "ACID properties", ensure that the effects of executing a transaction are well defined in any circumstances, including system failure. They are:

- **A**tomicity: a transaction must either complete as a whole, or not at all, leaving any data resource unchanged;
- **C**onsistency: the effects of a completed transaction are such that any data resource moves from an initial valid state to a new valid state;
- **I**solation: transactions must not interfere with each other even when executed concurrently: the effects must be as though they were executed serially;
- **D**urability: the effects of a completed transaction must be permanent and persist under any circumstances, including failures of hardware, software or system.

Successful completion of a transaction such that its effects are durable is termed *commitment*.

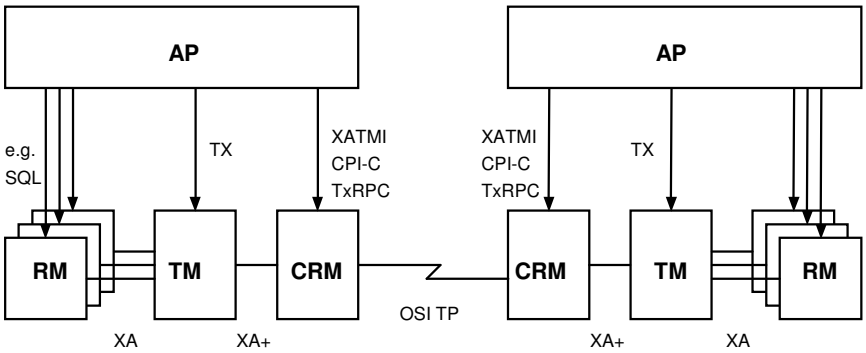
*Transaction Management* is a term describing the support of applications which need to perform operations with transactional properties. It is particularly suitable for workloads where data integrity and security, and high reliability are essential. *Transaction Processing* (TP) is a particular use of transaction management (and Information Management) facilities for handling large volumes of (usually) relatively simple line-of-business transactions (OLTP).

### Distributed Transactions

An increasingly important area of TP is *Distributed* Transaction Processing. In Distributed TP, a number of separate TP applications, distributed amongst several (possibly heterogeneous) systems, co-operate to support transactions which may access resources on more than one system. A special distributed co-ordination mechanism is used between all the applications participating in a distributed transaction to ensure that the ACID properties of the transaction are preserved. This is known as the *two-phase commit* mechanism and it is described below.

## Open Transaction Management

### The X/Open Transaction Processing Model




The X/Open model identifies several distinct functions within an active transaction processing system:

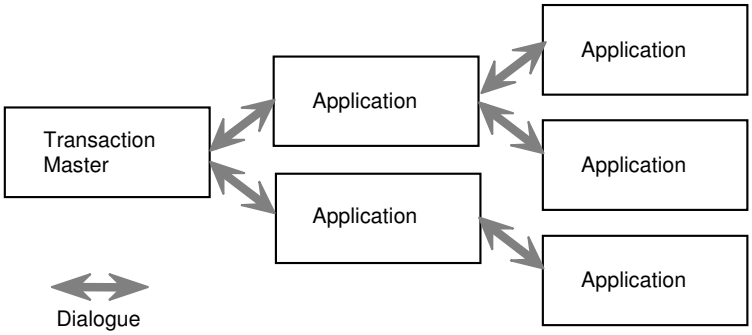
- a *Transaction Manager* (TM) which controls the co-ordination of one or more Resource Managers and/or Communication Resource Managers as directed by the Application Program;
- a *Resource Manager* (RM) which manages a resource - *e.g.* a Database Manager, a transactional print server, an ISAM file manager etc.;
- a *Communications Resource Manager* (CRM) which is used in Distributed TP to communicate with CRMs in other TP systems, providing application to application communication and inter-application transaction co-ordination;
- an *Application Program* (AP) which provides the user-specific functionality.

It should be noted that this is intended only as a programming model, identifying key programming interfaces; it is not intended to define an implementation architecture and does not, for example, imply any particular process structure.

### *Application Dialogues*

Applications co-operating in an interactive Distributed TP system communicate by means of an *application dialogue* which is an association between the applications, allowing messages and control information to be passed between them. The application which starts a dialogue is termed the *initiator* and an application which joins a dialogue is termed a *responder*. A responder may, in turn, initiate further dialogues and this gives rise to a directed graph, the *Dialogue Tree*, whose root is the original initiator.

Within a dialogue, an initiator may make requests of a responder which may return replies; for a particular transaction the pattern of request-reply pairs gives rise to another directed graph, a sub-graph of the dialogue tree, termed the *Transaction Tree*, whose root is termed the *Transaction Master*. The transaction master has a key role in distributed transactions: it makes the ultimate decision whether a transaction should be committed or not.




### *Two Phase Commit*

The *Two-Phase Commit* process is used to assure the ACID properties of a transaction which updates resources controlled by more than one Resource Manager. An important use of two-phase commit is for distributed transactions which may use resources on more than one system.

An essential part of the two-phase commit process is the action of *logging* information; this is the recording of that information on permanent or *non-volatile* storage.

The two-phase commit process involves several steps:

1. The application *starts* a transaction; it may then update resources controlled by RMs, which apply these updates to temporary or *volatile* copies of the resources; finally, it requests the TM to *end* the transaction;
2. In *phase 1* of the process, the TM asks each RM whether it is *ready* to commit its work: each RM logs all updates and notifies the TM that it is ready;
3. If and when all RMs have notified the TM that they are ready, the TM proceeds to *phase 2* in which it logs the decision to commit; it then requests each RM to commit its updates - *i.e.* to apply them permanently to the resources;
4. When each RM has committed its updates, it notifies the TM that it has committed; when all RMs have done so, the TM has established that the transaction has been successfully committed.

If any RM fails to declare itself ready to commit, the TM may then *roll-back* the transaction by requesting each RM to roll-back the local effects of the transaction, and discard any (temporary) updates to resources. If any RM, having declared itself ready in phase 1, fails to commit in phase 2, the TM may recover the transaction subsequently by re-attempting the commit process.

### *Queued Transactions*

In some cases, it is not desirable or acceptable for one application to issue requests to another and wait for a response before processing can complete. In such cases, applications can be decoupled by passing messages between applications via a queue which is under the control of a *Message Manager*. This technique is known as *queued transactions,* or, in the OSI TP standards, *queued data transfer*.

The sending of a message to a queue and, separately, the retrieval of the message from the queue may be performed as transactions. If sending, delivery and retrieval of the message are performed as transactions, and the queue itself has suitable characteristics of reliability and durability, then messages can neither be duplicated or lost. This technique is known as *transactional messaging*. Thus although queued transactions do not provide single distributed transactions, many of the properties associated with an interactive transaction (*e.g.* within a dialogue) are retained. In general, however, it is more difficult to design and reason about systems with the inherent asynchrony that results from the use of queued transactions.

# OpenVME Transaction Management Support

## Introduction

OpenVME provides extensive transaction management facilities. They are available both within a Transaction Processing environment (in conjunction, for example, with TPMSX) and also in standard job environments such as Batch or MAC. This section describes the facilities.

## Transaction Management

### *Commitment Co-ordinator (COCO)*

The *Commitment Co-ordinator* (COCO) links the activities of a set of Resource Managers within a VM so that they can collectively perform *transactions* which have the required ACID properties. In terms of the X/Open Transaction Processing model, COCO performs the role of a Transaction Manager (TM).

For any particular transaction each transaction manager is either the *master* (the transaction manager which logs the commit point) or a *slave*. A slave transaction manager can *act* as a master to further slave transaction managers thereby creating a transaction tree whose root is the master. Several transaction managers can thus contribute to the co-ordination of a distributed transaction.

COCO provides the facilities to suspend a slave transaction when contact is lost with the master during the "in-doubt" period of two-phase commit and, if necessary, make a *heuristic decision*. When contact is re-established, the TM determines if incompatible heuristic decisions have been made and, if so, informs the work manager so that it can invoke the application to resolve the outcome of the transaction.

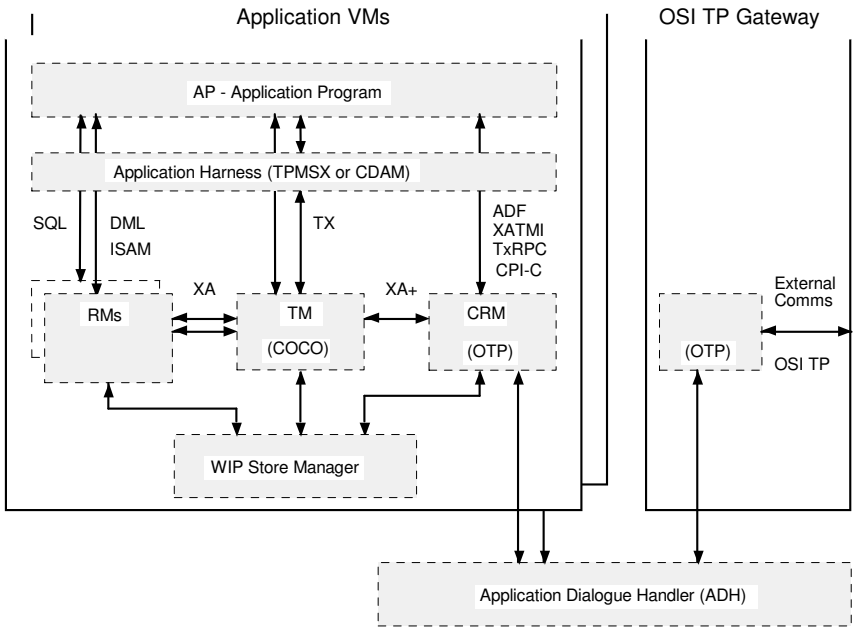
### *Work-In-Progress Store Manager (WSM)*

WIP Store Manager handles the data used to roll-back or recover transactions, sessions, databases etc. In effect, it handles the non-volatile data except for that managed directly by the Resource Managers themselves. WIP Store thus provides facilities for logging *Transaction Data* (transient, required only until the associated transaction has completed); *Partial Results* (required between transactions within a user session), *Assurance Data* (required permanently, for database recovery & auditing purposes), and miscellaneous Resource Manager and Transaction Manager log records. Whenever possible, WSM automatically combines several log items into a single write transfer to non-volatile storage media; this considerably reduces the overheads of securing critical data.

WSM may optionally exploit the use of virtual memory to provide a cache to reduce the number of read transfers from storage media. This feature is particularly valuable for storing partial results which need to be re-read when executing the next transaction within a session.

As the contents of the WIP Store are essential to the integrity of the applications and resource managers it supports, the non-volatile WIP Store is normally duplexed to minimise the risk of loss of data.

## Distributed Application Support




### *Application Dialogue Facility (ADF)*

The *Application Dialogue Facility* (ADF) provides APIs which allow one application to hold a dialogue with one or more other applications. The responding application may be in the same application server, in different application servers on the same system, or in different application servers on different systems. ADF may be used from CDAM or TPMSX work managers. The API to ADF is designed as a general peer-to-peer communications interface.

### *Other Application to Application APIs*

OpenVME provides support for several open application-to-application APIs. These include the XATMI (used in X/Open Distributed TP standards), CPI-C and TxRPC (used in OSF DCE). Functionally, the subsystems supporting these APIs are similar to the ADF subsystem and they exploit the underlying OSI TP services in a similar way.

### *OSI TP Handler (OTP)*

*OSI TP Handler* (OTP) is an OSI Application Entity which provides the OSI TP Application Layer service of OpenVME. The OSI TP protocol is used to communicate between distinct OSI TP Application Entities, for example on different application servers or systems; it uses the OSI Co-ordinated Commitment & Recovery (CCR) protocol as a carrier for distributed 2-phase commit and associated recovery.

OTP is used by ADF and by the subsystems supporting the XATMI, P2P and TxRPC APIs. Its main functions are enacting the OSI TP protocol state table transitions and performing protocol encoding & decoding. When a dialogue is co-ordinated, OTP is a Communications Resource Manager, interfacing to TM and it takes responsibility for transmission of two-phase commit protocol elements between application VMs in different application servers and/or systems.

There are two major components of OTP: one runs in-process in each VM using application dialogues; the other is used to handle external OSI TP communications. Communication between the in-process components of OTP in separate VMs on the same system is provided by the ADH subsystem.

There is one local OSI TP service per application server provided by OTP. The OSI TP Gateway executes within the OpenVME Communications Server Infrastructure, using the services of OTP, and comprises a *Scheduler VM*, a *Recovery VM* and an appropriate number of *Dialogue VMs*.

### *Application Dialogue Handler (ADH)*

The *Application Dialogue Handler* (ADH) provides an extremely efficient method of passing messages between VMs in a single OpenVME system.

ADH includes facilities for the definition of *routing tables* which are used to define the permissible routes between different applications, both within a system and external the system. Routing tables contain information about an applications server's own OSI TP service and about other, remote, OSI TP services with which the server communicates. ADH provides a name translation facility so that applications can transparently identify remote applications and ADH can determine their whereabouts on the network.

ADH also provides facilities for TPMSX services and CDAM applications to *listen* for incoming dialogues. These listening facilities are used in conjunction with the routing tables to identify the destination application for any particular application dialogue.

# The OpenVME TP Management System (TPMSX)

## Introduction

The OpenVME *Transaction Processing Management System* (TPMSX) provides a complete open processing environment for TP applications so that they can be managed in a controlled and reliable way. This processing environment provides the capabilities necessary to support high throughput systems, with many users, high data volumes and fast response time, with transactions processed safely and securely.

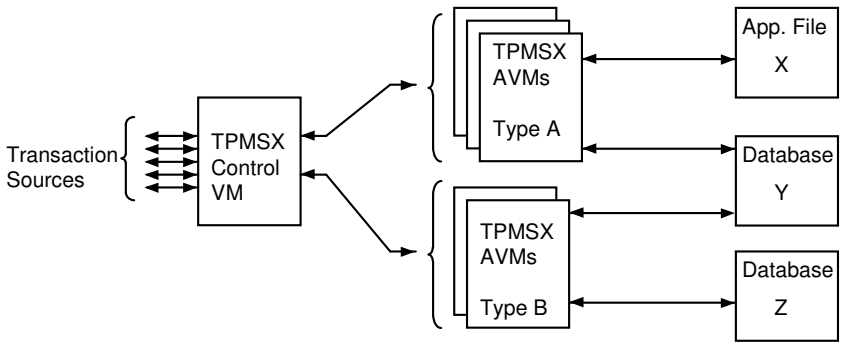
TPMSX conforms to the X/Open TP model and a set of X/Open standard APIs are supported as well as the OSI TP standards for communication between transaction managers in a distributed TP system.

TPMSX makes extensive use of the underlying OpenVME transaction management facilities described in the previous section.

## The Structure of a TPMSX service

A TPMSX service is constructed from several components, including:

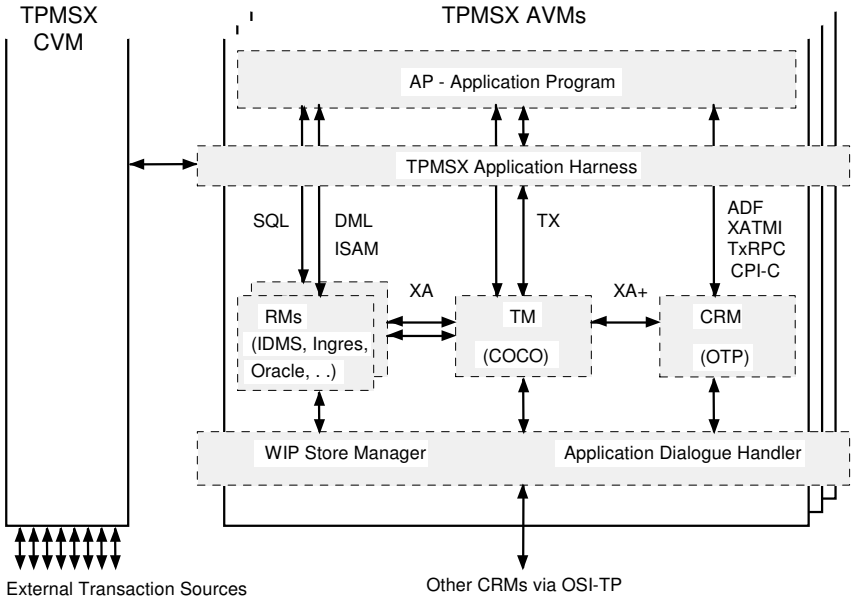
- A single *Control VM* (CVM) which queues & schedules messages for the Application VM(s);
- One or more *Application VMs* (AVMs) containing user-written application modules;
- A TPMSX *Spooler VM* which handles printing of documents;
- The TPMSX *Auxiliary VM* (XVM);
- Shared access to Resource Managers, including TPMSX itself.



In normal operation, all messages are initially handled by the CVM. Code in the CVM (optionally user-written) analyses the *message type* and determines the *AVM Type* required to handle the message; the message is then queued until an AVM of the correct type is available. There may be several AVMs of a particular type, allowing several similar messages to be processed concurrently. When an appropriate AVM is available, it is notified by the CVM that there is a message queued for processing by that AVM. The AVM retrieves the message from the queue and processes it, passing a final reply back to the CVM; it then becomes available again to handle a another message.

TPMSX provides a means of saving data, known as *partial results*, between successive transactions from a particular source; this data is stored, in the WIP Store, together with other recovery data generated to enable the TP service to be restarted tidily and co-ordinated with RMs and CRMs after a service break.

Transaction demarcation, commitment management and, in general, access to Resource Managers take place in the AVM, "in-process". In terms of the X/Open model each Application VM is provided with Virtual Resources corresponding to a Transaction Manager and such Resource Managers as it requires to access. One of these Resource Managers is TPMS itself, which provides managed interfaces to a work-in-progress store (WIP store), for logging temporary information, and to spooling and input-output resources.




The diagram above shows how the various functions supporting TPMS applications are supported within a TP service. It also identifies VME & TPMS subsystems with the elements of the X/Open transaction management architecture; these are described in more detail below.

### *The TPMSX Control VM (CVM)*

The CVM has the following functions:

- Control of "dumb" terminals & other transactions sources;
- Output message formatting, incorporating the required template;
- Input message analysis and routing to a suitable AVM;
- Control of service start-up (including recovery) & close-down;
- Control of AVM creation, concurrency etc. according to service requirements;
- Statistics gathering & message logging.

The characterisation of a CVM for a particular TP service is achieved by the use of a *parameter module*, generated when the service is created or updated. User-written procedures may also be incorporated into the CVM at certain points - *e.g.* for message validation and analysis or for special output processing.

Message analysis and the consequent routing of a message to a suitable AVM can be handled in one of several ways:

- based on knowledge of the screen template currently displayed on the terminal;
- by direct application control of a pre-determined sequence of interactions;
- based on a *message key*, usually the first few characters of a message;
- by the terminal user invoking an *action key*;
- by invoking a user-written *message analyser* program, incorporated in the CVM;
- based on OSI TP *routing tables*;
- by treating the message as a service-defined default message type.

Once a message has been analysed it is placed in a shared (global) data area. The CVM selects an AVM of the appropriate AVM type from a pool of similar AVMs, and a global flag event is caused to notify the selected AVM. The AVM can subsequently retrieve the message from the global data area.

### *TPMSX Application VMs (AVMs)*

AVMs are the environments in which the majority of the application-specific work is performed by the execution of user-written application modules. Identical AVMs are grouped into sets known as *AVM types*, which provide a basis for the allocation of system resources and for optimising performance. There may be many instances of an AVM type in a TP service, thus allowing concurrent processing of similar messages from several transaction source. The CVM maintains *pools* of AVMs of each type; the number of AVMs in a pool determines the concurrency of message processing within that AVM type.

An AVM is dedicated to handling a particular transaction until the transaction is completed. Successive transactions allocated to an AVM result in the *serial re-use* of that AVM. In the course of processing a message, the application may:

- access Resource Managers to manipulate conventional files (*e.g.* via RECMAN) or database information (*e.g.* IDMSX or a relational database);
- output intermediate replies to the transaction source (as *unsolicited output*);
- generate a *final reply* to the transaction source, to be output on successful completion of the transaction;
- generate new transactions which may be started immediately, after a specified interval, or periodically; this allows different aspects of the processing of a single message to be handled by different applications;
- interact with other applications by means of a *dialogue*, using the OpenVME distributed application facilities. In this way, a *single transaction* can be executed across several applications, in the same or different TP service and/or system.

Note that the two methods of initiating further work - generating new transactions and starting a dialogue - are significantly different. In the former case, there is no co-ordination between the effects of separate transactions and any recovery must be performed by executing a *compensating transaction*. In the latter case the effects of all applications are associated with a single transaction and are therefore automatically co-ordinated.

To indicate completion of a transaction, a *final reply* is generated by the application. There may be further user-written processing of this message in the CVM and it may be merged with a screen template and then, finally, it is transmitted to the terminal.

### *The TPMSX Spooler VM*

The TPMSX spooler is a special AVM type which manages most of the printing for a TP service. Spooler functionality has an in-process component which enables AVMs to place *documents* in a *folder file* for subsequent printing by the Spooler AVM. A folder file is thus a resource for which the in-process component of Spooler is the responsible Resource Manager. The out-of-process component of Spooler, the Spooler VM, handles all aspects of printing to the printers connected to it, including removing documents from folder files, printing them and handling exceptions and recovery across service breaks or failures.

In some circumstances it may be necessary to print some (usually small) document during the processing of a transaction. In such cases, the document is output directly to the printer under the control of the CVM.

#### *The TPMSX Auxiliary VM*

The *TPMSX Auxiliary VM* (XVM) is used to handle asynchronous actions for the TP service such as spooler recovery and transaction meter logging to SMF.

### The Distributed Transaction Processing System (DTS)

TPMSX provides an integrated scheme which supports a limited form of queued transactions between TP applications. This is known as the *Distributed Transaction Processing System* (DTS). An application may send a message to another application; successful transmission of this message is necessary before completion of the transaction. The destination TPMSX service queues incoming messages until the receiving application is ready to receive the message; the message is not finally removed from the queue until processing of the message by the receiving application is complete.

## Co-ordinated & Distributed Application Manager (CDAM )

### Introduction

The OpenVME transaction management facilities may be exploited in non-TP environments. A single application may therefore use several Resource Managers in a co-ordinated manner; equally, several applications, in the same or different systems, may co-operate using the distributed application facilities of OpenVME; optionally these facilities may be used to perform co-ordinated transactions across the applications.

This is particularly relevant for distributed or client-server systems for which there are several models of distribution, most of which can be used *in conjunction* with a transactional style. Indeed, transactions, whether formally supported or not, are an essential feature of almost any (possibly distributed) system in requirements such as data integrity, reliability and resilience are important.

This section describes the OpenVME Co-ordinated & Distributed Application Manager (CDAM). The facilities provided by CDAM allow the construction of non-TPMSX applications which are able to co-ordinate updates across one or more Resource Managers. One of the RMs may be OTP (accessed via ADF or one of the open application-to application APIs), in which case remote RMs can be included in distributed application co-ordination.

### CDAM Concepts

A *CDAM Service* consists of a set of catalogued objects representing a Resource Manager which is part of an application server environment. CDAM applications can be run in any type of VM, usually MAC or Batch, within the

scope of a *CDAM Run*. A CDAM Run consists of one or more applications each of which can optionally contain one or more transactions. During a CDAM Run, which is performed in the context of a CDAM service, the VM is *bound* to the application server and temporarily becomes one of its resources; CDAM thus provides a work-environment corresponding to the nominated CDAM service.

CDAM applications which make use of ADF or one of the open application-to-application APIs can be either initiators or responders and can participate in dialogues with other CDAM applications, TPMSX applications or non-VME applications.

A CDAM service also has a catalogued recovery task which provides the ability to recover CDAM runs, under the direction of the transaction manager, in the case of VM loss, system loss or communication failure. The recovery task is executed in a separate System Task VM which can either be started automatically, on system load, or manually when required.

Certain areas of application data may be declared as *Partial Results.* Partial Results may be data areas within the application program or may be SCL job-space variables. Partial Results data is automatically saved by CDAM at the end of each transaction so that if the transaction is rolled back, or the job is restarted after a break, the data is restored to its state at the start of the transaction.

A CDAM Run can be divided into a number of *CDAM Work Units*. These typically correspond to separate application programs and they can be used to scope Partial Results. This allows different phases of a CDAM Run to use different Partial Results. In addition, it is possible to have Partial Results scoped to the whole CDAM Run and these are typically used to allow a the run to re-start at the failed Work Unit.

Facilities are provided to allow CDAM responder applications to save *Compensating Transaction Data* (CT Data) in the event of a heuristic decision being taken because of communications failure during two-phase commit. If, when communications are restored, it is found that the heuristic decision was incorrect, the CT Data is used to run, automatically, a user written *Compensating Application* to resolve the *heuristic mix.*

# Chapter 8
# Information Management

## Introduction

OpenVME provides several information management services catering for a wide range of information models and the corresponding information storage & retrieval services. These services include:

- Relational Databases which support open standard SQL services;
- A high performance Codasyl database, IDMSX;
- record-based file services (supporting sequential, indexed etc. organisations);
- flat-file services (including fully X/Open conformant filesystems).

This chapter describes how the fundamental architectural components of OpenVME are used to provide information management services.

### Information Models

#### *Unstructured Files*

An unstructured file is structured as a contiguous sequence of bytes. Operations are provided to:

- read or write any contiguous sequence of bytes within the file;
- extend or truncate the size of the file;
- lock any contiguous sequence of bytes within the file;

#### *Record-based Files*

Record based (flat) files comprise a set of records accessed either sequentially or randomly using a key. The internal record structure is entirely application-defined. File organisations supported include: Sequential, Indexed Sequential and Hashed Random. Operations are provided to:

- select a record by key, relative to the current record or absolutely
- read or update the selected record;
- insert new records or destroy existing records;
- lock the selected record.

#### *The Codasyl (Network) Model*

The Codasyl model provides records and allows relationships to be defined between them to be defined using *set types*. A set is a one-to-many relationship

between records: one owner and many members. A record may participate in several set types either as owner or as a member. Records are manipulated using *Data Manipulation Language* (DML) operations. Access is *navigational* in that an application selects one record at a time and may then select another record by key value or by navigating set relationships. Operations are provided to:

- select a record by key value or by navigating from owners to members or vice versa;
- read or update the selected record;
- insert new records or delete existing records;
- connect records into sets or remove them from sets.

The description of information in the database is essentially static.

### *The Relational Model*

The Relational model organises a database as a set of *relations*; each relation is a set of *tuples* (records) with common structure. Each relation has a *primary key* whose value identifies a row uniquely. Relationships between relations are represented by including the primary key of one relation as a *foreign key* in another. In the relational model, operations are performed on sets of tuples and produce sets of tuples as results; thus the operations can be combined to provide more powerful operations. The basic operations, based on the Relational Algebra and generally expressed in *Structured Query Language* (SQL), include:

- *Selection* of tuples according to specified criteria;
- *Projection* (mapping) of tuples;
- *Join* of relations (cartesian product);
- *Update* of selected tuples.

Dynamic creation of relations is fundamental to the operational model and so the description of information in a database (the relational schema) is also dynamic. Features such as *Trigger*s, *Integrity Constraint*s and *Object Support* supplement the basic model.

The *declarative* nature of SQL (expressing *what* is to be performed rather than *how*) allows a wide variety of underlying implementations.

### *Object-Oriented Models*

Object-oriented models are a synthesis of Object-Oriented programming and database ideas using the concept of active objects that encapsulate stored information within themselves, allowing access to it only via a defined procedural interface. Recent developments have supplemented the basic model with the concept of *bulk data types* which formalise and simplify the handling of collections of objects.

There are several development approaches:

- the extension of relational databases to support abstract data types, rules and objects;
- persistent versions of object-oriented languages such as C++ and Smalltalk;
- fully object-oriented databases.

In the short term only the first is expected to have a significant impact in commercial applications.

## Codasyl Database - IDMSX

IDMSX is the OpenVME network database management system. It essentially conforms to the international CODASYL standard, which is also supported by other vendors on other platforms. IDMSX provides efficient access to information shared by many users who may have different views of that information. It performs all access to and manipulation of data and is responsible for data security, data integrity and automatic recovery.

Applications are written with embedded Data Manipulation Language (DML) statements. DML statements are translated during compilation into direct calls on underlying IDMSX interfaces. The in-process manner of invoking IDMSX results in very efficient database access.

The structure of an IDMSX database is defined in terms of *schema* and *sub-schema.* Schema describe the logical structure of an entire database and sub-schema describe subsets of the database available to particular application programs. Comprehensive facilities are provided to enable the optimal mapping of logical schema onto a physical storage structure. Data definitions, schema and sub-schema can all be stored in the Data Dictionary System.

The major features of IDMSX are:

- Data is structured internally so that the database can be shared by different users for different purposes, each requiring different logical views of the same information.
- Data is stored in an efficient way to minimise disc accesses. For each record type random, sequential or clustered physical placement may be chosen allowing the database to be optimised for a particular pattern of usage. Placement is independent of logical organisation.
- IDMSX provides a powerful indexing system which supports any number of alternative access keys. It also supports *indexed sets* which provide optimum performance when individual records are required to be selected from large groups.
- Data can be stored in a form that is CAFS-searchable; CAFS greatly improves the speed of access where the search key is not a database key.
- IDMSX supports automatic recovery from program or storage failures. This involves two techniques: *delayed update* which only updates the database when a task has been successfully completed; and *rollback*, which uses a journal of *before-looks* to restore the state of the database if a failure occurs before a task is completed. Rollbacks are not applicable if the IDMSX service is using the recovery facilities of the Open TP environment.
- Main store areas can be used to *cache* selected IDMSX data thus reducing the number of accesses to disc. Cache size may be fixed or dynamic.
- IDMSX supports the X/Open model of transaction management; in particular, it interfaces with the OpenVME transaction management system using an optimised interface which is functionally equivalent to the X/Open XA standard.
- A comprehensive set of management tools is provided to assist with archiving, recovery, database restructuring and physical re-organisation.

The performance of IDMSX is 2 to 3 times better than that of a typical relational databases, when comparing like with like. When a relational database is used without the benefits of a transaction processing work manager, or with front end tool sets providing advanced usability features, then the ratio is even higher. These ratios between non-relational and relational database performance apply to products across the whole industry and are not specific to ICL.

### *The Architecture of an IDMSX Service*

An IDMSX Database Service comprises:

- A description of the service in the Catalogue defining certain attributes and other catalogued objects which are associated with the service;
- The *IDMSX Database* files which hold the database itself;
- The *IDMSX Journal* files which hold copies of changed records when the database is being updated; the files enable recovery from failures in individual applications or the whole system. Journal files are as follows:

  The Central Journal which contains after-images of changed records;

  Quick-before-looks (QBL) files each of which contains before-images of records changed in one transaction (not used in the Open TP environment);

  Area journals which contain after-images of records changed in one or more areas of the database.

- An *IDMSX Diary* file in which all major events and activities on the database are recorded. Examples include starting and ending a database service, and starting and ending utility operations on the database. The database diary has several functions:

  To provide a record of database use;

  To record the names of the database and journal files of the service;

  To check whether applications and utilities have completed their operations tidily;

  and, unless the IDMSX service is using the recovery facilities of the Open TP environment:

  To control the use of QBL files in recovering the database after a failure; the names of the QBL files are recorded in the diary;

  To record the names of TP services accessing the database in order to co-ordinate recovery with them on a service restart;

- Miscellaneous files including dump files, workfiles etc.

### *IDMSX Run-time Structure*

At run-time the IDMSX database service is realised as:

- An IDMSX Service VM which controls the database service;
- Areas of global virtual store known as *service tables* which define:

  Semaphores & locks to control shared access to the database;

  Information on the storage of IDMS areas in physical files;

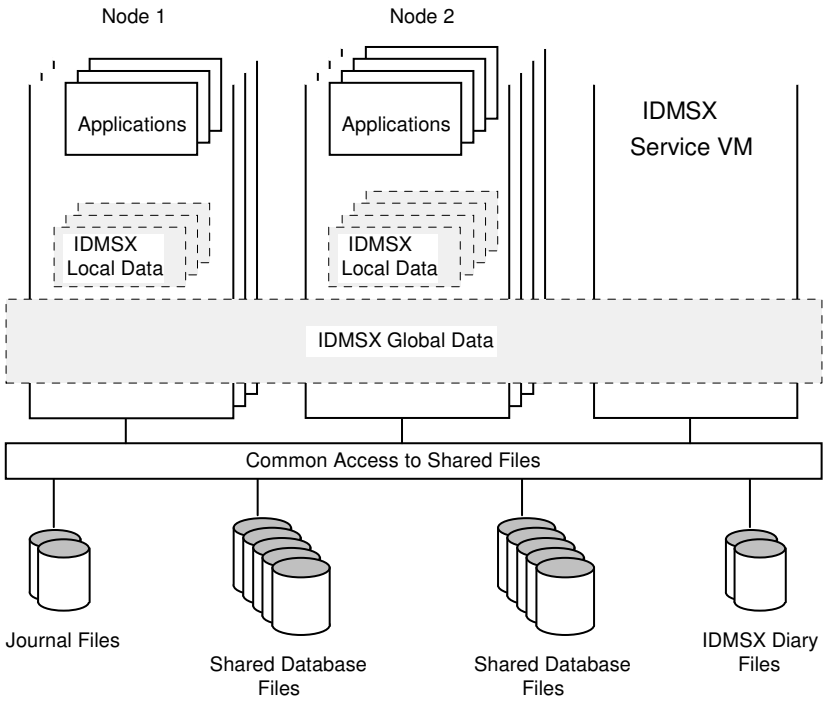
  Buffers for page (block) transfers;

  Workspace for pages defined as being held in virtual memory;

  Workspace for caching selected areas of the database;

  Work space to accumulate service statistics.

- Application VMs which use the database service. Each AVM has local data supporting the Virtual Resources which provide in-process access to the database. Shared Virtual Resources which co-ordinate the concurrent access to the database by several VMs are supported by the service tables described above. Files of the database service is assigned locally within each AVM as required enabling common shared access to the files from all AVMs.



### *Use of IDMSX Database Services*

An IDMSX service can be used by an Application VM running in almost any work environment, including Batch, MAC or TP.

An AVM may only access a single IDMSX database service at a time. However a TP service can use several IDMSX database services by using separate AVMs to access each service. This technique, in conjunction with the co-ordination features of the OpenVME transaction management system, can be used to allow a single transaction, spanning several AVMs, to operate on several databases.

A single AVM may access an IDMSX database service and any number of RECMAN files; it may also access one or more relational database (see later).

Several TP services may use a single database service concurrently. This technique allows several different applications to access a single database. It can also be used to allow extremely large numbers of terminals (exceeding the normal TPMSX limits) to access an IDMSX database by running multiple instances of the same TPMSX service, all sharing the same IDMSX service.

### *IDMSX Recovery Facilities*

Dump and restore utilities allow a security copy of the database to be made. Dumps may take place concurrently with normal operation.

Roll-forward and roll-back utilities are provided to process IDMSX journals. The roll-forward utility may be used to process a restored portion of the database by writing after-images from the journal file. The roll-back utility (only applicable if the IDMSX service is not using the recovery facilities of the Open TP environment) can be used to restore the database to a previous state (*e.g.* prior to some error) by writing before-images to it.

## Relational Databases

A choice of Relational database management systems (RDBMSs) is provided by OpenVME. An SQL service conforming to open standards is supplied by INGRES, Oracle and Informix. There are generally two major run-time components of a RDBMS:

- a front-end component, an instance of which is usually co-located with each application;
- a back-end component - the database server - which actually performs data manipulation operations on the database and to which shared access is made by all front-end components.

Applications either generate SQL explicitly or are written in a programming language which allows embedded SQL statements. In the latter case, a source pre-processor converts the embedded SQL into calls on the underlying APIs.

The structure of a database is defined using a Data Description Language (DDL). Relational databases additionally allow the dynamic creation of new tables (relations) although such tables may not, subsequently, be accessed as efficiently as those that are pre-defined.

## Oracle RDBMS

Oracle is a Relational DBMS supporting standard (and extended) SQL, networked SQL access, distributed database operation and a range of front-end tools and application development aids.

Oracle uses a front-end / back-end architecture in which client applications (front-ends) access a shared database server (back-end). SQL*Net is the Oracle proprietary mechanism used for linking front-end and back-end components. It supports a two-phase commit capability allowing distributed transactions to be performed across a number of servers. Front-end and back-end components may be co-located in which case communication between them can be optimised by the use, for example, of shared memory.

Each *instance* of an Oracle database server comprises one or more query execution engines which have common access to the database discs and communicate with each other via an area of shared memory known as the *Shared Global Area.* A *Lock Manager* co-ordinates resource sharing between execution engines, allowing them to synchronise access to resources such as data and peripheral devices.

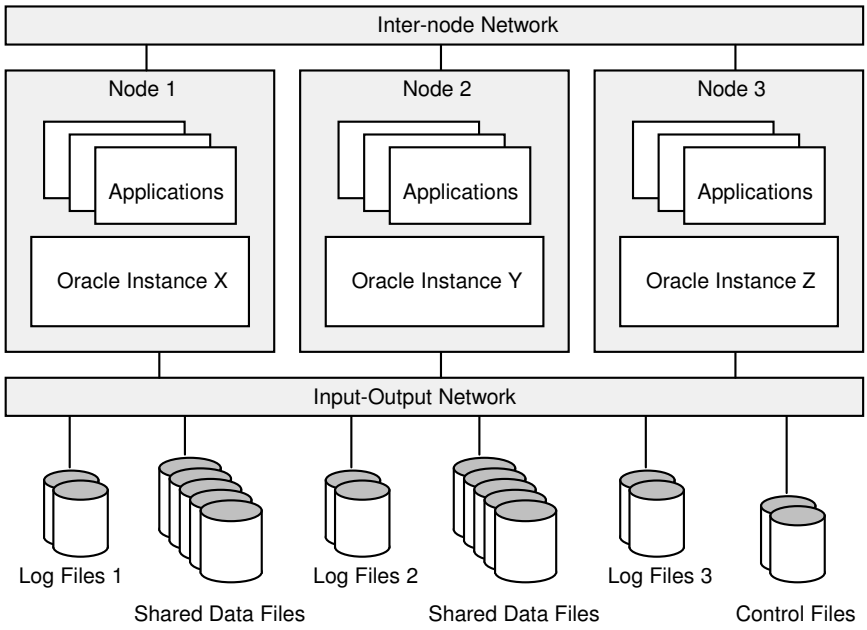
The lock manager performs the following services for applications:

- keeps track of current "ownership" of a resource;
- accepts requests for resources from applications;
- notifies the requesting application when a resource is available;
- allows an application to gain exclusive access to a resource.

### *Multi-node Operation*

Multi-node operation of the Oracle database server is based on a *shared disc* architecture. This allows an efficient multi-node implementation by minimising contention on shared memory areas whilst exploiting the uniform access available from each node to all database discs. In addition, a *Distributed Lock Manager* (DLM) provides a means for instances of Oracle on different nodes to communicate with each other and co-ordinate modifications of data on the shared discs. This is achieved by allocating distributed locks - *Parallel Cache Management* (PCM) locks - to data blocks and then using DLM facilities to control *ownership* by instances of data blocks.

Parallel cache management is used to ensure cache coherency between caches in different Oracle instances. Within a data block, transactional locking (down to row level) is performed entirely within the instance which owns the data block, thus reducing communications between instances.




The loosely coupled, shared disc architecture has the following benefits:

- high performance through the efficient exploitation of multiple nodes;
- incremental performance growth by adding additional nodes;
- high availability through hardware redundancy.

Oracle has a number of features which support high performance operation:

- Each Oracle instance has a cache in which data blocks are stored to reduce disc accesses.
- Oracle supports *fast commits* in which a transaction is committed by writing to a serial log file rather than by several random updates to the database files. Each instance of Oracle has its own independent log file.
- Oracle supports *group commits* which allow several transactions to be committed by a single write to the log file, thus reducing the effects of log file latency.
- Since an update is committed to the log file, writes may be *deferred* by only updating the Oracle cache.

- Oracle supports row-level locking. This fine-grain locking within a data block reduces the probability of a clash between different applications attempting to access the same resource (*c.f.* a whole data block).
- Oracle (from version 7.1) supports parallel query execution, index creation and data loading.
- On-line back-up and archiving are supported.

Oracle has several other important features:

- Oracle is a Resource Manager, providing the interfaces required by the X/Open distributed TP architecture. Oracle takes full advantage of OpenVME transaction management facilities.
- Applications using Oracle, running in the Open TP environment, can perform fully co-ordinated distributed transactions.
- Oracle Forms enables form-based applications to be developed which run under TPMSX.

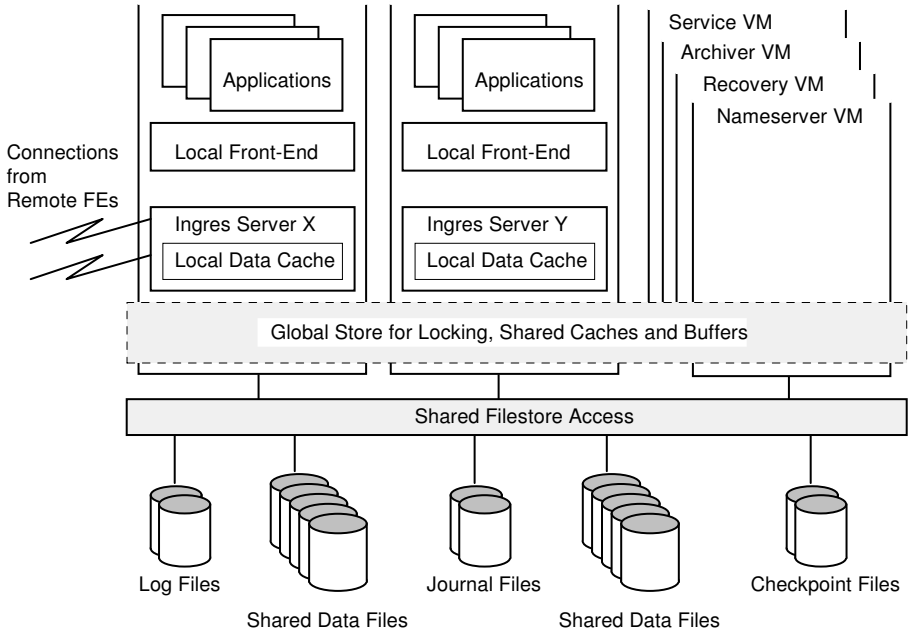
## INGRES Database

INGRES is a set of components providing an integrated database and application development tools. It provides 4GL and 3GL application building tools for both character and Windows environments. It has a sophisticated Query Optimiser to optimise query execution time and has a full range of system administration facilities.

The major run-time server software component is INGRES BASE. INGRES/NET allows an INGRES client application to access a remote INGRES database server. INGRES/Star allows multiple INGRES databases to be accessed as though they were a single database.

Additional tools and components provide application development and user interface capabilities:

- INGRES C and COBOL pre-compilers allow SQL statements to be embedded in programs;
- Report by Forms (RBF), Query by Forms (QBF) & Application by Forms (ABF);
- INGRES/Windows4GL: an O-O programming tool for building applications with GUIs.




The run-time components of an Ingres service include:

- Ingres *Front-End* processes which can be local to the OpenVME system (*e.g.* when an OpenVME application is using the Ingres service directly) or remote (*e.g.* when a PC application is accessing the Ingres service remotely). Front-end processes can be co-located in the same VM as back-end server processes.
- Ingres *Back-End* server processes which support access to the Ingres databases. Each server process is multi-threading, supporting multiple front-end processes and various Ingres system threads. Each back-end server processes maintains a local data cache using a value-block-locking scheme. The server processes have components at Access Levels 9 and 10 to ensure that applications, running at Access Level 10 or above cannot gain unauthorised access to global store areas (at Access Level 9).
- The Ingres Service VM which controls the other VMs associated with the Ingres service.
- An Archiver VM which extracts committed updates from the Log file and writes them to individual database journals;
- A Recovery VM which is used to restore a consistent database state after a service failure;
- A Name-server VM which is used by front-end processes as a directory to identify the route to the required Ingres server.

The Ingres database has several features allowing it to provide a high throughput, high reliability relational database service:

- A two-phase commit mechanism ensures data integrity for distributed applications;
- Each Ingres server has a local cache in which data blocks are stored to reduce disc accesses; a shared cache further reduces disc traffic when data is accessed by multiple servers.
- Ingres supports *fast commits* in which a transaction is committed by writing to a serial log file rather than by several random updates to the database files.
- Ingres supports *group commits* which allow several transactions to be committed by a single write to the log file, thus reducing the effects of log file latency.
- Since an update is committed to the log file, writes may be *deferred* by only updating the Ingres cache.
- Each Ingres server VM is fully multi-threaded, reducing VM switching overheads and memory occupancy;
- Ingres can exploit the Series 39 CAFS search accelerator in a totally transparent manner, giving large performance improvements for many complex queries;
- On-line back-up and archiving are supported.

Ingres has several other important features:

- Ingres database definitions can be held in the OpenVME Data Dictionary System (DDS) and a database can be automatically generated from the definitions;
- Ingres applications can be generated by Application Master which also includes facilities for database transition from IDMSX;
- Ingres ESQL/COBOL applications can be run in a TPMSX environment;
- Ingres is well integrated with OpenVME System Management mechanisms including operational, distribution and capacity management.

### INFORMIX Database

INFORMIX is a Relational Database Manager, with associated development and run-time tools, supporting X/Open conformant SQL. It is designed for Unix systems and therefore executes within the VME-X environment. INFORMIX is also the source of the C-ISAM product which supports the X/Open ISAM standard.

The major run-time server component is INFORMIX OnLine, a multi-media, high-performance database server. It is intended for OLTP environments and has full capabilities for database recovery and maintaining data integrity after a failure.

INFORMIX-TP/XA is an additional option linking INFORMIX-OnLine to X/Open XA-compliant transaction managers and thus allows it to act as a Resource Manager for an XA-compliant TP environment such as TPMSX. INFORMIX-TP/ToolKit is a set of library functions which can be used to create OLTP applications.

A variety of additional tools provide application development, interactive query and distributed database capabilities:

- INFORMIX-SQL: end-user interactive SQL tool;
- INFORMIX-ESQL/C: allows SQL statements to be embedded in a C program;
- INFORMIX-ESQL/COBOL: allows SQL statements to be embedded in a COBOL program;
- INFORMIX-4GL: an application development tool for generating database applications;
- INFORMIX-STAR: used to control a distributed database application.

INFORMIX OnLine uses an optimised device driver to gain efficient direct access to VME files, thereby achieving excellent performance.

## Object Database

The OpenVME architecture is extensible and thus allows for alternative databases such as object databases (for example the Fujitsu ODBII database) to be supported in the future.

## Use of the CAFS Information Search Processor

The CAFS-ISP is a hardware search accelerator which searches data as it is read from the disc for records which match user-defined search criteria. Records which match the search criteria are returned for further processing; other records are discarded. CAFS allows large volumes of data to be searched rapidly according to complex criteria. Search criteria can be numeric, alpha/numeric or straight text, and can involve:

- multiple criteria, combined with AND and OR operators, in each enquiry;
- precise or fuzzy matching (using wildcards);
- different types of questions: *e.g.* satisfying any three out of five selected criteria;
- a combination of the above.

CAFS is ideally suited to provide ad hoc enquiry facilities to new and existing applications, and for operating as a free text retrieval system. Interfaces are provided to allow application to use CAFS facilities directly, including:

- The *Direct CAFS Interface* (DCI), accessible, for example, from C, FORTRAN & SCL;
- The *Relational CAFS Interface (RCI)* allowing COBOL programs to operate on "logical files" which are relational views of IDMSX or RECMAN files, supported by CAFS;
- The *CAFS Search Option* which allows COBOL programs which perform serial record processing to use CAFS searches to select only the required records.

In addition, Ingres can use CAFS facilities to improve performance in a manner which is entirely transparent to database applications.

## Flat Files

### *File Description & Creation*

OpenVME supports various types of flat file including sequential, indexed sequential and alternate key record organisations as well as block organisations.

Every file has an associated *file description* stored in the Catalogue, defining the physical and logical characteristics of the file and the data in it. Standard file descriptions are provided for common file organisations and new descriptions can be created, optionally based on an existing description.

Each file organisation is supported by a specific *Record Access Mechanism* (RAM) type which provides the required mappings between the logical file organisation and its corresponding physical block organisation.

A new file is created by specifying a hierarchic name for the file (within the Catalogue), a file description and, optionally, an area of physical filestore in which the file data is stored. The information describing the file is stored in the Catalogue.

### *File and Record Access*

As with most other catalogued objects, there are two phases of file usage:

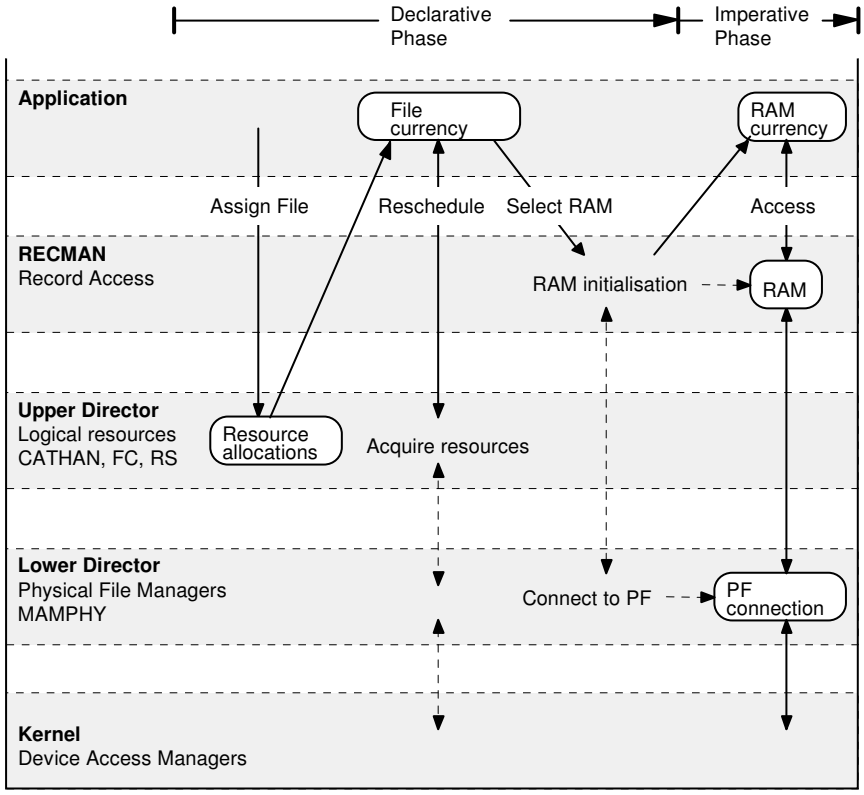
- The *declarative* phase during which the file is selected and the Virtual Resources for the specific file usage are created within the VM;
- The *imperative* phase during which accesses to data in the file take place.

### *The Declarative Phase*

During the declarative phase, the catalogue object describing the file is selected, establishing a currency for the file. The action of selection causes Virtual Resources to be established within various subsystems including File Controller, Resource Scheduler and, for Libraryfiles, Library Controller. Currencies returned by RS for the underlying physical resources are associated in FC with the file currency.

Having selected the file a resource allocation reschedule is performed to acquire access to use the physical resources associated with the file. For example, if accessing a magnetic tape file, it is at this point that the tape volume would be mounted.

The final action in the declarative phase is to select a Record Access Mechanism. Each usage of a file causes a Virtual Resource, an instance of the appropriate RAM type (determined by the file description in the catalogue), to be created. Data associated with the file usage is stored in the RAM. During its initialisation, the RAM requests the appropriate Physical File Manager (PFM) - *e.g.* MAMPHY for magnetic files - to create a Virtual Resource providing access to the underlying physical file. The PFM returns a *PFM currency* which provides an efficient route by which the RAM can subsequently invoke operations on the PFM Virtual Resource.




Main paths involved in declarative and imperative phases of file access.

### *The Imperative Phase*

During the imperative phase, operations on a file are performed by invoking the RAM via one of two procedural interfaces made available when the RAM was created. One of these allows additional parameters to be passed specifying, for example, a position in the file, an action (read, write *etc.*) or a buffer address. The other is used when no parameters need be passed - for example, when reading the next record of a serial file.

During this phase, the RAM invokes MAMPHY directly using the PFM currency returned during RAM selection. This provides a highly efficient route to the PFM and bypasses all the Upper Director subsystems which were involved during the declarative phase.

The internal structure of RECMAN itself is described in detail in Chapter 5.

### *Record Access Principles*

The action of a RAM is controlled by a set of *RAM parameters* stored in the RAM. The parameters are initialised when the RAM is selected and many can be changed by subsequent calls to the RAM. The most important RAM parameters are:

- Position: a pointer into the file contents;
- Record and Key buffer references;
- Imperative Action (*e.g.* select, read, write, destroy);
- Displacement of the record relative to the last accessed (next, previous or same).

The default invocation of a RAM uses stored RAM parameters to perform a similar action to that performed on the previous invocation. For example, to read records from a serial file, the Position would be initialised to "just before first" (record), the Displacement to "next" and the Imperative Action to "select and read". Successive calls would select the next record and read it into the specified record buffer. The storage of parameters in this way minimises the number of parameters that have to be passed and validated on each invocation of the RAM, helping to ensure that record access is as efficient as possible.

## Interchange Between Information Management Services

### Database Definition Interchange

The same business model within DDS that was used to generate schema for an IDMSX database can be used to generate DDL for a relational one.

### Bulk Data Interchange

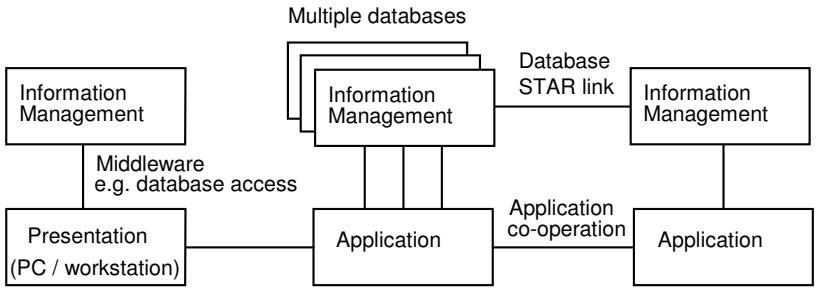
It is possible to move bulk data between databases (*e.g.* from IDMSX to INGRES) on a regular basis and then use specific tools to access (read) the data. This is particularly relevant where stable data is required for analysis purposes.

### Accessing Multiple Information Management Services

A single application can concurrently access IDMSX, INGRES, Oracle, RECMAN, etc., under TPMSX, MAC or Batch work environments. Thus new data could be stored in a relational database (say) and the application extended to access it together with IDMSX data. The OpenVME transaction management facilities give automatic co-ordinated update when more than one database is involved in a single transaction.

Where one of the databases is on a different system there are several possible models of distribution.

- The OpenVME transaction management facilities allow multiple applications, each accessing one or more local databases, to co-operate in a co-ordinated manner using standard open interfaces (*e.g.* XATMI).
- The relational databases provide a STAR capability whereby a local instance of the RDBMS provides a coherent view of a distributed set of (identical) RDBMS instances. The distribution is invisible to the application. Co-ordination of updates is effected by proprietary mechanisms.
- For PCs, middleware can be used to link applications designed to access local databases to remote (usually relational) databases.



## Client-server Access to Information Management Services

In the context of information management, the relevant forms of Client-server distribution are Remote Data Management and Distributed Data Management. In general both of these are restricted to the use of databases accessed via SQL. Note that Review allows SQL access to IDMSX data, thus making an IDMSX database remotely accessible using the same mechanisms and client APIs as for a relational database.

SQL has several major benefits for Client-server systems:

- there is a defined standard for SQL itself (although most vendors support optimised proprietary extensions to this standard);
- the granularity of SQL queries is such that they can be remotely executed in a distributed system without networking considerations dominating performance;
- many major PC software packages are able to generate SQL queries to access information held on a remote (shared) relational database; standard APIs and *middleware* facilitate this method of access.

A major characteristic of SQL is that it allows queries to be made which are not pre-determined. This is extremely powerful: if the required information is modelled in the database then it can be extracted, immediately, without the need for any user application.

### Distributed Data Management

Oracle, INGRES and Informix all supply STAR products which allow multiple instances of a RDBMS to be linked so that from the application only a single, integrated database is visible.

The unique benefit of distributed database technology is that it provides the ability to perform queries that find the data wherever it is held, if necessary by combining data from several databases. However the performance aspects of distributed database services mean that they can be used for MIS work but that distributed transaction processing should be used for high throughput applications.

When distributed data management is used by applications, the data types and layouts in the databases need to be known by the applications. Therefore there may be difficulties of management where the application components are under separate ownership or operational control, or have different technical origins or use different infrastructures.

## Remote Data Management

One method of achieving access to remote databases with relational tool-sets is to use the relational/NET products which are enable client applications to access remote database servers. Protocol proprietary to the database vendor is generally used for such access.

OpenVME supports interworking from applications and many standard PC packages such as spreadsheets by the use of middleware products such as TechGnosis's SequeLink, and Information Builders' EDA/SQL.

Such middleware products provide a mechanism for connecting from PC (etc.) packages, such as spreadsheets to databases, so that corporate data can be downloaded to them, using facilities (menus) within the packages themselves. Typical integration features include:

- Connection *from*: spreadsheets, generic database front-ends, Visual Basic etc.;
- *on* many client environments including Microsoft Windows on a PC;
- *via*: a wide range of LAN communications protocol stacks;
- *to*: a wide range of relational database management systems;
- *on* environments: OpenVME, UNIX, MVS and others;
- allowing access to all OpenVME data sources via ReView.

### *SQL access to IDMSX and other VME data sources (Review)*

SQL access to IDMSX data is enabled by the ReView facility which provides relational views of IDMSX data. Full read access is available, treating these views as though they were relational tables. This gives a common user interface across INGRES, IDMSX and other VME data sources.

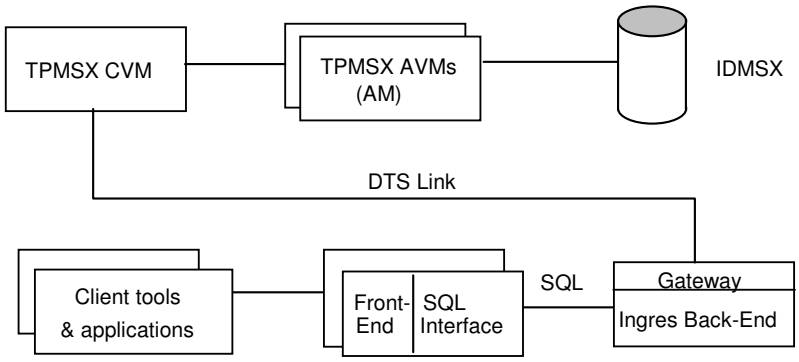
The full read access extends to interactive SQL, SQL embedded in C and COBOL, all INGRES tools and all database-independent tools which interface to INGRES and any other vendor's tools which access INGRES databases. The applications and tools may be run on any environment on which they are supported and which can network through to VME using INGRES/NET - PCs, UNIX systems, VME-X, TPMS, MAC, Batch.

ReView gives full SQL read access to:

- many IDMSX databases as though they were one relational database
- many ISAM files as though they were one relational database
- many sequential files as though they were one relational database
- any combination of the above
- any combination of the above together with one INGRES database.

In addition, similar techniques make it possible to access the OpenVME catalogue, operator picture files, X.500 directories etc..

The ReView architecture exploits the provision, within INGRES, for software gateways to provide mappings between the relational model and external (non-INGRES) data. The ReView architecture provides a link, via an application executed within a TPMSX service AVM, to an IDMSX database.




Using ReView, relational application can access a number of IDMSX databases as through they were one relational database, but the IDMSX databases must be on the same machine as each other and on the same machine as the INGRES back end. By allowing the DTS link to be inter-system, an application can access a number of IDMSX databases as though they were one relational database, where the IDMSX databases can be on different machines from each other and from the INGRES back end.

# Chapter 9
# Networking Services

## Introduction

The networking services element of the OpenVME architecture provides the means of communication between the components of a corporate system and also with other systems. This chapter describes the architecture of the integrated communications facilities of OpenVME and the higher level interworking services built above them.

The architectural approach adopted by OpenVME is to provide comprehensive and fully integrated support for a core set of networking services and to use software or hardware gateways to support other services. Together, these services support both the hierarchic (workstation/server) and peer-to-peer (co-operative processing) dimensions of distributed computing. Such an approach provides for a single stable, highly tuned networking environment within OpenVME itself, and, via the use of appropriate gateways, the ability to choose from a wide range of networks and networking standards within a central corporate network, and within other systems communicating with an OpenVME system.

# The Open Systems Interconnection Architecture

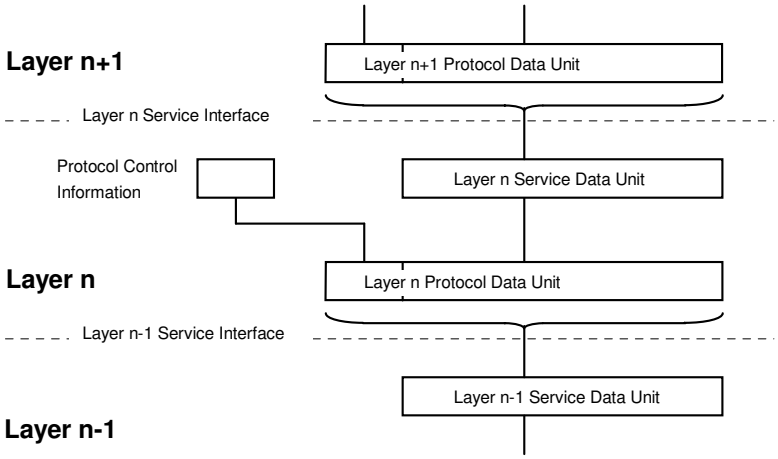
## The OSI Seven Layer Model

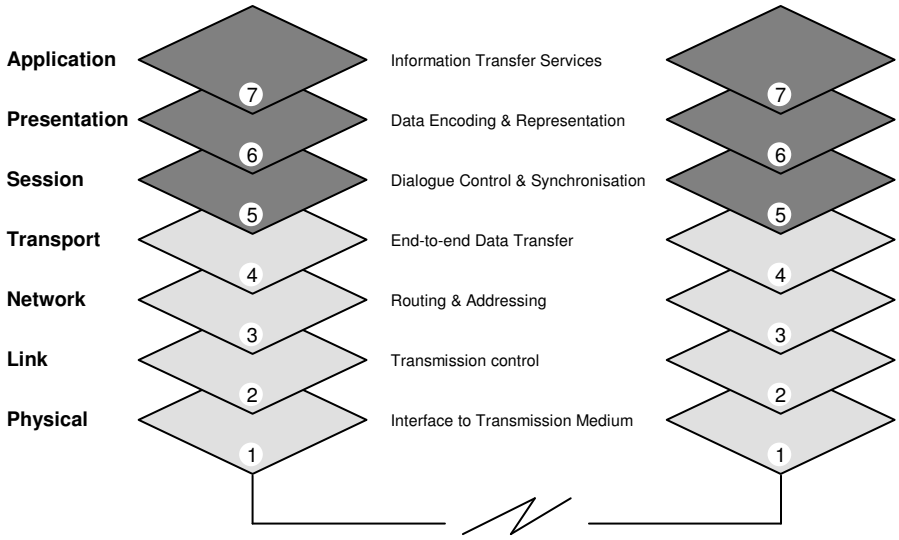
The Open Systems Interconnection (OSI) architecture divides communications functions into distinct *layers*. Each layer performs a well defined function in the communications process, providing a *service* to its immediate superior, which in turn provides a (higher level) service to its superior, thus creating a *stack* of layers.

Any communication is logically considered to be between *peer* functional entities (*i.e.* in the same layer), using protocol appropriate for that layer. This structuring of communications services and the corresponding protocols ensures a high degree of modularity and extensibility.

Each layer possesses a *Service Interface* through which the layer above interacts with it, sending and receiving *Service Data Units* (SDUs) and, in some cases, *indications* of certain conditions. When sending data, each layer adds its own *protocol information* to the SDU, creating one or more *protocol data units* (PDUs); these are passed to the layer below, via its service interface, as SDUs for that (lower) layer. When receiving data, the complementary process occurs, each layer removing its own protocol information before passing the received SDU to the layer above.

The general form of the relationship between adjacent layers is illustrated in the diagram below.






The OSI model defines seven layers:

1. Physical: the physical medium (*e.g.* cabling) and the standards used to represent information on the medium;
2. Link: control of data transmission over a physical link;
3. Network: establishment, maintenance & termination of communications between endpoints of a network, including addressing and routing;
4. Transport: provision of a reliable end-to-end communications service between systems, possibly on different networks;
5. Session: provision of logical communications paths between applications, including synchronisation and dialogue control;
6. Presentation: structuring and encoding of data passed between applications;
7. Application: provision of specific application services.

# Overview of the OpenVME Communications Architecture

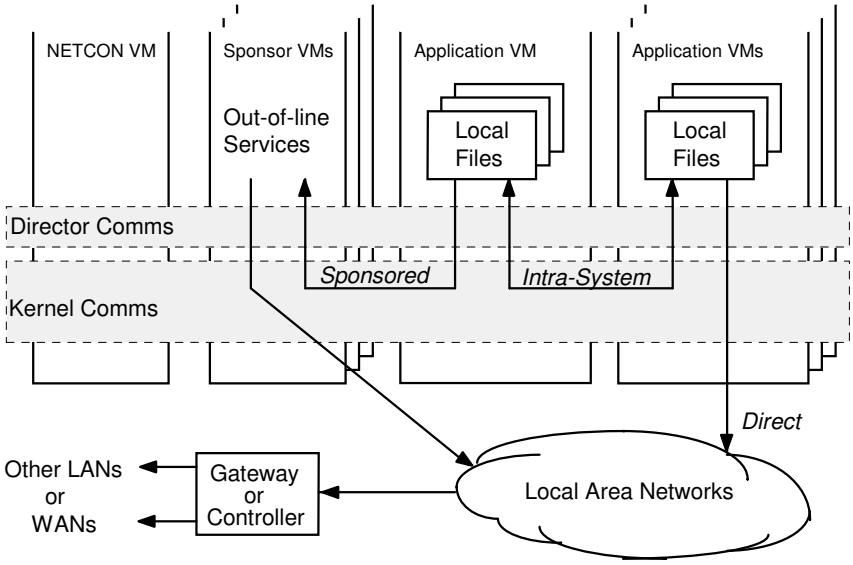
## Introduction

The core VME communications architecture is based on the OSI model and provides integrated support for the major OSI services and protocols. Support for the OSI layers is divided between those services provided in Kernel (OSI layers 1 - 5) and those provided in Director (some of OSI layer 6) and above (OSI layers 6 & 7). Broadly, Kernel provides the services in which a data unit may not be simply associated with a single VM and therefore cannot be handled "in-process". The Director subsystems known collectively as COSMAN provide a range of presentation facilities.

Efficient support is also provided for other protocol sets via gateways, accessed using the core OSI protocols. In particular, support for the Internet set of protocols (TCP/IP, UDP/IP) is provided using a combination of "in-process" software and gateway hardware.

The X/Open Transport Interface (XTI) API is supported for both OSI and IP protocols and open APIs for application services are provided where standards exist.

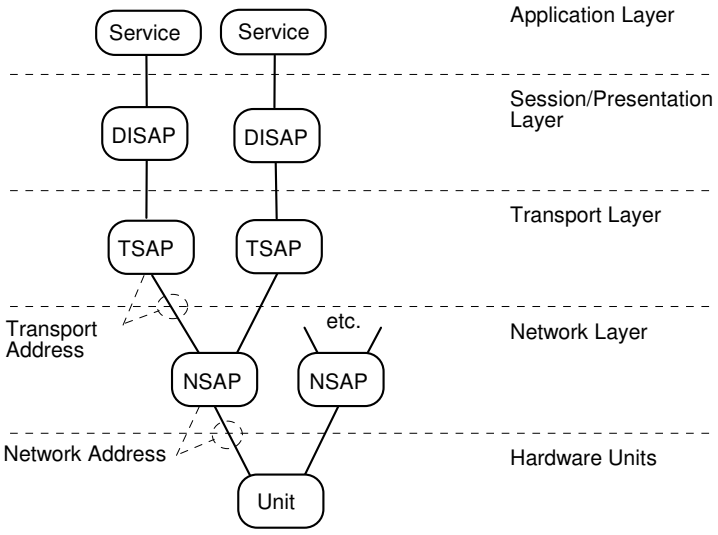
The diagram below illustrates the basic structure of the OpenVME Communications Architecture, showing key components:



### *SAPs and SAP Hierarchies*

Catalogued objects, *Service Access Points* (SAPs), represent endpoints of potential communications paths between peer layer services. Relationships between SAPs represent the use by one layer service of another and are labelled with the appropriate addressing and routing information; the resulting graph is termed a *SAP hierarchy*. At the highest layer, catalogued *Services* represent application level services and may possess an underlying SAP structure which defines the potential communications paths to or from that service. An "out-of-process" subsystem, NETCON, is responsible for controlling the establishment and termination of the underlying Virtual Resources, corresponding to the SAP structure, used to support an instance of a communications path.

A SAP can be be considered as representing a route into a protocol layer service. SAPs of different layers may be linked to form a *SAP Hierarchy,* representing a route through several layers of protocol handling; each such linkage may be labelled with routing or addressing information. At the uppermost layer, a SAP is usually the *route* of some specific catalogued *Service*, representing a route between an application providing that service and other applications, using a protocol stack whose layers correspond to the SAPs in the SAP hierarchy.




A *Local* SAP hierarchy represents the route, via layer services corresponding to its SAPs, by which a local service may be accessed by other services or, itself, access other services. Correspondingly, a *Remote* SAP hierarchy represents the route to or from a remote service (generally on some other system).

# OpenVME Core Networking services

## Introduction

This section outlines the core Networking Services provided by OpenVME, identifying the protocol standards used and the APIs available to applications requiring use of those services. As indicated above, additional services are available by the use of software or hardware gateways.

## Data Transmission

The core data transmission standards are based on standard OSI protocols and may be operated over Ethernet (OSLAN), FDDI or X25. Connection-oriented (CONS) and Connectionless (CLNS) network services are provided and may be accessed directly by applications.

## Data Interchange

Various classes of data interchange services across the transmission networks are provided:

### *Generalised real time (co-ordinated)*

OpenVME supports co-ordinated real time data interchange using the OSI TP service. APIs conforming to XATMI, TxRPC and CPI-C standards provide co-ordinated data interchange between several applications. An additional proprietary API is provided by the ADF subsystem.

### *Generalised real time (uncoordinated)*

OpenVME supports uncoordinated based real time data interchange using OSI TP (see also previous section) or DTS (ICL proprietary) services. These services may be used to provide uncoordinated real-time application to application data interchange.

Application access to OSI or IP-based transport services is also provided via an API conforming to the X/Open XTI (or the TLI).

Additional capabilities are provided through the use of proprietary ICL data interchange services for OSI only:

Application Data Interchange (ADI)
Remote Session Access (RSA) (terminal access)

These provide access to existing OpenVME applications; in the case of the terminal protocols, this enables access from PC-based frontware client/server applications.

### *Message Passing*

OpenVME supports messaging between applications (where there is no requirement for real time response) through the provision of X400 messaging services. The X400 service provided is based on an X400 Message Transfer Agent (MTA) and includes support for the X400 P1 service (for direct communication with other MTAs on large host systems), and the X400 P7 service (for communication with User Agents on PCs and workstations).

Messaging support is provided for both interpersonal messaging applications, and for client-server Electronic Data Interchange (EDI) based applications.

An API based on the X/Open message access and object management interfaces (MA & OM) is provided, which can be used by applications on the OpenVME system.

### *Remote Data Access*

OpenVME support for direct access to relational database systems via the use of SQL based protocols between the client and server systems, for the range of supported databases. Support for SQL access to Codasyl (IDMSX) databases and record-based files is provided via the ReView gateway {See Information Management}.

### *Bulk Data Transfer*

OpenVME supports bulk data transfer between applications through the use of the standard NIFTP protocol (used by ICL FTF) and via the OSI FTAM standard. The standard UNIX uucp, telnet and ftp facilities are supported over TCP/IP.

### *Distributed Function and Co-operative processing*

Application access to all of the networking services on OpenVME systems is provided through the use of X/Open standard application programming interfaces (APIs).

### *Shared Filestore*

OpenVME supports the NFS file server protocol. This permits files used by PC or Unix applications to be held in OpenVME filestore, and thus to take advantage of the capacity, security and filestore management facilities provided by VME. For example, the VME Custodian product can be used to archive UNIX filestore to a remote server.

## Supporting Services

OpenVME provides a number of supporting services for use by the various application components in the corporate network, as follows:

### *Directory Services*

OpenVME supports network Directory Services by hosting an X500 Directory Service Agent (DSA), providing local and remote directory services. Access to directory services is primarily via OSI protocols.

An API based on the X/Open directory and object management interfaces (OM & XDS) is also provided for applications running on the OpenVME system.

### *Message Switching Services*

A central message switching service is provided via the X400 service provided on the corporate server, which provides a centralised message switching system, allowing messages between various systems in the network to be controlled and routed through an OpenVME system.

## Gateways

Interworking with OpenVME systems from other networking environments is provided through transmission level gateways and free-standing gateways, for access to other core networking architectures

Gateways based on the use of open (i.e. ISO, TCP) standards are widely available, thereby greatly expanding range of choice in terms of the connectivity and access capabilities available to the users of OpenVME systems. For example, gateways providing OSI relaying onto frame relay, ISDN or Megastream networks are available.

## Terminal Access

### *ICAB-02 & ICAB-05*

The default terminal model for VME systems is based on a proprietary protocol. It is optimised for textual interactions between the user and the application which take place in terms of relatively coarse-grained dialogue elements. In a typical dialogue exchange, the application generates a screen which can contain several fields, each of which has attributes constraining the type of data which may be entered. The user can then exploit the local data entry and editing capabilities of the terminal to place new text or modify existing text in the displayed fields. When data entry is complete, the user causes the data to be transmitted to the application.

The ICAB-05 protocol embodies various enhancements allowing sophisticated control of display attributes and optimising the protocol between terminal and application (*e.g.* by omitting unchanged fields from the data returned to the application).

### *FORMS*

FORMS provides an enhanced user interface based on a distributed presentation architecture. The FORMS protocol:

- carries the logical elements of a user dialogue;
- allows screen definitions to be downloaded to the workstation.

FORMS is described in more detail in Chapter 12 (User Interface).

### *Asynchronous Terminals*

Flexible Terminal Handler enables all the common asynchronous terminal types to be supported and to have full access to screen-mode applications.

### *VTP*

VME provides "host end" support for the OSI VTP service, and maps it onto attribute and other protocol features known to VME TPMS services. The facilities supported at the terminal or terminal emulator are similar to those of proprietary VME terminals.

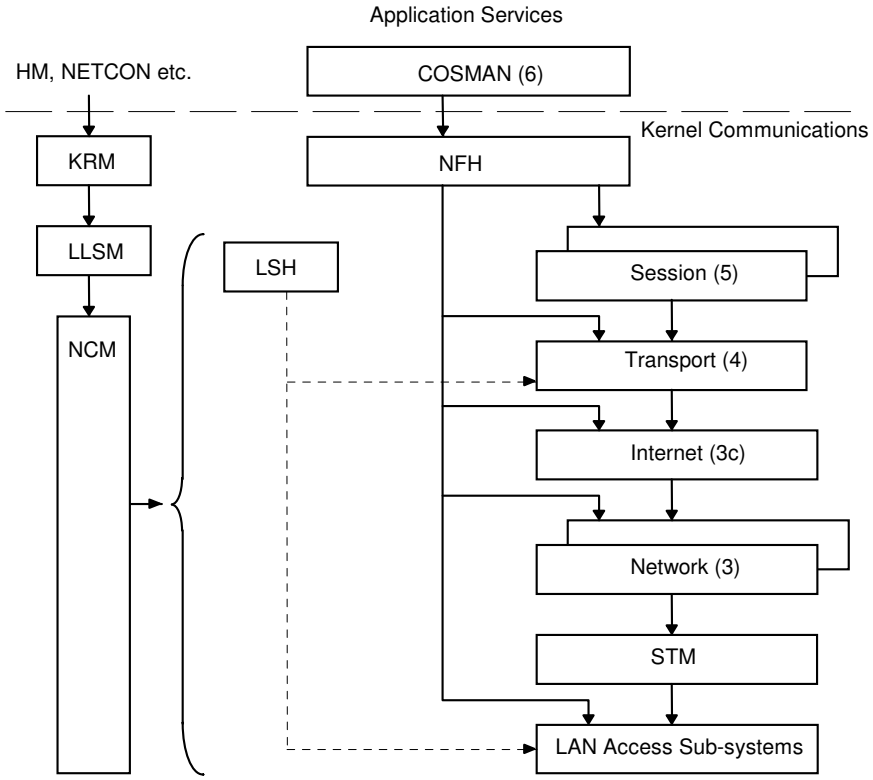
## The OpenVME Kernel Communications Architecture

### *Introduction*

Kernel communications architecture is based on a modular structure comprising infrastructure subsystems supporting a number of common functions (notably NCM and NFH), and subsystems for handling individual protocols or hardware devices. Protocol Handler subsystems are independent of each other and are structured so that, using the supporting infrastructure, one subsystem may pass messages to another. It is thus possible logically to *link* protocol handlers (and hardware device handlers) to support the protocol stack required by a particular communications path.

The modular nature of this architecture and the ability to link several protocol handlers to support a particular protocol stack has a natural correspondence with the layering of the OSI architecture.

Kernel Communications Outline Structure




*Key to Diagram*

| | |
|---|---|
| COSMAN | Communications and Slow device Manager |
| NCM | Network Connection Manager |
| NFH | Network File Handler |
| Session | Session Protocol Handler (OSI layer 5) |
| ICM | Intra-system Connection Manager |
| ETH | External Transport Handler (OSI Layer 4) |
| OSI Internet | Internet protocol handler (OSI Layer 3c) |
| Network | Network protocol handlers (e.g. X25, ICLCnn etc.) |
| STM | Stream Manager for driving streams over LAN |
| LAN Access | Device / controller drivers for LAN access |
| KRM | Kernel Reconfiguration Manager |
| HM | Hardware Manager |
| NETCON | Communications Network Controller |
| LSH | LAN Station Handler (LAN Station Management) |
| LLSM | Low-level Station Manager |

Two concepts are important in the descriptions that follow:

- a *unit* is the Virtual Resource within a communications subsystem associated with a specific communications path through that subsystem; in general each subsystem (including NCM) allocates its own units;
- a *file* is a Virtual Resource which is associated with a specific usage of a communications path by an above-kernel subsystem; NFH provides a common file-level interface to the various protocol handling subsystems within Kernel Communications.

### *Network Connection Manager (NCM)*

The NCM subsystem provides common interfaces for above-kernel subsystems to manage the Virtual Resources (units) supported by Kernel Communications subsystems. NCM also provides various protocol-independent functions and services for all the other Kernel Communications subsystems. These include:

- Unit declaration & deletion and managing the unit connection hierarchy;
- Establishment of links between other subsystems;
- Tracing, error logging, dumping etc.;
- Data buffering, maintaining buffer pools etc.;
- Timer handling etc.

The NCM management interfaces are usually invoked by Kernel Reconfiguration Manager (KRM), possibly via LLSM, on behalf of Hardware Manager (HM - in Director). For dynamically declared units, in turn, HM is invoked by NETCON which manages the declaration and deletion of such units. NCM provides a mechanism for NETCON to declare interest in certain units. This mechanism is used by NETCON to monitor state changes in units as well as the arrival of network connection requests.

### *Network File Handler (NFH)*

The *Network File Handler* (NFH) subsystem provides unshared Virtual Resources known as *files* through which an application may use underlying communications resources. It thus acts as a bridge, providing a common interface between the (shared) public communications handling in Kernel (accessed via NFH files) and the local handling used by an application within a VM.

NFH provides a common external interface to the normal "in-process" transfer functions of all Kernel communications subsystems. NFH provides services for above-kernel subsystems to enable file connections to be made through NFH to protocol handlers; it also provides services which are specific to individual connections such as tracing, event causing, input data queuing.

Associated with each NFH file are *queues* of data. These queues are held in public buffers, thus minimising the amount of buffer space which would otherwise have to be allocated in each VM. An input queue contains data

received by the system but not yet read by the application; an output queue contains data output by the application but awaiting transmission. A file may have several queues (input or output) associated with it, subject to restrictions for a particular file type or usage.

Communications resources are inherently asynchronous in their operation. However NFH minimises the impact of this asynchrony by ensuring that output data is buffered immediately, and that input data is buffered until read by a higher-level subsystem. Asynchronous completion of an output request and the arrival of input data (as well as certain exception conditions) are indicated to higher-level subsystems by means of an event.

### *Protocol Handling Subsystems*

The protocol handling subsystems carry out the functions of the various protocols and reflect the layers of the OSI model. Any particular protocol handler communicates with:

- the protocol handler (or hardware device or controller manager) at the layer below. Each subsystem provides a standard set of lower interfaces for use by underlying protocol handlers when indicating termination of actions initiated by the subsystem or unsolicited events;
- the protocol handler at the layer above, or with NFH if the connection to the handler is from above Kernel. Each subsystem provides a standard set of upper interfaces for use by NFH or higher level protocol handling subsystems;
- with other non-communications subsystems (HM, NETCON etc.) via NCM. each subsystem provides a standard set of interfaces for use by NCM - for example, to allow the subsystem to be initialised, and to allow units to be declared. Some subsystems also provide control action and attention interfaces which permit non-IO interactions between subsystems (*e.g.* loading a microprogram into a hardware device).

### *Communications Device Handling Subsystems*

The communications device handling subsystems manage the hardware devices or controllers which drive the physical communications medium. In practice, the hardware architecture of OpenVME systems uses a Local Area Network as a "system bus" to which hardware device and communications controllers are attached. Communications controllers connected to this LAN may either be gateways or true controllers; in the latter case the protocol stack becomes recursive, with the LAN protocol (*e.g.* layers 1-4) carrying messages to drive the controller (*e.g.* at layer 1 or 2).

## Director Communications Architecture

### *COSMAN*

VME Director provides applications with "in-process" access to communications facilities. The Director subsystems supporting these facilities are known collectively as *COSMAN* (COmmunications and Slow device MANager). In the case of communications resources, COSMAN provides a file-level interface which is closely aligned to that of NFH.

In terms of the OSI model, COSMAN supports certain presentation (layer 6) services, providing data encoding and decoding between an application and the underlying communications service. This allows applications to use the data structuring and encoding conventions independently of the protocols being used by underlying (or peer) communications layers.

### *Application Use of COSMAN Resources*

The set of interfaces provided by COSMAN for operating on a file comprises:

- a data input interface, to read a queued input data;
- a data output interface, to queue data for subsequent output;
- miscellaneous (queue) control interfaces;
- events used to notify the application of asynchronous conditions.

For data output requests, COSMAN performs presentation mappings as it transfers data from the application's local buffer into a local, on-stack buffer; NFH is then called to perform the transfer, and the on-stack buffer is copied to the queue in one or more public buffers; for data input requests, COSMAN performs presentation mappings as it transfers data from the queue in public buffers (via a reference provided by NFH) to the application's local buffer.

## Network Connection Management (NETCON)

### *Introduction*

*NETCON* (also known as *Communications Network Controller*) is an out-of-process subsystem responsible for instigating the creation, management and subsequent deletion of the Virtual Resources within an OpenVME system used to support a communications path.

NETCON is structured in a highly modular, layered fashion, with *layer managers* corresponding to the OSI layers. Communication between layers, across a layer service interface, is restricted to synchronous *requests* and asynchronous *indications*.

NETCON is implemented as a VM with privileged interfaces to various Director subsystems, including:

- Hardware Manager (HM) which mediates between NETCON and Kernel (NCM) or Director (COSMAN) communications subsystems in unit declaration;
- Network Controller, Upper Director (NCUD) which provides communication between user VMs and NETCON for passing connection and disconnection requests and indications;
- Resource Scheduler (RS) which provides the in-process interfaces through which user VMs interact with NCUD when requesting or accepting connections;
- Network Catalogue Manager (NCAT) provides commands for creating and modifying catalogued Service Access Points (SAPs), the relationships between SAPs and hardware communications units, and generic SAP descriptions. If the modifications are relevant to NETCON, NCAT passes the updated information to NETCON, via a task message, to allow it to modify its internal tables appropriately.

### *Network Controller Initialisation*

When the system is initially loaded, NETCON searches the catalogue for all SAPs, hardware communications devices and the relationships between them, building internal tables representing the SAP structures thus identified. Subsequent changes to the catalogue are notified to NETCON so that it can update its tables accordingly.

### *Unit Declaration & Connection*

Kernel and Director communications subsystems support *units* and NETCON is responsible for mapping layer hierarchies onto appropriate unit hierarchies. When NETCON needs to create a connection through a SAP, it requests the appropriate protocol handler or COSMAN (via HM) to declare a unit dynamically, and to connect it to a previously declared unit (either another dynamically declared unit or a permanent hardware device) with addressing information. It is possible for a unit of one layer to have several units of higher layers connected to it; such a unit is termed a *multiplexor*. The process of creating a unit (and connecting it) may require an exchange of protocol with a remote system - *e.g.* to establish an outward connection, to confirm an inward connection or for negotiation of connection parameters.

### *Unit Disconnection & Deletion*

The layer manager of a unit is notified of the disconnection of a unit in the layer above (or, at the uppermost layer, by an application). If the unit was dynamically declared and there are no other units in the layer above connected to the unit, the layer manager may initiate deletion (via HM) of the unit. Deletion of a unit is notified to NETCON by an event from the protocol handler. The process of disconnecting a unit and deleting it may require an exchange of protocol with a remote system - *e.g.* to request or confirm disconnection.

### *Incoming Connection Requests*

When an incoming connection request arrives in a Kernel Communications subsystem, the protocol handler informs NCM; NCM, in turn, notifies NETCON via an asynchronous event, identifying the hardware unit and any relevant protocol data. NETCON uses its tables to pass the request up through the layer managers responsible for the SAPs addressed by each layer of the protocol. Each layer manager may:

- declare a unit in one of the underlying communications subsystems;
- generate protocol to respond to the remote peer entity which originated the request;
- identify a SAP at a superior layer addressed by protocol at this layer.

The uppermost layer is usually the Application layer and at this layer protocol is used to identify a service to which the request is passed, directly or indirectly. As the request is passed up through the layer managers, units are dynamically declared, via HM, to support the required protocol stack.

### *Incoming Disconnection Requests*

Incoming disconnection requests may be originated at any layer and are notified to NETCON by an event from the relevant protocol handler. NETCON passes the disconnect request to layer manager responsible for the layer at which the disconnect request was originated. In general, this layer manager (or application) then causes a disconnect indication to layers above (if any) and generates a disconnect confirmation to be sent to the remote peer entity; the unit is then disconnected from the layer below and deleted.

### *Outgoing Connection Requests*

Outgoing connection requests are passed to NETCON from an application via RS and NCUD. The remote entity is usually identified as a remote service with an underlying remote SAP hierarchy (although a SAP may be identified explicitly). A service (or SAP) may have several routes and each layer manager uses a combination of user-specified preferences, load-balancing and quality-of-service parameters to determine the order in which routes are selected to attempt a connection.

NETCON passes the request down through layer managers corresponding to the selected SAP hierarchy. When the request reaches the manager responsible for the lowest layer of the SAP hierarchy (which is either a hardware communications device or a special SAP for intra-system connections), a unit is declared. As control is then returned successively to the layers above, units are dynamically declared and connected (via HM) to support the required protocol stack.

#### *Outgoing Disconnection Requests*

Outgoing disconnection requests generally originate at the uppermost layer of a protocol stack. The uppermost layer manager generates protocol to its corresponding peer to initiate disconnection. When this is confirmed to the layer manager by an event from the protocol handler, the unit is disconnected from the layer below and deleted.

## Above Director Communications Architecture

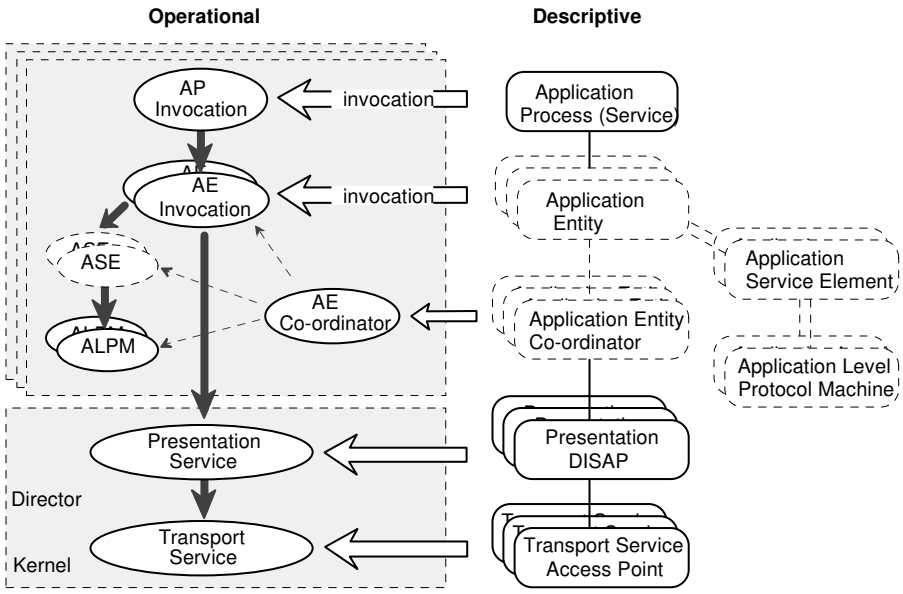
### Application Layer Architecture

In the OSI architecture, the Application Layer comprises A*pplication Processes* each which contains a number of A*pplication Entities*. An OSI application process denotes a collection of functionality and corresponds roughly with a VME service - *e.g.* a file transfer responder service. An application entity is a part of the functionality of an application process which is subject to OSI standardisation - *e.g.* an FT Responder's ability to operate the FTAM protocol.

The OSI architecture defines *invocations* - instances - of both applications processes and application entities. An Application Process Invocation is a specific instance of the usage of an application process (service). An Application Entity Invocation (AEI) is a specific instance of the usage of an application entity from an application process invocation and corresponds to a Virtual Resource providing the operations defined for the application entity.

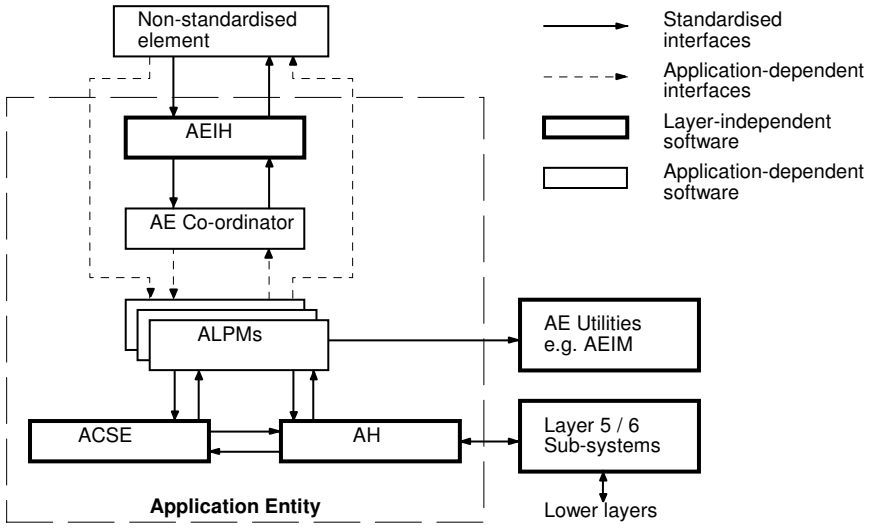
An application entity itself has internal structure, comprising a number of *Application Service Elements* (ASEs). Each ASE is supported by one or more Application Level Protocol Machines (ALPMs) whose behaviour is defined by protocol and service standards.

AEIs can communicate with each other using OSI presentation services, establishing a temporary relationship between them which is termed an *association*. The establishment and destruction of associations are functions of a special ASE termed the Association Control Service Element (ACSE); this ASE is present in every application entity.




In the OpenVME catalogue, an Application Process is represented by a service, known as the *principal* service. The service has routes via OSI Presentation SAPs (DISAPs), each addressed by a *DISAP Selector*, each representing an Application Entity within the Application Process. In the catalogue the AE co-ordinator(s) for each Application Entity is specified in the corresponding SAP node.

## Application Layer Software Structure



### *Non-Standardised Element (NSE)*

A Non-Standardised Element is that part of an Application Process outside its Application Entities which is therefore not subject to OSI standardisation. In the case of FTAM, for example, the NSE provides the user interface and filestore access functions.

### *Application Entity Invocation Handler (AEIH)*

Application Entity Invocation Handler provides interfaces to the declarative functions for an AE which are associated with the creation and destruction of AEIs. The interfaces to these functions are the same as those provided by the VME RSI for declarative functions on lower layer OSI services, and support the same event interface. AEIH uses loader facilities to *intercept* the RSI interfaces. If a request is made to initiate or listen for an application layer service (an Application Process), AEIH passes the request to the appropriate AE Co-ordinator; otherwise the request is handled in the default manner.

### *Application Entity Co-ordinator*

The AE co-ordinator is responsible for the creation of AEIs, establishing associations, and the creation & initialisation of the ALPMs required to support each ASE within the Application Entity. When the AE co-ordinator is entered to create an AEI it creates a corresponding Virtual Resource and returns a descriptor to the resource, enabling subsequent requests to the AEI to invoke the specific resource (AEI) directly.

### *Application Layer Protocol Machines (ALPMs)*

An Application Layer Protocol Machine handles a specific application layer protocol - *e.g.* the FTAM protocol or the X400 P1 protocol. APLMs use the facilities of AH to interface to the presentation layer and may optionally use utility procedures such as those provided by AEAM.

### *Application Control Service Element (ACSE)*

ACSE provides facilities for handling the creation and destruction of associations. These include facilities which allow an AEI to:

- establish an association with another AEI;
- accept an incoming association request from another AEI;
- offer an association to another VM;
- destroy an association.

### *Association Handler (AH)*

Each association is underpinned by a presentation connection. ACSE does not, itself, provide data transfer or synchronisation facilities; instead it makes use of presentation services directly available to the user of the association. AH provides facilities to:

- output data;
- notify inward data indications to a nominated ALPM or ACSE procedure;
- perform synchronisation actions;
- notify inward synchronisation actions.

### *Application Entity Area Manager (AEAM)*

Application Entity Area Manager provides ALPMs with facilities for allocating and managing variable length tables and buffers.

## Application Layer Services and APIs

### *OSI Application Entities*

The OpenVME system provides a number of standard OSI Application Entities:

- File Transfer And Manipulation
- X400 Message Handling Service
- X500 Directory System & User Agents
- Virtual Terminal
- Transaction Processing

### *OSI Application Layer Service Elements*

The OpenVME system includes a number of OSI Application Service Elements (ASEs) each supported by corresponding protocol machines.

- Application Control Service Element (ACSE)
- File Transfer And Manipulation (FTAMSE)
- Reliable Transfer (RTSE)
- Remote Operations (ROSE)
- X400 Message Transfer, Storage, Delivery, Access & Retrieval (MTSE, MSSE, MDSE, MASE, MRSE)
- X500 Directory Manipulation, Storage & Retrieval (DMSE, DSSE & DRSE)
- Virtual Terminal (VTSE)
- OSI Transaction Processing (TPSE)
- Commitment, Concurrency & Recovery (CCRSE)

### *OSI Application Layer APIs*

X/Open conformant APIs are provided to various OSI application layer services, most notably:

- The ACSE/Presentation Service APIs (XAP);
- The FTAM high-level API (XFTAM);
- The X400 Message Access Service (XMA);
- The X500 Directory Service (XDS);
- The XATMI API to the OSI-TP Service;
- The OSI Remote Operations Service (ROSE);
- The OSI Object Manager (XOM).

### *X/Open Transport Interface*

The Transport Interface (TI) subsystem supports the X/Open Transport Interface (XTI) which provides access to the OSI and TCP/IP & UDP/IP transport providers. Libraries are provided in both the X/Open Application Environment and within the VME C language to map calls in C on the XTI to equivalent calls on TI interfaces.

## Out of Process Communications Architecture

Several communications-related functions are performed out-of-process in dedicated VMs. Various infrastructure subsystems exist to provide a standard framework for protocol handling VMs. These are described below.

### *The Communications Service Infrastructure (CSI)*

The Communications Service Infrastructure provides a generalised framework for use in the implementation of application layer communications services related to application to application interworking. A CSI-based service has a scheduler VM and one or more protocol-specific support VMs. The infrastructure provides general purpose, protocol-independent functions and is responsible for the scheduling of connections to support VMs and providing management capability. Examples of services based on CSI include the X400 Message Transfer Agent (MTA) and the OSI-TP Gateway VM (OTP).

### *The Flexible Terminal Handler Infrastructure (FTH)*

Flexible Terminal Handler provides an environment for protocol-handling *sponsors*. Its prime use is for handling special terminal requirements, with sponsors mapping between the presentation protocol expected by the application and the actual terminal protocol. Sponsor modules exist to allow all the common de facto asynchronous terminal types to access standard interactive OpenVME services.

### *Out-of-line Layer Services*

Provision is made for a protocol to be handled by an *Out-of-line Layer Service* (OLS). The OLS mechanism enables a protocol handler to be written as an OpenVME application, executed in a dedicated OLS sponsor VM. Although out-of-line support of protocols is less efficient (involving inter-VM communication and switching penalties) it is a very flexible way of supporting protocols for which ultimate performance is not essential; it also allows user-written protocol handlers to be incorporated into the standard VME communications mechanisms.

An intra-system connection is established (by NETCON) between the user of an OLS and the OLS VM, so that the out-of-line support for the protocol layer is invisible to the user.

The NCOL subsystem provides supporting services for OLS VMs. In particular, it simplifies VM initialisation, establishment of communication and subsequent interactions with NETCON.

## Out of Process Communications Services

The following list summaries the major communications functions and services provided by out-of-process VMs.

- Communications Network Controller (CNC, NETCON)
- TCP/IP (VTI)
- Streams (STR)
- Layer protocol handlers (*e.g.* Yellow Book Transport Service)
- OSI TP Gateway (OTP)
- X400 Message Transfer Agent (MTA)
- X500 Directory System Agent (DSA)
- Remote Session Access (RGT)
- Asynchronous Sponsor Service (ASS)
- Asynchronous Terminal Handler (ATH)
- Virtual Terminal Sponsor (VTP)
- Transport Relay (RLY)
- X400 Remote Sponsor (X4SP)
- File Transfer copier (NIFTP & FTAM)

# Chapter 10
# Distributed Application Services

## Introduction

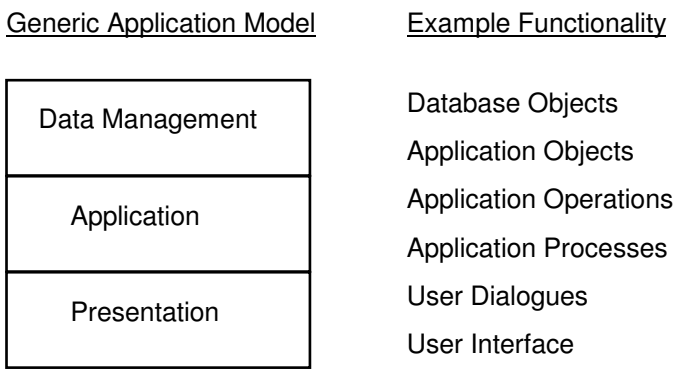
### Client-server Architectures

***Client-server*** architecture describes a distributed computing system in terms of the roles of, and relationships between interacting components - *clients* and *servers*. The architecture focuses on the distribution of functionality between the component that makes a request (a client) and the component that responds to it (a server). A **server** is a component acting on behalf of a client supporting a defined set of functions, known as *services.* A client, requiring a service, initiates an interaction by making a request to a suitable server and then awaiting a response from the server indicating completion of the request. It is important to note that a single component may take part in various interactions, in some of which it acts as a client and, in others, as a server.

The Client-server model is generally applicable. The term "component" usually refers to a software application but is also sometimes used to refer to the system or hardware platform on which it is executed.

#### *Generic Application Architecture*

A total application can be considered as having three major components: presentation, application logic and data management. Partitioning the functionality of an application in this way results in a modular structure which identifies relatively self-contained components which are potentially distributable.



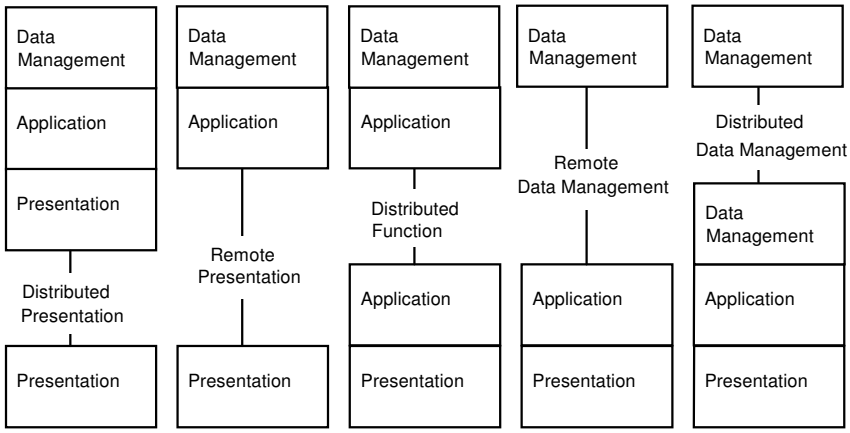
### *Distributed Applications*

An application is distributed if execution of different components or functions of the application takes place on different systems or hardware platforms. Distribution may occur *within* an application or *between* applications, or both. There are several models of distribution, each characterising a particular way of partitioning functionality; a total application may employ several models in combination.

### *Client/Server (or Workstation/server) Architecture*

The term *Client/Server* is commonly used to refer specifically to the class of systems in which one or more components running on a workstation platform act as clients to one or more server components running on a "server platform", usually shared between several workstations. This is also sometimes termed a W*orkstation-server* architecture to distinguish it from the more general use of the term Client-server described above.

Workstation-server architecture splits a single application into components executed on different platforms, the workstation and one or more servers, separated by a network. The application can be split in various ways, each termed a *style* or model of client-server distribution:

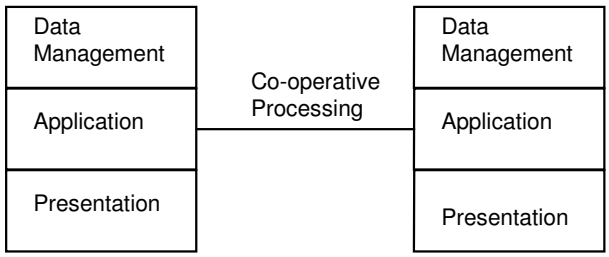



Source: Gartner

The term *remote* indicates that the component is remote from the application; *distributed* refers to distribution within a given component (presentation, application or data management).

### *Co-operative Processing Architecture*

A *Co-operative Processing* architecture is one in which applications interact with each other on a peer-to-peer basis (in contrast to the hierarchical relationships of the Workstation-server architecture). The architecture aligns well with an object-oriented approach in which each application encapsulates certain data and provides services which perform well defined operations on that data.




## Distributed Computing Infrastructure

Within the overall Client-server architecture, as well as the applications themselves, there is a requirement for a framework linking separate application components, enabling them to interact and co-operate to form coherent distributed systems, potentially in heterogeneous environments. The infrastructure which provides this distributed computing framework is generically termed *middleware*.

Middleware provides the means by which a client requiring a service can make a request which is passed to a suitable server, via underlying networking services.

### *Distribution Transparencies*

Middleware may provide one or more *distribution transparencies* which conceal various aspects of the distributed computing environment from applications:

- *Platform and Network Transparency*: each component operates independently of platform hardware and operating system, and of the underlying network services;
- *Location Transparency*: each component operates independently of the location of itself and of other components with which it interacts;
- *Migration Transparency*: a component may change its location at run-time without impacting operation of the system;
- *Concurrency Transparency*: a component may support multiple interactions concurrently;
- *Failure Transparency*: failure of a component does not compromise the ability of other components to recover from the failure;

- *Replication Transparency*: multiple instances of a component may be used to increase performance or resilience of the system;
- *Data Transparency*: the location, logical structure and physical representation of data managed by a component are hidden from other components accessing the data.

## Client-server Interactions

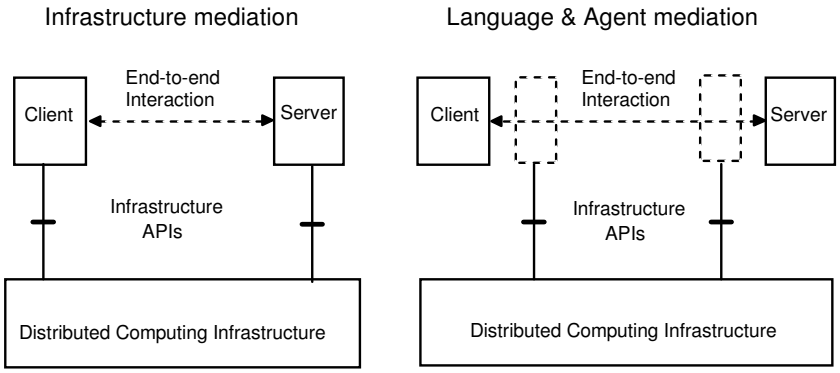
### *Component Use of Infrastructure*

From the application programmer's viewpoint, the use of distributed computing infrastructure may be implicit, if suitable transparencies are provided, or explicit otherwise.

Interactions between application components can be programmed by invoking infrastructure APIs explicitly. Such interactions are termed *infrastructure-mediated*.

Alternatively, interactions can be programmed by using language syntax to express interactions between client and server directly, so that the client apparently invokes a server API rather than an infrastructure API. Such interactions are termed *language-mediated* and depend on application development tools to construct the code that invokes the underlying infrastructure APIs.

A special case of mediated interaction is that in which interactions programmed as operations on a local API conceal a remote interaction. For example, remote printing, file and database services are invariably of this form. An agent component provides the local API for the client and the code that invokes the underlying infrastructure APIs to communicate with the server.



### *Types of Client-server Interaction*

The main types of interaction style in Client-server distributed computing are as follows:

- *Conversational interaction*: This is infrastructure-mediated interaction via dialogue over networking services. It has historically been the predominant style of interaction used for distributed application systems;
- *RPC interaction*: This is language-mediated interaction formulated in terms of procedure calls from client to server, according to procedure type definitions common to both;
- *Object request interaction*: This is language-mediated interaction formulated in terms of object-oriented interactions between client and server components;
- *Loosely-coupled interaction*: This is an interaction which takes place via one or more intermediate components. The interactions between client, server and intermediate components can be any of the other types of interaction identified above.

# OpenVME Distributed Application Services

## Run-time Services

The OpenVME distributed computing infrastructure provides supporting services which enable application components to interact with other components.

### *Networking Services*

Networking Services can be used explicitly for conversational interactions in a Distributed Function or Co-operative Processing architecture. They are invariably used implicitly, as an underlying communication mechanism, by higher level services supporting RPC or object request interactions.

OpenVME provides OSI and Internet networking services, accessed via the X/Open XTI transport interface. OSI application services such as VTP, OSI-TP, ROSE are also supported. Chapter 9 provides a detailed description of the OpenVME Networking Services.

Terminal access protocols are commonly used in distributed and remote presentation architectures, together with workstation frontware, to provide user-interface conformant presentation of unchanged applications. Chapter 12 provides a detailed description of the OpenVME User Interface services.

### *Networked Resource Usage*

Networked resources are used implicitly, via agent-mediated interactions. For external clients, OpenVME supports a wide range of services including:

- Networked File System (NFS)
- Message Handling Services (X400)
- Directory Services (X500)
- PC-LAN services

### *Remote Database Access*

Remote Database access can take place via agent-mediated interactions, with database *front-end* software or Client-server middleware providing the agent functionality. SQL is a key enabling interface for this type of distribution as it provides the data transparency that allows database queries to be expressed independently of the location or internal organisation of the server database.

All the relational database management systems on OpenVME support this type of distribution. In addition, ReView provides transparent data access, via an SQL interface, with all OpenVME data management facilities including IDMSX and flat files. Chapter 8 provides a detailed description of the OpenVME information management services.

### *Remote Procedure Calls*

Remote Procedure Calls are used for language-mediated interactions in a Distributed Function or Co-operative Processing architecture. They are sometimes used implicitly to support, for example, agent-mediated interactions invoking operations on networked resources. RPC services which operate in a heterogeneous environment must provide data transparency mechanisms to allow for different data representations on different platforms.

Extensions to basic RPC mechanisms allow RPC interactions to take place in a transactional framework. This technology is termed *transactional RPC*.

OpenVME supports several RPC capabilities, including:

- SVR4 Remote Procedure Call (RPC) with External Data Representation (XDR);
- DAIS has an RPC capability underlying its object request mechanisms (see below);

### *Distributed Transaction Management*

Transaction Management is an infrastructure service which can be used in conjunction with all styles of interaction. Whilst it is most obviously relevant to dedicated Transaction Processing environments, it may be used to ensure data integrity and consistency across any distributed computing system. For example, Remote Database access and RPC facilities can be enhanced in this respect by providing them within a transactional framework.

OpenVME provides transaction management facilities accessed via the standard X/Open APIs. The underlying OSI-TP service enables transactions to be distributed across several applications on different platforms.

These facilities are described in detail in Chapter 7.

### *Messaging services*

Messaging services provide support for loosely-coupled, agent-mediated interactions between co-operating components.

Transactional extensions to messaging services ensure that messages are delivered precisely once (if at all). The X400 definition is being enhanced in this way. In a complementary way, the OSI-TP service is being enhanced to support *message queues* which enable messages to be added to or removed from a queue in a transactional manner. This technique, termed *transactional messaging*, allows the guarantees of transactional operation to be extended to more loosely-coupled interactions.

The X400 Message Handling Service provides a means for passing structured messages with arbitrary content between components. The service is invoked via the X/Open MHS APIs. An important use of messaging services, frequently based on X400, is for *Electronic Data Interchange* (EDI).

### *Distributed Object Request services*

Distributed object request services can be used to provide language-mediated interactions between components in a co-operative processing architecture. Components are designed as *objects* with clearly defined interfaces. An object hides its internal implementation from other objects and can only be accessed via its interfaces.

Run-time services provide facilities for creating and managing objects, for routing client requests to appropriate server objects and for security, error management etc. Development tools greatly simplify the task of constructing distributed applications by providing an application environment model which abstracts above details of underlying platforms, networking services and infrastructure.

The OMG Common Object Request Broker Architecture (CORBA) is the emerging standard for object request environments. It is compatible with the ISO Open Distributed Processing (ODP) reference architecture.

The DAIS application development tools and middleware provide a CORBA conformant environment. DAIS is available on OpenVME as well as Unix, PC and other platforms, and is therefore highly suitable for integrating applications across heterogeneous, distributed computing systems.

### *Distributed Naming, Directory & Service Discovery services*

The components of a distributed system need to be able to identify the other components with which they require to interact. This is usually achieved by using a *naming scheme*. However in different environments the ways in which objects are named and those names decoded may vary considerably. It is therefore desirable to use a *federated* naming scheme in which names are decoded within the context of a previously identified environment; in such a scheme, an element of a name may identify a new environment in which the remainder of the name should be decoded.

Messaging services based on X400 standards can use the X500 directory service to provide addressing and routing information.

The OMG CORBA architecture allows clients to identify service requirements in terms of attributes. The Object Request Broker matches requests to advertised services of which it is aware and mediates in the establishment of an association between client and server.

### *Distributed Security services*

Distributed security is an infrastructure service which can used in conjunction with any style of interaction. OpenVME co-operates with ICL Access Manager to provide secure, authenticated logon. This is part of the Sesame architecture on which the ECMA security architecture is based.

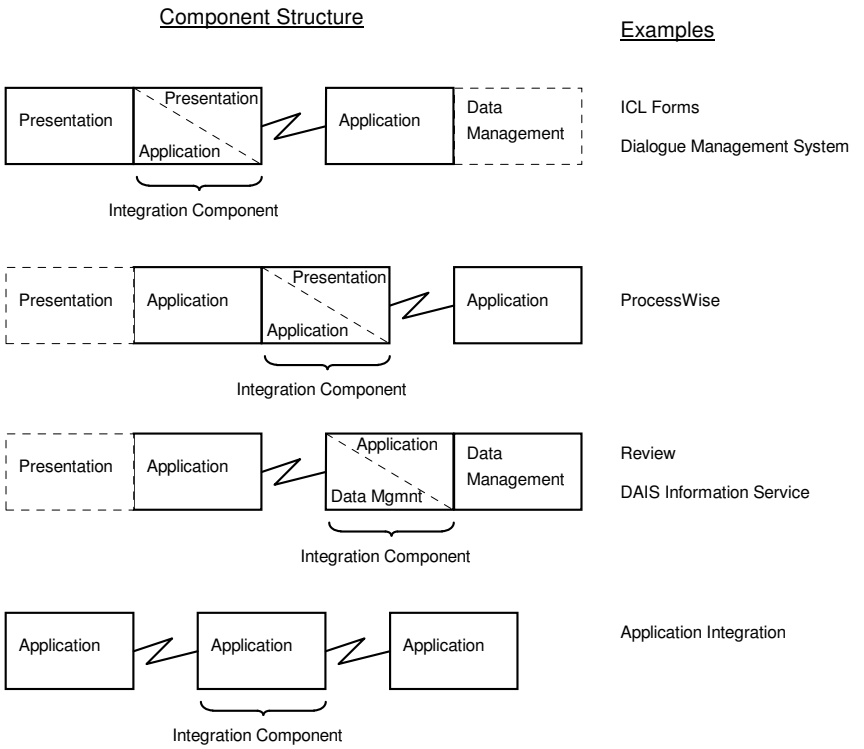
Distributed applications can use OSI association management features to provide secure, authenticated association between application servers. Finer grain security (*e.g.* related to individuals or roles) must be implemented by applications themselves. Extensions to the underlying security make additional services available via the *General Security Services API* (GSSAPI).

The DAIS/SE security extension provides transparent access protection to all data and services, encryption facilities, secure authentication and administration tools. DAIS/SE security services are accessed via the GSSAPI.

Within OpenVME itself extensive security mechanisms support both discretionary and mandatory security policies (see Chapter 15).

## Integration Tools

The run-time services and application development tools described above enable a coherent set of distributed components to be developed and executed. Distribution transparencies ensure that the components are combined into a total application in a naturally uniform manner.




The rationale for many distributed systems is the *linking* together of several separate applications to provide a single, coherent interface to the functions and information provided by the applications. One important additional capability is to *combine* functions or information from separate applications, thereby providing new functionality. The linking and combining of applications in this way is termed *integration* and the software components which provide a framework for developing and executing integration functions are examples of integration tools or components.

In many cases it is necessary or desirable to integrate applications developed using different methods and with different run-time component structures, interfaces and environments. It is often only possible to interact with these components through prescribed interfaces which may not have been particularly designed for use by other applications. Integration components can be used to adapt existing interfaces so that a component appears to provide the interfaces required to allow it to be integrated into the total application.

For example, an existing applications may only be accessible via a terminal (presentation) interface. An integration component can provide a terminal interface (via networking services) to the application and a procedural interface to other applications, allowing them to access the services provided by the existing application. The diagram above illustrates some application component structures showing examples of the use of integration components within the structures.

The integration tools used to construct integration components often require information about the components which are to be linked. The most effective method of achieving this is to use a common source of design information for all applications and development tools - a Data Dictionary. For example, the Dialogue Management System (DMS) can use screen design data in the ICL Data Dictionary System to allow a DMS application to be written solely in terms of logical fields, with the intermediate screen presentation entirely hidden.

## Distributed Application Development

Application Development tools play a key role in supporting a programming model which hides unnecessary details of distribution from the programmer. Tools are an essential ingredient of language-mediated distributed computing, delivering automated distribution transparency.

The use of the Open AM application development tools for the development of distributed applications is described in Chapter 11. Open AM is targeted at the development of applications to run within application servers interworking with Open TP facilities. The Open AM 4GL provides platform, network and location transparencies, allowing client and server applications to be developed with minimal dependence on how the applications are eventually partitioned and distributed. Application procedures and their formal interfaces are described in the Data Dictionary (DDS) and procedural invocations between client and server applications are automatically transformed into Remote Procedure Calls. Open AM generates X/Open conformant COBOL code (with embedded SQL if appropriate) and automatically generates code to interface to the underlying infrastructure - *e.g.* the X/Open XATMI Application Programming Interface for distributed transactional operations with data transparency.

DAIS application development facilities include a powerful set of code generation tools which hide the complexities of distribution. Service interfaces are defined in terms of the CORBA *Interface Definition Language* (IDL) which is an abstract notation for describing the set of service operations which comprise an interface.

The DAIS stub compiler generates *stub* routines for client and server programs. The stub code is responsible for any data conversion required for transparency

and for calling underlying infrastructure APIs. The application can invoke the stub code as though it were calling the server interfaces directly. Alternatively, client application calls can be written in *Distributed Programming Language* (DPL) embedded within the application which is then pre-processed to provide a target language source file. DPL provides interface type checking, dynamic creation and destruction of interface instances, and invocation of service operations.

## Distributed System Management

The OpenVME system management tools supporting distributed computing systems are described in Chapter 12. In addition DAIS and TPMSX both provide additional facilities for managing distributed applications operating within their respective environments.

# Chapter 11
# Application Development

## Introduction

This chapter describes the application development facilities of OpenVME. The ICL *Data Dictionary* (DDS) plays a central role in supporting these facilities. The *QuickBuild* family of application development tools, in conjunction with DDS, enable open applications for a wide range of application architectures to be developed and code automatically generated. Open interfaces to DDS allow dictionary data to be interchanged with third party CASE tools, thus allowing maximum flexibility to the application developer.

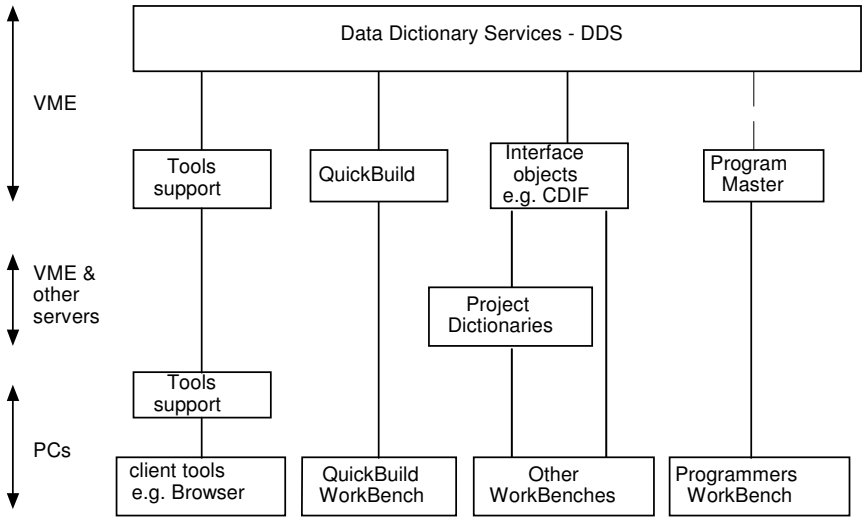
Equally, the OpenVME architecture provides an environment into which applications developed on other systems to open standards, can be simply ported.

A major theme of application development is to enable an application developer to design open applications in a manner which minimises the impact of differences in the underlying architecture. This is particularly important for distributed and client-server applications for which there is a large range of possible models of distribution. Application development tools are a key mechanism in hiding differences in environment and distributed application infrastructure from such applications.

## The Data Dictionary (DDS)

Many benefits are obtained from centring application development around a dictionary. The dictionary holds the master information about business processes and data and about computer processes and data. This single source of information is a model for and provides documentation of the application development process and provides a consistent source of application information. Tools can automatically generate applications, databases and forms from the information.

DDS has many features which make it particularly appropriate for use as a central repository for Corporate business dictionary data. QuickBuild and QuickBuild WorkBench are efficiently integrated with it. There are also links between DDS and leading CASE toolsets and UNIX dictionary products from leading vendors, in particular via the open CDIF (CASE Data Interchange Format) standard.




## Application Development Tools

### QuickBuild

QuickBuild is a uniquely powerful integrated CASE product set which, starting from diagrams on a WorkBench, automatically generates applications which may:

- be portable to other systems supporting COBOL 85 and relevant APIs;
- access relational databases and Codasyl databases;
- be distributed in structure;
- use 4GL with ICL FORMS;
- run in a Transaction Processing environment (*e.g.* TPMSX);
- run in a non-TP environment (*e.g.* CDAM).

Apart from DDS, the major components of QuickBuild are:

- Open Application Master (AM): 4GL system for application implementation;
- QuickBuild WorkBench (QBWB): a PC-based system analyst's workbench;
- Automatic System Generator (ASG): automatic generation of AM programs;
- Database Generator (DBG): automatic database generation from DDS;
- QuickBuild Pathway: a simplified environment for using other QuickBuild facilities.

### *Open Application Master*

At the heart of QuickBuild is *Open Application Master* (Open AM). Open AM can be used to develop stand-alone or co-operating applications, using Relational or Codasyl databases, which are portable.

The primary way to produce open applications is for the Open AM compiler to generate standard COBOL, which is portable, and thus hide platform and networking interfaces from the application code. The generated application code is targeted to be completely portable, in the sense that it is X/Open COBOL conformant (with embedded SQL) and has no machine, terminal or networking specific elements, all such elements being removed and relocated in middleware. When data is transferred between the client and server, its conversion to conform to different representations is provided automatically by middleware.

The primary model of distribution for applications generated by Open AM is that of *distributed function.* From a single AM application, client code is generated which runs on a workstation and server code which runs under a transaction manager (*e.g.* TPMSX). Alternatively, Open AM can be used to generate server code and other tools can be used to generate client code for the workstation.

Other models of distribution are also supported. In particular, co-operative processing between applications is supported. Such applications may run under a transaction manager or in free-standing environment supporting distributed dialogues via an appropriate API (such as XATMI).

### *QuickBuild WorkBench*

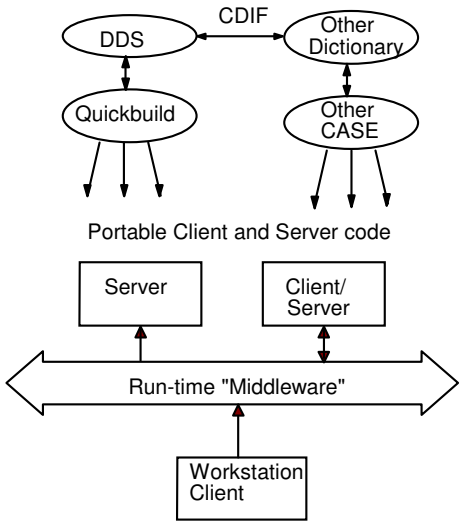
*QuickBuild WorkBench* (QBWB) is a system analyst's workbench which runs on a PC using a diagrammatic representation of the system design. Using From these diagrams, DDS representations of the program are generated and thence, using other QuickBuild components, applications and the databases which they access.

## Program Master & Programmer's WorkBench

Program Master and Programmer's WorkBench provide an application development environment for COBOL programs. They are integrated with the DDS Data Dictionary. An important difference between these tools and the QuickBuild set is that applications are developed and written in COBOL whereas with QuickBuild, a 4GL is used. Also they do not offer the same flexibility of generating open distributed applications.

## Application Development Tools - General

Any application development tools may be used to generate client applications running on workstations or UNIX platforms. With the support of Open TP interfaces by TPMSX, server applications running under TPMSX may also be generated by any appropriate tool.




Applications running under TPMSX generated by AM or written in a 3GL can interwork, using various interworking protocols, with applications generated by any application development tool.

There are links between DDS and leading CASE toolsets and UNIX dictionary products from leading vendors, in particular via the open CDIF (CASE Data Interchange Format) standard.

## Porting of Open Applications

OpenVME provides compilers for several standard languages including COBOL (74 or 85), C (K&R or ISO), FORTRAN (77), Pascal, C++, RPG2 and BASIC. Applications written in these languages can be recompiled to run on an OpenVME system. Debugging and run-time error handling facilities provide tracing and error reporting in terms of original source text and symbol names.

In addition OpenVME provides a full X/Open conformant Common Application Environment with standard libraries, allowing X/Open conformant applications to be simply ported to an OpenVME system. This environment includes the full base set of X/Open System Interfaces (XSI) and libraries together with the standard Commands and Utilities. ISO standard C and the associated libraries are provided together with COBOL 85. A comprehensive set of features is provided in addition to the mandatory base facilities. These include: C-ISAM & SQL (Informix), TCP/IP & UDP/IP (via sockets & XTI), and several Berkeley Unix and Unix SVR4 features to facilitate practical application portability. Profiling and symbolic debugging tools, including the ability to single-step through application code, are provided.

The Open TP environment provides a set of X/Open conformant APIs allowing applications that conform to the X/Open TP standards to be recompiled to run on an OpenVME system. Open TP application can interwork with applications running in other transaction management environments.

# Chapter 12
# User Access & User Interface

## Introduction

This chapter describes the features of OpenVME which support User Access and the provision of User Interfaces.

OpenVME is capable of supporting almost any terminal or workstation and the architecture allows the use of any platform or environment which supports user presentation, including dumb terminals, PCs, tone phones, videotext terminals, ATMs, multimedia etc. Character and graphical user interfaces are supported.

The OpenVME system is optimised for supporting large-scale server functions. Those functions which can effectively be supported locally for each user are expected to be provided by an intelligent workstation. The User Interface is such a function.

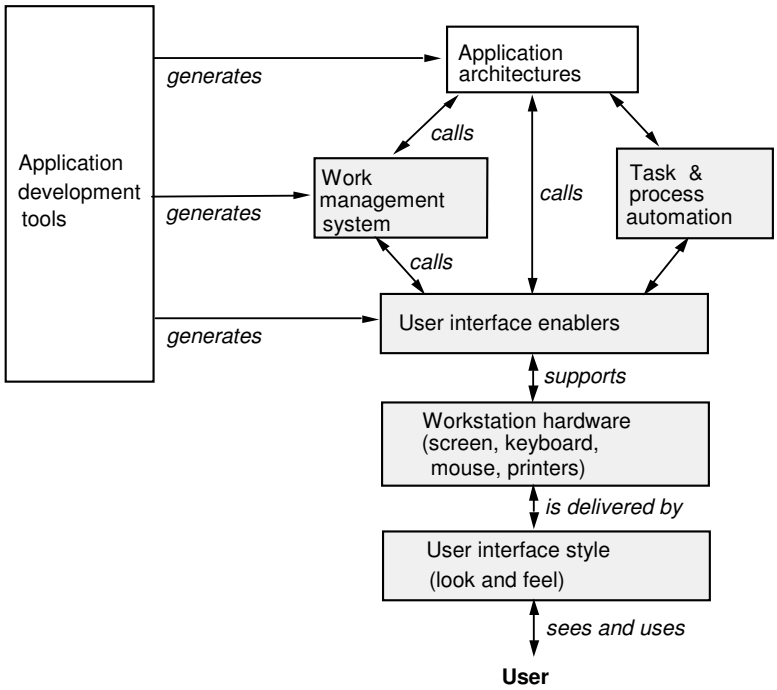
### Overview

OpenVME support for user interfaces is based on the following approach:

- Provision of user interfaces is achieved through the adoption of a Client-server architecture for all corporate server applications. This allows the user interface components to be implemented and supported separately from the application logic and information handling aspects of the application, implemented and supported on an OpenVME system.
- Microsoft Windows  is the default platform for the user interface component for all Client-server applications. This enables integrated simultaneous access to all corporate server applications from a single PC platform and with a single consistent user interface style.
- Access from other platforms and with other user interface styles is possible within the overall architecture and can be provided in response to specific market or application requirements.
- Facilities are provided to allow existing applications using generic text terminals to be upgraded to exploit this user interface architecture without change to the applications.
- The linking of the user interface component and application logic and information handling components of applications within the Client-server model is provided by the use of open standard interfaces. This allows the user interfaces provided to Client-server applications to be implemented using third party and commodity products (*e.g.* PC-based tools and applications).

## Reference Model

The diagram below identifies the major components of the user interface.




The *user interface style* defines the look and feel of the user interface in terms of user interface objects (*e.g.* a menu bar) and the actions which users can perform on these objects. It also prescribes the use of the workstation hardware. The preferred user interface style is the Common User Access (CUA) interface style which is supported by Microsoft Windows.

The *workstation hardware* includes the equipment used to deliver the user interface: screen, keyboard, mouse, printer and security devices. The standard hardware supports MS Windows (or eventually Windows NT).

*User interface enablers* make the user interface available to a range of applications. There are several types of enabler:

- Window managers / desktops / user environments enable users to use several applications concurrently; they support the user interface objects defined by the user interface style. The standard user environment is MS Windows (augmented where required by specific application exploitation of the underlying MS windowing facilities).

- User interface servers support a range of higher level objects (*e.g.* forms). ICL FORMS is an example of such a server allowing traditional TPMSX applications to deliver a CUA / Windows conformant user interface. Provision of the HLLAPI interface (directly or via DDE) to a server supporting VME (7561) terminal protocols allows a wide range of commodity PC applications and frontware tools to access information presented by existing applications.
- Terminal emulators give access to existing applications using a variety of terminal protocols and interface styles. The most significant emulations for VME services are VME (7561) and VT220. The target architecture is not optimised for applications driving such terminals directly.
- Object and data links enable information to be exchanged between applications. Object links enable objects managed by one application to be contained in objects managed by other applications. Data links provide ways of transferring data between stand-alone applications, terminal emulators and the dialogue logic of distributed applications.
- Data marshalling systems manage the passing of information between the user interface and the application. They enable the user interface of many different applications to be integrated together.

The *Work management system* enables an information system to maintain the user's working context. The features of this system include:

- Support for single system logon by a user for access to any permitted application. This capability is part of the ICL distributed security architecture, known as ICL Access.
- Definition and retention of the user's working environment in terms of applications and information used regularly;

*Task & process automation* provides facilities to allow the user to:

- Automate tasks which they perform regularly;
- Participate in organisational processes by using a process support (or workflow) system (*e.g.* Processwise). Such systems enable the automation of business processes and are powerful tools for the integration of existing applications into such processes.

*Application development tools* are used to develop individual applications and the interface provided by the applications to the user. The QuickBuild application development toolset generates server applications which work with any client applications, thereby allowing a free choice of presentation.

### *Distributed Presentation*

FORMS offers a distributed presentation service for single TPMSX services, but does not involve any application code within the FORMS client. FORMS separates the user interface server, executed on a PC in a Windows environment from the application dialogue handler, executed within TPMSX. It comprises two components:

- A FORMS sponsor VM which transforms TPMS template-based protocol into the FORMS protocol and manages the downloading of form definitions to the PC;
- A PC user interface server which maps the logical dialogue encoded in FORMS protocol to the Windows user interface style.

Interface features provided by FORMS include pull-down menus, scrolling and panning around large forms, field prompts and Help. The FORMS user interface server supports DDE links allowing PC-based applications (*e.g.* a spreadsheet) to transfer data to or from the application. New applications which use FORMS can be written in COBOL or AM or generated through QuickBuild.

The HLLAPI interface is an API which allows applications to interact with an object representing a screen image (based on IBM 3270). A user interface server is provided which supports the HLLAPI interface to an object which communicates with server applications on an OpenVME system via proprietary (7561) terminal protocols. A wide variety of PC based tools and applications can exploit this interface to access information presented by existing VME applications.

### *Remote Presentation*

In applications using this model, user interface functions are provided entirely on the server system and communication between client workstation and server is in terms of terminal protocols. Examples include the VME (7561) terminal protocol, VT220 and X-Windows. Applications of this type are not generally capable of delivering the required consistent user interface style. However, some integration is possible in a windowing environment. Such applications do not exploit the full capabilities of an intelligent workstation and may place a heavy load on underlying communications capabilities. This model is therefore not preferred in the architecture although it is recognised that there are circumstances in which it must be supported.

### *Distributed Function & Distributed Application*

In applications using these models the user interfacing functions and some of the application logic are supported on the workstation. Applications of this type are

supported by *middleware* linking the server and Windows-based parts of the application; the middleware provides APIs allowing PC-based components of the application to communicate with the server components.

For existing applications the ADI (proprietary communications) and HLLAPI interfaces are provided. The former provides an efficient means for application components to communicate raw data. The latter allows a wide range of commodity applications and frontware tools to access information presented by existing applications.

For new applications the XATMI (for real time applications) and the X400 XMA (or other proprietary PC-based) mail interface(s) (for mail/EDI applications) are provided.

ICL's Dialogue Management System (DMS) provides limited distributed functionality. It enables the construction of new applications on open platforms (PCs and UNIX) which can exploit the facilities offered by many different and existing TPMS applications as well as applications running on MVS and UNIX platforms, and other non-TP applications. DMS provides a modern and highly effective way of combining and encompassing existing mainframe-based applications within a new, PC-based application using sophisticated "dialogue management" facilities to determine when and how to invoke appropriate parts of existing applications.

### *Remote Data Management*

In applications using the this model all of the user interfacing and application logic are hosted on the workstation. Since the architecture supplies an SQL interface to all VME data sources [see Information Management], a common interface is supplied to all these sources, and there is a great range of query applications and tools which can exploit it. This is of particular use for MIS working.

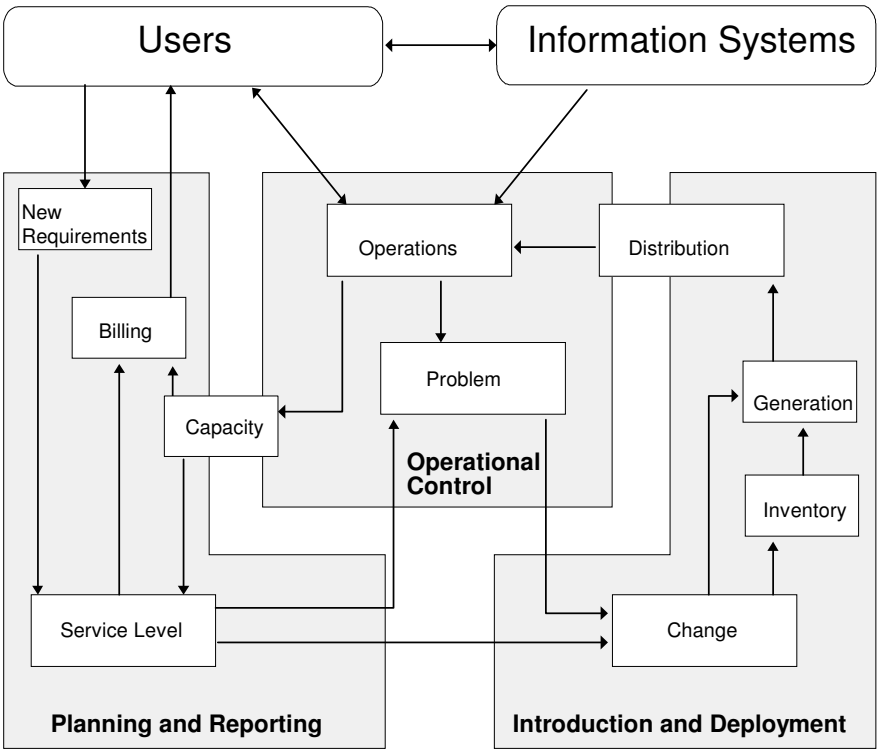
# Chapter 13
# System Management

## Introduction

The growth in use of distributed IT systems imposes new demands on the service provider who is faced with maintaining the availability of an application which now has potential dependencies on a range of platforms and networking services. System Management provides the tools which enable the service provider to control this complexity and diversity. This chapter describes the system management facilities of OpenVME.

### The System Management Process Model

The overall system management process follows the model illustrated below. This process model gives the service provider the mechanisms for ensuring that the corporate qualities are maximised to deliver the services required.

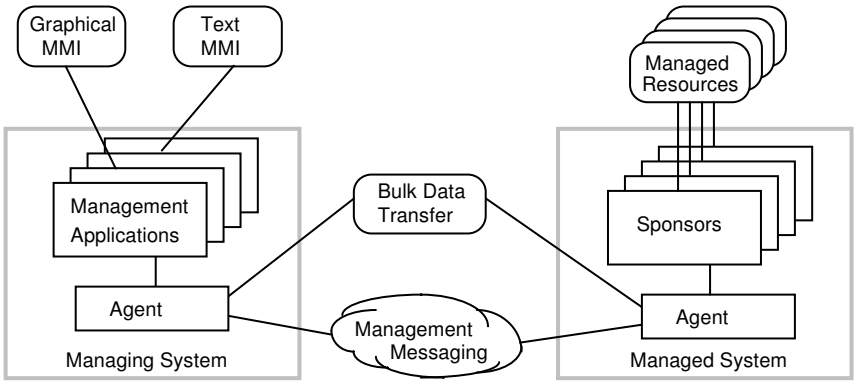


The model is divided into three key areas - based on different management concerns:

- *Operational Control* - providing real time system monitoring and control from a central location, dealing with issues and problems as they occur.
- *Introduction and Deployment* - to manage the whole process of making changes to the IT system.
- *Planning and Reporting* - to manage the quality of services and to assist in system planning and accounting.

## The System Management Functional Model

The generic functional model for System Management is illustrated below:




### *Managed Resources & Managed Objects*

The *Managed Resources* of a system include:

- Physical resources (*e.g.* nodes, printers, discs);
- Logical resources (*e.g.* VMs, files, units, connections);
- Application resources (*e.g.* queues, transactions, services);
- Domain resources: provided by a domain manager (*e.g.* network management)

The *Managed Object* model of a managed resource is a formal specification of how the various manageability features of the resource are viewed by the manager. The main purpose of the model is to provide a common framework for managing a type of resource. Managed objects are defined in terms of *attributes, operations, notifications* & *behaviour*.

### *Sponsors*

Sponsors provide a library of functions that assist in the implementation of managed resources. Sponsors simplify the management interface seen by a managed resource and can add functionality to the basic managed object model of a resource.

### *Agents*

Agents are concerned with providing access to managed resources (*e.g.* for management operations) and dissemination of information from managed resources (including notification of asynchronous events). Agents provide a standardised service for the exchange of management information between management applications and managed resources. They are responsible for identifying the managed resources affected by a request and establishing the necessary associations with agents in other systems.

### *Messaging & Bulk Data Transfer Services*

Messaging and Bulk Data Transfer services provide the underlying means of transferring management information between agents.

### *Management Applications*

Management Applications are a set of tools used by Service Providers to enact system management processes.

### *OpenVME support for the System Management models*

Delivery of enterprise system management capability is provided by the ICL TeamCARE product set. Mapping between the TeamCARE interfaces and other management environments is provided as part of the TeamCARE product set.

The OpenVME system is, by design, inherently manageable. Support for enterprise wide system management is provided by sponsors and agents which conform to the TeamCARE interfaces, ensuring that OpenVME is, and will continue to be, able to be integrated with the TeamCARE products.

OpenVME supports the systems management themes defined in the Process Model as follows:-

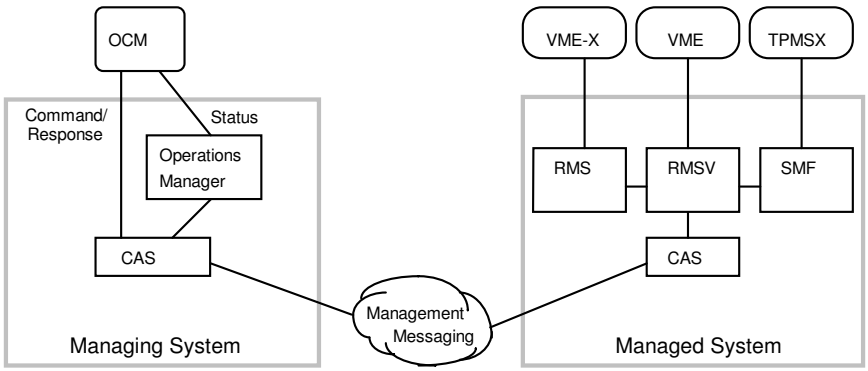
- For the Operation, Problem, Distribution, Generation, Capacity and Change systems management themes OpenVME provides sponsor and agent capability which will interwork with enterprise wide applications to provide system management solutions.

- For the Service Level, Inventory, Billing and New Requirements system management themes, OpenVME provides access paths to the necessary data for management via clearly defined, standard, interfaces.

# Operational Control

## Operations

Operations covers the provision of real time system monitoring and control from a central location, dealing with issues and problems as they occur.




*Operations Manager* provides the ability to manage a number of different types of system from a single location. It provides facilities to enable the managed resources to submit alerts and status information which is maintained within a central status database. This information may be aggregated and filtered by Operations Manager. Operator access to the database of information is via the *Operations Control Manager* (OCM) which provides a selective, graphical representation of the managed resources; each such managed resource may define a set of generic remote actions which can be initiated directly from the console. Remote command/response is also supported.

The *Community Alert Subsystem* (CAS) provides the messaging services between managing and managed systems.

Within the managed OpenVME system, the following sponsors are provided:

- the *Remote Management System for VME* (RMSV) provides sponsor facilities for status notifications and local actioning of commands for the OpenVME system;
- the *Remote Management System for VME-X* (RMSX) provides sponsor facilities for status notifications and local actioning of commands for managed resources owned by VME-X services;
- the *Standard Monitoring Facility* (SMF) provides sponsor facilities for the collection and dissemination of monitoring information, in particular for the Open TP application environment.

## Automated System Operation

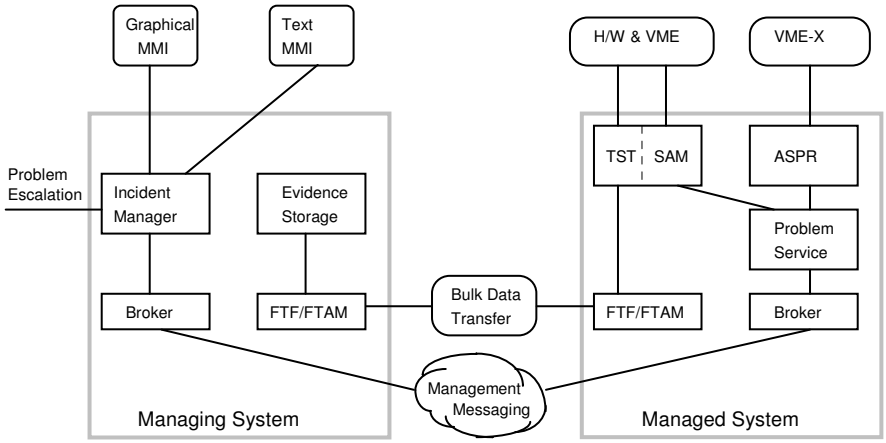
By default, the operational management of the system is performed by using the interactive operator facilities available within an OPER job. These allow an operator to receive and respond to prompts and to enter commands affecting the behaviour of the system.

OpenVME provides an optional feature, the *Automated System Operator* (ASO), which automates the responses to specified prompts, rather than requiring a human operator to respond. ASO can be programmed to perform specific actions in response to routine prompts. When a prompt arises for which it has no action specified, ASO can route the prompt to a local VME operator or, via Community Alert Subsystem, to the central Operations Control Manager. ASO exploits the Programmable Operator Facility infrastructure. The ASO task is based on POF, customised with procedures of the *Automated Operator Facility* (AOF).

## Problem

*Problem management* covers the diagnosing, resolving and prevention of problems.

OpenVME systems are provided with extensive automatic incident recognition and analysis capability which enables these systems to raise *calls* within a *call database* whenever the system detects a problem. Local call databases have the ability to communicate to a central *problem logging* application.



The main components supporting problem recognition & registration are:

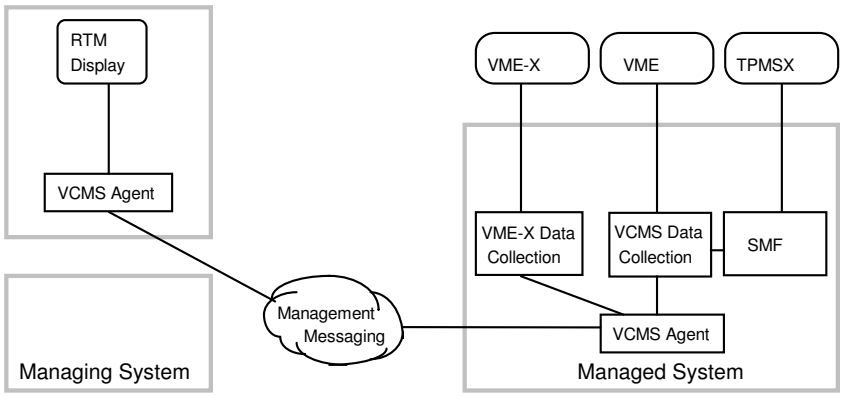
- *Incident Manager* provides centralised problem notification logging and management including control of the problem escalation interface to the vendors;
- *Problem Service* provides a local (to the system) call logging capability which may communicate centrally to Incident Manager;
- *Broker* provides agent capability for problem notifications;
- *Automatic System Problem Reporter* (ASPR) provides incident interpretation and threshold evaluation for exceptions occurring in managed resources owned by VME-X services;
- *Support And Maintenance* (SAM) provides incident interpretation and threshold evaluation for exceptions occurring in the OpenVME system;
- *Total System Teleservice* (TST) provides OpenVME applications with problem reporting and bulk evidence movement capabilities.

*Teleservice* provides the capability for electronic delivery of service between the customer and the vendor/service supplier. Services covered by Teleservice include *Diagnostic* services, *Problem Escalation* services, *Preventative Maintenance* services, *News* services, *Delivery* services and *Performance Evaluation* services.

*Telediagnostics* provides direct electronic access between the vendor/service supplier and a problem system. The level of access is based functionally at the individual unit level within a system.

## Capacity

*Capacity Management* monitors system usage and performance ensuring there is adequate capacity available and that it is being used effectively. Detailed performance statistics are used as an off-line tool to aid long term planning.



*Real Time Monitoring* provides a window into the current performance of the major system components using a hierarchical colour coded display. Time sampling of the performance data within a system with thresholding to provide colour changes is used to provide an ongoing graphical display of the system performance. The breaches of threshold may also create alerts via the messaging system to link into Operations Manager.

A *Statistical Analysis* package is used for providing capacity reports and trend analysis. The package supports a GUI and is not sensitive to the source of the statistical data. As a result it offers a strategic integration capability for mixed platform client-server environments.

*Sponsors* provide the mapping between the data requirements of Statistical Analysis packages and Real Time Monitor and the raw data gathered from the system.

## Introduction & Deployment

### Generation

*Generation Management* provides the means to configure and reconfigure the IT services in response to changing requirements. Mechanisms are provided for defining and updating configuration data. These are linked into the operation command and control structures to provide implementation capability.

### Distribution

*Distribution Management* provides the facilities for automatic, remote distribution and activation of software throughout the IT infrastructure. These facilities include:

- Software distribution, installation and control from the central management system;
- Monitoring of software delivery to target systems;
- Distribution by pre-defined schedules;
- Central log of software/data version by location.

*License Management* provides facilities for managing and optimising the use of software licensed for network use. OpenVME has the capability to act as a licence management server for a network, In this role, it receives issued licences from the software issuer and stores these in a protected manner. It responds to licence requests from client packages or applications to ensure that a licence exists for the level of operation that is required.

## Supporting Infrastructure

### Presentation

System Management supports a mix of *presentation* interfaces each tailored to the management role being undertaken, including high resolution graphical workstations (for real time operations and configuration design), form based working (for routine tasks such as help desk operation or distributed administration) and command line interaction (for diagnosis and problem solving).

### Management Infrastructure

#### *Management Messaging*

*Management Messaging* is the component of the systems management infrastructure that links elements of managing applications to each other and managing applications to management sponsors.

Community Alert Management (CAM) is the established ICL protocol for management messaging. CAM runs over an OSI transport or TCP/IP infrastructure.

The facilities provided by CAM will eventually be provided by a combination of CMIP, SNMP, X400, OSI TP and OSF DME.

## Relationship with other management domains

### Network Management

The established, de facto, standard for managing network resources is the Simple Network Management Protocol (SNMP). Based on TCP/IP it has been widely adopted for the linking of network resources and workstations into a managed domain. SNMP gateway facilities are provided which enable such managed domains to be interfaced into the system management functionality for monitoring purposes whilst retaining network management functionality within the domain manager. SNMP agent capability is provided on the OpenVME system to enable the networking functions of the servers to be included in the network domain.

## Workstation Management

The Distributed Support Information Standards (DSIS) Group consists of organisations which perform service, support and management activities for networked systems. The group is concerned with the potential costs and impact of distribution. These standards are intended to be protocol independent and are aimed at specifying data and operations which will enable DSIS compliant systems to support common service, support and management activities. OpenVME systems provide interworking with DSIS compliant networks. in particular the Problem Service can accept DSIS problem data.

Interworking with the PC LAN system management environments is provided via gateways.

# Chapter 14
# Platforms

## The Series 39 Hardware Architecture

### Introduction

The Series 39 hardware architecture is designed to allow a range of scaleable systems to be constructed from a set of modular components. At any time a range of systems is available in which the performance of the largest system is up to 100 times that of the smallest. All the systems are binary compatible and forward software compatibility is assured when new components are introduced.

Several features of the architecture are of particular relevance to the support of mission-critical application services:

- Disaster-tolerant configurations can be constructed in which central hardware components can be separated by up to 2 Km and Input/Output devices can be over 50 Km away.
- System configurations with sufficient components can be partitioned into two or more smaller configurations.
- The use of fibre-optical LAN technology for Input/Output connection allows considerably flexibility in configuring I/O controllers and devices between systems. Fibre-optical interconnect is also considerably more reliable and noise-immune than conventional wiring.
- The communications architecture allows simple, efficient interconnection to external system components. These include workstation platforms, other server systems or special-purpose server components such as the Goldrush parallel database server.

### Series 39 Hardware System Components

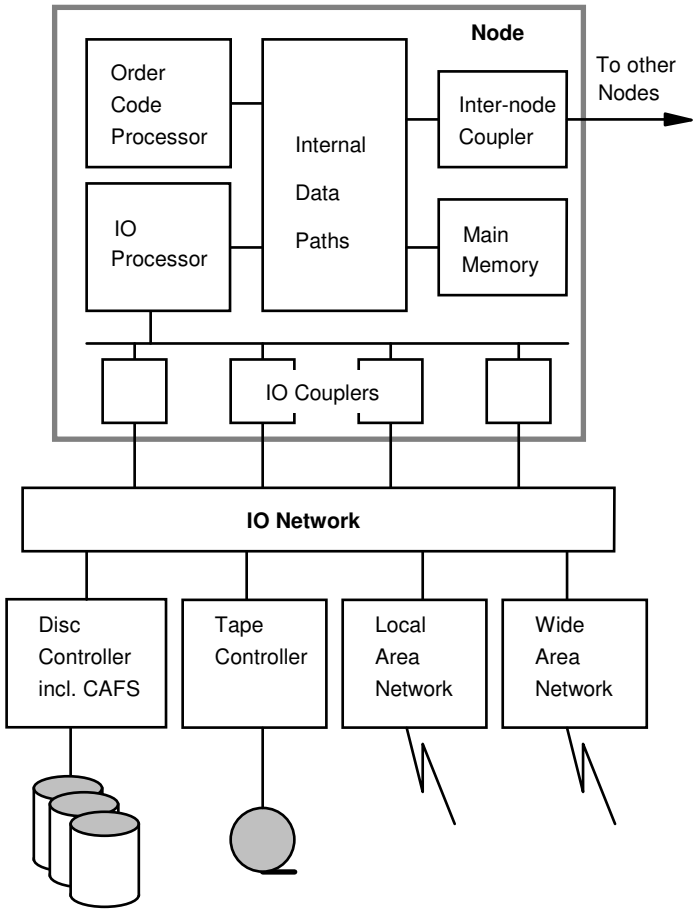
The major hardware components of a Series 39 system are:

- One or more Series 39 *processing nodes*;
- An *Input/Output network*;
- Input/Output *controllers*;
- Input/Output *devices*;
- An *inter-node network* (multi-node systems only).

Each processing node comprises:

- An *Order Code Processor* (OCP);
- An *Input/Output Processor* (IOP);
- A main *Memory*;
- A number of *I/O couplers*, connecting to the fibre-optical I/O network;
- An inter-node coupler (for multi-node systems only).

The Series 39 hardware architecture allows for arbitrarily large multi-node systems although some implementations are limited to eight nodes. In addition, the architecture allows for each node to contain several processors (IOPs and/or OCPs), providing an additional way of extending overall system performance. This flexibility allows a wide range of configurations to be constructed. For any particular set of requirements a set of components can be selected from which to build an appropriate configuration, and which also ensures excellent potential for future enhancement.



# Series 39 Nodal Systems

## Motivations

Many Virtual Resources are shared between several VMs. One way to implement such resources is by the use of shared memory to represent the state of the resources. This can be very efficient in that performing an operation on such a resource can be achieved by in-process execution of appropriate code in any VM which has access to the shared data. Alternative schemes, based on message passing, are usually very much less efficient.

However, large scale use of shared memory in multi-processor systems has a number of drawbacks:

- At the hardware level, the latency of memory accesses is adversely affected by the mechanisms required to ensure that, at appropriate times, all processors have a consistent view of the values stored in shared memory locations;
- The hardware engineering required to provide shared memory capabilities does not scale simply with increasing numbers of processors;
- The ability of the system to be resilient to the failure of a processor is compromised by the possibility that critical shared memory locations might become inaccessible or left in an inconsistent state;

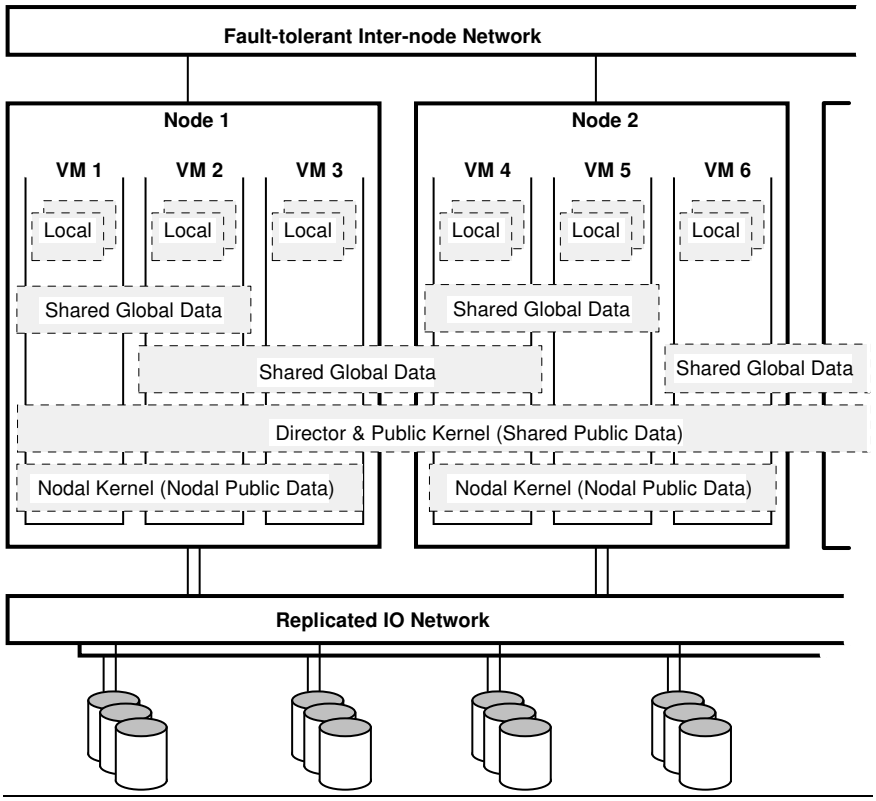
### *Nodal Architecture Hardware Structure*

The Series 39 Nodal hardware architecture addresses some of these issues by structuring the system into one or more *Nodes*, each of which contains:

- one or more processors;
- memory, only directly accessible to processors in that node;
- connection to an inter-node network which supports shared memory operations;
- logically direct connections to all input-output controllers.

The nodes are linked together by two networks:

- The Inter-node network which carries protocol used to maintain shared memory consistency;
- The IO network through which nodes access the IO controllers and devices.




### *Replicated Shared Memory*

Shared memory (whether Public or Global) is *replicated* in each node requiring access to the data. This replication is effected at the granularity of a page.

*Read* operations by a processor are always to the local memory (of that node). This eliminates the read latency often associated with accessing shared memory via a bus, for example.

*Write* operations to shared memory update the local instance of the shared memory and cause protocol to be sent to other nodes (via the inter-node network) to ensure that instances of the shared memory in other nodes are updated. This updating occurs asynchronously so that processing may proceed immediately after the write operation. *Semaphore* operations may be used to ensure that updates to shared memory by processes executing on one node are visible to processes executing on other nodes.

### *Inter-node Communication*

There is also a requirement for the instances of Nodal Kernel operating in each node to communicate with each other. For this purpose, an additional inter-node primitive operation is provided. A *Broadcast Interrupt* may be sent from a process (executing Nodal Kernel) in one node to all other nodes. The interrupt may be specified as a *process interrupt*, in which case it is notified as a vectored interrupt executed in the nominated process, or a *node interrupt*, in which case it is notified as a vectored interrupt on a pre-defined *Unit*. Kernel uses these mechanisms to implement:

- a *Vectored Event*, caused by a process in one node and notified to a process in another;
- a *Broadcast Call* which executes a nominated procedure in a process determined by the Unit on each node.

### *Nodal Architecture Software Principles*

The key principle of the OpenVME Nodal Architecture is to eliminate unnecessary sharing and interactions between concurrently executing processes in the system. This principle is reflected in several ways:

- Virtual Machines have individual instances of the unshared Virtual Resources they require; there is no necessity to co-ordinate with other VMs when operating on such resources.
- Real resources associated exclusively with a Node (such as processor time, memory, access to IO) are managed entirely within that node by Nodal Kernel; there is no necessity to co-ordinate with instances of Nodal Kernel on other nodes when operating on such resources.
- The IO architecture provides a separate *stream* (logical connection) from each Node to each IO device; there is no necessity for software to co-ordinate with other nodes when performing operations on an IO device.
- Virtual Addresses are allocated within an appropriately restricted context:

  Local segment numbers are allocated independently within each VM.

  Global shared data segment numbers are allocated locally within each VM, eliminating the potential for conflicts between address ranges of shared segments when a VM shares several areas with different VMs.

  Nodal Public segment numbers are allocated independently within each node by Nodal Kernel.

- Visibility of the nodal nature of the hardware architecture is restricted to Kernel. Kernel is responsible for ensuring that all higher layers are presented with a uniform view of all the resources of the system.

## The CAFS Information Search Processor

CAFS is an acronym for Content Addressable FileStore. CAFS is a special purpose processor which can be fitted to any Series 39 Disc Controller. Logically, CAFS operates between a disc drive and its controller, searching data as it is read from the disc for records which match user-defined search criteria. Records which match the search criteria are passed to the processing node; other records are discarded. CAFS allows large volumes of data to be searched very rapidly and by distributing intelligence from processing nodes into disc controllers (of which there can be very many) it greatly reduces the load on the processing nodes.

Search criteria can be numeric, alpha/numeric or straight text, and can involve:

- multiple criteria, combined with AND and OR operators, in each enquiry;
- precise or fuzzy matching (using wildcards);
- different types of questions: *e.g.* satisfying any three out of five selected criteria;
- a combination of the above.

# Chapter 15
# Support for Corporate Qualities

## Performance

### Introduction

*Performance* includes the following factors:

- *Connectivity:* The number of users requiring simultaneous access to the system.
- *Throughput:* The rate at which the system processes work, This includes measures of both the number of requests sent and received and the corresponding volumes of data.
- *Response time:* The time taken for the system to respond to requests for service.

Most applications can usually be implemented in several ways, with a number of options for partitioning the overall application functionality. This is particularly true of distributed applications for which there are several models of distribution. Deciding which model is appropriate in any particular case involves a complex trade-off between many different factors. Performance is usually an important factor and some implementation models may provide very different performance attributes from others. The overall amount and distribution of the processing power required to support a solution varies according to which model of distribution is chosen. This has implications for the overall financial cost of the solution.

The remainder of this section summarises the key performance aspects of OpenVME and then discusses the performance attributes implied by the various models of distribution.

# OpenVME Performance

## *OpenVME*

OpenVME is a high throughput and facility rich operating system. It has been designed to support large numbers of users and large amounts of data whilst providing good response times to interactive users *and* batch processing work simultaneously. Some of the architectural features which contribute to this capability are:

### *In-context Architecture*

Tasks run in *Virtual Machines* enabling the resources provided to be very effectively managed. Nearly all processor time is allocated, using pre-emptive scheduling mechanisms, at the priority appropriate to the task, which, for example, enable batch work to run alongside TP without severe impacting response times.

### *I/O Transfer Scheduling*

Virtual Machine *priority* determines the order in which transfers are actioned. This also prevents TP response times from being impacted by batch work. It is often more efficient to allow certain tasks, such as long-running queries, to use very large transfer sizes. This can reduce elapsed times by an order of magnitude, but might cause other, higher priority, tasks to suffer long transfer delays. The OpenVME architecture therefore ensures that very long transfers are interrupted in favour of short transfers of equal or higher priority. This strategy ensures that high priority tasks are not unduly delayed, whilst allowing lower priority or long-running tasks to make progress.

### *Multi-node Series 39 Architecture*

Additional power can be added with minimal disruption by adding more processing nodes. The multiple processors are invisible to applications and normally provide a performance increase of at least 0.9 per additional node. The unique nodal architecture avoids the memory bottleneck frequently associated with shared store symmetric multiprocessing systems.

OpenVME was designed at the outset for multiprocessor working and has therefore never been subject to the single-threading bottlenecks encountered by most other operating systems.

### *Performance Engineering*

The key functions in OpenVME itself and many other products including TPMSX and IDMSX have been engineered for performance. This means both that they are efficient and that their performance is thoroughly understood, so that throughput is predictable as well as high.

Transaction management environments can be specifically optimised for high performance: this is particularly true of TPMSX, and therefore, whatever distribution model is being used, it is the recommended environment for interactive server applications requiring high performance.

IDMSX has many facilities which allow the developer to maximise performance through appropriate application and data design.

### *Parallel Database Servers*

The OpenVME architecture allows the transparent integration of specialised servers dedicated to providing optimised support of a particular service. Parallel Database Servers support high performance relational database access. Their architecture is designed to ensure that "overheads" such as database lock management, I/O processing and I/O transfers are distributed evenly across the processing elements. The performance of parallel database servers is almost proportional to the number of processors, up to a specified maximum.

## OpenVME Client-server System Performance

A Client-server Architecture in which overall functionality is appropriately distributed and aggregated throughout the system is the way to provide high performance for corporate IT requirements:

- Complex forms of presentation (e.g. Windows) require large quantities of processing power to be placed in close proximity to the user - i.e. on the desktop. Significant communications delays spoil the "feel" of the presentation.
- The availability of cheap, powerful workstations means that for a large user population, provided the functionality can be distributed to the workstation, it is less cost-effective to provide equivalent processing power centrally.
- Corporate systems are characterised by the need to access shared information. Such information is most effectively made accessible by applications running on shared servers. Corporate Servers are specifically designed to support large numbers of users and large amounts of data whilst providing good response times.

Further performance aspects are discussed in terms of the most significant models of distribution.

### *Distributed Presentation*

The use of a block-mode (as opposed to character-mode) terminal interface to the server minimises the interrupt demand on the server platform, and provides the opportunity for it to support tens of thousands of terminals.

The use of either FORMS-style presentation or that provided through frontware (*e.g. via HLLAPI)* retains this efficiency in combination with a Windows user interface.

#### *Distributed Function*

The architectural support of this model allows developers to design applications to minimise network traffic for high throughput systems. Having the server application on the same system as the database means that database requests are not sent across the network. Having the client application on the same system as the presentation means that bit-mapped presentation requests are not sent across the network.

By exporting some of the processing demand, a central server may be able to support a larger population of users, etc.

#### *Remote Data Management*

As in all the models, for high-throughput access to databases, the usage of this model should be combined with the usage of a transaction management system.

To obtain high throughput, applications should be designed to minimise the volume of data passed between the application and the database manager.

#### *Distributed Application*

High throughput may be achieved though the use of transaction management systems and appropriate application design.

A design method which is often appropriate is to place application function associated with a given database together with that database. This tends to minimise communication traffic between distributed application components and therefore improve throughput.

## Security

### Summary

The provision of security within a client-server architecture through support of open security standards as described below provides OpenVME systems with the ability to provide high levels of security, and to do so in conjunction with a wide set of client systems and security products, both ICL and non-ICL, within which support for these standards is provided.

## Introduction

The provision of security features requires a number of separate capabilities:

- the ability to establish the identity and security attributes of an individual (a *subject*) wishing to access information, a service or any other resource;
- the ability to define security attributes of information, services and other resources (*objects*) of an OpenVME system, and policies defining the operation of security mechanisms;
- the ability to pass relevant security information between components of distributed applications;
- the ability to police accesses made to resources and to enforce relevant security policies according to the security information within the system;
- the ability to audit accesses made to resources and ensure that security violations are attributable to the individual who performed the access.

OpenVME aims to provide the above capabilities and to interwork with other systems in distributed networks possibly containing differing types of security and access products.

## Security in OpenVME

Security is a fundamental quality of OpenVME. All accesses to system resources must pass through a security barrier that checks to ensure that the caller has the required permissions to access the resource in the way requested. It is this basic architectural feature that has made it possible for OpenVME to become the first (and probably still the only) general purpose operating system to succeed in being evaluated at a level equivalent to the US DoD TSEC B1 (UKL 3).

The OpenVME Security Options (HSO) allow an installation to improve their security to the level they require at a pace that suits them. There are a number of options, including Personal Identification (allowing individuals to be personally authenticated) and Audit (allowing all accesses to system resources to be audited). The full security options suite supports "mandatory" access controls and allows system resources to be classified using military or commercial security classifications.

OpenVME will adopt the GSSAPI (Generic Security Standards API) which is becoming a de-facto standard; the GSSAPI allows applications to access different security services in a standard manner. An important feature of the GSSAPI is that it provides interfaces to applications that can be used to encrypt data prior to transmission across the network.

### *Establishing User Identity and Security Attributes*

This is normally carried out within the system on which a user performs initial "login", for instance by software based on a remote workstation: similar facilities to those provided on remote workstations are also provided by OpenVME for non-programmable terminals directly connected to the OpenVME system itself.

Integration of ICL Access Manager enables an OpenVME system to be part of a distributed network of systems in which end users perform a "single login" to all IT services. Access Manager is based on the ECMA security architecture of which the CEC-funded SESAME programme is a working demonstration.

The ECMA security architecture allows security administration to be devolved in a controlled manner within an organisation. The architecture allows a rich set of attributes to meet different access control needs and encourages the use of "real-world" attributes such as identity (*e.g.* person), role (*e.g.* generic job description) or security clearance.

A key feature of the ECMA architecture is the concept of a *Privileged Attribute Certificate* (PAC) which contains a person's privilege attributes and various controls on circumstances in which these can be used. A PAC is sealed by its originator and the use of *public key* technology allows any recipient to validate its integrity. A PAC can be passed form one application to another, thus providing the potential for full access control throughout a distributed network.

### *Transferring Security Information Between Applications*

Transfer of security related information between applications is enabled by providing support within the OpenVME systems for the open protocol standards defined for the transmission of such information.

In this context the standards supported include:

- overall security standards such as those being defined within OSI/ECMA, and implemented within the ICL Access Manager product set;
- application specific security standards, for instance the security features defined for use with standards such as OSI-TP, SQL, X500 and FTAM

As well as having the ability to accept security information when acting as a server system, OpenVME systems also provide facilities to allow this information to be passed on to other server systems, in cases where an OpenVME system is acting as a client to such systems

The use of open protocols to pass security information in this way allows the OpenVME security facilities to be used in conjunction with any remote system within which support for the relevant security standards is provided: this includes products such as ICL's Access Manager, as well as third party products.

### *Policing and Actioning Security Attributes*

Within the OpenVME system, the security attributes made available to the system are policed and actioned in a number of ways as follows:

- Access to data and other resources within the OpenVME system itself are policed through the use of the OpenVME discretionary security features, and through the use of the mandatory security features provided via the OpenVME high security product sets: in the latter respect OpenVME as a system is currently certified to security level B1.
- Access to the security attributes are provided to applications running on OpenVME systems for use for security purposes within the applications themselves: examples of such information would be user identification information for use in logging/audit trails. Access to this information will be provided through open standard GSS API programming interface.

### *Auditing*

The purpose of auditing is to ensure that any person who attempts to compromise the security of a system can be readily identified and held accountable for their actions. An audit log records details of any such access. OpenVME itself, as well as TPMSX, IDMSX and the relational databases provide audit logs.

### *Encryption*

The ICL Access Manager product provides standard encryption facilities. Encryption can be used:

- to conceal sensitive information;
- to seal high integrity information securely to prevent tampering.

## Availability

Availability is a measure of whether the system is there when the users need it; and whether it delivers the defined service and the agreed service levels.

Businesses are increasingly dependent upon their information systems to provide competitive advantage in their chosen market. The availability of these information systems contributes directly to providing this advantage.

OpenVME provides for resilient systems, with:

- multiple, independent copies of the data maintained, automatically by OpenVME. The independent copies (plexes) ensure continued availability of data in the event of hardware or environmental failure. The Series 39 hardware architecture allows data copies to be separated by many kilometres, thus providing the basis for survival from large-scale disasters;
- multiple VME processors, with automatic load distribution. This enables the application and database to be able to process the data in the event of loss of hardware or environmental failure. The Series 39 hardware architecture allows the processors to be separated by up to two kilometres for disaster standby;
- multiple, independent routes between major system components - nodes and controllers, and also to I/O devices such as discs, to ensure resilience to any single failure;
- multiple access from the OpenVME system to supporting networks, with automatic route switching, re-routing, or load sharing. This enables continued access between the OpenVME system and other systems over a resilient network;
- end-to-end hardware integrity checks on all data within the Series 39 hardware platform;
- end-to-end software integrity checks on dialogues between applications and, where required and permitted by security considerations, automatic substitution or recovery after failure of an application or application server;
- Hardware units taken out of service, upgraded or repaired, and put back into service while the OpenVME server system and applications continue to run. The Series 39 hardware is capable of continuous operation;
- TPMSX/IDMSX applications replaceable while the OpenVME system is operating. Certain applications can be upgraded while the system continues to operate;

All these features are invisible to the application and database, and can be implemented without changes to them.

High availability systems also require provision against failure of the system itself. Continuous availability also requires the ability to upgrade the system software while continuing to provide a service.

It is possible to meet both of these requirements by using twin OpenVME systems. By separating in the OpenVME system the functions of: network/client interworking, transaction execution ordering (usually specific to the application), and transaction/database manager, the backend functions can be implemented on two, independent OpenVME systems, with fast handover from one system to its twin in the event of failure. The TPMSX environment is particularly suited to this.

This standby approach to availability offers a range of solutions, from just replicating the key data up to immediate and invisible switch over from one OpenVME system to another. Each portion of the OpenVME system can be modified or upgraded, used to take archives, or to run other services; all without compromising the other key attributes of Corporate Servers, e.g. data integrity and control, or without affecting key services. The solution can be implemented in stages as the business demands and finance permits. This is often the most cost effective method of meeting the requirements, particularly when constrained by existing applications, working practices, or geographic factors, etc.

## Usability

Usability is defined as the degree to which users can achieve their goals with effectiveness, efficiency and satisfaction. The capabilities of OpenVME ensure:

*Effectiveness*: user operations are achieved accurately and completely through transaction management and other mechanisms.

*Efficiency*: an appropriate, consistent user interface may be chosen. Powerful application development tools are available which allow applications to be developed more quickly. Queries may be made to all Corporate Server data sources without applications having to be written.

*Satisfaction*: a user interface may be chosen which is subjectively acceptable to users; high availability, security and a good response time are available through the use of Corporate Servers. Facilities are available to present information in each user's chosen natural language.

# Potential for Change

## Introduction

The design of OpenVME ensures that OpenVME systems themselves and the customer information systems into which they are integrated, have, built into them, the ability to evolve to meet new market demands and new technical possibilities.

A major architectural strength of OpenVME is its adaptability to change. New standards can be efficiently supported using the gateway and sponsor components and through its in-context Virtual Machine architecture. Equally, new technology and system components (hardware and software) may be used to support current interfaces and standards without affecting existing applications. OpenVME has been designed using object-oriented techniques - data is associated with subsystems - and this allows new subsystems to be introduced and existing ones modified without cascading side-effects to other parts of the system and causing software decay.

## Transparency Mechanisms

OpenVME systems supply many *transparencies;* that is, they provide many features in such a way as to hide their potentially changeable attributes from application programs.

These transparencies are referred to in the following sections, which describe potential for change for each architectural element.

### *User interface*

A Client-server architecture allows the user interface to be modified without change to the server application. In many cases the user interface is modifiable by non-experts.

The supply of FORMS and generic frontware-enabling infrastructure allows existing OpenVME applications to be given Windows character and graphical user interfaces without re-writing the applications.

### *Distributed Application Services*

It is a major aim of the OpenVME Distributed Application Sservices to provide distribution transparencies.

### *Transaction Management*

Server applications can be written such that they are independent of the transaction source: the transaction management architecture caters for input from a variety of sources, including terminals, intelligent workstations, queued messages and mail messages.

### *Information Management*

*Data source transparency:* The supply of an SQL interface to all the OpenVME databases and file systems allows access to them from many new tool-sets. This access can be to single data sources or numbers of them in combination with relational database access. Data can be moved across data management systems without change to SQL applications.

Relational databases are modifiable in various ways without change to applications. The SQL language allows queries which are not pre-defined to be made.

Support of database management systems on advanced hardware can be achieved without change to existing relational applications.

### *Application Development*

The provision of the DDS dictionary allows business information to be kept which is independent of applications and data and can be used by many different CASE tools. The business model is independent of database type (IDMSX or relational).

The QuickBuild application development tools allow an application to be developed which work on different platforms and under different environments and such that its function can be distributed and re-distributed to conform to different client-server models.

### *Systems Management*

Generation Management provides the means to configure and reconfigure the IT services in response to changing requirements, and Distribution Management ensures that the same version of the software is installed on each system that receives the software. This is particularly important where there are many client instances.

### *Networking Services*

OpenVME supports interworking with a wide range of client platforms and environments. This means that server applications can continue unchanged whilst PCs go through several iterations of platforms, operating environments and windowing systems, including, for example, a change from Windows to Windows/NT.

The supply of middleware allows applications to be unaware of differences at the networking and distributed application services levels and hides the different data representations supported by client and server platforms.

### *Platforms*

The design of the OpenVME loading system normally provides transparency at the source code level in areas where other systems provide it only at the object code level.

OpenVME provides tape subsystem and tape device transparency. Media (printer, tape, disc) transparency is provided for sequential files. File connection transparency is provided through OpenVME "local names". Disc location transparency (over many kilometres) is provided.

OpenVME provides system size, node multiplicity, system version and hardware model transparencies.

# Appendix A: Standards

## Introduction

As a general statement, conformance to X/Open standards is supported. The relevant X/Open standard changes with time, for example: XPG3, XPG4, etc.

## Transaction Management

Standards in this area are being defined by X/Open. All the major TP vendors are contributing to this activity. By "Open TP" in this document we mean support of the X/Open transaction management interfaces and model. The X/Open interfaces for communication between TP applications are:

| | |
|---|---|
| XATMI | based on Tuxedo's interface |
| CPI-C | based on IBM SAA (extended for OSI TP) |
| TxRPC | based on DCE RPC. |

The IBM CICS interfaces are a competing standard. The architecture includes inter-operation with CICS systems.

The IBM standard application programming interface which can be used to create Windows applications that access unchanged 3270 mainframe applications is HLLAPI .

Co-ordination between participating systems is handled by the OSI-TP protocol (ISO/IEC 10026).

## Information Management

### Relational Database

The relevant standard relating to interfacing into the relational database management system is SQL. The ISO and ANSI standards for SQL are identical. The X/Open standard is converging towards the ISO standard at XPG4. All the major relational systems are close to supporting the standard, and will move closer over time.

The ISO OSI-RDA standard, when defined and supported, will enable networking between heterogeneous database products.

The SQL Access CLI (call level interface) as defined by the SQL Access Group (SAG) will ease the task of database-independent tools vendors.

Microsoft has produced a specification of their Open Database Connectivity architecture (ODBC). ODBC is a vendor neutral interface standard allowing Microsoft Windows applications to communicate with a variety of Database Servers. It will be supported from a variety of Microsoft and third party Windows applications. ODBC is claimed to be fully compliant with the SAG Call Level Interface Specification.

IDAPI is a similar interface from Borland (with support of IBM, Novell, Wordperfect and others).

The X/Open standard TP model includes a standard for the interface ("XA") between resource managers (such as relational databases) and a transaction manager. This enables co-ordinated access to heterogeneous database systems. It is supported by TPMSX.

### IDMSX

IDMSX essentially supports the international Codasyl standard. With the ReView development, SQL is be supported, for full read access.

## Application Development

### Dictionary and CASE

Data in DDS conforms to the international standard IRDS 4-layer architecture. The more detailed IRDS standards are not currently useful.

The relevant standard for exchanging data with other dictionaries and CASE tools is CDIF (CASE Data Interchange Format), which is promoted by EIA, the Electronics Industries Association, an accredited standards maker for ANSI. EIA is a US organisation, heavily populated by European (UK) members. At least 15 of the world's most well-known CASE vendors are participating in a CDIF prototyping exercise.

CDIF is now recognised as leading in the development of open standards for CASE information models. The new ISO DDSE standardisation work will be based on the CDIF models; this should confirm the use of consistent CDIF-based models for bulk transfer and for IRDS and PCTE schemas.

CDIF is also the clear leader in bulk transfer formats. ANSI X3H4 (IRDS) has already abandoned its own plans to develop an IRDS export-import standard, on the basis that CDIF will provide this capability, and ECMA TC33 TGRM are also looking at the possibility of using CDIF as the basis for the PCTE export-import capability.

No international standards for Analysis and Design Methods yet exist. SSADM is going through BSI processes and CORE is about to start. There is EC work on Euromethod.

### 4GLs

No standards for 4GLs exist. Where the 4GL generates a standard 3GL such as COBOL then lock-in can be avoided.

### 3GLs

For COBOL and C, the ISO and ANSI standards are the same. In both cases, the X/Open standard is a superset, and is the target standard. In the case of C, the ISO standard is expected to be adopted as the XPG4 definition.

## Distributed Application Services

Standards exist for the following environments: network services, PC-LAN, network filing, remote DBMS, distributed TP, message passing, DCE core, object request broker and ODP environments.

#### *OSI-TP*

This standard for transaction management has a significant impact on Open TP and forms the basis for ICL's Open TP products.

#### *DCE*

DCE establishes some de facto standards which are likely to influence industry perceptions in areas where they are adequate to the requirements in distributed computing.

#### *Open Distributed Processing (ODP)*

These standards define a reference model for open distributed processing. They are open (being underwritten by ISO), but have not yet been fully ratified.

#### *XDCS*

These are standards for distributed computing from X/Open. They are based on existing standards from other standards organisations.

#### *CORBA*

The OMG CORBA standard defines an architecture for an object-based distributed computing model which is compatible with the ISO ODP reference model. It is rapidly gaining acceptance as a common standard for distributed object-based systems.

## User Interface

The relevant standard for PCs is Microsoft Windows 3.

The relevant X/Open standard is X Windows from M.I.T.

## Networking Services

ICL's principal source of standards for communications and networking products is the Open Systems Interconnection (OSI) standards supported by ISO. Standards for LAN and WAN communication are supported below Transport level.

Where business cases apply, other Transport-level protocols such as TCP/IP or PC-LAN equivalents are supported.

Application-level profiles (X.400, FTAM, VTP) are increasingly supported by VME and VME-X over both OSI and TCP/IP networks. OSI-TP is supported over OSI. It can also be supported over TCP/IP by use of a relay supporting the RFC1006 profile and running in an external gateway.

## Systems Management

### *OSI Standards*

CSD products provide an external management interface which conforms to recognised open standards. Within the OSI domain this interface is targeted to be provided by the OSI Common Management Information Service (CMIS) running over the Common Management Information Protocol (CMIP).

### *DSIS*

The Distributed Support Information Standards (DSIS) Group is a consortium of companies who are jointly promoting the development of international standards for service and support information. Inter-operability with DSIS conformant platforms will be mandatory for Teleservice.

### *De Facto Standards*

SNMP has become the de facto Internet standard for the management of networks and the ability to interact with an SNMP domain is expected to be provided.

Other international and de facto standards (e.g. OSF DME) exist and will require support as they achieve a significant presence in the corporate system marketplace.

## Security

Standards in the field of security include the following:

### *Trusted Computer Security Evaluation Criteria (TCSEC)*

The US Department of Defence [sic] (DoD) publication Trusted Computer System Evaluation Criteria (TCSEC) is widely used as benchmarks for operating systems.

Because of its colour, this is commonly referred to as the "Orange Book". It concentrates on a number of areas of operating system security. These primarily affect confidentiality, accountability and assurance, but go further than just specifying what constraints must be available.

The Orange Book lays down a number of bands, ranging from D (no security) up to A1 (fully verified security), which define the minimum criteria against which individual computer systems may be certified. Those believed by ICL to be relevant to secure networks include:

- Level C2, which requires users to be individually identified. This identity is used as the basis for both access control and, through use of an audit trail, for providing individual accountability.
- Level B1, which requires Mandatory Access Control (MAC), with objects labelled with their security classification, and an informal statement of the security policy. It satisfies most Defence requirements without requiring formal design methodologies.

ICL believes that C2 is adequate for most commercial organisations; B1 may be required in some isolated pockets within them.

### *Information Technology Security Evaluation Criteria (ITSEC)*

The security services of a number of European countries (including Britain's CESG) have produced a list of "harmonised requirements" for the evaluation of computer systems. These differentiate between the level of functionality provided in a particular security domain, and the degree of assurance that this level or functionality is in fact provided. Assurance is a combination of the correctness of the functionality as a means of combating a perceived threat, and the effectiveness with which this functionality is provided.

In general, commercial clients are satisfied by a lower level of assurance than are defence or intelligence clients. ITSEC permits a system to be described in this way; TCSEC merges the functionality and effectiveness criteria in a way which makes it impossible to produce (say) a system which is believed (but not certified) to operate at Level B1.

ITSEC includes a number of "functionality classes"; some of these map directly onto corresponding TCSEC criteria. ITSEC also defines "effectiveness classes",

and of these two are believed by ICL to be relevant to commercial security systems.

- E2, where the supplier needs to state the security policy in the product
- E3, where it is necessary to describe the policy, and show how the development environment ensures that this cannot be compromised.

### *UK GOSIP V4.0*

The UK GOSIP V4.0 specification subsets ISO 7498/2 (ISO OSI Security Architecture) to derive a set of specifications which can be used by Government (non Defence) organisations with an interest in security. It highlights the need to carry out a risk analysis and recommends the CRAMM methodology.

GOSIP is primarily interested in the security of communication between computer systems, and specifically does not cover the security of these systems themselves. Thus it concentrates on the implementation of security policies in terms of the OSI 7-layer model and the measures which it is possible to take at each layer.

### *ECMA TR/46*

The ECMA Technical Report ECMA TR/46 establishes an architecture for a secure system, rather than a set of standards, and is thus of much greater interest as a framework for ICL product developments. Additional ECMA publications define the data elements and types necessary to support the ECMA TR/46 standards.

### *Open Systems Interconnection (OSI)*

ISO has ratified several OSI standards covering the basic Reference Model, specific communications services and protocols and network management functions.

# Appendix B: Glossary of Terms

| | |
|---|---|
| 3GL | Third Generation programming Language, e.g. COBOL or C |
| 4GL | Fourth Generation Language (very high-level) |
| Access Key | Value defining the maximum Access Level permitted to use a resource |
| Access Level | Value denoting the privilege level at which a process is currently executing |
| Account | Collection point for accounting & charging facilities |
| ACSE | Association Control Service Element |
| ADF | Application Dialogue Facility |
| ADI | Application Data Interchange |
| AE | Application Entity |
| AEI | Application Entity Invocation |
| ALPM | Application Level protocol Machine |
| AM | Application Master; also Access Manager |
| ANSA | Advanced Network Systems Architecture; a distributed computing architecture |
| ANSI | American National Standards Institute |
| APF | Access Permission Field |
| API | Application Programming Interface; allows applications to access services. |
| ASE | Application Service Element |
| ASG | Automatic System Generator |
| ASO | Automated System Operation |
| AVM | Application VM (TPMSX) |
| C | C language |
| CADES | Computer Aided Design and Evaluation System |
| CAE | X/Open Common Application Environment |
| CAFS | Content Addressable Filestore |
| CAM | Community Alert Management |
| CAS | Community Alert Subsystem |
| CASE | Computer Aided Software Engineering |
| Catalogue | Database describing all objects known to the OpenVME system |
| CDAM | Co-ordinated & Distributed Application Manager |
| CDIF | CASE Data Interchange Format. See Appendix A: Standards. |
| CEC | Commission of the European Communities |
| CESG | Computer Evaluation Security Group |
| CICS | Transaction Monitor developed by IBM |
| CLI | Call Level Interface (of SQL) |
| CLNS | Connection-less Network service |
| CMIS/CMIP | Common Management Information Standard/Protocol |
| Codasyl | An information model based on a network of records organised into sets. |

| | |
|---|---|
| CONS | Connection-oriented Network Service |
| Controlling File | File used to record file allocation and placement |
| CORBA | Common Object Request Broker Architecture |
| COSMAN | Communications and Slow device Manager |
| CRM | Communications Resource Manager (X/Open TP Model) |
| CSA | Concurrent Session Access |
| CSI | Communications Service Infrastructure |
| CTM | VME Common Target Machine |
| CVM | Control VM (TPMSX) |
| CUA | Common User Access |
| DAIS | Distributed Application Integration Services |
| DBMS | Database Management System |
| DCE | OSF's Distributed Computing Environment |
| DDE | Dynamic Data Exchange |
| DDS | ICL's Data Dictionary System |
| DES | Secret-key encryption algorithm used in the USA |
| Director | The part of VME which handles logical resources |
| DME | OSF's Distributed Management Environment |
| DML | Data Manipulation Language |
| DMS | ICL's Dialogue Management System |
| DNS | Distributed Name Service |
| DSA | X.500 Directory Service Agent |
| DSIS | Distributed Support Information Standards |
| DTS | Distributed Transaction Management System (TPMSX) |
| ECMA | European Computer Manufacturer's Association |
| EDI | Electronic Data Interchange |
| EIA | US Electronics Industries Association |
| Encryption | Secure encoding |
| Event | Object supporting inter-process communication & synchronisation |
| FDDI | Fibre Distributed Data Interchange |
| FIFO | First-in-first-out |
| File Section | Section of a partitioned file (e.g. reel of multi-reel tape file) |
| Filestore Category | An aggregation of filestore managed similarly |
| Filestore Description | Description of an allocation of filestore space |
| Filestore Volume | Disc volume (or partition), magnetic tape reel or cartridge |
| Frontware | Tools which enable client applications which interwork with unchanged server applications to provide an improved user interface and other functions. |
| FTAM | File Transfer and Access Method |
| FTF | File Transfer Facility |
| FTP | File Transfer Protocol; part of TCP/IP |
| GB | Gigabytes |
| GOSIP | Government OSI Profiles |
| GSI | General System Interface - commands which can be called from SCL |
| GSSAPI | General Security Services API |
| Hardware Unit | Central units, peripherals, other systems on local LANs |

| | |
|---|---|
| HCI | Human-Computer Interface |
| HLLAPI | High Level Language Application Programming Interface |
| HSO | VME High Security Option |
| I/O | Input-Output |
| ICAB-02 | Terminal access protocol used with VME |
| ICAB-05 | Extended version of ICAB-02 |
| IDL | Interface Definition Language |
| IDMSX | Integrated Database Management System |
| IEEE | (US) Institute of Electrical and Electronic Engineers |
| Installation | The root for enumerating objects in an installation |
| IPC | Inter-Process Communication |
| IRDS | Information Resource Dictionary System |
| ISAM | Indexed Sequential Access Method |
| ISDA | Interactive Screen Design Aid |
| ISDN | Integrated Subscriber Digital Network |
| ISO | International Standards Organisation |
| IT | Information Technology |
| ITSEC | IT Security Evaluation Criteria |
| IVDP | Interactive Video/Direct Print |
| Job | User job |
| Kernel | The part of VME which handles real resources |
| LAN | Local Area Network |
| Library | Collection of data files with similar properties |
| Libraryfile | Data file belonging to a library |
| Librarylist | List of libraries to be used as a context for module loading |
| Location | Site of an object (hardware, service etc.) |
| MAC | Multi-Access Computing |
| MB | Megabytes |
| MHS | Message Handling Service |
| Middleware | Infrastructure provided which insulates applications from their environment and the way they are distributed |
| MIS | Management Information Service. Allows managers, and others, to access information in ways that are not pre-determined. |
| MIT | Massachusetts Institute of Technology |
| MTA | X.400 Message Transfer Agent |
| NIFTP | Network Independent File Transfer Protocol |
| OCF | Operator Communications File |
| OCM | Operations Control Monitor |
| OCP | Order Code Processor |
| ODP | Open Distributed Processing, based on the reference architecture defined in ISO/IEC 10746. |
| OLTP | Online Transaction Processing |
| OMF | Object Module Format |
| OMG | Object Management Group |
| OPEH | Object Program Error Handler |
| OPENframework | ICL's System Integration architecture and method |
| OSF | Open Software Foundation |

| | |
|---|---|
| OSI | Open Systems Interconnection |
| OSI-TP | OSI standard for transaction management protocols |
| OSLAN | Open Systems LAN |
| OVEC | ICL's OpenVME Exploitation Centre |
| P2P | Peer to Peer |
| PAC | Privilege Attribute Certificate |
| PAS | Program Activity Sampler |
| PC | Personal Computer |
| PCTE | Portable Common Tools Environment |
| PFI | Physical File Interface |
| PID | VME Personal Identification Option |
| Pipe | A file connecting two processes, one writing, one reading |
| Platform | A collection of hardware and software components with the ability to process and store information. |
| PLI | Primitive Level Interface |
| PLT | Procedure Linkage Table |
| POSIX | Standards for operating system interfaces defined by IEEE. |
| Primitive File | A file viewed as a sequence of physical data blocks |
| Profile | A service or service description |
| PS | Access Manager Person Server |
| QB | ICL's QuickBuild |
| QBP | QuickBuild Pathway |
| QBWB | QuickBuild Work Bench |
| RAM | Record Access Mechanism |
| RDA | Remote Database Access |
| RDBMS | Relational Database Management System |
| Real File | Data file as a set of blocks |
| RECMAN | VME's Record Management software |
| ReView | A means of providing relational access to all OpenVME data sources |
| RIBA | Distributed Computing programme in which ICL is a collaborator |
| RISC | Reduced Instruction Set Computer |
| RPC | Remote Procedure Call |
| RSA | Remote Session Access |
| RSI | VME Restricted System Interface |
| SAP | Service Access Point |
| SCL | System Control Language |
| SCT | VME's Service Connection Task |
| Service | A functional capability |
| SESAME | Consortium developing the ECMA security architecture |
| SMF | Standard Monitor Facility |
| SNA | IBM's System Network Architecture |
| SNMP | Simple Network Management Protocol |
| SQL | Structured Query Language |
| SSADM | Structured Specification and Design Method |
| SVR4 | (UNIX) System V Release 4 |
| Task | A VM customised for a specific purpose |

| | |
|---|---|
| TCB | Trusted Computing Base |
| TCB | Transfer Control Block |
| TCP/IP | Transmission Control Protocol/Internet Protocol |
| TCSEC | Trusted Computer Security Evaluation Criteria |
| TP | Transaction Processing |
| TPMSX | ICL's Transaction Processing Management System |
| TxRPC | Transactional RPC |
| UCG | User Code Guardian |
| UDP/IP | User Datagram Protocol/Internet Protocol |
| UES | Access Manager User Environment Server |
| US | Access Manager User Sponsor |
| User Object | Object whose content is user-defined |
| UUCP | UNIX file transfer protocol |
| Virtual File | Virtual file mapped onto one or more real files |
| VM | Virtual Machine |
| VME | Virtual Machine Environment |
| VME-X | OpenVME Application Environment which supports X/Open interfaces, together with other key practical portability interfaces. |
| VTP | OSI Virtual Terminal Protocol |
| X/Open | A joint initiative by members of the information technology community (suppliers and users) to adopt, promote and adapt open standards. |
| X25 | A communications wide-area network protocol |
| XATMI | X/Open Application Transaction Manager Interface |
| XDCS | X/Open Distributed Computing Services |
| XDR | External Data Representation ( system independent data representation) |
| XDS | X/Open Directory Service |
| XMA | X/Open Message Access API |
| XPGn | X/Open Portability Guide Issue n |
| XSI | X/Open System Interface |
| XTI | X/Open Transport Interface |

# Appendix C:
# Catalogue Object Types

## Catalogue Object types

The VME Catalogue supports the following Object types:

| | |
|---|---|
| User (or Person) | User workgroup or individual identification |
| File Group | Group of Files, Libraries, Filegroups |
| Data File | A file with associated file or record organisation |
| Primitive File | A file viewed as a sequence of physical data blocks |
| File Section | Section of a partitioned file (e.g. reel of multi-reel |
| Filestore Category | An aggregation of filestore managed similarly |
| Filestore Volume | Disc volume (or partition thereof), magnetic tape |
| Controlling File | File used to record file allocation and placement |
| Event | Object supporting inter-process communication & |
| Configuration Block | |
| Hardware Unit | Central units, peripherals, other systems on local |
| Installation | The root for enumerating objects in an installation |
| Virtual File | Virtual file mapped onto one or more real files |
| User Object | Object whose content is user-defined |
| Location | Site of an object (hardware, service etc.) |
| Profile or Service | A service or service description |
| Task | Description of a VM customised for a specific |
| Service Access Point | Element of communications route & addressing |
| Job | User job |
| Library | Collection of data files with similar properties |
| Libraryfile | Data file belonging to a library |
| Account | Collection point for accounting & charging |
| Librarylist | List of libraries to be used as a context for module |
| Real File | Data file as a set of blocks |
| Filestore Description | Description of an allocation of filestore space |

# Appendix D:
# List of OpenVME Subsystems

| | |
|---|---|
| ACC | ACCOUNTING_MANAGER |
| ACO | ADVANCED_CAFS_OPTION |
| ACSE | AS_CONTROL_SERVICE_ELEMENT |
| ADH | APPL_DIALOGUE_HANDLER |
| AEAM | APP_ENTITY_AREA_MANAGER |
| AEIH | APPL_ENTITY_INVOCATION_HANDLER |
| AH | ASSOCIATION_HANDLER |
| AK | ALTERNATIVE_KEYS |
| AM | AREA_MANAGER |
| ANAL | NETSIM_ANALYSIS |
| AOF | AUTOMATIC_OPERATOR_FACILITY |
| ASR | ADDRESS_SPACE_AND_REGION_HANDLER |
| ATAF | APPL_TO_APPL_FACILITY |
| ATH | ASYNC_TERMINAL_HANDLER |
| ATMI | XATMI |
| AU | AUTOMATIC_UPGRADE |
| BA | BA_BLOCK_ACCESS_RAM |
| BC | BC_BULK_COPY |
| BL | SYSTEM_B_LOADER |
| BM | BUFFER_MANAGER |
| BR | BR_BASE_RESPONDER |
| BRF | BROWSE_FILE_UTILITY |
| BRS | BROWSER |
| BS | BS_OUTER_SCL |
| BUC | BUC_BUDGET_CONTROL |
| CD | CD_CREATE_DISC_HOLONS |
| CDAM | DISTRIB_AND_RECOVER_MGR |
| CDDJ | CREATE_DELETE_DIRECTOR_JOB |
| CDH | COMMUNICATIONS_DEVICE_HANDLER |
| CG | CG_CAFS_GENERATE |
| CH | CATALOGUE_HANDLER |
| CHAF | C_HEADERS_AND_FUNCTIONS |
| CHC | CHC_CHARGE_CONTROL |
| CLI | CLI |
| CME | CONCURRENT_MACHINE_ENVIRONMENT |
| CMMN | COMMON_FACILITIES |
| CNC | COMMUNICATIONS_NETWORK_CONTROL |
| COCO | COMMITMENT_COORDINATOR |
| COLL | COLLECTOR |
| COM | COM_FORMATTER |
| CPM | CPM_COMMS_PROCESSOR_MANAGER |
| CRC | CHECKPOINT_RESTART_CONTROLLER |
| CSA | CONCURRENT_SESSION_ACCESS |
| CSI | COMMS_SERVICE_INFRASTRUCTURE |
| CTU | CMMN_TOOLS_UTILITIES |
| CUM | CENTRAL_UNIT_MANAGER |
| CV | CV_CORRELATE_VOLUME_FACILITIES |
| DA | DA_OPERFY |
| DAN | DIAGNOSTIC_ANALYSER |
| DC | DC_DIRECTOR_COMMUNICATIONS |
| DCY | DUMP_COPY |
| DEM | DIRECTOR_ERROR_MANAGER |
| DES | DIAGNOSTIC_EVIDENCE_SYSTEM |
| DEVM | DIRECTOR_EVENT_MANAGER |
| DH | DEVICE_HANDLER |
| DIF | DATA_INTERCHANGE_FUNCTION |
| DIR | OSI_DIRECTORY_PRODUCT |
| DM | DISPLAY_MANAGER |
| DMG | DMG_DUMP_MANAGER |
| DN | DN_DUMP_ANALYSER |
| DSA | DIRECTORY_SYSTEM_AGENT |
| DSG | DATABASE_SYSTEM_GENERATOR |
| DSH | DIRECTOR_SEMAPHORE_HANDLER |
| ECFH | OCF_HANDLER |
| EDH | ENCODE_DECODE_HANDLER |
| EFSM | EMBEDDED_FILESTORE_MANAGER |
| ELAN | ELAN_ETHERNET_MANAGER |
| ELB | ENTRY_LEVEL_B |
| EM | EM_UPPER_EVENT_MANAGER |
| ER | ER_ERROR_MANAGER |
| ETH | ECMA_72_TRANSPORT_HANDLER |
| EXIM | EXPORT_IMPORT |
| EXS | EXECUTION_SCHEDULER |
| FAH | FORMS_AND_HELP |
| FC | FILE_CONTROLLER |
| FCD | FCD_CATALOGUE_DECATALOGUE |
| FCH | FILE_CONTINGENCY_HANDLER |
| FD | FD_FILE_DESCRIPTION_HANDLER |
| FDS | FORMS_DEVELOPMENT_SYSTEM |
| FG | FG_FILEGROUP_MANAGER |
| FID | FORMS_INTERFACE_DESIGNER |
| FL | FL_FILE_SERVICES |
| FMFC | FORM_FILE_CMDS |
| FMS | FMS_FILESTORE_MANAGEMENT_SCHEME |
| FORM | FORM_CALL_MANAGER |
| FP | FTAM_PROTOCOL_MACHINES |
| FPWH | FPWH_FTAM_WORM_HANDLER |
| FRE | FRE_FILE_RESILIENCE |
| FRS | FORMS_RUNTIME_SYSTEM |
| FSR | FILE_STORE_ROUTINES |
| FT | FILE_TRANSFER |
| FTAM | FTAM_TRANSFER_APPLICATION |
| FTC | FTC_FTAM_COORDINATOR |
| FTH | FLEXIBLE_TERMINAL_HANDLER |
| FTS | FORMS_TERMINAL |
| FU | FU_FILE_UTILITIES |
| GXD | GXD_GIVE_DETAILS_MANAGER |
| HM | HM_HARDWARE_MANAGER |
| IC | INSTALLATION_AND_CHANGE_HANDLER |
| ICM | INTRA_SYSTEM_CONNECTION_MANAGER |
| IDH | IDH_INTERCHANGE_DEVICE_HANDLER |
| IEH | INITIAL_ENTRY_HELP |
| IFH | INTERMEDIATE_FILE_HANDLER |
| IFS | INTERNAL_FILE_SERVICE |
| ILAN | INTERNODE_LAN_AND_COUPLER_MGR |
| INH | INTERNET_HANDLER |
| IPS | INPUT_SCHEDULER |
| IS | INDEX_SEQUENTIAL |
| ISDA | DDS_SCREEN_DESIGNER |
| IXC | INDEX_CONSTRUCTOR |
| JI7 | JI_MACROS |
| JI8 | JI_OSCL_PROCEDURES |
| JI9 | JI_BATCH_JOB_FUNCTIONS |
| JS | JOB_SCHEDULER |
| KC | KC_KERNEL_COMMUNICATIONS |
| KEM | KERNEL_ERROR_MANAGER |
| KEVM | KERNEL_EVENT_MANAGER |
| KL | KERNEL_LOADER |
| KMH | KMH_MICROPROGRAM_HANDLER |
| KRM | KERNEL_RECONFIGURATION_MANAGER |
| KSH | KERNEL_SEMAPHORE_HANDLER |
| LC | LIBRARY_CONTROLLER |
| LLSM | LOW_LEVEL_LAN_STATION_MANAGER |
| LM | LM_LOCK_MANAGER |
| LOCM | LOCATION_MANAGER |
| LSH | LAN_STATION_HANDLER |
| LSM | LOADSET_MANAGER |
| LST | LST_LISTINGS_MANAGER |
| LU | LU_LIBRARY_UTILITIES |
| MA | MESSAGE_ACCESS |

| | |
|---|---|
| MAC | MAC_SCHEDULER |
| MAL | MONITOR_ANALYSER |
| MCM | MONITOR_COLLECTION_MANAGER |
| MCOM | MONITOR_COUNTER_MANAGER |
| MEH | MEDIA_HANDLER |
| MF | MAPPED_FILE_RAM |
| MH | MODULE_HANDLER |
| MLAN | MLAN_MACROLAN_MANAGER |
| MLM | MONITOR_LOGGING_MANAGER |
| MLND | MLND_MLAN_DIAGNOSTICS |
| MM | MAGNETIC_MEDIA_PHYSICAL_FILE_MGR |
| MMAM | MAMPHY_ACCESS_MANAGER |
| MMDI | MAMPHY_DISC_IMPERATIVE_MGR |
| MMDV | MAMPHY_DISC_VOLUME_MGR |
| MMI | VISIONMASTER_MAC |
| MMUM | MAMPHY_UNIT_MANAGER |
| MNM | MULTI_NODE_MANAGER |
| MPT | MPT_MODULE_PROCESSING_TOOLS |
| MR | MR_MAC_RESPONDER |
| MSS | MONITOR_SYSTEM_SAMPLER |
| NCAT | NETWORK_CATALOGUE_HANDLER |
| NCM | NETWORK_CONFIGURATION_MANAGER |
| NCOL | NETWORK_OLS_CONTROLLER |
| NCUD | NETWORK_CONNECTION_UD_HANDLER |
| NFH | NETWORK_FILE_HANDLER |
| NM | NM_NODE_MANAGER |
| NSUN | NETSIM_SUNDRIES |
| NSXD | NSXD_DRIVERS |
| NXC | NXCMAN_TAPE_SYSTEM |
| ODH | OPER2_DEVICE_HANDLER |
| OFM | OCF_FILE_MANAGER |
| OM | OBJECT_MANAGEMENT |
| OPS | OUTPUT_SCHEDULER |
| OR | OR_OPERATOR_RESPONDER |
| OS_5 | OS_MACROS |
| OS1 | OUTPUT_SPOOLER |
| OS6 | OS_MACROS_SUPPORT_PROCS |
| OSCL | OSCL_STANDARD_OUTER_SCL |
| OSP | OUTPUT_SPOOLING |
| OTP | OSI_TRANSACTION_PROCESSING |
| PCH | PROGRAM_CHECKPOINTING |
| PCXA | PCXA |
| PERM | PERIPHERAL_MANAGER |
| PFFC | FORM_TEMPLATE_UTILS |
| PFFM | PFFM_FORMS_MERGER |
| PFMS | PFM_SERVICES |
| PLAN | PUBLIC_WRITE_LAN_COUPLER_MGR |
| PM | PROCESS_MANAGER |
| POF | PROGRAMMABLE_OPERATOR_FACILITY |
| PR | PROMPTER_MECHANISM |
| PREP | NETSIM_PREPARATION |
| PS | PACKAGE_SELECTION |
| PSI | PRESENTATION_SERVICE |
| PSP | PAGE_ORIENTED_SPOOLER |
| PT | PETE |
| PTH | PURSUIT_TERMINAL_HANDLER |
| PTR | PRINTFILE_TRANSF_RAM |
| PV | PV_PRIVACY_HANDLER |
| PWM | PUBLIC_WRITE_MANAGER |
| QISH | QUICK_INTRA_SYSTEM_HANDLER |
| QMH | QUICK_MESSAGE_HANDLER |
| RBS | RBS_RME_BASE_SUPPORT |
| RC | RC_RAM |
| RCH | RCH_REMOTE_COUPLER_HANDLER |
| RD | RD_DRIVERS |
| RF | RF_SLOW_DEVICE_RAM |
| RG | RG_REPORT_GENERATOR |
| RGT | RSA_GATEWAY_TASK |
| RI | RESOURCE_INTERFACE_MANAGER |
| RLY | TRANSPORT_RELAY |
| RM | RM_RESOURCE_MANAGER |
| RMF | RMF_MAPPED_FILE_MANAGER |
| RORO | ROLLON_ROLLOFF_MANAGER |
| ROS | REMOTE_OPERATION_SERVICE_ELEMENT |
| ROSE | ROSE |
| RR | RR_FILE_RECOVERY |
| RS | RESOURCE_SCHEDULER |
| RT | RT_TAG_MANAGER |
| RU | DROUTE_UTILITIES |
| RUN | NETSIM_RUN |
| SA | SEGMENT_ALLOCATOR |
| SAH | SHARED_ACCESS_HANDLER |
| SAM | SAM_INTERFACE |
| SCHH | SCHH_SECURITY_HANDLER |
| SCLS | SCL_SYSTEM |
| SD | SD_PHYSICAL_FILE_MANAGER |
| SDH | SLOW_DEVICE_HANDLER |
| SDM | SDM_SERIAL_DEVICE_MANAGER |
| SFA | NETSIM_SCREEN_FORMAT_ANAL |
| SL1 | SL_MACROS |
| SL2 | SL_PROCEDURES |
| SM | SM_SECTION_MANAGER |
| SMAN | SERVER_MANAGER |
| SMF | STANDARD_MONITORING_FACILITY |
| SMON | SMON_SYSTEM_MONITOR |
| SPR | SPR_SYSTEM_PROMPTER |
| SQ | SYSTEM_QUEUEING |
| ST | STATISTICS_PACKAGE |
| STM | STM_STREAM_MANAGER |
| STR | VME_STREAMS |
| STW | SYSTEM_TASK_WATCHDOG |
| SVC | SERVICE_CONNECTION |
| SVL | SUPERVISOR_LOADER |
| SVM | SERVICE_MANAGER |
| SW | SPOOL_WRITER |
| TC | TASK_CONTROLLER |
| TDDH | TRANSPARENT_DIRECT_DEV_HNLR |
| TDH | TRANSPARENT_DEVICE_HANDLER |
| TI | TI_TRANSPORT_INTERFACE |
| TIM | TIMER_MANAGER |
| TLB | TLB_MAG_TAPE_LIBRARY_APPL |
| TM | TAG_MANAGER |
| TME | TME_EMULATION |
| TP | TP_SCHEDULER |
| TPA | TP_ADAPTOR |
| TR | TR_RECORD_TRANSFORMATION |
| TRUG | SYSTEM_UPGRADE |
| TSM | TRANSPORT_SERVICE_MANAGER |
| TST | TOTAL_SYSTEM_TELESERVICE |
| TU | TU_TAG_UTILITIES |
| TXA9 | TX_ACCESS_LEVEL_9 |
| TXAC | TX_ACE |
| TXAP | TX_APPLICATION |
| TXCO | TX_CONTROL |
| TXDA | TX_DATA |
| TXDE | TX_DEVICE |
| TXIN | TX_INTERFACE |
| TXIT | TX_INST |
| TXJO | TX_JOURNAL |
| TXMA | TX_MATS |
| TXMC | TX_MAIN |
| TXPH | TX_PHAN |
| TXPR | TX_PREPARE |
| TXSL | TX_SLOTFILE |
| TXSM | TX_MONITORING |
| TXSP | TX_SPOOLER |
| TXST | TX_STATS |
| TXTE | TX_TESTING |
| TXVM | TX_VMS |
| TXWD | TX_WIP_STORE_DRIVER |
| TXXV | TX_AUXILIARY_VM |
| UCG | USER_CODE_GUARDIAN |
| UCI | UNIX_COMPATIBILITY_INTERFACES |
| UIM | USER_INTERFACE_MANAGER |
| UM | UM_USER_MANAGER |
| UO | UO_USER_OBJECT_MANAGER |
| UPG | UPG_UPGRADE_PROMPTER |
| USE | USER_SUPPORT |

| | |
|---|---|
| UTM | UPPER_TAG_MANAGER |
| VC | VC_DISC_VOLUME_COPIER |
| VCAP | VCAP_COMMON_APPLICATIONS |
| VCUT | VCUT_COMMON_UTILITIES |
| VDD | VDD_DISC_DISPLAY_FUNCTIONS |
| VDF | DUMP_FORMATTER |
| VDH | VM_DESCRIPTION_HANDLER |
| VF | VIRTUAL_FILES |
| VFS | FTAM_VIRTUAL_FILESTORE |
| VG | VI_VOLUTS |
| VHPC | VHSDC_PROTOCOL_DRIVERS |
| VHUD | VHSDC_UNIT_DRIVERS |
| VI | VI_DISC_INITIALISER |
| VISA | VISA_KERNEL |
| VMEP | VME_PATHWAY |
| VMEX | VME_XOPEN |
| VMI | VIRTUAL_MACHINE_INITIALISATION |
| VMM | VIRTUAL_MACHINE_MANAGER |
| VMPX | VME_PATHWAY_EXTENDED |
| VOD | VOD_ARCHIVE_HANDLER |
| VOLM | VOLUME_MANAGER |
| VP | VP_PRINT_TAPE_FUNCTIONS |
| VPU | VOLUME_PARTITION_UTILITIES |
| VR | VR_VIRTUAL_FILE_MANAGER |
| VSC | VSC_SWEEP_CYCLE_UTILITIES |
| VSM | VIRTUAL_STORE_MANAGER |
| VTC | VTC_TAPE_FUNCTIONS |
| VTD | VTD_TIDY_DISC_FUNCTIONS |
| VTF | VIRTUAL_TERMINAL_FORMS |
| VTI | VME_TCP_IP |
| VTPM | VIRTUAL_TERMINAL_PROTOCOL_M |
| VWM | WINDOW_MANAGEMENT |
| VX | VX_IO_HANDLER |
| VY | VY_COMMON_HOLONS |
| VZ | VOLUTS_GI_IO_HANDLER |
| WI | WORK_INTRODUCTION |
| WM | WORK_MANAGEMENT |
| WMT | WORKMIX_TASK |
| WMX | WORKMIX |
| WSM | WIP_STORE_MANAGER |
| X25H | X25_HANDLER |
| X400 | X400_CODE_HANDLER |
| X4M | X4MAIL |
| X4SP | X400_REMOTE_SPONSOR |
| XAP | XOPEN_ACSE_PRESENTATION_API |
| XCV | X_OPEN_COMMANDS_FROM_VME |
| XFM | XFM_VMEX_FILE_MANAGER |
| XTI | XOPEN_TRANSPORT_INTERFACE |

# Appendix E: Bibliography

## OpenVME Customer Publications

Full details of all OpernVME customer publications are available in the Publications Catalogue. The majority of OpenVME customer publicationas are also available on the OptICL OpenVME CD-ROM product.

### Overview

| | |
|---|---|
| Introduction to OpenVME | 58548 |

### Application Environment

| | |
|---|---|
| Advanced CAFS Option: User Guide | 54824 |
| Batch Recovery Techniques | 59550 |
| DAIS: System Overview | R30428 |
| Distributed Applications on VME | 30366 |
| Open VME Application Porting Guide | 54438 |
| RSI Option: Compiler Target Machine | 52102 |
| RSI Option: Extended Target Machine | 51531 |
| RSI Option: Magnetic Physical File Interface | 54494 |
| RSI Option: Non Magnetic Physical File Interface | 56911 |
| RSI Option: Primitive Level Interface | 51078 |
| RSI Option: Work Management | 51444 |
| VME-X Programmers Reference Manual | 55737 |
| Vocabulary of SCL Command Specifications | 52069 |

### Application Development

| | |
|---|---|
| Additional Dictionary Facilities (DDS.850) | 10397 |
| Application Analysis: An Interactive Guide | 16397 |
| Application Master: Reference (AM.300) | 15420 |
| Data Dictionary System: Summary | 17146 |
| FORMS COBOL Programmers Guide | 12340 |
| FORMS Overview | 18606 |
| FORMS Implementors Guide | 19455 |
| FORMS Interface Designers Guide | 12366 |
| FORMS Introduction to Interface Design | 12056 |
| FORMS Terminal: Users Guide | 56437 |
| FORTRAN 77 Language | R03792 |
| Object Module Format and Utilities | 55843 |
| Programmer's Workbench | 13692 |
| Programmer's Guide | 54571 |
| Programming Utilities | 58310 |
| Querymaster Advice and Guidance | 18013 |
| QuickBuild Exploitation Guide | 18862 |
| QuickBuild Overview (Series 39) | 11124 |
| QuickBuild: Developing Applications (AM.300) | 18653 |

| | |
|---|---|
| Quickbuild Pathway & ASG Reference | 17157 |
| QuickBuild: Planning and Implementing a System | 18530 |
| Running and Using a Dictionary Service (DDS.850) | 14558 |
| SCL Programming | 51776 |
| Screen Design Using DDS | R00400 |
| System Programming | 50440 |
| The DDS Model (DDS.850) | R00461 |
| Using DDS System Definition Language (DDS 800/850) | R00419 |
| Using the COBOL C2 Compiler | R00173 |
| Using the VME C Compiler | R50324 |
| VME/B Environment Option | 57586 |
| VME COBOL Range COBOL Language (C2) | R03790 |
| VME COBOL Range COBOL Syntax (C2) | R03791 |
| VME S3 Language | R00084 |

## Transaction Management & TPMSX

| | |
|---|---|
| Designing a TPMS Service | R00460 |
| Implementing a TPMS Service | R00462 |
| Introduction To TPMS | R00458 |
| Specifying a TPMS Service as a DDS Model | R00463 |
| Specifying a TPMS Service Using the Parameter File | R00464 |
| TPMS (XE) | 57685 |
| TPMS (XL) Option | R30168 |
| TPMS Global Contents and Glossary Of Terms | R00459 |
| TPMS Service Control and Maintenance | R00465 |
| Writing COBOL Applications for a TPMS Service | R00461 |

## Information Management

| | |
|---|---|
| IDMSX : Using Data Display | R00110 |
| IDMSX Easychange User Guide | R30236 |
| IDMSX High Performance Option User Guide | R30235 |
| IDMSX Part 1: Setting up a Simple Database | R00176 |
| IDMSX Part 2: Database Establishment | R00154 |
| IDMSX Part 3: Using a Database | R00155 |
| IDMSX Part 4: Database Programming | R00156 |
| IDMSX Part 5: Database Design | R00153 |
| Querymaster: Using Querymaster | R00433 |
| ReView User Guide | 59251 |
| VME INGRES Installation & Operations Guide | 19708 |
| VME INGRES/SQL Reference Manual | 16924 |
| VME INGRES/Embedded SQL for C | 12147 |
| VME INGRES/Embedded SQL for COBOL | 18473 |
| VME INGRES System Commands Reference Manual | 14696 |
| VME INGRES Database Administrators Guide | 13798 |
| VME INGRES Command Reference Summary | 16948 |
| VME INGRES/REPORTS: Report Writer Reference Manual | 14352 |

## Networking Services

| | |
|---|---|
| File Transfer | 57143 |
| Flexible Terminal Handler Asynchronous Sponsor Package | 50946 |
| Flexible Terminal Handler: Infrastructure | 53003 |
| Flexible Terminal Handler: Sponsor Writers Guide | 56630 |
| Introduction to VME Communications | 53002 |
| Implementing Local Area Networks | 52744 |
| Message Handling System | 57030 |
| TCP/IP protocols on OpenVME | 55343 |
| Terminal Access | 55078 |
| Wide Area Networks: Implementing Full XBM | 50635 |
| Wide Area Networks: Implementing Integrated X.25 | 50802 |

## System Management

| | |
|---|---|
| Automated System Operator (ASO) User Guide | 51273 |
| Accounting Charging and Budgeting | 52099 |
| FORMS Terminal: Administrators Guide | 50229 |
| Moving Workloads | 58177 |
| SAM Reference Manual | 53845 |
| System Exploitation | 52035 |
| System Management | 52397 |
| System Management Dictionary | 58416 |
| System Operating | 51092 |
| System Performance | 54496 |
| System Prompts | 59345 |
| The Filestore Management Scheme | 52185 |
| Work Scheduling | 55250 |
| VME-X System Administrator's Reference Manual | 58728 |
| VME-X User Guide | 55047 |
| VME-X Users Reference Manual | 52124 |

## Series 39 Hardware

| | |
|---|---|
| Advanced CAFS Option | R30130 |
| Guide to CAFS Exploitation | R30053 |
| Hardware Reference Manual | 50296 |

## Security

| | |
|---|---|
| Provding an ICL Access Service (VME) | 12244 |
| System Security | 52807 |
| Security Options | 52650 |

## OPEN*framework* Publications

The following books are published by Prentice Hall and cover the qualities, elements and specialisations of OPEN*framework*:

### Overview

| | |
|---|---|
| The Systems Architecture: an Introduction | ISBN 0-13-560186-X |

### Qualities

| | |
|---|---|
| Availability | ISBN 0-13-630948-8 |
| Usability | ISBN 0-13-630930-5 |
| Performance | ISBN 0-13-630666-7 |
| Security | ISBN 0-13-630658-6 |
| Potential for Change | ISBN 0-13-630617-9 |

### Elements

| | |
|---|---|
| User Interface | ISBN 0-13-630591-1 |
| Distributed Application Services | ISBN 0-13-630518-0 |
| Information Management | ISBN 0-13-630500-8 |
| Application Development | ISBN 0-13-630484-2 |
| Systems Management | ISBN 0-13-630450-8 |
| Networking Services | ISBN 0-13-630393-5 |
| Platforms | ISBN 0-13-630385-4 |

### Specialisations

| | |
|---|---|
| Services | ISBN 0-13-101213-4 |
| Transaction Management | ISBN 0-13-630377-3 |

## X/Open Publications

### Overviews

| | |
|---|---|
| An Introduction to X/Open and XPG4 (by CCTA) | ISBN 1-872630-68-5 |
| Open Systems in Practice | ISSN 0966-8063 |
| Open Systems: A Guide to Developing the Business Case | ISBN 1-872630-72-3 |
| X/Open and Interoperability (by Petr Janecek) | ISBN 1-872630-57-X |
| X/Open and Open Systems (by Colin Taylor) | ISBN 1-872630-55-3 |
| X/Open Systems and Branded Products: XPG4 | ISBN 1-872630-52-9 |

### Base Definition

| | |
|---|---|
| Commands and Utilities Issue 4 | ISBN 1-872630-48-0 |
| System Interface Definitions Issue 4 | ISBN 1-872630-46-4 |
| System Interfaces and Headers Issue 4 | ISBN 1-872630-47-2 |

## Communications & Interworking

| | |
|---|---|
| ACSE/Presentation Services API (XAP) | ISBN 1-872630-91-X |
| API to Directory Services (XDS) | ISBN 1-872630-18-9 |
| API to Electronic Mail (X.400) | ISBN 1-872630-19-7 |
| Byte Stream File Transfer (BSFT) | ISBN 1-872630-27-8 |
| EDI Messaging Package (XEDI) | ISBN 1-872630-25-1 |
| FTAM High-level API (XFTAM) | ISBN 1-872630-60-X |
| Guide to IPS - OSI Coexistence and Migration | ISBN 1-872630-22-7 |
| Guide to Selected X.400 and Directory Services APIs | ISBN 1-872630-23-5 |
| Guide to the Internet Protocol Suite | ISBN 1-872630-08-1 |
| Message Store API (XMS) | ISBN 1-872630-83-9 |
| Networking Services Issue 3 | ISBN 1-872630-42-1 |
| OSI-Abstract-Data Manipulation API (XOM) | ISBN 1-872630-17-0 |
| Protocols for X/Open Interworking: XNFS Issue 4 | ISBN 1-872630-66-9 |
| Protocols for X/Open PC Interworking: (PC)NFS | ISBN 1-872630-00-6 |
| X.400 (APIs and EDI Messaging) - | ISBN 1-872630-78-2 |
| X/Open Transport Interface (XTI) | ISBN 1-872630-29-4 |

## Data Management

| | |
|---|---|
| Data Management Issue 3 | ISBN 1-872630-40-5 |
| Data Management: SQL Call Level Interface (CLI) | ISBN 1-872630-63-4 |
| Data Management: SQL Remote Database Access | ISBN 1-872630-98-7 |
| Indexed Sequential Access Method (ISAM) | ISBN 1-872630-03-0 |
| Structured Query Language (SQL) | ISBN 1-872630-58-8 |

## Distributed Computing

| | |
|---|---|
| Common Object Request Broker: Architecture and Specification | ISBN 1-872630-90-1 |
| Distributed Computing Services (XDCS) Framework | ISBN 1-872630-64-2 |
| Distributed Internationalisation Services | ISBN 1-872630-75-8 |

## Languages

| | |
|---|---|
| COBOL Language | ISBN 1-872630-09-X |
| Programming Languages Issue 3 | ISBN 1-872630-39-1 |

## Miscellaneous

| | |
|---|---|
| Document Interchange Formats | ISBN 1-872630-51-0 |
| Document Interchange Reference Model | ISBN 1-872630-50-2 |
| Internationalisation Guide - Version 2 | ISBN 1-859120-02-4 |
| Internationalisation of Interworking Specifications | ISBN 1-872630-87-1 |

## Transaction Processing

| | |
|---|---|
| Communication Resource Manager Specifications | ISBN 1-872630-88-X |
| CPI-C | ISBN 1-872630-35-9 |
| Distributed TP: Reference Model | ISBN 1-872630-16-2 |
| Distributed TP: The Peer-to-Peer Specification | ISBN 1-872630-79-0 |
| Distributed TP: The TX Specification | ISBN 1-872630-65-0 |
| Distributed TP: The TxRPC Specification | ISBN 1-859120-00-8 |
| Distributed TP: The XA Specification | ISBN 1-872630-24-3 |
| Distributed TP: The XA+ Specification | ISBN 1-872630-93-6 |
| Distributed TP: The XATMI Specification | ISBN 1-872630-99-5 |
| Transaction Processing | ISBN 1-872630-89-8 |

## System Management

| | |
|---|---|
| Systems Management: Managed Object Guide (XMOG) | ISBN 1-859120-06-7 |
| Systems Management: Reference Model | ISBN 1-859120-05-9 |

## User Interface

| | |
|---|---|
| Window Management Issue 3 | ISBN 1-872630-41-3 |
| X Toolkit Intrinsics | ISBN 1-872630-14-6 |
| X Window s File Formats & Application Conventions | ISBN 1-872630-15-4 |
| X Window System Protocol | ISBN 1-872630-13-8 |
| Xlib - C Language Binding | ISBN 1-872630-11-1 |

## XPG4

| | |
|---|---|
| XPG4 Branded Products Part 7 | ISBN 1-873620-52-9 |
| XPG4 Component Defns. Part 2 | ISBN 1-872630-52-9 |
| XPG4 CSQs Part 6 | ISBN 1-872630-52-9 |
| XPG4 Guide to Branding Part 4 | ISBN 1-872630-52-9 |
| XPG4 Overview Part 1 | ISBN 1-872630-52-9 |
| XPG4 Profile Defns. Part 3 | ISBN 1-872630-52-9 |
| XPG4 TMLA Part 5 X/Open Systems & Branded Products | ISBN 1-872630-52-9 |

## Additional Sources

### ISO Reference Documents

| | |
|---|---|
| Open Systems Interconnection: Basic Reference Model | ISO 7498 |
| Systems Management Overview | ISO 10040 |
| Systems Management Functions | ISO 10164 |
| Structure of Management Information | ISO 10165 |
| Distributed Transaction Processing (OSI-TP) | ISO 10026 |
| OSI Remote Procedure Call | ISO 11578 |
| The Open Distributed Processing Reference Model | ISO 10746 |

### OSF Documents

| | |
|---|---|
| OSF/1 Operating System Overview | OSF-OS-PD-1190-1 |
| OSF DCE Overview | OSF-DCE-PD-1090-4 |
| OSF DME Overview | O-DME-RD-1 |

### OMG Documents

Object Management Group Architecture Guide
Object Model
The Common Object Request Broker: Architecture & Specification

### GOSIP Standards

Government OSI Profile Specification

### Miscellaneous Documents

| | |
|---|---|
| Security Framework for the Application Layer of Open Systems | ECMA TR/46 |
| Reference Model for Frameworks of CASE | ECMA TR/55 |

# Index