DATE: 16 March 1987

MICA Public Naming Conventions

PREFACE

Associated Documents:

Change History:

Revision	Date	Author/Modification
0.1	4-DEC-1986	Benn Schreiber/Original
0.2	16-JAN-1987	Benn Schreiber/Incorporate review comments.
0.3	1-MAR-1987	Benn Schreiber/comments from public review.

1 SUMMARY

This chapter defines the naming conventions to be used for names accessible from user-mode programs throughout MICA. Naming conventions are important for several reasons.

- Present a consistent, easy-to-remember name space to users and developers.
- Ensure that system software uses consistent naming to aid future developers in maintaining and extending the software.
- Well-defined naming conventions ensure that customer-written software is not invalidated by future releases of DEC products which add new symbols.
- Facilitate straightforward usage within PILLAR. The names will be similarly usable in all other Digital-supported languages.

2 SCOPE

This document covers the public naming conventions for

- Module names - The names assigned to program source modules.
- System services and system routines - The names of the system routines themselves and the names of the routine arguments.
- Files and Directories - The format for naming files which constitute the system software.
- Facility names - Specifying the software facility name based on the product name.
- Data structures - Naming records and the fields within the records.
- Types - PILLAR named types.
- Constants - Compile-time named constants.
- Message names - Symbols which define unique message values.
- Logical names - System or Group logical names used to alter, define or control a facility.
- Compile-time facility macros and procedures.

These are discussed in the following sections. The guidelines and conventions in this document cover all public software interfaces for

layered products as well as bundled MICA software.

3 NAMING GUIDELINES

As a general rule, names should not be short acronyms; rather the full English word(s) should be used. For instance, a parameter representing the desired access mode should be named "access_mode" rather than "acmode".

If the name consists of more than one word, the words should be separated with the underscore ("_") character.

There are, of course, going to be exceptions to this guideline. These exceptions will mainly be caused by the English name for something being too long to reasonably represent (larger than the maximum allowed symbol length \This value TBS\). In such cases, it is up to the engineer to use creative good judgment and derive an acceptable name that is not difficult to remember.

All Digital-supplied public symbols which can be referenced by users (and where the scope of the symbols overlaps with the user namespace) will contain a currency ("$") sign. Users should not use the currency sign in their definitions (except as noted in $$section(procedures); users may use "$$" to denote a non-public global procedure) to ensure separation of name spaces and avoid naming collisions in future system software releases.

When something is named in several different places throughout the system, it will have the same name. For instance, all services which accept an event object id as an argument will name the argument "event_id".

3.1 Case Sensitivity

Some languages (most notably C) treat upper and lower-case characters as representing different characters. Therefore, it is important that naming be consistent in case representation as well.

System services, RTL routines, item codes, message names, etc should have names which are completely capitalized.

\The C RTL will contain two definitions for each entry point: one with a lower case name, one with upper case.\

3.2 What Is A Facility?

A facility is a collection of code and/or data which operate together to perform a function or set of functions. For the purposes of the MICA naming scheme, each utility or layered product is typically considered to be a facility.

For MICA, most of the executive will be considered a single facility. \Need to work out details of function processor interface and naming\. Code that appears to provide executive functionality, but in reality resides elsewhere (Remote Procedure Call support, for instance) will be defined to be in separate facilities. Exceptions to this include support which is viewed to be part of the executive on VAX/VMS, such as the Send to Job Controller, Get Queue Information, and message lookup and formatting routines.

3.3 Facility Names

Good judgement must be used when defining facility names. In general, facility names should be the full name of the facility. For instance, the PILLAR compiler should use the facility name PILLAR, the linker should use LINK or LINKER.

Facility names must be carefully chosen so that messages issued from the various facilities can be easily identified without requiring extensive prior knowledge of the software or a need to feed facility names through an alphabet soup to English translation program.

There are many facilities which will be ported from VAX/VMS that have three-letter acronyms, such as the various components of the run-time library. It is acceptable that these facilities maintain their acronyms for maximum compatibility and to minimize confusion for both developers and users migrating from VAX/VMS to MICA.

If the facility name is eight characters or less, the facility name will be used as is. If the facility name exceeds eight characters (eg PERFORMANCE and COVERAGE ANALYZER), an acronym will be chosen which is sensible and easy to remember.

The facility name has no direct relation to its software distribution (bundled vs layered). Facilities which are bundled with the MICA operating system will have their own facility names. For instance, DEBUG will be the facility name for the debugger.

\NOTE: A mechanism must be put in place for the registration of facility names and facility code assignment.\

3.4 Module Names

Source module names are of the form

facility$modulename

The module name should directly correspond to the filename.

There are three types of source modules in PILLAR, implementation modules, definition modules, and combination modules.

3.4.1 Definition Modules

Definition modules contain only data and/or procedure definitions. Definition modules will have the string "_DEF" appended to the module name to indicate that it is a definition module.

3.4.2 Implementation Modules

Implementation modules contain only code. Implementation modules will be named as specified above.

3.4.3 Combination Modules

Combination modules are modules which contain both data definition and implementation components. Combination modules are named as specified above for implementation modules.

3.4.4 Definitions Internal To A Facility

Private data definitions (that is, definitions which are imported only by other modules in the defining facility) may be placed in definition modules or combination modules. Procedure definitions may be placed in the module containing the procedure implementation. Note that if such definitions are made public at some future time, the definitions must be moved to a definition module.

3.4.5 Public Definitions

Public data and procedure definitions (that is, definitions which are available for importation by user-level programs) must be placed in definition modules.

System services which have both a user-visible entry point and an

3.4 Module Names

Source module names are of the form

facility$modulename

The module name should directly correspond to the filename.

There are three types of source modules in PILLAR, implementation modules, definition modules, and combination modules.

3.4.1 Definition Modules

Definition modules contain only data and/or procedure definitions. Definition modules will have the string "_DEF" appended to the module name to indicate that it is a definition module.

3.4.2 Implementation Modules

Implementation modules contain only code. Implementation modules will be named as specified above.

3.4.3 Combination Modules

Combination modules are modules which contain both data definition and implementation components. Combination modules are named as specified above for implementation modules.

3.4.4 Definitions Internal To A Facility

Private data definitions (that is, definitions which are imported only by other modules in the defining facility) may be placed in definition modules or combination modules. Procedure definitions may be placed in the module containing the procedure implementation. Note that if such definitions are made public at some future time, the definitions must be moved to a definition module.

3.4.5 Public Definitions

Public data and procedure definitions (that is, definitions which are available for importation by user-level programs) must be placed in definition modules.

System services which have both a user-visible entry point and an

internal kernel-mode entry point must be defined in a definition module.

3.5 Procedures

Public procedures provided by Digital for MICA will be of the form

facility$entryname

In general, non-public procedures will not be visible, as the bulk of the system will be implemented in PILLAR, allowing the use of module-qualified symbols for inter-module communication. However, there will be some facilities coded in BLISS or other languages which do not support the concept of module-qualified symbols. In such languages, non-public procedures which must be declared global for inter-module communcation will be have names of the form

facility$$entryname

\Note that such names are available for use by customers in languages which do not support module-qualified symbols.\

\SIL does not support module-qualified symbols. However, a mechanism has been added to SIL to permit prefixing all exported names with a specified string. This is accomplished with the LINKAGE OPTIONS LOCAL PREFIX statement. When SIL programs are converted to PILLAR, this statement will be removed, and the symbols/references to these symbols will be through module-qualified symbols.\

3.5.1 System Routines

System routines are those routines which are

- Provided by Digital
- Run in user mode
- Require a documented, supported public interface
- Not officially part of the MICA RTL provided by ZK.
- Viewed by users as having "system" functionality.

Examples of system routines are "Send to Job Controller", "Formatted ASCII Output", "Put Message", "Get Message", and the "Image Autoloader".

The facility name for system routines is "EXEC".

3.5.2 System Services

System services run in kernel mode in the MICA executive. The facility name for MICA system services is "EXEC".

3.5.3 Kernel Routines

Kernel routines may only be called by the MICA operating system. Kernel routines are not visible to user programs. The facility name for kernel routines is "K".

\Since kernel routines will only be visible within the EXEC, they could be defined as module-local symbols, if the language in which the kernel is coded supports module-local symbols.\

3.6 Files

All files will be prefixed with "facility$" to facilitate identifying the facility to which a file belongs.

All files supplied with the MICA operating system which are facility-independent will be prefixed with "MICA$". This includes the operating system images, system support images such as the job controller, print symbionts, system accounting file, and utilities typically identified with the operating system rather than as their own facility. Facilities such as TPU, DEBUG, and the run-time library, although supplied with the operating system, are typically identified as their own facility, hence will use TPU$, DEBUG$, etc.

Long file names will use underscores ("_") to separate words within the file name. Consideration was given to using the hyphen as a separator, but was rejected because it would cause an inconsistency between file names and procedure names.

File types must be registered. The method for registering file types is TBD. Known registered file types are listed below.

FILE TYPE	CONTENTS
ANL	Output file for the ANALYZE command
BJL	BACKUP journal file
CLD	Command language description file
COM	Command procedure
DAT	Input or output data file
DIF	Output listing created by the DIFF command
DIR	Directory file
DIS	MAIL distribution list
DMP	Output listing from the DUMP command
EXE	VAX/VMS Image file
FDL	File definition language file

3.5.2 System Services

System services run in kernel mode in the MICA executive. The facility name for MICA system services is "EXEC".

3.5.3 Kernel Routines

Kernel routines may only be called by the MICA operating system. Kernel routines are not visible to user programs. The facility name for kernel routines is "K".

\Since kernel routines will only be visible within the EXEC, they could be defined as module-local symbols, if the language in which the kernel is coded supports module-local symbols.\

3.6 Files

All files will be prefixed with "facility$" to facilitate identifying the facility to which a file belongs.

All files supplied with the MICA operating system which are facility-independent will be prefixed with "MICA$". This includes the operating system images, system support images such as the job controller, print symbionts, system accounting file, and utilities typically identified with the operating system rather than as their own facility. Facilities such as TPU, DEBUG, and the run-time library, although supplied with the operating system, are typically identified as their own facility, hence will use TPU$, DEBUG$, etc.

Long file names will use underscores ("_") to separate words within the file name. Consideration was given to using the hyphen as a separator, but was rejected because it would cause an inconsistency between file names and procedure names.

File types must be registered. The method for registering file types is TBD. Known registered file types are listed below.

FILE TYPE	CONTENTS
ANL	Output file for the ANALYZE command
BJL	BACKUP journal file
CLD	Command language description file
COM	Command procedure
DAT	Input or output data file
DIF	Output listing created by the DIFF command
DIR	Directory file
DIS	MAIL distribution list
DMP	Output listing from the DUMP command
EXE	VAX/VMS Image file
FDL	File definition language file

HLB	Help library
HLP	Help text source file
IMAGE	MICA image file
INI	Initialization file
LIS	Listing file
LOG	Log file
MAI	Mail message file
MAP	Image memory allocation listing
MEM	Output file from RUNOFF
MSG	Message text source file
OBJ	VAX/VMS Object module
OBJECT	MICA object module
OLB	Object library
OPT	LINKER option file
RNO	RUNOFF input file
STB	Symbol table file
TJL	TPU journal file
TLB	Text library
TMP	Temporary file
TPU	TPU source file
TXT	Text file

\This list is incomplete; it does not contain any file types registered to layered products, or any new file types yet to be invented for MICA. Some of these file types will change to distinguish PRISM files from VAX/VMS files, such as object and image files.\

The three-character file type limit that was imposed on VAX/VMS file type naming does not exist on MICA or VAX/VMS (as of V4.0). When defining new file types, there is no need to be constrained to three characters.

3.7 Types

PILLAR type names will be of the form

facility$TYPE_name_of_type

facility is present to allow wildcard importation of names. The string "TYPE" provides differentiation of the facility-qualified TYPE name from other facility-qualified names. \Note: There is still some discussion regarding the use of the string "TYPE_" in these names. We will being using this naming convention, and will revisit once we have some experience using them. It is, after all, much easier to remove these strings than to add them as an afterthought.\

3.8 Data Structures

Data structure (record) names consist of two parts: the name of the record and the names of the fields within the record.

3.8.1 Fields

Fields within records will be named with an English name that is indicative of the contents of the field. Do not include any characters that indicate the size of the field. Field sizes are specified in the record definition.

The full name of a field is specified as

```
structure$fieldname
```

The structure name is the name of the structure (for example, in RMS, there will be FAB and RAB structures). This is one case where a qualified name is prefixed by something other than the facility name, and is done to enhance naming flexibility.

\Specifying the structure name is needed because some languages will have problems if more than one record has the same field name.\

3.9 Global Variables

Global variables are those data locations which are known to the linker as global symbols. In general, global variables are typically not provided in public program interfaces. However, in those cases where public global variables must be defined, the names are specified as

```
facility$G_variable_name
```

3.10 Constants

Numeric constants, enumerated type constants and literal values not part of a named set will have names of the form

```
facility$K_name
```

Due to internationalization requirements, string constants used for display purposes (either on the terminal or in a listing) must not exist within programs. String constants must be implemented via the message facility.

3.11 Messages

Message names will be of the form

facility$_status_name

The "status_name" string will be derived by using the first two or three words of the English message text.

Engineers must use good judgment to select message names, as these names will be used constantly by application programmers. Names must be chosen that are reasonable and easily remembered.

3.11.1 MICA Exec Status Codes

Status codes returned by the MICA executive will have the form

EXEC$_status_name

3.11.2 VMS Compatibility Status Code

Status codes returned by the VAX/VMS compatibility system services will have the form

SS$_status_name

The values for these status codes will be the same as on VAX/VMS.

3.12 Item Codes

Item code names (used in item lists) will be of the form

facility$K_name_of_item

Note that this creates a problem for system services which accept an item list as input. For these system services and system routines, the facility name may be specified as the service name or an acronym of the service name (for instance, JBC$K_itemname, SYI$K_itemname, etc).

3.13 Logical Names

Logical names have names of the form

facility$name

The name chosen for the facility should consist of one or more English words (should not present an internationalization problem), with underscores separating the words.

Although the MICA executive facility is EXEC, logical names defined by the operating system will use the facility SYS. This is done for compatibility and familiarity with VAX/VMS.

APPENDIX A

OUTSTANDING ISSUES

- Naming of TYPES.
- Function processor-related names.
- Compile-time facility procedure and macro names.

[End of file]

4–December–1987

Mica V1.0 Software Component Development Plan for Development Environment V1.0

Written and Issued by:

DMWalp—Development Environment Project Leader

Reviewed by:

Robert Bismuth—Development Environment Supervisor and Mica Product Project Leader

Jeff East—Cheyenne Product Project Leader

Benn Schreiber—Glacier Product Project Leader

TABLE OF CONTENTS

TABLES

Preface

This is a Mica Software component DEVELOPMENT/PROJECT PLAN. It is the description of the development project to design, build, test, evaluate, and deliver Development Environment V1.0. It is one of a set of plans for the Mica project and confirms to the overall Mica Project Team that all top-level project planning, evaluation, and management decisions have been made for this component.

Associated Documents

1. The Mica Project/Development Plan that describes the plan to deliver the total system, of which this software is a component.
2. The Cheyenne Project/Development Plan that describes the plan to deliver the total product, of which this software is a component.
3. The Glacier Project/Development Plan that describes the plan to deliver a total product, of which this software is a component.
4. The following Mica Working Design Document chapters that describes the goals, capabilities, and external characteristics of the facility that are delivered by this project:
    1. Object Module and Image File Format
    2. Linker
    3. System Dump Analyzer and System Debugger
    4. Condition, Exit and AST Handling
    5. Console Support

Change History

Date	Issue #	Description/Summary of Changes
20-Nov-1987	0.1	Initial Draft Complete
03-Dec-1987	0.2	Update to Reflect New Manpower Constraints

1 Development Strategy

The development environment project is responsible for providing software support for the Mica operating system group to build the Mica product. This project produces the scaffolding and development procedures from which the operating system is built. The other Mica subprojects are consumers of the development environment's deliverables.

Some of the development environment's deliverables are Glacier product components. Using these deliverables, customers build their own applications. The Quartz development effort is also a consumer of these components.

Many of the project's deliverables are required very early in the overall Mica schedule. This project's deliverables come in multiple phases. The first phases gets early baselevels of the deliverables to the Mica O/S group so that Mica can be developed. The later phases produce product versions of the deliverables.

2 Tasks and Estimates

The project is broken in a number of deliverables. Some deliverables are subdivided into phases and finally each phase is divided into the required tasks.

Some deliverables are broken down directly into tasks, by-passing the phase division. This is because of the relatively small number of days to complete the entire deliverable, and because a phase of the deliverable is not required at an early Mica baselevel.

The deliverables are defined by the project goals and requirements, and the phases of deliverables are defined by the overall Mica schedule.

Up to eight tasks make up each deliverable or deliverable phase:

1. Working Design Document (WDD)
   This task comprises the work required to complete a WDD chapter, including:

   Writing the overview
   Architect review of the overview
   General review of the overview
   Writing the chapter
   Primary review of the chapter
   Architect review of the chapter
   Formal review of the chapter
   Writing the final overview

2. Design
   This task involves completing the design beyond the WDD chapter, or doing the design alone if a WDD chapter is not required.

3. Coding
   The coding task involves coding the deliverable.

4. Debugging
   This task involves debugging the deliverable to such a level that basic test cases work.

5. Testing and Verification
   The testing and verification task involves a rigorous testing of the component. During this phase code reviews are done.

6. Integration
   During this phase, integration of the deliverable into the Mica system is completed.

7. Documentation
   The development and review of manuals is executed during the documentation task.

8. Maintenance
   This task involves maintaining the early deliverables that are used to build the overall system.

Not all tasks are required to complete each deliverable or deliverable phase.

Associated with each task is an estimated number of days to complete the task. These estimates are based on concentrated manpower days. The number of days is not elapsed time; for example, a task may take concentrated manpower only five days to complete, but may spread over several weeks because of outside review. Also listed with each deliverable or deliverable phase is the total number of days.

2.1 Object Language

The object language specifies the image file format, object modules format, and generic module format.

A design for the object language must be done. This includes doing the design work and describing the design in a WDD chapter.

A large amount of the design was already completed in earlier iterations of the Mica project.

The design must be approved by the Pillar Group and the Software Development Technologies (SDT) group at Spitbrook in addition to the Mica group.

08 - Total
08 - WDD
— - Design
— - Coding
— - Debugging
— - Testing and verification
— - Integration
— - Documentation
— - Maintenance

2.2 Linker

The linker produces Mica image files from Mica object modules. A linker is required for Mica development and is shipped as part of the Mica product.

All the development tasks are required to product the linker.

Development of the linker is done in four phases because of external dependencies in early baselevels.

2.2.1 Initial Phase

This initial phase of the linker is required by the Pillar development effort. It allows the Mica object language to get started.

During this phase, a WDD chapter is written, design work required for this phase is completed, and the linker is coded and debugged.

50 - Total
15 - WDD
05 - Design
20 - Coding
20 - Debugging
— - Testing and verification
— - Integration
— - Documentation
— - Maintenance

2.2.2 Object Module Diagnostic Support

This phase adds Object Module Diagnostic support to the linker. This support performs the functions that an object analyzer would perform.

This phase involves design, coding and debugging tasks.

05 - Total
— - WDD
01 - Design
02 - Coding
02 - Debugging
— - Testing and verification
— - Integration
— - Documentation
— - Maintenance

2.2.3 Librarian Support

During this phase, librarian support is added to the linker. This phase of the linker and the first release Pillar compiler is used by the general Mica development effort.

This phase involves design, coding and debugging tasks.

10 - Total
— - WDD
01 - Design
05 - Coding
04 - Debugging
— - Testing and verification
— - Integration
— - Documentation
— - Maintenance

2.2.4 Shareable Images Support

This phase of the linker facilitates the development of Mica. Support for shareable images, which are used extensively throughout the Mica system, is added during this phase.

Work is required in the design, coding and debugging tasks.

10 - Total
— - WDD
01 - Design
05 - Coding
04 - Debugging
— - Testing and verification
— - Integration
— - Documentation
— - Maintenance

2.2.5 Production Phase

This task comprises performance testing and performance enhancements. During this phase, the linker documentation is reviewed and open issues are resolved.

Testing, integration and documentation tasks make up this phase.

24 - Total
— - WDD
— - Design
— - Coding
— - Debugging
18 - Testing and verification
01 - Integration
05 - Documentation
— - Maintenance

2.3 Librarian

The librarian allows object modules to be grouped together and used as a single entity. Supported rudimentary help libraries are also included in the librarian.

The librarian is required for Mica development and is shipped as part of the Mica product.

There is NO support planned for macro libraries or text libraries.

All the development tasks are required to produce the librarian. Development of the librarian is done in three phases because of external dependencies on early baselevels.

OPEN ISSUE

What are help library support requirements?

2.3.1 Initial Phase

The initial phase is released with Pillar and the linker. All of the FRS functions are present with the exception of the shareable image support.

Since there is not a WDD chapter, a substantial design task is required. This phase also includes the coding and debugging tasks.

40 - Total
— - WDD
20 - Design
10 - Coding
10 - Debugging
— - Testing and verification
— - Integration
— - Documentation
— - Maintenance

2.3.2 Shareable Images Support

Shareable image support, in conjunction with the linker support, aids the development of Mica. Work is required in the design, coding and debugging tasks.

10 - Total
— - WDD
01 - Design
05 - Coding
04 - Debugging
— - Testing and verification
— - Integration
— - Documentation
— - Maintenance

2.3.3 Production Phase

The librarian documentation is reviewed and open issues are resolved. Testing, integration and documentation tasks make up this phase.

16 - Total
— - WDD
— - Design
— - Coding
— - Debugging
10 - Testing and verification
01 - Integration
05 - Documentation
— - Maintenance

2.4 Condition Handling Facility

The condition handling facility is at the heart of the Mica design, and must be completed early in the overall Mica Schedule. During its development, all tasks are involved in creating the facility. Development of the condition handling facility is done in three phases because of external dependencies for early baselevels.

2.4.1 Initial Phase

The first phase of the condition handling facility satisfies the requirements of the Base System subproject. Supplied is the ability to:

1. Raise software conditions.
2. Deliver a condition to a descriptor based handler.

   No reraise
   No vector handlers
3. Unwind to a specific target frame.
4. Unwind to the caller of a establisher.

During this phase, a WDD chapter is written, design work required for this phase is completed, and the facility is coded and debugged.

The WDD chapter is written by Robert Bismuth, the development environments supervisor.

20 - Total
— - WDD
05 - Design
08 - Coding

07 - Debugging
— - Testing and verification
— - Integration
— - Documentation
— - Maintenance

2.4.2 Complete Kernel Mode Phase

During this phase, the kernel mode functions not included in the initial phase are added to the facility, and problems with the initial phase are corrected.

The functions added are:

1. Vector handlers
2. Reraise
3. Stack handling
4. Nested unwinds

15 - Total
— - WDD
05 - Design
05 - Coding
05 - Debugging
— - Testing and verification
— - Integration
— - Documentation
— - Maintenance

2.4.3 Completion Phase

During this phase, the functions not included in the prior phases are added to the facility, and any problems with the prior phase are corrected. Also included is time for a code review.

The functions added are:

1. Exit handlers
2. User mode AST handlers
3. Exit unwind
4. Condition stack

15 - Total
— - WDD
05 - Design
05 - Coding
04 - Debugging
01 - Testing and verification
— - Integration
— - Documentation
— - Maintenance

2.4.4 Production Phase

NOT SCHEDULED!

Not required for FRS

In this phase, the condition handling facility documentation is reviewed and open issues are resolved. Testing, integration and documentation tasks make up this phase.

16 - Total
— - WDD
— - Design
— - Coding
— - Debugging
10 - Testing and verification
01 - Integration
05 - Documentation
— - Maintenance

2.5 P* Mica Object Language Fixes

Psimulator, Pdebug, Pconsole need to be updated to support the new image and object module format. This work is done at the same time the Pillar compiler and the linker are released.

This phase includes the design, coding and debugging tasks.

NOTE

Switching Mica's development to use Pillar and new object language is a major crossover point because:

- **Old and new object language formats cannot co-exist.**
- **The instruction layout changes.**
- **The XMI system is the only hardware at this time that supports the new instruction format.**
- **Old and new instruction formats cannot co-exist.**

OPEN ISSUE

Has the PRISM instruction format changed? If so, another version of Psimulator needs to be created and the emulators must be updated.

15 - Total
— - WDD
05 - Design
05 - Coding
05 - Debugging
— - Testing and verification
— - Integration
— - Documentation
— - Maintenance

2.6 Emulator and Simulator Maintenance

While the emulator and simulator are in use, maintenance needs to be done. Two days per month should be allocated for correcting problems reported with the emulator and simulator.

30 - Total
— - WDD
— - Design
— - Coding
— - Debugging
— - Testing and verification
— - Integration
— - Documentation
30 - Maintenance

2.7 System Debugger

The system debugger is a interactive debugging tool used to monitor the execution of Mica, and is required for development of the Mica system.

2.7.1 WDD Phase

In this phase, the design of the system debugger is written and reviewed. The System Dump Analyzer WDD work is also included in this effort, since SDA and the system debugger are in one WDD chapter.

15 - Total
15 - WDD
— - Design
— - Coding
— - Debugging
— - Testing and verification
— - Integration
— - Documentation
— - Maintenance

2.7.2 Completion Phase

When the Mica development effort is moved from the simulator and emulator hardware to the XMI-based system, Pdebug no longer works, and a new debugger is required.

This phase includes the design, coding, and debugging tasks.

This phase does not include support for a debugger symbol table.

25 - Total
— - WDD
10 - Design
10 - Coding
05 - Debugging
— - Testing and verification
— - Integration
— - Documentation
— - Maintenance

2.7.3 Production Phase

NOT SCHEDULED!

There are NO requirements to have the system debugger shipped at FRS.

In this phase, the documentation is reviewed and open issues are resolved. Testing, integration and documentation tasks make up this phase.

10 - Total
— - WDD
— - Design
— - Coding
— - Debugging
03 - Testing and verification
02 - Integration
05 - Documentation
— - Maintenance

2.8 Crash Dump Facility

This component writes system crash dump information to a file. This feature is required for Mica development and is also used to report Mica failure information to support organizations.

This deliverable is not broken into phases and contains all the required tasks.

15 - Total
— - WDD
04 - Design
05 - Coding
03 - Debugging
02 - Testing and verification
01 - Integration
— - Documentation
— - Maintenance

2.9 System Dump Analyzer

The System Dump Analyzer is a utility that is used to help determine the cause of system failures. This feature is required for Mica development and is also used to determine Mica failures from the field.

2.9.1 Completion Phase

Mica development requires a way to analyze system failures after the failure has occurred. This phase includes the design, coding and debugging tasks.

This phase does not include support to probe a running system.

27 - Total
— - WDD
10 - Design
10 - Coding
07 - Debugging
— - Testing and verification
— - Integration
— - Documentation

— - Maintenance

2.9.2 Production Phase

During this phase, the documentation is reviewed and open issues are resolved. Testing, integration and documentation tasks make up this phase.

NOT SCHEDULED!

There are NO requirements to have the system dump analyzer shipped at FRS.

13 - Total
— - WDD
— - Design
— - Coding
— - Debugging
05 - Testing and verification
03 - Integration
05 - Documentation
— - Maintenance

2.10 Source Control Mechanism

The source control mechanism comprises the procedures and policies with which the Mica development group manages the Mica sources. This mechanism is required for Mica development.

2.10.1 Initial Phase

The source control mechanism policy must be designed and documented very early in the development schedule of Mica before much code is written. This phase includes the design and documentation tasks.

10 - Total
— - WDD
07 - Design
— - Coding
— - Debugging
— - Testing and verification
— - Integration
03 - Documentation
— - Maintenance

2.10.2 Completion Phase

In this phase, the "master pack" must be created and additional source control procedures implemented.

05 - Total
— - WDD
— - Design
05 - Coding
— - Debugging
— - Testing and verification
— - Integration
— - Documentation
— - Maintenance

2.11 Mica Build Procedure

For the development of the Mica system, a set of procedures are required to build the system in a reproducible, controlled, and orderly manner.

2.11.1 Initial Phase

A design for Mica build procedures is done.

05 - Total
— - WDD
05 - Design
— - Coding
— - Debugging
— - Testing and verification
— - Integration
— - Documentation
— - Maintenance

2.11.2 Completion Phase

During this phase, the procedures to build the Mica system are created and documented. The phase includes the coding, debugging, and documentation tasks.

15 - Total
— - WDD
— - Design
05 - Coding
05 - Debugging
— - Testing and verification
— - Integration
05 - Documentation
— - Maintenance

2.11.3 Maintenance Phase

Maintenance needs to be done from the time at which the Mica build procedures are put into place. Two days per month should be allocated for correcting problems reported with the build procedures.

33 - Total
— - WDD
— - Design
— - Coding
— - Debugging
— - Testing and verification
— - Integration
— - Documentation
33 - Maintenance

2.12 Off-line System

OPEN ISSUE

The design has not been done and looks to be a packaging issue only.

05 - Total
— - WDD
01 - Design
01 - Coding
01 - Debugging
01 - Testing and verification
01 - Integration
— - Documentation
— - Maintenance

2.13 Console Terminal

The console terminal support allows read and write access to the console terminal. This support is required for the Mica FRS product, and is used by tools needed for Mica development.

OPEN ISSUE

What functions are the minimum required?

- **Xon, xoff ?**
- **Echo, no echo ?**
- **TBD?**

2.13.1 WDD Phase

In this phase, the WDD chapter for the console terminal is written and reviewed. Also included is the WDD work for the other console support.

15 - Total
15 - WDD
— - Design
— - Coding
— - Debugging
— - Testing and verification
— - Integration
— - Documentation
— - Maintenance

2.13.2 Completion Phase

During this phase, all the FRS console terminal support is completed. This phase includes the design, coding, debugging, testing, and integration tasks.

15 - Total
— - WDD
02 - Design
05 - Coding
04 - Debugging
03 - Testing and verification
01 - Integration
— - Documentation
— - Maintenance

2.13.3 Console Terminal as Client Phase

During the client phase, password protection is added to the console and the Mica-native system management user interface is invoked. This phase includes the design, coding, debugging, testing, and integration tasks.

05 - Total
— - WDD
01 - Design
02 - Coding
02 - Debugging
— - Testing and verification
— - Integration
— - Documentation
— - Maintenance

2.14 Console Support For Configuration Control and Diagnostics

This work allows Mica components to access Moraine's console configuration control and diagnostic functions. These functions are Moraine specific, and allow the bootstrap device to be set and modules to be enabled and disabled for examples. The functions are required by system management, system installation, and the configuration manager software.

This phase includes the design, coding, debugging, testing and integration tasks.

15 - Total
— - WDD
04 - Design
04 - Coding
04 - Debugging
02 - Testing and verification
01 - Integration
— - Documentation
— - Maintenance

2.15 Other Console Support

OPEN ISSUE

What is to be supported is still undecided, but may include:

Console storage device
Support of the console's ESP as a client

30 - Total
— - WDD
?? - Design
?? - Coding
?? - Debugging
?? - Testing and verification
?? - Integration
— - Documentation
— - Maintenance

2.16 Image Converter

The Object Module or Image converter takes in VMS format images and produces Mica format images. This does not mean the image's contents, instructions, and data are changed; only the format of the image is changed.

The Mica and Pillar development groups need a way to test the Mica images format before Mica object modules are supported.

This deliverable includes work in the design, coding and debugging tasks.

15 - Total
— - WDD
04 - Design
07 - Coding
04 - Debugging
— - Testing and verification
— - Integration
— - Documentation
— - Maintenance

2.17 P* Fixes for New DST

NOT SCHEDULED!

P* Fixes for New DST are not currently scheduled, because by the time that they could be completed they would no longer be useful.

Psimulator, Pdebug, and Pconsole need to be updated to support the new debugger symbol table (DST) format. This is done at the same time the Pillar compiler and the linker are released.

This deliverable includes work in the design, coding and debugging tasks.

15 - Total
— - WDD
03 - Design
07 - Coding

05 - Debugging
— - Testing and verification
— - Integration
— - Documentation
— - Maintenance

2.18 Object Module and Image Analyzer

NOT SCHEDULED!

Required functions for FRS are fulfilled by object diagnostics support in the linker.

The Object Module and Image Analyzer checks the complex Mica object modules for consistency.

An Object Module and Image Analyzer is required by the Pillar development group, Mica development group, and SDT, and is shipped as part of the Mica product.

Development of the Object Module and Image Analyzer is done in two phases because of external dependencies for early baselevels.

2.18.1 Initial Phase

The initial release contains all the FRS product functions.

Since there is not a WDD chapter, a substantial design task is required. This phase also includes the coding and debugging tasks.

45 - Total
— - WDD
20 - Design
15 - Coding
10 - Debugging
— - Testing and verification
— - Integration
— - Documentation
— - Maintenance

2.18.2 Production Phase

During this phase, the Object Module and Image Analyzer documentation is reviewed and open issues are resolved.

Testing, integration and documentation tasks make up this phase.

16 - Total
— - WDD
— - Design
— - Coding
— - Debugging
10 - Testing and verification
01 - Integration
05 - Documentation
— - Maintenance

2.19 XMI System Support

Currently no support such as that for the emulators and simulators is required for the XMI systems.

3 Resources

3.1 Staffing

The project is estimated to take 38 man-months of software engineering time. The figure is based on the factor that it take five real days to get four concentrated manpower days.

Kim Peterson and David Walp are current staff, and are sufficient manpower to finish the project by Q1 1990.

Kim's deliverables are:

Object Module Language
Linker
Librarian
P* Fixes
Emulator and Simulator Maintenance
Image Converter

Kim is scheduled to work on the files subproject during the two baselevels.

David is the project leader and his deliverables are:

Condition Handling Facility
System Debugger
Crash Dump Facility
System Dump Analyzer
Source Control Mechanism
Mica Build Procedures
Console Support
Off-line System

David is scheduled to work on the files subproject during the one baselevel.

3.2 Hardware and Software

This subproject development requires access to a standalone emulator from the start to Baselevel 4 (BL4) for about 50% of the time. The project has responsibility for the maintenance of the emulator software. Work on the condition handling facility, console terminal support, and the system debugger also require access to the emulator.

The development of the crash dump facility requires standalone time on a Pebble system for about 30% of the time from Baselevel 5 (BL5) to Baselevel 6 (BL6).

The development of the console support and off-line system requires Moraine standalone time from Baselevel 6 to Baselevel 7 (BL7) for about 30% of the time.

Table 1: Hardware Resources Summary

Date	Resource	
1988		
JAN (BL1)	1 Emulator (50%)	
FEB	|	
MAR	|	
APR (BL2)	|	
MAY	|	
JUN	|	
JUL (BL3)	|	
AUG	|	
SEP	|	
OCT (BL4)	+	
NOV		
DEC (BL5)		
1989		
JAN	1 Pebble (30%)	
FEB	|	
MAR (BL6)	+	
APR	1 Moraine (30%)	1 Moraine Time Sharing System (50%)
MAY	|	|
JUN (BL7)	+	+

3.3 Tools

None.

Note that this subproject supplies tools used by other subprojects to develop the Mica system.

3.4 Travel

Currently, this subproject has no travel requirements.

This subproject requires review and approval from the organization outside of this facility for the object language and condition handling facility. So far, the use of interoffice mail and electronic mail has been sufficient; however, that is subject to change.

4 Procedures

4.1 Schedules/Progress

Schedule and milestone tracking is done at two different levels, milestone level and monthly progress level. The milestones listed in Section 7 are tracked via Mica baselevel tracking.

The milestones are spaced around three months apart. The developers break the milestone work into monthly work items. Once a month a status report is written, listing the completion status of the previous month's work items and the planned work items for the upcoming month.

Also, a summarized status report is given at the monthly Mica project leaders meeting. Again, this status report includes the completion status of the previous month's work items and the planned work items for the upcoming month.

The project data is entered into a VAX Software Project Manager database, which will be updated as the project proceeds.

4.2 Design

This subproject is following the Mica design process procedures.

OPEN ISSUE

What are the Mica design process procedures?

All members of this group, Kim Peterson, Robert Bismuth and David Walp are considered primary reviewers for subproject deliverables. The Mica architecture group reviews all WDD designs. Additional Mica primary reviewers for each area are listed below; however, this is subject to change as people's responsibilities change and as the Mica design changes.

- Object Module Language

  David Ballenger
  Lou Perazzoli
  Benn Schreiber

- Linker

  David Ballenger
  Chuck Lenzmeier
  Mark Lucovsky
  Lou Perazzoli
  Benn Schreiber

- Librarian

  David Ballenger
  Chuck Lenzmeier
  Mark Lucovsky
  Benn Schreiber

- System Debugger and System Dump Analyzer

  Base System's Group Person
  Mark Ditto

- Source Control Mechanism and Mica Build Procedures

  Kris Barker
  Mark Lucovsky
  Benn Schreiber

- Off-line System

  Richard Brown
  Peter Brundrett
  Mark Ditto
  Debbie Girdler
  Mark Lucovsky

- Console Support

  Marilyn Fries
  I/O Group Person

Base System's Group Person

4.3 Code

It is the engineer's responsibility to create structured, efficient, and maintainable code and reliable debugger code. This code follows the naming standard and coding conventions for the Mica development group.

The FRS product deliverables are subject to the Mica coding review procedures.

OPEN ISSUE

What are the Mica coding review procedures?

4.4 Associated Documents

The following WDD chapters adhere to the Mica WDD CMS source and ECO control procedures.

Object Module and Image File Format
Linker
System Dump Analyzer and System Debugger
Console Support

4.5 Publications

Manuals for the Linker and Librarian components are required for FRS of the Glacier product.

Some type of DEC internal manuals are required for Condition Handling Facility, System Debugger and System Dump Analyzer.

5 Testing/Test Results

This subproject follows the Mica development testing procedures.

OPEN

What are the Mica development testing procedures?

Additional testing includes performance testing of the linker.

It should also be noted that a lot of testing will be done by Mica development, since some of the deliverables are tools. Testing of these components should place emphasis on features not used by the Mica development group.

6 Milestone Definitions

This subproject is made of a number of relatively small components, instead of a single large deliverable broken into baselevels. The subproject produces many deliverables that are tools used to build Mica, and these tools are required by Mica in certain baselevels. Because of these facts, the Development Environment milestones are the same as those for the Mica project baselevels.

7 Milestone Summary

Milestone	Deliverable
BL1	Object Language
Jan-88	Initial Linker
	Initial Condition Handling
	Initial Source Control Mechanism
	Initial Mica Build Procedures
BL2	Image Converter
Apr-88	P* Mica Object Language Fixes
	Linker with Object Module Diags
	Kernel Mode Condition Handling
	Console Support WDD Chapter
	System Debugger and SDA WDD Chapter
BL3	Initial Librarian
Jul-88	Linker with Librarian Support
	User Mode Condition Handling
	Complete Source Control Mechanism
	Complete Mica Build Procedures
	Console Terminal Support
BL4	Linker with Shareable Images Support
Oct-88	Librarian with Shareable Images Support
	System Debugger
BL5	(Kim works with Files Subproject)
Dec-88	(Dave works with Files Subproject)
BL6	(Kim works with Files Subproject)
Mar-89	System Dump Analyzer
BL7	Product Linker
Jun-89	Product Librarian
	Remaining Console Support
	Offline System Packaging

8 Dependencies, Risks, and Contingencies

8.1 Dependencies of this Project on Other Projects/Events

1. Object Language Review
   The object language must approved by Mica, Pillar and SDT groups before 1-Jan-1988.

2. XMI Hardware Schedule
   If there is a slip in the XMI system availability date, it affects this subproject's schedule, since continued support of the emulator and simulator is required, and because more features may need support in the simulator and emulator.

3. Pillar Schedule
   If there is a slip in Pillar V1 date, the transition to new object modules and XMI system is affected.

4. SIL V2
   SIL V2 work is required for use and testing of the condition handling facility. If SIL V2 schedule slips or does not include condition handling support, it impacts the scheduling of this subproject, the base systems group, and the entire Mica schedule.

5. Base Systems Group Work
   The condition handling work requires some rudimentary environment support from the base system group. If the environment is not in place when needed, it impacts the scheduling of this subproject, the base systems group, and the entire Mica schedule.

6. Intra-Subproject Work
   The deliverables in this subproject are ordered in such a way that early deliverables are used as tools to build later deliverables. The order of the deliverables cannot be changed at random.

7. Console Design
   The amount and type of work to be done for console support depends on the console design, which is still being defined.

8. System Management Design
   The off-line system design, and the amount and type of work, depend on the system management design.

8.2 Dependencies of Other Project/Events on this Project

1. Scheduled Baselevel Deliverables Through Baselevel 4
   A number of Mica subprojects, and the Mica overall schedule, depend heavily on this subproject's deliverables through Baselevel 4.

2. Object Module And Image Deliverables For Pillar
   The Pillar group depends on the subproject to produce the object language, linker, librarian, image converter, and emulator and simulator support. The SDT organization also depends on this work, but they will not require the deliverables as early as the Pillar group.

3. Console Support for Configuration Control and Diagnostics
   The configuration manager software depends on the console support for configuration control and diagnostics.

8.3 Risks

The dependencies listed in section 8.1 are all risks to this subproject.

This subproject's schedule through BL4 is very aggressive. There is no room for error in the schedule. There are no allowances for design rework and bad time estimates. Any change in personnel affects the schedule. No additional functions can be added.

The actual work itself is not technically risky. Digital has produced previous products which support most of the required functions of this project's deliverables.

9 Open Issues

1. What are help library support requirements?
2. Has the PRISM instruction format changed?
3. The design of the off-line system is not completed.
4. What console terminal functions are the minimum required?
5. What is in other console support?
6. What are the Mica design process procedures?
7. What are the Mica coding review procedures?
8. What manuals are required for FRS?
9. What are the Mica development testing procedures?

Digital Equipment Corporation - Confidential and Proprietary
For Internal Use Only

Mica Working Design Document
Object Architecture

Revision 2.1

2-December-1987

Issued by:
Lou Perazzoli

digital™

TABLE OF CONTENTS

FIGURES

TABLES

Revision History

Date	Revision Number	Summary of Changes
18-Feb-1986	0.1	Original (Jim Kelly)
27-Feb-1986	0.1	Sent out for initial review (Jim Kelly)
28-Feb-1986	0.2	Add more detail, incorporate review changes. Primary changes include changes in terminology, making process level root object containers visible to descendants, allowing an object container to be transferred, adding in an "identifer" level, adding object driver description.(Jim Kelly)
21-Mar-1986	0.2	Sent out for second review.
19-May-1986	0.3	Reformatted to adhere to WDD structure. Incorporated changes from second review. Changes included changes in terminology, adding an additional process root container, changing the way transferability of an object container is determined, adding temporary object support, and various other minor changes.(Jim Kelly)
06-Jun-1986	0.3	Sent out for third review (Jim Kelly)
17-Jun-1986	0.4	Incorporated changed from third review. Delivered for incorporation into the WDD.
10-Jul-1986	1.0	Incorpated changes from WDD consistency review. The most significant of these changes included removal of the identifier level, definition of the name table structure and capabilities, removal of bonded objects apabilities, addition of alias objects,addition of temporary objects. (Jim Kelly)
02-Dec-1986	1.1	Attempted to simplify and unify. Major changes include the removal of acronyms, simplification of lock strategy, unification of name strategy, addition of logical names, addition of object allocation. (Lou Perazzoli)
28-Jan-1987	1.2	Removal of rooted containers and addition of directory containers. Added object services (e$ routines). (Lou Perazzoli)
20-Mar-1987	1.3	Remove alias, etc, add support for UNIX fork and exec. Added algorithums for object services. (Lou Perazzoli)
30-May-1987	1.4	Minor changes(Lou Perazzoli)
15-Sep-1987	2.0	Removal of fork and exec support for UNIX, minor fix-ups, added SIL definitions for object structures and quota section. (Lou Perazzoli)
24-Nov-1987	2.1	Minor changes. Changed E$ routines to OBJ$ routines and changed names to match the coding standard. (Lou Perazzoli)

CHAPTER 1

OBJECT ARCHITECTURE

1.1 Overview

1.1.1 Introduction

This chapter describes the software architecture of objects. It describes what objects are, and defines the data structures and operations necessary to support objects.

1.1.2 What Is an Object?

Objects are abstract elements provided by an operating system that may be accessed by a user or a program. Typically, objects are defined in terms of the operations that may be performed upon them (for example, create, clear, set, get information, wait, delete) and their relationships to other objects. The reason for categorizing these elements as objects is to provide a single, standardized set of rules for creating, naming, protecting, accessing, and managing them. For example, each object has a unique ID value (called an object ID) which may be used to identify it. Objects at the job and process levels are only directly expressible by threads in that job or process.

1.1.3 Scope

It is important to understand that this chapter only defines the architecture of objects, not all object types. It is necessary that some objects or parts of objects be defined as part of this architecture.

1.1.4 Requirements and Goals

- Software development goals
  - — Provide an extensible, yet rigorous framework for the definition and manipulation of executive-controlled data structures.
  - — Maintain management consistency. The management of objects, in terms of actions taken to fulfill service requests, should be as object-type independent as possible. For example, standard routines and procedures can be established for determining whether access to an object should be granted.
  - — Provide new object definition support. It should be possible to add new object types to the system without having to modify existing system code. This means that the interface between the kernel/executive system software and objects must be well-defined, and that the kernel/executive need not have knowledge of the internals of all objects.
- Interface goals
  - — Provide consistent specification. The ways in which each object in the system may be specified by users should be minimized and kept consistent with the manner in which other objects are referenced.

- Provide consistent operations. There are some operations that apply to a set of objects within the system, such as wait. The definition of what these operations mean to each object should be kept simple and similar to their definition for other objects.
- Support level independence. Where possible, the operations that may be performed on an object type, and the behavior of that object, should not be dependent upon the level (system, job, or process) at which that object has been created. This allows applications to be developed in the relative safety of process and job levels before being moved to a more shareable level, with minimal change in behavior.
- Provide security and protection. The method of determining which objects a user may refer to, and which operations may be performed on those objects, should be the same for all objects. This is the basis for Mica security.

1.1.5 Functional Description

The object architecture runs in kernel mode at IPL 0. Through the use of mutexes, object architecture procedures can simultaneously execute on multiple processors.

The object architecture provides a framework for creating object-specific services. These services include creating, deleting, allocating, referencing, name translating, and getting information about objects. For example, the object service to create an event, and the object service to create a thread both invoke the same object architecture-defined routine to create the object.

The object architecture provides a hierarchical visibility structure for objects. When an object is created, it is placed at one of three levels: system, job, or process. Objects at the system level are visible to all threads on the system. Objects at the job level for a particular job are only visible to threads in that job. Objects at the process level for a particular process are only visible to threads in that process. For example, a thread cannot access an object that is at the process level for another process, because it cannot express an object ID for that object.

Each level can contain one or more object containers to catalog objects at that level. There are two types of object containers at the process level: Process-private object containers and display object containers. Objects stored in a process-private object container are only visible to the process with which the container is associated. Objects stored in a display object container are visible to the associated process, and any of its descendant processes.

Objects are referred to by object ID. If a program refers to an object name, this name must be translated to an object ID. An object name is unique within a container for each object type at each processor mode. When a user attempts to refer to an object using an object ID, the user's access rights are compared to the access rights associated with an object. If there is a match, the user may access the object.

An object may be allocated to a user, resource ID, job, process, or thread. This allows objects to be shared among restrictive classes of users.

An object ID is deleted when the object container holding the corresponding object is deleted, or when the object ID is explicitly deleted. The object itself, however, is not deleted until the ID is deleted, and there are no outstanding references to the object.

Table 1–1 summarizes key object architecture terms and components.

Table 1–1: Object Architecture: Terms and Definitions

Object Identification and Names		
Term	Type	Definition
Object ID	64-bit Value	Used to refer to an object.

Table 1–1 (Cont.): Object Architecture: Terms and Definitions

Object Identification and Names

Term	Type	Definition
Principal Object ID	Object ID	Associated with an object at object creation. An object has exactly one principal object ID.
Reference Object ID	Object ID	Optionally associated with an object. An object may have zero, one, or more reference IDs.
Object Name	Character String	Together with type and mode, an object name can be translated to an object ID. The combination of object type, mode, and name string is unique within a single object container.
Object Name Table	Data Structure	Tracks both logical names and object names within an object container. When an object container is created, a name table is also allocated, and that address is stored in the object container's body.

Object Hierarchy

Term	Definition	Description
Object Container	Object Type	Objects of this type contain pointers to other objects. They are used to organize large numbers of objects.
Object Level	–	Indicates the scope of visibility of an object container.
System Level	Object Level	Objects at this level are potentially accessible to all processes on the system.
Job Level	Object Level	Objects at this level are potentially accessible to all processes in a given job.
Process Level	Object Level	Objects at this level are potentially accessible to all threads in a given process. Containers at this level can be ether display or private.
Display Object Container	Object Container	Objects in such containers are accessible to a given process and all of its descendants.
Private Object Container	Object Container	Objects in such containers are accessible only to a given process, and not to its descendants.
Container Directory	Data Structure	Used to organize large numbers of object containers. All threads have the same system container directory. All threads in a job have the same job container directory. All threads in a process have the same process container directory.
Object Header	Data Structure	Fixed-format data structure that contains object type-independent data. This header is used by the executive without necessarily knowing the type of object it is accessing.
Object Body	Data Structure	A data structure that is specific to an object type.

Table 1–1 (Cont.): Object Architecture: Terms and Definitions

Object Type		
Term	Definition	Description
Object Type	–	Object type determines what operations can be performed on an object.
Object Type Descriptor (OTD)	Data Structure	Describes what operations are supported for what object types. There is one OTD for each object type.

Miscellaneous		
Term	Definition	Description
Object Service Routines	System Routines	Implement operations that can be performed on objects. Some object service routines are particular to a certain type of object; others are supported across all object types.
Object Allocation Block	Data Structure	Contains information about object allocation.

1.1.6 Object-Related Operations

The following types of operations can be performed on most types of objects:

- Creating an object
- Protecting an object
- Translating an object ID
- Deleting an object
- Creating references to an object
- Making a temporary object
- Marking a new object as temporary
- Allocating an object
- Deallocating an object
- Getting information about an object
- Changing the name of an object

1.1.7 Summary of Proposed Changes to Object Architecture

We propose that the following changes be made to the existing object architecture chapter:

- Removal of all ULTRIX dependencies. These include clone procedures and executive actions.
- Addition of Pillar definition records.
- Deduction of quota rules.

1.2 Functional Interface and Description

An *object* is a set of data structures representing an abstraction. Processes and events are examples of objects. In the case of an event object, the object represents the abstraction of an event that either has or has not taken place.

An object consists of a standard data structure called the *object header*, described in Section 1.4, and an object type-dependent data structure called the *object body*.

1.2.1 When Should an Element in the Executive be an Object?

An element should be an object when it:

- Needs a name for sharing.
- Needs to be referenced by user-mode software, and there are multiple instances of these elements.
- Needs to be protected via ACLs. Exceptions to this include structures that are exported by an RPC, such as job controller queues.
- Can be allocated.

No object can be dependent on another object such that there is an order dependence during object ID deletion. This allows various rundown operations to delete object IDs in any order, yet the rundown proceeds efficiently.

1.2.2 Object IDs

When an object is created, it is assigned a 64-bit value called its *object ID*. This value provides the fastest means for a user-mode routine to identify and access the object.

When an object is created, the ID returned is called the *principal ID*. Every object has only one principal ID. An object may also have one or more reference IDs which also point to the object. Reference IDs are discussed in Section 1.7.9.

The principal ID of an object that is accessible by two processes is the same in both processes. This extends to objects at all levels (including those in ancestors' process-display object containers, described in Section 1.5.9). This property allows IDs of shared objects to be communicated between cooperating threads.

1.2.3 Object ID Format

Object IDs are used to uniquely identify objects within the system. Object IDs are not addresses of objects, but values that may be used to generate the addresses of objects.

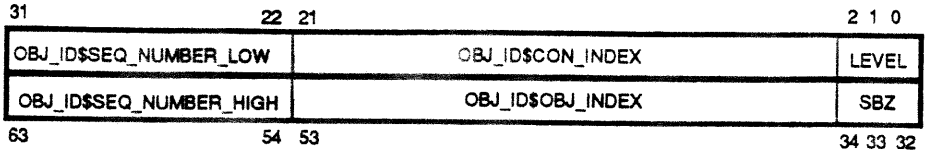
Object IDs are comprised of the following fields:

- Sequence number (*obj_id$seq_number_low*, *obj_id$seq_number_high*)
- Level (*obj_id$level*)
- Object container index (*obj_id$obj_index*)
- Container directory index (*obj_id$con_index*)

The format of an object ID is shown in Figure 1–1.

Object ID value zero (0) is reserved for use as an invalid value.

Figure 1–1: Object ID Format



1.2.4 Object ID Sequence Numbers

Three object ID fields are used to locate the address of the object.They are the level field, the object container index field, and the container directory index field. In addition, the sequence number high and low are used to help catch programming errors. Such an error might occur if a program used an object ID after the corresponding object had been deleted, and another object had been created with the same ID as the previously deleted object.

For example, consider a program with the following error. First, the program creates a FIZBOT object (which receives a corresponding object ID). Sometime later, the program deletes the FIZBOT object and creates a WALZOL object. Suppose that, due to a programming error, an operation on the FIZBOT object is attempted (such as a wait operation). If the WALZOL object had coincidentally received the same object ID that had been used for the FIZBOT object, a wait on the wrong object might go undetected.

To catch this type of error, each time an object ID is created, the sequence number fields (two 10 bit fields) are different each time an object ID is assigned to the same "slot".

When an object ID is created, the old sequence high field, which was saved, is incremented and reset to zero in the overflow case. In our example above, the improper use of the FIZBOT object ID would be detected and flagged as an error (no such object). Notice that the sequence high field cycles every 1024 allocations of an object ID.

The sequence low field receives a random value.

The use of sequence numbers does not totally eliminate the problem; it just reduces it to an extremely small probability for most programming situations.

1.2.5 Object Containers

Since programs are likely to have many objects at any point in time, it is desirable to be able to group objects together in some logical manner. For this purpose, a type of object called an *object container* is defined. When creating an object, it is necessary to specify the ID of the object container in which a pointer to the newly created object is to be stored.

From the executive's point of view, object containers are used when translating an object ID into a pointer to the object.

1.2.6 Object Levels

There are three levels of object visibility: process, job and system. The principal object ID level field determines the visibility of an object. The software process control block (SWPCB) contains level directors for the object levels. For each level, there is a pointer to the container directory corresponding to that level.

1.2.6.1 System Level

The purpose of the system level is to provide a place to keep objects shared by multiple jobs. Implementation of the group concept supported by VMS uses this level of sharing, by creating an object container at the system level that is only accessible to threads with the proper group identifier.

Objects at this level are potentially accessible by all processes in the system. If access to an object at this level is to be restricted, then access protection must be explicitly assigned.

1.2.6.2 System Level Director Mutex

A system-wide mutex exists for synchronizing access to the system level containers. A pointer to this mutex is located in each software process control block. This mutex is acquired for object ID translations at the system level and for other system-level operations, such as Delete Object ID.

1.2.6.3 Job Level

The purpose of the job level is to allow objects to be shared by all processes within a single job. If access to an object at this level is to be restricted so that only some processes within a job may access it, then access protection must be explicitly assigned.

1.2.6.4 Job Level Director Mutex

A job-wide mutex exists for synchronizing access to the job level containers. A pointer to this mutex is located in each process control block. This mutex is acquired for object ID translations at the job level, and for other job level operations.

1.2.6.5 Process Level

The process level has two types of containers, display and private.

The purpose of process-private object containers is to provide a location for objects that are not accessible to any other process. No access protection needs to be assigned to objects in this type of object container. This simplifies the programming effort, and also provides the fastest possible object access.

The purpose of process-display object containers is to provide a location for objects that are accessible to the associated process, and to all descendant processes of that process. Objects in this object container do not typically require any access protection. Note, however, that if access protection is assigned to objects at the process level, it is checked at the time of each access.

It is important to note that the process-display object containers are only accessible to a process and its descendant processes.

1.2.6.6 Process Level Director Mutex

A job-wide mutex exists for synchronizing access to the process level containers. A pointer to this mutex is located in each process control block. This mutex is acquired for object ID translations at the process level, and for other process-level operations.

This mutex must be job-wide to allow proper synchronization on display containers which are shared among a process and its descendants.

1.2.7 Container Directory Index Field of the Object ID

The container directory index field is used as an index into the object array of the container directory. This yields a pointer to the object container, which contains a pointer to the object represented by the object ID.

1.2.8 Container Directory

For each level there is a corresponding container directory. All threads share the same system container directory. All threads within the same job share the same job container directory. All threads within a process shares the same process container directory.

The total number of container directories within the system at any given time is equal to the number of jobs plus the number of processes plus one (for the system level container directory).

The container directory provides a structure containing pointers to all the object containers within that level. It also provides a method of naming object containers.

1.2.8.1 Object Index Field of the Object ID

The object index field is used as an index into the object array of the object container located by the directory index. This yields a pointer to the object header of the object represented by the object ID.

1.2.9 Expressibility of Object IDs

One characteristic of the architecture is that threads are not able to directly express the ID of all objects. Here, the term "directly express" means to generate an ID value that corresponds to an object.

All threads are able to directly express the ID of all objects at the system level. However, objects at the job and process levels are only directly expressible by threads in that job or process. For example, there is no object ID value a thread in job X can use to directly refer to an object at the job or process level in job Y. Also, a thread in process Q cannot directly refer to process level objects in any other process, unless the object is in a process-display object container of an ancestor process (process-display object containers are discussed in Section 1.2.6.5).

1.2.10 Shareability of Object IDs

If two threads can both express the principal ID of an object, then those ID values are the same value. This allows cooperating threads in different processes or jobs to communicate object IDs to one another.

Notice that this property even extends to objects in ancestor process-display object containers. This allows a thread in one process to communicate an object ID of an object in its process-display object container to a thread in a descendant process (process-display object containers are discussed in Section 1.2.6.5).

1.2.11 Translation of an Object ID to an Object Header Address

In order to translate an object ID to the address of the associated object, Mica performs the following steps:

1. Uses the level number as an index into the process control block to locate the corresponding level directory mutex.
2. Acquires the directory mutex.
3. Uses the level number as an index into the process control block to locate the corresponding container directory.
4. Compares the container directory index field to the size of the table to ensure that index is within the object array.
5. Uses the container directory index field to index into the object array.
6. Checks the object array element to ensure it contains the address of an object header.
7. Uses the resulting system address to locate the object container header for the object container.

8. Compares the object index field to the size of the table to ensure that index is within the object array.
9. Uses the object index field to index into the object array. Checks the resulting value to ensure that it is the address of an object.
10. Compares both sequence number fields in the object header to the sequence number fields in the object ID. To do this, Mica checks the address of the object header, which is found in the object array. If they are not identical, then that object ID is not valid.
11. Increments the pointer count field in the object header so the object's storage cannot be deallocated while a pointer to the object is held.
12. Releases the directory mutex, and returns a pointer to the object body to the caller.

The routine *obj$reference_object_by_id* described in Section 1.7.6 provides the mechanism to translate an object ID to the address of the object header. All routines within the executive use *obj$reference_object_by_id* for translating object IDs.

Figure 1–2 indicates how the structures fit together.

1.2.12 Container Directory IDs

It is necessary for programs to find container directories for each level. Since no consumer of the object architecture may have any knowledge about the construction of an object ID, programs call a system service that returns the object ID of the desired container directory.

The container directories are named as follows:

- The system level container directory is named *exec$system_container_directory*.
- The job level container directory is named *exec$job_container_directory*.
- The process level container directory is named *exec$process_container_directory*.

The *exec$translate_object_name* service may be used to obtain the object ID of a container directory by translating one of these three names. The caller must specify the container directory name for both the object name to locate and the container directory in which to locate the name.

1.2.13 Object Access By ID Only

One guideline to follow in system software design is that user-mode access to objects is only allowed by object ID, and is not allowed by name. If the object must be located via a name, the translation from name to ID is performed in a separate call preceding the object access call. This simplifies the coding of the object service routines, since they do not have to do any name translations or deal with logical names.

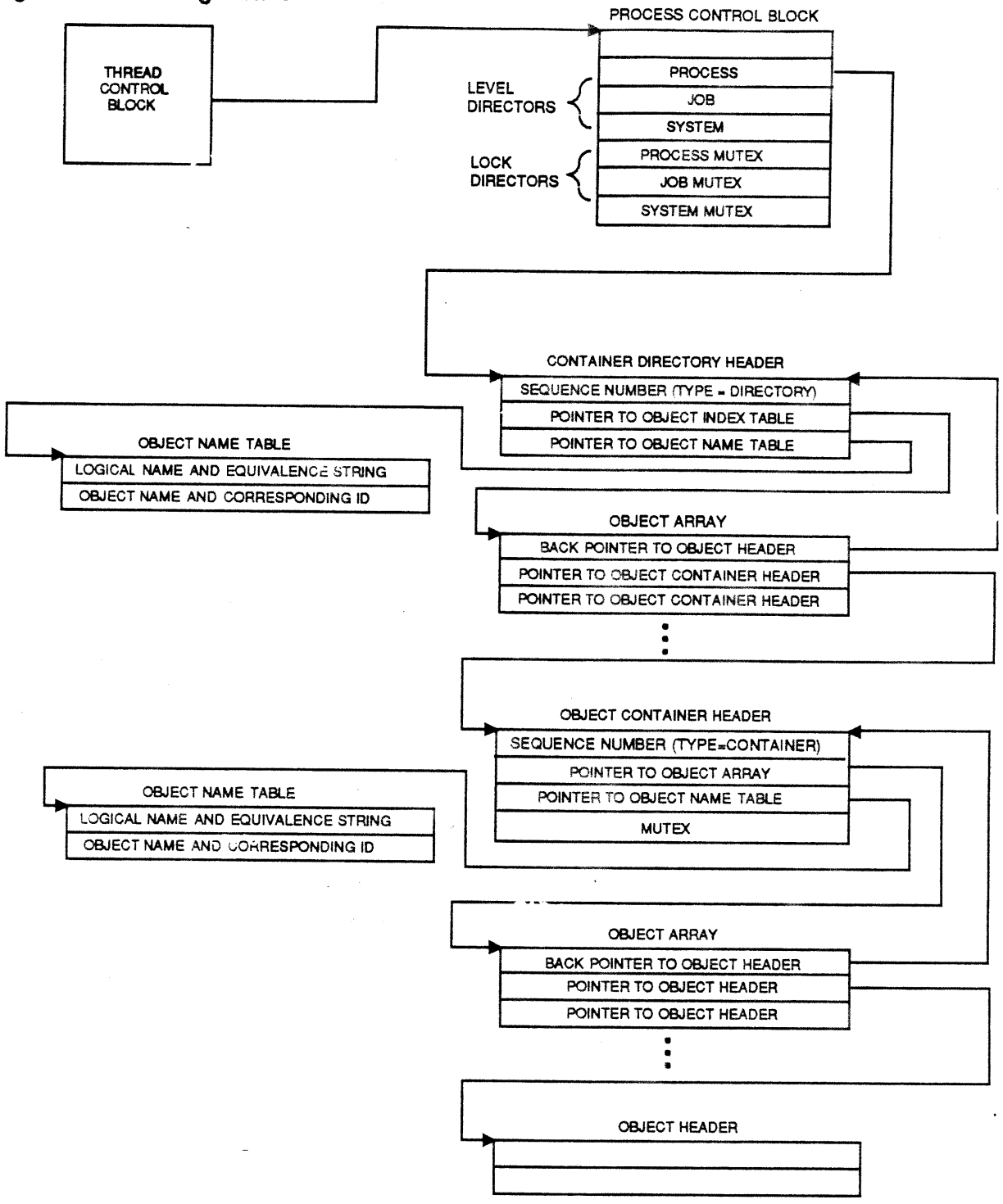
1.2.14 Object Service Routines

Each type of object has an associated set of services called *object service routines*. The service routines for a particular object type perform the operations supported by that object type.

For example, a FRAMITZ object type may define the operations:

- create_framitz
- get_framitz_information
- clear_framitz
- juggle_framitz

Figure 1–2: The Big Picture



- read_framitz
- close_framitz

Each of these operations is represented by a separate object service routine. These object service routines can be added to the system dynamically, but only as a complete group.

Every set of object service routines must supply a method for creating the object and providing information about the object. These services are named *create*_object_type, and *get*_object_type_*information*. Each set of object service routines must also implement routines to provide system defined object type-specific operations.

These operations are:

- Remove
- Delete
- Shutdown
- Allocate
- Deallocate

These operations are used by the executive, and, in some cases, by the object's service routines. The availability and location of these functions, as well as some status and control information, are provided in object type descriptors (OTDs).

1.3 Object Type Descriptor (OTD)

There is a single object type descriptor (OTD) object for each object type defined by the system. All OTD objects are created in a single system container.

An OTD object, hereafter referred to as just OTD, contains general information about an object type, not a particular instance of an object type. For example, an OTD indicates whether an object type supports the notion of wait. Anything an OTD specifies about an object type applies to all instances of that object type.

Part of the information in an OTD is the location of a number of routines that may be called by the executive. These routines have standard definitions across all object types but require object type-specific processing. These routines are located by the executive via standard offsets into the OTD. The operations are listed in the previous section.

Users of an object type do not need to know anything about the routines pointed to by the OTD. The designer of an object type, however, must specify and implement them.

1.3.1 Object Type Descriptor Body Format

```
e$object_type_descriptor : RECORD
    otd$type : e$object_type;
    otd$objhdr_listhead : e$linked_list;
    otd$count : integer;
    otd$dispatcher_object_offset : integer;
    otd$access_mask : POINTER e$access_mask;
    odt$allocation_listhead_offset : integer;
    otd$create_disable : bit; ! 0 - enable, 1 - disable
    otd$mutex : k$dispatcher_object (mutex);
    otd$allocate : obj$obj_allocate_procedure;
    otd$deallocate : obj$obj_deallocate_procedure;
    otd$remove : obj$obj_remove_procedure;
    otd$delete : obj$obj_delete_procedure;
    otd$shutdown : obj$obj_shutdown_procedure;
END RECORD;
```

All fields of the OTD have a standard definition. This allows the executive to use the information in an OTD without having detailed knowledge of the corresponding object type.

Figure 1–3: Object Type Descriptor (OTD) Format

OTD$TYPE (Type)	
OTD$OBJHDR_LISTHEAD (Linked List)	
OTD$COUNT (Count)	
OTD$DISPATCHER_OBJECT_OFFSET (Dispatcher Offset)	
OTD$ACCESS_MASK (Access Mask)	
OTD$ALLOCATION_LISTHEAD_OFFSET (Allocation Listhead Offset)	
OTD$CREATE_DISABLE (Create Disable)	
OTD$MUTEX (Mutex)	
OTD$ALLOCATE (Allocate)	
OTD$DEALLOCATE (Deallocate)	
OTD$REMOVE (Remove)	
OTD$DELETE (Delete)	
OTD$SHUTDOWN (Shutdown)	

The purpose of each of these fields is described below.

1.3.1.1 otd$type Field (Static)

This field contains a value indicating the type of object the OTD represents.

1.3.1.2 otd$objhdr_listhead Field (Dynamic)

This field contains the forward link to the first object header of this type, and the backward link to the last object header of this type. These links are used for consistency checking within the object architecture.

1.3.1.3 otd$count Field (Dynamic)

This field contains a count of the number of objects of the corresponding type that currently exist in the system. Modification of this field must be done using the RMALI instruction.

1.3.1.4 otd$dispatcher_object_offset Field

When nonzero, this object type supports the notion of wait. The value in this field is the offset of the dispatcher object (dispatcher objects are described in Chapter 6, The Kernel) from the start of the object body. Thus, when a wait operation is performed on the object, the executive adds the field contents to the address of the object header and issues a kernel wait operation on the dispatcher object.

The dispatcher objects that support waiting are:

- Events
- Semaphore
- Timer
- Thread
- Queue

These objects are created by providing the necessary structure in the object body and calling the kernel routine *k$initialize_xxxxx*, where xxxxx is the type of dispatcher object to be initialized. Once initialized, these objects can then be manipulated by using other kernel routines. For more details on the kernel support provided, see Chapter 6, The Kernel.

Any object type which has a non zero dispatcher field must allocate the object body from nonpaged pool. The wait routines which accept object types with non-zero dispatcher offsets call the kernel wait routines which operate at IPL 2 and cannot take page faults.

1.3.1.5 otd$access_mask Field (Static)

This field points to the supported access mask for use by the security routines in access validation. For more information, see the Chapter 11, Security and Privileges.

1.3.1.6 otd$allocation_listhead_offset Field

When nonzero, this object type may have objects allocated to it. For example, thread, process and job objects all support the notion of object allocation. Object allocation is described in Section 1.8.

1.3.1.7 otd$create_disable Field (Dynamic)

This flag may be set by the executive to prevent additional objects of this type from being created. It can be used to shut down the system in an orderly fashion.

Once set, this field may not be cleared. Therefore, access does not have to be interlocked. This flag is set when it contains a nonzero value.

1.3.1.8 otd$mutex Field

Provides synchronization for creation, deletion, and state changes among objects within a type.

1.3.1.9 otd$allocate Field

This field points to an allocate procedure. The allocate procedure is called in kernel mode at IPL 0 with the allocation mutex acquired. The address of the object body to be allocated and the allocation type are passed as input parameters.

The allocate procedure has the following type declaration:

```
obj$object_allocate_procedure :
    PROCEDURE (
        IN object_body : POINTER anytype CONFORM;
        IN allocation_object_hdr : POINTER e$object_header;
        );
```

The procedure *obj$null_allocate_procedure* is provided for use by object types which do not have an allocate procedure.

1.3.1.10 otd$deallocate Field

This field points to a deallocate procedure. The deallocate procedure is called in kernel mode at IPL 0 with the allocation mutex acquired. The address of the object body to be deallocated is passed as an input parameter.

The deallocate procedure has the following type declaration:

```
obj$object_deallocate_procedure :
    PROCEDURE (
        IN object_body : POINTER anytype CONFORM;
        IN allocation_object_hdr : POINTER e$object_header;
        );
```

The procedure *obj$null_deallocate_procedure* is provided for use by object types which do not have an deallocate procedure.

1.3.1.11 otd$remove Field

This field points to a remove procedure. The remove procedure is called when an object's object ID count field is decremented to 0. It is called in kernel mode, at IPL zero. It is passed two arguments, the address of the object body and the mode (user or kernel).

The remove procedure allows the object type-specific procedure to perform any necessary actions now that the ID of the object is removed. Once object ID count field is zero, it is impossible for a user to translate an object ID which points to this object.

The remove procedure has the following type declaration:

```
obj$obj_remove_procedure :
    PROCEDURE (
        IN object_body : POINTER anytype CONFORM;
        IN access_mode : k$processor_mode;
        );
```

The procedure *obj$null_remove_procedure* is provided for use by object types which do not have a remove procedure.

1.3.1.12 otd$delete Field

This field points to a delete procedure. The delete procedure is called when an object's pointer count field is decremented to zero. It is called in kernel mode at IPL zero. It is passed the address of the object body.

The delete procedure is responsible for manipulating object type-dependent data structures and deallocating the storage allocated for the object body extensions.

The delete procedure has the following type declaration:

```
obj$obj_delete_procedure :
    PROCEDURE (
        IN object_body : POINTER anytype CONFORM;
        );
```

The procedure *obj$null_delete_procedure* is provided for use by object types which do not have a delete procedure.

1.3.1.13 otd$shutdown Field

This field points to a shutdown procedure. The shutdown procedure is called once when an object type is permanently removed from the system (presumably at system shutdown time). It is called by the executive in kernel mode at IPL 0 with ASTs enabled. The purpose of this routine is to provide a way to perform object type-specific shutdown operations. Among other things, this provides the opportunity to dump any statistics that may have been gathered relating to the object type.

The parameters passed to this routine include the address of the OTD and the address of a callback routine that can be used to output information to a shutdown log file. This routine must be provided even if it simply returns without performing any actions.

The shutdown procedure has the following type declaration:

```
obj$obj_shutdown_procedure :
    PROCEDURE (
        IN otd_hdr : POINTER e$object_header;
        IN log_procedure : e$log_procedure;
        );
```

The procedure *obj$null_shutdown_procedure* is provided for use by object types which do not have a shutdown procedure. procedure.

1.4 Object Header

The object header is a fixed-format data structure that contains object type-independent data. This header is used by the executive without necessarily knowing the type of object it is accessing.

```
e$object_header : RECORD
    objhdr$pointer_count : integer;
    objhdr$object_id_count : integer;

    objhdr$type : e$object_type;
    objhdr$otd : POINTER e$object_header;
    objhdr$link : e$linked_list;

    objhdr$container : POINTER e$object_header;
    objhdr$level : e$level;
    objhdr$index : e$index;
    objhdr$seq_number_low : e$seq_number;
    objhdr$seq_number_high : e$seq_number;
```

```
        objhdr$dispatcher_object : POINTER anytype CONFORM; !# POINTER k$dispatcher_object    ;
        objhdr$name : POINTER e$object_name_block;
        objhdr$owner : e$identifier;
        objhdr$acl : POINTER e$access_control_list;
        objhdr$allocation_block : POINTER e$object_allocation_block;
        objhdr$access_mode : k$processor_mode;
        objhdr$transfer_flag : bit; ! 0 - transfer not allowed
                                    ! 1 - transfer allowed
        objhdr$reference_inhibit_flag : bit; ! 0 - reference ids allowed
                                             ! 1 - reference ids not allowed
        objhdr$temporary_flag : bit; ! 0 - permanent
                                     ! 1 - temporary
        objhdr$temporary_operation : bit;     ! 0 - make temporary
                                              ! 1 - mark temporary
    END RECORD;
```

1.4.1 Object Header Data Structure

Each of the fields in the object header are defined to be either static or dynamic. If a field is defined as static, its value is established during object creation, and remains constant for the life of the object. Static fields may be accessed any time the pointer count field has a nonzero value (see the description of the *objhdr$pointer_count* field in Section 1.4.1.1). However, if a field is defined to be dynamic, it is subject to modification. Access to dynamic fields is controlled by a protocol that is specific to each field. The specific protocol indicating when each dynamic field may be accessed is defined with the description of the field.

1.4.1.1 objhdr$pointer_count Field (Dynamic)

Whenever an access to an object will span a period of time that can not be protected by use of mutex locks on appropriate data structures, the pointer count field must be incremented. In particular, an address of an object may not be used to regain access to the object unless the pointer count has been previously incremented.

The pointer count represents the number of pointers that have been taken out on the object, plus one for a nonzero object ID count. When an object ID is translated by an object service routine, the pointer count is incremented by one. The pointer count for an object signifies the number of reasons the storage for an object should not be deallocated.

To increment this field, the directory-level mutex must first be held (see Section 1.5.3.4, below). This is necessary to avoid race conditions with the object ID being deleted.

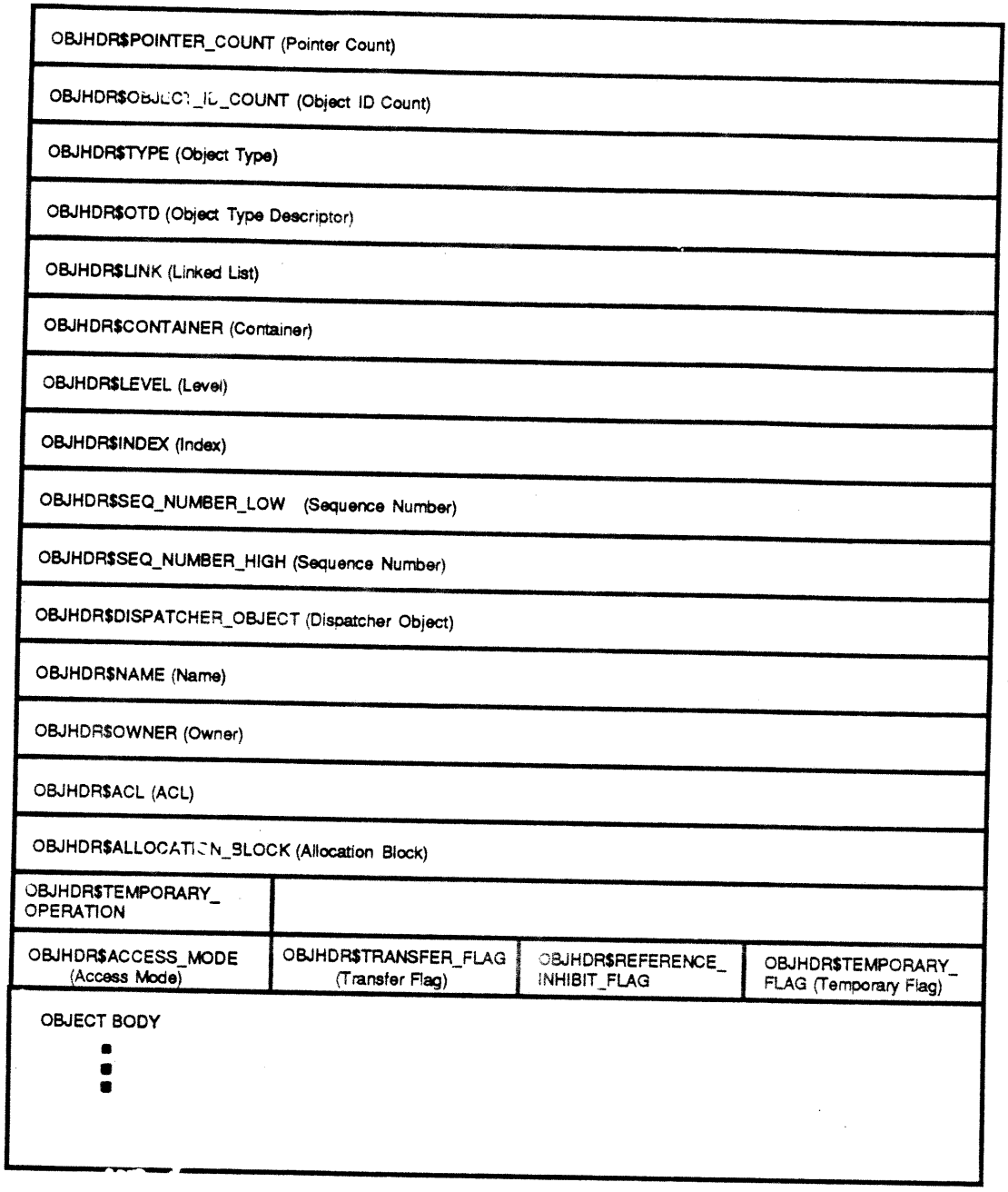
When an object ID, object container pointer, or reference object ID is deleted, the object ID count of the principal object is decremented. If the resultant object ID count is zero, the object type-specific remove routine is called to initiate any object specific removal action such as canceling I/O. A zero object ID count also causes the pointer count to be decremented, as does the dereferencing of a pointer. When the pointer count of an object reaches zero, the object type-specific delete routine is called, and the object header storage is deallocated.

Note, if the object ID count is zero then no object container has a pointer to that object header.

One of the major goals of the lock strategy for objects is to allow the pointer count to be decremented (using the RMALI instruction) without having to hold a lock to do so. This allows low overhead dereferencing of objects on such operations as I/O, wait, and the deleting of a reference object.

If the pointer count is zero and the object ID count is nonzero, a bug check is issued.

Figure 1–4: Object Header Format



1.4.1.2 objhdr$object_id_count Field (Dynamic)

The object ID count represents the number of object IDs or container directory index table pointers that can be used to refer to the object. For all objects except container objects, this is the number of object IDs that refer to the object. For container objects, this is the number of container directories that can refer to this object container (note that the object ID itself is not counted). Hence, for object containers, the object ID count is one unless the container is a display container.

The object ID count signifies the number of reasons that the remove action for an object should not be invoked.

If an object is marked as temporary (as discussed in Section 1.4.1.18, below), then it must have at least one reference to it.

To increment this field, the mutex field of the object container in which the object resides and the directory-level mutex must first be acquired (see Section 1.5.3.4, below). This is necessary to ensure that the object cannot be deleted while another reference is being established.

Note that this field is never directly incremented or decremented; the executive routines for object support operate upon this field.

1.4.1.3 objhdr$type Field (Static)

This field contains the integer value of the object type. When an object ID is translated to an object header, the type field is compared to the desired object type.

1.4.1.4 objhdr$otd Field (Static)

The object type descriptor field contains a pointer to the object type descriptor (OTD) for this particular object type. The executive and the object type's service routines use this field to use information or routines associated with the OTD. Note that there is exactly one OTD per object type defined in the system.

The OTD is described in detail in Section 1.3.1.

1.4.1.5 objhdr$link Field (Dynamic)

This field contains the forward link to the next object header of this type, as well as the backward link to the previous object header of this type. It is used for debugging purposes.

1.4.1.6 objhdr$container Field (Dynamic)

The container field is a pointer to the object container in which the object resides.

This field is dynamic only because the object may be transferred to a new object container. Such a transfer would result in the assignment of a new object ID. This field is zeroed when the principal object ID is removed from its object container.

This field may only be accessed by a thread holding both the directory level mutex and the mutex within the object container.

1.4.1.7 objhdr$level Field (Dynamic)

The level (system, job, process) at which the object's principal ID exists. This field is unaffected if the principal ID is deleted.

1.4.1.8 objhdr$index Field (Dynamic)

This field is the index into the object array of the container object specified in the container field for this object.

This field is dynamic only because the object may be transferred to a new object container. Such a transfer would result in a new object ID being assigned. This field may only be accessed by a thread holding both the directory-level mutex and the mutex within the object container.

1.4.1.9 objhdr$seq_number_low and objhdr$seq_number_high Fields (Dynamic)

When an object ID is translated into the address of an object, the value in the object ID's sequence field is compared to the value in the object header's sequence field. If they are not identical, then the object ID has been reallocated, and the one specified for translation is no longer valid.

This field is dynamic only because the object may be transferred to a new object container. Such a transfer would result in the assignment of a new object ID. This field may only be accessed by a thread holding both the directory-level mutex, and the mutex within the object container.

Note that it is possible to construct the object ID given an object header. Two parts of the object ID are in the object header, the sequence number and the object index. The container field points to the object header of a container. The index field in that container is the directory index. The level value in the container directory body is the level portion of the ID.

1.4.1.10 objhdr$dispatcher_object Field (Static)

If the object type allows wait operations this field points to the dispatcher object to be used in the kernel wait operation. If the object type does not support the notion of wait, this field has the value nil.

1.4.1.11 objhdr$name Field (Dynamic)

This field is used to associate a name string with an object. If the object has an associated name, this field contains a pointer to an name definition block which contains the name. Otherwise, this field has a value of zero.

When the object ID is deleted, the associated name, if any, must also be deleted.

The level directory, the object container, and the type-specific mutex must all be acquired in order to modify this field.

See Section 1.5.5 for a complete description of how object names are managed.

1.4.1.12 objhdr$owner Field (Static)

The identifier of the owner of the object. This field is used by the security access validation routines. For more information see Chapter 11, Security and Privileges.

1.4.1.13 objhdr$acl Field (Dynamic)

The format and use of this field is defined in Chapter 11, Security and Privileges.

This field is a pointer to the ACL for the object. A value of nil indicates that no ACL exists for the object.

1.4.1.14 objhdr$allocation_block (Dynamic)

The field is used by the object translation routines to ensure that the thread has access to the object. In order to translate an object ID to an object header, the requesting thread must have access to the object as determined by the ACL, and must be in the allocation class of the object. Object allocation and allocation classes are discussed in Section 1.8.

The allocation mutex and the type mutex must be acquired in order to modify this field.

1.4.1.15 objhdr$access_mode Field (Static)

This field contains the owner mode (kernel or user) of the object. Kernel-mode objects may not be created in user owned containers. Mode is also used to verify access to the object by the access validation procedures as described in Chapter 11, Security and Privileges.

1.4.1.16 objhdr$transfer_action Field (Static)

When clear (false) this flag indicates that the object cannot be transferred to another container. The transfer action field state is determined at object creation from the transfer state flag in the OTD. This flag is examined when an attempt is made to transfer an object using the *exec$make_temporary* service or job/process creation services.

Transfer action is discussed in Section 1.7.8.

1.4.1.17 objhdr$reference_inhibit Field (Dynamic)

When set (true) an attempt to create a reference ID to this object results in a failure.

1.4.1.18 objhdr$temporary_flag Field (Dynamic)

When set, this flag indicates the object is a temporary object. A temporary object has the property that when all reference objects to the object are deleted, then the object itself is also deleted. Therefore, when the object ID count field is decremented to one, this field is checked. If it is set, then object deletion may be invoked. See Section 1.7.7 for more information.

The level directory, the object container, and the type-specific mutex must all be acquired in order to modify this field.

1.4.1.19 objhdr$temporary_operation Field (Dynamic)

This flag describes the type of operation that was used (make temporary or mark temporary) to make the object temporary. This flag is only meaningful when the *objhdr$temporary_flag* is set.

1.4.2 object_body Record

The purpose and use of this record is left to the designer of a particular object type.

The *object_body* record is used to store object type information. The format and meaning of fields within this record are entirely the responsibility of the object's designer. It is also up to the designer of the object type to define the protocols for accessing various fields within this record.

In many cases, the *object_body* record can be allocated in a single block (with the object header) and is sufficient to hold all the object type-specific information associated with an object. Certainly this is true for simple objects like events. For more complex objects, like object containers, it may be necessary to have other data structures associated with the object.

The content and management of these additional data structures is entirely the responsibility of the object's designer. They must be connected in some fashion to the *object_body* record so that they are deleted when the object is deleted.

Object bodies are allocated from paged or nonpaged pool. If the object body contains any kernel dispatcher objects or kernel control objects it must be allocated from nonpaged pool.

Object bodies should not contain the object IDs of any objects. If a need exists to refer to another object a pointer to that object's body should be used. This prevents dependencies on other object IDs being valid.

1.5 Object Container Data Structures

Object containers are objects. Therefore, their primary data structures are:

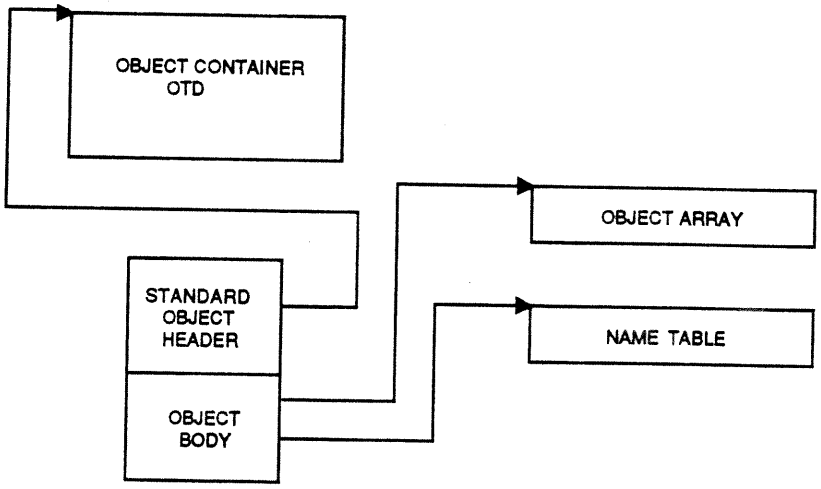
- Object container object type descriptor (OTD)
- Object container object header
- Object container body

To track objects and their associated names, object container bodies have two primary associated data structures. These data structures are:

- Object array
- Name table

Figure 1–5 illustrates the relationships between these object container data structures.

Figure 1–5: Object Container Data Structure Relationships



In addition to these data structures, object names are stored in name definition blocks.

\The exact data structures used to support object names, and is subject to change following performance analysis yet to be performed. The algorithms for manipulating the ultimate data structures will have to be implemented to match the data structures, and so, are also subject to change.\

The aspects of the current design that are not expected to change are:

- The name field of the object header points to a data structure containing the case-sensitive name of the object. This pointer provides the link necessary to delete the object name when the object is deleted.
- A field in the object container body (currently called the name table field) is used to locate the name management data structures for name assignment and translation purposes.

1.5.1 Container Directories as Compared with Object Containers

Container directories are similar in structure to object containers. The object header is the same, except for type, and the object body is nearly identical. The only differences are:

- Container directories do not have a mutex in their object bodies.
- Only container directories have a display index field.

There are some restrictions placed on the container directory that are enforced by the executive object manipulation routines. They are:

- The only objects that can be pointed to by (contained within) a container directory object are object containers.
- The only names that can be stored in a container directory are the names of the object containers that are pointed to by the container directory and logical names.

1.5.1.1 Object Container OTD Remove Procedure

When an object container is deleted, the OTD remove procedure removes the ID of each object within that container by calling the routine *obj$remove_obj_from_container*.

1.5.2 Object Container Object Header

The fields of the object container's object header are initialized and used as defined in Section 1.4. Note that the object container is an object, and is pointed to by the first pointer in the object index table.

Also, the object index field of the object header has a value of zero for a container directory. Object containers contain the value corresponding to the object index array element in the container directory, which points to this object container.

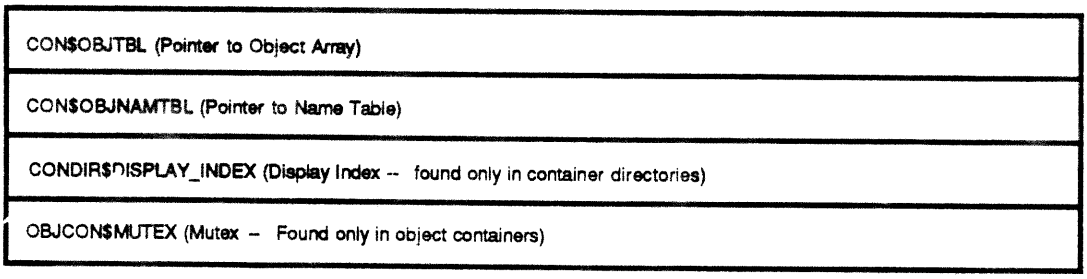
The following restrictions exist for object containers:

- An object container (or container directory) cannot be allocated.
- An object container (or container directory) cannot be temporary.

1.5.3 Object Container Body

The format of the object container's body is shown in Figure 1–6.

Figure 1–6: Object Container Body Data Structure



The purpose of each of these fields is described below.

The object array field is a pointer to the object container's object array. The object array is described in Section 1.5.4, below.

Note that the first object pointer of this table points to the object header for this container.

1.5.3.1 con$objtbl Field

This field contains a pointer to the object array.

1.5.3.2 con$objnamtbl Field (Static or Dynamic, Depending on Implementation)

The name table field is a pointer to the table used to translate names for objects and logical names in this object container. The name table is described in Section 1.5.5, below.

1.5.3.3 condir$display_index Field (Static)

This field is only in the container directory.

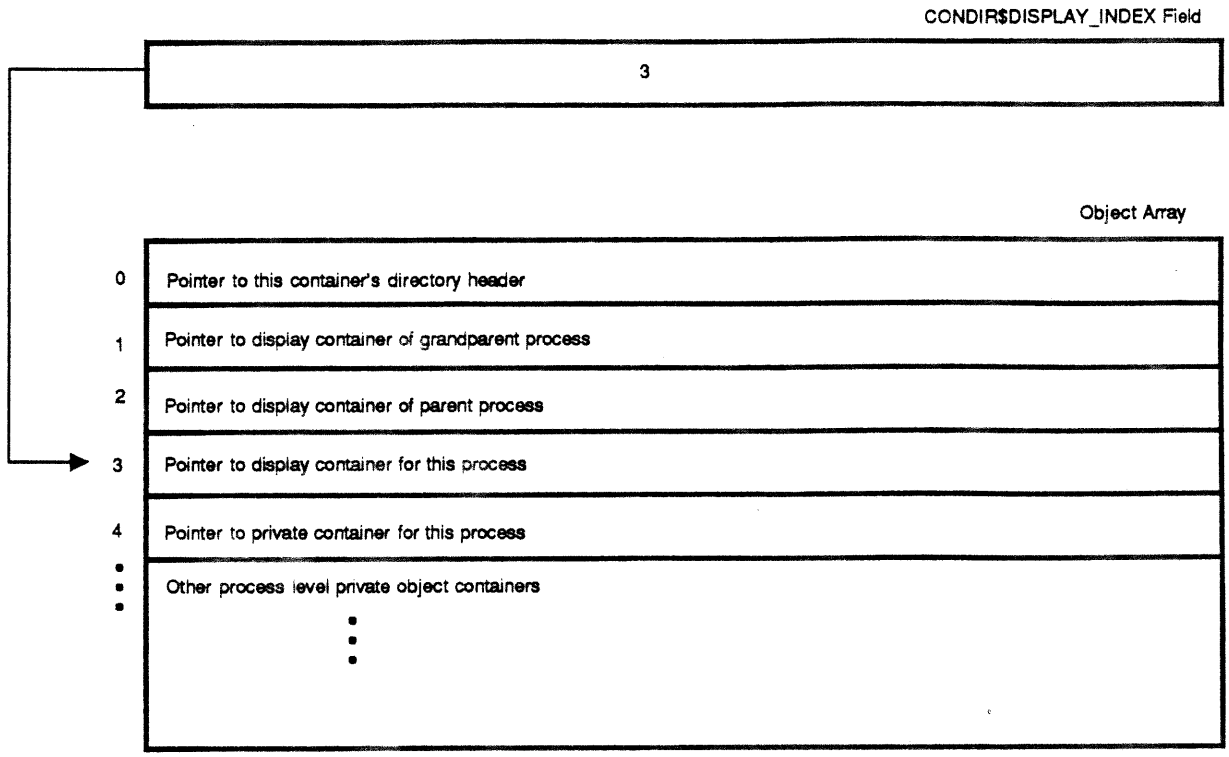
The purpose of the display index field supports the capability of a process to access objects in its ancestors' display object containers (as discussed in Section 1.2.6.5).

The first element in the object array (element 0) always points to its own object header.

In all cases except process level, the second element (element 1) always points to the first object container for that level. In these cases, the display index field contains the value one.

However, for process level container directories, the process's ancestors' display containers are pointed to by the second through nth elements (elements 1, 2, 3, etc.). In this case, the display index field contains the element number containing the current process's display object container. This element is then followed immediately by private process object containers. Figure 1–7 illustrates the use of the display index field in a process level container directory.

Figure 1–7: Container Directory Header Display Index Field Usage



1.5.3.4 objcon$mutex Field (Static)

The mutex field is a kernel mutex which is operated on by issuing calls to procedures within the kernel. These procedures are described in Chapter 6, The Kernel.

This mutex is used to synchronize access to certain fields in the object header.

1.5.4 Object Array Data Structure

The primary function of the object array is to hold a pointer to each created object in the object container. The format of the object array is shown in Figure 1–8.

The purpose of each of these fields is described below.

1.5.4.5 objtbl$max_index Field

This field indicates the current length of the object array by providing a count of the number of elements in it minus one. The value in this field is the maximum valid array index value.

This field value has an architectural limit of 1,048,575 (20 bits).

Figure 1–8: Container Object Array Format

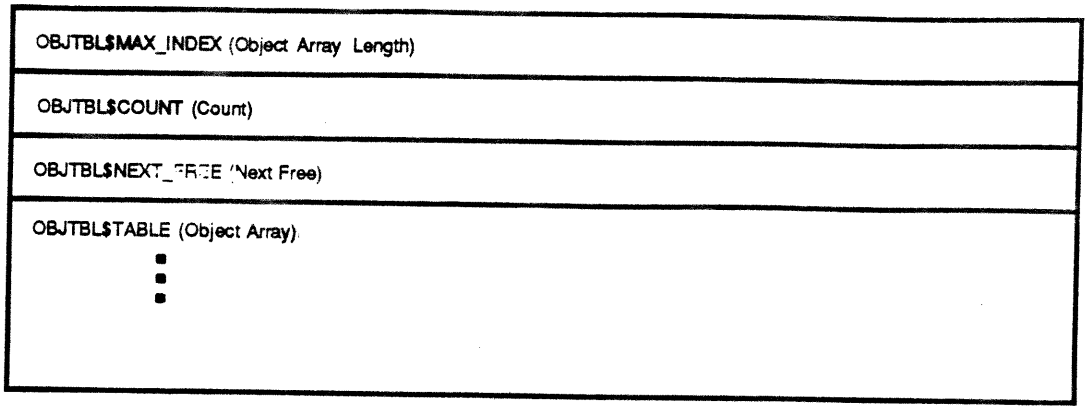
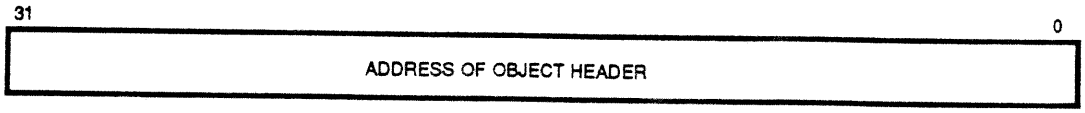


Figure 1–9: Address of Object Header



1.5.4.6 objtbl$count Field

This field is initialized to 1 when the object container is created, to represent the fact that this object container ID is in the array. It is then incremented each time an object ID within this object container is created. Similarly, it is decremented when an object ID in this object container is removed from the container as part of deletion.

1.5.4.7 objtbl$next_free Field

When created, an object container has a pointer to itself as the first element of the object array, and the remaining elements are free (unused). To facilitate quick allocation of free object array elements, they are organized into a linked "free list".

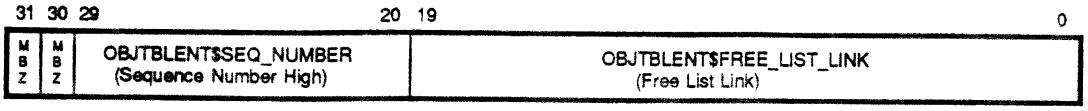
The next free field is the listhead of the free list, and contains the index of the next object array element to be allocated. The free list link field of the first element contains the index of the second, which contains the index of the third, and so on. The last element has a free list link value of zero to indicate the end of the list has been reached.

1.5.4.4 objtbl$table Field (Object Array)

Object array elements are used for two purposes. First, each object in the object container has an element in the object array that is associated with it. Such an element contains a pointer to the header of the object. In this case, the object element is referred to as "in use." The format of an object array element when it is in use is discussed below.

As discussed in Section 1.2.4, each time an object ID is allocated, it receives a new sequence number. This sequence number high field is associated with an object array element, and is stored within the element when the element is not in use. When the element is in use, the sequence number is moved to the sequence high field of the associated object's header. When the object is removed from the object container, the sequence number is incremented, reset to zero on overflow, and moved back into the object array element.

Figure 1–10: Free Object Array Element Format



The format of a free object array element is:

The free list link field is used to link all unused object array elements together on a single list for quick allocation (as discussed in the next section).

Notice that the most significant bit (MSB) of object array elements may be used to determine whether the element is currently in use or free. If the MSB is 1, then the element contains an address in system address space and the element is currently in use. If the MSB is 0, then the element is currently free and contains the next sequence number to use and a link to the next element on the free list.

Object array element index values begin at 0 and extend to the value in the *objtbl$max_index* field.

1.5.5 Name Table Data Structure

The purpose of the name table is to track both logical and object names within an object container.

When an object container is created, a name table must also be allocated, and the address of the name table must be stored in the name table field of the object container's body. The name table contains both object names which translate to an object ID and logical names which translate to an equivalence string.

The name table provides for a mechanism to translate names to name definition blocks. The actual implementation details of the name table need not be defined, only the functions provided by the name table are necessary. For example, the name table could be implemented as a hash table or a binary tree.

1.5.5.1 Name Definition Block

The name definition block contains information describing the name associated with a particular object or logical name. This information includes the type, mode, object ID, and other information about the name.

1.5.6 Object Naming Principles

Names are qualified by object type and mode. The combination of object type, mode, and name string is unique within a single object container.

For example, you can have two widget objects named SYS$ZEBRA in a single object container, but only if one is a user-mode object, and the other is a kernel-mode object. Likewise, you can have two user-mode objects named DEVICE_CONTROL in the same object container, but only if they have different object types.

Object names are limited to 255 characters in length. All object names are stored exactly as received (that is, case-sensitive) and may be looked up (translated) either case-sensitive or case-blind.

When a case-sensitive name translation is requested, either one or zero object IDs is returned. However, when a case-blind name translation is requested, many object IDs may match the name. To illustrate this, consider two objects, one named "pQr" and the other named "pqR". If a case-sensitive translation of "PQR" is requested, no object IDs is returned. If a case-blind translation of the same string, "PQR", is requested, then only the first name found which matches is returned.

Software which intends to use the case-blind feature of name translation is responsible for ensuring consistency. For example, all names strings could be converted to upper case before calling the object service routines. This prevents multiple names from matching in a case-blind search.

1.5.7 Naming

Names are used in the object architecture to label objects and equivalence strings. The same naming mechanism is used when a name is applied to either an object or an equivalence string. The only difference is the translation of an object name yields an object ID, while the translation of a logical name yields an equivalence string.

A name, like an object, exists within an object container. It is further qualified by the type of the object it names (logical names have the special object type of 0), and the mode in which it was created (user or kernel).

Through use of object name translation services a name can be searched for in any object container to which the user has access. The object name translation services provide recursion. See the description of *exec$translate_object_name* for more details.

The logical name translation services do not automatically provide recursion, rather it is the responsibility of the facility using the logical name translation services to provide for recursion, search lists, and parsing. This allows for a facility such as RMS to have different logical name evaluation rules.

1.5.7.1 Logical Names

Logical names are names that translate to equivalence strings. Each logical name can have up to 127 equivalence strings. The term "multivalued logical name" means a logical name with more than one equivalence string. Each equivalence string can be up to 255 bytes in length, and each equivalence string can have any, all, or none of the following attributes:

- Terminal—The equivalence string is not to be subjected to further logical name translations.
- Concealed—The application should not return the equivalence string to the user.

Logical names can have the following attributes:

- No show—The logical name should not be displayed in general show logical name services.
- No alias—The logical name cannot be duplicated in this table at an outer access mode. If another logical name with the same name already exists in the table at an outer access mode when a new logical name with no alias is created, it is deleted.
- Confine—The logical name is not copied from the process to its spawned subprocesses. This applies only to logical names at the process level in private object containers.

1.5.7.2 Object Services Providing Name Support

The following object services provide generalized name support.

- *exec$set_object_name*
- *exec$translate_object_name*
- *exec$clear_object_name*
- *exec$create_logical_name*
- *exec$translate_logical_name*
- *exec$delete_logical_name*

These routines are described in the Internal System Services Manual.

1.5.7.3 Container Directory Names

The three container directories to which the process control block points are named. The process container directory is named *exec$process_container_directory*, the job container directory is named *exec$job_container_directory*, and the system container directory is named *exec$system_container_directory*.

Each of the container directories has a name table (see Figure 1–2). This name table contains object container names which translate to the ID of an object container, and logical names, which translate to equivalence strings. When an object container is created, the name, if specified, is stored in the container directory's name table.

By creating "multivalued" logical names in the container directory name table, a search list for object containers can be created.

1.5.8 Object Name Translation

Object name translation involves taking a specified name and object type, and returning the related object ID (if any). The name table performs this function such that given a name, all name definition blocks containing that name are searched for the specified type.

The executive object support routine *obj$translate_object_name* provides the mechanism for object service routines to translate an object name to an ID.

```
PROCEDURE obj$translate_object_name (
    IN container_id : e$object_id;
    IN name : string (*);
    IN object_type : e$object_type;
    IN access_mode : k$processor_mode;
    IN case_blind : boolean;
    OUT object_id : e$object_id;
    ) RETURNS STATUS;
```

By specifying an OBJECT_TYPE of zero, the first object found which matches the name is returned.

obj$translate_object_name performs the following:

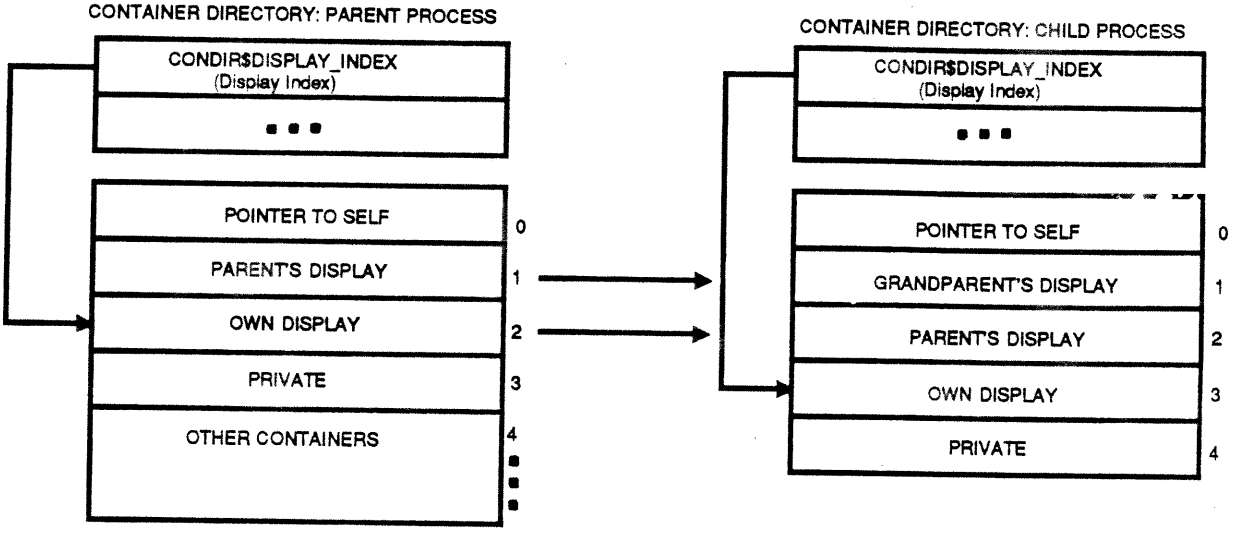
1. Acquires the directory-level mutex.
2. Translates the ID of the object container to a pointer to an object header and check access for read.
3. Ensures that the object header is a container or a directory.
4. Acquires the container mutex and releases the directory mutex if the object type is container.
5. Looks up the specified name in the container name table that matches the specified object type and mode.
6. Raises an error condition if the specified name is not found.
7. Returns the object ID which corresponds to the name if the name is found.
8. Releases the mutex.

1.5.9 Initializing a Process Level Container Directory

Container directories at system and job levels are always initialized to be empty (that is, they initially do not point to any object containers other than themselves). Container directories at the process level, however, must be initialized so that all of the process's ancestors' display containers are accessible. Also, in order to preserve the object ID of objects in its ancestors' display containers, it is necessary that they be pointed to by the same object array element in each process's container directory. To achieve this, process-display object containers are always the second through nth entries of a process level container directory.

A new process level container directory is initialized by copying elements starting at element one, and continuing through the value specified in the display index from the parent process's container directory. These entries are followed immediately by the new process's display and then private object containers. Figure 1–11 illustrates the initialization of a new (child) process's container directory.

Figure 1–11: Process Level Container Directory Initialization



In this example, the *condir$display_index* field of the parent's container directory indicates the last object container which must be copied to the new process's container directory. The new process's *condir$display_index* field is then updated to point to its own display container.

If the user deletes a display container, the table element within the container directory that pointed to the display container cannot be reused. If this were allowed, a private container could be copied as a display container.

1.5.10 System Global Variables and Data Structures

This section identifies the system global variables and data structures required to support the object architecture. The data structures that are accessible via known system locations include:

- System level container directory (see Section 1.2.8)
- System level directory mutex (see Section 1.2.6.2)
- Allocation mutex

1.5.10.1 OTD Objects

Each object of type OTD is created in a reserved system level object container. When a set of object services for an object type is loaded, the object service image is called at its initialization entry point. During the initialization operations, the corresponding OTD object must be created by calling the *obj$create_otd* service.

The *obj$create_otd* procedure has the following declaration:

```
PROCEDURE obj$create_otd (
    OUT otd_object_id : exec$object_id;
    IN  object_item_list : ITEM LIST CONFORM OPTIONAL;
    IN  transfer_state : boolean;
    IN  dispatcher_offset : integer;
    IN  shutdown_procedure : PROCEDURE e$shutdown_procedure;
    IN  remove_procedure : PROCEDURE obj$remove_procedure;
    IN  delete_procedure : PROCEDURE obj$delete_procedure;
    IN  allocate_procedure : PROCEDURE obj$allocate_procedure;
    IN  deallocate_procedure : PROCEDURE obj$deallocate_procedure;
    OUT otd_value : integer;
    );
```

At system shutdown, the operating system searches the OTD object container to locate each object type:

Each object type is unloaded by:

1. Setting the OTD's create disable flag (to prevent additional objects of that type from being created).
2. Calling the shutdown routine specified in that object type's OTD when all objects of a particular type have been deleted.

1.5.10.2 Allocation Mutex

The allocation mutex is a system-wide mutex which is acquired when an object's allocation is checked, modified, or deleted.

1.6 Relative Ordering of Object Architecture Mutexes

The following lists illustrates the relative ordering of object architecture mutexes. The mutexes are listed in order of lowest value to highest value.

Process level container directory mutex

- Process level object container mutex
- Job level container directory mutex
- Job level object container mutex
- System level container directory mutex
- System level object container mutex
- OTD type mutex
- Allocation mutex

1.7 Object Creation

1.7.1 Creating an Object

Each type of object is supplied with one or more service routines which create the object type. For example, a FRAMITZ object might be created with a *create_framitz* object service.

Object creation routines should have the following procedure declaration:

```
PROCEDURE exec$create_xxxxxx (          !where xxxxxx is the object name
  OUT xxxxxx_id : exec$object_id;
  IN object_independent_items : exec$object_parameters CONFORM OPTIONAL;
  IN object_dependent_items : exec$item_list CONFORM OPTIONAL;
    ) RETURNS STATUS;
```

The object independent items consist of the following:

- Object ID of object container in which the new object is to be created
- Name to associate with the object
- Access control information

The format and processing of object ownership and access control information is discussed in Chapter 11, Security and Privileges.

In addition to these standard parameters, object creation services may accept type-specific parameters. For example, an event creation service may accept an initial state of the event. The name of an object creation routine and the parameters allowed by that routine are specified by the designer of the object type.

1.7.2 Object Names

When creating an object, it is possible to assign a character string name to the object. This is particularly useful for objects that are to be shared. It allows one process to create an object (assigning it a name) and then, sometime later, another process can locate and access the object via its name.

Also, object names provide "create if" semantics. An attempt to create an object of same type, mode and name in the same container will fail, but the ID of the object which had the same name, type and mode is returned as is a warning status code.

1.7.3 Create Object Algorithm

The user calls an *exec$create_object* service specifying the container ID, ACL, returned object ID, and optionally a name. Other object type-specific arguments may be required.

Every create object service performs the following steps:.

1. Probes the user's arguments.
2. Calls *obj$create_initialize_object* passing in the following:
   - Object ID of OTD object
   - Pool type (paged or nonpaged)
   - The number of bytes of storage to allocate with the object header for the object body
   - The object name, if any
   - Ownership mode (kernel or user)
   - ACL
   - Transfer flag state

The *obj$create_initialize_object* routine locates the specified object type and allocates the storage.

At this point the object header has been created and initialized. The pointer count field is one, the object ID count and the container fields are zero.

3. Initializes the object body. This action is object type-dependent, for example, if the object supports waiting, a call to *k$initialize_xxxx*, where xxxx is a dispatcher initialization routine, is issued.

   At this point the object has been created but has no ID.

4. Attempts to add a pointer to the object header in the specified object container by calling *obj$insert_object_into_container* with the address of the object header, the container ID, the mode, and the object ID to be returned.

   obj$insert_object_into_container does the following:

   - Acquires the specified directory-level mutex.
   - Translates the ID of the object container to a pointer to an object header, and checks for write access. Ensures that the object header is a container.
   - Checks the transfer state of the container. If the container is transferable, and the object is not transferable, *obj$insert_object_into_container* returns an error status to the caller.
   - Acquires the container mutex.
   - Releases the directory mutex.
   - If an object name was specified, *obj$insert_object_into_container* checks to see if the same name already exists within this container with the same type and mode. If a name conflict occurs, *obj$insert_object_into_container* calls the OTD delete routine, releases the container mutex, and returns the ID that corresponds to the collided name.
   - The next free field in the object container's body is examined.
     - If the next free field is nonzero *obj$insert_object_into_container* does the following:
       * Removes the element specified in the next free field from the free list, and creates a new next free element.
       * Constructs the sequence number high field by using the value found in the object array, stored in the object header, and the address of the object header is stored in the element.
       * Constructs the object ID from the element number, the sequence number high, a random 10-bit number for the sequence number low, and the object index field of the object container.

         The object header is completed.
       * Sets the object ID count to 1, the container field points to the header of the object container, and so on.
       * Increments the create count field in the OTD for this object type using the RMALI instruction.
       * Increments the total ID field in the object container.
       * Releases the object container mutex, and returns the object ID to the user.
     - If the next free field is zero, *obj$insert_object_into_container* does the following:
       * Checks some resource quota to see if the object container can be expanded. If the object container cannot be expanded, *obj$insert_object_into_container* calls the OTD delete procedure. The container mutex is released, and a container full error is returned to the user.

* Allocates a new (larger) object array data structure.
* Copies the contents of the old object array to the new object array.
* Links all new object array entries beyond the length of the old object array into the free list.
* Sets the new object array's length.
* Changes the object array field in the object container to point to the new object array.
* Deallocates the old object array data structure.
* Creates the object ID as in the above case with a nonzero *objtbl$next_free* field.

At this point, if the object header is successfully added to the specified object container, both object ID count and pointer count fields contain the value one.

5. Adds the name, if any, to the object container name table. The container, the sequence number and object index fields are initialized.
6. Returns the object ID and status to the user.

If an ID cannot be created, the object type-specific delete procedure found in the OTD is invoked. Note that the container field and object ID count in the object's header were zero prior to the object's deletion. An error status is returned.

Once the object has been inserted in the specified container, the object creation routine may not reference the object body without calling the *obj$reference_object_by_id* routine.

The following status values could be returned from *obj$insert_object_into_container*:

- Invalid object container—Returned object ID is set to zero (error).
- Container full—Returned object ID is set to zero (error).
- Resource limit exceeded—Returned object ID is set to zero (error).
- Name collision—Returned object ID is set to the ID of the object with the same name and type (warning).
- Success—Returned object ID points to newly created object.

These rules cause all objects to be created in a uniform way. If two users attempt to create an object in the same container with the same name, type, and mode, only one is created, yet both users get returned a valid ID. One user gets a success code indicating the object was created, the other gets a warning code indicating the object already exists.

1.7.4 Object Modes

When creating an object, its mode (user or kernel) is established. Any object creation routine which is called from user mode creates a user-mode object.

Object creation routines which are called from kernel mode allow the mode to be specified as an argument. Object service routines that are called from user mode have the argument supplied by the jacket routine, which calls the corresponding entry point of the routine with the mode argument supplied.

This capability allows executive software to create objects that are either inaccessible to user mode or cannot be deleted from user mode.

Note that it is not possible to create an object owned by kernel mode in an object container owned by user mode. If this were allowed, the user would be able to delete the kernel mode object ID by deleting the container.

1.7.5 Object Access Protection

Object access protection is provided by a combination of user identifiers and access control lists (ACLs), and mode (user or kernel). This access protection information is stored for each object in its object header, and is processed by system-wide access validation routines without taking the object type into consideration.

Each object service routine is required to request access validation each time an object is accessed.

The exact format of the access protection information and procedures for performing access validation are provided in Chapter 11, Security and Privileges.

1.7.6 Object ID Translation

The executive procedure *obj$reference_object_by_id* provides translations from an object ID to a pointer to an object body. The procedure has the following declaration:

```
PROCEDURE obj$reference_object_by_id (
    IN object_id : e$object_id;
    IN object_type : e$object_type;
    IN access_mode : k$processor_mode;
    IN desired_access : e$access_type;
    OUT object_body : POINTER anytype CONFORM;
    ) RETURNS STATUS;
```

In order to translate an object ID to the address of the associated object, *obj$reference_object_by_id* performs the following steps:

1. Uses the level number as an index into the process control block to locate the corresponding level directory mutex.
2. Acquires the directory mutex.
3. Uses the level number as an index into the process control block to locate the corresponding container directory.
4. Compares the container directory index field to the size of the table to ensure that index is within the object array.
5. Uses the container directory index field to index into the object array.
6. Checks the resulting value to ensure that it is the address.
7. Uses the resulting system address to locate the object container header for the object container.
8. Compares the object index field to the size of the table to ensure that index is within the object array.
9. Uses the object index field to index into the object array.
10. Checks the resulting value to ensure that it is the address of an object.

    At this point, the address found in the object array is the address of the object header.
11. Compares both sequence number fields in the object header to the sequence number fields in the object ID. If they are not identical, that object ID is not valid.
12. Checks the object type to ensure this object is of the desired type.
13. Validates the intended access to the object by calling the security procedure *e$validate_access*.
14. Releases the directory mutex and returns the error status indicated by *e$validate_access* to the user if access is denied.
15. Does not validate allocation access if the intended access is show.

16. Checks to see if the object is allocated by examining the allocation information field in the object header.
17. If the allocation information field is not nil, then *obj$reference_object_by_id* performs the following steps:
    - Acquires the allocation mutex.
    - Checks the allocation information field in the object header again. If the field is nil, *obj$reference_object_by_id* releases the allocation mutex and continue as if the field was always nil.
    - Checks to ensure that the current thread is within the allocation class by calling the procedure *e$validate_allocation*.
    - Releases the allocation mutex.
    - Releases the directory mutex, and *obj$reference_object_by_id* returns the error indicated by *e$validate_allocation* to the user if allocation does not match.
18. Increments the pointer count field in the object header so the object cannot be deleted while a pointer to the object is held.
19. Releases the directory mutex, stores the pointer to the object body, and returns to the caller.

1.7.7 Object Deletion

The user calls the *exec$delete_object_id* service specifying the object ID of the object to delete.

The *exec$delete_object_id* routine performs the following steps:

1. Probes the user's argument.
2. Calls *obj$remove_obj_from_container* with the source ID and the mode of access.

The *obj$remove_obj_from_container* routine performs the following steps:

1. Acquires the source container directory-level mutex.
2. Acquires the source container mutex.
3. Translates the source object ID to a pointer.
4. Acquires the type-specific mutex.
5. If address of the source container matches the container field in the source object header, then *obj$remove_obj_from_container* zeros the container field and removes the object's name (if any) from the name table (in this case, this is the principal ID).
6. Decrements the object ID count of the source object.
7. Decrements the total ID field of the object container.
8. The *obj$remove_obj_from_container* routine performs the following steps if the resultant object ID count is zero:
    - Releases the type-specific mutex.
    - Checks the allocation block field in the object header. If the field is not nil *obj$remove_obj_from_container* does the following:
        — Acquires the allocation mutex.
        — Ensures the allocation block field is not nil.
        — Calls the type-specific deallocate routine, passing in the address of the object body and the allocation type.

  - — Clears the allocation pointer in the object's header.
  - — Unlinks the object allocation block and calls *obj$dereference_object*.
  - — Releases the allocation mutex.
  - — Returns the storage occupied by the object allocation block to the system.
- Removes the object ID and name from the source container. This frees the slot, updates the sequence number, and links it into the free list.
- Releases the source container directory level and the source container mutexes.
- Calls the type-specific remove object ID routine.
- Calls *obj$dereference_object* to decrement the *pointer count*.
- Returns to the caller.

9. If the resultant object ID count is one, the source object is temporary, and the container field in the source object's header is not zero, then *obj$remove_obj_from_container* performs the following steps:
   - Constructs the object ID of the principal object.
   - Deletes object ID and name from source container.
   - Releases the source container directory level, the source container, and the type-specific mutexes.
   - Calls an internal object routine to delete the principal ID. This routine is identical in function to *obj$remove_obj_from_container* with the following exception: The object ID count is examined once all the mutexes have been acquired. If the object ID count is not one, the principal ID is not deleted.
   - Returns to caller.
10. If the resultant object ID count is greater than one, or the object ID count is one and the source object is either not temporary or its principle object ID has already been deleted, then *obj$remove_obj_from_container* performs the following steps:
    - Removes the object ID and name from the source container.
    - Releases the source container directory level, the source container, and the type-specific mutexes.
    - Return to caller.

The procedure *obj$dereference_object* performs the following steps:

1. Decrements the pointer count field in the object header using the RMALI instruction.
2. If the pointer count field becomes zero, *obj$dereference_object* performs the following steps:
   - Issues a bug check if the object ID field is not zero.
   - Calls the OTD delete routine to deallocate the object body extensions and any associated structures.
   - Unlinks the object header from the OTD list.
   - Decrements the create count field in the OTD.
   - Deletes the object header and object body storage allocated in the *obj$create_initialize_object* object call.
3. Returns to the caller.

There is one interesting race condition that can arise during the deletion of an object ID that refers to a temporary object. This race condition, however, does not cause any system database to be corrupted. Assume that there are two threads that are simultaneously executing on two different processors: one is creating a reference to a temporary object, and the other is deleting a reference to the same temporary object. Further assume that the reference being deleted is the only reference to the object save its principle object ID.

Both threads execute the locking sequence using two different object IDs, and thus possibly acquiring different mutexes for the container directory and object container. The type-specific mutex, however, is the same, and one or the other of the threads acquires the mutex first. If the thread that acquires the mutex is the thread creating a reference, then nothing unusual can occur. If the thread that is deleting a reference acquires the mutex first, then the following anomaly can occur.

The thread deleting the reference discovers that the resultant object ID count is one, the subject object is temporary, and that its principle object ID has not been deleted. The reference object ID is deleted, the mutexes are released, and the Delete Object ID routine is called, specifying the principle object ID. But before the principle object ID can be deleted, the other thread completes the creation of the reference, thus incrementing the object ID count back to 2. It then releases its mutexes, whereupon the original thread now acquires the appropriate mutexes and checks the object ID count. The object ID count is not one and the principle object ID of the object is not deleted.

1.7.8 Transferring an Object Container

When a new process is created, it is given two object containers to serve as its display- and private-process level object containers. These object containers are created as part of process creation. See Chapter 5, Process Structure for more details on the Create Process and Job system service.
The object architecture provides support for the process and job creation services to transfer object containers from one job/process to a new job/process.

When an attempt is made to transfer an object container, every object in the container must meet the following conditions:

1. The thread issuing the system service must have write access to the object container.
2. The object being transferred must have an object ID count of one. This also prevents temporary objects from being transferred.
3. Reference objects must refer to a system level object.
4. The object being transferred must have the transfer action flag in the TRANSFER state.
5. If the object is allocated, the thread doing the transfer must pass the allocation check.

When an object is transferred, its name, if any, is transferred as well. If the object was allocated at the job, process, or thread level, then its allocation is transferred to the new entity being created. For example, if a job is being created, the object's allocation would transfer to the job level.

When a container is transferred, Mica performs the following steps:

1. Acquires the directory mutex.
2. Translates the container ID to an object header pointer with access as write.
3. Ensures that the resulting container object transferable. An error condition is raised if the container is not transferable. Displayed containers or the default private container are set as no transfer.
4. For each element in the object container Mica performs the following steps:
   - Checks to see if the object ID is a reference ID.
   - Checks the allocation of the object if the object ID is not a reference ID. If the allocation is at at thread level, then Mica transfers the allocation to the new entity.

If the object ID is a reference ID, checks to ensure that the principal ID is at the system level.

If all objects satisfy the criteria, removes the object container from the container directory and release the directory mutex.

5. Adds a pointer to the object container in the new container directory.
6. Updates the object header to point to the new container directory, also, the ownership of the objects needs to be changed to match the owner of the new entity.

1.7.9 Create Reference

When an object has a pointer to it outstanding, its data structures cannot be deleted. The executive can increment the pointer count field in the object header to prevent an object's storage from being deleted. A user-mode program may achieve similar behavior by creating a reference ID to the object ID.

Creating a reference ID creates another object ID for an object. When a reference ID is created for an object, the object ID count in the object's header is incremented. This allows a user to create a reference ID to an object, and thus ensures that the object's storage cannot be deleted until the reference object ID is deleted.

The level of the target container that receives the new reference ID must be less visible than the container of the principal object ID.

The *exec$create_reference_id* service, and the *exec$make_temporary* service both create reference IDs to the specified object.

The declaration for this executive function is:

```
PROCEDURE e$create_reference_id (
    IN mode : INTEGER;
    IN source_object_body : exec$object_id;
    IN target_container : exec$object_id;
    ) RETURNS reference_object_id;
```

The *e$create_reference_id* routine performs the following steps:

1. Raises an error condition if the level of the *target_container* is not less visible than the level of the source container.
2. Acquires the *target_container* directory-level mutex.
3. Acquires the *target_container* mutex.
4. Acquires the source container directory-level mutex.
5. Translates the source object ID to a pointer.
6. Checks for read access.
7. Checks the allocation class of source object container.
8. If the source object is of type container, then *exec$create_reference_id* releases the target container directory level, target container, and source container directory-level mutexes. An error condition is raised.
9. Allocates the object index in the target container.
10. If allocation of the object index fails, then *exec$create_reference_id* releases the target container directory level, target container, and source container directory-level mutexes. An error condition is raised.
11. Acquires the source object type-specific mutex.

12. Increments the object ID count of the source object.
13. Stores a pointer to the source object in target container at allocated index position.
14. Constructs the object ID of reference using the target container directory, the allocated target container index, and the sequence number of the source object.
15. Releases the target container directory level, target container, source container directory level, and type-specific mutexes.
16. Returns the object ID of newly created reference.

1.7.10 Make Temporary

Making an object temporary creates a new principal object ID for the object at a more visible level and marks the object as temporary. The original object ID becomes a reference object ID. If the object has a name, then the name is moved to the target object container.

When an object is made temporary, it is possible that a name conflict exists in the target container. The caller can specify that the object that is being made temporary is to be deleted and its object ID is to become a reference to the object which caused the name conflict. This provides the capability for two or more processes to share objects without regard to who created them first.

The declaration for this executive function is:

```
PROCEDURE e$make_temporary (
    IN mode : INTEGER;
    IN replacement : BOOLEAN;
    IN source : exec$object_id;
    IN target_container : exec$object_id;
    ) RETURNS reference_id : exec$object_id;
```

The *e$make_temporary* routine performs the following steps to make an object temporary:

1. If the level of the target container is not more visible than the source container, then *e$make_temporary* raises an error condition.
2. Acquires source container directory-level mutex.
3. Translates source object ID to a pointer.
4. Releases the source container directory-level mutex and raises an error condition if the source object is of type container.
5. Checks for delete access on source object.
6. Acquires target container directory-level mutex.
7. Acquires target container mutex.
8. Checks for temporary create access on target object container.
9. Acquires source object type-specific mutex.
10. If the source object ID is not the principle object ID, the source object ID count is not 1, or the source object is already temporary, then *e$make_temporary* releases the source container directory level, the target container directory, the target container, and the type-specific mutexes. An error condition is raised.
11. Checks for name conflict in target container if source object has a name.
12. If a name conflict exists and replacement is not specified, then *e$make_temporary* releases the source container directory level, the target container directory, the target container, and type-specific mutexes. An error condition is raised.

13. If a name conflict exists and replacement is specified, then *e$make_temporary* performs the following steps:
    - Raises an error condition if the object whose name conflicts is not temporary.
    - Increments the object ID count of the object in the target container whose name conflicts with the source object and save pointer to object.
    - Releases the target container directory level and target container mutexes.
    - Decrements the object ID count of the source object (the result is guaranteed to be zero).
    - Releases the type-specific mutex.
    - Calls the type-specific remove object ID routine for the source object.
    - Deletes the name of source object from source container.
    - Stores a pointer to the target container object in source container at the source object ID.
    - Retrieves the sequence number from the target object, and inserts in the source object ID.
    - Releases the source container directory-level mutex.
    - Returns the source object ID with the new sequence number inserted.
14. Allocates an object ID in the target container.
15. If allocation of the object index fails, then *e$make_temporary* releases the source container directory-level, the target container directory, the target container, and type-specific mutexes. An error condition is raised.
16. Stores a pointer to the source object in target container at the allocated index position.
17. Removes the source object name, if any, from its previous name table, and inserts it in the new target container name table.
18. Sets the principle object container address to the target container address.
19. Marks the source object temporary and indicates that a make temporary operation was performed.
20. Increments the object ID count of source object.
21. Releases the source container directory-level, the target container directory, the target container, and the type-specific mutexes.
22. Returns the source object ID.

1.7.11 Mark Temporary

The Mark Temporary routine allows an object to be marked as temporary after it has been created. This operation is used to cause a permanent object to be deleted when all its reference object IDs are deleted. If the object has no reference object IDs, then its principle object ID is deleted immediately. Objects can only be marked temporary that reside at the job or system levels.

The declaration for this executive procedure is:

```
PROCEDURE e$mark_temporary (
    IN mode : INTEGER;
    IN source : exec$object_id;
    )
```

The *e$mark_temporary* routine performs the following steps.

1. Raises an error condition if the level of the source object is a process.

2. Acquires the source container directory-level mutex.
3. Acquires the source container mutex.
4. Translates the source object ID to a pointer.
5. Checks for delete access on source object.
6. Acquires the source object type-specific mutex.
7. Sets the temporary flag in the source object and indicates that a mark temporary was performed. Note, if the object was already temporary, it does not change the state of the *objhdr$temporary_operation* flag.
8. If object ID count of source object is 1 and the source object ID is the principle object ID, then *e$mark_temporary* performs the following steps:
   - Decrements the object ID count (it is guaranteed to be zero).
   - Releases the type-specific mutex.
   - Calls the type-specific remove object ID routine.
   - Deletes the object ID and name from source container.
   - Releases the source container directory-level and source container mutexes.
   - Returns to the caller.
9. If object ID count of source object is not 1 or the source object ID is not the principle object ID, then *e$mark_temporary* releases the source container directory level, the source container, and the type-specific mutexes.
10. Returns to the caller.

1.8 Allocating an Object

An object may be allocated to one of the following classes:

- Identifier object
- User object (This is the user object which exists for each active user on the system.)
- Job object
- Process object
- Thread object

The identifier object class consists of all the identifiers in the rights database which have corresponding identifier allocation objects. Identifier allocation objects are created with the following service and exist in the system level container *mica$identifier_allocation*.

```
PROCEDURE exec$create_identifier_allocation (
    OUT identifier_allocation_id : exec$object_id;
    IN  object_item_list : exec$item_list;
    IN  identifier : INTEGER;
    ) RETURNS STATUS;
```

The active user ID class consists of the set of all user IDs currently active on the system. An object which is allocated to an active user ID is automatically deallocated when no users of that ID are currently on the system. For example, if an object is allocated to user KOLSEN, it is automatically deallocated when the last user with the ID KOLSEN is removed from the system.

The job class consists of the set of all threads within the job. When the job terminates, all objects allocated to the job class are automatically deallocated.

The process class consists of the set of threads within the specified process, and any thread within a descendant process. When the specified process terminates, all objects allocated to that process class are automatically deallocated.

The thread class consists of the thread that allocated the object. When the thread terminates, all objects allocated to the thread class are automatically deallocated.

The *e$allocate_object* argument that specifies the visibility of the object ID determines the allocation classes to which an object can be allocated. The rules for determining allocation class are as follows:

- If the ID is at the system level, the object may be allocated to any one of the five classes.
- If the ID is at the job level, the object may only be allocated to the job, process, or thread classes.
- If the ID is at the process level, the object may only be allocated to the process or thread classes.

1.8.1 Object Allocation Block

When an object is allocated the object header contains a pointer to the object allocation block. The object allocation block contains:

- A forward link to the next allocation block in this class.
- A backward link to the previous allocation block in this class.
- The allocation type (identifier, active user, job, process or thread).
- The allocation ID which identifies the allocation. For example, if the allocation is identifier, the ID of the identifier object is stored here.
- A pointer to the object header which refers to this allocation block.

Each allocation class has a listhead. The listhead for the thread allocation class is located in the tread object, the process in the process object, the job in the job object. The listhead for each active user is maintained in the "user" object structure. Job, process, thread and user objects are defined in Chapter 5, Process Structure. The listhead for an identifier object is located in the specific identifier object.

The listhead offset is stored in the OTD for the object type. This allows the object architecture to deallocate all object allocated to a particular object (like a thread) when that object is deleted.

There is a single mutex which guards access to all allocation class listheads. This mutex is known as the allocation mutex.

The *e$allocate_object* routine allocates the specified object to the specified allocation object. The declaration of this procedure is:

```
PROCEDURE e$allocate_object (
    IN object_id : exec$object_id;
    IN allocation_id  : exec$object_id;
    IN access_mode : integer;
    );
```

The *e$allocate_object* routine performs the following:

1. Translates the *allocation_id* with an access type of read to obtain the object class and other information.
2. Ensures that the calling thread is within the specified allocation class.
3. Ensures that the allocation class is compatible with the visibility of the specified object ID.

4. Translates the specified ID by calling *obj$reference_object_by_id*, with an access type of ALLOCATE.
5. Acquires an allocation mutex if the translation succeeds.

   This prevents other threads from attempting to allocate the same object simultaneously.
6. Checks to ensure that the object ID count for the object is 1.
7. Checks to ensure that the object is not currently allocated.
8. If all checks pass, *e$allocate_object* calls the OTD allocate routine for that object type, passing in the object body and the allocation type.
9. If the allocate routine does not return denial, *e$allocate_object* allocates an object allocation block, initializes the block, and links it into the appropriate allocation list.
10. Adds a pointer to the object allocation block to the object header and from the object header to the allocation block.
11. Releases the allocation mutex.
12. Dereferences the allocation object.
13. Dereferences the allocated object.
14. Returns status to the user.

\We need to be able to have a protected subsystem assume the allocation class as the thread whose behalf it is working \

1.9 Deallocating an Object

Allocated objects may be explicitly deallocated via a call to *exec$deallocate_object* or implicitly deallocated when an allocation class ceases to exist. For example, when a process terminates, any objects allocated to that process are implicitly deallocated.

The *e$deallocate_object* procedure deallocates the specified object. The declaration of this procedure is:

```
PROCEDURE e$deallocate_object (
    IN object_id : exec$object_id;
    ) RETURNS STATUS;
```

The *e$deallocate_object* routine performs the following:

1. Translates the specified object ID by calling *obj$reference_object_by_id* with an access type of DEALLOCATE.
2. If *obj$reference_object_by_id* succeeds and the object was allocated, then the thread is in the allocation class of the object.
3. Acquires the allocation mutex.
4. Checks to see if the object is currently allocated, if the object is not allocated, release the allocation mutex, call *obj$dereference_object* and return an error status of not allocated.
5. Calls the type-specific deallocate routine, passing in the address of the object body and the allocation type.
6. Clears the allocation pointer in the object's header.
7. Unlinks the object allocation block, and calls *obj$dereference_object*.
8. Releases the allocation mutex.
9. Returns the storage occupied by the object allocation block to the system.

10. Returns status to the user.

Implicit deallocation which occurs when an allocation class terminates. For example, upon thread termination, *e$deallocate_object* does the following:

1. Acquires the allocation mutex.
2. Removes the first entry from the terminating class's allocation list.
3. Uses the back pointer to the object header get the object type.
4. Calls the type-specific deallocate routine passing in the address of the object header and the allocation type.
5. Clears the allocation pointer in the object's header.
6. Unlinks the object allocation block.
7. Releases the allocation mutex.
8. Returns the storage occupied by the object allocation block to the system.
9. Repeats from step 1 until there are no more entries on the list.

1.10 Quotas and Objects

Objects are allocated from system paged and nonpaged pool, and as such are charged against the quota for a process or a job. The following rules indicate how and what is charged for object creation and reference. When an object is deleted, the same rules are used to return quota.

- Objects created in system level object containers are not charged against quotas. Objects at this level are permanent, and may outlive the life of the entity which created the object.
- Objects which are created at job or process level are charged to the level at which the object was created.
- If an object is permanent or was at one time permanent, (that is, marked as temporary) there are no charges for references. Note, objects of this type are in job or system containers.
- If an object was made temporary, a charge is made for the reference at the level of the reference. Since the creator of the object did not desire to create a permanent object, each referencer of the object is charged the total cost of the object. Note, the entity that initially created the object was charged for the object creation, and is not charged again when the object is made temporary.

\The exact methodology of charging and releasing quotas is still under discussion. One method is to add a field to the object header that indicates the charged amount for this object. This charged amount field would be used for creation, references, transfers, and deletion.\

\Chapter 5, Process Structure should describe a pair of procedures to charge and return quotas that allow the user to specify the amount, the type, and the entity to charge. \

1.11 Executive Support Functionality

This section describes the executive procedures which are invoked by object service routines for standard operations. Using a single set of executive procedures for common object operations enhances reliability and maintainability.

1.11.1 obj$reference_object_by_id

Given an object ID, type, mode and desired access, this procedure returns the address of the object body for the ID and increments the pointer count field for the object. An error is returned if the object ID is invalid.

```
PROCEDURE obj$reference_object_by_id (
    IN object_id : e$object_id;
    IN object_type : e$object_type;
    IN access_mode : k$processor_mode;
    IN desired_access : e$access_type;
    OUT object_body : POINTER anytype CONFORM;
    ) RETURNS STATUS;
```

1.11.2 obj$translate_object_name

Given an object container, object name, object type and access mode, this procedure returns the object ID corresponding to the name.

```
PROCEDURE obj$translate_object_name (
    IN container_id : e$object_id;
    IN name : string (*);
    IN object_type : e$object_type;
    IN access_mode : k$processor_mode;
    IN case_blind : boolean;
    OUT object_id : e$object_id;
    ) RETURNS STATUS;
```

1.11.3 obj$create_initialize_object

This routine is called with the number of bytes to allocate in addition to the object header. It allocates the storage from the appropriate pool, and initializes the object header.

```
PROCEDURE obj$create_initialize_object (
    IN otd_id : e$object_id;
    IN access_mode : k$processor_mode;
    IN acl : POINTER e$access_control_list;
    IN name : string (*);
    IN transfer : boolean;
    IN reference_inhibit : boolean;
    IN pool_type : e$pool_index;
    IN object_size : integer;
    OUT object_body : POINTER anytype CONFORM;
    ) RETURNS STATUS;
```

1.11.4 obj$insert_object_in_container

This routine accepts an object body pointer and a object container ID, and adds the object to the specified object container returning the new object ID.

```
PROCEDURE obj$insert_object_in_container (
    IN object_body : POINTER anytype CONFORM;
    IN access_mode : k$processor_mode;
    IN container_id : e$object_id;
    OUT object_id : e$object_id;
    ) RETURNS STATUS;
```

1.11.5 obj$insert_object_and_reference

This routine accepts an object body pointer and a object container ID, and adds the object to the specified object container returning the new object ID and pointer to the corresponding object body. The reference count on the object is increased by one.

This procedure is provided for object services which need to immediately reference an object after it has been created. It avoids race conditions with the object name colliding, the ID of the colliding object being returned, and before the object can be referenced, the object is deleted.

```
PROCEDURE obj$insert_object_and_reference (
    IN object_body : POINTER anytype CONFORM;
    IN access_mode : k$processor_mode;
    IN container_id : e$object_id;
    OUT object_id : e$object_id;
    OUT new_object_body : POINTER anytype CONFORM;
    ) RETURNS STATUS;
```

1.11.6 obj$remove_obj_from_container

This routine accepts an object ID and removes the object ID from its associated object container.

```
PROCEDURE obj$remove_obj_from_container (
    IN object_id : exec$object_id;
    IN access_mode :k$processor_mode;
    ) RETURNS STATUS;
```

1.11.7 obj$dereference_object

This routine decrements the pointer count of the specified object. If the resulting pointer count is zero, the object deletion routine for the specified object is invoked.

```
PROCEDURE obj$dereference_object (
    IN object_body : POINTER anytype CONFORM;
    );
```

1.11.8 obj$get_principal_object_id

This routine returns the principal object ID for a given object body. If the object has no principal ID, the invalid ID zero is returned.

```
PROCEDURE obj$get_principal_object_id (
    IN object_body : POINTER anytype CONFORM;
    ) RETURNS e$object_id;
```

1.11.9 obj$set_object_acl

This procedure updates the ACL field in the object header.

```
PROCEDURE obj$set_object_acl (
    IN object_body : pointer exec$object_body;
    IN acl : pointer exec$acl;
    ) RETURNS old_acl POINTER exec$acl;
```

1.12 Revision History, 31 AUG 1987

1. Removed ULTRIX fork and exec semantics.
2. Added Pillar record definitions.
3. Added e$insert_object_and_reference procedure.
4. Added allocation listhead offset to OTD.
5. Added quota rules.

1.13 Revision History, 04 MAY 1987

1. Added fork and transfer actions for reference IDs. The actions are essentially share.
2. Added identifier allocation objects.
3. Added *e$create_otd* service.

1.14 Revision History, 30 April, 1987

1. Changed names of container directories to *exec$level_container_directory*.
2. Changed *show_xxxx* routines to *get_xxx_information* routines.
3. Changed the Object type descriptor to be an object of type OTD.
4. Removed OTD flink from OTD body.
5. Changed all internal object routines which returned a pointer to an object header to return a pointer to the object body.
6. Added reference ID inhibit flag to object header.
7. Removed group owner and acl modification time from object header.
8. Added sequence number field to object header.
9. Added no show attribute to logical names.
10. Changed generic object creation procedure definition to reflect current thoughts on system services.
11. Changed semantics of when the OTD remove routine is called. It is now called after the object ID has been removed from the object container with no mutexes held.
12. Changed semantics of how a temporary object is deleted when it's object ID count is one so the race condition on deleting the principal ID is eliminated.
13. Added *e$set_object_acl* service.
14. Removed *exec$transfer_object* service.
15. changed the name of the following internal services to make their names more descriptive of their operations:
    1. *e$allocate_initialize_object* becomes *e$create_initialize_object*
    2. *e$deallocate_object* becomes *e$delete_object*
    3. *e$create_object_id* becomes *e$insert_object_into_container*
    4. *e$remove_object_id* becomes *e$remove_object_from_container*

5. *e$translate_id* becomes *e$reference_object_by_id*
6. *e$decrement_pointer_count* becomes *e$dereference_object*

1.15 Revision History, 20 March, 1987

1. Removed alias object.
2. Removed allocate object.
3. Added reference IDs.
4. Added allocation classes.
5. Added allocation mutex.
6. Added OTD object flink/blink and sequence number.
7. Added exec action to object header.
8. Added allocate procedure to OTD.
9. Added deallocate procedure to OTD.
10. Removed transfer procedure from OTD.
11. Added fork action of transfer.
12. Added ACL time to object header.
13. Added algorithms for the following:
    1. create object
    2. delete object
    3. remove object
    4. fork action
    5. make temporary
    6. mark temporary
    7. translate ID
    8. allocate
    9. deallocate
    10. translate name

1.16 Revision History, 28 January, 1987

1. Restructure and reorganize material.
2. Remove CIT and replace with container directory
3. Added additional allocation size field to OTD.
4. Added container pointer to object header.
5. Added object creation rules.
6. Added object deletion rules.

1.17 Revision History, 14 January, 1987

1. Eliminated concept of nested object containers. Object containers are in a flat name space.
2. Eliminated concept of support for general wait mechanism in objects. Wait is supported by using a kernel support object within the object.
3. Simplified naming and eliminate CINB, ENB, and other naming structures.
4. Added support for logical names.
5. Simplified locking rules on containers and objects.
6. Added rules for alias object creation.
7. Added allocate objects.
8. Added rules for temporary objects.
9. Added FORK dispatch element to SSVT for support of ULTRIX style fork services.
10. Changed security mechanism to ACL based and removed object types reserved for security.
11. Changed definition of reference count in object header.
12. Removed alias count and added share count to object header.
13. Removed container lists. This functionality may be obtained through logical names.
14. Added definitions of executive routines for internal object support.
15. Added definitions of object services for user object support.

The following system services will exist in the chapter on MICA system services. They have been included with the object chapter for the chapter review and have not been reviewed or finalized.

Please note that the names for various parameters may not match the current naming guidelines. These will be remedied in the future.

```
!----------------------------------------------------------------------
! Template CREATE_object and GET_object_INFO procedure definitions.
!----------------------------------------------------------------------

PROCEDURE exec$create_xxxxxx (
    OUT principal_id : exec$type_object_id;
    IN obj_independent_items : exec$type_item_list (*) CONFORM OPTIONAL;
    IN obj_dependent_items : exec$type_item_list (*) CONFORM OPTIONAL;
    ) RETURNS INTEGER; !STATUS;
    EXTERNAL;

PROCEDURE exec$get_xxxxxx_information (
    IN object_id : exec$type_object_id;
    IN items : exec$type_item_list (*) CONFORM;
    ) RETURNS INTEGER; !STATUS;
    EXTERNAL;


!----------------------------------------------------------------------
! General object service routines.
!----------------------------------------------------------------------

PROCEDURE exec$translate_object_name (
    IN object_name : STRING (*);
    IN object_type : LONGWORD;
    IN container_id : exec$type_object_id OPTIONAL;
    IN container_name : STRING (*) OPTIONAL;
    IN case_blind : BOOLEAN = FALSE;
    IN substitute : BOOLEAN = FALSE;
    OUT principal_id : exec$type_object_id;
    ) RETURNS INTEGER; !STATUS;
    EXTERNAL;

PROCEDURE exec$get_object_information (
    IN object_id : exec$type_object_id;
    IN object_id_items : exec$type_item_list (*) CONFORM OPTIONAL;
    IN object_items : exec$type_item_list (*) CONFORM OPTIONAL;
    ) RETURNS INTEGER; !STATUS;
    EXTERNAL;

PROCEDURE exec$allocate_object (
    IN object_id : exec$type_object_id;
    IN allocation_object_id : exec$type_object_id;
    ) RETURNS INTEGER; !STATUS;
    EXTERNAL;

PROCEDURE exec$get_allocated_object_id (
    IN allocation_object_id : exec$type_object_id;
    IN object_type : LONGWORD OPTIONAL;
    IN OUT context : QUADWORD;
    OUT object_id : exec$type_object_id;
    ) RETURNS INTEGER; !STATUS;
    EXTERNAL;
```

```
PROCEDURE exec$deallocate_object (
    IN object_id : exec$type_object_id;
    ) RETURNS INTEGER; !STATUS;
    EXTERNAL;

PROCEDURE exec$set_object_name (
    IN principal_id : exec$type_object_id;
    IN object_name : STRING (*);
    ) RETURNS INTEGER; !STATUS;
    EXTERNAL;

PROCEDURE exec$clear_object_name (
    IN principal_id : exec$type_object_id;
    ) RETURNS INTEGER; !STATUS;
    EXTERNAL;

PROCEDURE exec$delete_object_id (
    IN object_id : exec$type_object_id;
    ) RETURNS INTEGER; !STATUS;
    EXTERNAL;

PROCEDURE exec$create_reference_id (
    IN principal_id : exec$type_object_id;
    IN object_container_id : exec$type_object_id OPTIONAL;
    OUT reference_id : exec$type_object_id;
    ) RETURNS INTEGER; !STATUS;
    EXTERNAL;

PROCEDURE exec$transfer_obj_make_temp (
    IN principal_id : exec$type_object_id;
    IN object_container_id : exec$type_object_id;
    IN replace : BOOLEAN = FALSE;
    OUT reference_id : exec$type_object_id;
    ) RETURNS INTEGER; !STATUS;
    EXTERNAL;

PROCEDURE exec$make_object_temporary (
    IN object_id : exec$type_object_id;
    ) RETURNS INTEGER; !STATUS;
    EXTERNAL;



!----------------------------------------------------------------------
! Container directory object service routines.
!----------------------------------------------------------------------

PROCEDURE exec$get_contr_dir_information (
    IN container_directory_id : exec$type_object_id;
    IN items : exec$type_item_list (*) CONFORM;
    ) RETURNS INTEGER; !STATUS;
    EXTERNAL;

PROCEDURE exec$get_object_container_id (
    IN container_directory_id : exec$type_object_id;
    IN OUT context : QUADWORD;
    OUT object_container_id : exec$type_object_id;
    ) RETURNS INTEGER; !STATUS;
    EXTERNAL;
```

```
!----------------------------------------------------------------------
! Object container object service routines.
!----------------------------------------------------------------------

PROCEDURE exec$create_object_container (
    OUT principal_id : exec$type_object_id;
    IN obj_independent_items : exec$type_item_list (*) CONFORM OPTIONAL;
    ) RETURNS INTEGER; !STATUS;
    EXTERNAL;

PROCEDURE exec$get_obj_contr_information (
    IN object_container_id : exec$type_object_id;
    IN items : exec$type_item_list (*) CONFORM;
    ) RETURNS INTEGER; !STATUS;
    EXTERNAL;

PROCEDURE exec$get_object_id (
    IN object_container_id : exec$type_object_id;
    IN object_type : LONGWORD OPTIONAL;
    IN OUT context : QUADWORD;
    OUT object_id : exec$type_object_id;
    ) RETURNS INTEGER; !STATUS;
    EXTERNAL;




!----------------------------------------------------------------------
! Object type descriptor object service routine.
!----------------------------------------------------------------------

PROCEDURE exec$get_otd_information (
    IN object_type_desc_id : exec$type_object_id;
    IN items : exec$type_item_list (*) CONFORM;
    ) RETURNS INTEGER; !STATUS;
    EXTERNAL;




!----------------------------------------------------------------------
! Logical name object service routines.
!----------------------------------------------------------------------

PROCEDURE exec$create_logical_name (
    IN logical_name : STRING (*);
    IN container_id : exec$type_object_id OPTIONAL;
    IN container_name : STRING (*) OPTIONAL;
    IN create_if : BOOLEAN = FALSE;
    IN logical_name_items : exec$type_item_list (*) CONFORM;
    IN equivalence_name_items : exec$type_item_list (*) CONFORM;
    ) RETURNS INTEGER; !STATUS;
    EXTERNAL;

PROCEDURE exec$translate_logical_name (
    IN logical_name : STRING (*);
    IN container_id : exec$type_object_id OPTIONAL;
    IN container_name : STRING (*) OPTIONAL;
    IN case_blind : BOOLEAN = FALSE;
    IN logical_name_items : exec$type_item_list (*) CONFORM OPTIONAL;
    IN equivalence_name_items : exec$type_item_list (*) CONFORM OPTIONAL;
    ) RETURNS INTEGER; !STATUS;
    EXTERNAL;
```

```
PROCEDURE exec$delete_logical_name (
    IN logical_name : STRING (*) OPTIONAL;
    IN container_id : exec$type_object_id OPTIONAL;
    IN container_name : STRING (*) OPTIONAL;
    OUT deleted_container_id : exec$type_object_id OPTIONAL;
    ) RETURNS INTEGER; !STATUS;
    EXTERNAL;

PROCEDURE exec$get_logical_name (
    IN container_id : exec$type_object_id;
    IN OUT context : QUADWORD;
    OUT logical_name : STRING (*);
    ) RETURNS INTEGER; !STATUS;
    EXTERNAL;
```

PRODUCT PROJECT DESCRIPTION

Change History:

Revision No.	Created	Author
0.1	6/9/87	Robert Bismuth Based on 6/9/87 meeting with Benn Schreiber, David Ballenger and Lou Perazzoli.
0.2	6/12/87	Robert Bismuth Based on 6/11/87 meeting with David Ballenger, Benn Schreiber, Tom Miller, Chris Saether, Jeff East, and Lou Perazzoli.
0.3	6/17/87	Robert Bismuth Misc. typo corrections. Incorporating discussion with John Gilbert and the above list of engineers on 6/17/87.
0.4	7/13/87	Robert Bismuth Incorporate comments from review meetings of 6/19/87 and 6/24/87.
0.5	7/17/87	Robert Bismuth Include MICA discussion section.
0.6	9/10/87	Robert Bismuth Close DECnet, installation, and character cell terminal issues Update for 32 bit support Include recent changes in workgroup dependency Clarify installation goal for DB platform Clarify reliability goal for MICA Replace naem CRDK with QUARTZ Close dev. environment for CXO issue Modify DB platform RPC requirements
0.7	9/21/87	Robert Bismuth Incorporate Myles Connors' comments regarding AIA
0.8	11/3/87	Robert Bismuth Incorporate review comments from Benn Schreiber and Jeff East

TABLE OF CONTENTS

1 Introduction

This paper outlines two products which the OS Group is building. Briefly, these are:

1. Cheyenne:

   A high performance, highly available database machine built out of the QUARTZ database software and a MICA subset operating system running on ROCK systems. \Note that in this document the MICA subset is referred to as the database platform.\

2. Glacier:

   A high performance, tightly coupled compute server for DIGITAL's hardware/software base.

Both of products are described here with the assumption that the database platform forms the basis of the compute server.

Both these products are assumed to be built around hardware which implements the PRISM architecture and a base system derived from the current MICA design. This paper also briefly outlines some of the constraints on and design of MICA.

2 MICA Overview

2.1 Requirements

MICA has two basic requirements:

1. Immediate requirement: provide a common subset on top of which the Cheyenne and Glacier products may be built.
2. Long term requirement: provide a state-of-the-art, portable operating system for future DIGITAL computer systems in the mid 1990s. This path is to be evolutionary.

2.2 Goals

1. Meet the above two requirements.
2. Robustness and reliability. In particular MICA must deal with all hardware and software errors in a reliable and predictable fashion.
3. Quality: meet the expectations for a new operating system.
4. Schedule.
5. Performance: goals required for both short and long term requirements.
6. Ease of installation and management.
7. Eventual support for all PRISM implementations, both 32 and 64 bit systems. Immediate support for the MORAINE/ROCK processors and packaging.
8. Provide a flexible I/O software architecture which support all ROCK hardware configurations defined as products.
9. Provide a flexible system interface which allows MICA to be extended without modification or addition of component software executing in kernel mode.

\Note that device support and object support are considered exceptions to this.\

10. Provide a programming environment in which DIGITAL's Application Integration Architecture may be implemented as the user applications' interface to MICA.

    \This would then eventually give a standard applications interface to MICA which results in source level compatibility for applications between MICA and other DIGITAL operating systems supporting this standard interface.\

2.3 Non-goals

1. Provide tools, documentation and features for user modification of the MICA executive and/or kernel.
2. Constrained by any of DIGITAL's existing operating systems.

   \For example, provide direct system service compatibility with any of DIGITAL's existing operating system products.\

2.4 Description

MICA has been designed as an object oriented system with a rich set of services available from user mode (note that most of these services will eventually be accessed via the Application Integration Architecture library). It is designed to support both the 32 and 64 bit PRISM architectures and should require minimal effort to move from a 32-bit only implementation to a 32/64-bit implementation when 64-bit systems become available. Currently, key points in the design are:

1. Priority based pre-emptive schedule with provision for calss scheduling.
2. Flexible memory management system which supports all allowed PRISM memory management implementations.
3. Multiple threads of execution within a single address space (i.e. "light weight processes").
4. A layered I/O architecture for the support of physical devices, file systems, and concepts such as volume shadowing, volume striping, virtual terminals, etc.
5. A centralized ACL based security architecture for all objects.
6. Protected subsystems: user processes which act as servers, with amplified security profiles and /or privileges, on behalf of client processes, charging back resource usage to those clients.
7. DECNET Phase 5 support.
8. Workgroup support: a new distributed computing environment which provides file, record, security profile and batch/print queue sharing. \Note that this is initially just support of DFS.\
9. Implemented almost entirely in the PILLAR language which provides block structure, strong typing, structured condition handling and has been designed as a portable system implementation language.

The design of MICA is described in the MICA Working Design Document. This document is currently being developed, given the above requirements.

2.5 Dependencies

\Dependencies are listed mainly for the short term.\

2.5.1 Internal Dependencies

1. The PRISM SRM.
2. The PRISM emulators and their configurations.
3. The XMI Prism development system.
4. Suitable 32-bit hardware and configurations. \The MORAINE/ROCK hardware. Note we eventually need 64-bit configurations.\
5. The 32-bit PILLAR compiler and an eventual 64-bit PILLAR compiler.
6. Continued support and extensions for the 32-bit SIL language until the 32-bit PILLAR compiler is available.
7. A PRISM C compiler and runtime library.
8. A development strategy that allows transition from our current 32-bit PRISM emulator/SIL compiler environment to the 32-bit XMI-system/PILLAR compiler environment, and from there to the MORAINE/ROCK hardware.
9. A testing strategy.

2.5.2 Corporate Dependencies

1. The PRISM Calling Standard: immediately the 32-bit standard.
2. DECNET Phase 5 architecture: both the architecture and the phased development of it for other DIGITAL products.

   In addition to these development issues, MICA must be validated for DECNET support.
3. Corporate distributed security strategy, both for workgroup support and for its effect on MICA's security architecture.
4. The FILES-11 ODS-2 specification. \Note that we have an ECO proposed and previously accepted for this but are will require VMS to support this ECO prior to our FRS for either of the two products.\
5. The Applications Integration Architecture. \MICA will need a native implementation of this for software development both internally and externally. This implementation will be phased over various future versions of MICA as parts of AIA become defined.\
6. The corporate RPC. \Required for interface to protected subsystems.\

2.5.3 External Dependencies

1. RTL and debugger support from SDT.

2.6 Outstanding Issues

1. Testing:

   Current MICA project plans have not been specific regarding testing strategies. Given the goals of the initial products based on MICA, rigorous testing is a now even more important a requirement of the development plan.

2. PILLAR RTL:

   Since PILLAR has very little in the way of built-in functions (such as I/O), an RTL is necessary. This RTL is for internal use only (i.e. during development) and may eventually be replaced by the superset functionality of a native Applications Integration Architecture.

2.7 Resolved Issues

1. Relationship with ULTRIX:

   We have resolved that there is no direct ULTRIX compatibility designed into MICA. This does not preclude adding limited compatibility in the form of run time libraries at some later date.

2. Failure modes and effects:

   We have planned a document, following after our design, which details the failure modes and effects of our products.

2.8 Impact on Previous MICA Design

Given the current ordering of requirements for MICA is changing the previous design. Each of the two products defined below require only subsets of MICA. A fully functional, general purpose MICA is not required until such time as the corporation needs MICA as a product in its own right.

However, while Cheyenne and Glacier allow us to remove some functional component of MICA from the project, they both require extra functionality, specific to each product, be implemented by MICA.

This is covered below separately for each product.

Finally, MICA will need to support both 32 and 64-bit PRISM architecture machines with a minimal of development effort once 64-bit systems become available.

3 Database Platform Product: Cheyenne

3.1 Requirements

The only requirement of the database platform is that it provide the host environment required by the CXO QUARTZ project. In particular, this implies:

1. Possibly new operating system interfaces and facilities.
2. Specified performance goals, both I/O and operating system overhead.
3. Specified availability goals.
4. The end product must fit in with DIGITAL's OLTP offerings.

3.2 Goals

1. Quality, robustness and reliability.
2. Support of the QUARTZ database software with a clean interface between MICA and the QUARTZ software.
3. Operation of Cheyenne (i.e. platform and QUARTZ software) as a "black box", i.e. a "box" with a concisely defined and specified hardware and software interface for database transactions. Cheyenne must appear as a product, not a collection of products.

   \This does not imply that the platform would be built out of a single ROCK system. This "box" might in fact be composed of several ROCK systems connected by a high performance interconnect. This helps to insure availability goals and also an extensible system with a reasonable price/performance scale.\
4. "Online" software installation or upgrade: replacement or modification of component parts while the entire system is running with minimal service interruption.

   \The current vision is that system software upgrades will take place while the system is running: to use the upgrade, a ROCK will need to be rebooted.\
5. Performance as required for the QUARTZ product: capable of supporting up to 600 tps in a fully configured system.
6. A host environment that gives the reliability, availability, and fault tolerance required for the QUARTZ product. \Figures presented for QUARTZ are: 99.95 availability.\
7. Large amounts of disk storage: QUARTZ has the goal of simultaneous support of up to 2100 HDAs giving 300 Gbytes total storage: one third large disks, two thirds small disks, all configured as 150 Gbytes of shadowed storage.
8. Simplified system management, including remote system management.
9. The Cheyenne is easily expandable from a basic hardware configuration to the top end configuration which meets or exceeds the above performance, availability and storage capacity goals.
10. Eventual expansion to include a TP monitor.

3.3 Nongoals

1. Access or use without a client system.
2. Access to processes on the platform other than database system processes. \Note that database system processes also include processes required for system management.\
3. Use of the underlying system as a compute server.

3.4 Description

The primary design center of the database platform is QUARTZ. The platform supplies all the operating system services which the QUARTZ software requires.

The platform is a MICA-subset which must minimally be capable of:

1. Supporting free running server processes.
2. Supporting the security model required by QUARTZ.
3. Implementing the specific network transport and/or protocol required for communication with QUARTZ processes.
4. Supporting a diagnostic environment for hardware diagnostics.

In addition, QUARTZ requires that MICA support a common logging facility and high performance interprocess communications. Note that these facilities must function in a multi-ROCK Cheyenne configuration between the individual ROCK systems.

Minimally, in addition to the QUARTZ requirements, MICA will need to provide remote system management, and high speed disk/file access.

The current design model consists of a Cheyenne system composed of multiple ROCK systems interconnected by high speed communication links. Through MICA, QUARTZ would distribute its work load via these links and thus meet the performance, availability and connectivity goals. Communication from client systems to the Cheyenne system would have no application visible knowledge of the internal organization or topology of ROCK systems making up the configuration.

3.5 Major Dependencies

3.5.1 Internal Dependencies

1. MICA: the subset needed for this product, and all its dependencies.

3.5.2 External Dependencies

1. QUARTZ.
2. Front-end software for access to QUARTZ servers. \The QUARTZ software will implement data management and a protocol for data access via LAN. To sell this as a product, there clearly need to be front end packages which run on client systems and use the QUARTZ/MICA system. Currently no group in DIGITAL is developing such packages. It is our responsibility to make sure that plans are in place for this area so that DIGITAL has a complete product.\
3. The corporate RPC standard. \This is required by the system management model.\
4. DECnet Phase 5: the QUARTZ network interface.
5. DECnet performance over the CI to/from VMS client systems.

3.6 Outstanding Issues

1. System management:

   System management is carried out via a remote system management interface from a client system. The details of this and the role of the system console in this are in design. That QUARTZ will use the system management interface to activate utilities. The interface must be extendable in this respect.

2. Security:

   What is the QUARTZ security model and how does it relate to the corporate distributed security model?

3. Transaction processing platform:

   It has been claimed that the database server should be the basis of a future transaction processing (TP) platform. Since TP has particular requirements and constraints, we should investigate current TP models to ensure that our base system design does not prevent a platform from being built in the future.

   \There are examples in VMS today of original design decisions for the base system which later made it impossible or difficult to implement efficient TP systems on VAXes.\

3.7 Resolved Issues

1. Development environment for CXO:

   The development requirements for CXO have been identified and plans made to deliver the required environment in the right time frame.

2. High speed interconnect for use between ROCK systems in a single Cheyenne configuration:

   The CI has been selected as the interconnect device.

3. Files–11:

   QUARTZ will use Files–11 volumes for files. MICA shadow set support will also be used.

4. Backup and restore facilities:

   The QUARTZ group will provide backup/restore/rebuild facilities for database files.

5. Network support:

   DECnet will be the network software implemented for Cheyenne and used by QUARTZ.

6. Transport protocols:

   The only transport protocols used by Cheyenne are DECnet protocols and SCA.

7. Client systems:

   DECwest has no currently planned role in any client software or client systems.

8. IPC:

   High performance interprocess communication is essential to the QUARTZ design. We are currently designing this facility to meet their needs.

9. Fault tolerance:

   We are dealing with the requirements of fault tolerance by providing the capability of multiple ROCK systems within a Cheyenne configuration. ROCK systems have the following requirements:

   1. 100% data integrity.
   2. Every system failure is detected and the source identified. \"System" here includes hardware and software.\
   3. Software must provide failover, reconfiguration and graceful degradation.

3.8 Impact on Previous MICA Design

The following facilities and areas of MICA are not believed to be needed for the Cheyenne MICA-subset operating system:

1. All utilities (both VMS and ULTRIX), with the exception of:
   1. The linker and PILLAR compiler.
   2. Some SET/SHOW/MONITOR functions. \Perhaps just an RPC server for these on the database platform with the actual command utilities integrated on the client systems.\
   3. BACKUP. \Not for database recovery - for entire volume backup and for installation.\
   4. Some system management utilities.
2. Terminal driver (except console support).
3. DCL, shells or any command interpreter.
4. Job controller: no batch or print management, very little (if any) local job management because QUARTZ processes are managed by QUARTZ and not by users directly. The only other processes in a Cheyenne system would be system management processes which are also very limited and controlled.

   It is very likely that the Process Architecture should be reviewed in light of the QUARTZ model.
5. Local VMS or ULTRIX compatibility libraries as defined for MICA today.

6. Vector support.
7. SDT languages and a large part of the RTLs.
8. Workgroup/DFS support.
9. RMS ISAM support.

MICA is not a highly available system. It does meet certain goals of behavior in failure modes and it also guarantees data integrity. MICA must provide the basis to allow QUARTZ to be made highly available by using multiple MICA systems. The fundamental tool for this is a high speed inter-rock CI based IPC mechanism.

4 Compute Server Product: Glacier

4.1 Requirements

We are expecting more specific requirements to be generated by our Product Management Group.

However, one requirement currently identified and assumed in this document is that compute servers and clients must be members of the same Workgroup.

4.2 Goals

1. Quality, robustness and reliability.
2. Seamless integration with client systems:
    1. Application compatible programming interfaces between client operating systems and Glacier's MICA operating system.
    2. Provide transparent image activation on the compute server from client systems. \This should make Glacier essentially an extension of a client.\
3. Client systems are VAXes, in particular workstations. \There is some discussion of other future client systems, VAXmates for example.\
4. The client system interface to the compute server is designed for future support by multiple operating systems.
5. The programming interface for Glacier is DIGITAL's Application Integration Architecture.
6. Performance: both processor, memory access/addressing and I/O speed/capacity. \Note that no specific goals have been identified in this area.\
7. Glacier provides parallel and vector processing capabilities.
8. Provide the basis from which a full functionality MICA operating system will be implemented as part of our new-architecture long range goal.
9. Provide an easy to use, simple, and consistent user interface for system management.
10. Layer support for the Glacier entirely on top of a client operating system with no internal changes. \This may be in conflict with goal 2 and tradeoffs may occur.\

4.3 Nongoals

1. Access to or use of the compute server without a client system. \Note that we do not include a LAT Server as a potential client system.\
2. Full participation in local or wide area networks as a node distinct from any client nodes.
3. Implement MICA system services that are completely compatible with VMS or ULTRIX system services.
4. A DECwindow server distinct from any client system. \This is a non goal because DECwindow services are provided by client systems, not the compute server itself.\
5. Making MICA system services directly available to applications.

4.4 Description

The current model of Glacier is a system which derives most of its programming environment and some of the programming interface and from its client systems. Glacier itself has system service capabilities and also uses services in its clients to complete requests from application images running on the compute server. This model is described in:

```
docd$:[MICA.papers.rpc-compute-server]rpc-compute-server.paper
```

The associated files in that directory discuss specific aspects of the model. What follows is a brief description of this model.

Glacier is based on a PRISM/MICA system that is connected to its client systems by a high-speed LAN. The server is accessed by client systems and largely acts as a slave to those systems, becoming an extension of each client system. A process, created by a client system, in the compute server, is always associated with the job in the client system that created it. Such processes are called bound processes.

Each bound process can make use of two types of system procedures: those which use native MICA system system services and those which make RPC calls back to the client system. In this way, a bound process is given control over MICA and PRISM specific facilities (such as memory management, multiprocessing, scheduling, etc.) while having capabilities and context of its associated job/client system available (such as logical name context, command information, user information, etc.).

In order to meet the goal of application transportability between client and server, bound processes may only use services provided by DIGITAL's Application Integration Architecture. AIA is designed to provide a system-independent implementation of system services/facilities. These include most of the basic services an operating system normally directly provides, plus facilities like DECwindows. This implementation of the AIA is built on top of the two possible types of system procedures described above.

Key to the Glacier design is that activating an image on a compute server should be indistinguishable from activating an image on a client. Therefore, a bound process has the following properties:

1. Its maximum life span is that of the process which created it.
2. It has the same effective security profile as the job/process which created it. \The FRS product will use DECnet proxies for this.\

3. It is associated with a specific process running on the client system in the associated job. That process forms the bound process' RPC server for nonlocal system procedures.

Thus, from a client's point of view, Glacier is a tightly coupled extension of its computing resources. In the current model, no client is aware of any other client's use of the compute server: bound processes on the compute server are unaware of each other.

From Glacier's point of view, in addition to bound processes (which are essentially slaved to client jobs) there are also free-running processes. These are necessary to implement server processes for facilities such as system management and workgroups. Free-running processes have the following important characteristics:

1. All the native MICA system procedures are available to them but they have no remote client procedures available since they are not associated with any particular client.
2. While they may be started via a special mechanism from a client, they have a life which is not implicitly linked to any job on any client. \Note that this makes the management and failure detection of these processes nontrivial.\
3. They are not automatically tied to the security profile of any client job.
4. Specific privileges are required for a client to create them.

The only interface to Glacier is via a client system. Client systems access the compute server via the RPCs' transport and via workgroup support (i.e. DFS for FRS). The RPC transport is bi-directional: client systems use it to create processes and compute server processes use it to make calls to client systems. Glacier has no directly connected terminals (other than the system console), so that all terminal usage by compute server processes is via the DECwindows component of the AIA or via callback RMS.

File access for a compute server is either provided by the local file system to locally attached disks, or via DFS support to disks located on client systems. It is implicit in this model that the compute server is in the same workgroup as its clients. DFS will provide file access services for FRS and is eventually expected to provide both record services, and the complete workgroup distributed security environment.

A compute server may also make its local disks available to processes running on client systems via the workgroup.

DFS and RPC transports are the only networking requirements which explicitly need to be implemented on the compute server. Other LAN/WAN access by applications on the compute server is obtained via callbacks to client systems or via using the DECnet facilities provided by MICA.

4.5 Major Dependencies

4.5.1 Internal Dependencies

1. MICA: the subset needed for this product.
2. DFS architecture implementation.

4.5.2 Corporate Dependencies

1. A satisfactory RPC standard and the tools to use and implement that standard.

   \Satisfactory means: capable of being used in our model. This may mean greater involvement in the review/specification of the corporate RPC standard.\

2. DIGITAL's Application Integration Architecture.

   \This is currently the biggest unknown quantity. Its central role in our model implies our future participation in its design and specification. For Glacier, it is partially a native MICA issue and partially a distributed issue.

   The MICA group has the opportunity to help specify this architecture. However, the architecture's success depends on managment commitment within other development groups at DIGITAL. We must monitor this.\

3. Corporate distributed security strategy: critical for post FRS workgroup support.

   \This is now a corporate wide issue under the direction of Butler Lampson with the major authors from SRC.\

4. Corporate ULTRIX strategy. \This item has far ranging consequences since it is unclear what exactly the future strategy is. However, it is now clear that ULTRIX systems will be clients of Glacier, post FRS.\

5. DFS. As DFS V2 becomes a reality, we will need to track and implement it, together with any changes which come out of DRG review.

4.5.3 External Dependencies

1. Support of DFS by other client operating systems.
2. Distributed debugger support from SDT.
3. SDT compilers. \Not for development of the product, but for availability at FRS.\
4. The DFS product for post FRS workgroup distributed record and file access services.

4.6 Outstanding Issues

The following are the open issues surrounding our current model:

1. Division of system services between local and remote:

   Some analysis of this has been done, but little is understood about the relative efficiency versus implementation costs per service. This needs to be looked into, possibly via examining sample applications which might be ported to Glacier or by prototyping some services once the tools required to implement RPCs are in place.

2. Changes in client operating systems:

   As yet, no detailed analysis has been made of how to implement the requirements of this compute server model in any potential client operating systems. It is possible that compute server support can be layered on top of any client system, as desired. If not, any changes required in client operating systems must be identified and a plan agreed upon with other development groups for their implementation.

\DECwest owns the "total system" problem for this product.\

3. Quotas and accounting:

   Should quotas for a bound process originate from the client system or should they be defined on Glacier? Is accounting information only returned to the client system when a bound process exits? Is it captured on Glacier?

4. Application availability:

   Currently no applications running on DIGITAL supplied systems use the Application Integration Architecture. Given that this architecture is currently being design and is expected to take some time to be available to customers, it is unlikely that there will be many applications available at FRS of Glacier which use this architecture.

   Thus, to provide some initial attraction for the product, it may be necessary to provide support for some VMS or ULTRIX system services via the RPC techniques described above. To resolve this issue, applications typical of those which would be run on Glacier should be studied for their requirements in terms of host services.

5. Failure modes and behavior:

   Little in-depth study has been made of dealing with server and network failures. These should be completely characterized for the product and documented. The question of failover (thought to be too large a problem for FRS) should be at least characterized and future product goals identified.

6. RTL versus AIA:

   It is unclear exactly what AIA will include. Currently, AIA is known to include the SMG$ and parts of the LIB$ VMS libraries. It may also include the general PILLAR RTL.

   We must resolve what general RTL support is required. If AIA includes what currently is defined as our RTL, then implementation and definition will continue as a part of the MICA AIA project. If not, we must plan and produce the RTL.

4.7 Resolved Issues

1. The availability of general RPC support for user written applications:

   The FRS product will not make general RPC support available to user written applications. RPC support will be design with that goal in mind for some future release.

2. Transport protocols:

   The only protocols implemented for Glacier are DNA protocols and the DFS protocols.

3. Network support:

   DECnet will be implemented on Glacier.

4. Model for system management:

   The system management model has been selected and is in design.

5. Scaling:

   The issue of scaling has been largely posponed to a future release. For now, when a client may use multiple compute servers, the target server will be selected by a fairly simple mechanism (such as a logical name).

Support for application fan out or load balancing is independent of the basic compute server design and thus need not be included for FRS.

6. Character cell terminal support:

   Glacier will provide support for character cell terminals. This support will be minimal line I/O, unless requirements show a need for more functionality. The maximum amount of functionality which would be implemented is the SMG library.

7. Client operating systems:

   Both ULTRIX and VMS are client operating systems for this product. \Note that there may be only 1 at FRS.\

4.8 Impact on Previous MICA Design

The impact on MICA includes all points mentioned for the database platform. However, Glacier requires the following MICA facilities be implemented on the database platform subset operating system:

1. The Application Integration Architecture library. \Note that the DECwindow part would be RPC stubs.\
2. Workgroup support (note that this implies support of DFS).
3. Vector support.
4. SDT languages, RTLs, etc.
5. The C compiler and C-RTL.

5 Bibliography

A large part of the input for this document comes from the research papers located in:

docd:[MICA.papers...]

on the software development cluster.

Mica Working Design Document
Booting

Revision 0.2
05–NOV–1987

Preliminary Draft

Issued by:
Mark Lucovsky

digital™

TABLE OF CONTENTS

EXAMPLES

TABLES

Revision History

Date	Revision Number	Author	Summary of Changes
05–NOV–1987	0.2	Mark Lucovsky	Revised SCB contents. Modified description of bootstrap device drivers. Added brief section on system startup. Revised locating the boot device algorithm
20–OCT–1987	0.1	Mark Lucovsky	First Draft

CHAPTER 1

BOOTING

1.1 Introduction

This chapter discusses the bootstraping process for PRISM processors, how the Mica system uses this process to bootstrap itself, and finally, the provisions made that allow operating systems other than Mica to bootstrap themselves.

NOTE

This chapter assumes 32-bit PRISM processors.

The purpose of the bootstrap process is to define a process capable of handing over a cold machine to system software. On PRISM processors, this bootstrap process occurs in three phases.

1. Hardware Bootstrap
2. Primary Software Bootstrap
3. Secondary Software Bootstrap

The three phases of bootstrap are responsible for initializing the PRISM processors to a known and architecturally defined state, loading and passing control to an operating system independent primary bootstrap program, and finally loading and passing control to an operating system dependent secondary bootstrap program.

1.2 Hardware Bootstrap

The hardware bootstrap is defined by the PRISM System Reference Manual. The purpose of the hardware bootstrap is to:

- initialize each PRISM processor to an architecturally defined state
- initialize portions of system memory to an architecturally defined state
- load the PRISM primary software bootstrap program (PSB) into system memory, and pass control to it.

1.2.1 SRM Bootstrap Summary

This section summarizes the actions that occur during a cold bootstrap. The intent is to summarize the description of bootstraping as described by the PRISM System reference Manual.

The following steps occur in the PRISM bootstrap sequence.

1. Test memory for bootstraping
2. Build and initialize the *Restart Parameter Block* (RPB)
3. Load Epicode into the bootstrap master
4. Build an initial system page table
5. Load PSB
6. initialize the *internal processor registers* (IPRs) of each processor
7. Transfer control to PSB

1.2.1.1 Memory Testing

The hardware bootstrap is responsible for testing and initializing enough memory for epicode, the RPB, the initial page table, and PSB. The lowest good memory is used for this purpose.

1.2.1.2 Restart Parameter Block

The RPB is the fundamental communications mechanism between the hardware bootstrap and the primary and secondary software bootstraps. The hardware bootstrap process allocates the first good 64kb piece of system memory for the RPB.

NOTE

This seems to limit PRISM systems to about 180 processors based on the size of the system–wide RPB, and the size of the per–processor RPB's.

The RPB consists of a single per–system RPB, and a per–processor RPB for each processor in the PRISM system. Example 1–1 illustrates the layout of the RPB.

Example 1–1: Restart Parameter Block Layout

```
VALUE
    boot$k_max_boot_device_name = 16;
    boot$k_max_boot_file_name = 64;

TYPE

    boot$t_physical_address: large_integer;
    boot$t_processor_id: integer;
    boot$t_logical_block: large_integer;
    boot$t_boot_device_name: string(boot$k_max_boot_device_name);
    boot$t_boot_file_name: string(boot$k_max_boot_file_name);
    boot$t_network_boot: large_integer;
    boot$t_machine_register: integer;
    boot$t_scalar_registers: ARRAY[2..63] OF boot$t_machine_register;

    !
    ! Bootstrap Options Longword
    !
    boot$t_boot_options: RECORD
        bo_debug_boot: bit;                          ! Invoke debugger in PSB and secondary bootstrap
        bo_debug_sys: bit;                           ! Invoke debugger in system initialization
        bo_logical_block: bit;                       ! Use an alternate secondary bootstrap
        bo_offline: bit;                             ! Boot in the offline state
        bo_non_dasd: bit;                            ! Boot device is not direct access
        bo_hsc_valid: bit;                           ! Indicates that bo_hsc_number is valid
        bo_ci_valid: bit;                            ! Indicates that bo_ci_number is valid
        bo_single_processor: bit;                    ! Boot on single processor only
        bo_hsc_number: integer[0..255] SIZE(bit,8);  ! HSC node number for boot device
        bo_ci_number: integer[0..255] SIZE(bit,8);   ! CI number for boot device
        LAYOUT SIZE(quadword,*)
            bo_debug_boot           POSITION(bit,0);
            bo_debug_sys            POSITION(bit,1);
            bo_logical_block        POSITION(bit,2);
            bo_offline              POSITION(bit,3);
            bo_non_dasd             POSITION(bit,4);
            bo_hsc_valid            POSITION(bit,5);
            bo_ci_valid             POSITION(bit,6);
            bo_single_processor     POSITION(bit,7);
            bo_expansion: FILLER(bit,*);
            bo_hsc_number           POSITION(bit,16);
            bo_ci_number            POSITION(bit,24);
        END LAYOUT;
    END RECORD;

    !
    ! Hardware PCB
    !
    boot$t_hwpcb: RECORD
        hwpcb_ksp: boot$t_machine_register;     ! Kernel Stack Pointer
        hwpcb_usp: boot$t_machine_register;     ! User Stack Pointer
        hwpcb_asn: word;                        ! Address Space Number
        hwpcb_ast: word;                        ! ast information
        hwpcb_ptbr: longword;                   ! Page Table Base Register
        LAYOUT
            hwpcb_ksp;
            hwpcb_usp;
            hwpcb_asn;
            hwpcb_ast;
            hwpcb_ptbr;
        END LAYOUT;
    END RECORD;
```

Example 1–1 Cont'd. on next page

Example 1–1 (Cont.): Restart Parameter Block Layout

```
!
! Boot State Longword
!
boot$t_state_longword: RECORD
    sl_bip: bit;                                    ! Boot In Progress
    sl_rip: bit;                                    ! Restart In Progress
    sl_pss: bit;                                    ! Powerfail Sequence Started
    sl_psc: bit;                                    ! Powerfail Sequence Completed
    sl_stc: bit;                                    ! Self Test Complete
    sl_el: bit;                                     ! Epicode Loaded
    sl_pe: bit;                                     ! Processor Enabled
    sl_sr: bit;                                     ! Slave request ? Slave Processor Ready
    sl_cts: bit;                                    ! Control Transfered to System Software
    sl_le: bit;                                     ! Load Epicode request
    LAYOUT SIZE (longword,*)
        sl_bip;
        sl_rip;
        sl_pss;
        sl_psc;
        sl_stc;
        sl_el;
        sl_pe;
        sl_sr;
        sl_cts;
        sl_le;
    END LAYOUT;
END RECORD;

!
! Per-System Restart Parameter Block
!
boot$t_restart_parameter_block: RECORD
    rpb_physical_address: boot$t_physical_address;      ! Physical Address Of Rpb
    rpb_version_number: integer;
    rpb_number_of_processors: integer;                  ! Number of processor slots
    rpb_per_processor_address: boot$t_physical_address; ! Physical Address of per-processor rpb
    rpb_checksum_area: boot$t_physical_address;         ! Physical Address of Checksum area
    rpb_checksum: integer;                              ! Checksum Value
    rpb_page_size: integer;                             ! System Page Size in bytes
    rpb_asn_size: integer;                              ! ASN size zero or 16
    rpb_physical_address_bits: integer;                 ! number of bits in physical address
    rpb_bootstrap_master: boot$t_processor_id;          ! ID of bootstrap master processor
    rpb_epicode_pool_length: integer;                   ! Size of available epicode pool
    rpb_epicode_pool: boot$t_physical_address;          ! pointer to free epicode memory
    rpb_bootstrap_options: boot$t_boot_options;         ! Bootstrap options
    rpb_logical_block_boot: boot$t_logical_block;       ! Logical Block of Secondary Bootstrap
    rpb_system_device: boot$t_boot_device_name;         ! Name of System Device
    rpb_system_filename: boot$t_boot_file_name;         ! Name of Secondary Bootstrap File
    rpb_network_bootstrap: boot$t_network_boot;         ! Network Boot Descriptor
    LAYOUT
        rpb_physical_address;
        rpb_version_number;
        rpb_number_of_processors;
        rpb_per_processor_address;
        rpb_checksum_area;
        rpb_checksum;
        rpb_page_size;
        rpb_asn_size;
        rpb_physical_address_bits;
        rpb_bootstrap_master;
        rpb_epicode_pool_length;
        rpb_epicode_pool;
        rpb_bootstrap_options;
        rpb_logical_block_boot;
        rpb_system_device;
        rpb_system_filename;
        rpb_network_bootstrap;
    END LAYOUT;
END RECORD;
```

Example 1–1 Cont'd. on next page

Example 1–1 (Cont.): Restart Parameter Block Layout

```
!
! Per-Processor Restart Parameter Block
!
boot$t_per_processor_rpb: RECORD
    pprpb_state_longword: boot$t_state_longword;            ! Per Processor State Longword
    pprpb_epicode_length: integer;                          ! Length of Epicode
    pprpb_epicode_address: boot$t_physical_address;         ! Address of Processors Epicode
    pprpb_restart_scbb: boot$t_physical_address;            ! System Control Block for restart
    pprpb_restart_pcbb: boot$t_physical_address;            ! Privileged Context Block for restart
    pprpb_restart_ipie: boot$t_machine_register;            ! restart ipie
    pprpb_restart_sisr: boot$t_machine_register;            ! restart sisr
    pprpb_restart_icie: boot$t_machine_register;            ! restart icie
    pprpb_restart_prbr: boot$t_machine_register;            ! restart prbr
    pprpb_restart_scalar_registers: boot$t_scalar_registers; ! restart scalar register set
    pprpb_restart_pc: boot$t_machine_register;              ! restart pc
    pprpb_restart_ps: boot$t_machine_register;              ! restart ps
    pprpb_restart_vc: boot$t_machine_register;              ! restart vc
    pprpb_restart_vl: boot$t_machine_register;              ! restart vl
    pprpb_restart_vml: boot$t_machine_register;             ! restart vml
    pprpb_restart_vmh: boot$t_machine_register;             ! restart vmh
    pprpb_vector_save_area: boot$t_physical_address;        ! pointer to 1 page vector save area
    pprpb_hwpcb: boot$t_hwpcb;                              ! hardware privileged context block
    LAYOUT
        pprpb_state_longword;
        pprpb_epicode_length;
        pprpb_epicode_address;
        pprpb_restart_scbb;
        pprpb_restart_pcbb;
        pprpb_restart_ipie;
        pprpb_restart_sisr;
        pprpb_restart_icie;
        pprpb_restart_prbr;
        pprpb_restart_scalar_registers;
        pprpb_restart_pc;
        pprpb_restart_ps;
        pprpb_restart_vc;
        pprpb_restart_vl;
        pprpb_restart_vml;
        pprpb_restart_vmh;
        pprpb_vector_save_area;
        pprpb_hwpcb;
    END LAYOUT;
END RECORD;
```

The hardware bootstrap is responsible for initializing the RPB. This includes the per–system RPB, and one per–processor RPB for each processor in the system.

The following fields in the per–system RPB are initialized by the hardware bootstrap.

- rpb_physical_address;
- rpb_version_number;
- rpb_number_of_processors;
- rpb_per_processor_address;
- rpb_checksum_area;
- rpb_checksum;
- rpb_page_size;
- rpb_asn_size;
- rpb_physical_address_bits;

All other fields, including the per–processor RPB's for all processors in the system are initialized to *zero()*.

NOTE

The hardware bootstrap should also initialize *rpb_bootstrap_master*.

The state longwords for each processor in the system should have the *sl_bip,sl_stc* fields set. The state longword for the bootstrap master should have its *sl_pe* field set.

1.2.1.3 Epicode Loading

Epicode is loaded into the bootstrap master processor. The length of the epicode, and the base of the epicode are loaded into the *pprpb_epicode_length*, and *pprpb_epicode_address* fields of the per–processor RPB indexed by *rpb_bootstrap_master*.

NOTE

After epicode is loaded, the state longword of the bootstrap master should have its *sl_el* field set.

1.2.1.4 Initial Page Tables

Initial page tables are created to provide a virtual memory environment for the primary and secondary software bootstrap. The initial page tables map four regions of memory.

- The page table. 8kb from x"ffbfe000" to x"ffbfffff".
- Primary software bootstrap memory. 256kb from x"ffc00000" to x"ffc3ffff".
- Boot device I/O memory. 64kb from x"ffc40000" to x"ffc4ffff".
- RPB's. 64kb from x"ffc50000" to x"ffc5ffff".

The pages all have kernel read/write, user no access with all fault bits **for, fow, foe** cleared.

NOTE

There may be invalid pages in the space from x"ffc40000" to x"ffc5ffff" depending on system configuration.

The hardware bootstrap should create a descriptor of allocated physical memory.

1.2.1.5 Loading Primary Software Bootstrap

The hardware bootstrap loads the primary software bootstrap program (PSB) into the 256kb of memory mapped at x"ffc00000".

1.2.1.6 IPR Initialization

The internal processor registers for the bootstrap master processor are loaded.

NOTE

The PCBB is made to point to the *pprbp_hwpcb* field of the RPB for bootstrap master processor. I am assuming that the contents of this record is still *zero()*.

1.2.1.7 Transfer Control to PSB

The last phase of the hardware bootstrap is the transfer of control the system software contained in PSB. After the transfer, the bootstrap master is executing PSB code and all *bootstrap slaves* (non–bootstrap master processors) are not enabled or executing system code. To accomplish the transfer, the following occurs:

1. The kernel stack pointer of the bootstrap master is set to x"ffc40000".
2. The following fields of the per–system RPB are initialized
   - rpb_bootstrap_options;
   - rpb_logical_block_boot;
   - rpb_system_device;
   - rpb_system_filename;
   - rpb_network_bootstrap;
3. The **pc** of the bootstrap master is loaded with the value x"ffc00000", and the bootstrap master is started.

NOTE

The SRM indicates that the *pprpb_hwpcb* in the per–processor RPB of the bootstrap master is initialized and active. I dont believe it is initialized at this point in time, but the pcbb register does point to it.

The state longword for the bootstrap master should have its *sl_cts* field set.

1.3 Primary Software Bootstrap

The primary software bootstrap is implemented as PSB. It is intended to be a relatively operating system independent piece of software. It will however contain some **ODS–II** specific file system primitives.

The primary software bootstrap is responsible for:

- Initializing a System Control Block (SCB) for the bootstrap master processor.
- Determining system type and performing system specific initialization.
- Creating an allocated physical memory descriptor.
- Sizing and testing available memory.
- Determining the bootstrap device to be used by the secondary software bootstrap.

NOTE

In the case of network and tape bootstraping, the boot device used to load the secondary software bootstrap may be different than the boot device to be used by the secondary software bootstrap.

- Locate and load the secondary bootstrap program.
- Transfer control to the secondary bootstrap program.

1.3.1 System Control Block Initialization

A system control block (SCB) is created for the bootstrap master processor. All vector locations contain a pointer to vector specific code. These entry points save minimal machine in the bootstrap masters RPB, and then call *boot$fatal_exception()* passing the vector number for that exception.

Once *boot$fatal_exception()* is entered, a "last gasp message" is printed on the console, and then a **HALT** instruction is executed.

The exception to this is the breakpoint exception vector at SCB offset x"70". The breakpoint exception vector points to the entry point of the *elevated IPL kernel debugger*.

The physical address of the SCB is then stored into the **SCBB** register. If the *bo_debug_boot* field is set in the *rpb_bootstrap_options* field of the per–system RPB, a **BPT** instruction is executed and the elevated IPL kernel debugger is entered.

1.3.2 System Initialization

PSB is responsible for determining the type of system that it is part of. Once the system type has been determined, system specific initialization is performed. The system type is determined by reading the contents of the **SID** register.

NOTE

It is assumed here that there is no per–processor processor–type specific initialization that needs to be done.

1.3.3 Allocated Memory Descriptor

PSB is responsible for determining what portions of physical memory have been allocated by the hardware bootstrap process, and accounting for the physical memory that it allocates and passes to the secondary software bootstrap. The hardware bootstrap always allocates physical memory from lowest good memory to higher addresses. PSB allocates physical memory from highest good memory towards lower addresses. PSB does its allocation in this way to try to leave the lower 1Gb of physical memory free for system IO purposes.

The tracking of allocated physical memory is maintained in the data structure described in Example 1–2.

Example 1–2: Physical Memory Allocation Descriptor

```
!
! Physical Memory Allocation Descriptor
!
boot$t_memory_allocation_desc: RECORD
    mad_base_page: integer;              ! Physical page number of lowest good memory
    mad_first_free: integer;             ! Physical page number of first free page
    mad_last_free: integer;              ! Physical page number of last free page
    mad_top_page: integer;               ! Physical page number of highest good memory
END RECORD;
```

The only fields of the allocated memory descriptor that are initialized at this time are the *mad_base_page*, and *mad_first_free*. The rest of the structure can not be initialized until memory is sized and tested.

NOTE

Information used to determine the correct contents of *mad_base_page*, and *mad_first_free* is passed to PSB by the hardware bootstrap TBD how.

The above data structure assumes that memory appears physically contiguous. Will this be the case, or do we really need a bit map ?

1.3.4 Sizing and Testing Memory

PSB is responsible for sizing and testing all of system memory. Assuming the existence of the system configuration table pointed to by the RPB, memory sizing is accomplished by finding all modules that contain memory and adding their size (in pages) to the total memory size. Once the total memory size is determined, the highest page number is calculated and is placed in both the *mad_last_free*, and *mad_top_page* fields of the allocated memory descriptor.

For each module that contains memory, and whose self tests have passed and include memory diagnostics memory, memory tests are not performed in software. For all other enabled modules that contain memory, a simple write/read memory test is performed on the memory contained in that module. In order to test memory, the memory must first be mapped into and then tested using virtual addresses to access the memory. The address space from x"00000000" to x"007fffff" is reserved for this purpose.

1.3.5 Locating The Boot Device

The hardware bootstrap is responsible for providing PSB with enough information to locate the boot device that will be used to load the secondary software bootstrap. Most of this information comes from the user that entered the boot command from the system console. The information needed to locate the boot device is passed from the hardware bootstrap to PSB through fields in the RPB. Table 1–1 illustrates the RPB information that is used to locate the boot device, and how it is used.

Table 1–1: Boot Device Location Information

RPB Field	Usage
rpb_bootstrap_options.bo_hsc_valid	If set, the boot device is located on HSC node *rpb_bootstrap_options.bo_hsc_number*
rpb_bootstrap_options.bo_ci_valid	If set, the boot device is located on CI number *rpb_bootstrap_options.bo_ci_number*
rpb_bootstrap_options.bo_hsc_number	Contains the HSC node number of the boot device
rpb_bootstrap_options.bo_ci_number	Contains the CI number of the boot device
rpb_system_device	If non-null, contains the device name of the boot device; otherwise, bootstrap fails

Since the information stored in the RPB does not fully specify the location of the boot device, PSB performs an implementation dependent limited scan of IO space to determine the exact location of the boot device. Example 1–3 illustrates the algorithm used during the scan for the Rock implementation.

Once the boot device is located, a device driver capable of reading logical blocks from the boot device is initialized.

1.3.5.1 Bootstrap Device Drivers

Bootstrap device drivers conform to a standard interface supporting reads and writes of 512 byte logical blocks. The drivers are standalone, elevated IPL device drivers similar in nature to the VAX/VMS bootstrap device drivers.

Device drivers are described by a *device driver interface record*. Example 1–4 illustrates the format of a device driver interface record.

Example 1–3: Boot Device Scan Algorithm for Rock Systems

```
!
! Determine Device Name
!
If rpb_system_device == "" then boot_failure();

!
! Determine Boot Device Class
!
If device == "DU??" then bootclass = disk_controller
If device == "MU??" then bootclass = tape_controller

!
! Locate all CI adapters and bootclass Controllers
!
foreach XMI bus
    record and init each CI adapter and each bootclass controller
end;

If rpb_bootstrap_options.bo_ci_valid == false AND rpb_bootstrap_options.bo_hsc_valid == false then
    foreach bootclass controller found in XMI bus scan
        If controller == device then
            we have found it
            return
        end;
    end;
end;


!
! Determine Range of CI's to Scan
!
If rpb_bootstrap_options.bo_ci_valid then
    min_ci = rpb_bootstrap_options.bo_ci_number;
    max_ci = rpb_bootstrap_options.bo_ci_number;
else
    min_ci = 0;
    max_ci = 255;
end;

!
! Determine Range of HSC's to Scan
!
If rpb_bootstrap_options.bo_hsc_valid then
    min_hsc = rpb_bootstrap_options.bo_hsc_number;
    max_hsc = rpb_bootstrap_options.bo_hsc_number;
else
    min_hsc = 0;
    max_hsc = 255;
end;

!
! Scan CI's for HSC's that contain device
!
for ci = min_ci to max_ci
    for hsc = min_hsc to max_hsc
        determine if ci,hsc contains device
        if device is found we are done
    end
end

If device is not found we failed
```

From Example 1–4, it is obvious that bootstrap device drivers must provide a write logical block interface. This interface is not used during the bootstrap process, but is there to provide crash dump support in running Mica systems.

Once the boot device driver interface is created, and its address stored in *boot$g_boot_device_driver*, it performs the following:

- Optionally create and initialize a *ram disk*.
- Create a logical block oriented channel to the *primary bootstrap device*.
- Optionally create a logical block oriented channel to the *alternate bootstrap device*.

Example 1–4: Boot Device Driver Interface

```
TYPE
    !
    ! Read Logical Block From Primary Boot Device
    !
    boot$t_read_logical_block: PROCEDURE  (
                                          IN logical_block: boot$t_logical_block;
                                          IN buffer_address: POINTER anytype CONFORM;
                                          ) RETURNS boolean;
    !
    ! Write Logical Block From Primary Boot Device
    !
    boot$t_write_logical_block: PROCEDURE  (
                                          IN logical_block: boot$t_logical_block;
                                          IN buffer_address: POINTER anytype CONFORM;
                                          ) RETURNS boolean;

    boot$t_device_scratch_pad: quadword_data(256);

    !
    ! Descriptor used by system software to determine the boot device
    !
    boot$t_device_descriptor: RECORD
        dd_csr_physical_address: quadword;                    ! Physical Address of Device CSR
        dd_device_flags: quadword;                            ! Device Specific Flags
    END RECORD;

    !
    ! Boot Device Driver Interface
    !
    boot$t_device_driver_interface: RECORD
        ddi_primary_block_read: boot$t_read_logical_block;      ! Logical block read for primary bootstrap
        ddi_alternate_block_read: boot$t_read_logical_block;    ! Logical block read for alternate/or nil
        ddi_primary_block_write: boot$t_write_logical_block;    ! Logical block write for primary bootstrap
        ddi_alternate_block_write: boot$t_write_logical_block;  ! Logical block write for alternate/or nil
        ddi_primary_scratch: boot$t_device_scratch_pad;         ! primary device driver scratchpad
        ddi_alternate_scratch: boot$t_device_scratch_pad;       ! alternate device driver scratchpad
        ddi_primary_device_descriptor: boot$t_device_descriptor;! primary device descriptor
        ddi_alternate_device_descriptor: boot$t_device_descriptor;! alternate device descriptor
        ddi_enabled: boolean;                                   ! true when device driver enabled
    END RECORD;

VARIABLE
    !
    ! Pointer to the boot device driver
    !
    boot$g_boot_device_driver: POINTER boot$t_device_driver_interface;
```

1.3.5.2 Non-Dasd Boot Devices

If the *bo_non_dasd* field in the *rpb_bootstrap_options* field of the RPB is set, and if the boot device driver senses that it is not capable of direct access, the device driver creates and initializes a ram disk device driver. The boot device driver then transfers an operating system specific file system block–by–block into the newly created ram disk. At the end of the transfer, the ram disk device driver becomes the boot device driver by placing the address of its device driver interface record into *boot$g_boot_device_driver*. The 32MB of address space from x"00800000" to x"027fffff" is reserved for use by the ram disk device driver.

1.3.5.3 Primary Bootstrap Device

The primary bootstrap device is accessed by the value of *boot$g_boot_device_driver^.ddi_primary_block_read*, and *boot$g_boot_device_driver^.ddi_primary_scratch*. This is the interface that is used to load the secondary bootstrap program, and one of the interfaces available for use by the secondary bootstrap program to load the rest of its operating system.

1.3.5.4 Alternate Bootstrap Device

The Mica system supports the notion of an alternate bootstrap device. If PSB detects that the *bo_logical_block* and *bo_non_dasd* fields in the *rpb_bootstrap_options* field of the RPB are not set, then PSB locates the alternate bootstrap device. PSB assumes that the primary bootstrap device contains an **ODS–II** file system. PSB reads the file *[sys$kernel]mica$boot_alternate.txt*. The file contains a device name which is the name of a device that is in the same "controller class", and is physically related to the boot device driver found in Example 1–3. If no alternate boot device can be located, then *boot$g_boot_device_driver^.ddi_alternate_block_read* is set to *zero()*; otherwise, it is assigned to the logical block read routine for the alternate device driver.

1.3.6 Locate and Load Secondary Bootstrap

After the primary boot device is set up, PSB must locate and load the secondary bootstrap. The secondary bootstrap program exists in two forms.

- Logical Block Format
- Native File System Format

1.3.6.1 Logical Block Format

Logical block format is indicated by the *bo_logical_block* field in the *rpb_bootstrap_options* field of the RPB being set. Logical block format is used when the file system available through the *ddi_primary_block_read* interface is not **ODS–II**, or when a simple 512 program is to be loaded. This is the secondary bootstrap format that will be used by Ultrix.

NOTE

Logical block format booting is limited to a single logical block due to restrictions in the SRM.

The internal layout of logical block format secondary bootstrap program resembles a relocated memory image. The image is loaded into virtual address x"00000000" by calling *ddi_primary_block_read*, and passing it the logical block number stored at *rpb_logical_block_boot* in the RPB, and virtual address x"0". Of course memory to back virtual address x"0" for *ddi_logical_block_size* is first allocated.

1.3.6.2 Native File System Format

Native file system format is indicated by the *bo_logical_block* field in the *rpb_bootstrap_options* field of the RPB being clear. Native file system format assumes that the file system available through *ddi_primary_block_read* is an **ODS–II** file system. The secondary bootstrap program is located by searching for a file whose name matches the *rpb_system_filename* field in the RPB. If this field is null, the file name is defaulted to *[sys$kernel]mica$sysboot.exe*.

The internal layout of the native file system format is a ~~PRISM~~ Mica system image file as described in Chapter 29, Linker. Once the file is located it is loaded into the virtual address space specified by its image base address.

NOTE

There are portions of the virtual address space that are in use by PSB at this time. In order to avoid a collision, secondary bootstraps should not be based in the following ranges.

- **x"00800000" to x"027fffff"—Used by PSB's ram disk**
- **x"ffbfe000" to x"ffc5ffff"—Used by PSB, RPB's ...**

PSB does not support images that require fixups.

1.3.7 Transferring Control to Secondary Bootstrap

After loading the secondary bootstrap PSB passes control to it. A single parameter is passed to the secondary bootstrap in register 14. The contents of register 14 is the address of the secondary bootstrap parameter block. Example 1–5 illustrates the layout of the secondary bootstrap parameter block.

Example 1–5: Secondary Bootstrap Parameter Block

```
!
! Secondary Bootstrap Parameter Block
!
boot$t_secondary_bootstrap_pb: RECORD
    sbp_memory_usage: POINTER boot$t_memory_allocation_desc;    ! PSB's memory allocation descriptor
    sbp_boot_device: POINTER boot$t_device_driver_interface;    ! the value of boot$g_boot_device_driver
END RECORD;
```

Before transfering control to the secondary software bootstrap, PSB examines the *bo_debug_boot* field in the *rpb_bootstrap_options* field of the RPB. If it is set, then PSB issues a **BPT** instruction which transfers control to the elevated IPL kernel debugger.

At the time PSB transfers control to the secondary software bootstrap, the system is in the following state:

- The bootstrap master processor is in control.
- The bootstrap master processor is executing within the virtual address space described by the page table stored at x"ffbfe000".
- The bootstrap master processor is executing at IPL 7.
- The bootstrap master processor is executing of off an SCB initialized such that only exception vector that will not cause a call to *boot$fatal_exception()* when vectored through is the breakpoint exception vector~~at offset x"70"~~.
- The bootstrap device to be used by the secondary software bootstrap is enabled, and contains a vector to the devices primary block read routine. The alternate block read routine is either **NIL**, or a vector to the alternate block read routine.
- The contents of the *boot$t_memory_allocation_desc* pointed to by the secondary bootstrap parameter block accurately represents the state of available and allocated system physical memory.
- System virtual address space from x"ffbfe000" to x"ffc5ffff" is reserved for the boot device driver interface, the per–system and per–processor RPB's, and the system page table. If this address space needs to be reclaimed, the secondary bootstrap must make its own provisions for a bootstrap device driver, relocate its kernel stack out of the region (this includes possibly changing the value of *hwpcb_ksp* in the bootstrap master processors *pprpb_hwpcb* field).
- If the *bo_non_dasd* field in the *rpb_bootstrap_options* field of the RPB is set, then the virtual address space from x"00800000" to x"027fffff" is potentially in use by a ram disk. The secondary software bootstrap should not attempt to reclaim this space until it has created its own ram disk device driver and copied the contents by reading the ram disk block by block and then writing the blocks to its own disk.

1.4 Secondary Software Bootstrap

The secondary software bootstrap program is an operating system specific bootstrap. The secondary software bootstrap program has different responsibilities for different operating systems /environments. There are currently two forms of secondary software bootstraps.

1. Ultrix logical block bootstrap
2. Mica secondary software bootstrap

1.4.1 Ultrix Secondary Bootstrap

In the case of Ultrix booting, the secondary bootstrap consists of a single 512 byte boot block that is loaded and transfered to. Once active, the program will load a "real" secondary software bootstrap which is intelligent enough to understand the Ultrix file system. The "real" secondary software bootstrap is responsible for loading, and transfering control to Ultrix in an Ultrix specific manner.

1.4.2 Mica Secondary Bootstrap

The secondary software bootstrap (SSB) for the Mica system is implemented in *[sys$kernel]mica$sysboot.exe*. It is responsible for loading the images that make up the Mica operating system. Section 1.5 describes the operation of SSB in greater detail.

1.5 SSB for the Mica System

SSB for the Mica system is implemented in *[sys$kernel]mica$sysboot.exe*. It is responsible for the following loading the *system sharable images* that make up the Mica operating system, and transfering control to the loaded system.

1.5.1 Loading Mica

Mica is loaded by SSB as a set of system sharable images. The loading of the Mica system occurs in the following stages:

- Locating the components of the Mica system.
- Loading and partially activating each of the images.
- Linking all of the images together.

1.5.1.1 Locating the Mica Images

SSB assumes that the file systems available on the boot device specified by the *spb_boot_device* field of the secondary bootstrap parameter block argument is an **ODS-II** file system. SSB applys a simple search list to the boot device when trying to locate files on the device. The search algorithm simply searches for a file over the primary block block read interface, and if the file can not be found it searches over the alternate block read interface.

SSB locates the Mica images by reading the *[sys$kernel]mica$components.dat* file from the boot device. The contents of this file is a list of directory qualified filenames that specify the system sharable images that make up the Mica operating system. If this file can not be found, then the Mica boot fails.

1.5.1.2 Loading the Images

Once the list of images has been located, each image is partially activated into system memory. Partial activation involves the following:

- Determining an appropriate base for the image.
- Allocating memory for the image.
- Loading the code, data, and transfer vector sections from the image file into system memory.
- Processing the images local relocation table.
- Recording information about the loaded image in a *loaded image description table* maintained by SSB.

- Loading the images external relocation table and immediate activation tables and linking them to the images entry in the loaded image description table.

1.5.1.3 ~~Linking the~~ Images

The final stage of loading the Mica operating system involves ~~linking~~ all of the images described in the loaded image description table together. This is a very simple operation which basically involves processing the external relocation table for each of the loaded images.

1.5.2 Transfering Control to Mica

After the Mica system is fully loaded, control is passed to it. The entry point of Mica is determined from the loaded image description table, and the external relocation table of SSB. SSB has a single unresolved external reference to the system initialization entry point named *mica$system_initialize()*. SSB examines its external relocation table, and resolves its external reference with an address contained in one of the previously loaded images. Once this linkage occurs, control is transfered to Mica at *mica$system_initialize()*. The *mica$system_initialize()* entry point is passed the loaded image description table, and the original value of the secondary bootstrap parameter block.

Once the Mica executive has initialized itself it creates an initial user mode process under the *SYSTEM* username. The image that is started is located on the system disk in *[sys$kernel]mica$startup.exe*. This process is responsible for all "System Startup".

MICA Working Design Document
Type, Record and Name Appendix Overview

08–OCT–1987

Issued by:
Mark Lucovsky

digital™

1 Overview

This paper describes the components of Appendix A, Types, Records, and Names for the working design document of the Mica system. The protocol for submitting code examples to the author of the appendix is also discussed.

The Type, Record and Name Appendix includes all data types, declarations, and constants that appear in the working design document and that are intended to be exported within the Mica system and out of the Mica system. While the author of the appendix is responsible for assembling its contents, it is assumed that the individual authors of the chapters will provide significant portions of the appendix.

1.1 Goals/Requirements

The goal of Appendix A, Types, Records, and Names is to provide a reference for all code examples that appear in the working design document. The appendix will compile without error under the current SIL compiler, and will conform to the coding and naming standards outlined in Chapter 2, Naming and Coding Standards.

1.2 Appendix Contents

Each chapter author is responsible for submitting the code examples present in their chapter to the author of Appendix A, Types, Records, and Names. The code examples required in the appendix are:

- Data type declarations
- Variable declarations
- Constant declarations
- Facility Names

Procedure declarations and sample program fragments need not be submitted to the appendix.

The submitted code must contain correct SIL types, values, and variables. The code must be contained in a module that should compile without error. If the code does not compile, the submitter must be able to provide the author of Appendix A, Types, Records, and Names the reason for the module not compiling.

If a module submitted to Appendix A, Types, Records, and Names does not compile, it will most likely be because it depends on declarations that are not explicitly imported or defined within the module. It is the responsibility of the author of the submitted module to understand the reason for the module not compiling. The author of the module and the author of Appendix A, Types, Records, and Names will then determine who is responsible for providing a declaration that will allow the module to compile. The module shown in Example 1 does not compile. After reviewing the module, it is clear that the declaration for *e$type_time_value* is missing, and that the base systems group is responsible for providing this declaration.

The author of Appendix A, Types, Records, and Names will coordinate all naming collisions and duplicate names for the same data type.

Example 1: Process Accounting Summary

```
MODULE e$accounting_def;

TYPE
    e$fpu_class: ( e$class_a, e$class_b, e$class_c );
    e$type_accounting_counter: large_integer;

    !
    ! Process Accounting Summary
    !
    ! The final accounting record contains this information in TLV format
    ! in addition to fields identifying the process, image name, user ...
    !
    e$type_accounting_summary: RECORD
        acct_cpu_cycles: e$type_accounting_counter;              ! Number of cycles used by the process
        acct_total_page_faults: e$type_accounting_counter;       ! Total number of page faults
        acct_hard_page_faults: e$type_accounting_counter;        ! Number of page faults for non resident pages
        acct_soft_page_faults: e$type_accounting_counter;        ! Number of page faults fixed from reclaim list
        acct_dzro_page_faults: e$type_accounting_counter;        ! Number of demand zero page faults
        acct_com_page_faults: e$type_accounting_counter;         ! Number of copy on modify page faults
        acct_peak_virtual_memory: e$type_accounting_counter;     ! Peak virtual memory size
        acct_peak_working_set_size: e$type_accounting_counter;   ! Peak working set size
        acct_start_time: e$type_time_value;                      ! Start time of process
        acct_end_time: e$type_time_value;                        ! End time of process
        acct_page_file_usage: e$type_accounting_counter;         ! Peak page file usage
        acct_paged_pool_usage: e$type_accounting_counter;        ! Peak paged pool usage
        acct_non_paged_pool_usage: e$type_accounting_counter;    ! Peak non paged pool usage
        !
        ! IO Accounting
        ! Request IO's are counted once.
        ! Each FPU that passes on an IRP (execute_io's) must also record the transfer
        ! by incrementing the counter for its class of FPU
        !
        acct_request_io_count: e$type_accounting_counter;        ! Number of request_io's
        acct_execute_io_count: ARRAY[e$fpu_class]                ! Number of execute_io's per fpu class
                               OF e$type_accounting_counter;
    END RECORD;

END e$accounting_def;
```

P.TBD Software Documentation Set Structure - November 12, 1986

Table 1 P.TDB Full Set General Information

Manual	Writer	Status	Pages	Art	VMS Source	Engineer
Introduction to the P.TBD Documentation Set	Marjorie	New	25	5		Benn
Comparing P.TBD and VMS	Liz	New	350	40		Tom M
P.TBD Release Notes Version 1.0	Jim	New	50	0		
P.TBD Master Glossary	Bill		120	0		
P.TBD Master Index	Jim		400	0		

Table 2 P.VMS General User Subkit

Manual	Writer	Status	Pages	Art	VMS Source	Engineer
Volume 1: Using P.VMS						
Introduction to the P.VMS General User Subkit	Marjorie	New	10	0		
Introduction to P.VMS	Helen	Revised	200	53	Jan 15	Tom M
P.VMS Mail Utility Manual	Marjorie	Revised	100	12	Dec 15	Benn
P.VMS General User Master Index	Marjorie		100	0		
Volume 2: System Messages						
P.VMS System Messages and Recovery Procedures	Marjorie	Revised	625	0		Kris B
Volume 3A: Using DCL						
Guide to Using Command Procedures on P.VMS	Bill	Revised	210	2	Jan 19	Kris B
Guide to Using DCL on P.VMS	Bill	Revised	150	6	Jan 15	Kris B
Volume 3B: Using DCL (Continued)						
P.VMS DCL Dictionary	Bill	Revised	725	10	Nov 26	Kris B
Volume 4: Text Processing						
Guide to Text Processing on P.VMS	Helen	Revised	150	16	Jan 15	Benn
Text Processing Utility Manual	SDT	Revised	425	2		Benn
DIGITAL Standard Runoff (DSR) Manual	Marjorie	Revised	240	1		
Volume 5: Using Files and Devices						
Guide to Files and Devices on P.VMS		Revised	100	15	Jan 01	Chris S
P.VMS Sort/Merge Utility Manual	SDT	Revised	90	0		
Introduction to P.VMS Security Features	Marcia	New	40	5	Nov 30	Jim K

Table 3 P.TBD System Management Subkit

Manual	Writer	Status	Pages	Art	VMS Source	Engineer
Volume 1: Introduction						
Introduction to the P.TBD System Management Subkit	Marcia	New	10	0		
Introduction to P.TBD System Management	Marcia	New	150	18	Jan 08	Debbie
P.TBD System Management Master Index	Marcia		150	0		
Volume 2: Setting Up the System						
Guide to Setting Up a P.TBD System	Marcia	Revised	200	18	Jan 08	Debbie
P.TBD Disk Quota Utility Manual	Marjorie	Revised	15	13	Feb 02	Jay
P.TBD Install Utility Manual	Marjorie	Revised	30	13	Feb 02	Lou
P.TBD Terminal Fallback Facility Manual		Revised				
Volume 3: Maintaining the System						
Guide to Maintaining a P.TBD System	Marcia	Revised	200	20	Jan 08	Debbie
P.TBD Backup Utility Manual	Marjorie	Revised	135	13	Jan 19	Debbie
P.TBD Bad Block Locator Utility Manual	Marjorie	Revised	15	0	Feb 02	
P.TBD Error Log Utility Manual	Marcus	Revised	50	8	Dec 01	Marilyn
P.TBD Mount Utility Manual	Marcus	Revised	60	8	Jan 26	Jay
P.TBD System Generation Utility Manual	Marcia	New	100	8	Nov 01	Debbie
P.TBD Verify Utility Manual	Marcus	Revised	40	11	Nov 01	Jay
Volume 4: System Security						
Guide to P.TBD System Security	Marcia	New	200	19	Nov 30	Jim K
P.TBD Access Control List Editor Manual	Marcia	Revised	30	11	Nov 15	Jim K
P.TBD Authorize Utility Manual	Marcia	Revised	60	10	Nov 01	Debbie
Volume 5: System Performance						
Guide To P.TBD Performance Management	Helen	New	100	26	Feb 02	Kathy S
P.TBD Accounting Utility Manual	Helen	Revised	55	9	Sep 25	Debbie
P.TBD Monitor Utility Manual	Helen	Revised	150	23	Nov 07	Kathy S
Volume 6: Clusters						
Guide to PRISM Clustering and Multiprocessing		New	150	20		Chris S
Volume 7: Networking						
Introduction to DECnet	Marcus	Revised	180	20		Jim K
P.TBD Networking Manual	Marcus	New	450	48	Feb 01	Jim K
P.TBD DTS/DTR Utility Manual	Marcus	Revised	20	13	Feb 01	Jim K
P.TBD Network Control Program Manual	Marcus	Revised	200	14	Feb 01	Jim K

Table 4 P.TBD Programming Subkit

Manual	Writer	Status	Pages	Art	VMS Source	Engineer
Volume 1: Introduction						
Introduction to the P.TBD Programming Subkit	Liz	New	10	0		
Guide to Programming on P.TBD	Mary	New	400	30	Dec 12	Kim
P.TBD Programming Master Index	Liz		150	0		
Volume 2: Program Development						
P.TBD Command Definition Utility Manual	Mary	Revised	50	0	Nov 15	Kris B
P.TBD Debugger Manual	SDT	Revised	500	20		Benn
P.TBD Librarian Manual	Mary	Revised	60	0	Dec 12	
P.TBD Linker Manual	Mary	Revised	165	10	Dec 15	Benn
P.TBD Message Utility Manual	Mary	Revised	40	2	Dec 05	Kris B
Volume 3A: System Routines						
P.TBD System Services Reference Manual	Gordon	New	500	30		Kim
Volume 3B: System Routines (Continued)						
VMS Compatibility Routines Reference Manual	Mary	Revised	450	60	Jan 16	Kim/Robert
Volume 3C, 3D, and 3E: System Routines (Continued)						
Introduction to P.TBD Run-Time Library	SDT	Revised	1800	120	Feb 01	Benn
RTL DECtalk (DTK$) Manual						
RTL Library (LIB$) Manual						
RTL Mathematics (MTH$) Manual						
RTL General Purpose (OTS$) Manual						
RTL Parallel Processing (PPL$) Manual						
RTL Screen Management (SMG$) Manual						
RTL String Manipulation (STR$) Manual						
Volume 3F: System Routines (Continued)						
P.TBD Utility Routines Manual	Liz	Revised	310	15		Benn
Volume 3G: System Routines (Continued)						
P.ULTRIX RTL Manual		Revised	500	4		Dave B
Volume 4: File System						
Guide To P.TBD File Applications		Revised	325	40	Dec 15	Chris S
Record Management Services Manual		Revised	330	4	Dec 15	Robert
P.TBD Analyze/RMS_File Utility Manual		Revised	50	6	Dec 12	Robert
P.TBD Convert and Convert/Reclaim Utility Manual		Revised	50	0	Dec 12	Robert
P.TBD File Definition Language Facility Manual		Revised	95	0	Dec 12	Robert
P.TBD National Character Set Utility Manual		Revised				

Table 4 (Cont.) P.TBD Programming Subkit

Manual	Writer	Status	Pages	Art	VMS Source	Engineer
Volume 5A: I/O and System Programming						
P.TBD I/O User's Manual	Gordon	Revised	300	100	Dec 30	Jeff/Chuck
P.TBD System Dump Analyzer Manual	Gordon	New	125	10	Feb 01	Kim
P.TBD Executive Debugger Manual	Gordon	New	30	0		
Volume 5B: I/O and System Programming						
P.TBD RPC and Protected Subsystems Manual		New				Jeff
P.TBD LAT Control Program Manual		Revised	30	0		
P.TBD Device Function Processor Manual	V1.1	New	350	200	Jan 13	Jeff/Chuck
Volume 6: Pillar Programming						
Pillar Reference Manual	Bill	New	200	0		Don
Pillar User Manual	Liz	New	300	45		Don
Pillar Language Summary	V1.1	New	10	0		Don

Table 5 P.TBD Condensed Documentation Set

Manual	Writer	Status	Pages	Art	VMS Source	Engineer
P.TBD Condensed Set General Information						
P.TBD Mini-Reference	V1.1		250	5		
P.VMS General User's Condensed Volume						
Introduction to P.VMS	Marjorie		130			
General User Tasks	Marjorie		220			
General User Reference Information	Marjorie		500			
General User Volume Index	Marjorie		50	0		
P.TBD System Manager's Condensed Volume						
Introduction to System Management	Marcia		150			Debbie
System Management Procedures	Marcia		250			Debbie
System Management Utilities	Marcia		350			Debbie
System Management Volume Index	Marcia		50	0		
P.TBD Programmer's Condensed Volume						
Introduction to Programming	Liz		25			
Programming Utility Reference	Liz		125			
Programming Service/Routine Reference	Liz		650			
Programming Volume Index	Liz		50	0		

Table 6 PRISM Installation Booklets

Manual	Writer	Status	Pages	Art	VMS Source	Engineer
Installing P.TBD on a . . .		New	50	0		
Developer's Guide to PTBDINSTAL	V1.1	Revised	135	0		
DECnet Key Installation Guide		New	10	4		Jim K

Table 7 PRISM C Language Documentation

Manual	Writer	Status	Pages	Art	VMS Source	Engineer
Guide to PRISM C	Helen	Revised	500	22		
PRISM C Installation Guide	Helen	New	15	0		Dave B
PRISM C Language Summary	V1.1	Revised	10	0		Dave B
PRISM C Online Release Notes	Helen	New	10	0		Dave B

Table 8 PRISM Online System HELP Library

Manual	Writer	Status	Pages	Art	VMS Source	Engineer
PRISM Online System HELP Library	Bill		0	0		

Table 9 PRISM Marketing Handbooks

Manual	Writer	Status	Pages	Art	VMS Source	Engineer
PRISM Software Handbook	CMC		300			
PRISM System Overview and Technical Summary	CMC		300			
VMS Software Languages and Tools Handbook	CMC		240			
VMS Software Information Management Handbook	CMC		220			

Table 10 PRISM Software Working Design Document

Manual	Writer	Status	Pages	Art	VMS Source	Engineer
PRISM Software Working Design Document	Jim		1200	0		

Functional Organization by Volume Within Subkit

P.TBD Version 1.0

- General Information

- P.VMS User Interface Subkit
  1 Introduction
  2 Using P.VMS
  3 Text Processing
  4 Files, Devices, and Security
  5 System Messages

- P.ULTRIX User Interface Subkit
  1 Introduction
  2 Using P.ULTRIX
  3 Text Processing
  4 System Messages

- System Management Interface Subkit
  1 Introduction
  2 Setting Up the System
  3 Maintaining the System
  4 Security
  5 Performance
  6 Workgroups
  7 Networking

- Programming Interface Subkit
  1 Introduction
  2 Programming Utilities
  3 System Routines
  4 Files
  5 System Programming
  6 Device Support
  7 Pillar Programming

TO: DECWEST

FROM: Don MacLaren -- 29-Mar-1985

SUBJECT: Systems Programming Language

This is another try at stimulating some discussion of the Systems Programming Language Question. As a starting point, I provide my some of my own opinions about functional advantages and disadvantages of VAXELN Pascal. Note that these opinions are uncorrupted by personal experience with the language -- I did all my VAXELN work in PL/I and MACRO.

1 WHY THE QUESTION?

If we are going to do the system software for a new architecture, we will, deliberately or accidentally, design a systems programming language for it. If the system designers leave this to chance, odd things can happen. I can argue the latter point at length. Maybe it's enough to examine what happened with VAX/VMS. The VMS systems programming language turned out to be the union of MACRO, BLISS, and SDL. The official edict was to use BLISS if feasible. The system designers used Macro, and this was also the publication language, e.g. in the System Services manual. At one time (and maybe yet) the best definition of most RMS features was in the PL/I User's Guide, PL/I being one of the several high-level languages customers prefer to use for systems programming.

Starting soon to think about the language will increase its final quality, even if the language is only a minor variation on an existing language. We should determine what language capabilities are desirable and then choose or design the language -- subject of course to the constraints of schedule and compiler practicality.

Our experience with VAXELN shows that expressing the system in the language from the beginning can improve both.

2 STRENGTHS OF VAXELN PASCAL

As the systems programming language for VAXELN, the most striking feature of EPascal is its integration with the system. It may not be possible to acheive quite this this effect in the more general context of a complete operating system, but it's certainly worth trying.

2.1 Completeness

Everything can be done in the language (well, almost); MACRO is not needed.

2.2 Types And Type Checking

This is the central feature of Pascal. EPascal also applies the notion of compile-time checking to some tricky systems programming things, e.g. what's allowed in an interrupt service routine.

2.3 Flexible Types And Dynamically Sized Data Items

2.4 Strings In The Language

2.5 Type Escapes

These features, especially type casting, lack elegance but they are important for two reasons.

- Systems programming at times requires redescribing data, e.g. to do pointer arithmetic or to get at the parts of a floating point number.
- The language is open ended in regards data structures. One can manipulate data whose structure can't be described within the language's type structure.

2.6 Inline Routines

2.7 Argument List Notation

The capabilities for keyword notation, optional arguments, and variable-length argument lists seem especially significant for the system services typically found in operating systems.

2.8 Modules

There are some questionable details in the EPascal treatment of modules. However the language does provide an explicit form of module that blends with the system treatment

- of source files and separate compilation
- of object modules and linking
- of debugging

3 WEAKNESSES OF VAXELN PASCAL

Considered as a language for a new architecture and operating system, VAXELN Pascal has some functional weaknesses.

3.1 Missing Data Types And Instructions

EPascal is complete for VAXELN, but it doesn't support all VAX data types (e.g. decimal) and all useful instructions, e.g. EMUL.

How EPascal relates to the new architecture's instruction set remains to be seen.

3.2 I/O

Pascal's treatment of files is unrelated to the system's treatment of files (any system). The result is obscurity, inefficiency, and runtime library code that's irrelevant for systems programming.

The text i/o capabilities are primitive.

3.3 Inter-Language Data Structure Definition

Whatever language we choose for the system, many customers will program in one or more other languages. A functional equivalent of SDL is needed. Shouldn't this be part of the systems language?

This is more a question of compiler capability than language. A variant of EPascal front end could be the shell for an SDL-like utility. However the language capabilities should support this usage, and this at least requires review of the language. For example, structures likely to be acessed by other languages should be simple and follow appropriate naming conventions. Does the language help with this?

3.4 Foreign Routine Interfaces

The EPascal features for specifying parameters and calling conventions don't encompass all the conventions used in other VAX languages, e.g. descriptors as used in the VMS languages. A more comprehensive treatment is important for the new system, especially if we accept the idea that the systems language encompasses the SDL function.

This point depends in part on the assumption that many customers will work with multiple languages -- at least the systems language plus their own favorite. In this situation they always end up having to write some routines to brdige the gaps.

Note the implication that routines violating the system's conventions will be written in the systems language. A compiler option can produce discoraging messages for these.

3.5 The Mysterious Linker

Although EPascal was intended to be complete for creating a program (as opposed to building a whole system), the capabilities of the linker aren't reflected in EPascal. Things like shareable images are hard to explain in EPascal terms.

3.6 Inefficient Constructions

There are rough spots in EPascal that promote the generation of inefficient code, e.g.

- value parameters
- sets
- the way functions specify the returned value

In comparison with MACRO, EPascal and other high level languages lose a lot in cases where two or more distinct data nodes are being manipulated, e.g. when setting bits in one node while testing the contents of another. The problem is that the compiler can't tell that the nodes are distinct. Each assignment goes through to storage, even if several bits are being set in the same word. This would look even worse on a RISC machine.

Problems like this are inevitable when using a high-level language, but why accept future code inefficiency that can be avoided by design effort now?

DRAFT PROJECT PLAN

Date: 27 August 1987
From: Don MacLaren
Dept: DECwest Eng.
Phone: (206) 865-8730
MS: ZSO
ENET: DECWET::DON

Subj: DECwest Compiler Project, Description and Plan

NOTE: In this document, PRISM refers to the mainline 64-bit PRISM architecture, not the 32-bit uPRISM.

As part of PRISM project, the DECwest compiler group is producing a highly optimizing compiler for both C and Pillar, which is a new systems programming language whose development is also part of the project. The compiler group is also responsible for some related PRISM utilities, a highly optimizing VAX compiler for Pillar, and some compilers used in the initial development of PRISM hardware and software.

The purpose of this document is to provide people outside the compiler group with the information they need to understand the project and to produce related documents such as business plans and formal project plans. The document covers the PRISM language strategy, DECwest languages and related software, properties of the new compiler, and the schedule through V1 of the Pillar compiler. The document will be updated as required during the project.

If you want to be on the distribution list for updates or you want a copy of this document, please send mail to DECwet::Pillar. Questions about the content of the project plan should be directed to Darryl Havens or Don MacLaren.

There is no formal product management for the compiler project, but it is closely related to the compute server project. For that, contact Cathie Richardson in regards to business product management and Terry Morris in regards to technical product management.

CONTENTS

1 MILESTONES AND SCHEDULE

The milestones listed in this section are the points at which significant software items are available outside the compiler group in stable form. To be available in this sense, software must have passed the applicable test system and, by noon Pacific time of the specified date, must be in the DECwest cluster directory used for software distribution. Documentation must have been dispatched so that it will arrive at the ZK mailroom in time for delivery on the specified day.

As a result of the redirection of the PRISM project, the compiler group's original implementation plan has been revised to get the production PRISM Pillar compiler as quickly as possible. The schedule given here runs only through delivery of that compiler. The schedules for PRISM C, VAX Pillar, the utilities, and the advanced code-optimization features will all be determined later.

The schedule allows for reasonable support of the SIL compilers. It requires timely availability of the PRISM simulator, linker, and librarian, all running on VAX/VMS. Apart from this there are no significant dependencies on other groups. There are no resources to spare within the compiler group.

MILESTONES:

1. SIL V1.0, Feb. 18, 1987. The SIL compilers generate code for VAX or PRISM. PRISM code is packaged in VAX object modules. The reference manual distributed with this compiler is out of date. The SIL User Manual covers mainly the SIL command and the structure of programs and modules. The compiler comes with a set of examples.

2. SIL V2.0. Nov. 87. This version of SIL adds:

   - Structured exception handling including the necessary VAX runtime support.
   - An option to probe and capture arguments for entries to the MICA executive.
   - Some minor language improvements.
   - A SIL Reference Manual and an expanded SIL User Manual.

3. Pillar Reference Manual, Rev. 1.0. Dec. 1987. This is a clean and complete Pillar reference manual. The second Pillar language review begins at this point.

4. SPASM. December 1987. SPASM is the PRISM assembler. This version runs on VAX/VMS but generates true PRISM object modules. It will have a simple macro facility if this is required by the schedule for development of PRISM hardware diagnostics. In the long run, the Pillar compile time facility (CTF) will serve as a SPASM's macro facility.

5. Close Pillar language review. March 1988, about 3 months after distribution of the manual. After this date, no new comments will be accepted. However, there will be a chance

to comment on language changes proposed as a result of the review.

6. Pillar language freeze. April 1988, about one month after the review closes. All language extensions, changes, and clarifications are recorded in the notes file.

7. Pillar V1. June 6, 1988. This is the first production Pillar compiler. It runs on VAX/VMS generating PRISM object modules.

2 DEVELOPMENT TEAM

Compiler group:

Don MacLaren	--	DECwet::DON
Tony Ercolano	--	DECwet::ERCOLANO
John Hamby	--	DECwet::HAMBY
Darryl Havens	--	DECwet::HAVENS
Lois Hayes	--	DECwet::HAYES
Gary Kimura	--	DECwet::KIMURA
Jay Palmer	--	DECwet::PALMER
Lu Anne Van de Pas	--	DECwet::VANDEPAS

Technical Writers:

Helen Custer	--	DECwet::CUSTER
Liz Hunt	--	DECwet::HUNT
Bill Muse	--	DECwet::MUSE

3 PRISM LANGUAGE STRATEGY

This section describes the original PRISM language and compiler strategy worked out between DECwest and SDT. The recent high level events have disrupted the PRISM project, but original language and compiler strategy is still the right strategy. It is compatible with marketing PRISM as a compute server, as a work station, as a complete system with one or more operating systems, and as an architecture with either or both word sizes.

PRISM programming is done in high level languages. Four language products are planned for FRS: C, FORTRAN, Pascal, and Pillar. Additional languages will be provided in a second wave. All PRISM languages share a common language environment that makes it easy for a user to mix languages in a single application. The key components of this common environment are

1. The PRISM calling standard.

2. The PRISM object language.

3. A standard (as yet, unnamed) for general language-independent compiler features such as the form of command options and the arrangement of listings.

4. The PRISM debugger, which is based on the new debug symbol table (DST) architecture.

5. The PRISM performance and coverage analysis utility (PCA).

6. The language sensitive editor, LSE.

7. The source code analyzer, SCA.

Under the compute server approach, the user interfaces to the debugger and PCA are on VAX (integrated with the VAX versions), while LSE and SCA are the VAX versions of these utilities.

Pillar and C are the systems programming languages for PRISM. DEC is using Pillar because of its significant advantages in the areas of type checking, program structure, and integration with the DEC software environment. Most customers will use C for systems, because it is a standard language, it is practical for systems programming, and it has good portability when programmers are careful in their use of the language. Customers will perceive the Pillar and C compilers to be of the same quality -- the highest possible. They will not feel forced to use Pillar.

In addition to the language products, BLISS and a new assembler, SPASM, are also available for use within DEC. Customers can obtain these only by special arrangement. BLISS is used for porting existing software from VMS to PRISM. SPASM is used only in special contexts where normal programming language concepts do not apply.

Pillar, C, and SPASM are implemented by DECwest. FORTRAN, PASCAL, and Bliss are implemented by SDT.

4 PILLAR

Pillar is a high-level systems programming language for use on 32- and 64- bit Digital Equipment systems. The Pillar design emphasizes general features for high-level programming: modules, data type declarations, control structures, and the use of procedures. Examples of Pillar features are:

- A flexible module structure in which information that is not logically part of a definition module can be hidden in a separate implementation module even though it is needed at compile time.

- A treatment of data types, with roots in Pascal, that provides flexible types (parametric types) to describe dynamically sized data and records with variants.

- The sort of type escapes necessary for systems programming, but with some safeguards.

- Inline procedures that can be defined in modules.

- Four distinct modes for parameters: input, output, input-output, and bind, this last mode being for the unusual case of an argument that must be addressed in its original

storage.

- Parameters with matching extents: the extents of the parameter's data type are determined by the extents of the actual argument.

- Structured exception handling.

When compared with low-level systems programming languages, Pillar has the following advantages:

- Software is more portable, because hardware dependencies are isolated in declarations and small procedures. Accidental hardware dependencies are avoided.

- Code is easier to read and maintain. This was emphasized in the design of Pillar's syntax.

- Program development is faster because the compiler detects more errors, and the language provides explicit help in difficult areas, such as exception handling.

- Pillar yields the fastest object code for the PRISM architecture.

Although system independent in most respects, Pillar has been designed to take full advantage of DEC's software technology. For example, exception handling and messages are provided in a way that extends the existing VAX/VMS capabilities. Also, specific hardware features are supported via system dependent modules built into the compiler. The PRISM Pillar compiler has PRISM specific modules for scalar operations (e.g.; shift instructions), vector operations, and privileged operations.

4.1 Pillar Development

Pillar has been developed as a fundamental part of the PRISM project to implement the PRISM executive and PRISM software components that operate above the executive (compilers,linkers, runtime libraries and such). The first draft Pillar reference manual (Rev. 0.0) was distributed for review in November, 1985. The proposed language was closely tied to VAXELN Pascal. As a result of the review the language was redesigned, and it is no longer coupled to Pascal. In February 1987, the redesigned Pillar was made available via the SIL compilers that support most, but not all, of the language.

The documentation distributed with SIL V1.0 was incomplete in regards to SIL, and the SIL language is not exactly a subset of Pillar, which has continued to evolve. The next true Pillar manual will be Rev 1.0. It is scheduled for December, 1987. As soon as the manual is available, the next language review will start.

The style, structure, and principal features of Pillar are set. However there is still plenty of opportunity for improvement via the review. To get as good a language as possible within the

project constraints, the review will be done online over a three month period. All comments and proposed language changes will be published in a notes file as they are received. This way all interested parties will get up to date information on language issues, and people's comments can be timely. A review team will consider all significant issues, and all language changes will be listed in the notes file.

The language resulting from the review will be described in Rev 2 of the Pillar manual, and implemented in the Pillar V1 compiler, which will be the first production Pillar compiler. This compiler will

- o provide an absolutely sound base for system software development and further compiler development.
- o produce clean, moderately optimized code.
- o implement a smooth language that is functionally a superset of SIL.
- o run on VAX/VMS, produce PRISM object modules, and use the PRISM libarian and linker (which will also temporarily operate on VAX/VMS).

It is neither possible nor desirable to make Pillar exactly compatible with SIL. To ease the transition from SIL to Pillar. the compiler will have two features:

1. Whenever possible, the compiler will recognize an obsolete SIL construction, issue a very specific error message, and emit an LSE diagnostic record with correction information. This should make conversion from SIL to Pillar rather easy.
2. There will be a /SIL command option for compatibility. Under this option, the compiler will, whenever possible, recognize an obsolete SIL construction and do the right thing without any error message.

The cases excluded by the phrase "whenever possible" are expected to arise only from code depending on accidental features of SIL and the limitations of SIL's type checking and range checking. The compatibility option will not be removed until after both PRISM and VAX versions of the Pillar compiler are available.

The language supported by the V1 Pillar compiler will be almost the complete language required for FRS of the PRISM system, and the missing pieces will be provided rapidly. (Rev 1 of the manual will indicate which pieces are expected to be deferred until after V1.) Most of the compiler development resources after V1 will be devoted to code optimization, the VAX back end for Pillar, and the PRISM C compiler. However, after people have some experience with Pillar V1, there will be another language review to consider possible language extensions that might be made before or after PRISM FRS.

VAX Pillar requires some additional system-specific features, e.g.; for VAX descriptors. Shortly after V1, there will be a review of VAX features. This review should also cover any issues about the integration of Pillar with the VMS environment, e.g.;

in regards to message files. The schedule for the Pillar VAX compiler depends on project priorities. If it has high enough priority, it can be ready for use in December 1988.

4.2 Pillar Documentation

There are two manuals for PRISM Pillar.

- The Pillar Reference Manual is a concise language reference manual. For the most part it is system independent. In the few instances where the manual does deal with system dependent rules, it will cover both VAX and PRISM. Until the Pillar language is frozen, this manual serves as the language standard. It's being written by Don MacLaren and edited by Bill Muse. Once Rev 2 is published, Bill will convert it to a normal reference manual.

- The Pillar User Manual. This manual is for experienced programmers, but it does not require knowledge of Pillar. It explains how to use Pillar on PRISM, emphasizing the solution of problems that occur in systems programming. There will also be a VAX version of this manual. Liz Hunt is the technical writer for this manual.

Rev 1 and Rev 2 of the Pillar manuals will be distributed (hard copy and labeled company confidential) to everyone on the Pillar and SIL interest lists. The open review procedures will be announced along with Rev 1.

Requests for documentation and general questions about Pillar should be sent to DECWET::PILLAR.

4.3 Pillar Definition Modules

This section discusses Pillar definition modules, which can be used by other compilers and utilities to get information about routines programmed in Pillar. This is especially important for system routines.

Compilation of a Pillar source module, ALPHA, generally produces a definition module in addition to an object module. The definition module contains the declarations of all of ALPHA's exported symbols in a compiled form. An exported symbol is one that may be used in another module. If another module uses symbols from ALPHA, it names ALPHA in an import statement, and the compiler reads ALPHA's definition module. This is more efficient than compiling ALPHA's declarations each time they are used in another module, and it prevents errors in ALPHA from showing up while compiling another module.

In large systems or application programs there is always a danger of inconsistency resulting from failure to recompile a module when declarations on which it depends have changed. To prevent this, Pillar definition modules contain a signature for each exported declaration, and both definition and object modules

record the signatures on which they depend. Consistency of signatures can be checked by the compiler or by the linker (a feature of the MICA linker). Because there is a signature for each symbol, it is not necessary to recompile everything just because a simple change is made to a module.

Pillar definition modules are the means for making system interfaces available to all languages. When the declaration of a system interface is needed in a language other than Pillar, it can be obtained two ways. The other language's compiler can directly import the definition module, or the definition module can be translated into an include or require file in the other language. The direct import method has these advantages:

- It's less bother; there are no require files to manage.
- Module consistency checking can be used.
- The other language's compiler can understand some things that can't be expressed in the language or that require nonstandard expressions such as %DESCRIPTOR.

Because the direct import method has not been used before, and because customers may prefer the concrete form of a require file, we are supporting both methods. There is a definition module utility that translates Pillar definition modules into other languages: C, FORTRAN, Pascal, and Bliss. Additional languages will be supported as they are implemented on PRISM. This utility is structured as a shell plus a set of back ends, one for each target language. The shell reads the definition modules and builds a symbol table in memory. Each language-specific back end accesses the symbol table through a set of routines provided as part of the shell. The shell and C backend are implemented by DECwest, the other backends by SDT.

The parts of the shell that build and access the symbol table will be packaged so that they can be incorporated into other utilities. In particular, this will be the way in which the SDT compilers directly import Pillar definition modules.

5 PRISM C

For systems programming, PRISM customers are most likely to use the C language, and this usage will be intermixed with applications programming in C. (There is no clear boundary between systems and applications programming.) The compiler will be heavily used in the scientific, technical, and educational markets. It will feature the industry's most advanced code optimization methods: interprocedural analysis, vectorization and decomposition, instruction scheduling, global register allocation, and performance profile feedback to improve all aspects of code generation and optimization.

Within the PRISM project, PRISM C will be used for ULTRIX-related software (including ULTRIX itself if it is ported to PRISM). C will also be used for parts of the Applications Interface Architecture (AIA) on MICA.

PRISM C will be an implementation of the ANSI C standard. This standard specifies certain syntax for implementation-specific extensions to the standard language. This syntax will be used in VAX C, V3.0, for both old and new extensions. PRISM C will contain the extensions from VAX C that do not depend on the target architecture. A point to note here is that the VAX C, V3.0, extensions related to optimization are being defined in a way that is not dependent on the architecture, so they will be supported by PRISM C.

PRISM C will support 64-bit integers as well as 16- and 32-bit integers. This may require an extension. Built-in functions for privileged operations and atomic memory access may be required, especially for the ULTRIX executive. No other language extensions are planned for PRISM C. However, the compiler's ability to import PRISM definition modules will be a valuable feature for users of PRISM C on MICA.

Validation for the C compiler will use a test system derived from the VAX C test system. The compiler will also be tested via its use in compiling ULTRIX software.

PRISM C is documented in the Guide to PRISM C. This has the same organization as the Guide to VAX C, and the two manuals differ only where the systems differ. Helen Custer is the technical writer for this manual.

Direct questions about the PRISM C language to Lu Anne Van de Pas.

For information on the C runtime library on MICA, see the MICA project plan.

6 MISCELLANEOUS LANGUAGE SOFTWARE

This section covers the other PRISM software being developed by the DECwest compiler group.

6.1 Message File Compiler

Pillar has features for defining conditions and messages either casually (in a Pillar program) or in a message file. The Pillar message file compiler accepts a Pillar message source file. It produces a message file, an object module, and a Pillar definition module. The definition module can be imported by any module that needs to reference a message in the file. Note that there is a single source for all information about messages and conditions defined in the file.

To produce messages in a different natural language, the message file is edited and recompiled. The message file compiler will be able to import the original definition module and check the modified source file for consistency with it.

Pillar message files will be the system message files for PRISM. For VAX/VMS Pillar, the compiler will generate VMS message files.

6.2 SPASM. the Simplified PRISM Assembler

SPASM is the new assembly language for PRISM. It is not compatible with the language accepted by the interim assembler. SPASM will use the Pillar lexical analyzer, so it will be possible to use the full Pillar compile time facility with SPASM. However, to meet the diagnostic group's short term requirements for a macro capability, SPASM will have a simple intrinsic macro language.

SPASM will be used only in special contexts where normal programming language concepts do not apply. Customers can only obtain the assembler by special arrangement.

Gary Kimura is responsible for SPASM, including definition of the language.

6.3 Pillar Runtime Support

Pillar object code may use out of line complex code sequences and a few runtime routines that are not known to the user. These will be implemented by the DECwest compiler group. Because the executive of the PRISM operating system (MICA) is written in Pillar, this runtime will be packaged with the executive.

6.4 DST Analysis

The MICA ANALYZE command will have an option to analyze the debug symbol table in an object module or image. The DST specification is being done by the debug group in SDT. The DECwest compiler group is doing the analysis program.

7 THE DECWEST PRISM COMPILER

This is the PRISM compiler for Pillar and C, and it also includes the SPASM assembler. It will feature the industry's most advanced code optimization methods: interprocedural analysis, inline routine expansion, vectorization and decomposition, instruction scheduling, global register allocation, and performance profile feedback to improve all aspects of code generation and optimization.

The general goal for the compiler group is to produce a compiler that will meet PRISM performance goals and that will be perceived as being of higher quality than any existing VAX compiler. Here quality encompasses ease of use, compile speed, reliability (correct behavior), and object code performance. Object code performance is receiving special emphasis in the PRISM project, but all the goals are important. In particular, reliability must

not be compromised.

The compiler has roots in the family of compilers using the VAX Code Generator (VCG), but the design is completely new. It has been influenced by eight years of experience with the VCG compilers and by new work done at DECwest in the context of the SIL compilers and their Pascal predecessors. This section describes some of the newest features of the design: the modular organization and the code optimization methods.

7.1 Compiler Organization And Command Interface

The compiler is structured so that it can be easily integrated with new environments and hosted on a variety of systems. It contains:

1. a small super shell that contains all functions related to the host operating system and command interface,
2. a small language driver routine for each language,
3. a compiler shell providing general routines used by all parts of the compiler,
4. a separate front end for each language,
5. a back end that does optimization and code generation including all target-dependent code generation.

A complete compilation is controlled by the language driver. Supported by the super shell, it interprets the command line and establishes an environment for the compilation. The driver then calls the compiler shell to initiate the real work. To integrate the compiler with a new program-development environment, one only needs to modify the language drivers. For example, a fancy program development system can provide its own drivers thus bypassing the normal command interface.

The expansion of displayed text (e.g.; error messages) can be controlled by the language driver (using the super shell). There is complete flexibility in regards to the translation of messages into national languages.

The compiler can be packaged in various ways: one big image, a set of related shareable images, three separate images, etc. Whatever arrangement is finally chosen, the DECwest compiler group regards development of this compiler as one project being done by one team.

The shell and super shell are used in the Pillar definition module utility and the message file compiler.

7.2 Code Optimization

Almost all optimization is done in the compiler's common back end so that it will apply to all languages except SPASM. As in the

VCG compilers, the operators of the intermediate language are n-tuples, but all traces of PL/I have been removed, and the basic classification of data types deals only with size and alignment. From the point of view of the front ends, certain key operators, such as those for data references and procedure calls, are simplified. The intermediate language is always processed by the compiler's global optimizer, which converts the difficult operators into forms most appropriate for optimization.

The first phase of the global optimizer scans the intermediate language and extracts information about the calling relations between procedures and the usage of variables within procedures. This information is then analyzed to get sharp information about procedure calls and data aliasing. Here, as in many places in the back end, there is an option for quick analysis or deeper, more time consuming analysis. The user will not see these options directly, rather there will be a few practical command options to control the compiler. The default mode is for the maximum optimization consistent with quick compiling.

The second phase of the global optimizer performs conventional global optimization on the intermediate language. Procedures are processed separately using the results of the preceding inter-procedural analysis. The flow analysis method is a variation of recursive descent analysis that works on flow graphs and accommodates moderate usage of goto's. This method's running time is linear in the number of graph nodes, and the storage required for bit vectors depends (more or less) on the nesting depth of control structures rather than the number of nodes in the graph. Recognition of equivalent expressions uses a combination of hashing and self-adjusting binary trees, so the time spent is at worst n*log(n).

Inline procedure expansion and inter-procedural analysis both yield many opportunities for value propagation, which can result in the recognition of constant conditionals. The flow analysis and recognition of equivalent expressions is designed to exploit this, and the flow graph is simplified whenever possible. The flow analysis can be repeated on the simplified graph, and there will be a provision for repeating the interprocedural analysis using the sharper information found by flow analysis.

The global optimizer does standard optimizations such as loop unrolling and result incorporation. Over time we add many specialized optimizations of this sort, the most interesting being vectorization and decomposition (into parallel threads of execution). The optimizer also collects interference and life-time information for later optimization phases. When it's finished with a procedure, it produces the optimized intermediate language in the form expected by the local code generator.

The Local Code Generator (LCG) reads the intermediate-language operators produced by the global optimizer. It generates unbound code blocks which implement the operators. The unbound code blocks look something like instructions, but use register temporaries and symbol nodes rather than hardware registers to keep track of the storage of operands. The actual instructions being emitted by the LCG may also contain pseudo-opcodes rather than real instructions.

The code blocks are unbound in the sense that their interrelationship is not constant at the end of the code generation phase of the compiler. This allows the code scheduler to freely reorder the instruction stream based on the machine which the code is being generated for.

The LCG may also output more than one sequence of instructions for a given operator. For example, when a string of characters is to be moved from one location to another, the code generator might provide code blocks to move the string three different ways:

1. a straight inline code sequence, or

2. a loop to move the string, or

3. a call to a complex code sequence to move the string.

It is then up to another phase of the compiler which runs after the code generator to select which sequence should be used based on profile information, register usage, instruction stream cache information, etc.

The Code Block Optimizer (CBO) works on the unbound code blocks produced by the LCG. Actions of the CBO include:

1. Keeping track of the constants which are too large to be placed in the instruction stream itself and allocating the storage in linkage section to hold the constants.

2. Keeping track of constants which have been loaded into register temporaries. This allows the CBO to ensure that no constants are loaded into register temporaries which have already been loaded, therefore cutting down on the number of memory load instructions which are performed.

3. Coalescing register temporaries so that two register temporaries may exist in the same register temporary. This cuts down on the number of register temporaries that the register allocator must deal with. It also guarantees that a value in one register temporary which is simply being moved to another register temporary will not use two separate hardware registers.

4. Performing some peephole optimizations that apply before scheduling and register allocation. The CBO takes out some unnecessary instructions or changes their sequences so that there are fewer instruction code blocks.

Because the CBO is dealing with all of the code blocks output by the LCG at once, it has a much better view of the operations actually being performed. It also has the graphs that the optimizer built which it can use to make some decisions about how to optimize out code blocks. All of this allows the CBO to actually peephole code blocks across basic blocks.

The instruction scheduler runs after the Code Block Optimizer. It rearranges the order in which PRISM instructions are issued to minimize execution time due to stalled instruction issue cycles. It is an optional phase of the PRISM compiler. For each specific

PRISM processor, the scheduler uses a different model to describe its characteristics with respect to when it will stall on an issue cycle.

The scheduling algorithm can be tuned for various levels of optimizations. It can schedule basic blocks or entire procedures. Scheduling basic blocks is the simplest and fastest scheme where the instruction scheduler only rearranges instructions within a single basic block, one basic block at a time. When scheduling an entire procedure, the instruction scheduler is allowed to rearrange and move instructions anywhere within the procedure. The scheduler will use flow graph information (provided by the optimizer) and profile information to help schedule entire procedures. Intermediate degrees of scheduling will also be defined as warranted.

The register allocator runs after the instruction scheduler. Its task is to assign register temporaries to actual hardware registers, insert spill code as needed, finalize the decision on which procedures need or do not need a call frame, insert prologue and epilogue code, and complete the storage allocation. It uses flow graph information, call graph information (both provided by the optimizer), and profile information to help it assign register temporaries to hardware registers. An underlying goal in register allocation is to minimize hardware register usage, spill, and the need for call frames.

Like the scheduling algorithm, the register allocation algorithm is tuned for various levels of optimizations. A fast allocator will only process one procedure at a time and use a simple fixed point method for allocation. The more thorough allocator will use all of the flow information available and deal with global register usage. It will also assign parameters to nonstandard registers for procedure calls where it is possible and beneficial.

Several different register allocators, each using different algorithms, were experimented with during the development of the SIL compiler. The final allocator selected for use in that compiler uses a non-backtracking form of coloring algorithm and allocates registers across procedures within a compilation unit. This allocator was selected for that compiler based on its output relative to the actual processor time required to complete the compilation.

The optimizations discussed so far are based on the analysis of the procedures in a single compilation unit. The compiler is designed to work with very large programs, but there is still a need to carry out interprocedural analysis and register allocation across separate compilation units. Within DEC, this has been named universal optimization. A long-term goal for both PRISM compiler projects is to provide universal optimization in a way that is not tied to a single language or compiler.

To efficiently support universal optimization and processor sensitive optimization, especially instruction scheduling, the DECwest compiler will provide deferred code generation. The intermediate language representation of a module (or multiple modules) can be saved in a deferred object module and compiled later. The deferred compilation starts with the global optimizer

phase. The target processor can be specified at this time, and profile information or the results of universal optimization may be used.

7.3 Performance Analysis

Performance analysis of program code is receiving special attention in the PRISM Pillar and C compiler. In addition to its value for programming compute-intensive applications and system software, this sort of performance analysis contributes to the development of the compiler's code optimization methods and to the evaluation of hardware architectures and designs. The traditional separation between hardware performance analysis and compiler development has handicapped both VAX and PRISM development.

On PRISM, the most useful data for performance analysis appears to be execution profiles generated by code inserted by the compiler. This can be related to the structure of the program as seen by the user and also to the fine structure used in code optimization. The compiler will have the capability to generate this form of profile code, and the results can be fed back to improve optimization in a subsequent compilation of the same program. We have successfully experimented with this in the PRISM SIL compiler, including an experiment where the compiler varied the number of hardware registers and reported the resulting numbers of loads and stores.

The compiler will have a special option to accept information produced by the PRISM timing simulator. It will be able to display this information, the regular profile information, and its own code scheduling assumptions as part of the machine code listing. In addition, it can display interesting statistics. The point of this is to get all the relevant information together in a useful form for design feedback. This should eliminate inconsistencies in the hardware- and compiler- design assumptions.

8 VAX SOFTWARE

Pillar will be a product on VAX/VMS. The software involved is the Pillar compiler, the related runtime support, and the Pillar definition module utility.

The VMS Pillar compiler differs from the PRISM compiler only in the super shell and in those parts of the back end that depend on the target architecture. All of the compiler's general optimization apparatus is used on both VAX and PRISM.

The Phase 1 review for VAX Pillar will be held sometime after the PRISM Pillar V1 compiler is available. At that time, VMS-specific specifications for the compiler will be available for review.

8.1 VAX Software Dependencies

The plan for the VAX Pillar compiler depends on VAX Debug accepting the new format debug symbol table and providing language-specific support for Pillar ("SET LANGUAGE PILLAR").

9 BOOTSTRAP SOFTWARE

PRISM has a new hardware implementation architecture, a new operating system, and a new systems programming language. Developing the new system requires an elaborate bootstrap process. Most of the compiler group's work prior to June 1987 has been on software to be used in the bootstrap and then discarded. The compiler group is responsible for:

- The SIL cross compiler. This produces PRISM object code packaged in VAX object modules. The code can be used on the PRISM emulators or under PRISM simulators running on VAX. This compiler features instruction scheduling and global register allocation across procedures. It uses the PRISM calling standard and has options to assist in hardware evaluation.

- The VAX/VMS SIL compiler. This is used for the development of modules and programs that do not require PRISM-specific functions (e.g.; privileged instructions). The compiler does support the vector operations via a runtime package. Programs such as the PRISM and C compiler and the MICA linker will be developed using this compiler.

- An interim macro assembler. This is a modification of the VAX macro assembler that produces PRISM object code packaged in VAX object modules. Note that this assembly language will not be supported on PRISM.

The SIL compilers have an option to translate Pillar data declarations and procedure declarations into interim Macro.

The SIL compilers (PRISM and VAX) will be supported until the Pillar compilers (PRISM and VAX, respectively) are available.

Preliminary 64- and 32-bit Pascal compilers for PRISM were developed, used for a while, and retired.