Alpha AXP Firmware Porting Guide

Revision/Update Information: Version 1.0

Order Number: EK-AXFRM-PG. A01

November, 1994

This document was prepared using VAX DOCUMENT Version 2.1.

Contents

Preface xiii

1 Porting Strategy

Overview 1–3
- Introduction 1–3
- Overall Process of Porting 1–4
  - Overall Process 1–4
- Porting Scenarios 1–5
  - Overview 1–5
  - Hardware Under Development 1–5
  - Hardware Functional 1–5
- Porting Strategy 1–6
  - Reference Platform 1–6
  - Cloning Files 1–6
  - Building Multiple Console Images 1–6
  - Summary of Porting Strategy 1–7
  - Applying the Strategy 1–7

Process of Porting 1–9
- Introduction 1–9
- Porting Process 1–10
  - Process Steps 1–10
- Requirements and the Development Environment 1–12
  - Requirements 1–12
  - Build Environment 1–12
  - Code Management System 1–12
- Porting Factors 1–13
  - Factors Affecting Effort 1–13
  - Similar Platform 1–13
  - Different DECchip 1–13
  - Different Platform 1–13
  - Operating System 1–13
  - Using a Reference 1–14

2 Console Overview

Overview 2–3
- Introduction 2–3
- Console Image 2–4
  - Code Sections 2–4
  - Loading the Image 2–4
- Console Interfaces 2–6
  - Introduction 2–6
  - Hardware Interface 2–6

Driver Model 2–7
User Interface 2–7
Configuring Commands 2–8
Operating System Interface 2–8

Functionality 2–9
Introduction 2–9
Console Functionality 2–10
Introduction 2–10
PALcode Functions 2–10
Console Functionality and Porting 2–10
Console Initialization 2–11
Introduction 2–11
Powerup Process 2–11

3 Cloning, Building and Testing the Reference Image

Overview 3–3
Introduction 3–3
Reference 3–3
Overview of Cloning Files 3–4
Introduction 3–4
Model Port 3–4
DECchip 21066 3–4

Cloning and Modifying the Reference Source Code 3–5
Introduction 3–5
Development Environment 3–6
Overview 3–6
Major Components 3–6
Save Sets 3–7
Directory Tree Structure 3–8
Working Directory 3–8
Configuration Directories 3–9
Platform Base 3–10
Configurations 3–10
Configuration Build Files 3–11
Configuration Subdirectories 3–11
Creating the Environment 3–11
Engineering Environment 3–13
Engineering VMScluster 3–13
System Load 3–13
Cloning Files 3–14
Introduction 3–14
Categories of Files to Clone 3–14
Cloning and Modifying Platform-Specific Files 3–15
Introduction 3–15
Source Files to be Cloned 3–15
Cloned Source Files Example 3–15
Cloning and Modifying Code Fragments 3–16
Overview 3–16
Changing Conditional Statements 3–16
List of Files 3–17
Cloning Build Files 3–18
Introduction 3–18

Build Files 3–18
Modifying Build Files 3–19
Configuration Build File 3–19
Modifying the Configuration 3–19

Building the Reference Image 3–21
Introduction 3–21
Building the Reference Console 3–22
Three Phases 3–22
Build Steps 3–22
Configuration Log Files 3–25

Testing the Reference Console 3–27
Introduction 3–27
Using the Console Interface 3–28
Introduction 3–28
Commands Description 3–28
Introduction to Console Commands 3–28
Console Prompt 3–28
Console Shell 3–28
Commands 3–28
Syntax 3–30
Running Diagnostics 3–31
Scripts 3–31
Generic Diagnostic Scripts 3–31
Diagnostic Session 3–32
Sample User Session 3–32
Diagnostic Messages 3–35
Types 3–35
Error Messages 3–35
Bootstrapping 3–37
Introduction 3–37
Bootstrap Procedure 3–37
State Assumed by the Operating System 3–37
Conditions for Bootstrapping 3–37
Steps for Bootstrapping 3–38
Boot Environment Variables 3–38
Boot Environment Variables After Loading 3–39
Boot Command Syntax 3–39
Booting the OpenVMS Operating System 3–40
Overview 3–40
BOOT_OSFLAGS Variable 3–40
Other Boot Options 3–40
Recovering from Errors During System Booting 3–41
Introduction 3–41
Bootstrap Flags Description 3–41
Bootstrap Flags List 3–41
Bootstrap Flags Values 3–41
Booting from the Ethernet 3–42
Introduction 3–42
Two Protocols Supported 3–42
MOP Booting 3–42
MOP Network "Listening" 3–42
BOOTP Booting 3–42
Environment Variables 3–43

Ethernet Boot Example 3–43
Booting DEC OSF/1 3–44
Boot Environment Variables 3–44
Boot Flag Definitions 3–44

4 Developing a Minimal Image for the Target

Overview 4–3
Introduction 4–3
Overview of Modifications Needed 4–4
Memory Modification 4–4
Chip-Dependent Modification 4–4
Kernel Modification 4–5

Defining Memory 4–7
Introduction 4–7
Defining the Memory Configuration 4–8
Platform Memory Configuration 4–8
SROM Memory Code 4–8

Modifying PALcode 4–11
Introduction 4–11
PALcode Modification 4–12
Introduction 4–12
PALcode Functionality 4–12
PALcode Modification Steps 4–12
PALcode Changes 4–14
Overview 4–14
Supporting Interrupts 4–14
Selecting Different Console 4–14
Building PALcode 4–15
MMS Description Files 4–15
Platform MMS File 4–15
Image File Description 4–15
MMS Build Files 4–16
PALcode MMS Description File 4–17
pal_descrip.mms File 4–17
pal_descrip.mms Example 4–17

Modifying the Kernel Files 4–19
Introduction 4–19
Adding Console Terminal Support 4–20
Introduction 4–20
Modified Macros 4–20
Console Terminal Address 4–20
Platform Utility File Example 4–21
Time of Year (TOY) 4–23
TOY Files 4–23
TOY Example 4–23
NVRAM - Environmental Variables 4–24
Overview 4–24
Modifying NVRAM Files 4–24
Address Changes Example 4–25
Changing the NVRAM Driver 4–26
Environmental Variable Debug 4–27
Turning Off NVRAM Drivers 4–27

Tracing the Transition from PALcode to Kernel 4–28
Overview 4–28
Showing Progress During Initialization 4–29
Terminal Echoing 4–29
jputc Routine 4–29
jputc Example 4–29
printf Statements 4–30

Modifying the Driver Files 4–31
Introduction 4–31
Configuring the Bus 4–32
Defining Addresses 4–32
platform_io.h Example 4–32
Bus Windows for Device Controllers 4–34
Bus Windows Description 4–34
Sequencing Driver Startup 4–35
Introduction 4–35
Driver Startup Process 4–35
Specifying Driver Startup 4–35
Driver Startup Example 4–35
Disabling Hardware Drivers from Loading 4–36
Robust Mode 4–36
Loading and Unloading Drivers 4–36
Powerup Process 4–36
Tracing Driver Initialization 4–37
Using printf to Display Drivers 4–37
Using Polled Mode for Drivers 4–38
Polling Overview 4–38
Setting Polled Mode 4–38
Conditional Directive 4–38
Hard Coding the Interrupt Mode 4–39

Modifying the Build Files 4–41
Introduction 4–41
Modifying the Configuration Build File 4–42
Adding Drivers, Files and Options 4–42
Configuration Build File Example 4–42
Default Options 4–42
Hardware and Architecture Name 4–42
Value Statements 4–43
Option Statements 4–43
Removing and Adding Kernel Files 4–43
Groups 4–43
Group Driver 4–44
Default Build Options and Values 4–45
options.bld File 4–45
Options and Values 4–45
Build Options Example 4–45
Modifying for Nightly Builds 4–47
Modifying build.com 4–47

Debugging 4–49
Introduction 4–49
Using XDELTA 4–50
XDELTA Overview 4–50
Building with XDELTA 4–50

XDELTA Example 4–50
Default Options 4–50
Turning Off XDELTA 4–51
Invoking XDELTA 4–51
Invoking from OpenVMS 4–51
XDELTA Commands 4–52
Commands 4–52
Extended XDELTA 4–57
Overview 4–57
Commands 4–57
Debug Session Example 4–58
Building with Extended XDELTA 4–59
Command Summary 4–60

5 Debugging the Hardware

Overview 5–3
Introduction 5–3
Hardware Debugging Techniques 5–4
Overview 5–4

Serial ROM 5–5
Introduction 5–5
Serial ROM (SROM) Functionality 5–6
Introduction to Serial ROM 5–6
Functions of Serial ROM Code 5–6
Loading into Bcache 5–6

SROM Mini-Console 5–7
Introduction 5–7
SROM Mini-Console 5–8
Overview 5–8
Mini-Console 5–8
Features 5–8
Hardware Required 5–8
Setting Up the SROM Serial Port Connection 5–9
Using a Workstation for the Connection 5–9
Connecting the Workstation Procedure 5–9
Starting and Running the Mini-Console 5–10
SROM Mini-Console Command Set 5–11
Command and User Interface Features 5–11
Command Summary 5–12

6 Adding Functionality

Overview 6–3
Introduction 6–3
After Adding Functionality 6–3

Adding Functionality 6–5
Introduction 6–5
Selecting the Console Device 6–6
VGA or Character Cell 6–6
Forcing Output to the Serial ASCII Port 6–6
Terminal/VGA Driver Sequence 6–6
Console Selection Algorithm 6–7

Adding Drivers to the Build Files 6–8
    Adding Drivers Description 6–8
    Format 6–8
Developing and Adding Scripts 6–9
    Encapsulating Commands 6–9
    Adding Scripts 6–9
    Invoking Scripts 6–9
    Inodes 6–9
Adding ARC Support 6–10
    Adding ARC Support Description 6–10
Adding and Modifying Console Commands 6–11
    Introduction 6–11
    Process Overview 6–11
    Existing Command Files 6–11
Modifying for Bootstrapping 6–12
    Introduction 6–12
    Modifications 6–12
Programming Diagnostics 6–13
    Introduction 6–13
    Diagnostic Service Routines 6–13
Diagnostic Models 6–14
    Introduction 6–14
    Test-Directed Diagnostics 6–14
    Unstructured Diagnostics 6–14
Describing Data Structures 6–15
    Introduction 6–15
    Process Control Block 6–15
Diagnostic Restrictions 6–16
    Restrictions Description 6–16
Related Console Services 6–17
    Introduction 6–17
    Routines 6–17

7 Appendices: Helpful Commands, Tips and Routines

Appendix A: Logical Search String Definitions 7–2
    Building Environment Logicals 7–2
    CP$SRC Generic 7–2
    CP$SRC Specified Configuration 7–3
Appendix B: Getting Started with CMS 7–7
    Overview 7–7
    Hierarchy 7–7
    Creating CMS Libraries 7–7
    Setting the Library 7–7
    Using CMS with DECwindows 7–7
    CMS Class Definition 7–8
    Show Classes 7–8
    CMS Groups 7–9
    CMS Elements 7–9
Appendix C: Complete List of Modified Files 7–10
    Overview 7–10
    Files: CMS Generations 7–11
Appendix D: Initialization, Kernel, and Diagnostic Routines 7–12
    Introduction 7–12

Sequence of Kernel Modules 7–12
System Initialization Routines and Data Structures 7–12
entry.c Module 7–13
kernel.c Routines 7–14
kernel_ alpha.mar Routines 7–15
kernel_def.sdl Module 7–15
kernel_vax.mar 7–16
process.c Routines 7–16
powerup.c Routines 7–17
Diagnostic Services Routines 7–18
Appendix E: SROM Mini-Console Command Summary 7–19
Commands 7–19
Appendix F: XDELTA Commands 7–20
Appendix G: Basic Console Commands 7–22
Syntax 7–22
Appendix H: Diagnostic Environment Variables 7–23
Environment Variables 7–23

Glossary

Overview Glossary–1

Index

Examples

3–1 Modifying Preprocessor Directives 3–16
3–2 Modifying a Cloned Build File 3–19
3–3 Boot Command 3–39
3–4 Booting from the Ethernet 3–43
3–5 Setting the Boot Variables for DEC OSF/1 3–44
4–1 SROM Memory Configuration Registers 4–9
4–2 pal_descrip.mms for a Sample Platform 4–17
4–3 Defining the Address in the Platform Utility File 4–21
4–4 TOY Base Address - combo_def.h 4–23
4–5 The nvram_def.h File 4–25
4–6 The esc_nvram_driver.c File 4–26
4–7 Tracing kernel_alpha.mar 4–28
4–8 Tracing kernel.c with jputc 4–29
4–9 Tracing kernel.c with printf 4–30
4–10 PCI and EISA I/O Address 4–33
4–11 Defining PCI Bus Windows 4–34
4–12 Setting Robust Mode in powerup_platform.c 4–36
4–13 Tracing Driver Startup in filesys.c 4–37
4–14 Conditionally Defining Polled Mode in a Driver 4–38
4–15 Defining Polled Mode in a Driver Using Hard Coding 4–39
4–16 The options.bld File 4–45
4–17 Modifying build.com 4–47
4–18 Including XDELTA in the Build File 4–50
4–19 Excluding XDELTA from the Configuration Build File 4–51

4–20 Invoking XDELTA at the Console Prompt 4–51
4–21 Invoking XDELTA at the User Level 4–51
4–22 Using Extended XDELTA to Debug 4–58
4–23 Extended XDELTA Command Summary 4–60
5–1 Connecting the Workstation 5–9
7–1 CMS SHOW CLASS Output 7–8
7–2 Common Console Code Groups 7–9
7–3 Ported Files 7–10
7–4 Build Files 7–11
7–5 Kernel Source Files 7–11
7–6 Drivers 7–11

Figures

1–1 Porting Process 1–4
2–1 Console Image 2–5
2–2 Alpha AXP Firmware Interface 2–6
2–3 Class, Port, and Bus Driver Model 2–7
3–1 Major Components of the Development Environment 3–6
3–2 Directory Tree 3–8
3–3 Directory Tree, Configurations 3–9
7–1 CP$SRC Generic Environment 7–2
7–2 CP$SRC for Specified Configuration Environment 7–4
7–3 XDELTA Command Reference 7–21

Tables

1 Course Conventions xvi
1–1 Porting Procedure 1–10
3–1 cp_common_login.com Parameters 3–12
3–2 Cloned Platform-Specific Source Files from Another PC Platform 3–15
3–3 Cloned Build Files from Another PC Platform 3–18
3–4 Building the Alpha AXP Firmware 3–22
3–5 Console Commands 3–29
3–6 Syntax of Common Console Commands 3–30
3–7 Diagnostic Command Scripts 3–31
3–8 Environment Variables 3–38
3–9 OpenVMS Boot Options 3–40
3–10 Other OpenVMS Boot Options 3–40
3–11 Bootstrap Flags 3–41
3–12 DEC OSF/1 Boot Flag Definitions 3–44
4–1 Platform-Dependent Memory Functionality 4–4
4–2 Platform-Dependent Chip Functionality 4–4
4–3 Platform-Dependent Kernel Functionality 4–5
4–4 PALcode Files for a Sample Platform 4–15
4–5 XDELTA Functions and Commands 4–52
4–6 Extended XDELTA Commands 4–57

5–1	Command Summary	5–12
6–1	Driver Keywords within Build File	6–8
6–2	Command Files	6–11
6–3	Diagnostic Service Routines	6–13
6–4	Diagnostic Programmer I/O Routines	6–17
6–5	Environment Variable Manipulation	6–18
6–6	Dynamic Memory Allocation	6–18
6–7	Condition Handling	6–18
6–8	Timer Services	6–19
6–9	Semaphores	6–19
6–10	Multiprocessor	6–19
6–11	Ctrl/C/kill Checking	6–19
7–1	Building Environment Logicals	7–5
7–2	Creating a CMS Library	7–7
7–3	Module entry.c Routines	7–13
7–4	kernel.c Routines	7–14
7–5	kernel_alpha.mar Routines	7–15
7–6	kernel_def.sdl Data Structures	7–15
7–7	kernel_vax.mar Routines	7–16
7–8	process.c Routines	7–16
7–9	powerup.c Routines	7–17
7–10	Diagnostic Service Routines	7–18
7–11	Command Summary	7–19
7–12	Syntax of Common Console Commands	7–22
7–13	Environment Variables	7–23

Preface

About This Document

Document Description

This document provides information on how to port the Common Console. This document is designed to be used sequentially for porting and as reference.

Intended Audience

This document is written for system designers who are creating the firmware for a new hardware platform using the Common Console.

Goals of the Document

Specifically, this document provides information to help the system designer:

- Identify a porting strategy
- Clone, build and test the console on a reference platform
- Debug the hardware on the new target platform
- Develop a minimal image for the new target platform
- Add basic functionality to the new console

Nongoals

This document does not cover the following:

- The architecture of the Common Console
- Instruction in C programming

Document Organization

This document is presented in a task-oriented format. Appendices are included to provide reference information, such as a listing of commands for the Common Console and the SROM mini-console.

This document is divided into chapter, each covering a related group of topics.

Each chapter consists of:

- An **Overview** to the subject matter of the chapter
- The **text** of the chapter, which includes outlines, text, tables, figures and examples
- A **Summary** that highlights the main points presented in the chapter

Chapter Descriptions

Each chapter is as follows:

- **Porting Strategy** — Introduces the porting process.
- **Console Overview** — Introduces the interfaces and functionality of the Common Console.
- **Cloning, Building and Testing the Reference Image** – Provides information on how to clone, modify, build and test the source code of a console that runs on an existing platform. This platform is called the “reference” platform.

- **Developing a Minimal Image for the Target** — Provides information on how to get to the first console prompt by defining memory, modifying PALcode, modifying the kernel and driver files and implementing additional debugging procedures.
- **Debugging the Hardware** — Provides an overview of debugging the new hardware platform by setting up the Serial ROM (SROM) serial port connection and using the Serial ROM (SROM) mini-console.
- **Adding Basic Functionality** — Provides information on functionality that can be incrementally added to the minimal image to produce a fully functional console, including adding console commands, device drivers and scripts.
- **Appendices** — Provides reference information on logical search string definitions, the Code Management System (CMS), files modified during porting, kernel modules and routines, the SROM mini-console, XDELTA and Common Console commands.

Resources

For more information about topics discussed in this document, see the following:

- *DECchip 21064 Microprocessor Hardware Reference Manual*
- *Alpha AXP System Reference Manual V5.0*
- *Advanced RISC Computing Specification Version 1.2*
- *Device Driver Interface Guide for the Alpha AXP Firmware*
- *Alpha Firmware Design Document Version 0.10*
- *Software Engineering Manual, 1988, A-DG-ELEN571-00-0*
- Specifications, such as, PCI, EISA, SCSI, X, and DSSI bus

For information on a specific chip, see the specification for that chip.

Document Conventions

Table 1 describes the conventions used throughout this document.

Table 1 Course Conventions

Convention	Meaning
Ctrl/x	Press and hold the key labeled CTRL while you press another key (X). Many control keys have special meanings.
UPPERCASE	Commands used with the OpenVMS command interpreter appear in uppercase characters, preceded by a dollar sign prompt ($) and indicate words you type exactly as they appear. For example, you would type the following commands as they appear: `$ DIRECTORY` `$ TYPE LOGIN.COM`
lowercase	Lowercase characters represent commands typed to the Alpha AXP Firmware command interface. The commands are preceded by a triple greater-than prompt (>>>) and are words you type exactly as they appear. An example follows. `>>>ps`
Mixed case	Mixed case characters represent elements that you must replace according to the description in the text, and typically appear with hyphenation or an underscore. In the following example, you must supply a file name in place of "File-spec" in the following example: `$ TYPE File-spec`

Note

Whenever possible, commands, file names, logicals, and other system data, appear in a different font within a code example to distinguish them from their text description.

Ellipsis (. . .)	Horizontal ellipses indicate that you can enter additional parameters, values, or information. For example, you can enter any number of file specifications in the following example:
	`$ TYPE file-spec, . . .`
	Vertical series of periods or ellipses mean that not all of the data that the system would display in response to the particular command is shown, or that not all the data a user would enter is shown.
	`$ TYPE MYFILE.DAT`
	`        .`
	`        .`
	`        .`
	`$`

Square Brackets ([ ])	Square brackets indicate that the enclosed item is optional. (Square brackets are not optional, however, in the syntax of some file specifications.) For example, the logical name is optional in the following command: `$ MOUNT/FOREIGN $TAPE1` Braces indicate that you must select from the included items.
Quotation Marks and Apostrophes	The term quotation marks refers to double quotation marks ("). The term apostrophe refers to a single quotation mark (').

1 Porting Strategy

Overview

Introduction

This chapter describes the strategy, the process, requirements and various factors that influence porting the Alpha AXP Firmware to an Alpha AXP hardware platform.

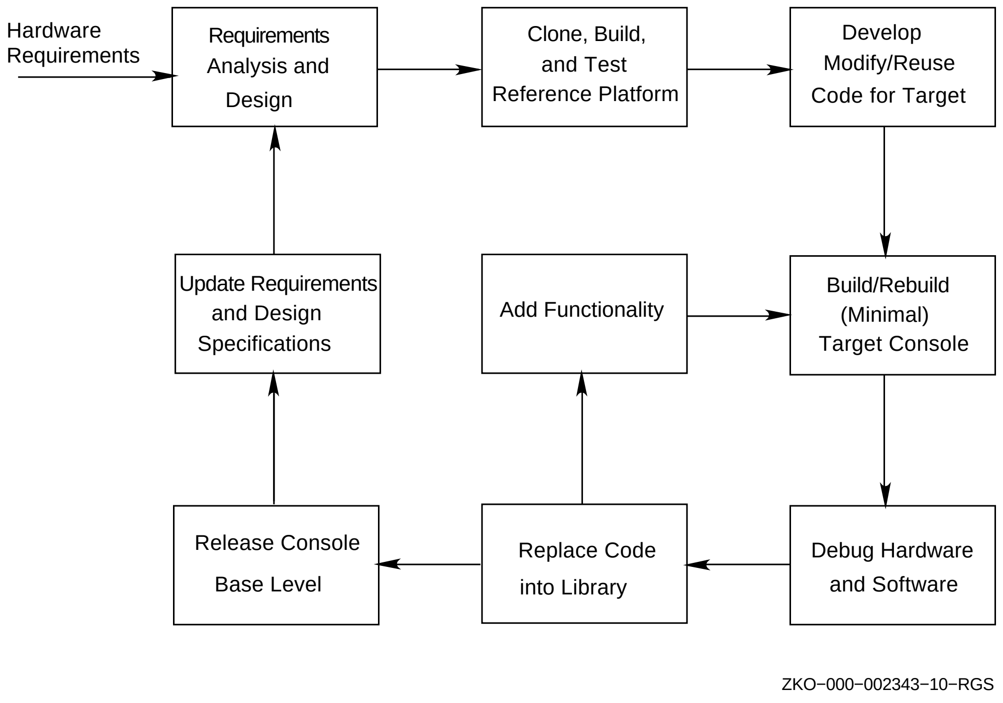
Overall Process of Porting

Overall Process

Figure 1–1 shows the overall process of porting the Alpha AXP Firmware. The details of each step are discussed sequentially in the following chapters.

The hardware requirements of the platform are the input to the analysis and design. Note that once the console is built and tested, console development is an iterative process of adding layers of functionality, rebuilding, debugging, and then replacing code back into the library source pool.

Figure 1–1 Porting Process


Hardware Requirements
Requirements Analysis and Design
Clone, Build, and Test Reference Platform
Develop Modify/Reuse Code for Target
Update Requirements and Design Specifications
Add Functionality
Build/Rebuild (Minimal) Target Console
Release Console Base Level
Replace Code into Library
Debug Hardware and Software
ZKO−000−002343−10−RGS

Porting Scenarios

Overview

The current state of the hardware platform that the Alpha AXP Firmware is being ported to influences the porting strategy used.

The hardware platform may be in one of the following states:

- Currently under hardware development
- Designed, tested and already functional

Hardware Under Development

Hardware is under development when it is a new hardware implementation that has not previously run an operational console. The time needed to port the Alpha AXP Firmware to hardware under development is significant since both the hardware and software must be tested because neither can be assumed to be working properly.

The strategy for porting to hardware under development consists of an iterative process. This process starts with developing a minimal console image, then adding features and functionality incrementally. An example of this follows:

- Ensure the basic hardware is working by using the SROM mini-console to echo characters on the serial terminal and access memory and I/O controllers.
- Build a skeletal image with minimal functionality by disabling the time-of-year (TOY) clock, hardware interrupts, machine checks, and drivers.
- Add functionality incrementally. For example, when adding drivers, add one driver at a time. As each driver is added, initialize the driver first in polled mode and, once it is functional, switch the driver to interrupt mode.

Hardware Functional

Hardware is functional when the basic system, bus, controllers, and devices work properly. The target platform to be ported may be hardware configuration or implementation variation of the basic hardware design that is already operational.

Since a base level console image has already been built and tested for the existing system implementation, the developer can focus on making the changes necessary to support the new requirements, for example, adding or modifying drivers, commands, and test scripts.

Porting Strategy

Reference Platform

The recommended porting strategy starts with the files used to create the Alpha AXP Firmware for an existing system: the reference platform. The reference platform and image is assumed to be a known working system.

When selecting the reference platform, first identify the requirements of the target platform. For example, if the target platform is a large system, it is important for the reference platform to support a comprehensive set of diagnostics. In general, the reference platform should be selected for a hardware implementation that is as similar as possible to the target platform, using an image that has the required commands and features. This is described later in the *Porting Factors* section of this chapter.

Cloning Files

Once a reference model is selected, files and code fragments are cloned to create an image similar to the reference platform.

The cloned image can be tested using the reference platform, and then modified as a series of progressive stages for the target platform.

Building Multiple Console Images

One factor that will influence your porting strategy is the method you choose to configure the console image.

Currently, there are two console image configuration strategies:

- Create a minimal console image that is stored in the ROM, placing additional functionality in images that can be down-line loaded as appropriate.
  - For example, an image can be configured with full diagnostics, appended with a MOP header, allowing it to be loaded by field personnel.
  - This same image can have a complete set of console commands that are not available with the ROM image.
- Create a full console image that is stored in the ROM and use overlays for platform-specific functionality.

Summary of Porting Strategy

The following functions summarize the recommended porting strategy:

- Build a minimal cloned image, lacking drivers, features, and functionality.
- Make minimal modifications to the PALcode and console kernel to allow debugging.
- Add features and functionality, such as drivers, commands, diagnostics, and scripts.

Applying the Strategy

The porting strategy is implemented by performing a specific set of steps, as discussed in the following section.

Process of Porting

Introduction

This section provides an overview of the steps to be followed when porting the Alpha AXP Firmware to a new platform.

Porting Process

Process Steps

The process of porting the Alpha AXP Firmware from a reference platform to a target platform is shown in the table.

Table 1–1 Porting Procedure

Step	Action	Description
1.	Decide on a reference platform from which to clone the PALcode and console software.	Select a platform with a similar DECchip implementation and bus structure.
2.	Clone a new target console image from the reference platform.	Fetch existing platform-specific source files for the reference platform from the CMS library or `CP$REF` directory. Rename the reference platform-specific files to the new target platform name. Do not modify the platform-specific code within these directives until after the image is successfully built and tested.
3.	Build the new console image for the reference platform.	Refer to *Chapter 3*.
4.	Test the new image on the reference platform.	The new target image should run on the reference platform. Testing should include: Network bootstrapping Running diagnostics with command scripts (see the CMS Library: Group TEST_CMD, and the `.CMD` files) Booting the operating system
5.	Modify the code for the target platform.	Make minimal changes to PALcode and console kernel that will let the image execute on the target platform. Changes should include debug tracing of code by echoing characters to show progression.
6.	Build a small console image by removing unnecessary files from the configuration build file.	Eliminate drivers that are unnecessary initially: Ethernet `er_driver.c`, `ew_drvier.c`, Floppy `ide_driver.c`, SCSI `n810.c`, `pke_driver.c`, and so on. Eliminate building (compiling/linking) with XDELTA debugger. Reducing the memory footprint will expedite X-load time and may allow console execution from the backup cache.
7.	Debug the hardware.	Debugging the hardware should include a thorough visual inspection, power and ground voltage checks, clock signal checks, and use of the SROM mini-console to test deposit/examine functions to memory, I/O control registers, checking bus data paths. The SROM mini-console image is placed into an SROM using the `SROM$CONSOLE.MAR` source file. The DECchip provides serial port access using the same signal lines as to load the SROM, once the SROM code is loaded. A level shifter is necessary to convert to EIA voltage levels.

(continued on next page)

Table 1–1 (Cont.) Porting Procedure

Step	Action	Description
8.	Build the image to compile drivers to use the polling method to handle device interrupts.	This is used initially as each driver is added. After basic tests are performed with the driver, hardware interrupts are enabled. Hardware interrupts depend on PALcode and associated interrupt handling routines.
9.	Get the load path working first.	Working with the minimal image, first establish a floppy load path and then establish a network (NI) load path.
		If no I/O subsystem exists, the SROM mini-console can be used to downline load the console image using the SROM serial port.
		The console can also be configured with the `SROM_UART.MAR` driver for communication using the SROM serial port.
10.	Add features and functionality such as commands, test scripts, and bus or port drivers.	Run diagnostics with command scripts and create new test scripts to test any new functions and drivers.

Requirements and the Development Environment

Requirements

To effectively port the Alpha AXP Firmware to the target platform, a developer should first have a working knowledge of the following:

- Build environment
  - Directory structure (build tree)
  - Logicals used during the automated build process and during development
  - Files used to create the target image, as defined in the `*Configuration*_files.bld` configuration build file
- Code base
  - Code Management System (CMS) and its library of routines are optionally used for internal development

A developer should become familiar with the build process and its environment according to the stated strategy: a working model of the Alpha AXP Firmware is built first, then modified for the platform.

General knowledge of the CMS user interface is required to `FETCH` a particular generation of source code from the CMS library. Developers can also use the `CP$REF` directory to locate source files.

Build Environment

Procedures used in the build environment will establish a working directory structure (build tree), assigning logicals and build definitions. Logicals can locate the platform-specific build files that need to be modified. The `CP$REF` directory contains a copy of the latest replaced elements in the CMS library.

Code Management System

CMS is a software management and maintenance tool that tracks changes and acts as a repository. A developer can access source routines, (called elements), make changes, create elements, group them, insert specific generations into classes, and replace them back into the software library.

Reference

For more information, refer to:

- *Chapter 3, Cloning, Building and Testing the Reference Image*
- *Appendix B: Getting Started with CMS*

Porting Factors

Factors Affecting Effort

A number of factors can affect the extent of the effort needed to port the Alpha AXP Firmware to a new platform. The following list weights these factors in order of increasing effort:

- Similar hardware platform using the same DECchip implementation
- Similar platform using a different DECchip implementation
- Different platform

Similar Platform

A port to a similar hardware platform requires less effort because there are fewer modifications to make to those files. For example, if the new target platform has a similar I/O bus structure, the existing drivers can be used, possibly some without modification.

Different DECchip

Porting to a system that uses a different DECchip may require additional effort. The Alpha AXP architecture is flexible enough that different implementations of the DECchip may have additional internal processor registers, additional circuitry, such as secondary cache, or I/O control. There are five PALcode instructions that are hardware implementation specific that may change as well.

The DECchip 21066 has different internal processor registers (IPRs) used for integrated memory/backup cache and I/O controllers. PALcode must be modified since it writes and reads these IPRs. Serial ROM (SROM) code must be modified to initialize these registers. Other differences in the various DECchip implementations can affect PALcode, for example, the number of interrupts. However, a number of platforms use these various DECchips and have ported the Alpha AXP Firmware.

Different Platform

Porting to a different platform would involve modifications to SROM, PALcode, and other source code routines, as well as creating new drivers and new routines for the PALcode and SROM. The SROM and PALcode must have knowledge of the system buses, memory, cache, and I/O, to properly initialize and test those buses and system.

Operating System

The operating system and the console callback routines are another consideration when porting. The Alpha AXP Firmware bootstraps and supports the following operating systems:

- OpenVMS
- DEC OSF/1

The Alpha AXP Firmware may provide testing and setting of variables to allow loading of an Advanced RISC Computing (ARC) compliant console, upon the next reset.

Using a Reference

The amount of time to implement the port can be reduced by using an existing platform as a reference. Locating existing code routines and making the necessary modifications, minimizes the amount of code that needs to be created, and is efficient code reuse.

The model port discussed throughout this book begins with a reference platform image that is built and tested. Modifications are kept to a minimum at first to minimize debugging. Drivers and much of the console functionality are disabled, being added later in layers.

2
Console Overview

Overview

Introduction

This chapter discusses the console image, its interfaces, functionality, and how it initializes the system.

Console Image

Code Sections

The Alpha AXP Firmware consists of three separate pieces of code:

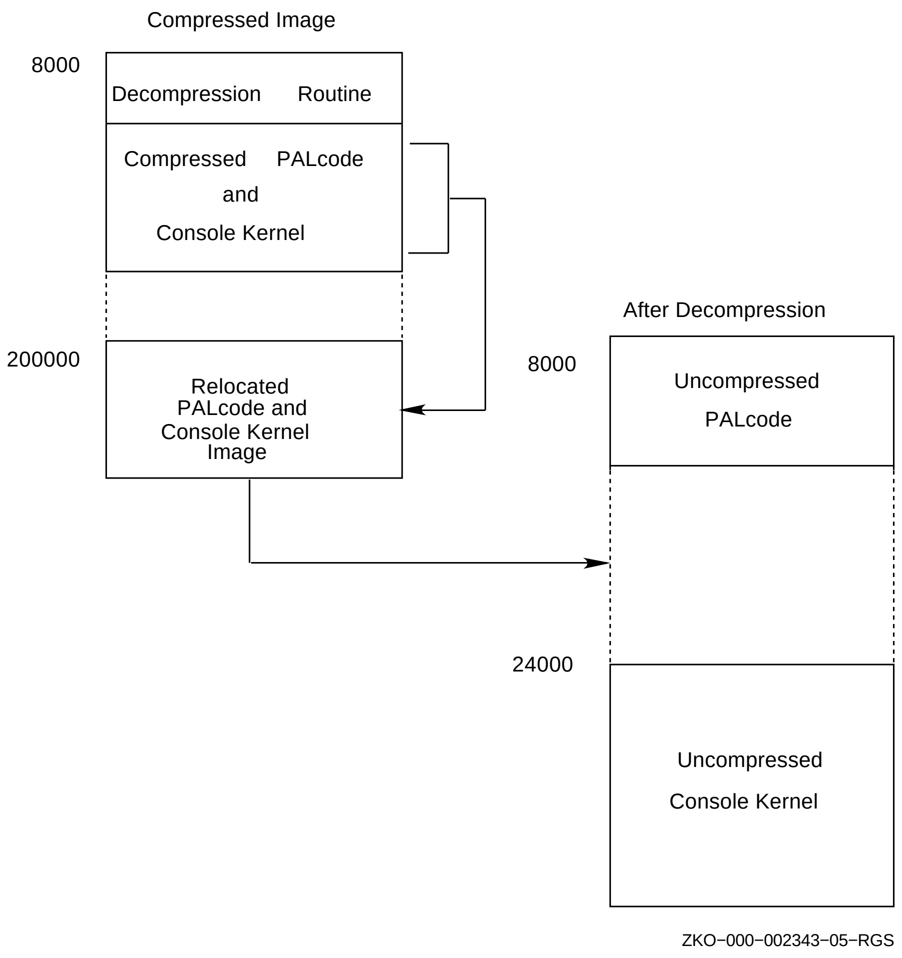
- Decompression program
- PALcode
- Console kernel

The image is a combination of compressed PALcode and console kernel, and the decompression program, which is uncompressed.

Loading the Image

Figure 2–1 shows that the SROM loads the combined compressed image and decompression program starting at address 8000_{16}. Control is then passed to the decompression routine, which relocates the entire image to upper memory, (approximately 200000_{16}). The PALcode and kernel are decompressed and relocated back to lower memory, beginning at address 8000_{16}. The console kernel is typically linked with a starting address of 24000_{16}.

Figure 2–1 Console Image

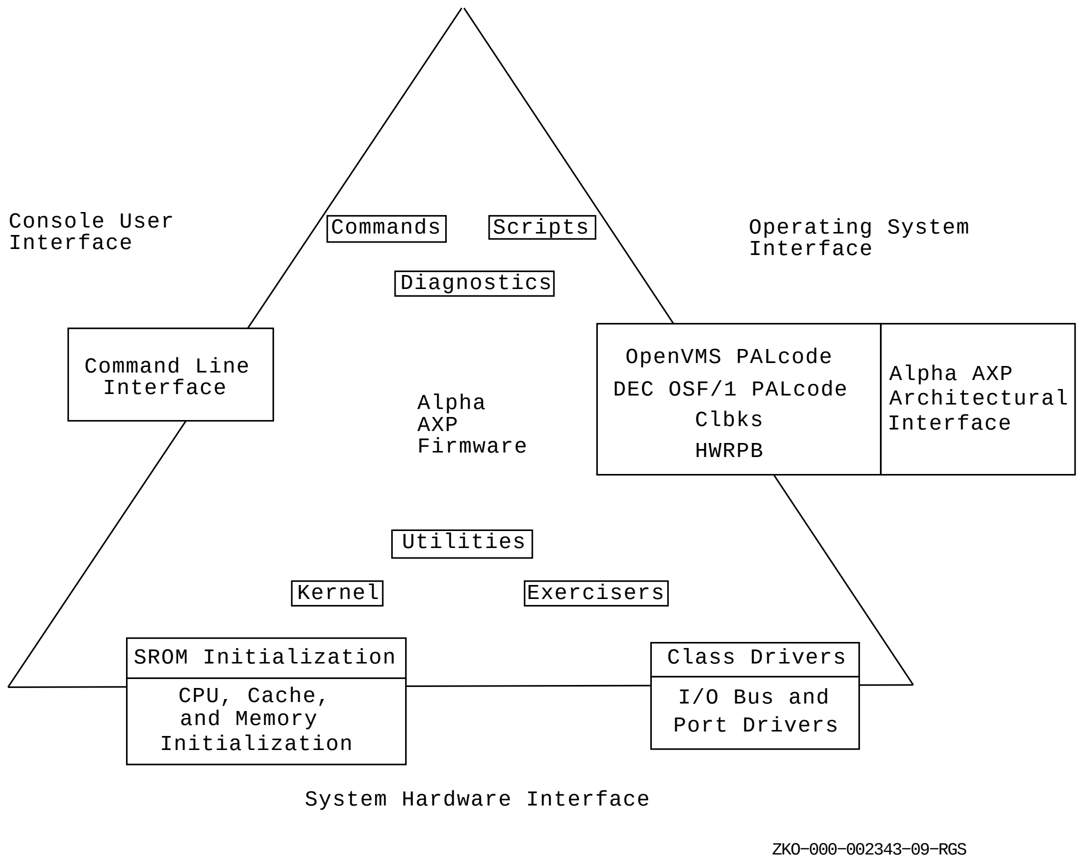


Console Interfaces

Introduction

The Alpha AXP Firmware provides a uniform interface to different operating systems, hardware platforms and console terminal devices.

Figure 2–2 Alpha AXP Firmware Interface



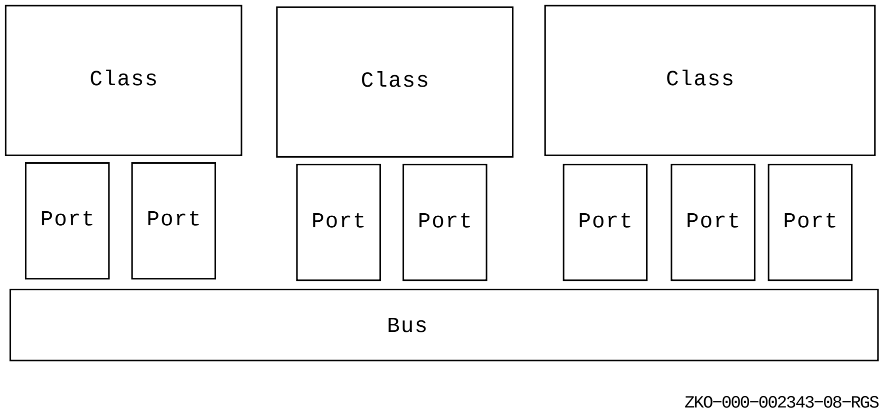
Hardware Interface

The Alpha AXP Firmware provides an interface to the system hardware. The SROM code is executed initially at **powerup** (when the system is turned on) and when reset. The SROM is responsible for initializing and testing enough of the system to load and start the console image. The console provides a set of drivers which are used by the console, diagnostics, bootstrap, and operating system callbacks to access various devices.

Driver Model

The driver model supported is based on the class, port, and bus model. This separation of driver code simplifies maintenance, and the addition of new device support.

Figure 2–3 Class, Port, and Bus Driver Model



A class driver functionally interfaces with an application's request for input/output, and with the port driver to fulfill that request with the hardware. The bus driver has knowledge of the entire system bus and path to uniquely identify the physical location of a device for data access.

There is one class driver per device class, such as the SCSI class driver, and the network protocol class driver. However, there may be one or more port drivers and associated hardware controllers for that device, for example, the NCR53C810 and Adaptec 1740 SCSI controllers, and the DECchip 21040 Ethernet LAN controller for the PCI. There is only one bus driver per system.

Reference

For more information about drivers, refer to the *Device Driver Interface Guide for the Alpha AXP Firmware*.

User Interface

The second external interface is the console user interface. Users, operators, and customer service representatives communicate with the system through a system console device and the Alpha AXP Firmware.

The console presents the user with a flexible command-line interface (CLI). The CLI is a UNIX style shell of commands, operators, and a scripting facility that can be fully configured. It provides the ability to:

- Initialize and test the system
- Examine and alter system state
- Boot the operating system

Configuring Commands

The console can be configured to provide a rich set of commands, or a less extensive set when the image is built. This depends on the preference of the developer and system requirements: functionality and image size.

Reference

Refer to *Appendix G: Basic Console Commands* for a list of the essential commands.

For debugging purposes, the console can be configured with the `SROM_UART.MAR` driver for terminal communication using the SROM serial port.

Operating System Interface

The third external interface is for the operating systems.

The console is compliant with the *Alpha System Reference Manual* (SRM), providing callback support for the OpenVMS and DEC OSF/1 operating systems.

Functionality

Introduction

This section describes the functionality of the Alpha AXP Firmware and its initialization process.

Console Functionality

Introduction

The Alpha AXP Firmware and PALcode provide functionality for implementing and controlling hardware functions, such as:

- Initializing the hardware system
- Diagnostic testing of the system for proper operation and report errors
- Bootstrapping the operating system
- A user interface for monitoring and controlling the system
- Callback services to the operating system, which simplify operating system control of, and access to, system hardware
- Controlling and monitoring the state of each processor in a multiprocessor configuration

PALcode Functions

PALcode is the Privileged Architectural Library for the Alpha AXP architecture that provides any operating system access to low-level Alpha AXP chip functions, atomic operations and primitives. In addition, PALcode can simulate complex hardware instructions.

Console Functionality and Porting

For each area of functionality, there are components that are specific to a hardware platform.

Subsystems that may have to be modified for a specific platform and the corresponding portion of the console that must be customized include:

- Chip Functionality - PALcode
- Cache and Memory - Serial ROM (SROM) code and PALcode; SROM code initializes the DECchip during hardware reset
- Devices - Device Drivers, Boot Device Parameters
- Input/Output - Platform-Specific Powerup Process
- Interface to the Operating System - Hardware Restart Parameter Block (HWRPB)

Console Initialization

Introduction

During console initialization, the console kernel:

1. Starts the console initialization process
2. Initializes all remaining registers to their default values
3. Builds and loads the Process Control Block (used by the console for unexpected interrupt handling)
4. Starts the timer process
5. Starts the powerup process

A prompt is displayed on the console terminal device at the end of console initialization.

Powerup Process

The following functions describe the powerup process. However, its sequence is platform specific:

- Initialize the file system.
- Configure memory with the `memconfig` routine, (this may happen earlier in SROM code).
- Build the Hardware Restart Parameter Block (HWRPB) and related data structures at the beginning of good memory.
- Set up its initial stack space and initial heap, which is a block of memory used by the console routines. Stack space for the console is set up there.
- Initialize drivers.
- Run self-tests.
- Start a kernel entry process.
- Start the shell.
- Run a specified platform-specific powerup script.
- Dispatch to the appropriate function based on console state.

Reference

For more information, refer to the `kernel.c` file.

3 Cloning, Building and Testing the Reference Image

Overview

Introduction

This chapter describes how to perform the first step in porting the Alpha AXP Firmware to a new platform; cloning, building and testing the reference image.

Reference

The following notes conferences and documents provide additional information about porting and building a console image:

- *Common Console Development Environment User's Guide*, by Peter H. Smith, July 29, 1993.
- *Cobra Console Build Process*, January, 1993.

Overview of Cloning Files

Introduction

A developer must modify files to build a cloned console image from a reference platform. The majority of modifications are related to name changes, not functionality changes.

This chapter summarizes the files to be modified when building and testing a cloned console image.

Model Port

The model represented in this chapter, along with examples of files modified, were to port the console to an Alpha AXP PC target platform. The model clones an image from a previous Alpha AXP PC design. However, the model's design incorporates a different DECchip implementation for the microprocessor, the DECchip 21066, and an integrated Ethernet controller.

DECchip 21066

The DECchip 21066 is an implementation of the Alpha AXP architecture that differs from the DECchip 21064. The DECchip 21066 memory and I/O bus structure are different than that of the DECchip 21064. The DECchip 21066 has integrated DRAM memory and PCI I/O bus controllers, and includes embedded graphics accelerator support for connection to a video RAM (VRAM) frame buffer.

These added features affect the port in the following way:

- PALcode. There are differences in internal processor registers. PALcode is the privileged code that can read and write internal registers.
- Bus, port, and class drivers. The system and I/O bus are architecturally different, together with certain I/O controllers.

Cloning and Modifying the Reference Source Code

Introduction

This section describes the following:

- Development environment
- Directory tree structure
- Engineering environment
- Cloning files
- Cloning and modifying source files
- Cloning and modifying code fragments
- Cloning build files
- Modifying build file

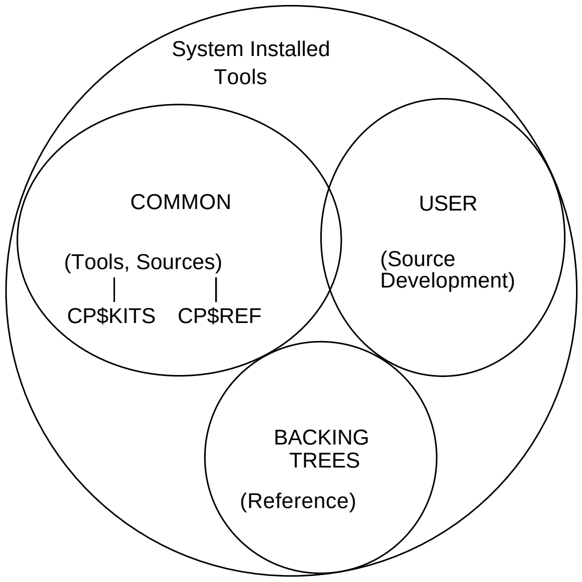
Development Environment

Overview

The development environment consists of tools and source programs required to build an Alpha AXP Firmware image. These tools and source code pool are mutually shared, existing in a logically defined directory structure. Developers must execute a command procedure that properly defines `CP$XX` logicals, development directories, directory search lists, directories of tools and sources and other information required to create the image.

The procedure `CP_COMMON_LOGIN.COM` constructs this environment, and is the first phase of the image build procedure. Parameters are included on the command line to specify options for the build, for example, the name of the top-level directory, under which a directory tree is created for a work area and to support building the image.

Figure 3–1 Major Components of the Development Environment



Major Components

Figure 3–1 depicts the major components of the build environment:

- System installed tools

  The normal system utilities and editors installed on the system would be used by developers, along with other preferred tools, such as a C compiler.

- Common area - tools and sources

Tools not installed on the system can be found in the CP$KITS directory. Some examples of tools that are shared are: command procedures, AWK scripts, MACRO64 and PALcode assembler, GEM C compiler, MMS description files, and utilities.

The CP$REF logical references the source code area where C and Macro kernel routines, drivers, and other source files, such as command procedures, can be found. The CP$REF directory mirrors the latest source code elements placed into the CMS library.

- User area - directory and source development

  This is the developer's working area, known as the directory tree. It contains sources developed for the target port, the final image, and other supporting or intermediate files.

- Optional backing trees - reference files

  The backing trees are directories containing a snapshot version of source code, object files, and configuration information. They are optionally used as a secondary source, (or backed against), during the image build process. By default, files in the user's working directory are backed against files in the CP$REF directory.

  Backing trees are helpful to establish a reference, provide isolation from other development efforts, and reduce image build time.

Save Sets

The Alpha AXP Firmware development environment can be replicated on a system that uses the OpenVMS operating system by copying the following three backup save sets:

- Source module save set derived from CP$REF
- Tools and utilities derived from CP$KITS
- User working area derived from the directory tree, contains a pre-built console image

Directory Tree Structure

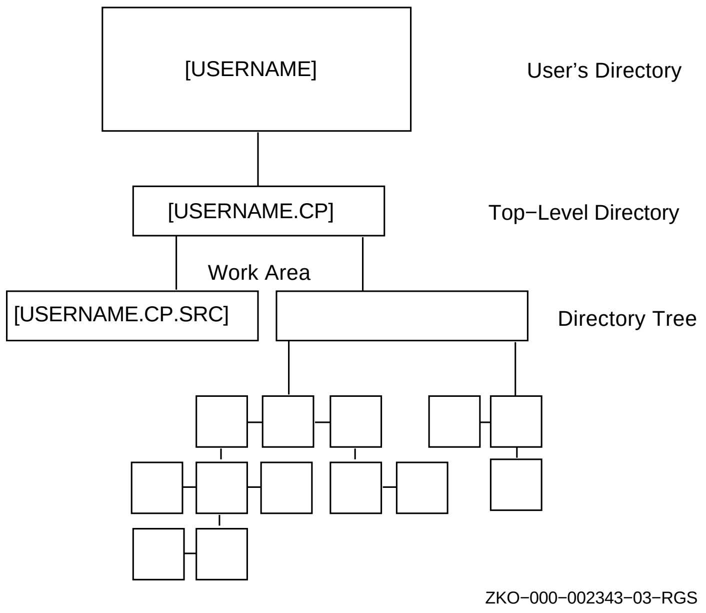
Working Directory

A developer specifies the top-level directory to be used with a parameter to the `CP_COMMON_LOGIN.COM` procedure, which creates the development/build environment.

A directory tree is established under the top-level directory specified, creating subdirectories if necessary, and is used as a work area. The directory `[USERNAME.CP.SRC]` contains source files modified for the target port. The rest of the directory tree contains files to support building the target image, intermediate files, executable files, and the final image.

A subdirectory of a user's account should be specified for the top-level directory to separate the directory tree and related files from the user's files. Figure 3–2 shows the top-level directory of `[USERNAME.CP]`. If no top-level directory is specified, the directory tree is created under the current (default) directory.

Figure 3–2 Directory Tree

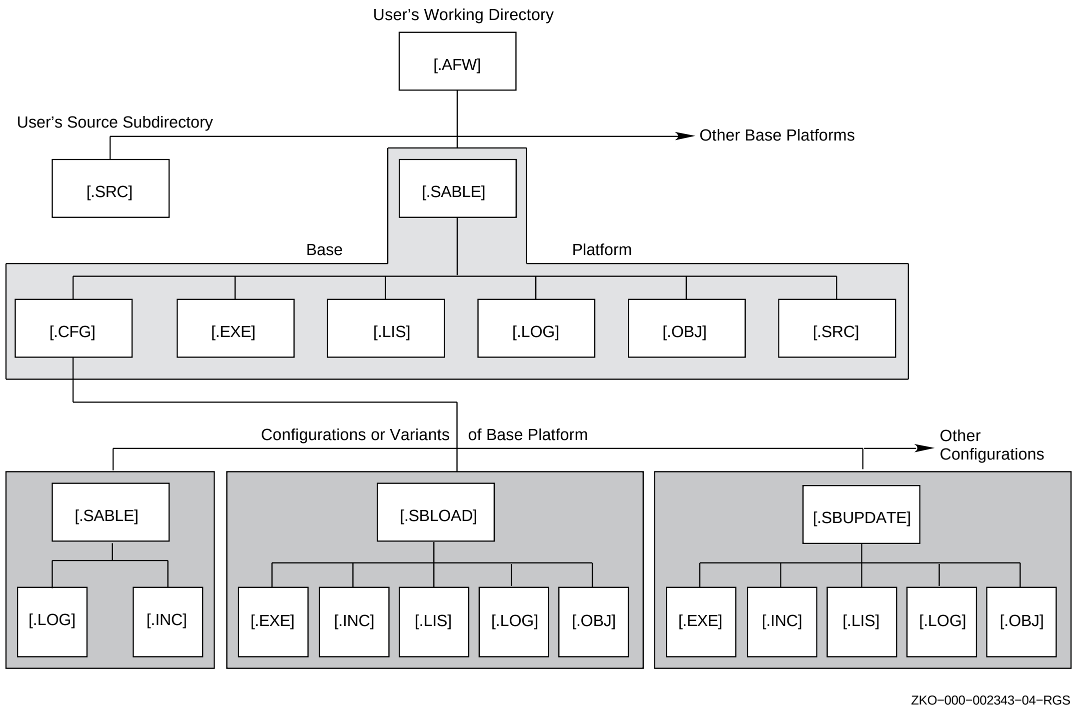


Configuration Directories

The directory tree consists of a number of subdirectories, created under the user-specified working directory, as shown in Figure 3–3. These subdirectories are grouped into two distinct sections in Figure 3–3:

- Platform base
- Configurations (or variants) of the platform base

Figure 3–3 Directory Tree, Configurations



Note that the user's source (working) subdirectory is also shown in Figure 3–3, and is located at the same directory level as the platform base. This is used by the CP$SRC logical search list to locate source modules.

Reference

Information about the logical search lists can be found in: *Appendix A: Logical Search String Definitions*.

Platform Base

The platform base directory is the first directory created, and is named after the hardware platform to be ported. It contains files used to create a base level image for the target platform.

These files are common to all other configurations (variations) created for that same platform. For example, many source files can be used for different image variations that are configured for the application. An example variation is: a small, FEROM-based image that contains a subset of commands and functionality. This sharing of common build files eliminates redundancies and reduces the total disk storage as configurations are added.

The platform's base subdirectories and contents are:

- `[.src]`, source files for the platform
- `[.obj]`, object files
- `[.lis]`, listing files
- `[.exe]`, executable files
- `[.log]`, log files from various build procedures
- `[.cfg]`, contains the configurations and their subdirectories, for example, `[.SABLE]` `[.SBLOAD]`, and `[.SBUPDATE]`

Configurations

Configurations are variations of the hardware base platform image. A developer can configure the image to vary in functionality to requirements. Each configuration has its own subdirectory under the base platform. These subdirectories contain the configuration files generated during the second phase of the build process: from the `NEWBUILD.COM` build procedure. Three typical configurations are shown in Figure 3–3:

- `[.SABLE]`
  - A platform base configuration of Alpha AXP Firmware for the Sable hardware platform. The base configuration is the same name as the hardware platform and is the FEROM-based image.
- `[.SBLOAD]`
  - A super-set of the Sable base image, containing an extensive command set, diagnostics, and exercisers. This image is typically loaded with MOP or BOOTP protocol (not a FEROM-based image).
- `[.SBUPDATE]`
  - An image used to update the Flash EROMs (FEROM) in the field.

Configuration Build Files

Each configuration has its own build file: *platform*_files.bld. A developer creates this file, customizing it to build a target image for the particular configuration and hardware platform. The configuration build file specifies source files, the driver startup order, and compile options, for example:

```
SABLE_FILES.BLD
```

The SABLE_FILES.BLD file creates a configuration with the name of Sable, and is the platform base (configuration).

Configuration Subdirectories

All configurations have at least two subdirectories:

- [.INC], included header files created during the build
- [.LOG], log files for that configuration

Note that in Figure 3–3, configurations other than the base platform have additional subdirectories created for that configuration:

- [.EXE], executable images
- [.LIS], listing files
- [.OBJ], object files

Logicals can locate files in the directory tree. For example, the CP$LOG logical can locate the log files:

```
"CP$LOG" = "CPUSER:[USERNAME.AFW.SABLE.LOG]"
    = "CFW:[CONSOLE.FRIDAY.SABLE.LOG]")
```

Reference

For more information about logicals defined for the development environment, refer to *Appendix A: Logical Search String Definitions*.

Creating the Environment

The configuration login procedure creates the development environment. It is invoked manually or automatically by adding the following command to the user login.com file:

```
$ @ALPHA_FW:[COBRA_FW.REF]CP_COMMON_LOGIN.COM Configuration -
  Directory -
  Backing_tree
```

Table 3–1 cp_common_login.com Parameters

Parameter	Description
Configuration	Name of the configuration image. If this parameter is omitted, only logicals are defined.
Top_dir	Name of the top-level directory under which the directory tree is constructed.
Backing_tree	Name of the backing tree, or a list of backing trees, that can be used to reference the build against (using precompiled code to save compilation time). Also, CP$REF can be specified to force a complete build. If this parameter is omitted, the last backing tree specified is used.

Reference

The CP_COMMON_LOGIN.COM procedure parameters and their implementation details are discussed later in this chapter in the *Building the Reference Console* section.

Engineering Environment

Engineering VMScluster

Access to the compute server for the engineering software VMScluster can be seen in the `$ SHOW CLUSTER` example.

```
$ SHOW CLUSTER

+--------+----------+---------+
|  NODE  | SOFTWARE |  STATUS |
+--------+----------+---------+
| MAY21  | VMS V5.5 | MEMBER  |
| MAY31  | VMS V5.5 | MEMBER  |
| DUFFY  | VMS V5.5 | MEMBER  |
| EGRESS | VMS V5.5 | MEMBER  |
+--------+----------+---------+
```

System Load

A `$ LOAD` command can test system loading conditions for optimum response from the compute server.

```
$ LOAD
                                            Tot  Free
Node    Rating  Load  VUPs CPUs Idle  Mem  Mem   CPU type
----    ------  ----  ---- ---- ----  ---  ----  --------
DUFFY::   2071   0.9  24.0   1   36%  96   45.2  VAX 4000-500
EGRESS::     0   0.3    ?    1   74%  192 156.0  VAX 4000-700A
MAY21::   2414   0.8  24.0   1   40%  96   54.2  VAX 4000-500
MAY30::      0   1.0    ?    1   94%  128  83.9  VAX 4000-700A
Best node is MAY21
$
```

Cloning Files

Introduction

To begin a port, files can be cloned from existing files in the CMS library or the CP$REF reference directory. The reference directory contains a copy of the latest files replaced into CMS.

Categories of Files to Clone

Files that need to be cloned can be categorized as follows:

- Platform-specific
- Build
- Kernel
- Driver

Cloning and Modifying Platform-Specific Files

Introduction

Table 3–2 and Table 3–3 show that platform-specific files can be cloned from other platforms. Modifications needed to build the reference console require replacing the old platform name in the file name with the new platform name.

Source Files to be Cloned

Specifically, the files that need to be cloned concern the following:

- Memory Configuration
- Initialization Routines
- Macro Utilities for Character I/O to the Combination Chip Serial Port (COM1, COM2)
- Powerup Routines

Cloned Source Files Example

Table 3–2 shows the source files that were cloned from another PC platform.

Table 3–2 Cloned Platform-Specific Source Files from Another PC Platform

File	Description
memconfig_platform.c	Configures and tests memory, diagnostics; checks and marks bad pages
platform.c	Platform-specific utilities: initialization, bus sizing, bus read/write routines, halt, L.E.D., and dump routines
platform_util.mar	Platform-specific MACRO routines: serial I/O, load/store, and CSR read/write routines
powerup_platform.c	Starts drivers at proper phase: memory, PCI, tt, builds HWRPB, tests memory, creates null and shell process

Cloning and Modifying Code Fragments

Overview

Some files are used for multiple platform builds. These files should remain in a common source pool.

Any code contained in these common files that must be platform specific is conditionalized so that it is used for a specific platform build only.

Modifications must sometimes be made to these files to account for the specifics of a given platform. To do this, the conditionalized code must first be cloned within the file. The new platform name must be placed in the condition statements and then, at some later time, the code should be modified for the new platform.

Note

Modifications must be carefully made to avoid altering the build and possibly keeping the code from executing for other platforms.

Changing Conditional Statements

Preprocessor directives are changed in the platform-specific code sections to reflect conditional compilation for the target. No functional changes should be made to the code until a working image is built and tested. Example 3–1 shows an example of this as annotated by the callouts.

Example 3–1 Modifying Preprocessor Directives

```
                  /*********************************************
                   * Substitute reference platform directives *
                   *********************************************/
#if Reference_Platform_Name ❶
.
.
#endif
                        /***********************************
                         * With target platform directives *
                         ***********************************/
#if Target_Platform_Name ❷
.
.
.
#endif
```

These files can be retrieved from the CMS library, or from the CP$REF reference directory.

List of Files

An example of a set of files that contained conditional code that was cloned and modified is as follows:

- Build
  - `build.com`
  - `options.bld`
  - `pal_descrip.mms`
- Kernel
  - `call_backs.c`
  - `callbacks_alpha.mar`
  - `entry.c`
  - `ev_action.c`
  - `filesys.c`
  - `hwrpb.c`
  - `kernel.c`
  - `kernel_alpha.mar`
  - `net.c`
  - `nvram_def.h`
  - `startstop.c`
  - `timer.c`
- Driver
  - `combo_driver.c`
  - `eisa_driver.c`
  - `eisa.h`
  - `er_driver.c`
  - `ide_driver.c`
  - `ipr_driver.c`
  - `kbd_driver.c`
  - `mop_driver.c`
  - `n810_driver.c`
  - `pke_driver.c`

Reference

For more information, refer to *Appendix C: Complete List of Modified Files*.

Cloning Build Files

Introduction

Table 3–3 shows the build files that were cloned from another PC platform.

Table 3–3 Cloned Build Files from Another PC Platform

File	Modification Notes
platform_files.bld	Creates a new image for the target platform: PLATFORM
msload_files.bld	Creates an image that can be downline MOP loaded
platform_platform.mms	Combines and creates the compressed image from the following: • Decompression routine • PALcode • Alpha AXP Firmware for the target platform

Build Files

Configuration build files specify the source files and other information necessary to build the target image, such as option and value statements that are platform specific.

The platform-specific MMS file assembles, compiles, links, and compresses the final image that will be written into the EE Flash ROM.

In this case, the use of two configuration build files created two different Alpha AXP Firmware images. One build file (`platform_files.bld`) creates a ROM image, the other (`msload_files.bld`) creates an image that can be downline loaded using MOP protocol.

Modifying Build Files

Configuration Build File

The configuration build file *platform*_files.bld is one of the files used to create the image for the target, along with other command procedure and script files.

A developer would clone this build file.

Modifying the Configuration

Example 3–2 shows how to change the option statement within the platform's configuration build file. Note that Example 3–2 clones platform_files.bld, changing the configuration for the Platform platform ❶, to the *XYZ* platform ❷.

Example 3–2 Modifying a Cloned Build File

```
                 /***************************************************
                  * Substitute reference platform option statements *
                  ***************************************************/

# file: platform_files.bld --  small, ROM, build file for Platform.

platform PLATFORM ❶
architecture ALPHA


                 /******************************************
                  * With target platform option statements *
                  ******************************************/

# file: XYZ_files.bld --  build file for new XYZ target.
#
#       small image, ROM version

platform XYZ ❷
architecture ALPHA
```

Building the Reference Image

Introduction

This section describes building the reference console image.

Building the Reference Console

Three Phases

There are three basic phases to building an Alpha AXP Firmware image:

1. Create the build environment with `ALPHA_FW:[COBRA_FW.REF]CP_COMMON_LOGIN.COM`
2. Build the intermediate files with `CP$SRC:NEWBUILD.COM`
3. Assemble, compile and link the final image with `CP$SRC:DESCRIP.MMS`

Build Steps

These phases consist of the steps shown in the table.

Table 3–4 Building the Alpha AXP Firmware

Step	Action	Description
1.	Copy files that can be modified for the target from the `CP$SRC` directory to your working subdirectory `[.SRC]`. [1]	Developers should use the working subdirectory to modify and create files for the target platform.
2.	Modify files for the target, ensuring that any modified or newly created source code files are in the working subdirectory `[.SRC]`.	The user's subdirectory is the first directory searched for sources during the build.

Note

Do not modify files in the `CP$REF` directory, or in the backing tree directories.

Step	Action	Description
3.	Add new dependent source files and other definitions to the configuration build file.	This file defines a particular configuration of a hardware platform, (a target's source files, build options). This file should reflect the sources in the CMS library, `CP$REF`, or files in the working subdirectory `[.SRC]`. For example, `SABLE_FILES.BLD` builds the base configuration for the Sable hardware platform. A template can be found in the CMS directory.

[1]Alternatively, for a known stable environment, (for example during powerup/debug), fetch sources from CMS GROUP CURRENT into your local working (source, [.SRC]) subdirectory. Another option is to fetch a specific source code generation from a previously defined CMS CLASS into [.SRC].

(continued on next page)

Table 3–4 (Cont.) Building the Alpha AXP Firmware

Step	Action	Description

4.	Execute the DCL command procedure:
	
	`$ @CP_COMMON_LOGIN.COM -`
	`P1 -`
	`P2 -`
	`P3`
		This defines the build environment; logicals and directory structure. Optional parameters define the following:
		
		• P1 = configuration variant of base platform, or base platform [2]
		• P2 = the top-level working directory (build area)
		• P3 = a backing tree for an alternate source reference
		
		Backing trees are used as a secondary reference source for the build. Target dependency files may be sourced from the backing tree, expediting the build, and or, built from a known reliable source. Multiple backing trees can be specified to ensure the search order.
		
		For example:
		
		`$ @ALPHA_FW:[COBRA_FW.REF]CP_COMMON_LOGIN.COM -`
		`  Sable -`
		`  [USERNAME.AFW] -`
		`  Friday`

5.	Execute the DCL command procedure for the target hardware platform:
	`$ @CP$SRC:NEWBUILD.COM`
	`Platform`
		This scans all source files (described in *Configuration*`_files.bld`), creating .H, .MAR, .C, and other dependencies (needed by MMS in the next step), using AWK script text filters.
		
		A target of the base platform is specified. For example, using SABLE as the target:
		`$ @CP$SRC:NEWBUILD.COM SABLE`

[2]Note that a base platform must be the first configuration built before any other configurations. The base platform or alternate configuration is specified in parameter P1.

(continued on next page)

Table 3–4 (Cont.) Building the Alpha AXP Firmware

Step	Action	Description
6.	Create and execute the MMS description file for the target hardware platform: `$ MMS/DESC=CP$SRC:DESCRIP.MMS Platform`	The MMS description file creates the executable Alpha AXP Firmware image for the specified platform by assembling, compiling, and linking the target image. The developer creates the platform MMS file, for example, `platform_sable.mms`. The target images are defined within this platform description file: • PALcode, which has its own description file, `PAL_DESCRIP.MMS` The PALcode description file `PAL_DESCRIP.MMS` is executed by the platform-specific description file. It describes the target PALcode image and its dependencies. The PALcode image `EV4P3_SABLE.EXE` for the Sable platform can be found in the `CP$EXE` directory. • Console kernel and drivers • Decompression image To create the image `CP$EXE:SABLE.EXE` for the hardware platform SABLE: `$ MMS/DESCRIPTION=CP$SRC:DESCRIP.MMS SABLE`
7.	Replace modified or newly created source code files that have successfully built and tested into the CMS library group = CURRENT.	A copy is also placed in the source reference directory `CP$REF`, the default build directory.

Note

New files must be created, then inserted into the CMS group CURRENT. Modified files can be directly replaced.

(continued on next page)

Table 3–4 (Cont.) Building the Alpha AXP Firmware

Step	Action	Description
8.	Execute the DCL command procedure: `hardware platform_make_rom.com`	Creates compressed and other images in CP$EXE by combining the PALcode and Alpha AXP Firmware executable image: • `CFW_SBROM.EXE`, compressed, checksummed image padded to 512K bytes • `CFW_SBROM.SYS` MOP [3] image that can be downline loaded • `CP$EXE:CFW_SBROM_E32.HEX` and `CP$EXE:CFW_SBROM_E34.HEX`, files are used during manufacturing and programming of the Flash EEROMs This step is normally done only for a software release to create the hex files. The following example creates a compressed image for the Sable hardware platform: `$ @CP$SRC:SABLE_MAKE_ROM.COM`

[3]MOP = maintenance-oriented protocol for IEEE 802.3 communication and file loading.

Configuration Log Files

The log files for various configurations can be found in the backing tree directory under the platform's [.LOG] directory, for example:

```
$ DIR CP$LOG:

Directory CFW:[CONSOLE.FRIDAY.SABLE.LOG]

SABLE_BUILD.LOG;1   SBFSAFE_BUILD.LOG;1 SBLOAD_BUILD.LOG;1  SBMIN_BUILD.LOG;1
SBUPDATE_BUILD.LOG;1
```

Note

This applies only to the nightly builds that are done on the engineering cluster; otherwise refer to the user default directory and the appropriate batch log files.

Testing the Reference Console

Introduction

Once the reference console is built and a console prompt is displayed on the console terminal, the reference console should be tested.

This section describes the following:

- Using the Console Interface
- Running Diagnostics
- Bootstrapping
- Recovering from Errors During Bootstrapping
- Identifying the Bootstrap Device
- Initializing the Boot Device
- Booting Over the Network

Using the Console Interface

Introduction

The console supports a set of common commands useful for system configuration and operating system bootstrap.

Commands Description

The default set of console commands for Alpha AXP platforms include commands for:

- Booting the operating system
- Accessing or modifying information about the system
- Initializing programs and processors
- Testing the system

Introduction to Console Commands

The console interface consists of a console prompt, console shell interface and console commands.

Console Prompt

The system is by definition "halted" or in "console mode," whenever the firmware is executing.

When halted, the firmware communicates with an operator through the device designated as the system console.

The firmware delivers the following prompt on the console terminal indicating that it is awaiting command input.

```
>>>
```

Console Shell

The console shell is a command-line interpreter that is a subset of the Bourne shell.

The console shell is very flexible because it supports traditional UNIX functions such as pipes, I/O redirection, command-level scripting and control functions.

Built around a multitasking kernel, the console provides an excellent environment for support of much more complex functions, such as system exercisers, MOP listener and remote console.

Commands

The following table contains a list of tasks you can perform using the commands that are common to all Alpha AXP platforms.

Table 3–5 Console Commands

To	Use	Function
Get Information About the System	`show`	Displays the current value for an environment variable and other system information, for example, `SHOW CONFIG` displays information about the system and devices.
Get Online Help	`help, man`	—
Access Data	`examine, deposit`	These commands act on byte streams. The console manipulates these byte streams by performing typical device operations - open, read, write, close. Device refers to any such byte stream or address space regardless of its actual physical implementation. The default device is physical memory. The console provides the following device drivers for Alpha AXP devices: • pmem: - physical memory • vmem: - virtual memory • gpr: - general-purpose registers • fpr: - floating-point registers • ipr: - internal-processor registers
Access Memory	`alloc`	Before randomly experimenting with memory, it is important to find a "safe" area in memory to alter. Since the console itself and other critical data structures reside in memory, care should be taken not to alter them. The `alloc` command can allocate a "safe" area in memory.
Boot the Operating System	`boot`	The `boot` command initializes the processor, loads a program image from the specified boot device and transfers control to that image. If you do not specify a boot device in the command line, the default boot device is used.
Resume Program Execution	`continue`	Continues execution on the specified processor, or the primary processor if one is not specified. The `continue` command is valid only if an operator has halted the system by one of two methods: either by pressing the Halt button on the control panel or by entering Ctrl/P on the console terminal.
Initialize the Console	`initialize`	Initializes the console, a device, or the specified processor. If a processor is not specified, the primary processor is initialized.
Set the Value of an Environment Variable	`set`	Sets or modifies the value of an environment variable. Environment variables pass configuration information between the console and the operating system. Environment variables that can be set or modified include `auto_action`, `bootdef_dev`, `boot_file` and `boot_osflags`. Additional environment variables control execution of diagnostics. A default value is associated with any variables stored in NVRAM. This default is used if the variable is not set, or if NVRAM is unreadable.
Start a Program	`start`	Starts program execution on a processor at the specified address or start drivers.

(continued on next page)

Table 3–5 (Cont.) Console Commands

To	Use	Function
Test the System, Subsystem or Device	`test`	Tests the entire system, a subsystem or a specific device, depending on the device list argument. A list of the subsystems and devices that can be tested can be obtained from the `show config` and `show device` commands. Testing can be performed on disk drives, DSSI disks, SCSI disks, memory, network subsystem, Future Bus devices (if present) or a subset of the device system.

Syntax

The table shows a summary of the basic common console commands.

Table 3–6 Syntax of Common Console Commands

Command	Options	Parameters
`boot`	[-file filename] [-flags root, bitmap] [-halt]	[boot_device]
`continue`	—	—
`deposit`	[-{b,w,l,q,o,h}] [-n val] [-s val]	[device:]address data
`examine`	[-{b,w,l,q,o,h,d}] [-n val] [-s val]	[device:]address
`help`	—	[command]
`man`	—	[command]
`initialize`	[-c] [-d device_path]	[slot-id]
`set`	—	envar value
`set host`	[-dup] [-task t]	node
`show`	—	envar, config, device, error, fru, hwrpb, memory
`start`	—	address
`test`	—	cpu, memory, ethernet, scsi

Reference

For more information about console commands, see:

- *Common Console User Interface Functional Specification, October 20, 1993, Bill Cummins*
- *Preliminary APS Firmware Specification, March 30, 1994, AVS Engineering*

Running Diagnostics

Scripts

The reference console image should be tested on the reference platform. Testing should include running diagnostics with command scripts. A developer can create additional scripts for the particular application.

Reference

Refer to the CMS library group: `TEST_CMD`, and the `.CMD` files. These files are also available in the `CP$REF` directory.

More information can be found in: *Alpha Diagnostic Programmer/User Interface*, Rev 2.2.

Generic Diagnostic Scripts

The following list of diagnostic command scripts are basic and commonly used on all platforms. This is not an exhaustive list. These scripts consist of console commands that exercise functional areas of the system, such as memory or the CPU. Developers can customize or create additional scripts for their platform.

Table 3–7 Diagnostic Command Scripts

Command Script	Syntax	Description
`test`	test [device_name] test [disk] [dssi]...[scsi] [memory]	Test the system, a subsystem, or a specific device.
`exer`	exer [qualifiers] [device] exer -p (pass_count)... dk*.* exer_read [qualifiers] [device] exer_write [qualifiers] [device]	Functional read and write from/to a device(s).
`memexer`	—	Exercise memory.
`memtest`	memtest [-sa start_ address] [-ea end_address]	Test a section of memory.
`nettest`	nettest [-f file] [-mode port_mode] [-p pass_count]	Test the network ports using MOP loopback.

Diagnostic Session

Sample User Session

The following sample diagnostic session provides an example of how various diagnostic commands can be used to run diagnostics in an Alpha AXP console environment.

```
>>> set d_* -d ❶
>>> set d_harderr continue ❷
>>> cpu_tst ❸

      ID  Program         Device     Pass Hard/Soft Test     Time
-------- -------- --------------- -------- --------- ---- --------
00000011  cpu_tst         kn7aa0        0    0    0      10:32:55 ❹

*** Hard Error - Error #10 on FRU: kn7aa0
Data compare error
      ID  Program         Device     Pass Hard/Soft Test     Time
-------- -------- --------------- -------- --------- ---- --------
00000011  cpu_tst         kn7aa0        1    1    0    2 10:33:25

*** End of Error *** ❺

^C ❻

Testing Complete
      ID  Program         Device     Pass Hard/Soft Test     Time
-------- -------- --------------- -------- --------- ---- --------
00000011  cpu_tst         kn7aa0        1    1    0      10:34:00

* End of Run - Failed * ❼

>>> set d_harderr halt ❽
>>> set d_passes 0 ❾
>>> set d_startup on ❿
>>> set d_complete on ⓫
>>> set d_trace on ⓬
>>> set d_eop on ⓭
>>> set d_report full ⓮
>>> cpu_tst -t 1,2 ⓯

      ID  Program         Device     Pass Hard/Soft Test     Time
-------- -------- --------------- -------- --------- ---- --------
00000014  cpu_tst         kn7aa0        0    0    0      10:35:29 ⓰
00000014  cpu_tst         kn7aa0        1    0    0    1 10:35:34 ⓱
00000014  cpu_tst         kn7aa0        1    0    0    2 10:35:40 ⓲
00000014  cpu_tst         kn7aa0        1    0    0      10:35:42 ⓳
00000014  cpu_tst         kn7aa0        2    0    0    1 10:35:49 ⓴

*** Hard Error - Error #10 on FRU: kn7aa0
Data compare error
      ID  Program         Device     Pass Hard/Soft Test     Time
-------- -------- --------------- -------- --------- ---- --------
00000014  cpu_tst         kn7aa0        2    1    0    2 10:36:12

Extended Error Information:

Address: 00400008
Expected: A5A5A5A5
Received: A5A5A5A6
```

```
*** End of Error *** ㉑
Testing Complete
      ID  Program          Device     Pass Hard/Soft Test     Time
-------- -------- --------------- -------- --------- ---- --------
00000014  cpu_tst          kn7aa0        2    1    0      10:36:36
* End of Run - Failed * ㉒
>>>
```

❶ Set all global diagnostic environment variables to default values.

❷ Continue-on-error is set for hard errors through the d_harderr environment variable.

❸ Use the cpu_tst command to test the first CPU.

❹ The startup message is printed for the cpu_tst diagnostic that is run.

❺ A hard error is reported by the cpu_tst diagnostic.

❻ Enter Ctrl/C to terminate further testing.

❼ The diagnostic completion message is printed after terminating the testing.

❽ Halt-on-error is set for hard errors through the d_harderr environment variable. (default)

❾ The pass count is set to 0, indicating test indefinitely, through the d_passes environment variable.

❿ The printing of the diagnostic startup message is enabled through the d_startup environment variable.

⓫ The printing of the diagnostic completion message is enabled through the d_complete environment variable.

⓬ Test trace messages are enabled through the d_trace environment variable.

⓭ End-of-pass messages are enabled through the d_eop environment variable.

⓮ Reporting of extended error information is enabled by setting the d_report environment variable to FULL.

⓯ The cpu_tst diagnostic is run again, but only to run tests 1 and 2.

⓰ The startup message is printed for the memory diagnostic that is run.

⓱ The test trace message for test 1 is printed.

⓲ The test trace message for test 2 is printed.

⓳ The end-of-pass message is printed indicating that the first pass of the memory diagnostic has completed.

⓴ The test trace message for test 1 is printed again for the second pass.

㉑ A hard error is reported by the `cpu_tst` diagnostic in the second pass. This time the extended error information is printed because the `d_report` environment variable is set to FULL.

㉒ The diagnostic completion report is printed and the diagnostic is aborted since halt-on-error is set for hard errors.

Diagnostic Messages

Types

Six types of diagnostic messages are displayed which all have the same basic format:

- Startup
- Test Trace
- Status
- Error
- End of Pass
- Completion

Error Messages

The D_REPORT environment variable can be set to SUMMARY or to FULL, which provides two levels of error information.

There are three types of errors that signify the severity of the error that occurred. They are:

- Soft

  A soft error implies that a recoverable error has occurred. An example of this may be an ECC Single Bit Error or an error occurring in an operation that can be successfully retried. The default action is to continue on soft errors.

- Fatal

  A fatal error report also signifies that a nonrecoverable error occurred. A proposed example of this error is an unrecoverable systemwide error that may call for a system bugcheck or something similar. The action on a fatal error is always to halt and cannot be set differently.

- Hard

  A hard error report implies that a nonrecoverable error has occurred. Most errors encountered during testing, such as data compare errors, checksum error, or failed write/read operations are classified as hard errors. After detecting a hard error, the error can be cleaned up and if continue on hard error is set, the program should be able to continue. The default action when a hard error is signaled is to halt.

The following example shows a hard error message.

```
*** Hard ❶ Error - Error #15 ❷ on FRU: kn7aa0 ❸
Data compare error ❹

      ID  Program        Device     Pass   Hard/Soft Test     Time
-------- -------- -------------- -------- --------- ---- --------
00000011❺ cpu_tst❻    kn7aa0❼     5❽    1❾  9❿   2⓫    10:36:12⓬

Extended Error Information: ⓭

Address: 00400008
Expected: A5A5A5A5
Received: A5A5A5A6

*** End of Error *** ⓮
```

❶ Error Type. Possible values are:

- Hard - Hard Error
- Soft - Soft Error
- Fatal - Fatal Error
- None - No error type displayed

❷ Error Number

❸ FRU Callout

❹ Error Message

❺ PID

❻ Program/Module name

❼ Device Name

❽ Current Pass Count

❾ Current Hard Error Count (will also be set to 1 if a Fatal Error occurred)

❿ Current Soft Error Count

⓫ Current Test Number

⓬ Timestamp

⓭ Extended Error Message

⓮ Error Message Delimiter

Bootstrapping

Introduction

Bootstrapping is the process of locating, loading and transferring control to the primary program image. The primary program image may be a primary bootstrap program such as Alpha Primary Boot (APB), ULTRIXboot, or any other applicable program specified by the user or residing in the boot block.

Bootstrap Procedure

The system firmware uses a bootstrap procedure defined by the Alpha AXP architecture.

Usually, a bootstrap can be attempted **only** by the primary processor, commonly referred to as the boot processor.

To bootstrap the operating system. The firmware uses device and optional file name information specified either on the boot command line or in appropriate environment variables.

To begin a bootstrap, the firmware writes boot device information to the HWRPB and appropriate environment variables to be used by the boot callbacks and secondary bootstrap programs. The console locates a boot callback for the specified device. If a suitable callback is found, control is transferred to it.

State Assumed by the Operating System

The operating system assumes that the following is set up at boot time:

- A valid and accurate HWRPB exists.
- Memory has been mapped appropriately.
- All memory is initialized to at least a valid error checking state.
- Drivers have been initialized with a solid set of callbacks to communicate with the operating system.
- A legal boot device is used to boot the operating system.

Conditions for Bootstrapping

There are only three conditions where the boot processor attempts to bootstrap the operating system:

- The `boot` command is typed on the console terminal.
- The system is reset and the `auto_action` environment variable is set to `boot`.
- An operating system restart is attempted and fails.

Steps for Bootstrapping

The steps the console performs for bootstrapping the system are as follows:

1. Set the Boot In Process (BIP) flag. If the BIP flag is already set, the boot fails.
2. Identify the boot device.
3. Search the device database for a callback that matches the specified boot device. If none is found, the boot fails.
4. Load the boot parameters into the Hardware Restart Parameter Block (HWRPB).
5. Initialize the boot device and begin loading the boot image.
6. Load the boot image from the boot device.
7. Transfer control to the boot device.

If the bootstrap fails, the console will display a message on the console terminal and return to the console prompt. If bootstrap succeeds, the operating system is responsible for clearing the Bootstrap In Progress (BIP) flag.

Boot Environment Variables

Table 3–8 lists some boot environment variables that are not platform specific and are implemented by all systems.

Table 3–8 Environment Variables

ID	Variable	Attributes	Function
01	AUTO_ACTION	NV,W	The action the console should take following an error halt or powerfail. Values are: • BOOT - Attempt bootstrap • HALT - Halt, enter console I/O mode • RESTART - Attempt restart; if restart fails, try boot • Anything else - Halt, enter console I/O mode; the default value when the system is shipped in "HALT" (544C 4148$_{16}$)

Key to variable attributes:

NV - Nonvolatile. The last value saved by system software or set by console commands is preserved across system initializations, cold bootstraps, and long power outages.
W - Warm nonvolatile. The last value set by system software is preserved across warm bootstraps and restarts.

(continued on next page)

Table 3–8 (Cont.) Environment Variables

ID	Variable	Attributes	Function
02	BOOT_DEV	W	The default device or device list from which booting is attempted when no boot path is specified by the `boot` command. This variable may be a boot search list. The console derives the value from `bootcmd_dev` at console initialization; the value is preserved across warm bootstraps. The format of value is independent of the console presentation layer.
03	BOOTDEF_ DEV	NV	The device or device list from which booting is to be attempted.

Key to variable attributes:

NV - Nonvolatile. The last value saved by system software or set by console commands is preserved across system initializations, cold bootstraps, and long power outages.
W - Warm nonvolatile. The last value set by system software is preserved across warm bootstraps and restarts.

Reference

Environmental variables with an ID of 00_{16} to $3F_{16}$ are common to all implementations of the Alpha AXP console.

For more information, refer to the *Platforms* section of the *Alpha AXP System Reference Manual V5.0*.

Boot Environment Variables After Loading

The console indicates the actual bootstrap path and device used in the BOOTED_DEV environment variable. The console sets BOOTED_DEV after loading the primary bootstrap image and prior to transferring control to system software.

Boot Command Syntax

Example 3–3 shows the syntax used to boot.

Example 3–3 Boot Command

```
>>> boot [-file filename] [-flags longword[,longword]]
        [-protocols ethernet_protocol] [-halt]
        [boot_device]
```

Options to the command are the file name to boot, operating system flags, Ethernet protocols: TCP/IP or MOP, whether the CPU should halt after loading in the file, and the device used to boot from: DKA100, EZA0, ERAWA, and so on.

Booting the OpenVMS Operating System

Overview

The syntax for the booting OpenVMS and specifying boot flags is:

```
    >>> boot device_name -flags system_root,option_flags
```

Table 3–9 shows booting OpenVMS with various options: system disk drive zero, system root zero, including invoking the XDELTA debugger.

Table 3–9 OpenVMS Boot Options

Option	Function
`>>> boot dka0`	Normal, nonstop bootstrap (default)
`>>> boot dka0 -fl 0,1`	Conversational boot; stops in SYSBOOT
`>>> boot dka0 -fl 0,2`	Includes XDELTA with the system, but does not take the initial breakpoint
`>>> boot dka0 -fl 0,4`	Stops the boot procedure at the initial breakpoint
`>>> boot dka0 -fl 0,6`	Includes XDELTA with the system, and take the initial breakpoint
`>>> boot dka0 -fl 0,7`	Includes XDELTA with the system, stops in SYSBOOT, and takes the initial breakpoint at system initialization

BOOT_OSFLAGS Variable

The operating system boot flag options can also be set in the console variable `BOOT_OSFLAGS` using the console set command, for example:

```
>>> SET BOOT_OSFLAGS 0,1
```

Other Boot Options

Table 3–10 shows some other useful OpenVMS flags.

Table 3–10 Other OpenVMS Boot Options

Flag	Value	Function
BOOTBPT	10	Takes bootstrap breakpoint
HALT	100	Halts before transferring control to secondary bootstrap
CRDFAIL	2000	Marks pages containing CRDs (correctable read data errors) as bad
DBG_INIT	10000	Displays extensive, detailed debug messages during boot process

Recovering from Errors During System Booting

Introduction

Error recovery during system booting is controlled by flags in the primary CPU's per-CPU slot in the HWRPB.

Bootstrap Flags Description

The bootstrap flags detect failed bootstraps and prevent repeated attempts to automatically bootstrap a failed system.

Bootstrap Flags List

The bootstrap flags are as follows:

- Bootstrap-in-progress (BIP) flag
- Restart-in-progress (RIP) flag
- Restart capable (RC) flag

Bootstrap Flags Values

Based on the values of the bootstrap flags, the console takes action as follows:

Table 3–11 Bootstrap Flags

BIP	RC	RIP	Console Action
Set	Clear	NA	Bootstrap fails
Set	Set	NA	Restart processor, if permitted
Clear	NA	Clear	Restart processor, if permitted
Clear	NA	Set	Restart fails

The console sets the BIP flag and clears the RC flag prior to transferring control to system software.

System software sets the RC flag to indicate that sufficient context has been established to handle a restart attempt.

System software clears the BIP flag to indicate that the bootstrap operation has been completed. The RC flag should be set prior to clearing the BIP flag.

Reference

For more information about bootstrap error recovery, refer to the *Alpha AXP System Reference Manual V5.0*.

Booting from the Ethernet

Introduction

Some platforms support up to two local Ethernet ports, referenced by the firmware as devices, such as: eza0 and ezb0.

Whenever a network bootstrap is selected, the bootstrap routine makes continuous attempts to boot from the network. The network bootstrap continues, until either a successful boot occurs, a fatal controller error occurs, or the boot is terminated by pressing Ctrl/C.

Two Protocols Supported

Two Ethernet protocols are supported for network bootstraps:

- DECnet MOP
- TCP/IP BOOTP

MOP Booting

Whenever the environment variables contain the string `mop`, bootstrap uses the DECnet MOP program load sequence for bootstrapping the system and the MOP "dump/load" protocol type for load-related message exchanges.

MOP Network "Listening"

Whenever the console is running, it "listens" on each of its ports for other maintenance messages directed to the node and periodically identifies itself at the end of each 8-to-12 minute interval, prior to a bootstrap retry. In particular, this "listener" supplements the MOP functions of the console load requester typically found in bootstrap firmware and supports.

BOOTP Booting

Whenever the environment variables, `eza0_protocols` and `ezb0_protocols`, are set to `bootp`; the console attempts an Internet boot.

The console implements Bootstrap Protocol (BOOTP) and Trivial File Transfer Protocol (TFTP) client protocols for network bootstrapping in an Internet environment. Supporting TFTP and BOOTP requires pieces of UDP, IP and ARP.

It is important to note that Internet booting is a **two-stage** operation. First, BOOTP provides the client with information needed to obtain an image. The client then uses a second protocol, TFTP, to obtain the image. Both BOOTP and TFTP use User Datagram Protocol (UDP) as the primary transport mechanism to send datagrams to other application programs.

All Internet drivers are located in the `inet_driver.c` module.

Environment Variables

Environment variables can define the default protocols to be used for booting from each port, for example, `eza0_protocols` and `ezb0_protocols`. These variables can be set to either `mop` or `bootp`. By default, these variables are set to `mop bootp`, enabling both boot protocols. Alternately, MOP and BOOTP attempts are made until the boot succeeds, passing control to the loaded image.

Note

The volatile environment variable must be set for booting with the TCP/IP protocol. There are other environmental variables involved as well. For more information, refer to the *Device Driver Interface Guide for the Alpha AXP Firmware*.

Ethernet Boot Example

The protocol can be specified in the command line, if an environmental variable is not used, or to override the setting of the variable.

Example 3–4 shows the syntax used to boot from the Ethernet.

Example 3–4 Booting from the Ethernet

```
>>> boot -file filename -protocols TCP/IP ERAWA
```

Booting DEC OSF/1

Boot Environment Variables

To boot the DEC OSF/1 operating system requires specifying the correct flag and file with the `boot` command, or setting the environment variable, as shown in Example 3–5.

Example 3–5 Setting the Boot Variables for DEC OSF/1

```
>>> set boot_osflags "A"
>>> set boot_file "/vmunix"
```

Boot Flag Definitions

Table 3–12 describes the flags that can be specified when booting the DEC OSF/1 operating system. The following flags can be set in the environment variable `boot_osflags`, or specified with the `boot` command.

Table 3–12 DEC OSF/1 Boot Flag Definitions

Flag	Definition	Description
A	Auto reboot	Automatic reboot to multiuser mode
D	Enable full dumps	Partial system dumps are default
I	Interactive boot	Boot user-selected kernel
K	Boot `kdebug`	Allow kernel debugging

4 Developing a Minimal Image for the Target

Overview

Introduction

Once the basic hardware is proved to be working, developers can build a skeletal image with minimal functionality: disabling the time-of-year (TOY) clock, hardware interrupts, machine checks, and most drivers.

This chapter describes how to develop the minimal image by:

- Defining memory
- Modifying PALcode
- Modifying the kernel and driver files
- Debugging to get to the first console prompt
- Modifying the build files to build the minimal image
- Booting the operating system during debugging

Overview of Modifications Needed

Memory Modification

For memory, the following may need to be modified:

Table 4–1 Platform-Dependent Memory Functionality

Functionality	Description/Example Files
Memory, Backup Cache	Serial ROM Code (SROM) - The functionality that SROM implements may vary between platforms. In some cases, SROM may need to have knowledge about the system, such as the memory structure and cache in order to initialize the console. Memory configuration registers may need to be set up for the new platform.

Chip-Dependent Modification

For functionality related to the chip, the following may need to be modified:

Table 4–2 Platform-Dependent Chip Functionality

Functionality	Description/Example Files
PALcode	You must initialize character output hardware (combo chip) and other chips for your particular platform. You must disable the following in the PALcode until after debugging: • Machine checks • CRD, correctable read error interrupts • Performance counter interrupts • Memory controller interrupts

Kernel Modification

For functionality related to the kernel, the following may need to be modified:

Table 4–3 Platform-Dependent Kernel Functionality

Functionality	Description/Example Files
I/O	Account for the address ranges that vary from platform to platform for I/O space (for example, ISA/EISA, PCI).
Powerup Process	Each platform should spawn its own powerup script process. Some changes here are related to ensuring that the device drivers will not initialize automatically.
	Character put calls and print calls should be placed throughout the powerup script and `kernel.c` so the progress of the process can be monitored.
Drivers	To get up and running, compile all drivers to run in polled mode instead of interrupt mode.
	When first trying to execute the console image, it could be beneficial to disallow the phase 5 drivers from loading. These drivers directly control hardware such as the VGS, keyboard and floppy and can cause the system to hang. To do this, set Robust Mode.
	Trace driver initialization and display a message for every driver's initialization start and end.
Timer	Identify and change any platform-specific code and change it as appropriate.
Environment Variables	Environment variables are a mechanism to manage console state. They are typically stored in EE Flash ROM or a suitable nonvolatile device, such as NVRAM.

Defining Memory

Introduction

This section describes how to define the memory configuration.

Defining the Memory Configuration

Platform Memory Configuration

Memory sizing and configuring includes the following activities:

- Initializing memory
- Sizing memory
- Configuring memory into the appropriate locations
- Setting interleaving (optional)
- Marking bad pages
- Creating bitmaps

Depending on the platform, either SROM or the console will perform each of these functions. In addition, how the memory board is set up is also platform specific.

Each platform has a specific memory configuration file that contains routines that perform the memory sizing and configuring for that particular platform. Specifically, the file is identified as `memconfig_platform.c`, where `platform` is the name of the current platform.

SROM Memory Code

Example 4–1 shows part of the contents of the `SROM_DEF.MAR` file that defines the memory registers. This file defines all control and status registers (CSR) for a sample platform that are used by the SROM during initialization.

Example 4–1 SROM Memory Configuration Registers

```
;                              SROM_DEF.MAR
                                   .
                                   .
;;;;;
;;;;;Sable Memory CSRs
;;;;;
ERR_ADDR        == ^x000        ; ERR Address
CTRAP1_ADDR     == ^x020        ; CTRAP1 Address
CTRAP2_ADDR     == ^x040        ; CTRAP2 Address
CONFIG_ADDR     == ^x060        ; CONFIG Address
EDC1_ADDR       == ^x080        ; EDC1 Address
EDC2_ADDR       == ^x0a0        ; EDC2 Address
EDCCTL_ADDR     == ^x0c0        ; EDCCTL Address
STRBUF_ADDR     == ^x0e0        ; STRBUF Address
REFCTL_ADDR     == ^x100        ; REFCTL Address
CRDCTL_ADDR     == ^x120        ; CRDCTL Address
RES0_ADDR       == ^x140        ; RES0 Address
RES1_ADDR       == ^x160        ; RES1 Address
RES2_ADDR       == ^x180        ; RES2 Address
RES3_ADDR       == ^x1a0        ; RES3 Address
RES4_ADDR       == ^x1c0        ; RES4 Address
FRCREF_ADDR     == ^x1e0        ; FRCREF Address

        .ENDM
```

Modifying PALcode

Introduction

The PALcode is dependent on the chip implementation and sensitive to coding constraints imposed by that implementation. Before modifying PALcode, it is important to be familiar with PALcode and the chip specifications for a particular platform.

This section provides an overview of the following:

- PALcode Modification
- PALcode Changes
- Modifying the PALcode Reset Routine
- Building PALcode
- PALcode MMS Description File

PALcode Modification

Introduction

To enable the minimal image to execute on the target platform, PALcode may need to be modified.

PALcode Functionality

The PALcode executes in privileged mode, (interrupts disabled, physical addressing, IPR read/write). The function of PALcode is to initialize the IPRs, the system hardware, and handle interrupts and address translation faults, and other exceptions. It also provides certain operating system primitives and atomic operations.

At powerup/reset, SROM code loads the PALcode, passing control to the reset routine. After the reset routine initializes the hardware, control is passed to the console firmware that will boot the operating system.

PALcode Modification Steps

A PALcode modification strategy is recommended in order to support the first powerup of the target platform and to support troubleshooting by helping to distinguish between hardware and software problems.

- Initialize the hardware that controls character I/O to the console terminal and other hardware. For an example, refer to the `reset` PALcode routine in `PAL_EV4.MAR`. This should include:
  - Initialization of internal processor registers (IPR) that control DECchip functions, interrupts and errors, system logic, backup cache, for example, ABOX_CTL, ICCSR and BIU_CTL IPRs
  - Initialization of interrupt, memory, cache, I/O bridge, and I/O control logic

Note

Much of the same initialization occurs when the SROM code is executed. This may or may not completely overlap initialization done with the PALcode `reset` routine.

- Turn off interrupts with a macro during the first powerup attempt. For example, the `genipltbl` macro in `PAL_EV4.MAR` disables the following interrupts:
  - Correctable read data errors (CRD)
  - Performance counters
  - Serial line
  - Memory and I/O controller errors

  Reference

  Refer to the chip-specific hardware documentation for specifics on your platform.

  - Enable timer interrupts after the basic code is working and reliable operation is required from floppy, disks, and Ethernet hardware.
- Disable machine checks.
- Place character output (PUTC) traces in strategic code blocks, informing the developer of progression through the PALcode.
- Prove that PALcode can transfer control by having it execute an image loaded in another area of memory by doing the following:
  - Build an uncompressed Alpha AXP Firmware image.
  - Load the image into another area of memory; the PALcode is position-independent code (PIC).
  - Execute this PIC image by having the PALcode transfer control. This should display the character O/P traces.

PALcode Changes

Overview

When porting to a different hardware platform, there are several different areas of PALcode that may need to be modified. Typically those areas involve:

- Supporting interrupts
- Selecting different console support

Supporting Interrupts

Interrupt handling can vary with system design as well as between different implementations of DECchips. For example, one design might use a chip commonly used in the PC design: the priority interrupt control chip. This chip must be initialized, along with certain internal processor registers (IPRs) within the DECchip, to enable proper interrupt decoding/encoding.

The number of hardware interrupt signals are different between the DECchip 21064 and the DECchip 21066/21068, six as opposed to three, respectively. This means that the hardware enable registers (HIER) have to be initialized differently, and the hardware interrupt request register (HIRR) has to be interpreted differently.

Selecting Different Console

Various systems use different hardware to control input/output to the console terminal. One may use the combination chip that controls two serial ports (COM1 and COM2), for the character I/O. Another system may incorporate a VGA control module or both communication ports and VGA.

Building PALcode

MMS Description Files

The PALcode is just one of the three code sections of the Alpha AXP Firmware image, (see Figure 2–1). PALcode is built from a number of macro files using the MMS utility and description files invoked when creating an Alpha AXP Firmware image, namely:

- `descrip.mms`, the main description file executed by the developer to build an Alpha AXP Firmware image
- `platform_`*`configuration`*`.mms` created by the developer, specific target/dependency files for the platform configuration
- `pal_descrip.mms`, describes the PALcode target and dependencies

Platform MMS File

The following code fragment creates a PALcode image for a sample platform, and is part of the `platform_sample_platform.mms` build file, where `sample_platform` is the platform name.

```
.
.
cp$exe:pal_lca4_$(platform).exe : cp$src:lca4_sample.mar,
                                  cp$src:pal_ev4.mar,
                                  cp$src:osfpal_common.mar,
                                  cp$src:osfpal_machine.mar
mms := mms
mms /descrip=cp$src:pal_descrip.mms lca4_$(platform)
.
.
```

Image File Description

The PALcode image is comprised of a number of files, as can be seen in Example 4–2, its MMS description file. The main files are described in Table 4–4.

Table 4–4 PALcode Files for a Sample Platform

File	Description
`pal_ev4.mar`	The main body of PALcode containing most routines.
`osfpal_common.mar`	PALcode that supports the DEC OSF/1 operating system, for example: • Privileged and unprivileged CALL PAL instructions • Arithmetic, memory management and other exception routines

(continued on next page)

Table 4–4 (Cont.) PALcode Files for a Sample Platform

File	Description
`osfpal_machine.mar`	Macro definitions, and console callback routines, for example: • Data cache flush (CFLUSH) • Interprocessor interrupt request (WRIPIR) • Console service routines (CSERVE), put character, get byte stream, and so on

`lca4_platform.mar`	Conditional definitions, for example:
	• Platform
	`platform_system = 1`
	`lca4            = 1`
	• Debug
	`.`
	`.`
	`; enable_debug_boot = 1`
	`;disable_dcache     = 1`
	`disable_mchkcrd     = 1`
	`.`
	`.`

`alphamac.mlb`	PALcode library of definitions and macros

MMS Build Files

The final phase of the build process assembles, compiles, and links the target image using the MMS utility and its description files that specify the target and its dependencies. The file `descrip.mms` is the main description file that calls the specific platform file, for example `platform_sample.mms`, that the developer creates for the target platform to be ported. It is within this platform description file that the PALcode description file `pal_descrip.mms` is called.

Reference

Example 4–2 shows the target and dependencies within `pal_descrip.mms` that creates the PALcode for the Platform platform. Table 4–4 describes the PALcode files.

PALcode MMS Description File

pal_descrip.mms File

The PALcode description file `pal_descrip.mms` is executed by the platform-specific description file, for example, `platform_sample.mms`, during the last phase of the build process. The `pal_descrip.mms` file describes the target PALcode image and its dependencies.

pal_descrip.mms Example

Example 4–2 shows a section of `pal_descrip.mms` that creates the PALcode for a specific sample platform.

Note that two object files are linked together.

```
pal_lca4_platform.obj
osfpal_lca4_platform.obj
```

Table 4–4 describes the files used to create the PALcode.

Example 4–2 pal_descrip.mms for a Sample Platform

```
.suffixes
.suffixes       .exe .obj .c .h .b32 .req .mar .sdl

src = cp$src:
cfg = cp$cfg:

!
!       We define the linker, librarian, macro assembler and
!       C compiler to be MMS macros so that we may switch transparently
!       between Alpha and VAX versions of these tools
!
.include        $(cfg)macros.mms
.include        $(src)setup.mms
!
first_target :
        @ write sys$output "You have to explicitly specify a target"
.
.
lca4_platform : cp$exe:pal_lca4_platform.exe
        @ continue
.
.
cp$exe:pal_lca4_platform.exe : cp$obj:pal_lca4_platform.obj,
                              cp$obj:osfpal_lca4_platform.obj
        p_link cp$obj:pal_lca4_platform/map=cp$exe:/exe=cp$exe:
                                      /full
                                      /system=0+osfpal_lca4_platform
.
.
.sdl.mar
        sdl/lang=mac=cp$src/vms $(mms$source)
!
!       Macro libraries
!
cp$src:alphamac.mlb            : cp$src:starlet.mar,-
                                 cp$src:paldef.mar,-
                                 cp$src:alpha_defs.mar,-
                                 cp$src:osfalpha_defs.mar,-
```

(continued on next page)

Example 4–2 (Cont.) pal_descrip.mms for a Sample Platform

```
                                  cp$src:impure.mar,-
                                  cp$src:pal_macros.mar,-
                                  cp$src:logout.mar,-
                                  cp$src:pal_def.mar
        library/create/macro cp$src:alphamac -
                cp$src:starlet.mar,paldef,alpha_defs,osfalpha_defs,impure,
                       pal_macros,logout,pal_def

!
! Build procedure for PAL code
!
fp_dep = cp$src:entry_v4.mar,cp$src:gccmac.mar,cp$src:fp_v4.mar,
                cp$src:long64.mar,cp$src:rtwdiv.mar
fp_asm = cp$src:entry_v4+gccmac+fp_v4+long64+rtwdiv

cp$obj:pal_lca4_platform.obj : cp$src:lca4_platform.mar,cp$src:pal_ev4.mar,
                              cp$src:alphamac.mlb
        p_assemble cp$src:lca4_platform+pal_ev4+alphamac
                          /lib/lis=cp$lis:pal_lca4_platform
                          /obj=cp$obj:pal_lca4_platform/show=meb
.
.
cp$obj:osfpal_lca4_pjlatform.obj : cp$src:lca4_platform.mar,
                                 cp$src:osfpal_common.mar,
                                 cp$src:osfpal_machine.mar,cp$src:alphamac.mlb
        p_assemble cp$src:lca4_platform+osfpal_common+osfpal_machine+alphamac
                         /lib/lis=cp$lis:osfpal_lca4_platform
                         /obj=cp$obj:osfpal_lca4_platform/show=meb
.
.
```

Modifying the Kernel Files

Introduction

This section describes the following:

- Adding Console Terminal Support
- Time of Year (TOY)
- NVRAM - Environment Variables
- Environmental Variable Debug
- Tracing the Transition from PALcode to Kernel
- Showing Progress During Initialization

Adding Console Terminal Support

Introduction

Macros are defined in `kernel_alpha.mar` to support character I/O to the terminal.

Modified Macros

The `combott_putc` macro can be modified to the address required by the hardware. An example of this can be seen in Example 4–7.

The `combott_putc` macro must also be modified in the platform utility file, for example `platform_util.mar`, for character I/O debug support by XDELTA. Example 4–3 shows the `combott_putc` macro definition. This macro is also used when the developer strategically places `jputc` and `jgetc` in `kernel.c` for tracing code progression. Example 4–8 shows how the kernel code can be traced.

Console Terminal Address

Input/output addresses are specified in the following macro files:

- `kernel_alpha.mar`

  The Alpha AXP kernel functions of the Alpha AXP Firmware are contained in this file.
- `platform_util.mar`

  The platform utility file defines macro routines that read and write character data and registers to the serial communication port. The address for this controller is also defined in this file.

Example 4–7 shows how `kernel_alpha.mar` is modified, and Example 4–3 shows the `combott_putc` macro definition in `platform_util.mar`.

Platform Utility File Example

Example 4–3 shows the I/O base address definitions and the `combott_putc` macro in `platform_util.mar` that were used for a specific platform. Note that the I/O address varied with the revision of the hardware as defined by `lca4_pass2`.

Example 4–3 Defining the Address in the Platform Utility File

```
;                        PLATFORM_UTIL.MAR
                                .
                                .
lca4_pass2      = 0

.if ne lca4_pass2
io_base = ^x1c
.endc

.if eq lca4_pass2
io_base = ^x30
.endc

;+
;
; This macro is used to put a character to the platform serial port.
;
; Inputs:
;       ascchar = Character to be displayed
;       rcom,rs = Scratch registers
;-

com1 = ^x3f8
com2 = ^x2f8
thr = 0
rbr = 0
dll = 0
ier = 1
iir = 2
lcr = 3
mcr = 4
lsr =  5
msr = 6
scr = 7
lsr$v_dr == 0
lsr$m_dr == 1
lsr$v_thre == 5 + 8
;;; lsr$m_thre == 1@lsr$v_thre
lcr$m_sbs  == ^x04
lcr$m_dla  == ^x80
mcr$m_dtr  == ^x01
mcr$m_rts  == ^x02
mcr$m_out1 == ^x04
mcr$m_out2 == ^x08
baud_9600  == ^x0c
char_8bit  == ^x03
```

(continued on next page)

Example 4–3 (Cont.) Defining the Address in the Platform Utility File

```
.macro  combott_putc    ascchar,rcom,rs,?lab1,?lab2
        lda     'rcom',io_base(r31)     ; io and EISA bus address
        sll     'rcom',#28,'rcom'
        lda     'rs',com1+lsr(r31)      ; line status register
        sll     'rs',#5,'rs'            ; shifted com1 address
        bis     'rs','rcom','rcom'      ; tt port address in rcom
lab1:
        ldl     'rs',('rcom')
        mb
        srl     'rs',#lsr$v_thre,'rs'   ; extract the bit
        blbc    'rs',lab1               ; if not ready to txmit, spin.
        lda     'rcom',io_base(r31)     ; io and EISA bus address
        sll     'rcom',#28,'rcom'
        lda     'rs',com1+thr(r31)              ; transmit holding register
        sll     'rs',#5,'rs'            ; shifted com1 address
        bis     'rs','rcom','rcom'      ; tt port address in rcom
        and     'ascchar',#^xff,'rs'
        stl     'rs',('rcom')   ; xmit the character
        mb                              ; wait for the write
.endm  combott_putc
                                .
                                .
```

Time of Year (TOY)

TOY Files

A few files may need modification for controlling TOY clock interrupts, depending on the type of hardware used for this real time clock. They are:

- `toy_driver.c`
- Included header files, for example, `combo_def.h`
- `timer.c`

TOY Example

If the VTI 82C106 Combo Chip is used for the TOY, a developer may need to modify the address definitions in `combo_def.h`. Note that for this platform, Example 4–4 shows that COM1 = 0x3F8 and the TOY = 0xC170.

Example 4–4 TOY Base Address - combo_def.h

```
/****************************  COMBO_DEF.H  *********************************/
                                   .
                                   .
/* base addresses of various devices within combo chip                     */

#define COM1 1016
#define COM2 760
#define LPTD 956
#define LPTS 957
#define LPTC 958
#define KBD_MS 96
#define TOY_RAM 49520
                                   .
                                   .
```

NVRAM - Environmental Variables

Overview

Environmental variables are a mechanism to manage console state. A certain number of these variables (I.D. 0 - $3F_{16}$) are required by the Alpha AXP architecture. An example use of one of these is `boot_dev`, the device used to boot the operating system for the latest attempt.

Environment variables are typically stored in EE Flash ROM or a suitable nonvolatile device, such as a NVRAM.

Reference

For more information, refer to *Alpha AXP System Reference Manual V5.0, Platform* section.

Modifying NVRAM Files

The files that may need to be modified are:

- `esc_nvram_driver.c`, Driver for ECS (Intel 82374EB) and SIO (Intel 823781B) EEROM
- `nvram_driver.c`, protocol driver
- `nvram_def.h`, NVRAM base address and structure definitions

Address Changes Example

Example 4–5 shows part of the `nvram_def.h` file. The base address and other definitions might need to be modified to support the target platform.

Example 4–5 The nvram_def.h File

```
/***************************  NVRAM_DEF.H  **********************************/
                                    .
                                    .
/* for NVRAM accessed via the ESC and SIO chips */

#define NVRAM_BASE      0x800   /* base address */
#define NVRAM_PAGE_REG  0xC00   /* base address */

struct srm_nvram {
    unsigned short int checksum;
    unsigned short int version;
    unsigned char text[NVRAM_EV_LEN-4];
    } ;

#if PLATFORM

/* Defines for configuration tables.*/
#if PLATFORM
#define NUMBER_OF_ENTRIES        104
#define LENGTH_OF_IDENTIFIER     2000
#define LENGTH_OF_DATA           2048
#define LENGTH_OF_ENVIRONMENT    1500
#define LENGTH_OF_EISA_DATA      2500
#endif
                                    .
                                    .
```

Changing the NVRAM Driver

Example 4–6 shows part of the esc_nvram_driver.c file that manipulates the address for a particular platform. This file is responsible for reads and writes to the NVRAM or EE Flash ROM.

Example 4–6 The esc_nvram_driver.c File

```
/*******************  ESC_NVRAM_DRIVER.C  **********************/
                                      .
                                      .
#if PLATFORM

#define  BankSet8  0x80000B00
#define  BankValid 1

int irl (unsigned int p)
{

    __int64     platform_address = 0x10000000;

    platform_address <<= 4;
    platform_address += (__int64)p;

    return(*((int *)platform_address));

}
                                      .
                                      .
#endif
```

Environmental Variable Debug

Turning Off NVRAM Drivers

If the NVRAM device is not accessed, the NVRAM environmental variable device drivers can be loaded but not started automatically for the first powerup.

To keep the NVRAM device drivers from starting automatically, use keywords to specify them as files instead of drivers and place the keywords in the platform-specific build file (for example, `platform_files.bld`).

The NVRAM drivers are:

- `nvram_driver.c`
- `esc_nvram_driver.c`

Tracing the Transition from PALcode to Kernel

Overview

A put character debug routine was placed in the kernel_alpha.mar routine to echo a character to the terminal connected to COM1 at the point where control is passed from PALcode to the Alpha AXP Firmware.

Example 4–7 shows a modified combott_putc routine for a target port that used an EISA-based I/O bus configuration and combination communication chip.

Example 4–7 Tracing kernel_alpha.mar

```
.if ne PLATFORM_DEBUG ! AVANTI_DEBUG
;;;     put a char to com1 for first platform poweron.
;;;     remove this code after that.

io_base = ^x30
com1 = ^x3f8
thr = 0
lsr =  5
lsr$v_thre == 5 + 8

.macro  combott_putc    ascchar,rcom,rs,?lab1,?lab2
        lda     'rcom',io_base(r31)     ; io and EISA bus address
        sll     'rcom',#28,'rcom'
        lda     'rs',com1+lsr(r31)      ; line status register
        sll     'rs',#5,'rs'            ; shifted com1 address
        bis     'rs','rcom','rcom'      ; tt port address in rcom
lab1:
        ldl     'rs',('rcom')
        mb
        srl     'rs',#lsr$v_thre,'rs'   ; extract the bit
        blbc    'rs',lab1               ; if not ready to txmit, spin.
        lda     'rcom',io_base(r31)     ; io and EISA bus address
        sll     'rcom',#28,'rcom'
        lda     'rs',com1+thr(r31)      ; transmit holding register
        sll     'rs',#5,'rs'            ; shifted com1 address
        bis     'rs','rcom','rcom'      ; tt port address in rcom
        and     'ascchar',#^xff,'rs'
        stl     'rs',('rcom')   ; xmit the character
        mb                              ; wait for the write
.endm  combott_putc

_krn$_start_::
```

Showing Progress During Initialization

Terminal Echoing

The printf and jputc routines can be placed strategically throughout the kernel module kernel.c to provide console terminal echoing during powerup initialization.

jputc Routine

The put character routines can be located within krn$_idle idle process routine to display a character at the point where the platform-specific initialization routine is called.

Note

Note that platform_util.mar defines the macro combott_putc for the combination port put character operation used by jputc.

The base address used for the EISA bus and the COM1 port address might have to be changed for the target platform.

jputc Example

Example 4–8 shows one example of placing the jputc routine.

For a more detailed example, see the current version of kernel.c in CP$REF:.

Example 4–8 Tracing kernel.c with jputc

```
.
.
krn$_idle(char *p)
{
    struct PCB *pcb;
    struct SEMAPHORE *s;
    int i;
    char name[8];
    int id;
    struct LOCK *lock;
    struct impure *IMPURE;
    int hcode;
    int delay_count;
    int size;
    int base;
    int *adr;
    char *free;

    free = p;

/* Perform primary platform-specific initialization */

    platform_init1();
```

(continued on next page)

Example 4–8 (Cont.) Tracing kernel.c with jputc

```
#if PLATFORM
    jputc('\n');
    jputc('Z');
#endif
.
.
```

printf Statements

After the platform has been initialized and drivers have been loaded, `printf` statements can be used in `kernel.c` to echo character strings to indicate results.

Example 4–9 shows how to use a `printf` statement to trace progress as the Alpha AXP Firmware kernel starts. Note that the Avanti platform indicates results to a LED display operator control panel.

Example 4–9 Tracing kernel.c with printf

```
.
.
/* Set up the semaphore and process queue headers */
#if AVANTI
    PowerUpProgress(0xfe);
#else
    printf("initialized idle PCB\n");
#endif

    if (primary()) {

        primary_cpu = id;
        timer_cpu = primary_cpu;

#if AVANTI
        PowerUpProgress(0xfd);
#else
        printf("initializing semaphores\n");
#endif
        pcbq.flink = (void *) &pcbq.flink;
        pcbq.blink = (void *) &pcbq.flink;
        krn$_sem_startup();
.
.
```

Modifying the Driver Files

Introduction

This section describes the following:

- Configuring the Bus
- Bus Windows Device Configuration
- Sequencing Driver Startup
- Disabling Hardware Drivers from Loading
- Tracing Driver Initialization
- Using Polled Mode for Drivers

Configuring the Bus

Defining Addresses

The implementation of I/O bus drivers may change with each hardware system design.

Developers must consider how the bus driver and the devices on the bus are sized and configured.

Developers define the I/O bus driver implementation in the platform-specific I/O files, for example, `platform_io.h` and `platform_io.c`.

platform_io.h Example

Example 4–10 shows part of a sample `platform_io.h` file that defines the Peripheral component interconnect (PCI) and Extended Industry Standard Architecture (EISA) I/O bus addresses.

Example 4–10 PCI and EISA I/O Address

```
                        /* PLATFORM_IO.H */
                               .
                               .
* Register offsets into PCI config space; only longword aligned regs shown */

#define VEND_DEV_ID             0x0
#define COM_STAT                0x4
#define REV_ID                  0x8
#define CACHE_L_SIZ             0xC
#define BASE_ADDR0              0x10
#define BASE_ADDR1              0x14
#define BASE_ADDR2              0x18
#define BASE_ADDR3              0x1C
#define BASE_ADDR4              0x20
#define BASE_ADDR5              0x24

#define RESERVED0               0x28
#define RESERVED1               0x2C
#define EXP_ROM_BASE    0x30
#define RESERVED2               0x34
#define RESERVED3               0x38
#define INT_LINE                0x3C

                               .
                               .
/*                                                                       *
 *              EISA BUS DEFINITIONS                                     *
 *                                                                       */

#define CONFIG_ADDR_SEL 0x1e
#define Eisa_IO         (BaseIO<<28)
#if lca4_pass2  /* pass 1 lca */

#define IO_ADDR_SEL     0x1c
#define DMEM_ADDR_SEL   0x30

#define DMEM_ADDR_SEL   0x20
#define SMEM_ADDR_SEL   0x20
#define MEM_ADDR_SEL    0x20
#endif
```

Bus Windows for Device Controllers

Bus Windows Description

Device controller bus windows can be specified in the port driver for the particular device. For example, in the `n810_driver.c` file, bus windows are defined for the NCR 53C810 SCSI device chip that interfaces with the PCI.

Example 4–11 shows part of the `n810_driver.c` port driver that defines the PCI window space for the NCR 53C810 SCSI device chip.

Reference

For more information about class and port drivers and their API, refer to the *Device Driver Interface Guide for the Alpha AXP Firmware*.

Example 4–11 Defining PCI Bus Windows

```
/******************  N810_DRIVER.C  *********************/
                                   .
                                   .
#if PLATFORM & (PLATFORM_PASS_NUMBER == 2)
#define window_base 0x40000000
#endif

#ifndef window_base
#define window_base 0
/* To make it work as it did before */
#define PCI_MEM_BASE 0x80000000
#endif

#define n810_phys(x) (x + window_base)
#define n810_virt(x) (pb->scripts+(int)(x)-(int)&n810_scripts)
                                   .
```

Sequencing Driver Startup

Introduction

Developers can specify the phase, and the order in that phase, in which a driver should start during the boot process.

Driver Startup Process

The driver startup table identifies when a specific driver is started. The driver startup table includes the initialize routine name, the device driver name, and the phase in which the driver is started.

The source routine DST.C defines the driver startup table structure.

The DST.C routine is built by newbuild and should not be edited directly.

Specifying Driver Startup

To specify the phase in which a driver should start, modify the appropriate information in the configuration build file.

Driver Startup Example

An example driver startup table for phase 3 drivers is as follows:

```
# .
# kbd_driver MUST be started after graphics drivers & before serial port
driver  tt_driver.c           3       group driver ❶
driver  vgag_driver.c         3       group driver ❷
driver  kbd_driver.c          3       group driver ❸
driver  combo_driver.c        3       group driver ❹
```

In this example, the terminal class driver tt_driver.c ❶ must start before the serial terminal port driver combo_driver.c ❹ (VLSI 82C106 PC/AT integrated combination I/O chip driver).

The graphic VGA driver vgag_driver.c ❷ is started before the serial terminal port driver. Starting the VGA driver first allows the VGA monitor to be used as the console device (if present) instead of the serial terminal port.

The keyboard driver kbd_driver.c ❸ must be started after graphics drivers, but before the serial terminal port.

Disabling Hardware Drivers from Loading

Robust Mode

When first trying to execute the console image, it could be beneficial to stop the phase 5 drivers from loading. The phase 5 drivers directly control hardware such as the VGA, keyboard, and floppy, and can cause the system to hang.

To stop the phase 5 drivers from loading, set robust mode in the powerup routine.

Example 4–12 shows an abstract of code from the powerup routine that disables phase 5 drivers from loading.

Example 4–12 Setting Robust Mode in powerup_platform.c

```
int     _align (LONGWORD) robust_mode =  1;
.
.
.
       if (!robust_mode && !apu_start)
       {
           qprintf("Start driver phase 5\n");
           ddb_startup (5);
       }
```

Loading and Unloading Drivers

Later, after the console prompt is displayed, drivers can be loaded manually with the console command:

```
>>> init device_name
```

For some platforms, drivers can be enabled for polling and testing devices, or shut down with the console commands:

```
>>> configure -p device_name
>>> shutdown device_name
```

Powerup Process

The powerup process controls the screen and finishes all the initialization. This process is started by krn$_create in the kernel.c module. At this point, the tt driver is running in the initialization process.

Tracing Driver Initialization

Using printf to Display Drivers

The printf statements can be conditionally compiled to display each driver at the start and end of its initialization. Two printf statements can be placed in the ddb_startup routine in the filesys.c file.

Example 4–13 shows the printf statements in ddb_startup routine in the filesys.c file. The ddb_startup routine initializes the drivers listed in the configuration build file in their respective phase.

Example 4–13 Tracing Driver Startup in filesys.c

```
ddb_startup (int phase) {
        struct DST *dstp;
        int     status;
        int     i;

        for (dstp=dst, i=0; i<num_drivers; i++, dstp++) {
            if (dstp->phase != phase) continue;
#if 0
            qprintf ("calling module %s\n", dstp->name);
#endif
            status = (*dstp->startup) (phase);
            if (status != msg_success) {
                qprintf ("%08X exit status for %s_init\n", status, dstp->name);
            }
#if 0
            qprintf ("exiting module %s\n", dstp->name);
#endif
        }
        return msg_success;
}
#endif
```

Note

There are printf statements also in the ddb_startup_driver routine. This routine starts or restarts an individual driver when given the driver's name. Note that this can fill up the event log if the driver is repeatedly started with a test script. Refer to filesys.c for more information.

Using Polled Mode for Drivers

Polling Overview

When porting the Alpha AXP Firmware, first build an image with the drivers set to use the polled mode for handling interrupts. Using polled mode may help to eliminate unexpected hangs by avoiding the use of hardware, PALcode, and interrupt service routines (ISR).

Setting Polled Mode

The driver database (DDB) structure has a “setmode” member that defines the address of the class driver routine. The class driver routine sets various driver modes:

- Start or stop a driver
- Hardware interrupt or polled method

A developer defines the interrupt handling mode to be `DDB$K_POLLED` or `DDB$K_INTERRUPT` in the port block (PB) for each port driver. The interrupt handling mode definition can be hard coded or defined by a conditional compilation directive and compiled accordingly. Examples of each follow.

Conditional Directive

Example 4–14 shows `DRIVER_MODE` being defined in the combination port driver `combo_driver.c` as the value of `DDB$K_POLLED` ❶. The port block permanent mode is set to this value ❷ to indicate the current status of the port for this instantiation. This value is passed to the set mode routine in `tt_driver.c` class driver ❸.

Example 4–14 Conditionally Defining Polled Mode in a Driver

```
                      /*  COMBO_DRIVER.C  */
.
.
#if DEBUG
#define DRIVER_MODE DDB$K_POLLED ❶
#else
#define DRIVER_MODE DDB$K_INTERRUPT
#endif
                /* connect the read/write routines to the port block */
            ttpb->rxread = combott_rxread;
.
.
            ttpb->perm_mode = DRIVER_MODE; ❷
            ttpb->perm_poll = 0;
            spinunlock(&spl_kernel);
.
.
            /* set the polling/interrupt mode */
            tt_setmode_pb (ttpb, DRIVER_MODE);  ❸
```

Hard Coding the Interrupt Mode

Example 4–15 shows the port block mode being hard coded to the value of DDB$K_INTERRUPT ❷ or DDB$K_POLLED ❸. If the vector has not been set ❶, the software polled method is used.

Example 4–15 Defining Polled Mode in a Driver Using Hard Coding

```
                      /*  DAC960_DRIVER.C  */
if( pb->pb.vector )  ❶
        {
 pb->pb.mode = DDB$K_INTERRUPT;  ❷
 pb->pb.desired_mode = DDB$K_INTERRUPT;
 int_vector_set( pb->pb.vector,
 dac960_interrupt, pb );
 }
else
 {
 pb->pb.mode = DDB$K_POLLED;  ❸
 pb->pb.desired_mode = DDB$K_POLLED;
 }
```

Reference

For more information regarding drivers and DDBs, refer to *Device Driver Interface Guide for the Alpha AXP Firmware* or the *Alpha Firmware Design Document Version 0.10.* Note that the DDB and driver modes are defined in KERNEL_DEF.SDL.

Also note that interrupt request lines (IRQx) are defined in the genipltbl routine within pal_ev4.mar.

Modifying the Build Files

Introduction

This section describes how to modify the build files when building the minimal image for the target platform.

Modifying the Configuration Build File

Adding Drivers, Files and Options

Once additional drivers and other files have been created, they are added to the configuration build file. Other modifications concerning the platform are also made to this file: for example, preprocessor directives for conditional compilation, and constants specific to the platform. These modifications are entered into the file in a format that is slightly different than C programming: they are called option and value statements.

Configuration Build File Example

An example of changes made to `Configuration_files.bld` follows. These changes must be made when porting to a new platform or adding a configuration to the same platform. Callouts label these changes.

Default Options

The file `options.bld` that is included in the configuration file ❶, defines the default option and value statements used during a build. By default, the definitions in `options.bld` are defined as zero.

```
# file: PLATFORM_FILES.BLD --  build file for platform.
#
#       small image, ROM version
include CP$SRC:OPTIONS.BLD ❶
.
.
```

The developer can enable individual options and values with definitions placed in the *Configuration*`_files.bld` file. These statements create individual header files in the `CP$INC` directory by the next phase of the build process: the `newbuild.com` procedure.

Hardware and Architecture Name

Every configuration file contains the name of the hardware platform and its architecture, for example:

```
# file: platform_files.bld --  build file for platform.
#
#       small image, ROM version
include CP$SRC:OPTIONS.BLD
platform SABLE ❶
architecture ALPHA ❷
.
.
```

❶ Platform is the hardware platform identifier.

❷ Alpha or VAX is the architecture name.

Value Statements

Value statements define constants, for example, the number of picoseconds per CPU clock cycle or the size of the heap.

```
value DEFAULT_PSEC_PER_CYCLE 5260 ❶
value MIN_HEAP_SIZE     0x14000    # 80K ❷
```

❶ `DEFAULT_PSEC_PER_CYCLE 5260`

- Defines the number of picoseconds per CPU clock cycle (approximately 190 MHz), used in the timer routine `TIMER.C`

❷ `MIN_HEAP_SIZE 0x14000`

- Defines the size of the memory (81920_{10}) heap used for dynamic memory allocation

Option Statements

Option statements are used like C preprocessor `#def` directives, typically for conditional compilation, for example, adding diagnostic support or for specifying a protocol/server.

```
.
.
option DIAG_SUPPORT
option MSCP
option BOOTP_SUPPORT
.
.
```

The `DIAG_SUPPORT` option conditionally enables the routine that initializes the diagnostic environment variables in `ev_driver.c` used by diagnostics.

Removing and Adding Kernel Files

Files are removed from the build by deleting them from the build file.

Files are added to the kernel source routines with the keyword `file` followed by the file name.

```
# Screen Files

file    allocfree.c                     group base
file    alphamm.c                       group base
file    ansi.c                          group base
```

The keyword `file` indicates a kernel source code file.

Add new source files names (if any) and specify a group.

Groups

The keyword `group` compiles all related files into the same object file, for example, base. This is useful image reduction, saving the inter-module call frame linkage.

```
file    kernel.c                        group base
file    kernel_support.c                group base
```

Group object compilation is typically implemented for a base-level code release, otherwise there would be too much recompilation when changes are made to a single file.

An MMS qualifier `/MACRO=opt`, used in the third phase of the build process, takes advantage of this feature.

Group Driver

The `driver` files are a separate group of platform-specific files that are driver-assist routines. They contain hardware-specific functions, such as bus control register read/write routines, and other routines specific to the platform.

```
file    hwrpbtt_driver.c                   group driver
file    iic_platform_driver.c              group driver
```

Default Build Options and Values

options.bld File

The options.bld file defines the default option and value statements used during a build. This allows newbuild.com to create individual header files in CP$INC directory during the build.

By default, all definitions in options.bld are defined as zero, except for ALPHA_CONSOLE, value, and message statements. The developer can enable individual options and values with definitions placed in the *Configuration*_files.bld configuration build file.

Options and Values

The options.bld file defines the following options and values:

- Platform
- Architecture
- Miscellaneous
- Values
- Messages

Build Options Example

Example 4–16 shows the contents of options.bld that contains the default option and value definitions.

Example 4–16 The options.bld File

```
# options.bld
#
# This file defines the options which are used in the common console build
# environment.  An include file is created for each option in cp$inc.  If
# the option is defined but not invoked, the include file will set the option
# to 0.
#

#
# ALPHA_CONSOLE is always defined.  Use for conditionalizing sources which are
# built in environments other than the common console build environment.
#
define option ALPHA_CONSOLE     # To prevent warning
option ALPHA_CONSOLE

#
# Platform options
#
define option PLATFORM A        # Platform A Rom based image
define option PLLOAD            # Platform A loadable image
define option PLBUPDATE         # Platform B update image
define option COMMON            # Common to all platforms

#
# Architecture options
#
define option ALPHA
define option VAX
```

(continued on next page)

Example 4–16 (Cont.) The options.bld File

```
#
# Miscellaneous options
#
define option ARC_SUPPORT        #
define option CONSOLE_DRIVER     #
define option DIAG_SUPPORT       #
define option DE200              #
define option EXTRA              # Include extra stuff.
define option INTERACTIVE_SCREEN #
define option MINIMAL_MORGAN     #
define option MODE64             # Full 64 bit mode.
define option MODULAR            # Modular console (requires OVERLAY)
define option MSCP               # client MSCP support
define option MSCP_SERVER        # server MSCP support
define option OVERLAY            # overlays
define option RELEASE            # This is a release version (?)
define option TGA                #
define option TGA_VMS_BUILD      #
define option UNIPROCESSOR       # obsolete
define option VALIDATE           # Build "checked" version of code.
define option XDELTA_ON          # Turn XDELTA on.
define option XDELTA_INITIAL     # Modular, take initial Bpt
define option RUNS_ON_EISA       # EISA based system
define option BOOTP_SUPPORT      # Turn on BOOTP, need to build inet driver
define option SYMBOLS            # Turn on Symbols, need to build vlist

#
# Values
#
define value MIN_HEAP_SIZE      0x18000         # 96K
define value HEAP_SIZE          0x100000        # 1024K
define value HEAP_BASE          0

define value DEFAULT_PSEC_PER_CYCLE 8000
define value DISCRETETIMER_BASE 0x4000
define value ENET_ID_BASE       0x3800
define value EEROM_LENGTH       8192
define value NVRAM_EV_LEN       2048
define value MAX_ELBUF          8192
define value MAX_RECALL         16
define value RTC_BASE           0x8000
define value RTC_OFFSET_REG     0x70
define value RTC_DATA_REG       0x71

define value MAX_PHYSMEM_MB     (2*1024)
#
# Regular expressions for suppressing undefined options during option
# scanning (see OPT.MMS).
#
suppress option /DEBUG/
suppress option /TRACE/

#
# Messages
#
message file generic_messages.c

message prefix msg_
```

(continued on next page)

Example 4–16 (Cont.) The options.bld File

```
message value msg_success      0
message value msg_failure      1
message value msg_def          2
message value msg_halt         3
message value msg_loop         4
message value msg_error        5
```

Modifying for Nightly Builds

The build file `build.com` creates a multitude of Alpha AXP Firmware images for a number of platforms on a nightly basis. `build.com` is submitted to a batch queue and invokes procedures and utilities that a developer might invoke manually.

- cp_common_login.com
- newbuild.com
- mms/description=cp$src:descrip.mms

Modifying build.com

To take advantage of the automated nightly build, the new target configuration must be added to `build.com` as illustrated in Example 4–17, (see the notes that follow the example).

Example 4–17 Modifying build.com

```
.
.
$ q$build_val          == "SBMIN,SBFSAFE,SABLE,SBUPDATE,SBLOAD,JENSEN," + -
                          "JNLOAD,MEDULLA,TURBO," + -
                          "PLATFORM❶,MSLOAD❷," + -
                          "COBRA,CBLOAD"
.
.
$ BUILD_PLATFORM: subroutine ❸
$    on warning then exit $status
$    call checkspace 44000
$    call newbuild
$    call mms
$    exit 1
$ endsubroutine
$ !
$ BUILD_MSLOAD: subroutine ❸
$    on warning then exit $status ❹
$    call checkspace 54000 ❺
$    call newbuild MSLOAD ALPHA "PLATFORM EXTRA"
$    call mms PLATFORM
$    exit 1
$ endsubroutine
.
.
```

The following callouts describe the entries placed in the nightly build file `build.com`, as shown in Example 4–17.

❶ Enter the name of the base platform configuration.

- Example: PLATFORM, the compressed, ROM-based Alpha AXP Firmware image for the Platform platform

❷ Create the names of configuration variations of the base platform.

- Example: MSLOAD, a variation that can be downline loaded with MOP protocol

❸ Enter the names of the subroutines called to build the particular configuration.

❹ Upon any warning, error, or severe error, exit this procedure (for the current command level) with the condition code in 32-bit longword global symbol.

❺ Indicate the number of disk blocks required for the image build to the Checkspace routine.

Debugging

Introduction

This section describes the following:

- Using XDELTA
- XDELTA Commands
- XDELTA Symbolic Extensions

Using XDELTA

XDELTA Overview

XDELTA is a debugger for system level, (elevated IPL interrupt priority level) code. XDELTA (like DELTA, the user mode debugger):

- Is nonsymbolic
- Uses the same command syntax
- Displays no visible prompt
- Prints short error messages, for example, "Eh?"

Building with XDELTA

To use XDELTA to debug the Alpha AXP Firmware, add the following to your *Configuration*_files.bld.

- Option XDELTA_ON
- XDELTA files xdelta_isrs.mar and xdelta.mar

XDELTA Example

For example, the following code fragment was extracted from the Sable platform configuration build file sbload_files.bld. It builds an image that can be downline loaded using MOP protocol.

Example 4–18 Including XDELTA in the Build File

```
.
.
.
option XDELTA_ON        1
file    xdelta_isrs.mar
file    xdelta.mar
.
.
.
```

Note that in Example 4–18 the XDELTA_ON option is specifically turned on with a one. However, the following statement is equivalent:

```
option XDELTA_ON
```

Default Options

There are default options defined within the build file cp$src:config.bld. This file defines the XDELTA option as zero. This remains as zero if not redefined in the build file for the configuration. Definitions in the *Configuration*_files.bld supersede definitions in the generic file.

Turning Off XDELTA

XDELTA can be disabled by commenting out its option with a pound sign `# option XDELTA_ON 1` or explicitly setting it to zero `option XDELTA_ON 0` as shown in Example 4–19. The files `xdelta_isrs.mar` and `xdelta.mar` must also be commented out or they are still part of the console image. Another alternative is to remove all three lines from the build file.

Example 4–19 Excluding XDELTA from the Configuration Build File

```
.
.
.
option XDELTA_ON        0
#file    xdelta_isrs.mar
#file    xdelta.mar
.
.
.
```

Invoking XDELTA

To invoke XDELTA, type `bpt` (breakpoint) from the console prompt, for example:

Example 4–20 Invoking XDELTA at the Console Prompt

```
>>> bpt
```

Invoking from OpenVMS

To gain experience with the XDELTA command set, a developer can invoke XDELTA at the user level using a small sample program. A logical must first be defined, as shown in the following example.

Example 4–21 Invoking XDELTA at the User Level

```
$ DEFINE LIB$DEBUG SYS$LIBRARY:DELTA

$ RUN/DEBUG Program_Name

Alpha/VMS DELTA Version 1.5

Brk 0 at 00020000

00020000!      LDA             SP,#XFFD0(SP)
```

XDELTA Commands

Commands

Table 4–5 summarizes some of the typical commands used with XDELTA. This is not intended to be an exhaustive list.

Table 4–5 XDELTA Functions and Commands

Function	Command Syntax	Example
		BREAKPOINTS
Set breakpoint	Address,N;B (N is a breakpoint number from 1 to 8)	**2000C,1;B**
Display breakpoint	;B	**;B** **1 0002000C**
Clear breakpoint	0,N;B	**0,1;B**
Proceed from breakpoint	;P	**;P**

(continued on next page)

Table 4–5 (Cont.) XDELTA Functions and Commands

Function	Command Syntax	Example
		DISPLAYING MEMORY
Open location and display contents	Address/	**200D0/**47E03406
Open location and display instruction	address!	**200D0!** `BIS            R31,#X01,R6`
Open location and display contents in ASCII mode	address"	**200d0"**.4àG
Replace contents of a given address	Address/contents new_contents	Replace 47E03406 with 12abcd78, then with an ASCII 'ab'. **200d0/**47E03406 *12abcd78* **200d0/**12ABCD78 **200d0/**12ABCD78 **'ab'** **200d0/**12AB4241

(continued on next page)

Table 4–5 (Cont.) XDELTA Functions and Commands

Function	Command Syntax	Example
	DISPLAYING MEMORY	
Set display mode: byte, word, longword, or quadword	[B, [W, [L, [Q	. . . Brk 1 at 000200D0 000200D0! BIS R31,#X01,R6 **200D0/**47E03406 = Examine Loc. 200D0 **[B** = Set mode to byte **200D0/**06 = Examine Loc. 200D0 **[W** = Set mode to word **200D0/**3406 = Examine. Loc. 200D0 **[L** = Set mode to longword **200D0/**47E03406 = Examine. Loc. 200D0 **[Q** = Set mode to quadword **200D0/**5FFF041F 47E03406 = Examine. Loc. 200D0 **[A** = Set mode to address **200D0 s** = Step from Loc. 200D0 00000000 000200D8! ADDL R3,#X01,R7
Display contents of previous location	ESC	**200d0/**12ABCD78 ESC$ 00000000 000200CC/6B5A6C41 ESC$ 00000000 000200C8/22020030 ESC$ 00000000 000200C4/47FF0405
Display contents of next location	Line Feed LF	**200d0/**12ABCD41 LF 000200D4/5FFF041F LF 000200D8/40603007 LF 000200DC/2007FFFA
Display indirect	TAB or /	**10000/**00083089 TAB 00083089/8847FF04
Display range of locations	Address_1st, address_last/	**200d0,200e0/**47E03406 000200D4/5FFF041F 000200D8/40603007 000200DC/2007FFFA 000200E0/F800000B

(continued on next page)

Table 4–5 (Cont.) XDELTA Functions and Commands

Function	Command Syntax	Example
	SETTING/DISPLAYING REGISTERS	
Set base register	'value',N;X	**80000000,0;X**
Display base register	Xn Return or Xn=	**X0** **00000003** **X0=00000003**
Display GPR	Rn/ (n is decimal)	**r0/**00000001
Display a group of GPRs	GPR_First,GPR_Last/	**r0,r3/**00000001 R1/00000001 R2/00010000 R3/00000000
Deposit GPR	Rn/xxxxxxxx new_data	**r1/**00000001 **2** **r1/**00000002
	START and STOP	
GO	;G	**20000;G**
Single step into subroutines	S	Brk 0 at 00020000 00020000! LDA SP,#XFFD0(SP) **s** 00020004! BIS R31,#X09,R25 **s** 00020008! STQ R27,(SP)
Single step over subroutines	O	00020040! STQ R0,#X0010(SP) **s** 00020044! JSR R26,(R26) **s** = Error, user trying to step into kernel system space (R26 = 805717B0) Eh? 00020040! STQ R0,#X0010(SP) **s** 00020044! JSR R26,(R26) **o** = Rather step over. 00020048! LDA R27,#XFFA8(R2)
Exit from XDELTA	EXIT	**EXIT**

(continued on next page)

Table 4–5 (Cont.) XDELTA Functions and Commands

Function	Command Syntax	Example
EXPRESSIONS and MISCELLANEOUS		
Show value	+,-,*,%{divide} expression=	**1+2+3+4=**0000000A
Shift a value +left -right	value@shift	**2@2** 00000008 **2@-1** 00000001

Extended XDELTA

Overview

XDELTA has been enhanced to include features such as symbolic lookup/substitution, and other useful features. The additional debug support provided to Alpha AXP Firmware developers allows improved usage for the debugger.

This extended support is not found in the standard version of XDELTA. To use these new features, developers must build the firmware with the supporting files and option symbols as described in *Building with Extended XDELTA*.

Commands

The following commands have been added to the basic commands set of XDELTA.

Table 4–6 Extended XDELTA Commands

Command	Description
;R	Toggle register display mode. If on, the instruction registers are displayed when the instruction is displayed. This also displays the console symbol associated with the procedure address in R27.
;C	Continue back to the instruction following a single step that called a routine, such as JSR or BSR. Install a breakpoint at R26, proceed, when the breakpoint is reached, remove the breakpoint.
U	Until. Do step overs until a routine call or return instruction is found, such as, JSR, BSR, or RET.
T	Trace. Do step overs until a breakpoint is encountered. Routine call instructions are displayed when encountered, but other instruction displays are inhibited.
| Symbol_ Name	Display symbol. Start command input with a vertical bar followed by a symbol which will be looked up in the console symbol table and its value substituted.
| symbol* Return	Display matching symbols and their addresses. If instruction display mode is set, display overlay and offset as well as actual address.
;O	Display the current resident overlays and their base addresses.
;W	Walk stack.
;S	Display processes.

Debug Session Example

Example 4–22 shows an example debug session using the new features and commands of the extended XDELTA.

Example 4–22 Using Extended XDELTA to Debug

```
>>>
>>>set ovly_debug 3
>>>show dev
Load Overlay CIXCD - mem/feab2e0, off/fe576fc, buf/feb9b00

Brk 0 at 00081798

00081798! BPT   U   <- Until
0008179C! RET   R31,(R26) U  <- Until
0004C6F0! LDL   R0,0050(R3)
...     <- (display removed for clarity)
0004C70C! RET   R31,(R26) ;R   <- Enable registers
U
0004C64C! MOV   R29,SP        R29/ 00000000 0FEAAE70   SP/ 00000000 0FEAAE70
0004C650! LDQ   R26,0150(R29) R26/ 00000000 0004C64C  R29/ 00000000 0FEAAE70
...
00048628! BSR   R26,0020CF    R27/ 00000000 00074058  00050968  decc$gprintf U
...
00048634! BSR   R26,-0000C4   R27/ 00000000 00071EB0  00048328  hose_ordinal S
00048328! LDA   SP,-0050(SP)  SP/ 00000000 0FEAAFE0   SP/ 00000000 0FEAAFE0 ;C
                 Continue --^
Brk 8 at 00048638

00048638! LDQ   R1,-0048(R2)  R1/ 00000000 00000000   R2/ 00000000 00071F18
|ovly_debug/00000003    <- Symbol examine
|get_pte;b    <- Symbol breakpoint
;B
 1 00065428
 2 000380B0

|get_pte/00303089
0006E124/00000000
0006E128/000380B0
```

Building with Extended XDELTA

To use the new features of the extended XDELTA debugger, the appropriate source files from the `CP$SRC` area must be added, as follows:

1. Change your build file where it refers to `xdelta.bli` `xdelta.obj` files to use `alpha_xdelta.bli` and `alpha_xdelta.obj` files.
2. In your `'platform'.c` file, add the `con$checkchar` routine.

   This routine checks for a character available and if so, returns it. If no character is available, it returns zero. This should be patterned after the `con$getchar` routine, except that it does not wait for a character to be available.
3. To get symbols, include the following in your build file:
   - `sym_driver.c`
   - `vlist.c`
   - `sym_include`
   - `sym_exclude`
   - `option SYMBOLS`

Command Summary

Example 4–23 depicts the most commonly used commands of XDELTA, along with the additional commands of extended XDELTA.

Example 4–23 Extended XDELTA Command Summary

```
Primary Commands                    Secondary Commands
================                    ==================
                                    NU = System Xdelta, Not used
0-9      numbers
a-f      numbers                    ;b      breakpoint
f        fp_regs                    ;c NU   bug check
.        current loc                ;x      x regs
q        last quantity              ;l NU   loadable images
r        register                   ;e      execute
x        x register                 ;g      go
g        system space               ;h      crt/hardcopy
h        p1 space                   ;i NU   shareable
,        end of current field       ;m NU   writes allowed
+ or sp  add                        ;p      proceed
@        shift                      ;q NU   queue
*        multiply                   ;w NU   locate sys adr
%        divide
-        sub                        ;r      toggle register display
cr       close                      ;c      bpt at (r26) ;p
lf       next                       ;o      display overlays
esc      previous                   ;w      walk stack
tab      indirect                   ;s      display processes
/        open
"        ascii
!        instruction                Instruction Stepping
=        current                    ====================
;        secondary
[        new display mode           S       single step
:        pid                        O       step over
p        proc reg                   U       step until
'        ascii deposit              V       step until (don't print)
\        quote                      T       trace

Symbols
=======

|              symbol command (precedes symbol)
|sym(op)       (op) maybe any other single char operator
|sym*(cr)      display symbols
|sym;b         set breakpoint at code adr of PD

Display Mode
============

b       byte
w       word
l       longword
q       quadword
i       instruction
c       char/ascii
a       64 bit address
```

5 Debugging the Hardware

Overview

Introduction

This chapter describes the following:

- Hardware Debug Techniques
- SROM and its Function
- SROM Mini-console

Hardware Debugging Techniques

Overview

Debugging the hardware should include the following:

- A thorough visual inspection of the system
- Power and ground voltage checks
- Clock signal frequency check
- Use of the SROM mini-console for simple deposit/examine functions to memory and I/O control registers

Serial ROM

Introduction

This section describes the functionality of the Serial ROM used to load the Alpha AXP Firmware image.

Serial ROM (SROM) Functionality

Introduction to Serial ROM

A processor loads Serial ROM (SROM) code into its instruction cache and starts execution. The function of the SROM is to test the path used to load the console and then pass control to the console. If control cannot be passed to the console, SROM identifies the area at fault and the Field Replaceable Unit (FRU) that needs to be replaced.

Since much of the SROM code is specific to the hardware platform and CPU configuration, such as memory size and cache timing, the SROM code must be modified.

Functions of Serial ROM Code

Specifically, when testing the base system hardware, the SROM code:

1. Initializes, accesses and tests the processor(s) and backup cache (Bcache)
2. Initializes the path to the console stored in Flash EEPROM(S)
3. Identifies space in the backup cache or memory in which to load the console
4. Loads the console image and transfers control

Note

Optionally, a data block may be set up to define the current state of write-only processor registers (IPRs), as well as other configuration information. This data block is passed from SROM to the console, and later, to the operating system.

Loading into Bcache

When the console image is loaded, it may be loaded into backup cache instead of memory. For example, when SROM is configuring the system, it can turn on the backup cache, loading the console into faster memory.

It is important to load the console image into backup cache for purposes of accessing I/O devices using mailboxes and memory interleaving.

Up to this point, only one memory module is enabled and a portion of the memory has been briefly tested. Complex memory testing is done later by the console.

SROM Mini-Console

Introduction

This section describes the Serial ROM (SROM) mini-console.

SROM Mini-Console

Overview

There are a number of ways to debug the basic hardware: diagnostic bus, a dedicated microprocessor, or with the SROM mini-console. This section explains the SROM mini-console and simple connection to the serial port on the microprocessor.

Mini-Console

The Alpha AXP SROM mini-console provides basic hardware debugging capability through a serial connector interface to the SROM serial port of the Alpha AXP microprocessor. Using a minimum of hardware you can exercise cache, memory, and I/O subsystems until the system is functional enough to support a more fully featured console program.

The mini-console resides in the instruction cache (I-cache) and is designed to be loaded at reset through the SROM interface directly into the I-cache.

Features

The mini-console provides:

- Commands to examine and deposit data in memory and internal CPU registers
- A case-independent command language
- Support for variable baud rates and processor speeds

Hardware Required

To use the SROM mini-console, you need the following:

- Alpha AXP DECchip
- An SROM containing the mini-console
- A clock source
- A level shifter to convert from TTL signal levels to EIA
- A connection from the DECchip SROM interface to a serial terminal, any standard terminal or a workstation with a terminal emulator utility, such as DTE or tip

Setting Up the SROM Serial Port Connection

Using a Workstation for the Connection

To use the mini-console, you must first establish a connection from your Alpha AXP system SROM serial port to a suitable terminal device or emulator. A workstation can be used for a serial line connection to the SROM signal lines of the DECchip 21064 using a tip or SET HOST/DTE connection.

Connecting the Workstation Procedure

Example 5–1 Connecting the Workstation

```
❶ $ SET PROTECTION=(S:RWLP,O:RWLP,G,W) /DEVICE/OWNER=Your_ACCNT_NAME TTA3:
❷ $ SET HOST/DTE TTA3:
  %REM-I-TOQUIT, connection established

  Press Ctrl/\ to quit, Ctrl/@ for command mode
❸ U
  SROM>
```

In Case of Problems

If problems occur, use CONTROL@ for DTE command mode, and use the DTEPAD>SHOW DTE command to check port settings.

The following notes clarify the commands used in Example 5–1, when connecting a workstation to the SROM serial port with the OpenVMS operating system.

❶ First ensure that the asynchronous device has the correct device protection for user access. This allows you to assign a channel to the specified terminal port. Setting device protection requires (OPER) operator privilege.

Connect a DEC423 6 conductor serial cable to the modified modular jacks (MMJ) between the workstation and the Alpha AXP system.

❷ Establish a connection using the command:

```
$ SET HOST/DTE terminal_port#
```

❸ Type the SROM mini-console “U” command to allow autobaud, receiving the SROM mini-console prompt.

Reference

Refer to *Alpha AXP SROM Mini-Debugger User's Guide* for more information about the SROM mini-console, such as command examples and connecting a UNIX workstation to the SROM serial port.

Starting and Running the Mini-Console

After the SROM serial port connection has been made, you can initialize the mini-console by typing an uppercase U. This returns an SROM> prompt, which indicates that you are ready to begin debugging hardware.

For example:

```
U
SROM>
```

The uppercase U automatically detects the baud rate. Baud rates up to 19.8K are supported. Prior to the first command prompt, synchronization is affected by typing an uppercase U at the terminal whenever the Alpha AXP microprocessor is turned on.

SROM Mini-Console Command Set

Command and User Interface Features

The following list describes some features of the mini-console command language.

- Uppercase or lowercase characters can be used.
- The delete key provides primitive command-line editing.
- Numbers are input and output in hexadecimal format.
- For commands that prompt for input, the default input value is all zeros.
- Addresses are masked on an even longword boundary in `lw` mode and on an even quadword boundary in `qw` mode.
- Data and address inputs are taken one longword at a time.
- The looping commands initiate an infinite loop. To exit, press any key.

Command Summary

Table 5–1 summarizes the command set for the Alpha AXP SROM Mini-console.

Table 5–1 Command Summary

Command	Function
`dm`	Deposits data to one memory location
`em`	Examines one memory location
`qw`	Sets quadword data mode (64-bit data)
`lw`	Sets longword data mode (32-bit data)
`fm`	Fills a block of memory with a data pattern
`bm`	Displays a range (block) of memory locations
`hm`	Sets high-memory mode (64-bit address)
`lm`	Sets low-memory mode (32-bit address)
`ba`	Gets a base address and sets the base address mode flag
`sa`	Sets the base address mode flag
`ca`	Clears the base address mode flag
`wm`	Performs a looping write to one memory location
`rm`	Performs a looping read from one memory location
`!m`	Performs a looping write/read sequence at one memory location
`dc`	Deposits data to one CPU register
`ec`	Examines contents of CPU registers
`xm`	Loads an external image to memory
`st`	Starts execution at an address

6
Adding Functionality

Overview

Introduction

Once a minimal console is running and the console prompt is displayed on the console terminal, functionality can be added to create a fully operational console.

Functionality should be added incrementally. For example, when adding drivers, each driver should be added one at a time. In addition, each driver should first be brought up in polled mode and, once operational, should be switched to interrupt mode for ongoing use.

After Adding Functionality

Once you have added the necessary functionality, stabilize the code as follows:

- Powerup - Set robust mode on so drivers come up automatically.
- Drivers - Run drivers in interrupt mode.
- Machine Check Routines - Machine check routines must be modified to dump platform specific status and error registers.
- PALcode - Deal with any other disabled error interrupts and exceptions such as CRDs.
- `hwrpb.c` - Modify for your platform.
- All Modules - Check all modules that you build with to ensure they include the correct `platform.h` files.

Adding Functionality

Introduction

This section describes the following:

- Selecting the Console Device
- Adding Drivers to Build Files
- Developing and Adding Scripts
- Adding ARC Support
- Modifying for Bootstrapping
- Adding and Modifying Console Commands
- Programming Diagnostics

Selecting the Console Device

VGA or Character Cell

The Alpha AXP Firmware uses bus sniffing routines at startup to identify whether a Video or a serial ASCII communication port exists. The console will use the Video Graphics Adapter (VGA) by default, unless it is at fault or is configured otherwise. The Windows NT operating system requires a VGA, while OpenVMS and DEC OSF/1 do not, (unless DECwindows or X windows are installed and enabled).

It is suggested to initially use the serial communication port and a character cell terminal to expedite getting to the console prompt for the first time.

Developers should ensure that terminal device drivers are listed in the proper order in the `platform_files.bld` build file.

Forcing Output to the Serial ASCII Port

To force the console to use the serial communication port (for example, COM1), do one of the following:

- Set the CONSOLE environment variable to the value SERIAL.
- Unplug the keyboard of the graphic terminal from the connector and then reinitialize the system. (Since the console checks a flag to see if a keyboard is present and a VGA device exists, unplugging the keyboard forces the serial terminal to become the primary console communication device.)

Note

To return to using the graphic terminal, set the CONSOLE environment variable to the value GRAPHIC.

There are various terminal/graphic configurations that could exist, for example, both VGA and serial terminal, or multiple VGA boards, possibly on different buses.

Terminal/VGA Driver Sequence

All terminal drivers are phase three drivers in which class drivers should start before port drivers.

The sequence of driver startup should be defined in the *platform*`_files.bld` build file as follows:

1. `TT_DRIVER.C`, terminal class driver
2. All PCI bus graphic drivers (in any sequence)
3. EISA/ISA bus graphic drivers
4. `KBD_DRIVER.C`, keyboard driver

5. `COMBO_DRIVER.C` (or other serial port driver)

Note

A graphic driver must locate the device it supports, then call the `primary_graphics_console_sel` console selection function. This function fills in the HWRPB and selects the console device using the console selection algorithm.

Console Selection Algorithm

The console selection algorithm adheres to the following guidelines when selecting the primary console device:

- The display controller in the lowest numbered PCI slot is selected, unless that device exhibits faults.
- PCI graphics devices have precedence over ISA/EISA bus graphics devices.
- Option cards and built-in (embedded) graphics controllers are handled exactly alike, the console does not distinguish between them.
- If no graphic device exists, is at fault, or the keyboard is not present, the serial port is used.
- If the console has been forced to use the serial port, it will use the serial port regardless of the graphics devices present.
- If there are multiple serial ports, the first one, COM1, is used.

Adding Drivers to the Build Files

Adding Drivers Description

Drivers are added to the configuration build file with the keyword `driver`, followed by the name of the driver source file, its phase to be started when initialized, and its group name `driver`.

Drivers should be added in the order that they are to run when its `driver_init` process is started. Phase zero drivers are initialized very early in the startup process, the null driver being the first. Phase five drivers are the hardware drivers that need to be initialized last.

```
#
#       Lines that have a name in column 0 and a value as the second
#       field are entered into the Driver Startup Table (dst).  The second
#       field indicates what phase the driver is started in.
#
driver  nl_driver.c             0       group driver
driver  rd_driver.c             0       group driver
driver  toy_driver.c            0       group driver
driver  eisa_driver.c           0       group driver
driver  buf_driver.c            0       group driver
```

Format

Use the following format to add new drivers:

`driver source_filename startup_phase group name`

Table 6–1 describes these fields.

Table 6–1 Driver Keywords within Build File

Field	Description
The keyword `driver`	Indicates a driver file
Source_filename	The name of the driver source code to be compiled
Startup_phase	The phase number, (or order) in which the driver should be started when booting the console software
Group name	The object file group name; all files in the same group are created as one large object file, if the MMS qualifier /???? is applied

Developing and Adding Scripts

Encapsulating Commands

A developer creates a script file of various commands, assigning a command name that will invoke the script when typed from the console prompt. An example is the `memexer_sable.` file, which could contain the following console commands:

```
#Memory
set d_report full
echo "Testing the memory"
memtest -bs 100000 -rb -p 0 &
```

Adding Scripts

This memory exerciser test script is added to the configuration file and is macro coded into the Alpha AXP Firmware image when built. The script is added with the keyword `encapsulate`, the name of the script file, its command name, its file attributes, and `-script`, for example:

```
encapsulate cp$src:memexer_platform memexer ATTR$M_READ|ATTR$M_EXECUTE -script
```

Invoking Scripts

To invoke this memory exerciser script `memexer_platform`, use the console command:

```
>>> memexer
```

Inodes

Encapsulated files do not have a file extension, for example, the `memexer_platform.;1` file. An inode is created in the in-memory file system that describes the location, length, and attributes of the encapsulated file.

Adding ARC Support

Adding ARC Support Description

Two different mechanisms are used to boot the Advanced RISC Computing (ARC) compliant console and Windows NT operating system.

- Always boot the Alpha AXP architecture compliant console, then boot ARC
- SROM code loads either Alpha AXP Firmware or ARC console from different ROMs

Both mechanisms share NVRAM environmental variables. The platform that uses the following example always boots the Alpha AXP Firmware and checks the operating system type. For example, the following command sets the operating system type to `nt`:

```
>>>set os_type nt
```

This can also be set to OpenVMS or OSF.

Reference

A template for the `Platform_files.bld` file can be found in the CMS library or in:

```
   CP$SRC:*_FILES.BLD
```

Adding and Modifying Console Commands

Introduction

You can add new commands or modify existing commands, depending on the specific requirements of your platform. For example, when porting the console to a target platform that has a unique device configuration, you may want to add your devices to the list of devices that can be accessed with the examine command.

Process Overview

An overview of the process that you must follow when adding or modifying a console command is as follows:

1. Write the new routine or modify the command file associated with the command you would like to change.
2. Insert new header information in specifically formatted comments at the beginning of the command file that lists the routine name, the access field, the stack size and the command name.
3. Place the name of the command file in the build list, if it is not already listed. The build process builds a new command table that is then used by the file system initialize routine to create inodes.

Existing Command Files

The common commands that you can modify and their corresponding command files are shown in the table.

Table 6–2 Command Files

Command	File
`boot`	`boot.c`
`continue`	`continue.c`
`deposit`	`deposit.c`
`examine`	`examine.c`
`help`	`help.c`
`man`	`man.c`
`initialize`	`initialize.c`
`set`	`set.c`
`start`	`start.c`
`test`	`test.c`

Modifying for Bootstrapping

Introduction

Once the console is initialized and a console prompt is displayed on the console terminal device, there should be little or no additional changes needed to bootstrap the operating system.

Modifications

Changes that may have to be made relate to areas that are platform specific.

Specifically, changes may need to be made to the:

- HWRPB through the HWRPB setup routine to ensure that the data structures fed into the HWRPB conform to the target platform.
- List of legal boot devices in the boot command file when adding a new boot device whose type is not listed.
- Device driver initialize routine for the new boot device to ensure that the exact pathname to the physical location of the device is created in the inode for that driver. (For boot devices, it is critical to have the exact pathname listed.) This information is later used by the operating system for identifying the boot device.

Programming Diagnostics

Introduction

Diagnostic services are available for developing new diagnostic tests that can be run from the firmware.

Diagnostic services provide an environment for programming diagnostics. Only a small subset of diagnostic services are required to be used by any diagnostic.

Diagnostic Service Routines

The following table shows service routines that are provided.

Table 6–3 Diagnostic Service Routines

Routine	Function
`diag_init`	Initializes the diagnostic environment
`diag_print_end_of_pass`	Prints the diagnostic end-of-pass message
`diag_print_status`	Prints a diagnostic status message
`diag_print_test_trace`	Prints the diagnostic test trace message `diag_report_error_lock()` and releases lock
`diag_start`	Starts execution of the diagnostic tests
`report_error`	Displays generic error reporting routine
`report_error_lock`	Displays generic error reporting routine with write lock on the standard error channel left taken out
`report_error_release`	Provides complete error report started by `report_error_lock()` and releases lock

Diagnostic Models

Introduction

Depending on the type and purpose of the diagnostic, a diagnostic can be written according to one of two models:

- Structured/test-directed (diagnostic with a test structure)
- Unstructured (diagnostic without a structure)

Test-Directed Diagnostics

The test-directed model provides a structured test environment with all control functions being performed by the diagnostic services instead of the diagnostic program.

In the test-directed model, a diagnostic dispatch table is set up to describe each test, and the `diag_start` routine performs test dispatching.

The `diag_start` routine also prints test trace and end-of-pass messages, if enabled, and provides for loop-on-error control when returned from a test with the appropriate status message.

Unstructured Diagnostics

The second model has no structure imposed on it and is used mainly by exercisers and utilities. Since no structure is imposed in this model, less control is performed by the diagnostic services, and more must be performed by the diagnostic program itself.

A diagnostic dispatch table is not necessary, and `diag_start` is not used for dispatching. If the diagnostic program wants to print the test trace and end-of-pass message and does not call `diag_start`, the program must call the `diag_print_test_trace` and `diag_print_end_of_pass` routines directly.

Describing Data Structures

Introduction

The diagnostic services use the following data structures:

- Process Control Block (PCB)
- DIAG_EVS
- IO Block (IOB)
- Dispatch Table

Process Control Block

The Process Control Block (PCB) is the primary data structure used by diagnostics and any other process that may want to report an error using the standard error reporting service `report_error`.

The PCB contains a number of fields used to provide process information.

The PCB also contains fields that contain diagnostic run-time environment information, status information and error information. Status information is either automatically updated by the diagnostic services or by the diagnostic program. The status information is used by the status monitoring program to display the status of running processes.

Error fields must be updated by the diagnostic program before reporting an error.

Diagnostic Restrictions

Restrictions Description

The diagnostic routines have the following restrictions:

- Before calling `diag_init`:
  - Load `pcb$a_dis_table` with dispatch table
  - Load `pcb$a_sup_dev` with list of supported devices
- After calling `diag_init`:
  - Load `pcb$a_rundown` with local rundown (cleanup) routine (can do before `diag_init` but results in unnecessary IOB errors)
  - Load `pcb$b_dev_name` array if user did not specify a device (if `pcb$b_dev_name` array is NULL after `diag_init` call, you must load this with a device name)
  - If test does not use `fopen`, you must `malloc` space for IOB and call `create_iob`
- In local rundown (cleanup) routine:
  - Call `diag_rundown` **before** removing the IOBs (`diag_rundown` prints out a completion message and must print out some fields contained in the IOBs)

Related Console Services

Introduction

The console routines most likely to be used by the diagnostic programmer are grouped as follows:

- I/O
- Environment Variable Manipulation
- Dynamic Memory Allocation
- Condition Handling
- Timer Services
- Semaphores
- Multiprocessor
- Ctrl/C/kill Checking

Routines

The specific routines most likely to be used by the diagnostic programmer are shown in the table.

Table 6–4 Diagnostic Programmer I/O Routines

Routine	Function
`fopen`	Prepares a device for use by the other I/O routines. In preparing the device, a file descriptor is returned for future reference to the device.
`fclose`	Closes the device associated with the designated file descriptor for further access.
`fread`	Allows multiple bytes or blocks of data to be read from the I/O device designated by the file descriptor and written to a buffer in memory.
`fwrite`	Allows multiple bytes or blocks of data to be written to the I/O device designated by the file descriptor and read from a buffer in memory.
`fseek`	Allows the file offset to be positioned for subsequent reads and writes.
`ftell`	Returns the current file offset.
`printf`	Allows the specified string to be printed. Its arguments and effect is the same as the standard C `printf` function.
`read_with_prompt`	Allows the program to prompt the user for input.

Table 6–5 Environment Variable Manipulation

Routine	Function
`ev_write`	Causes the specified environment variable to have the specified value. If the environment variable does not already exist in the environment variable namespace, it is created with the specified value.
`ev_delete`	Clears the specified environment variable from the environment variable namespace.
`ev_read`	Returns the value of the specified environment variable to the caller.

Table 6–6 Dynamic Memory Allocation

Routine	Function
`dyn$_malloc`	Causes the amount of memory specified to be allocated to the caller. A pointer to the memory block is returned to the caller. `dyn$_malloc` is modeled after the standard C `malloc` function.
`dyn$_realloc`	Trims or expands a block of memory that was previously allocated with the `dyn$_malloc` routine. A pointer to the trimmed or expanded memory block is returned. `dyn$_realloc` is modeled after the standard C `realloc` function.
`dyn$_free`	Frees a block of memory previously allocated by `dyn$_malloc` and returns the block to the memory pool.

Table 6–7 Condition Handling

Routine	Function
`exc_vector_set`	Establishes an exception routine to be associated with the specified exception type
`exc_vector_clear`	Clears the exception routine established by the last `exc_vector_set`
`int_vector_set`	Establishes an interrupt routine to be associated with the specified interrupt
`int_vector_clear`	Clears the interrupt routine established by the last `int_vector_set`

Table 6–8 Timer Services

Routine	Function
`krn$_sleep`	Causes a delay of the specified number of milliseconds

Table 6–9 Semaphores

Routine	Function
`krn$_seminit`	Initializes a semaphore
`krn$_semrelease`	Releases a semaphore
`krn$_wait`	Waits on the specified semaphore
`krn$_post`	Posts (signals) the specified semaphore

Table 6–10 Multiprocessor

Routine	Function
`krn$_setaffinity`	Sets the affinity mask for the calling process
`primary`	Returns a value to indicate whether the caller is currently running on the primary processor in a multiprocessor configuration

Table 6–11 Ctrl/C/kill Checking

Routine	Function
`killpending`	Checks if a Ctrl/C or `kill` command has been entered, and if so, terminates the process

7 Appendices: Helpful Commands, Tips and Routines

Appendix A: Logical Search String Definitions

Building Environment Logicals

Logicals and subdirectories created by `CP_COMMON_LOGIN.COM` are referred to throughout the build process by automated procedures. Optionally, the developer can use these to check files, build results, and log files.

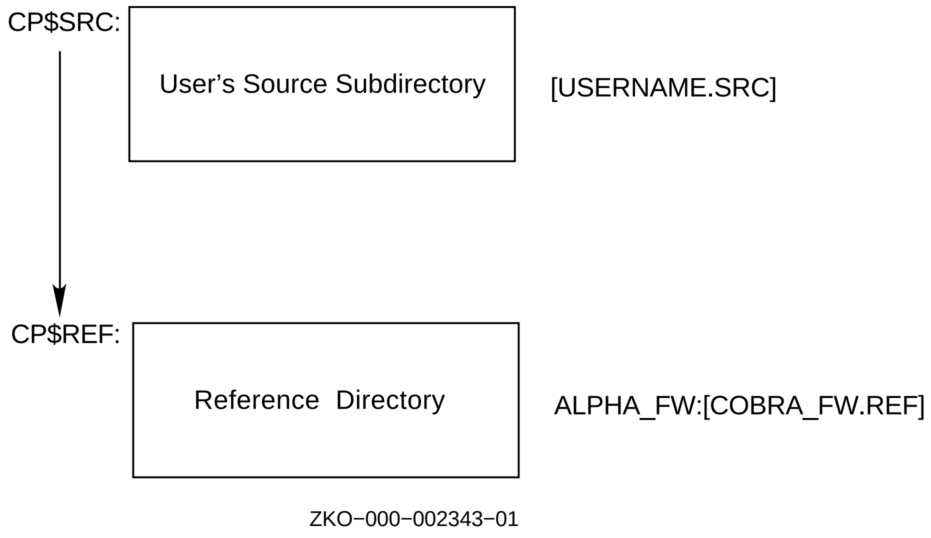
CP$SRC Generic

`CP$SRC` defines the directories (search string) to find the source code when building the Alpha AXP Firmware. For the basic generic case, when no parameters are specified to `CP_COMMON_LOGIN.COM`, the logical `CP$SRC` is defined as follows:

```
"CP$SRC" = "CPUSER:[USER.SRC]" = "CP$REF"
```

Source code is read from the user's source subdirectory (`.SRC`), having precedence over source code in `CP$REF`. `CP$REF` is the latest reference copy of recently replaced source modules in the CMS library. Figure 7–1 shows the search string of the source directories in a generic environment.

Figure 7–1 CP$SRC Generic Environment



CP$SRC Specified Configuration

When the configuration (SABLE), working directory ([USER.AFW]), and backing tree (FRIDAY) are specified with parameters to CP_COMMON_LOGIN.COM, the logical CP$SRC search string is defined as follows:

```
"CP$SRC" = "CPUSER:[USER.AFW.SABLE.SRC]"
      = "CPUSER:[USER.AFW.SRC]"
      = "CP$CFG"
      = "CFW:[CONSOLE.FRIDAY.SABLE.SRC]"
      = "CFW:[CONSOLE.FRIDAY.SRC]"
      = "CFW:[CONSOLE.FRIDAY.COMMON.SRC]"
```

Figure 7–2 illustrates how the source code can be located from multiple directories, and in a top-down order. Development code could exist in the user’s hardware platform source subdirectory ([.SABLE.SOURCE]), and/or in the user’s source subdirectory ([.SRC]). In the search string, each subdirectory has precedence over the next one down. The build will use the first occurrence of the source module located, and the search for that module is discontinued. If the module is not located in the platform’s source or user’s working directory, the search continues until finally the backing tree’s directories are searched [CONSOLE.FRIDAY...], and the logical search string is exhausted.

Figure 7–2 CP$SRC for Specified Configuration Environment

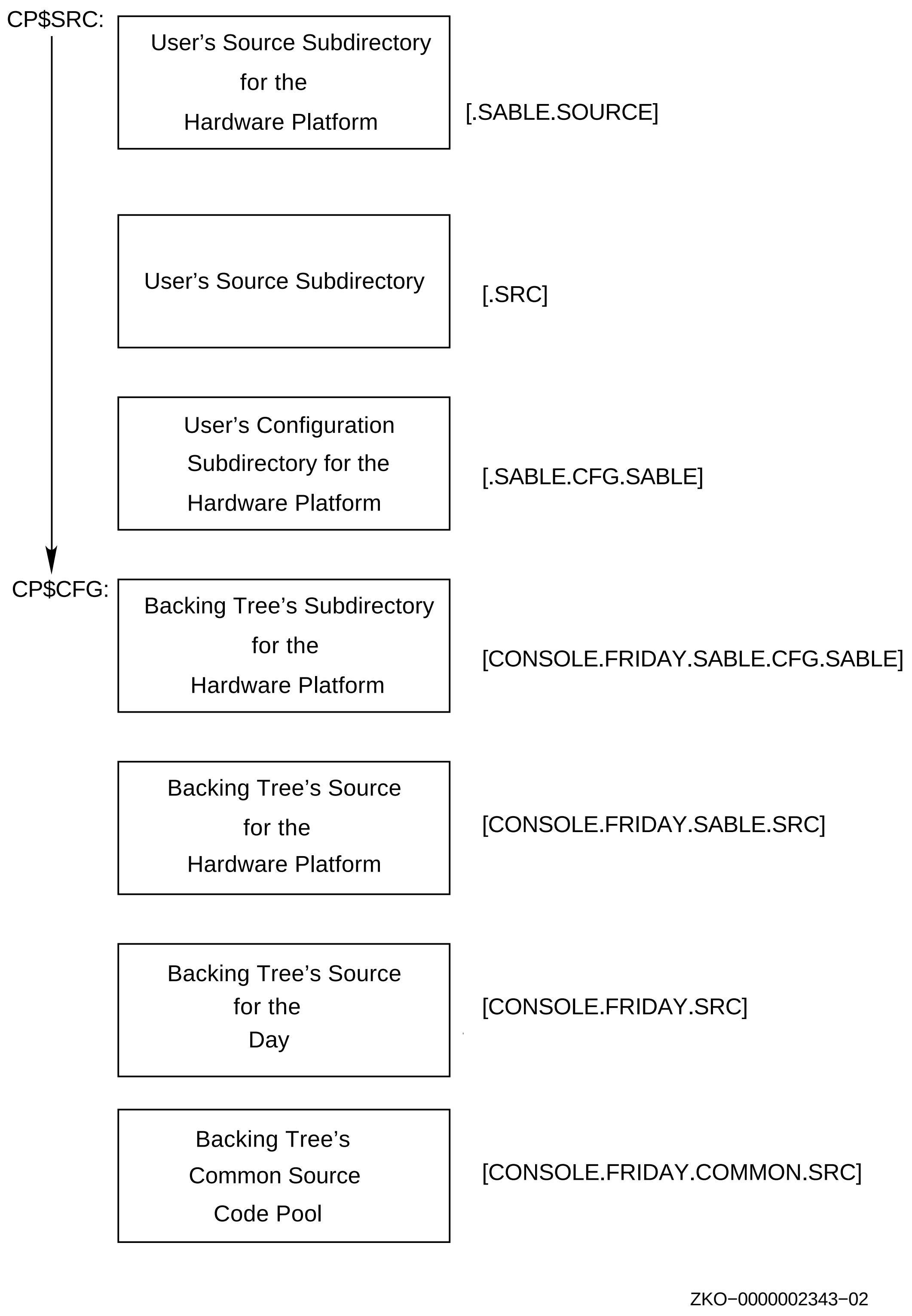

CP$SRC:
User's Source Subdirectory for the Hardware Platform
[.SABLE.SOURCE]
User's Source Subdirectory
[.SRC]
User's Configuration Subdirectory for the Hardware Platform
[.SABLE.CFG.SABLE]
CP$CFG:
Backing Tree's Subdirectory for the Hardware Platform
[CONSOLE.FRIDAY.SABLE.CFG.SABLE]
Backing Tree's Source for the Hardware Platform
[CONSOLE.FRIDAY.SABLE.SRC]
Backing Tree's Source for the Day
[CONSOLE.FRIDAY.SRC]
Backing Tree's Common Source Code Pool
[CONSOLE.FRIDAY.COMMON.SRC]
ZKO−0000002343−02

Table 7–1 describes the logical assignments and files produced by `CP_COMMON_LOGIN.COM` command procedure.

Table 7–1 Building Environment Logicals

Logical	Description
CP$	Top level of user's working directory
CP$CFG	All hardware platform configuration data; includes backing tree
CP$CFGL ‡	User's hardware platform configuration data
CP$CMD	DCL command and GNU AWK build procedures
CP$CMS	CMS Library of source files
CP$EXE	Platform images (`.EXE` loadable/bootable kernel and PALcode), `.MAP` (memory allocation listing), and `.STB` (symbol tables) files; includes backing tree
CP$EXEL ‡	User's platform images `.EXE`, `.MAP`. `STB`
CP$INC	All included header files for platform and backing tree
CP$INCL ‡	User's included header files for platform
CP$KITS	All tools necessary for the build environment
CP$LIS	All program listings for all routines; includes backing tree
CP$LISL ‡	User's platform listings
CP$LOG	Platform's build log files, for example: • `SABLE_BUILD.LOG` • `SABLE_MMS.LOG` • `SABLE_NEWBUILD.LOG`
CP$OBJ	Compiled object code for platform and backing tree
CP$OBJL ‡	User's platform compiled object code for platform
CP$REF	Latest copy of all CMS replaced sources
CP$ROOT	Top level user (build) working directory
CP$SDML	Documentation extracted from routines automatically with `awk` procedures during build
CP$SPECIFIC	Same as CP$ROOT, top level user (build) working directory

‡Logicals appended with an "L" suffix, (CP$CFGL, CP$EXEL, CP$INCL, CP$LISL, CP$OBJL, CP$SRCL), specify the user's subdirectories created for the platform configuration specified. This does not include the reference and backing trees, so cleanup is easily performed using these logicals.

(continued on next page)

Table 7–1 (Cont.) Building Environment Logicals

Logical	Description
CP$SRC	Source code search string for build. Note order of directories: 1. User's platform 2. User's source 3. User's and backing tree's hardware platform configuration 4. Backing tree's platform source, source, and finally the common source directory
CP$SRCL ‡	User's platform source directory
CP$SYNC	Source routines that create the software version. This can be seen in the output from `DESCRIP.MMS`
CP$TMP	Temporary scratch area

‡Logicals appended with an "L" suffix, (CP$CFGL, CP$EXEL, CP$INCL, CP$LISL, CP$OBJL, CP$SRCL), specify the user's subdirectories created for the platform configuration specified. This does not include the reference and backing trees, so cleanup is easily performed using these logicals.

Caution

Use of logicals is encouraged, however, exercise care when deleting files. For example, `CP$SRC` (and others) can equate to the CMS reference and backing tree directories.

Use `CP$SRCL` and other logicals appended with an "L" suffix. They specify only the user's subdirectories. Use these to delete files in your working directory.

Appendix B: Getting Started with CMS

Overview

The Digital Code Management System (CMS) is a library system for software development and maintenance that provides an efficient method for storing project files and tracking all changes to those files. CMS stores source files in a library, keeps track of changes made to the files, and records user access to the files.

A CMS library is an OpenVMS directory containing specially formatted files that CMS uses to operate. Once you have created a library, it must be reserved exclusively for use by CMS.

Hierarchy

The CMS library contains elements (or files), optionally arranged in groups.

Creating CMS Libraries

A CMS library can be created in two steps, as shown in the table.

Table 7–2 Creating a CMS Library

Step	Command	Function
1.	`$ CREATE/DIR [.CMS]`	Executes the DCL command to create the CMS directory
2.	`$ CMS CREATE LIBRARY _` Directory spec: [.CMS] _Remark: CMS Library for Common Console	Creates a CMS directory and inputs a remark for the library description

Setting the Library

The command procedure `CP_COMMON_LOGIN.COM` sets the CMS library for you. To use an alternate CMS library that you created, execute the CMS set library command for the specified CMS directory, for example:

```
$ CMS SET LIB [.CMS]
```

```
%CMS-I-LIBIS, library is USER$COM:[COMMON_CONSOLE.CMS]
%CMS-S-LIBSET, library set
-CMS-I-SUPERSEDE, library list superseded
```

Using CMS with DECwindows

If you have installed DECwindows on your system, you can also use the CMS DECwindows interface (with CMS Version 3.2 and higher). Use the /INTERFACE=DECWINDOWS qualifier to invoke the CMS DECwindows interface.

CMS Class Definition

A class is a set of specific element generations. It can be used to define a system version, such as a base level, consisting of different generations of several elements. An element generation can belong to zero, one, or several classes, but a class may contain no more than one generation of a given element.

Show Classes

The following classes can be found within the Alpha AXP Firmware CMS library CP:[COBRA_FW.CMS] with the command:

```
$ CMS SHOW CLASS
```

Example 7–1 CMS SHOW CLASS Output

```
Classes in DEC/CMS Library CP:[COBRA_FW.CMS]

BUSCON          "buscon snapshot"
CBPROTO_10_18   "CB proto initial build"
CFW_AUG12       "Cobra pre-ESP build."
.
.
V3.4            "Start for the v3.4 firmware release"
V9.9            ""
X1.1-23187      "Group 0-A Dec 20"
X1.2-24092      "Group 0-C Jan 08 - MP Support"
$
```

CMS Groups

Example 7–2 shows the groups within the CMS library. Groups contain the elements of source code files.

Use the show command to list the CMS groups in the Alpha AXP Firmware CMS library, for example:

```
$ CMS SHOW GROUP
```

Example 7–2 Common Console Code Groups

```
Your CMS library list consists of:

   CP:[COBRA_FW.CMS]

Groups in DEC/CMS Library CP:[COBRA_FW.CMS]

BASE_LOGS        "benchmark comparison log files"
CURRENT          "Cobra split current list."
EV5_PAL          "EV5 PALcode elements"
GAMMA_SROM       "New group to contain all gamma SROM files"
MEDULLA_BFILES   "medulla build files"
MEDULLA_SROM     "Medulla Single Board Computer Serial ROM"
MICRO            "new group for mini console code (87C652)"
SABLE_LOGS       "base log files for sable regression scripts"
SABLE_SCRIPTS    "sable regression test scripts"
SER_CON          " new group for serial console code (87C654)"
SPORT            "TURBO SPORT specific.  Includes SPORT controller, GROM, and SROM."
SROM             "Make a separate group for the srom files"
TEST_CMD         ""
TGA              "tga console driver"
```

CMS Elements

Modify the show group command to list the elements within a group, for example:

```
$ CMS SHOW GROUP/CONTENTS

Groups in DEC/CMS Library CP:[COBRA_FW.CMS]

TGA              "tga console driver"
    TGA.
    TGAFONT.H
    TGAX.
    TGA_BL0.TXT
    TGA_DRIVER.C
    TGA_DRIVER_BT463.C
    TGA_DRIVER_BT485.C
        .
        .
    TGA_TEST_VERIFY.C
    TGA_TEST_VRAM.C
    TGA_VERSION.C
```

Appendix C: Complete List of Modified Files

Overview

The following list of 52 files were modified to port the Alpha AXP Firmware to a particular platform: an Alpha AXP PC.

Example 7–3 Ported Files

```
aputchar.c;12  build.com;34  callbacks_alpha.mar;33
call_backs.c;28  combo_driver.c;31 dv_driver.c;6
eisa.h;16  eisa_driver.c;10 entry.c;23
er_driver.c;27  ev5_ipr_driver.c;3 ev_action.c;84
ew_driver.c;72  exer.c;15  filesys.c;12
hwrpb.c;45  ide_driver.c;21  ipr_driver.c;11
isacfg.c;5  isacfg.h;1  kbd_driver.c;36
kernel.c;113  kernel_alpha.mar;31 kernel_support.c;12
lca4_mustang.mar;4 memconfig_mustang.c;6 mop_driver.c;16
msload_files.bld;10 mustang.c;23  mustang.sdl;6
mustang_files.bld;18 mustang_io.h;8  mustang_util.mar;3
n810_driver.c;46 net.c;16  options.bld;28
osfpal_common.mar;31 osfpal_machine.mar;48 pal_descrip.mms;13
pal_ev4.mar;93  pke_driver.c;27  platform_mustang.mms;4
powerup_mustang.c;15 show_hwrpb.c;7  srom$21066.mar;1
tga_driver.c;48  tga_driver_copy.c;17 tga_driver_port.c;49
tga_test_vram.c;12 timer.c;30  toy_driver.c;10
vgag_driver.c;34
```

Files: CMS Generations

The following examples show files modified by cloning code sections. Current files are located in the `CP$REF:` directory.

Note

Example 7–3 shows the complete list of files modified to port to an Alpha AXP PC.

Example 7–4 Build Files

```
build.com
options.bld
pal_descrip.mms
```

Example 7–5 Kernel Source Files

```
call_backs.c  callbacks_alpha.mar  entry.c
ev_action.c  filesys.c   filesys.c
hwrpb.c   kernel.c   kernel_alpha.mar
net.c   nvram_def.h   startstop.c
timer.c
```

Example 7–6 Drivers

```
combo_driver.c
eisa_driver.c
eisa.h
er_driver.c
ide_driver.c
ipr_driver.c
kbd_driver.c
mop_driver.c
n810_driver.c
pke_driver.c
```

Appendix D: Initialization, Kernel, and Diagnostic Routines

Introduction

After PALcode vectors to the console address in memory, the console is initialized.

During console initialization, the file system is initialized and a sequence of code modules and their corresponding routines are run.

Sequence of Kernel Modules

The modules that are run are as follows:

1. `krn$_start` code runs.
2. `krn$_init` code runs.
3. `krn$_idle` code runs and becomes the CPU's idle process, this process:
   a. Builds and loads PCB, SCB, memory, and semaphores
   b. Executes routine reschedule, which starts the following:
      - Timer process, `krn$_timer`
      - Dead eater process `krn$_dead`
      - Powerup process `krn$_powerup`, which starts the following:
        – Runs file system initialization
        – Runs memory configuration
        – Builds HWRPB
        – Files system init1,2,3, and driver init
        – Runs self-tests
        – Starts entry process `krn$_entry`
        – Starts shell process

System Initialization Routines and Data Structures

Key modules in system initialization are:

- `entry.c`
- `kernel.c`
- `kernel_alpha.mar`
- `kernel_def.sdl`
- `kernel_vax.mar`
- `powerup.c`
- `process.c`

When running on an Alpha AXP platform, various PALcode routines are also executed. See the *Alpha System Reference Manual* (SRM) for information on the Privileged Architecture Library (PAL).

entry.c Module

Module `entry.c` contains the routines shown in the table.

Table 7–3 Module entry.c Routines

Routine	Function
`sec_init`	Initializes a secondary processors HWRPB slot. The primary uses this routine to ensure that all secondaries initialize processor state.
`init_cpu_state_all_cpus`	Initializes all secondaries. Initialize various state on each processor before beginning the boot sequence.
`system_reset`	Resets the system.
`node_halt`	Halts the given processor.
`node_halt_primary`	Forces the primary processor into console mode, so that it can request that the primary reboot the system.
`node_halt_secondaries`	Forces a secondary processor into console mode when the primary boots.
`restart_cpu`	Restarts a processor.
`secondary_start`	Starts a secondary processor. Executed as the result of a start command (without parameters) issued by the operating system via the HWRPB.
`sec_boot_prep`	Updates secondaries NVRAM with primary's data. The primary uses this routine to ensure that all secondaries update their nonvolatile RAMs for NVR (`[boot_spec]`, NVR`[expected_entry]`, and so forth), to match the primary. Executed only on cold boot.
`boot_reset`	Performs a system reset to reboot system.
`boot_system`	Performs either a cold or warm bootstrap. The `boot_type` determines the nature of the bootstrap: default, cold or warm boot.

(continued on next page)

Table 7–3 (Cont.) Module entry.c Routines

Routine	Function
`request_reboot`	Requests the primary processor to reboot the system. The secondary uses this routine to request that the primary reboot the system.
`system_reset_or_error`	Resets or error dispatch; only for a system reset or error entry.
`entry`	Starts console entry server routine. Started as a new process at powerup; executes forever. Provides a firmware process context for executing callback routines, boots and restarts.

kernel.c Routines

Module `kernel.c` contains the routines shown in the table.

Table 7–4 kernel.c Routines

Routine	Function
`null_procedure`	Always returns success without side effects
`krn$_unique`	Creates a systemwide unique identifier
`krn$_idle`	Creates idle process for every CPU in the system
`krn$_remove_node`	Removes a processor from the system
`krn$_replace_node`	Activates a processor
`schedule`	Finds the next executable process
`show_version`	Displays the current firmware version
`reschedule`	Schedules the next process
`establish_setjmp`	Saves a `setjmp` environment for the current process
`find_setjmp`	Finds a `setjmp` environment for the current process
`spinlock`	Gains ownership of a spinlock
`spinunlock`	Releases a spinlock

kernel_ alpha.mar Routines

Some key kernel routines from `kernel_alpha.mar` are shown in the table.

Table 7–5 kernel_alpha.mar Routines

Routine	Function
`krn$_init`	Sets data structures for the kernel; idle task when done
`getpcb`	Returns address of current PCB
`do_halt`	Halts, but save arguments
`swap_context`	Saves current execution context; set up new process context
`setjmp`	Saves state information for use by `longjmp`
`longjmp`	Restores previously information saved by most recent call to `setjmp`

kernel_def.sdl Module

Module `kernel_def.sdl` contains the following key data structures.

Table 7–6 kernel_def.sdl Data Structures

Data Structure	Description
QUEUE	Standard console queue (double linked list)
SEMAPHORE	Semaphore Queue (FIFO)
NI_GBL	NI Global Data Definition
INODE	File Entry Structure
RAB	Read Ahead Buffer
IOB	IO Block Structure
FILE	File Descriptor Structure
DDB	Driver Database
TIMERQ	Queue of Sleeping Processes
POLLQ	Queue of Poll Routines
PBQ	Queue of Port Blocks
DIAG_DIS_TABLE	Diagnostic Dispatch Table
DIAG_EVS	Diagnostic Environment Variable States
VAX_HW_PCB	VAX Hardware Context
VAX_EXC	VAX Exception Context
PCB	Process Control Block

(continued on next page)

Table 7–6 (Cont.) kernel_def.sdl Data Structures

Data Structure	Description
LOCK	Spinlock Data Structures
HQE	Handler Queue Entry
VSJD	VAX SetJmp Data
ASJD	Alpha SetJmp Data
SJD_UNION	SetJmp Data Union
SJQ	SetJmp Queue

kernel_vax.mar

Some key kernel routines from `kernel_vax.mar` are shown in the table.

Table 7–7 kernel_vax.mar Routines

Routine	Function
`krn$_init`	Sets data structures for the kernel; idle task when done
`CONSOLE_DISPATCH`	—
`Begin_Code`	—
`dispatch_reentr`	—
`getpcb`	Returns address of current PCB
`console_exit`	Exits the console to the PC saved on console entry
`setjmp`	Saves state information for use by `longjmp`
`longjmp`	Restores previous information saved by most recent call to `setjmp`

process.c Routines

Module `process.c` contains the key routines shown in the table.

Table 7–8 process.c Routines

Routine	Function
`krn$_process`	Performs process startup/rundown functions
`krn$_findpcb`	Translates a PID into a PCB
`krn$_setaffinity`	Sets a process’s affinity mask
`krn$_setpriority`	Sets a process’s software priority
`krn$_create`	Creates a process

(continued on next page)

Table 7–8 (Cont.) process.c Routines

Routine	Function
krn$_delete	Deletes a process
krn$_dead	Deallocates resources from dead processes
krn$_walkpcb	Finds the state of a given process
krn$_kill	Kills a process
killpending	Returns the "kill pending" state for a process
check_kill	Terminates a process if the kill pending bit is set

powerup.c Routines

Module `powerup.c` contains the routines shown in the table.

Table 7–9 powerup.c Routines

Routine	Function
powerup	Powerup process; processor-specific startup; controls the powerup screen
bp_determination	Determines an eligible candidate for boot processor
read_bp	Reads the boot processor (BP) bit of the specified processor
write_bp	Writes the boot processor (BP) bit of the specified processor
read_bpd	Reads the boot processor disabled (BPD) bit of the specified processor
write_bpd	Writes the boot processor disabled (BPD) bit of the specified processor
read_sync	Reads the SYNC bit of the specified processor
write_sync	Writes the SYNC bit of the specified processor

Diagnostic Services Routines

Table 7–10 Diagnostic Service Routines

Routine	Function
`diag_init`	Initializes the diagnostic environment
`diag_print_end_of_pass`	Prints the diagnostic end-of-pass message
`diag_print_status`	Prints a diagnostic status message
`diag_print_test_trace`	Prints the diagnostic test trace message `diag_report_error_lock()` and releases lock
`diag_start`	Starts execution of the diagnostic tests
`report_error`	Displays generic error reporting routine
`report_error_lock`	Displays generic error reporting routine with write lock on the standard error channel left taken out
`report_error_release`	Provides complete error report started by `report_error_lock()`and releases lock

Appendix E: SROM Mini-Console Command Summary

Commands

Table 7–11 summarizes the command set for the Alpha AXP SROM mini-console.

Table 7–11 Command Summary

Command	Function
`dm`	Deposits data to one memory location
`em`	Examines one memory location
`qw`	Sets quadword data mode (64-bit data)
`lw`	Sets longword data mode (32-bit data)
`fm`	Fills a block of memory with a data pattern
`bm`	Displays a range (block) of memory locations
`hm`	Sets high-memory mode (64-bit address)
`lm`	Sets low-memory mode (32-bit address)
`ba`	Gets a base address and sets the base address mode flag
`sa`	Sets the base address mode flag
`ca`	Clears the base address mode flag
`wm`	Performs a looping write to one memory location
`rm`	Performs a looping read from one memory location
`!m`	Performs a looping write/read sequence at one memory location
`dc`	Deposits data to one CPU register
`ec`	Examines contents of CPU registers
`xm`	Loads an external image to memory
`st`	Starts execution at an address

Appendix F: XDELTA Commands

The following figure shows an XDELTA Command Reference.

Figure 7–3 XDELTA Command Reference

```
                       XDELTA Functions and Commands

Function            Command                      Example/Description

                           --- BREAKPOINTS ---

Set breakpoint      addr,N;B                     800055F6,2;B
                    (N is a number 2-8)

Display breakpoint  ;B                           ;B
                                                 1  8000B17D
                                                 2  800055F6

Clear breakpoint    0,N;B                        0,2;B


                        --- DISPLAYING MEMORY ---

Set display mode    [B                           Byte
                    [W                           Word
                    [L                           Longword
                    [|                           Instruction
                    "                            ASCII

Open location and   address/                     GA88/00060034
display contents

Open location and
display instruction address!                     2002!SUBL2   #04,SP

Open location and   address"                     40000"T
display contents in
ASCII mode

Replace contents
of given address    addr/contents new            GA88/00060034 GA88
                                                 GA88/00060034 'A'
                                                 Replace as ASCII

Display contents    ESC (Escape)                 80000A88/80000BE4 ESC
of previous                                      80000A84/00000000
location

Display contents    addr/contents LF             80000004/8FBC0FFC LF
of next location    (Line Feed)                  80000008/50E9002C

Display indirect    TAB                          80000A88/80000BE4 TAB
                                                 80000BE4/80000078
                    or
                    /                            80000A88/80000BE4/80000078

Display range of    addr,addr/contents           G4,GC/8FBC0FFC
locations                                        80000008/50E9002C
                                                 8000000C/00000400


                        --- DISPLAYING REGISTERS ---

Set base register   'value',N;X                  80000000,0;X

Display base        Xn RETURN                    X0
register            or                           00000003
                    Xn=                          X0=00000003

Display general     Rn/                          R0/00000003
register            (n is hexadecimal)


                      --- START and STOP ---

GO                  ;G                           G B17D;G

Proceed from        ;P                           ;P
breakpoint

Single step         S                            1 brk at 8000B17D
into subroutines                                 S
                                                 8000B17E/9A0FBB05

Single step         O                            1 brk at 8000B17D
over                                             S
subroutines                                      8000B17E/9A0FBB05

Exit from XDELTA

                    EXIT                         1 brk at 8000B17D
                                                 EXIT
                                                 0000201F/BICL3   AP,R3,-(SP)
                                                 exit
                                                 $


                       --- EXPRESSIONS and MISCELLANEOUS ---

Show value          expression=                  1+2+3+4=0000000A
                    (+, -, *, %{divide})

Shift a value       value@shift                  n@2
+left
-right

Executing stored    addr;E ret                   80000E58;E
command strings

List names and      ;L                           ;L
locations of
loaded executive
images
```

Appendix G: Basic Console Commands

Syntax

The table shows a console command summary.

Table 7–12 Syntax of Common Console Commands

Command	Options	Parameters
`boot`	[-file filename] [-flags root, bitmap] [-halt]	[boot_device]
`continue`	—	—
`deposit`	[-{b,w,l,q,o,h}] [-n val] [-s val]	[device:]address data
`examine`	[-{b,w,l,q,o,h,d}] [-n val] [-s val]	[device:]address
`help`	—	[command]
`man`		[command]
`initialize`	[-c] [-d device_path]	[slot-id]
`set`	—	envar value
`set host`	[-dup] [-task t]	node
`show`	—	envar, config, device, error, fru, hwrpb, memory
`start`	—	address
`test`	—	cpu, memory, ethernet, scsi

Appendix H: Diagnostic Environment Variables

Environment Variables

The following global environment variables are available to the user for control of diagnostics. These environment variables are global to all diagnostic programs executing. These environment variables can be overwritten by the user on the command line using the SET command.

Table 7–13 Environment Variables

Variable	Values	Definition
D_BELL	OFF (Default) ON	Bell on error if error is detected
D_CLEANUP	ON (Default) OFF	Determines whether or not cleanup code is executed at end of diagnostic execution
D_COMPLETE	OFF (Default) ON	Determines whether or not to display the diagnostic completion message
D_EOP	OFF (Default) ON	Determines whether or not to display end-of-pass messages
*D_GROUP	FIELD (Default) MFG Arbitrary String	Determines diagnostic group to be executed—maximum length of 32 characters
D_HARDERR	HALT (Default) CONTINUE LOOP	Determines action taken following hard error detection
D_LOGHARD	ON (Default) OFF	Determines whether or not hard errors are logged to the module EEPROM
D_LOGSOFT	OFF (Default) ON	Determines whether or not soft errors are logged to the module EEPROM
D_OPER	ON (Default) OFF	Set to whether or not an operator is present
*D_PASSES	1 (Default) 0 - Forever Arbitrary Value	Determines number of passes to run a diagnostic module
D_QUICK	OFF (Default) ON	Determines mode of testing - normal or quick verify

(continued on next page)

Table 7–13 (Cont.) Environment Variables

Variable	Values	Definition
D_REPORT	SUMMARY (Default) FULL OFF	Determines level of information provided by diagnostic error reports
D_SOFTERR	CONTINUE (Default) HALT LOOP	Determines action taken following soft error detection
D_STARTUP	OFF (Default) ON	Determines whether or not to display the diagnostic startup message
D_STATUS	OFF (Default) ON	Determines whether or not to display status messages
D_TRACE	OFF (Default) ON	Determines whether or not to display test trace messages

Glossary

Overview

This glossary includes terms that are used in the development of the Alpha AXP Firmware, its documentation and build environment.

Alpha AXP Firmware

A common pool of console source code shared across multiple Alpha AXP hardware platforms that can be custom built.

ARC

Advanced RISC Computing. A specification that defines the architecture for industry-standard computing platform, based on the MIPS family of microprocessors.

Build

The creation of the Alpha AXP Firmware by using the automated procedures `CP_COMMON_LOGIN.COM` and `NEWBUILD.COM`.

Common Console

The name used to describe the Alpha AXP Firmware product; used synonymously with console firmware.

Configuration

A name for a variant of a common console build for a particular hardware platform that becomes part of the file and directory structure naming convention. This can be defined in `CP_COMMON_LOGIN.COM` with the keyword “configuration” and defaults to the platform name if not defined. For example, `SBLOAD` builds an image for the Sable hardware platform that includes the basic console firmware, XDELTA debugger, diagnostics, and can be loaded with MOP.

DDB

Driver database. The DDB is a data structure that contains the name of the class driver and addresses of its routines.

EISA

Extended Industry Standard Architecture. A parallel, I/O, 32-bit data bus, that is an extension to its predecessor, the 16-bit ISA bus. It provides 33.32 MB/second data throughput at 8.33 MHz.

Encapsulated Files

Source files listed with the keyword "encapsulate" in the `HardwarePlatform_FILES.BLD` file, for example, `SABLE_FILES.BLD`, are created in the file system table when the firmware is built. These images are assigned a file name with attributes that can be invoked by a console command.

FEROM

Flash EROM memory can be randomly accessed with fairly fast access times (comparable to dynamic RAMs), yet provides nonvolatile storage. FEROM can be erased electrically and selectively in multiple sections (zones), and then rewritten in seconds.

ISA

Industry Standard Architecture. A 16-bit parallel I/O data bus that is an extension to its predecessor, the 8-bit data bus. It provides approximately 8 MB/second data throughput.

MOP

Maintenance-oriented protocol that allows downline loading of files from a host node over the Ethernet.

NVRAM

Nonvolatile, random-access memory. Contains environmental variables used throughout the console, for example the default boot device variable is `DEF_BOOT_DEV`.

PALcode

Privileged Architectural Library. A library of machine instructions running in a privileged mode that isolates the operating system from the Alpha AXP architecture and hardware implementation.

PB

The port block is a data structure that maintains state and carries specific information about a port driver on a per-instance basis, for example, interrupt mode and physical locations of that device. The PB is passed to the class driver to perform port functions.

PCI

Peripheral component interconnect. This 32-bit (and 64-bit PCI - 2) parallel multiplexed address and data bus was designed by a consortium of vendors as a high-speed peripheral and general interconnect. The 32-bit version operates at 132 Mbyte at 33 MHz.

Platform

A name for a hardware platform that forms the base configuration of an Alpha AXP Firmware build; files and directory structure. Other build configurations for the same platform share its common code base, but differ in some variation. This is defined in `CP_COMMON_LOGIN.COM` with the keyword “platform”.

SROM

Serial read-only memory. A special ROM chip that is read serially into the Instruction cache (Icache) DECchip 21064 upon powerup or during reset. The SROM code initializes the DECchip, memory, and optionally other hardware, such as external backup cache, then loads the console firmware into memory.

TOY

Time-of-year interrupt clock. See `toy_driver.c` and `timer.c` for supporting code.

Index

A

Address
i/o, 4–20
i/o, defining, 4–32
ARC
adding support, build file, 6–10
Architecture
statement, build file, 4–42
AUTO_ACTION
default value on system ship, 3–38
variable, 3–38

B

Base
platform, directory, 3–10
Bcache
loading the image into, 5–6
Boot
auto_action, 3–38
Boot Flags
DEC OSF/1, 3–44
Boot Processor, 3–37
BOOTDEF_DEV, 3–39
Booting
boot flag options, 3–40
command syntax, 3–39
command syntax, Ethernet, 3–43
DEC OSF/1, 3–44
DEC OSF/1 boot flags, 3–44
environment variables, 3–38
ethernet, 3–42
OpenVMS, 3–40
process, 3–37
with bootp, 3–42
with mop, 3–42
BOOTP
booting, 3–42
BOOT_DEV, 3–39
BOOT_OSFLAGS
OpenVMS boot options, 3–40
Build
adding drivers, 6–8
adding files, 4–43
cpu resources, 3–13
directories, generic, 3–9
Build (cont'd)
environment, 1–12
environment, logicals, 7–2
file, configuration, 3–11
files, cloning, 3–18
files, modifications, 3–19
major components, 3–6
options.bld, 4–45
platform configuration file, 4–42
procedure, 3–22
reference documents, 3–3
with xdelta, 4–50
without xdelta, 4–51
build.com
modifying, 4–47
Building
a cloned image, 3–4
Bus
sniffing, console terminal, 6–6
Bus windows
i/o, defining, 4–34

C

Class
CMS definition, 7–8
Cloning
build files, 3–18
code sections, 3–16
files, 3–14
CMS
class definition, 7–8
classes, 7–8
code management system, 7–7
creating, 7–7
DECwindows qualifier, 7–7
elements, 7–9
groups, 7–9
hierarchy, 7–7
introduction, 7–7
using, 7–7
Code Management System
CMS, 7–7
COM1
output, 6–6
Commands
console, 3–28
console, basic, 7–22

Commands (cont'd)
diagnostic environment variables, 7–23
diagnostic services routines, 7–18
Entry, 7–14
interface, configuring, 2–8
Powerup, 7–17
scripts, 3–31
srom mini-console, 5–11
to load drivers, 4–36
to unload drivers, 4–36
config.bld
options, 4–50
Configuration
build file, 3–11
directories, 3–9, 3–10
file, modifying, 4–42
subdirectories, 3–11
Configuring
memory, 4–8
Console
commands, basic, 7–22
device, PALcode, 4–14
functions, 2–10
initialization, 2–11
memory image, 2–4
terminal i/o, 4–20
terminal selection, 6–7
Conventions
document, xvi
CP$REF
directory, 7–2
CP$SRC
logical search string, 7–2, 7–3

D

Debug
code tracing, initialization, 4–29
hardware, 5–4
tracing drivers loaded, 4–37
DEC OSF/1
boot environment variables, 3–44
boot flags, 3–44
DECchip
21066, 3–4
Description
file, PALcode, 4–17
Development
creating the environment, 3–11
environment, 1–12, 3–6
Device
console terminal selection, 6–7
controller bus windows, defining, 4–34
Diagnostic
messages, 3–35
sample user session, 3–32
Diagnostic Services
routines, 7–18
Diagnostics
scripts, 3–31
Directories
configuration, 3–10
structure, build, 3–8
Disabling
hardware drivers, 4–36
Driver
adding to build, 6–8
model, 2–7
sequence, console device, 6–6
startup phase, build file, 4–35
Drivers
disabling, 4–36
loading, 4–36
unloading, 4–36

E

Encapsulation
scripts, build file, 6–9
scripts, inode, 6–9
Engineering
VMScluster, 3–13
VMScluster, load, 3–13
Entry, 7–14
Environment
creating, 3–11
development, 3–6
Environment variables
booting DEC OSF/1, 3–44
diagnostic, 7–23
D_BELL, 7–23
D_CLEANUP, 7–23
D_COMPLETE, 7–23
D_EOP, 7–23
D_GROUP, 7–23
D_HARDERR, 7–23
D_LOGHARD, 7–23
D_LOGSOFT, 7–23
D_OPER, 7–23
D_PASSES, 7–23
D_QUICK, 7–23
D_REPORT, 7–24
D_SOFTERR, 7–24
D_STARTUP, 7–24
D_STATUS, 7–24
D_TRACE, 7–24
ethernet, 3–43
Environmental
variables, 4–24
Error
messages, 3–35
esc_nvram_driver.c
environmental variables, 4–26

Ethernet
booting, 3–42
environment variables, 3–43
protocols, 3–42

F

Files
build.com, modifying, 4–47
cloning, 3–14
environmental variables, 4–24
kernel_alpha.mar, 4–28
modified for port, list, 7–10
needing modification, 3–17
nvram_def.h, 4–25
options.bld, 4–45
platform_files.bld, modifying, 3–19
platform_io.h, i/o addresses, 4–32
ported, cms generation, 7–11
Flags
bootstrap in progress, 3–41
cold start, 3–41
Functions
of PALcode, 2–10
of the console, 2–10

G

Group
driver, build file, 4–44
keyword, build file, 4–43

H

Halt
auto_action, 3–38
Hardware
debug, 5–4
interface, 2–6

I

I/O
address, 4–20
address definition, 4–21
defining bus windows, 4–34
defining the address, 4–32
terminal, 4–20
Image
memory, 2–4
Initialization
code tracing, 4–29
console, 2–11
Interface
hardware, 2–6
operating system, 2–8
user, 2–7
Interrupt
mode, polled, 4–39
Interrupts
PALcode, 4–14
polled mode, drivers, 4–38

J

jputc
for debugging, 4–29

K

Kernel
adding files, 4–43
routines, modules, 7–12
Kits
developer, save sets, 3–7

L

Load
command, 3–13
Logical
build environment, 7–2
Logicals
build environment, 7–2

M

Memory
console image, 2–4
defining the configuration, 4–8
Messages
diagnostic, 3–35
error, 3–35
MMS
PALcode description file, 4–17
PALcode files, 4–16
platform description file, PALcode, 4–15
Mode
hardware interrupt mode, 4–38
polled interrupt mode, 4–38
MOP
booting, 3–42

N

Network
listening, 3–42
NVRAM
environmental variables, 4–24
nvram_def.h
environmental variables, 4–25

O

OpenVMS
boot options, 3–40
Operating System
interface, 2–8
Option
statements, 4–43, 4–45
statements, build, 3–19
statements, build file, 4–42
statements, default, 4–42
Options
build file, 4–45
XDELTA, 4–50
options.bld
build file, 4–45
Output
changing from vga, 6–6
changing to com1, 6–6

P

PALcode
console device, 4–14
description of files, 4–15
functions, 2–10
image, 4–15
interrupts, 4–14
mms description file, 4–17
mms files, 4–15, 4–16
modification steps, 4–12
porting changes, 4–14
strategy of porting changes, 4–12
Platform
base directory, 3–10
statement, build file, 4–42
platform_files.bld
modifying, 3–19
PLATFORM_IO.H
i/o address definitions, 4–32
platform_util.mar
address definition, 4–21
Polled
interrupt mode, drivers, 4–38
interrupts, defining, 4–38
Porting
changing between COM1 from VGA, 6–6
cloning code, 3–16
cloning files, 3–14
environment, 1–12
environmental variables, 4–25
factors, 1–13
list of files modified, 7–10
modifying build files, 3–19
procedure, 1–10
process, overall, 1–4
requirements, 1–12
Porting (cont'd)
scenarios, 1–5
strategy, 1–6
Power-Up
process, 2–11
Powerup, 7–17
process, 4–36
printf
for debugging, 4–30
Process
powerup, 4–36
Protocols
mop, 3–42
tcp/ip, 3–42

R

Requirements
porting, 1–12
Restart
auto_action, 3–38
Robust
mode, 4–36
Routines
services, diagnostic, 7–18

S

sable_files.bld
modifying, 4–42
statements, 4–42
Save
sets, developer kits, 3–7
Scripts
adding, build file, 6–9
command, 3–31
encapsulation, inode, 6–9
generic, diagnostic, 3–31
invoking, 6–9
Search
strings, logicals, 7–2
SROM
memory configuration registers, 4–8
mini-console commands, 7–19
serial rom, introduction, 5–6
SROM Mini-Console
baud rate, 5–10
commands, 5–11
hardware required, 5–8
hardware setup, 5–9
introduction, 5–8
starting, 5–10
synchronization, 5–10
u command, 5–10
Statements
option, 4–43
value, 4–43

Subdirectories
configuration, 3–11

T

Terminal
console i/o, 4–20
driver sequence, 6–6
selection, 6–7
TOY
debug, 4–27
files, 4–23
timer of year clock, 4–23

U

User
interface, 2–7

V

Value
statements, 4–43, 4–45
statements, build file, 4–42
statements, default, 4–42
Values
build file, 4–45
Variables
environment, diagnostic, 7–23
environmental, 4–24
VGA
driver sequence, 6–6
output, 6–6

X

XDELTA
building option, 4–50, 4–51
building with extensions, 4–59
commands, 4–52
extended command summary, 4–60
extended commands, 4–57
extended, debug example, 4–58
extensions and features, 4–57
invoking, 4–51
OpenVMS boot options, 3–40
overview, 4–50