KXT11–C Peripheral Processor Software User's Guide

AA–Y615A–TK

March 1984

This manual will help you use the KXT11–C Peripheral Processor software.

Operating System: RT–11 Version 5.0
RSX–11M Version 4.1
RSX–11M–PLUS Version 2.1
VAX/VMS Version 3.4

Software: MicroPower/Pascal Version 1.5

To order additional documents from within DIGITAL, contact the Software Distribution Center, Northboro, Massachusetts 01532.

To order additional documents from outside DIGITAL, refer to the instructions at the back of this document.

digital equipment corporation · maynard, massachusetts

First Printing, March 1984

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by DIGITAL or its affiliated companies.

Printed in U.S.A.

A postage-paid READER'S COMMENTS form is included on the last page of this document. Your comments will assist us in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

digital™

DEC	IAS	Rainbow
DECmate	MASSBUS	RSTS
DECnet	MicroPower/Pascal	RSX
DECsystem–10	PDP	UNIBUS
DECSYSTEM–20	PDT	VAX
DECUS	P/OS	VMS
DECwriter	Professional	VT
DIBOL	Q–BUS	Work Processor

M31000

CONTENTS

FIGURES

TABLES

PREFACE

This manual helps you use the software tools provided by DIGITAL to program and use the KXT11-C Peripheral Processor. Detailed information about KXT11-C hardware and supporting software is provided by reference manuals which are cited throughout this manual.

Reader Assumptions

This manual helps the programmer using MicroPower/Pascal or one of the KXT11-C peripheral processor tool kits. It is possible to program and use KXT11-C's with traditional LSI-11 software development tools; however, guidelines for doing this are not discussed at any length in this manual.

This manual assumes you are familiar with DIGITAL's LSI-11 series of computer products and have programming experience in one of the following environments:

- MicroPower/Pascal-RT-11, -RSX or -VMS host application development environment, writing programs in the MicroPower/Pascal or the MACRO-11 language.
- RT-11, RSX-11M or RSX-11M-PLUS host system environment, writing and debugging MACRO-11 assembly programs for stand-alone PDP-11 applications.

Document Structure

This manual contains four chapters and six appendixes.

- Chapter 1 summarizes the features of the KXT11-C hardware and its supporting software and provides examples of typical applications.
- Chapter 2 describes the procedures for programming KXT11-C application systems.
- Chapter 3 provides information about programming a KXT11-C with MicroPower/Pascal software and using KXT11-C peripheral processors in MicroPower/Pascal arbiter applications.
- Chapter 4 describes how to use KXT11-C peripheral processors in applications based on RT-11 or RSX-11 system environments.
- Appendix A describes the communication protocol that the KX and KK device handlers use to pass messages between the arbiter processor and KXT11-C peripheral processors. It assists those who want to write their own KK or KX handler.

- Appendix B lists the CSR and interrupt vector assignments for KXT11-C devices.
- Appendix C lists the CSR and interrupt vector assignments, KX device handler logical unit numbers, and system ID switch settings for the KXT11-C two-port RAM.
- Appendix D shows a sample KXT11-C MicroPower/Pascal configuration file.
- Appendix E describes the procedures for calculating checksums for PROMs that are loaded by the DECprom program.
- Appendix F describes how your application can check for nonfatal error conditions reported by the native firmware's self-test routines.

Associated Documents

The following software and hardware documentation supports KXT11-C application development for host and target environments. The documents you will need depend on which host development system you are using and whether you are developing KXT11-C applications in the MicroPower/Pascal environment or in a traditional MACRO-11 environment.

MicroPower/Pascal Application Development Environment

The MicroPower/Pascal-RT documentation set, QJØ29-GZ, provides a complete description of MicroPower/Pascal software tools in the RT-11 host system environment.

The MicroPower/Pascal-VMS documentation set, QDØ29-GZ, provides a complete description of MicroPower/Pascal software tools in the VAX/VMS host system environment.

The MicroPower/Pascal-RSX documentation set, QPØ29-GZ, provides a complete description of MicroPower/Pascal software tools in the RSX-11M and RSX-11M-PLUS host system environments.

Tool Kit Application Development Environment (for arbiter application development)

KXT11-C Software Toolkit/RT Reference Manual, AA-AU63A-TC

KXT11-C Software Toolkit/RSX Reference Manual, AA-AU64A-TC

PROM Loading Utility Programs

PB-11 System User's Guide, AA-848B-TC

VAX/VMS DECprom User's Guide, AA-W754A-TK

Host Operating System Documentation

The MicroPower/Pascal documentation sets include documents from the sets listed below that are necessary to facilitate MicroPower/Pascal application development.

The RT-11 documentation set, QJØ13-GZ, provides a complete description of the software tools for the RT-11 host environment.

The RSX-11M documentation set, QJ628-GZ, provides a complete description of the software tools for the RSX-11M host environment.

The RSX-11M-PLUS documentation set, QR5ØØ-GZ, provides a complete description of the software tools for the RSX-11M-PLUS host environment.

The VAX/VMS documentation set provides a complete description of the software tools for the VAX/VMS host environment.

You will also need the following hardware reference documentation to configure your application hardware to use the standard device handlers, or to write device handlers that are hardware- and software-compatible with other system components:

Microcomputer Manual

KXT11-CA User's Guide hardware reference manual, EK-KXTCA-UG

Microcomputer Handbooks

MICRO/PDP-11 Handbook, EB-24944-18

Microcomputers and Memories, EB-2Ø912-2Ø

Microcomputer Interfaces Handbook, EB-23144-18

PDP-11 Architecture Handbook, EB-23657-18

Document Conventions

1. Throughout the text, the term RSX-11 is used to simplify references that apply to both the RSX-11M and RSX-11M-PLUS operating systems.

2. Pascal-reserved words that must not be abbreviated are shown in uppercase characters in syntax examples. Within those examples, lowercase characters are used for parameters or other syntax elements that you must include for your particular application.

3. All hardware device register and memory addresses specified in this manual are octal values (for example, 77xxxx). Addresses for the arbiter's Q-BUS memory are shown as 22-bit values; while those for the KXT11-C are shown as 16-bit values.

4. All command strings terminate with a carriage return. The symbol used in this manual to represent a carriage return is <RET>.

5. To produce certain characters in system commands, you must type a combination of keys concurrently. For example, while holding down the CTRL key, type C to produce the CTRL/C character. Key combinations such as this are documented as <CTRL/C>, <CTRL/O>, and so forth.

6. In examples, you must distinguish between the capital letter O and the number Ø. Examples in this manual represent these characters as follows:

   Letter O: O

   Number Ø: Ø

7. The sample terminal output in this manual contains version numbers where they would normally appear. The version numbers include xx in those fields that can vary from installation to installation.

CHAPTER 1

INTRODUCTION

The KXT11-C Peripheral Processor is an LSI-11 single-board 16-bit computer with local memory and communication ports. You can use it as a self-contained stand-alone system or as a component (peripheral processor) of an LSI-11-based multiple processor system.

In a multiple processor system, you can add up to 14 user-programmed KXT11-C Peripheral Processors to traditional LSI-11 Q-BUS systems and communicate with them from the LSI-11 CPU acting as arbiter. The software architecture is master/slave (not to be confused with the bus-master/bus-slave hardware concept) which means the KXT11-C application (slave) performs operations only on command from the arbiter application (master). The master application runs in the LSI-11 (Q-BUS) arbiter processor and controls the KXT11-C application by sending it messages over the KXT11-C two-port RAM (TPR) registers in the I/O page. The KXT11-C can also transfer data to and from main memory with its DMA transfer controller facility (DTC).

When configured for stand-alone operation, the KXT11-C is completely self-contained with no Q-BUS required.

The DIGITAL-supplied software supports KXT11-C single-board computers as stand-alone systems or as peripheral processors in a multiple processor environment. You can program the KXT11-C using the MicroPower/Pascal or MACRO-11 language. In peripheral processing applications, you can then incorporate the processors into arbiter applications based on the RT-11, RSX-11M, RSX-11M-PLUS or MicroPower/Pascal operating environment.

In addition to DIGITAL's standard LSI-11 software development tools, you can choose from the following three software products that provide tools for KXT11-C application development:

- MicroPower/Pascal
- KXT11-C/RT-11 Peripheral Processor Tool Kit
- KXT11-C/RSX-11 Peripheral Processor Tool Kit

MicroPower/Pascal software provides tools for developing KXT11-C stand-alone or peripheral processor applications in MicroPower/Pascal and MACRO-11 under the control of the MicroPower/Pascal run-time kernel. Included are handlers for the following KXT11-C on-board devices:

Asynchronous serial I/O

Synchronous serial I/O

Parallel I/O and counter-timers

Real-time clock

DMA transfer

For peripheral processor applications, device handlers provide communication through the two-port RAM (TPR). They allow a MicroPower/Pascal application on the KXT11-C to communicate with a MicroPower/Pascal, RT-11, or RSX-11 application in the arbiter processor. MicroPower/Pascal provides a device handler for MicroPower/Pascal arbiter applications. The handlers for RT-11 and RSX-11 are available in the tool kits.

The KXT11-C peripheral processor tool kits provide tools for using peripheral processors in traditional Q-BUS systems. Support is provided for RT-11, RSX-11M, and RSX-11M-PLUS arbiter applications to load and communicate with KXT11-C peripheral processors across the Q-BUS. The handlers communicate with KXT11-C processors programmed in MicroPower/Pascal.

If those tools do not meet your needs, you can program the KXT11-C in the MACRO-11 assembly language using the standard PDP-11 application development tools (MACRO-11, LINK, ODT, MACDBG, and so on).

If your KXT11-C application program uses ROM, you can load it with the DECprom program (VMS systems) or the PROM/RT-11 program (RT-11 systems).

1.1 KXT11-C HARDWARE FEATURES

Figure 1-1 shows the general layout of the following major hardware components. DIGITAL-supplied software supports most but not all hardware features; the software documentation describes the features supported.

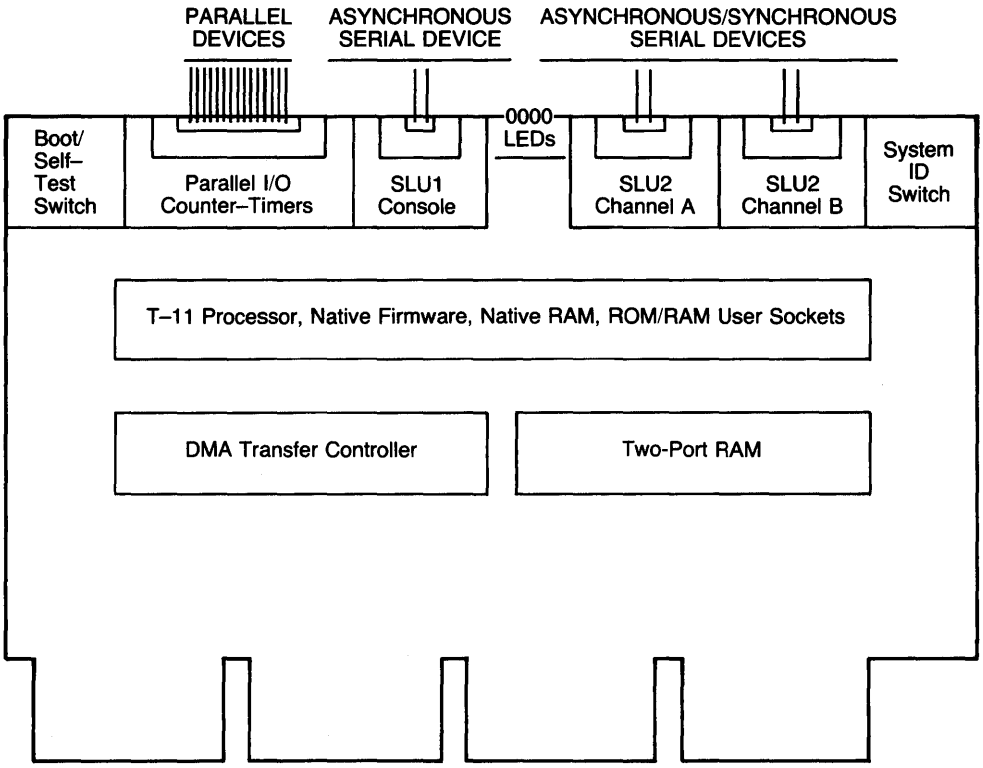


Figure 1-1: **KXT11-C Hardware Features**

- DIGITAL DCT-11 microprocessor -- A 16-bit, 7-MHz microprocessor that executes the PDP-11 basic instruction set.
- On-board memory -- Local memory consisting of 32K bytes of static read/write memory (RAM), sockets for up to 32K bytes of PROM or static RAM, and 8K bytes of native firmware. Additional features include eight memory map configurations, and battery backup for native RAM.
- Native firmware -- Provides:

  Power-up self test

  Optional loopback tests

  Hardware initialization

  Serial ODT accessible from a console terminal or via the Q-BUS

  Application start-up: ROM, TU58 boot, or load from the Q-BUS arbiter

- Two-port RAM (TPR) -- A 16-word interface to the Q-BUS that passes control information and data between the KXT11-C and the arbiter. The RAM is divided into three areas: one area for KXT11-C native firmware commands and two areas for application message passing.

- Two-channel DMA transfer controller (DTC) -- A device that provides for memory and I/O data transfers between the local KXT11-C and global memory on the Q-BUS using direct memory access. Permitted transfer combinations are:

  Local memory to global memory

  Global memory to local memory

  Local memory to local memory

  Global memory to global memory

- Parallel I/O interface -- Contains two bidirectional 8-bit input/output ports, one 4-bit control port, and three 16-bit programmable counter-timers.

- Console port (DLART) -- Provides DL11-compatible asynchronous serial communication.

- Multiprotocol controller -- Provides two-channel synchronous/asynchronous serial I/O communication

- System ID switch -- Sets the system identification address and establishes the KXT11-C for Q-BUS or stand-alone operation.

- Boot/Self-test switch -- Selects bootstrap and firmware self-test operations.

- Configuration jumpers -- Electrical jumpers on the KXT11-C circuit board that select some of the hardware configuration options (other options are software configurable).

Refer to the KXT11-CA User's Guide hardware reference manual for detailed information about the KXT11-C hardware.

1.2 USING THE KXT11-C AS A PERIPHERAL PROCESSOR

Peripheral processor applications help you off-load tasks from a conventional LSI-11 processor application. This improves overall system performance by distributing the application task load. You can add up to 14 KXT11-C peripheral processors to a traditional LSI-11 (Q-BUS) system configuration in the same way you add other I/O device controllers (Figure 1-2). The difference between the KXT11-C and conventional I/O devices is that the KXT11-C is user programmable while conventional I/O devices are not.

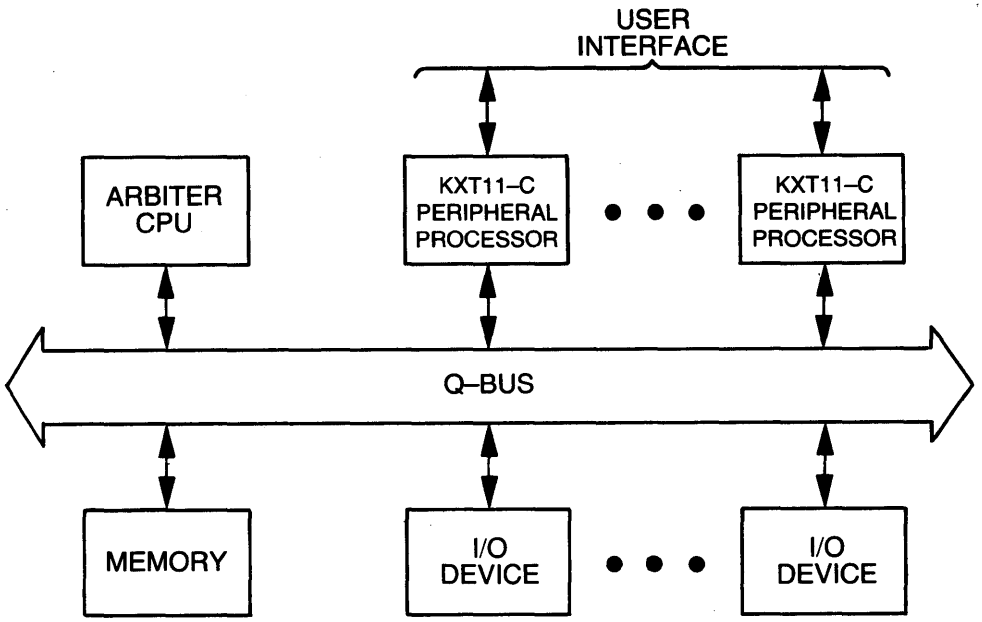


Figure 1-2: **Adding KXT11-C Peripheral Processors to Traditional LSI-11 Systems**

The following are examples of tasks that KXT11-C peripheral processors can perform.

- Guaranteed response time -- A dedicated KXT11-C can assure a specific interrupt response time.
- Data collection and reduction -- The arbiter is relieved of the overhead of the interrupts required to control devices and the CPU Q-BUS time required to refine and format the data.
- Machine control -- The arbiter processor gives high-level commands to the KXT11-C. The KXT11-C translates the commands to the level required by the device(s) and monitors progress. The arbiter application merely issues commands and manages higher-level application tasks.
- Communication protocol handling -- The KXT11-C relieves the arbiter processor from handling communication line interrupts, packing/unpacking messages, and formatting them. Only data is transferred to and from the arbiter.

1.2.1 Peripheral Processor Hardware Configuration

The application system consists of one Q-BUS arbiter CPU (LSI-11, LSI-11/2, SBC-11/21, LSI-11/23, LSI-11/23-PLUS or LSI-11/73) and up to 14 KXT11-C peripheral processors attached to the Q-BUS. The KXT11-C cannot be a bus arbiter.

Communication between the arbiter and the peripheral processor takes place over the Q-BUS through the TPR. The TPR's control, status and data registers, and vectors appear in the arbiter's I/O page. Programs can access the registers in ways similar to those of I/O devices. The peripheral processor's hardware configuration options determine where its TPR area appears in the arbiter's I/O page.

1.2.2 Peripheral Processor Application Software Configuration

A master application in the arbiter processor directs slave peripheral processor operations (Figure 1-3). Communication takes place over the Q-BUS using messages to control the KXT11-C hardware and its application software and to transfer data. Communication can take place using the TPR or the DMA transfer controller (DTC). If desired, the peripheral processor can interrupt the arbiter application when it completes the requested task.

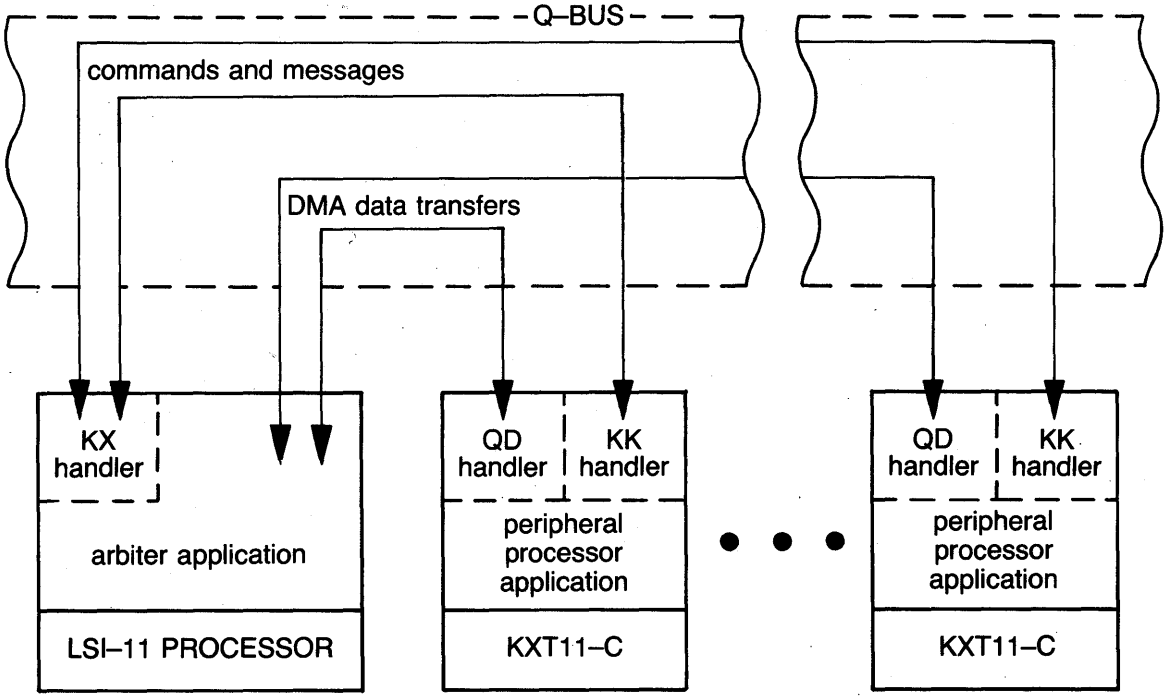


Figure 1-3: **Peripheral Processor Application Software Configuration**

1.2.3 KXT11-C Application Development Software Tools

DIGITAL provides the following software tools to support development of KXT11-C applications for peripheral processing and stand-alone environments.

Tools for Developing KXT11-C Peripheral Processing or Stand-alone Applications

- MicroPower/Pascal
- Traditional development tools: MACRO-11, TKB/LINK, MACDBG, ODT
- VAX DECprom or RT-11 PROM/RT-11 for loading PROMs

Tools for Developing Arbiter Applications Using the KXT11-C

- MicroPower/Pascal
- KXT11-C peripheral processor tool kits for RT-11 and RSX-11 and the traditional development tools: MACRO-11, TKB/LINK, MACDBG, ODT

1.2.3.1 **MicroPower/Pascal** - MicroPower/Pascal supports development and use of KXT11-C applications in the environments listed above. You can write application programs in MicroPower/Pascal or MACRO-11 and place the target application in RAM or ROM. MicroPower/Pascal software provides a complete set of tools for building, debugging, and loading peripheral processor applications. It includes a modular kernel that provides 43 primitive services, system initialization, event/priority scheduling, and interrupt and exception dispatching. Device handlers provide support for all on-board devices and for communication between the arbiter and one or more KXT11-C peripheral processors. The KXT_LOAD procedure loads KXT11-C peripheral processors from memory image files residing in the arbiter's mass storage.

MicroPower/Pascal provides the following tools for the KXT11-C.

- DD handler (TU58 DECtape II) -- The DD device handler supports logical and physical I/O operations on a TU58 cartridge tape subsystem. The TU58 subsystem can be interfaced through any or all of the three serial I/O ports on the KXT11-C. Each subsystem can have two drives.
- XL handler (asynchronous serial line) -- The XL handler supports concurrent operation of all three serial I/O ports in asynchronous mode with full modem control for one of the ports.
- XS handler (synchronous serial line) -- The XS device handler supports channel A of the synchronous/asynchronous controller (MPCC) serial I/O ports in synchronous mode using bit-oriented block mode transfers.

- YK handler (parallel I/O) -- The YK device handler supports the 20 bits of the parallel I/O port and three counter-timers. It supports most combinations of 4-bit, 8-bit, or 16-bit port configurations and the interlocked, strobed, pulsed, and three-wire handshake modes. Ports can be input, output, bidirectional, or mixed mode. The YK handler also supports DMA data transfers (using one of the DTC channels) to and from the port, the three independent 16-bit counter-timers, and the pattern matching feature.

- QD handler (DMA transfer controller) -- The QD device handler supports DMA data transfer operations using the DMA transfer controller (DTC). The handler provides two high-speed channels for a MicroPower/Pascal application to transfer data between global Q-BUS memory and local memory on the KXT11-C. It can also perform transfers completely within the KXT11-C memory or the Q-BUS memory.

- KK and KX handlers (arbiter-to-KXT11-C Q-BUS interface) -- The KK and KX handlers aid in programming the Q-BUS communication interface between the KXT11-C and the arbiter application. The handlers use the two-port RAM (TPR) to pass variable-length messages between applications.

  When using these handlers, the KXT11-C functions as a typical Q-BUS device in a master-slave relationship with the Q-BUS arbiter. The arbiter (master) controls all interactions between the two processors. The KXT11-C (slave) waits for a command from the arbiter processor before initiating a message transfer. The handlers use a request-reply protocol to assure that message transfers are complete and correct.

  Appendix A describes the KK/KX communication protocol to assist those who want to write their own KK or KX handler.

- Device handler support routines -- A group of MicroPower/Pascal functions that provide a Pascal interface to the KXT11-C device handlers.

- Kernel features specified using the KXT11C macro:

  BHALT parameter to enable/disable debugging traps in response to the arbiter's BHALT signal.

  POWER parameter to determine the action to take on power failure and restoration.

  CLOCK parameter to enable/disable the KXT11-C's real-time clock at system start-up.

  RESET parameter to determine the action to take in response to the arbiter's BRESET signal.

  MAP parameter to identify the memory map used on the KXT11-C.

- KXT_LOAD peripheral processor load procedure -- KXT_LOAD is a MicroPower/Pascal utility procedure for loading one or more KXT11-C peripheral processors. The procedure reads a memory image file (.MIM file type) from a specified mass storage device and loads the memory of a designated KXT11-C over the Q-BUS. KXT_LOAD is described in Appendix H of the MicroPower/Pascal Runtime Services Manual.

1.2.3.2 **KXT11-C Peripheral Processor Tool Kits** - The KXT11-C peripheral processor tool kits are for RT-11, RSX-11M and RSX-11M-PLUS arbiters. They provide the software for the arbiter application to load and communicate with user-programmed KXT11-Cs. The tool kits contain the following software:

- KX device handler (Q-BUS interface) -- The KX handler is the device handler/driver for LSI-11 (Q-BUS) systems. It allows the arbiter application to communicate with the KK handler running under MicroPower/Pascal in the KXT11-C. See Appendix A for a description of the KK/KX communication protocol.

- KUI load utility -- The KUI program loads memory image files into a KXT11-C over the Q-BUS. It accepts the RSX-11 .MIM and .TSK file types and the RT-11 .MIM, .SAV and .LDA file types.

1.2.3.3 **Traditional PDP-11 Application Development Software Tools** - You can also program the KXT11-C in MACRO-11 using the traditional PDP-11 assembly language tools provided with RT-11 and RSX-11 systems. These tools include: MACRO-11, TKB (RSX-11 task builder) or LINK (RT-11 linker), and ODT. In addition, RT-11 supports the MACDBG remote symbolic debugger which operates on an RT-11 host connected by serial line to the target KXT11-C. Refer to the appropriate software product description (SPD) for complete information.

1.2.3.4 **DECprom and PROM/RT-11 Utility Programs for PROM Loading** - The VMS DECprom utility and the PB-11 system's PROM/RT-11 utility load EPROMs and PROMs. DECprom accepts input files of the .MIM, .SAV, .LDA, .EXE and .TSK types. It can generate ROM checksums in the format used by the ROM verification self-test routines in the native firmware. PROM/RT-11 accepts input files of the .MIM .LDA and .SAV types.

1.3 USING THE KXT11-C AS A STAND-ALONE SYSTEM

The KXT11-C also satisfies applications where a single-board computer is required. When used as a stand-alone system, the KXT11-C operates independently from the Q-BUS (though it can receive power from Q-BUS-wired backplanes). All the peripheral processing tools described in Section 1.2.3.1 are applicable to stand-alone application development except for the KK and KX device handlers.

CHAPTER 2

KXT11-C APPLICATION SYSTEM DEVELOPMENT CYCLES

This chapter summarizes the following procedures and decisions you must make when developing a KXT11-C peripheral processing application system or a KXT11-C stand-alone application.

- Decide which application processes the KXT11-C will perform.
- Decide how to match process requirements with the KXT11-C's functionality and performance.
- Decide which software tools you will use for KXT11-C and arbiter application development.
- Program and test each component of the application system.
- Combine and test the application system (arbiter and all KXT11-Cs)

The following sections provide a general description of these procedures and decisions. Succeeding chapters will help you understand how to most effectively use the software development tools in the application development process.

2.1 PERIPHERAL PROCESSOR APPLICATION DEVELOPMENT CYCLE

The following subsections describe the procedures to follow when developing a peripheral processor application and the specific considerations for such applications.

2.1.1 Partitioning the KXT11-C Application

Determine if your application can be partitioned to take advantage of multiple processors. There must be a set of processes that can be performed usefully in parallel and within the capabilities of the KXT11-C hardware. It must be possible to direct and monitor the progress of the process through messages or transfer of blocks of data. Some characteristics of processes that are good candidates for KXT11-C applications:

- Input/Output processes with critical interrupt latency requirements -- By assigning processes with critical interrupt latency requirements to dedicated KXT11-C peripheral processors, you can assure that the rest of the application does not interfere with the service of critical devices.

- Input/Output processes with a high frequency of interrupts -- KXT11-C peripheral processors can relieve the arbiter from the continual context switching required to process interrupt-driven I/O.

- Input/Output data reduction processes -- By assigning one or more KXT11-C peripheral processors to I/O processes that require large quantities of input data and produce a small amount of output, you can save arbiter processing time. The peripheral processor receives the data, decodes it and reduces it to the required subset, discarding the rest.

- Computational processes -- The KXT11-C can perform parallel computational operations by using the DMA transfer controller (DTC) to transfer data directly to its memory, perform the operation, and transfer the data back to Q-BUS memory.

- Real-time control functions -- You can assign a KXT11-C to control functions that require constant interaction with a device but little interaction with the main application. The arbiter can then direct the peripheral processor with high-level commands.

2.1.2 Choosing the Application Software Development Tools

Decide which of the available software tools best suits your application. You can choose from the following system application development environments.

- MicroPower/Pascal
- RT-11
- RSX-11M
- RSX-11M-PLUS

Each of these arbiter application environments contains a device handler/driver that communicates with a device handler in one or more KXT11-C applications, and a utility to load the KXT11-C with application software. With MicroPower/Pascal, you can write your KXT11-C application programs in Pascal or MACRO-11. With RT-11 and RSX-11 you can write your own system and application software in MACRO-11.

MicroPower/Pascal provides you with modular system building blocks, important functions such as semaphores, queues and ring buffers, a choice of Pascal or MACRO-11 languages, and a complete set of application building, loading and symbolic debugging facilities.

If you choose MicroPower/Pascal, be sure that its device handlers support the hardware features you need. The handlers support most but not all features of the hardware. Refer to the MicroPower/Pascal System User's Guide and the MicroPower/Pascal Runtime Services Manual for detailed information on the MicroPower/Pascal device handlers. If you decide to modify a MicroPower/Pascal device handler, you can obtain its source file from DIGITAL as part of a separate source kit.

Your arbiter application design decision depends on what you need from an operating system. RT-11 and RSX-11 are general-purpose, real-time oriented operating systems, single user and multiuser respectively. The other choice, MicroPower/Pascal, is for dedicated application systems. You can find more information about these systems in succeeding chapters and in the documentation listed in the Preface.

2.1.3 Designing the KXT11-C Application System

In a KXT11-C peripheral processor application, you must design the communication protocol between the arbiter application and the KXT11-C. This is an application-level protocol for controlling and passing data between the KXT11-C application processes and the arbiter. Think of the KXT11-C application as an intelligent I/O device used by the arbiter. You design the protocol to command the device to perform its functions (for example, start, stop, and transfer data). The commands are generally formatted into messages and sent through the two-port RAM (TPR).

MicroPower/Pascal provides the KK/KX device handler pair to facilitate TPR communication in applications using MicroPower/Pascal for the KXT11-C and arbiter. RT-11 and RSX-11 versions of the KX device handler are also provided in the KXT11-C peripheral processor tool kits to allow a KXT11-C using the MicroPower/Pascal KK device handler to communicate with an RT-11 or RSX-11 arbiter application.

In addition to commands, you can also use the TPR to send data as messages, depending on the amount and frequency of occurrence, or you can transfer it directly using the DMA transfer controller (DTC). When you use the DTC, the arbiter typically passes a message to the peripheral processor specifying the location of the data buffer to transfer. The KXT11-C application then directs the DTC locally to make the transfer.

In general, you should use the TPR to send small messages or infrequently issued messages. You should use the DTC to send large messages or frequently issued messages, especially if this can be done in parallel with other KXT11-C processes. When to use the DTC depends on the applications and must be determined on a case-by-case basis.

When using the TPR to send large but infrequent messages, if your application makes no other use of the DTC, you need not build its handler into your application, thereby saving memory space.

2.1.4 Coding the KXT11-C Application

Code and test each application process in stages that minimize the number of new variables at each stage. The exact procedure will be application-dependent, but here are some suggestions.

- Develop the KXT11-C application and arbiter application separately in a stand-alone environment.
- Isolate communication between the arbiter and KXT11-C to one process or process group on each side. Create separate test processes to validate and exercise communication linkages on each side.

- Separately test and debug all application processes before integrating them in the peripheral processing environment.

- Remove the test processes and integrate the communication processes only when you are confident that the arbiter and KXT11-C applications are operating correctly. This avoids the need for loading two or more targets for testing until the bulk of the application system has been debugged.

2.1.5 Developing Test Processes

The test processes you create should exercise and validate the functions of the arbiter and KXT11-C applications. Each process should simulate the operation of the other peripheral processing system component. For example, the test process on the arbiter should simulate the operation of the KXT11-C application. An arbiter-resident test process is useful for performing regression tests on the KXT11-C, because the actual KXT11-C application should produce the same results as those simulated by the test process.

2.1.6 Building an Application Debugging Configuration

The application configuration for debugging is slightly different from the final configuration. If you are using the PASDBG or MACDBG remote debugger you must build a debugger service module (DSM) into your application. If you are using MicroPower/Pascal you must enable the debugging option in the compiler and include debugging symbols in the MERGE, RELOC, and MIB build steps.

If you are developing a ROM application you must decide how to test it. ROM applications are usually debugged using RAM or a ROM emulator.

Next you must decide how you are going to load the application image into the target. Your options for loading the application for debugging are:

- Load using a remote debugger.

- Boot target system from TU58 DECtape II media.

- Load from arbiter's mass storage with the KUI load utility.

- Load from arbiter's mass storage with the KXT_LOAD procedure.

You cannot use the last three options with remote debuggers; thus, they are of less utility than the first option in the debugging stage of development. In general, you should use a debugger to load your application unless your application needs the memory occupied by the debugger service module. In this case, you will have to break up your application into smaller modules that can be debugged separately, or use console ODT for debugging.

The PASDBG and MACDBG debugger service modules occupy no more than 800 bytes of the application's memory image. However, all application programs written in MicroPower/Pascal will become a few percentage points larger when the debugging option is enabled in the compiler.

2.1.7 Configuring the Hardware for Application Debugging

Be sure the hardware is configured properly and the system configuration is operable. Hardware configurations for specific environments and uses of the KXT11-C are discussed in succeeding chapters of this manual. Full information on all KXT11-C options and how to configure the hardware can be found in the KXT11-CA User's Guide hardware reference manual.

2.1.7.1 **KXT11-C Debugging Configuration** - The following steps describe the hardware configuration requirements for the KXT11-C.

1. Set the base address of the TPR in the arbiter's (Q-BUS) I/O page using the system ID switch and the base address range jumper on the KXT11-C. Be sure the addresses you select do not conflict with those assigned to another KXT11-C or with other Q-BUS devices. Appropriate address selection depends on the peripheral processor's environment.

2. Select the appropriate boot/self-test power-up option with the boot/self-test switch. The boot options permit execution from ROM, booting from a TU58, waiting for boot-and-load over the Q-BUS, and console ODT operation. The self-test options determine whether self-tests, ROM tests and I/O loopback tests are performed before booting the application.

3. Configure the I/O ports used by your application. Each I/O port has several optional configurations. These are fully described in the KXT11-CA User's Guide hardware reference manual, and discussed as they relate to each MicroPower/Pascal handler in the MicroPower/Pascal Runtime Services Manual and the MicroPower/Pascal System User's Guide.

4. Select the appropriate KXT11-C memory map. This is discussed in Chapter 3 for MicroPower/Pascal applications. If you are building a ROM application, it is advisable to start debugging with a RAM configuration so you can set breakpoints in memory locations that will ultimately reside in ROM.

2.1.7.2 **System Hardware Debugging Configuration** - The system configuration required for debugging a KXT11-C application consists of the KXT11-C to be debugged, the arbiter CPU, and its Q-BUS I/O devices, if any. If you are using PASDBG or MACDBG, you must connect a serial asynchronous communication line between the host system (RT-11, RSX-11, VMS) and the KXT11-C console port. Otherwise, use the peripheral processor's console ODT by connecting a terminal to its console port.

When you use MicroPower/Pascal software in the arbiter and the KXT11-C, you can connect multiple invocations of PASDBG running on one or more host processors (depending on your host operating system) to the arbiter and KXT11-C processors over separate serial lines. This is the best way to debug peripheral processor applications. You can also use a single invocation of PASDBG and a terminal to debug several targets. Refer to Section 3.6 for detailed information.

2.1.8 Testing and Debugging the Application

Run the arbiter or the KXT11-C application with the communication test process until you detect an error; then use a debugger to isolate the error. If you use PASDBG you can debug your application symbolically, setting breakpoints, examining the state of the application, changing the contents of memory locations, and controlling the execution of the application. If you use console ODT, you must use octal numbers and a memory load map.

2.1.9 Completing the Test Cycle

Repeat the preceding procedures (Sections 2.1.1 through 2.1.8) for each processor used in the application system. When you are through with this procedure, you will be confident that each processor application is running correctly (to the extent that this can be verified with test programs). You can then proceed to testing sections of the peripheral processor application together.

2.1.10 Testing and Debugging the Integrated Application System

After completing the test cycle described above, integrate all the application system components as follows.

1. Remove all test processes.
2. Configure the peripheral processor system hardware and the debugging equipment.
3. Build a debugging configuration of the software including the actual communication processes.

It is this stage of debugging where multiple invocations of PASDBG are useful. If you tested the individual processes well, this phase should proceed easily with most of the problems being isolated in the communication process.

2.1.11 Building the Final Application System

For the final application configuration:

- Remove any debug aids.
- Generate an optimized kernel.

- Decide how the final application will be loaded.

The arbiter and KXT11-C applications must be loaded. For the arbiter application, you can use mass storage, down-line loading, or ROM. The software to support arbiter application loading depends on the facilities provided by the arbiter's operating system (MicroPower/Pascal, RT-11, or RSX-11).

KXT11-C software provides facilities for loading your application from the following media.

- Memory-image binary files on a Q-BUS mass storage device using the KXT_LOAD procedure, the KUI program or other user-created load program issuing commands to the native firmware through the command registers of the TPR
- Native ROM on the KXT11-C
- TU58 DECtape II cartridge tape subsystem or other RSP (radial serial protocol) device connected to the KXT11-C

Your choice depends on your application requirements.

2.1.12 Configuring the Final Application System Hardware

Review your final hardware configuration. You may need to select a memory map that is different from the one used during debugging. Make sure there are no CSR and vector address conflicts in the arbiter. Compare the CSR and vector assignments of devices you add for application loading with those assigned to the KXT11-C processors in the system.

2.2 STAND-ALONE PROCESSOR APPLICATION DEVELOPMENT CYCLE

The stand-alone application development cycle is the same as the application development cycle for traditional LSI-11 systems. Specific information relating to hardware and software configurations for KXT11-C stand-alone operation is included in succeeding chapters.

CHAPTER 3

PROGRAMMING A KXT11-C USING MICROPOWER/PASCAL

Developing an application for the KXT11-C when using MicroPower/Pascal is similar to developing other MicroPower/Pascal applications, with the following major differences.

- When using the MPBLD utility you must specify that the target processor is a KXT11-C.
- The MicroPower/Pascal device handlers for the KXT11-C are in the DRVK.OBJ library.
- You will debug and test a target system that has multiple processors.
- Separate memory images (.MIM files) must be built for each processor in the application system; one for the arbiter processor (if the arbiter will run MicroPower/Pascal) and one for each KXT11-C.

3.1 KXT11-C SYSTEM FILES

KXT11-C software for MicroPower/Pascal consists of %INCLUDE files, device handler interface routines, prefix files, and device handler modules (refer to Table 3-1).

%INCLUDE files define Pascal device-handler interface routines and the standard I/O packet structures that the device handlers use. You include these files in your program with the %INCLUDE command so you can use the associated interface routine or packet.

Device handler interface routines provided in the run-time hardware support library (RHSLIB.OBJ) perform all message construction, semaphore creation, queue signaling, and queue waiting needed to send and receive commands or data to and from the device handler. Using these interface routines, or the MicroPower/Pascal file system, greatly simplifies the programming of your application.

Prefix files configure I/O device handler operating characteristics and cause the inclusion of the device handler in the application image. When you use the MicroPower/Pascal utility program MPBLD, you select handlers by specifying the prefix file of the handler to be included. This file may be a MicroPower/Pascal-supplied file or a customized version of a MicroPower/Pascal-supplied file. Also, when using a device handler interface routine, you must include RHSLIB.OBJ as an additional library module. Chapters 5 and 6 of the MicroPower/Pascal System User's Guide provide complete information on using the MicroPower/Pascal libraries and bootstraps.

Table 3-1: **KXT11-C System Files Provided with MicroPower/Pascal**

File	Description
CKPFX.MAC	Prefix file for KXT11-C line-frequency clock (CK) handler (standard MicroPower/Pascal clock handler).
CFDKTC.MAC	Sample KXT11-C system configuration file that specifies all features, handlers, and debugging support. Used in generation of installation verification program, and peripheral processor verification program.
CLKLIB.PAS	%INCLUDE file for device interface routines that access clock (CK) handler.
DDPFXK.MAC	Prefix file for TU58 DECtape II (DD) device handler.
DRVK.OBJ	Device handler object module library for KXT11-C devices. Contains KK, DD, QD, XL, XS, YK, and CK device handler modules. Use this library in merge step when building device handler modules into your application.
DRVM.OBJ	Device handler object module library for mapped arbiter applications. Contains arbiter Q-BUS (KX) device handler and other standard MicroPower/Pascal device handler modules. Use this library in merge step when building arbiter application's device handler modules.
DRVU.OBJ	Device handler object module library for unmapped arbiter applications. Contains arbiter Q-BUS (KX) device handler and other standard MicroPower/Pascal device handler modules. Use this library in merge step when building arbiter application's device handler modules.
IODEF.PAS	%INCLUDE file that defines common I/O definitions for Pascal KXT11-C-resident device handler %INCLUDE files. Used when compiling KXT11-C device handlers written in Pascal and when using KXT11-C MicroPower/Pascal device handler interface routines described in Appendix H of MicroPower/Pascal Runtime Services Manual.
IOPKTS.PAS	%INCLUDE file that defines standard Pascal I/O packet structure used by KX device handler and the other standard Pascal arbiter-resident device handlers. Use this file when compiling Pascal device handlers and user programs to run on arbiter.
KKPFXK.MAC	Prefix file for KK (KXT11-C-resident two-port RAM) device handler.
KKINC.PAS	%INCLUDE file defining device handler interface routines that access KK (KXT11-C-resident two-port RAM) device handler.
KXPFX.MAC	Prefix file for KX (arbiter-resident two-port RAM) device handler. Used in peripheral processor verification and demonstration programs.

(Continued on next page)

File	Description
KXINC.PAS	%INCLUDE file defining device handler interface routines that access KX (arbiter-resident two-port RAM) device handler.
KXLINC.PAS	%INCLUDE file defining Pascal-callable procedure KXT_LOAD that loads KXT11-C from arbiter.
QDPFXK.MAC	Prefix file for DMA transfer controller (DTC) device handler (QD).
QDINC.PAS	%INCLUDE file defining device handler interface routines that access DMA transfer controller (QD) device handler.
RHSLIB.OBJ	Device-handler interface routine object module library. Contains modules for KK, KX, QD and YK device handler interface routines, KXT_LOAD (load KXT11-C from arbiter) procedure, and standard MicroPower/Pascal device handler interface routines. Use this library in merge step when building Pascal program compiled with %INCLUDE file KKINC.PAS, KXINC.PAS, QDINC.PAS, YKINC.PAS, KXLINC.PAS or RHSDSC.PAS.
XLPFXK.PAS	Prefix file for asynchronous serial line (XL) device handler for KXT11-C serial devices.
XSPFXK.PAS	Prefix file for synchronous serial line (XS) device handler.
XSINT.MAC	Source file for synchronous serial line (XS) device handler's interrupt service routine. Provided as instructional aid in creating application-specific version of XS device handler.
XSDRVK.PAS	Source file for synchronous serial line (XS) device handler. Provided as instructional aid in creating application-specific version of XS device handler.
YKPFXK.MAC	Prefix file for parallel port and counter-timer (YK) device handler.
YKINC.PAS	%INCLUDE file defining device handler interface routines used with parallel I/O (YK) device handler.
PPVFY.COM	Command file that generates and loads peripheral processor verification and demonstration programs. In RSX-11 versions of MicroPower/Pascal kit, has .CMD file type.
IOPVFY.PAS	Pascal source file for KXT11-C half of peripheral processor verification and demonstration programs.
ARBVFY.PAS	Pascal source file for arbiter half of peripheral processor verification and demonstration programs.

3.2 BUILD PROCEDURE

Building an application that runs on the KXT11-C is similar to building Q-BUS processor applications. You can build your application automatically using the MPBLD command procedure, or build your application manually issuing commands to MERGE, RELOC, MIB, the

MACRO-11 assembler, and the MicroPower/Pascal compiler. In manual builds, you must specify the appropriate library file names to each build program. In contrast, MPBLD automatically supplies the appropriate file specification for a corresponding build step, depending on CPU type and other entries.

Chapters 5 and 6 of the MicroPower/Pascal System User's Guide provide complete information on using the MicroPower/Pascal libraries and bootstraps.

3.2.1 Optimizing the Kernel

Optimizing the kernel in the KXT11-C environment is the same as for the LSI-11 environment. An optimized kernel is a customized version of the kernel that contains only the primitives your application uses; unused primitives are discarded. Building an optimized kernel minimizes its size and consequently the amount of memory the application will occupy.

You can optimize the kernel as one of the final application building steps after debugging is complete. If memory space is at a premium, you can eliminate some of the kernel's primitives before debugging. When you specify YES in the OPTIMIZE parameter of the SYSTEM macro, you can list in the PRIMITIVES macro the kernel primitives your application uses. You can then determine how much RAM and ROM your application occupies by obtaining the maps generated by the MIB and RELOC utilities. These maps show the size of the RAM and ROM used by the MicroPower/Pascal kernel, and all other processes. As you omit primitives from the list, you can obtain new copies of the maps to determine the additional amount of memory reclaimed. Alternatively, you can use an application building style that automatically configures the kernel primitives used by your application code. Chapter 7 of the MicroPower/Pascal System User's Guide describes this procedure.

3.2.2 Building KXT11-C Applications with MPBLD

MPBLD is an interactive command procedure utility that builds application images. (Refer to Appendix B of the MicroPower/Pascal System User's Guide for more information about MPBLD.) The following paragraphs list MPBLD questions for building KXT11-C applications and discuss the appropriate answers. Message text enclosed by angle brackets (<>) represents information the MPBLD procedure supplies.

Questions 1 through 5 do not relate exclusively to the KXT11-C.

Question 6: Do you wish to modify <kernel configuration file>? [no]:

If you have not created a configuration file that includes the KXT11-C, or if you are not using the standard CFDKTC.MAC configuration file supplied with MicroPower/Pascal, you should answer YES to this question. This allows you to create a configuration file or edit a copy of the standard configuration file to include the KXT11-C parameters. If you answer NO to this question, the kernel will be built with the configuration file parameters that exist in the file.

For KXT11-C applications you must specify the following macros and arguments.

PROCESSOR

You must specify the following parameters.

```
mmu=NO
type=KXT11C
vector=<address>
```

where <address> is the address of the first free memory location above the highest interrupt vector used by the application.

MEMORY

Specify the memory allocation scheme you selected. The KXT11-C has eight different jumper-selected memory mapping schemes; maps Ø through 7 described in Section 3.3.1.

KXT11C

Select parameters required for your application. Be sure the MAP parameter agrees with the memory configuration you specified in the MEMORY macro.

DEVICES

Specify the interrupt vectors of the devices used by the application. Appendix B lists standard KXT11-C device vector assignments.

Question 7: Satisfied with edit? [yes]:

At this point you exit the editor and proceed to question 8 (YES) or repeat the editing session (NO). If you answer NO, you return to the editor with the originally specified file as input. This question is asked each time you leave the editor.

Questions 8 through 1Ø do not relate exclusively to the KXT11-C.

Question 11: Mapped image? [no]:

Answer NO. The KXT11-C has no memory mapping hardware.

Question 11a: Is the target a KXT11-C? [no]:

Answer YES. This answer selects the KXT11-C device handler library DRVK.OBJ.

Question 12 does not relate exclusively to the KXT11-C.

Question 13: Does this system contain any ROM? [no]:

If the KXT11-C contains ROM, answer YES; otherwise, answer NO.

Question 14: What is the base address of RAM (octal address)? :

Enter the lowest address assigned to RAM. The answer you give will depend on the KXT11-C memory map you selected. Specify the address as an octal number.

Question 15: Instruction set hardware? {NHD, FPP, EIS, FIS} [default]:

Since the KXT11-C does not support FPP, EIS or FIS hardware, answer NHD or <RET>. The default is NHD because you answered NO to question 11.

Question 16: LSI-11/2 mode on compilations? [no]:

Answer NO. The processor is not an LSI-11/2.

Question 17: Handler Prefix filespec? :

This question is the first of a series of questions (17 through 20) that allow you to specify a standard DIGITAL-supplied KXT11-C device handler and tailor it to your particular needs by editing its prefix file. The sequence will repeat until you have no further device handlers to specify. The MicroPower/Pascal Runtime Services Manual describes the standard device handlers. The associated prefix files are described in the MicroPower/Pascal System User's Guide.

If your application uses one of the KXT11-C devices and you are not providing your own handler for this device, enter the name of one of the following DIGITAL-supplied KXT11-C device handler prefix files. If you are not using one of these handlers or you have entered all the handlers to be used, answer <RET>. MPBLD will then skip to question 21.

Prefix Filename	Device
CKPFX.MAC	Line clock
DDPFXK.MAC	TU58 DECtape II
KKPFXK.MAC	KK (KXT11-C-resident two-port RAM) handler
QDPFXK.MAC	DMA transfer controller
XLPFXK.MAC	Asynchronous serial lines SLU1, SLU2 channel A and SLU2 channel B
XSPFXK.PAS	Synchronous serial line
YKPFXK.MAC	Parallel I/O ports and counter-timers

Question 18: Is this a Pascal-implemented handler? [no] :

Answer YES if you specified the XSPFXK.PAS prefix file in the previous question.

Question 19: Do you wish to modify <prefix filespec>? [no]:

Answer YES if you want to edit the prefix file specified in question 17 to include handler features other than the default features. Refer to the MicroPower/Pascal System User's Guide for device-handler prefix file descriptions and to the MicroPower/Pascal Runtime Services Manual for device handler descriptions.

Question 20: Satisfied with the edit? [yes]:

At this point exit the editor and return to question 17 (YES) or repeat the editing session (NO). If you answer NO, you return to the editor with the originally specified file as input. This question is asked each time you leave the editor.

The remaining MPBLD questions do not relate exclusively to the KXT11-C.

3.3 SOFTWARE AND HARDWARE CONFIGURATION GUIDELINES

This section tells you how to configure the MicroPower/Pascal software and KXT11-C hardware. There are four main areas of concern:

1. The location of RAM and ROM
2. The system configuration, stand-alone or peripheral processor
3. The I/O device options
4. The bootstrap start-up option

3.3.1 Configuring Memory

KXT11-C native memory has eight jumper-selectable configurations (maps) as shown in Figure 3-1. Each map defines a particular combination of native (local) RAM and user-installed RAM or ROM residing in the user sockets of the KXT11-C.

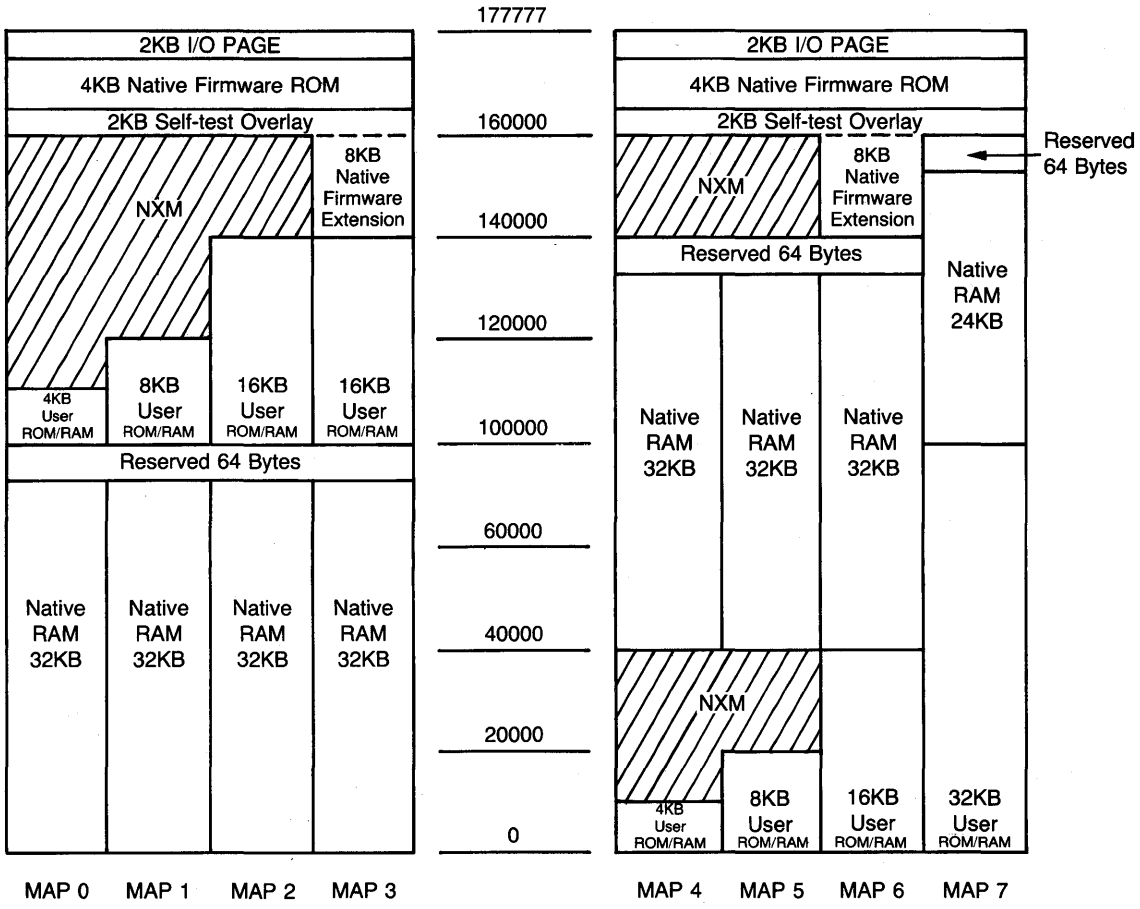


Figure 3-1: **KXT11-C Memory Map Configurations**

3.3.1.1 **Memory Configuration Steps** - When you configure KXT11-C memory, you must specify parameters in the software configuration file and install jumpers on the KXT11-C circuit board:

1. Select a map according to the requirements of your application.
2. Specify the number of the map you selected in the KXT11C configuration file macro.
3. Install additional ROM or RAM in the user sockets.

NOTE

If your application uses ROM, consider substituting RAM in its place during debugging so you can use the PASDBG debugging program.

4. Specify, in the configuration file's MEMORY macro, each contiguous block of RAM and ROM defined by the selected map and the ROM or RAM installed in the user sockets.

5. Install the map selection jumpers on the KXT11-C circuit board. (The KXT11-CA User's Guide hardware reference manual shows the location of these jumpers.)

6. Select an appropriate setting for the boot/self-test switch (see Table 3-3).

3.3.1.2 **Memory Selection Rules** - MicroPower/Pascal software can use any KXT11-C memory map option as long as you observe the following configuration rules.

1. You can select any memory map for RAM-only applications that is appropriate for the type of RAM devices installed in the user sockets.

2. You can select maps 4, 5, 6, or 7 for MicroPower/Pascal applications that use ROM. MicroPower/Pascal memory allocation rules require that you configure all ROM in memory addresses lower than those assigned to RAM. Thus, you should not specify maps Ø to 3 if your MicroPower/Pascal application uses ROM.

3. If your application will be loaded from TU58 DECtape II, you must configure RAM to start at address Ø.

4. Do not configure your application to use the highest 64 bytes of native RAM. This area is reserved for KXT11-C native firmware. The location of this 64-byte area depends on which memory map you select with the boot/self-test switch (see Figure 3-1). If you are going to use RAM in the user sockets, assign it to low memory addresses (maps 4 to 7). In this way the reserved 64-byte area will reside at the highest RAM locations rather than within the RAM address range.

5. Do not configure your application to use nonexistent memory addresses (address space identified as NXM in Figure 3-1).

6. Never configure the 8KB block of memory addresses shown in maps 3 and 6 and designated 8KB Native Firmware Extension. These addresses reference locations in the native firmware socket that are outside the address space of the native firmware ROM provided by DIGITAL.

Table 3-2 summarizes MicroPower/Pascal map usage.

Table 3-2: **MicroPower/Pascal Usage of KXT11-C Memory Maps**

Map	Usage
Ø	32KB of native RAM; 4KB RAM in user sockets mapped high. Do not allocate memory between 77700 and 77777.
1	32KB of native RAM; 8KB RAM in user sockets mapped high. Do not allocate memory between 77700 and 77777.
2	32KB of native RAM; 16KB RAM in user sockets mapped high. Do not allocate memory between 77700 and 77777.
3	32KB of native RAM; 16KB RAM in user sockets mapped high. Do not allocate memory between 77700 and 77777.
4	32KB of native RAM; 4KB ROM/RAM in user sockets mapped low. Do not allocate memory between 137700 and 137777.
5	32KB of native RAM; 8KB ROM/RAM in user sockets mapped low. Do not allocate memory between 137700 and 137777.
6	32KB of native RAM; 16KB ROM/RAM in user sockets mapped low. Do not allocate memory between 137700 and 137777.
7	24KB of native RAM; 32KB ROM/RAM in user sockets mapped low. Do not allocate memory between 157700 and 157777.

3.3.2 Using Battery Backup

The battery backup feature of the KXT11-C preserves the contents of the native RAM during power failure so that the application program can recover after a power failure.

3.3.2.1 **Configuration Considerations** - You must configure the hardware and software to set up RAM for battery backup operation.

To configure the hardware:

- Obtain an appropriate 5V battery.
- Connect the battery to the KXT11-C backplane.
- Install the battery backup jumper on the KXT11-C circuit board.
- Install ROM in the user sockets and configure the jumpers appropriately for the ROM devices used.
- Select a memory map with the ROM starting at location Ø.

Refer to Chapter 6 of the KXT11-CA User's Guide hardware reference manual for detailed information.

To configure the software:

- Edit the kernel configuration file so the POWER parameter of the KXT11C macro specifies NONVOL (that is, POWER = NONVOL). This causes the appropriate kernel power failure and power restoration routines to be built into your application.
- Specify the use of RAM and ROM using the MEMORY macro.
- Build the system so all code resides in ROM.

Configuring memory in this way affects the way you use PASDBG (see Section 3.6.7).

3.3.2.2 **Programming Considerations** - To make effective use of battery backup, you must design your application with the following information in mind.

On restart after a power failure, the MicroPower/Pascal kernel completely initializes itself and the user's application; it does not attempt to recover the application's context. Your application must recover its own context after a power failure. I/O transactions that occurred during the failure are lost.

The kernel's power-up routine generates an interrupt and maintains a recovery indicator to facilitate application recovery. Each time power is restored, the kernel's power-up routine generates a trap through vector 214. Whether or not your application responds to the interrupt depends on the actions that need to be performed when a power restart occurs. If no action is required, the application can ignore the interrupt. The application's initialization routines may also test the kernel's one-word recovery indicator (KXT$RI). If the value is nonzero, a power restart rather than a cold start is in progress. If your application requires a direct response to the interrupt, you must write a power-failure interrupt service routine that connects to this vector (CONNECT_INTERRUPT in Pascal or CINT$ in MACRO-11). The routine can examine KXT$RI to determine whether a cold start or recovery occurred so it can direct an action to take after the power failure. When your interrupt routine completes, the native firmware will reinitialize to reconfigure the peripheral controller circuits which are not protected during a power failure.

In another recovery technique your application can maintain a system of I/O status flags and perform buffer validation after a restart from a power failure. Refer to the description of the KXT11C macro in Chapter 4 of the MicroPower/Pascal System User's Guide for additional programming information.

3.3.3 Configuring the KXT11-C System Environment

This section tells you how to set up the KXT11-C for its system environment. The features you can select include stand-alone operation or peripheral processor operation, and the system's start-up options (application bootstrap device, console ODT operation, and self-test program execution). When you select these features, you must configure jumpers and switches on the KXT11-C circuit board and change some of the parameters of the KXT11C macro in the configuration file.

The system ID switch and the boot/self-test switch specify to the native firmware the desired environmental and operational features of the KXT11-C. The boot/self-test switch determines when the self-tests will be performed and how the application program will be initialized.

3.3.3.1 **Selecting Stand-Alone or Peripheral Processor Operation** - You can select stand-alone or peripheral processor operation with the system ID switch on the KXT11-C circuit board.

To use stand-alone mode, select switch positions Ø or 1. In stand-alone mode, the two-port RAM (TPR) is disabled since no Q-BUS is required. To use peripheral processing mode, select switch positions 2 to 15. You can configure a maximum of 14 KXT11-C peripheral processors in a Q-BUS system by selecting switch positions 2 to 15.

In peripheral processing mode, the TPR is enabled and connected to the Q-BUS. Each switch position selects a different base address for the TPR registers in the arbiter's I/O page. The switch selects addresses from a high or low range depending on whether or not you install the TPR base address jumper on the KXT11-C circuit board.

Appendix C lists system ID switch settings and the associated I/O page base addresses of the two-port RAM. When selecting the TPR base addresses, avoid conflicts between the KXT11-C processor's I/O page registers and I/O page registers allocated to other devices on the application system's Q-BUS.

Do not configure TPR addresses in the high range for system ID switch settings 8 to 15, as many of these addresses are allocated for standard devices and processor registers.

Each system ID switch number you select must be unique among all system ID switch numbers for KXT11-C processors on the same Q-BUS. The number need not be in a continuous sequence with the system ID switch numbers selected for other KXT11-C processors on the Q-BUS.

Specify the CSR address implied by the system ID switch and TPR base address jumper setting you select in the KX device handler prefix file. You can insert this information in the file manually with an editor or automatically during execution of MPBLD (described in Appendix B of the MicroPower/Pascal System User's Guide).

3.3.3.2 **Selecting KXT11-C Initialization and Self-test Options** - KXT11-C firmware provides initialization and diagnostic self-test functions that are selected by the boot/self-test switch on the KXT11-C circuit board. They are power-up features that provide for hardware initialization, automatic self-tests, console ODT, application bootstrapping, and execution of control routines that handle local restart interrupts to allow the arbiter to gain control of the KXT11-C. Table 3-3 summarizes the boot/self-test switch functions.

Table 3-3: **Initialization/Self-Test Options**

Initialize/Test Feature	Boot/Self-Test Switch Position*															
	Ø	1	2	3	4	5	6	7	8	9	1Ø	11	12	13	14	15
Start-up in ROM	X	X	X													
Boot from TU58 DECtape				X												
Load RAM from arbiter and run						X	X									
Enter debugging (ODT) mode					X											
Perform user ROM tests			X							X						
Perform auto self-tests	X	X	X		X					X						
Dedicated test mode									X	X						
Reserved								X				X	X	X	X	X

*Switch positions 7 and 11 to 15 are reserved. If you apply power to the KXT11-C with these positions selected, it will not function and the LED display will indicate a fatal error.

3.3.3.2.1 **ROM Application Start-up** - Boot/Self-test switch positions Ø, 1 and 2 allow you to execute a ROM application. The native firmware transfers control to the ROM code by emulating a trap to location 24. Consequently you must configure the ROM to start at address Ø (maps 4 to 7) to assure that the contents of vector 24 are preserved.

Switch position Ø inhibits the automatic self-tests. By using this switch position, you can reduce application start-up time to a minimum. Choose this position only when necessary to maintain acceptable application performance.

Switch position 1 inhibits the user ROM checksum test. This allows you to start an application that was loaded into ROM that contains no checksum or that was loaded into ROM with a checksum that was calculated by an algorithm that is incompatible with the test. Appendix E describes the procedure to use with the DECprom program to calculate and program ROM checksums for KXT11-C applications. The KXT11-CA User's Guide hardware reference manual describes the checksum algorithm.

Use switch position 2 if your ROM application supports the user ROM checksum test.

3.3.3.2.2 **TU58 and RSP Bootstrap** - If you are going to load your application image from a TU58 DECtape II drive or an RSP (radial serial protocol) link, select switch position 3. This causes the TU58 primary bootstrap to execute on power-up and load your application using RSP from a TU58 DECtape II subsystem or other system through SLU1 (DLART). Once the primary bootstrap begins receiving the boot block from the TU58, it checks that the first word of the block is in the range 240(octal) to 277(octal). If true, it considers this block to be a valid boot block and it loads the remaining blocks into RAM and starts the program. If the first word's value is invalid, the primary bootstrap continues to check, alternating between unit 0 and unit 1, until it finds a valid boot block.

When your application will be loaded from TU58 DECtape II, you must configure RAM to start at address 0.

3.3.3.2.3 **Loading from the Arbiter** - You can load your peripheral processor application from a system storage device via the arbiter. If your arbiter application runs under MicroPower/Pascal, you can use the Pascal routine KXT_LOAD described in Appendix H of the MicroPower/Pascal Runtime Services Manual. If your arbiter application runs under RSX-11 or RT-11, you can use the KUI utility program described in the KXT11-C Software Toolkit/RSX Reference Manual or KXT11-C Software Toolkit/RT Reference Manual.

Boot/Self-test switch positions 5 and 6 instruct the KXT11-C to wait for a command over the Q-BUS from KXT_LOAD or KUI. The automatic self-tests will be performed first if you select switch position 5; they will be inhibited if you select switch position 6.

3.3.3.2.4 **Automatic Self-Tests** - The automatic self-tests are a subset of the self-test functions that the native firmware provides. These tests will run when the boot/self-test switch is in positions 1, 2, 3 and 5.

The automatic self-tests include a:

- CPU test
- RAM test, native and user (if RAM resides in user sockets)
- ROM test, native and user (if selected)
- CSR test, NXM test of all native CSRs and the TPR (read-only test)
- DMA test, local-to-local DMA transfers

The tests report diagnostic errors using the LED display on the KXT11-C circuit board and the TPR system control registers. The LED display reports automatic self-test diagnostic data and general KXT11-C status. Control registers 2 and 3 of the TPR contain the test status information on completion of the nonfatal tests. Register 2 indicates which tests failed and register 3 specifies the type of failure. Register 3 is overwritten only if an error was encountered, and will contain the valid discrete error code for the last test that found an error.

The native firmware considers failure of the following self-tests as a fatal condition and will not allow the application to run.

- CPU test
- Native RAM test
- Native ROM test
- CSR test (TPR portion)

The tests report fatal errors only in the LED indicators since the firmware disables the TPR under these conditions.

The results of the remaining nonfatal tests do not prevent the application from running and will be reported in the LED display and the TPR system control registers. The user application should check for these nonfatal error conditions to see if they affect the application. Appendix F describes the procedures to follow.

Chapter 6 of the KXT11-CA User's Guide hardware reference manual discusses the meaning of all LED displays; Chapter 5 in that document describes the TPR registers and associated error codes.

If you do not select the automatic self-tests, you can reduce application start-up time to a minimum. However, these tests are useful diagnostic tools. You should bypass them only when necessary to maintain acceptable application performance.

3.3.3.2.5 **Debugging (ODT) Mode** - When you use boot/self-test switch position 4, the KXT11-C enters console ODT through SLU1 (DLART). Select this switch position when you want to debug your application with PASDBG.

3.3.3.2.6 **Dedicated Test Mode** - This mode is for dedicated diagnostic testing; the tests permit no execution of application code. You select the tests with boot/self-test switch positions 8, 9 and 10. These switch positions cause RAM to be mapped to low memory addresses by overriding the selected memory map jumper settings. ROMs jumpered for high memory mapping are mapped to their corresponding positions in low memory.

Switch positions 8 and 9 select the automatic self-tests and I/O port loopback tests; position 9 includes the ROM test. You must install loopback connectors on the I/O ports for these tests.

Switch position 10 selects the automatic self-tests, then causes the KXT11-C to wait for self-test commands through the TPR from the arbiter.

Chapter 6 of the KXT11-CA User's Guide hardware reference manual provides further information.

3.4 I/O PROGRAMMING AND CONFIGURATION CONSIDERATIONS

The following subsections discuss how you can use the device handlers DIGITAL provides for the KXT11-C with MicroPower/Pascal software.

Examples illustrate features of these handlers that differ from the non-KXT11-C MicroPower/Pascal device handlers.

3.4.1 Using the Parallel I/O (YK) Handler

The YK device handler operates the parallel I/O and counter-timer controller on the KXT11-C. It allows your MicroPower/Pascal programs to interface with the two 8-bit parallel ports, the 4-bit special-purpose I/O port, and the three independent 16-bit counter-timers. The handler provides the functions of read, write, pattern recognition, DMA read, DMA write, counter-timer set, counter-timer read, and counter-timer clear.

The parallel I/O port has configuration options you can select by editing the handler's prefix file. Many of these options interact with one another in subtle ways. You must be thoroughly familiar with the operation of this device and with the configuration rules before building an application that uses the YK device handler. (Refer to the KXT11-CA User's Guide hardware reference manual and Chapter 4 of the MicroPower/Pascal System User's Guide.)

The parallel I/O port can read and write streams or individual bits of data using its event synchronization options. Your programs can access the YK handler from MicroPower/Pascal or MACRO-11 routines by using the standard primitive I/O requests or the MicroPower/Pascal device handler interface routines listed below.

YK_PORT_READ -- Reads data from a parallel port

YK_PORT_WRITE -- Writes data to a parallel port

YK_SET_PATTERN -- Sets pattern recognition modes for a parallel port

YK_SET_TIMER -- Sets a timer to an initial value, triggers a timer after setting it, or sets a timer to periodically signal a binary semaphore

YK_READ_TIMER -- Reads a timer's or counter's current count

YK_CLEAR_TIMER -- Deactivates a timer or counter

Refer to Chapter 4 and Appendix H of the MicroPower/Pascal Runtime Services Manual for detailed information.

Typical parallel I/O port applications include:

- Transferring a series of bytes or words through a port with handshake protocol
- Setting or reading the bits of external state lines
- Generating a time base to software
- Generating a waveform for external output
- Counting pulses from an external input

The paragraphs that follow provide examples to illustrate these applications. All examples use the MicroPower/Pascal device handler interface routines listed above.

3.4.1.1 **Transferring a Series of Bytes or Words Through a Port** - This example illustrates how a program can send a stream of parallel data from a buffer to an output port connected to a parallel printer. The example makes full use of the parallel I/O ports. The printer must operate under handshake signal control and the user's program must monitor the printer's error status data and send control/indicator data to it.

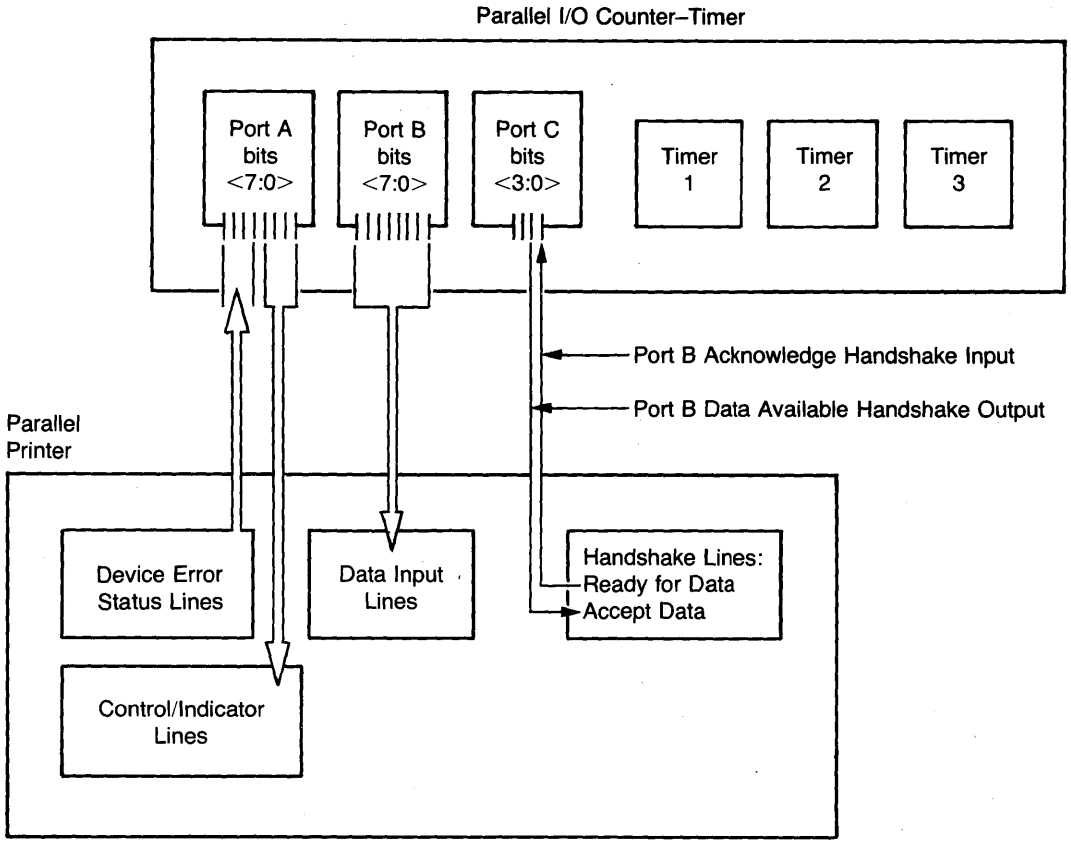


Figure 3-2: **Sending Data to a Parallel Printer**

Configuration

Figure 3-2 shows the external connections between the parallel I/O port and the printer. The listing of the YK handler prefix file that follows shows you how to specify this configuration.

Port A is configured as a bit port. The least significant bits (Ø to 3) are configured as output lines to the printer's control and indicator lines. The most significant bits (4 to 7) are configured by default as input lines to receive error input signals from the printer. Pattern matching is specified to detect changes in error input signals.

Port B is configured as an output port with interlocked handshake for the data bytes going to the printer.

Port C is configured as a bit port with bit Ø as an output for the printer's data-accepted line. Bit 1 is configured by default as an input for the printer's ready-for-data line.

```
      .MCALL  YKCI$
YKCI$                           ; Beginning of Prefix File


; Port A Setup

YKCP$ CHAN=A, PTYPE=YK$BIT      ; Port A is a Bit Port.
YKCP$ CHAN=A, OUT=17            ; Least Significant 4 bits are outputs.
YKCP$ CHAN=A, PAT=TRUE          ; Pattern Recognition to detect
                                ; changes in Error Input Lines.

; Port B Setup

YKCP$ CHAN=B, PTYPE=YK$OUT      ; Port B is an Output Port
YKCP$ CHAN=B, HSH=YK$INL        ; with Interlocked Handshake.


; Port C Setup

YKCP$ CHAN=C, PTYPE=YK$BIT      ; Port C is a Bit Port.
YKCP$ CHAN=C, OUT=2             ; Bit 1 is a Handshake Output.

YKCE$                           ; End of Prefix File
```

MicroPower/Pascal Program Fragment

The following MicroPower/Pascal program fragment illustrates how a program can write a line of data to the YK handler configured to operate a parallel printer.

The %INCLUDE commands in the following example refer to the RT-11 logical device LB:. If your host environment is RSX-11 or VMS, substitute the appropriate logical device. Refer to the MicroPower/Pascal Installation Guide for this information.

```
%include 'LB:IODEF'             { get the INCLUDE file for the common}
                                { I/O definitions}
%include 'LB:YKINC'             { get the INCLUDE file for the
                                { device handler interface routines}
VAR
    LP_DATA : PACKED ARRAY [1..512] OF CHAR ;   { LP data }
    DATA_LEN : INTEGER ;                        { number of characters}
    RESULTS  : IO$STATUS ;                      { write results }

BEGIN

RESULTS := YK_PORT_WRITE
           ( PORT_NUM := PORT_B ,
             BUFFER := LP_DATA ,
             BYTE_COUNT := DATA_LEN ) ;
END ;
```

3.4.1.2 **Transferring Data to and from Analog Devices** - This is an example of analog-to-digital (A/D) and digital-to-analog (D/A) I/O. It illustrates how a program can receive a constant flow of 8-bit A/D data, filter the data in some way, then send the reduced data to a D/A converter. The example uses two external input clocks to control the rate of flow of input and output data individually.

This example assumes that:

An input buffer cannot be filled with incoming data in less time than it takes to filter and process the previous input buffer.

The rate of data output equals the input data rate times the ratio of output buffer size divided by input buffer size.

These assumptions prevent input data from being lost and keep output data flowing continuously. The example does not incorporate a method of detecting a violation of either assumption which could occur if the external clocks are misadjusted.

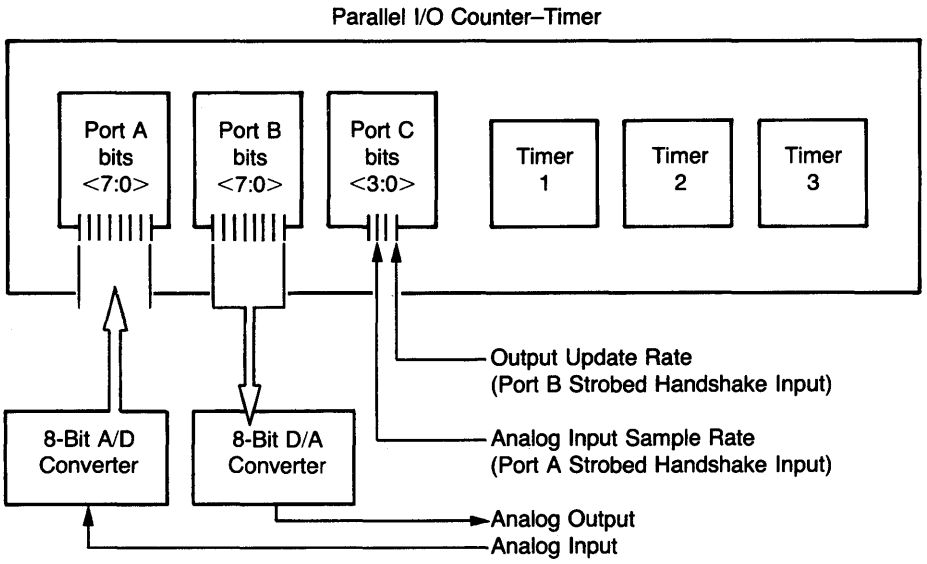


Figure 3-3: **Transmitting Data to and Receiving Data from Analog Devices**

Configuration

Figure 3-3 shows the external connections between the parallel I/O port and the A/D and D/A converters. The listing of the YK handler prefix file that follows shows you how to specify this configuration.

The YK configuration macros set up port A as an input port with a strobed handshake and port B as an output port with a strobed handshake.

```
      .MCALL   YKCI$
YKCI$                                       ; Beginning of Prefix File.

;Port A Setup

YKCP$ CHAN=A, PTYPE=YK$INP, HSH=YK$STR      ; Port A is an input port
                                            ; with strobed handshake.

;Port B Setup

YKCP$ CHAN=B, PTYPE=YK$OUT, HSH=YK$STR      ; Port B is an output port
                                            ; with strobed handshake.

;Port C Setup

YKCP$ CHAN=C, PTYPE=YK$BIT, OUT=<YK$B1+YK$B3> ; Port C is a bit port
                                            ; with bits 1 and 3 set for
                                            ; output

YKCE$                                       ; End of Prefix File
```

MicroPower/Pascal Program Fragment

The following MicroPower/Pascal program fragment illustrates how a program can use the YK handler to read data from an A/D converter and write data to a D/A converter.

The %INCLUDE commands in the following example refer to the RT-11 logical device LB:. If your host environment is RSX-11 or VMS, substitute the appropriate logical device. Refer to the MicroPower/Pascal Installation Guide for this information.

```
[ SYSTEM(MICROPOWER)] PROGRAM C3EX2 ;

{$NOLIST}
%include 'LB:IODEF'             { get the INCLUDE file for the common}
                                { I/O definitions}
%include 'LB:YKINC'             { get the INCLUDE file for the
                                { device handler interface routines}
{$LIST}

CONST

    IN_BUF_SIZE = 256 ; {size of input buffers }
    LAST_BUF = 3 ;    { number of input/output buffers pairs, a minimum of
                        3 are needed because the filter will (at certain
                        points in time) be reading data from the end of a
                        first input buffer and the beginning of a second
                        at the same time.  Meanwhile the YK handler will
                        be filling a third. }

    OUT_BUF_SIZE = 64 ; { The size of the output buffers must be proportional
                          to the data reduction ratio. }

TYPE
    IN_DAT_BUF = PACKED ARRAY [1..IN_BUF_SIZE] OF BYTE_RANGE ; { data buffer }
    IN_DAT_AREA = RECORD
              DAT_LEN : UNSIGNED ;     { length of buf, or # of bytes in it}
              DAT_SECT : IN_DAT_BUF;   { input data buffer }
              END ;

    OUT_DAT_BUF = PACKED ARRAY [1..OUT_BUF_SIZE] OF BYTE_RANGE; { out buffer }
    OUT_DAT_AREA = RECORD
              DAT_LEN : UNSIGNED ;     { length of buf, or # of bytes in it}
              DAT_SECT : OUT_DAT_BUF; { output data buffer }
              END ;
```

```
VAR
    IN_BLK : ARRAY [1..LAST_BUF] OF IN_DAT_AREA ;     { input records }
    OUT_BLK : ARRAY [1..LAST_BUF] OF OUT_DAT_AREA;    { output records }
    IN_STATUS : IO$STATUS ;                { input status }
    OUT_STATUS : IO$STATUS ;               { output status }
    CURRENT_BUF : UNSIGNED ;               { current buffer being filtered }
    IN_REPLY_DESC : STRUCTURE_DESC ;       { input Reply sem descriptor }
    OUT_REPLY_DESC : STRUCTURE_DESC ;      { output Reply sem desc }
    IN_REPLY_PKT : YK_REPLY ;              { YK reply Packet }

{ Beginning of Process }

BEGIN

OUT_REPLY_DESC .ID := NO_SEM ;             { prevent replies from output}

IF CREATE_QUEUE_SEMAPHORE ( DESC := IN_REPLY_DESC ) THEN
    BEGIN
    { Send all the input buffers to the YK handler }
    FOR CURRENT_BUF := 1 TO LAST_BUF  DO
      BEGIN
      IN_BLK [ CURRENT_BUF ] .DAT_LEN := IN_BUF_SIZE ;
      IN_STATUS := YK_PORT_READ  (
                     PORT_NUM := PORT_A ,
                     BUFFER := IN_BLK [ CURRENT_BUF ] .DAT_SECT ,
                     BYTE_COUNT := IN_BLK [ CURRENT_BUF ] .DAT_LEN,
                     REPLY := ADDRESS (IN_REPLY_DESC ) ) ;
      END ;
    WHILE (    ( IN_STATUS . ERROR_CODE = IE$NORMAL )
    AND        ( OUT_STATUS . ERROR_CODE = IE$NORMAL )   ) DO

      BEGIN
      RECEIVE (    VAL_DATA := IN_REPLY_PKT ,
                   VAL_LENGTH := SIZE (YK_REPLY) ,
                   DESC := IN_REPLY_DESC ) ;

      IF IN_REPLY_PKT .STATUS .ERROR_CODE <> IE$NORMAL  THEN
          IN_STATUS := IN_REPLY_PKT .STATUS
      ELSE
          BEGIN
          {  now go do the filtering of the data coming in from
             IN_BLK [ CURRENT_BUF ] .DAT_SECT and put the results
             into OUT_BLK [CURRENT_BUF] .DAT_SECT .   Also put
             the number of output bytes into the length entry of
             OUT_BLK .    }

          OUT_STATUS := YK_PORT_WRITE  (
                        PORT_NUM := PORT_B ,
                        BUFFER := OUT_BLK [ CURRENT_BUF].DAT_SECT ,
                        BYTE_COUNT:=OUT_BLK [ CURRENT_BUF ].DAT_LEN ,
                        REPLY := ADDRESS (OUT_REPLY_DESC ) ) ;

          IF CURRENT_BUF = LAST_BUF THEN
              CURRENT_BUF := CURRENT_BUF + 1
          ELSE
              CURRENT_BUF := 1 ;

          IN_BLK [ CURRENT_BUF ] .DAT_LEN := IN_BUF_SIZE ;
          IN_STATUS := YK_PORT_READ  (
                     PORT_NUM := PORT_A ,
                     BUFFER := IN_BLK [ CURRENT_BUF ] .DAT_SECT ,
                     BYTE_COUNT := IN_BLK [ CURRENT_BUF ] .DAT_LEN,
                     REPLY := ADDRESS (IN_REPLY_DESC ) ) ;
          END ;
      END ;
    END ;
END.
```

3.4.1.3 **Receiving Data from a 12-Bit Analog-to-Digital Converter** - This example shows how you can use the parallel I/O port to receive analog data of higher resolution than 8 bits and also use one of the port's timers to control the sample rate instead of supplying an external clock.

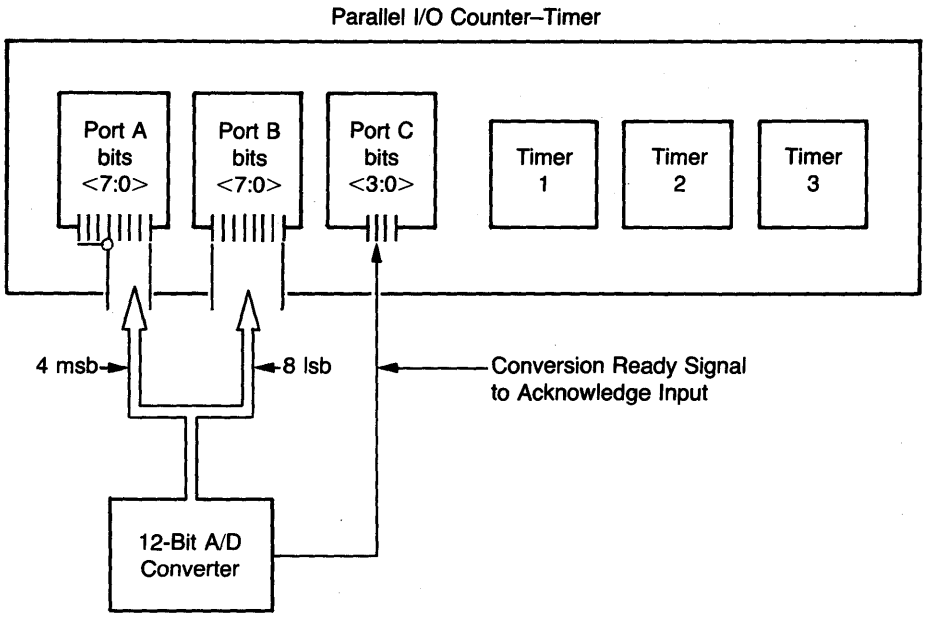


Figure 3-4: **Receiving Data from a 12-Bit Analog-to-Digital Converter**

Configuration

Figure 3-4 shows the external connections between the parallel I/O port and the 12-bit A/D converter. The listing of the YK handler prefix file that follows shows you how to specify this configuration.

```
      .MCALL  YKCI$

YKCI$                                       ;Beginning of Prefix File

;Port A Setup

YKCP$ CHAN=A, PTYPE=YK$INP, HSH=YK$STR ; Port A is an input port with
                                        ; strobed handshake.

;Port B Setup

YKCP$ CHAN=B, PTYPE=YK$BIT, PLNK=TRUE  ; Port B is a bit input port
                                        ; linked to Port A.

;Port C Setup

YKCP$ CHAN=C, PTYPE=YK$BIT, OUT=<YK$B3+YK$BØ> ;  Port C is a bit port with
                                        ; bits Ø and 3 set for output.

;Timer Setup

YKCT$ TNUM=3, TEXTO=YES, TOUT=YK$TPL   ; Timer 3 enabled for pulsed output.

YKCE$                                   ; End of Prefix File
```

MicroPower/Pascal Program Fragment

Programming for the 12-bit A/D input is similar to that shown in the portion of the previous example that received 8-bit A/D values. You still have to provide the YK handler with multiple buffers so it can continue to receive input after it returns a buffer. In addition, a SET TIMER command must be sent to provide the clock output that triggers the A/D device.

3.4.1.4 **Using the Counter-Timers to Count External Pulses** - This example shows how to create an external pulse counter that is unaffected by timing delays caused by software. Timing accuracy is guaranteed by setting up the hardware to stop the counting process at the instant it presents an interrupt request to the processor.

The example uses timer 3 to time the counting interval and timer 1 to count the number of pulses. Timer 3 causes the software to be signaled and stops timer 1 from counting by shutting off its gate input. That way, when the software reads the number of counts from timer 1 it will be exact. As explained in the hardware documentation, timer 3 must be set up to have a one-shot output and run in the noncontinuous mode to accomplish this. To expand this into a 32-bit counter, you can link timer 2 to timer 1.

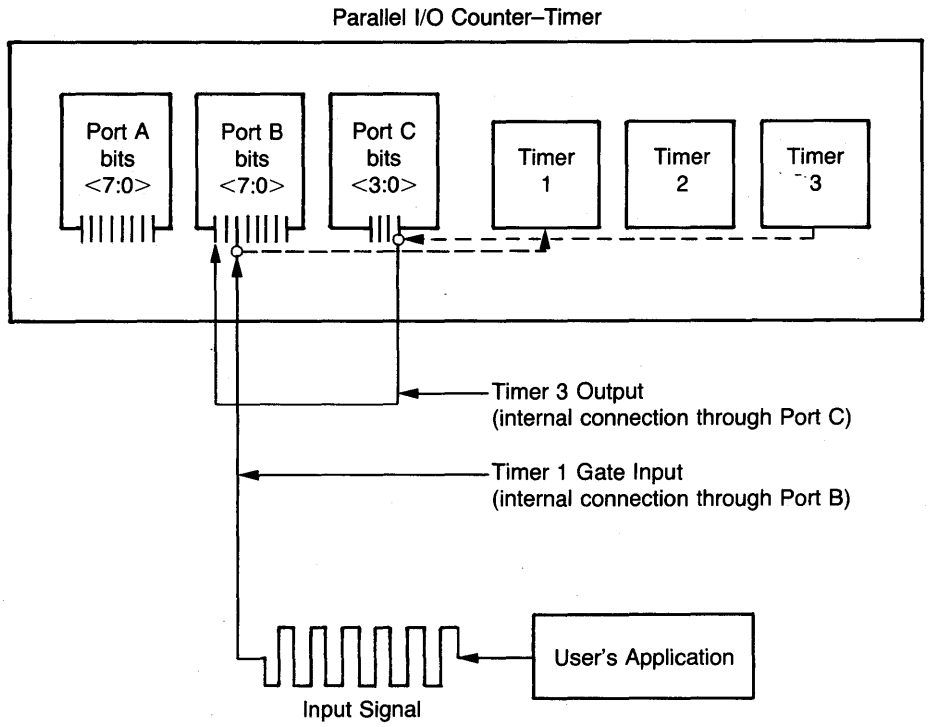


Figure 3-5: **Using the Counter-Timers to Count External Events**

Configuration

Figure 3-5 shows the external connections between the parallel I/O port and the application environment. The listing of the YK handler prefix file that follows shows you how to specify this configuration.

```
      .MCALL  YKCI$

YKCI$                                   ; Beginning of Prefix File


;Port B Setup

YKCP$ CHAN=B, PTYPE=YK$BIT              ; timer 1's gate input via bit 7


;Port C Setup

YKCP$ CHAN=C, PTYPE=YK$BIT, OUT=1       ; Timer 3's output via bit Ø.


;Timer Setup

YKCT$ TNUM=1, TEXTG=YES                 ; Enable timer 1's gate input.
YKCT$ TNUM=1, TEXTC=YES                 ; Enable timer 1's count input
                                        ; via port B bit 5.
YKCT$ TNUM=3, TEXTO=YES, TOUT=YK$T1S    ; Enable timer 3's output
                                        ; in one-shot mode.

YKCE$                                   ; End of Prefix File.
```

MicroPower/Pascal Program Fragment

The following MicroPower/Pascal program fragment illustrates how a program can use the YK handler to count pulses.

```
[ SYSTEM(MICROPOWER)] PROGRAM C3EX3 ;

{$NOLIST}
%include 'DEF:IODEF.PAS'
%include 'DEF:YKINC.PAS'
{$LIST}

CONST
      COUNT_INT = 100 ;                    { define the counting interval }

VAR
      YK_IO_STATUS : IO$STATUS ;           { Status code returned from YK }
      NIL_REPLY_DESC : STRUCTURE_DESC ;    { Dummy reply descriptor }
      REPLY_DESC : STRUCTURE_DESC ;        { Actual reply descriptor }
      REPLY_PKT : YK_REPLY ;               { Reply packet from the YK handler}
      COUNTS : UNSIGNED ;                  { Number of counts during interval}

BEGIN

NIL_REPLY_DESC .ID := NO_SEM ;

IF CREATE_QUEUE_SEMAPHORE (DESC := REPLY_DESC ) THEN
    BEGIN
      { First set the counter to zero }
      YK_IO_STATUS := YK_SET_TIMER
                              ( TIMER_NUM := TIMER_1 ,
                                TIMER_VALUE := 0 ,
                                MODE := [ init_constant, trigger ] ,
                                REPLY := ADDRESS (NIL_REPLY_DESC)   ) ;

      { Now start the time interval, and the counter }
      YK_IO_STATUS := YK_SET_TIMER
                              ( TIMER_NUM := TIMER_3 ,
                                TIMER_VALUE := COUNT_INT ,
                                MODE := [ init_constant, trigger ] ,
                                REPLY := ADDRESS ( REPLY_DESC )  ) ;

      { Wait for the time interval to expire }
      RECEIVE ( VAL_DATA := REPLY_PKT,
                VAL_LENGTH := SIZE (YK_REPLY) ,
                DESC := REPLY_DESC ) ;

      { Read the number of pulses, and put the answer in COUNTS }
      YK_IO_STATUS := YK_READ_TIMER ( timer_num := TIMER_1 ,
                                      pt_time := ADDRESS ( COUNTS ) ) ;

      END ;
END.
```

3.4.1.5 **Using the Counter-Timers to Supply External Pulses** - This example shows how to set up a timer to generate external pulses but not reply at the completion of any of the cycles. This configuration is useful when you need an output waveform that is independent from the application software time base.

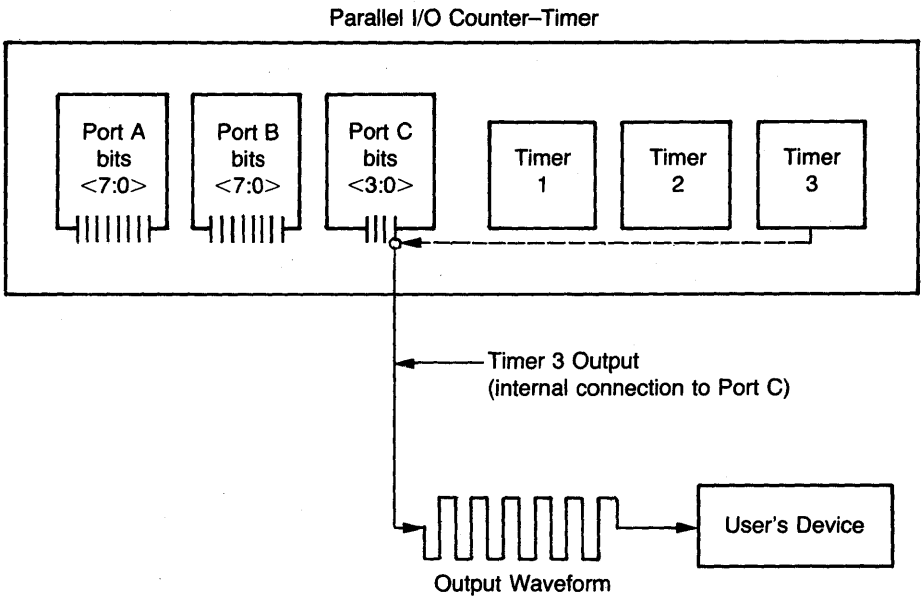


Figure 3-6: **Using the Counter-Timers to Supply External Pulses**

Configuration

Figure 3-6 shows the external connections between the parallel I/O port and the application environment. The listing of the YK handler prefix file that follows shows you how to specify this configuration.

```
      .MCALL  YKCI$

YKCI$                                   ; Beginning of Prefix File


;Port C Setup

YKCP$ CHAN=C, PTYPE=YK$BIT, OUT=1       ; Timer 3's output via bit Ø


;Timer Setup

YKCT$ TNUM=3, TEXTO=YES                 ; Enable timer 3's output and
                                        ; leave it in the default (square
                                        ; wave mode).

YKCE$                                   ; End of Prefix File
```

MicroPower/Pascal Program Fragment

The following MicroPower/Pascal program fragment illustrates how a program can use the YK device handler to generate a software-independent time base.

The %INCLUDE commands in the following example refer to the RT-11 logical device LB:. If your host environment is RSX-11 or VMS, substitute the appropriate logical device. Refer to the MicroPower/Pascal Installation Guide for this information.

```
[ SYSTEM(MICROPOWER)] PROGRAM C3EX4 ;

{$NOLIST}
%include 'LB:IODEF'             { get the INCLUDE file for the common}
                                { I/O definitions}
%include 'LB:YKINC'             { get the INCLUDE file for the
                                { device handler interface routines}
{$LIST}

CONST
      COUNT_INT = 100 ;         { define the counting interval }

VAR
      YK_IO_STATUS : IO$STATUS ;          { Status code returned from YK }
      NIL_REPLY_DESC : STRUCTURE_DESC ;   { Dummy reply descriptor }
BEGIN

NIL_REPLY_DESC .ID := NO_SEM ;

    BEGIN
      { First set the counter to the initial value and trigger it. }
      YK_IO_STATUS := YK_SET_TIMER
                        ( TIMER_NUM := TIMER_3 ,
                          TIMER_VALUE := COUNT_INIT ,
                          MODE := [ init_constant, trigger, continuous ] ,
                          REPLY := NIL_REPLY_DESC ) ;

    END ;
END.
```

3.4.2 Using the Asynchronous I/O (XL) Device Handler

The KXT11-C XL device handler supports asynchronous I/O operations on devices connected to any of the three serial I/O ports on the KXT11-C peripheral processor. This handler is similar to the non-KXT11-C XL handler described in the MicroPower/Pascal Runtime Services Manual except it also supports the multiprotocol communication controller (7201 chip) on the KXT11-C. This allows the KXT11-C to concurrently service the three serial ports SLU1, SLU2 channel A, and SLU2 channel B.

The first serial I/O port on the KXT11-C (SLU1) is a standard DL asynchronous receive/transmit (DLART) device. The second port (SLU2 channel A) provides all the features of a DLV11-E, including modem control, but is a distinct hardware type. The third port (SLU2 channel B) provides all the features of a standard DLART device but is a different hardware device. When you use SLU2 channel A or channel B, you must specify one of the following TYP parameters for the LINDF$ macro in the handler's prefix file.

TYP	Description of Type
TT$DM	Multiprotocol channel using no modem control
TT$DMP	Multiprotocol channel using partial modem control
TT$DMM	Multiprotocol channel using full modem control (SLU2 channel A only)

A description of the modem control options can be found in the MicroPower/Pascal System Users Guide.

3.4.3 Using the Synchronous I/O (XS) Device Handler

The XS device handler supports synchronous serial I/O via the KXT11-C multiprotocol communication controller (7201 chip) SLU2 channel A, allowing you to establish an elementary bit-oriented communication channel. The handler performs the following functions of bit-oriented communication procedures.

Synchronization (flag detection)

Transparency (bit stuffing)

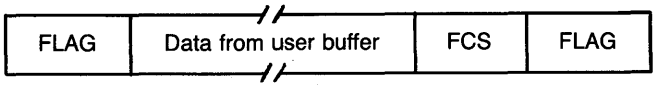
Invalid frame detection

Frame abortion detection

Frame check sequence (FCS) checking/calculation

The handler can be used by user-written software as a component in performing such bit-oriented communication procedures as CCITT X.25, ISO HDLC, IBM SDLC, ANSI ADCCP, and others. However, the XS handler does not, in itself, completely perform any of these procedures. The XS handler provides no explicit modem control, although the line-enable and line-disable services control modem signals as necessary to activate and deactivate a modem.

3.4.3.1 **XS Device Handler Functions** - The XS handler provides basic ISO HDLC style framing and error detection.



Data sent by the handler has the FCS (frame check sequence) appended to form a frame. If any errors are detected as the handler sends the frame it is aborted and resent. Frames received by the handler use the embedded FCS to verify the frame; the FCS is then removed. If FCS checking indicates the frame was received in error or if the frame was aborted or invalid, the frame is discarded with no indication to the user. Otherwise the data is returned to the user.

The source files of this handler are provided on the MicroPower/Pascal distribution kit to assist you in writing your own handler for the multiprotocol controller.

3.4.3.2 **Functions Performed by User Software** - Because frames received in error are discarded, user software must be able to determine when a frame has not been received. A sequence number/timeout scheme can be used for this purpose: A sequence number is included in the data portion of each frame. As the user software receives each frame it responds by sending another frame acknowledging receipt with the first frame's sequence number. If, after a period of time, the originator of the first frame determines that no acknowledgment has been received, it resends that frame.

For further information about the techniques of data communication, refer to Technical Aspects of Data Communication, John E. McNamara, DIGITAL Press, 1982, or a good book of your choice on data communication principles.

3.4.3.3 Accessing the XS Handler -

Configuration

The XS handler makes no assumptions about the configuration of the multiprotocol communication controller, other than assuming the interrupts have been enabled for the device. Refer to the KXT11-CA User's Guide hardware reference manual for complete information on this device.

MicroPower/Pascal Program Fragment

This example shows you how to access the XS handler only; it is not an example of data communication protocol. The handler is configured with the standard XSPFXK.PAS prefix file.

The %INCLUDE command in the following example refers to the RT-11 logical device LB:. If your host environment is RSX-11 or VMS, substitute the appropriate logical device. Refer to the MicroPower/Pascal Installation Guide for this information.

```
program exampl;                    { XS handler use -- program fragment }

{$nolist}
%include 'LB:iopkts.pas'
{$list}

const

  get_frame = IF$READ_PHYSICAL;
  send_frame = IF$WRITE_PHYSICAL;
  enable_line = IF$DEV_DEP_Ø1;
  disable_line = IF$DEV_DEP_Ø2;
  kill_requests = IF$DEV_DEP_Ø3;

  buf_size = 33 ;                  { size of buffers }

  start_pattern = '1234567890ABCDEFGHIJKLMNOPQRSTUVW' ;  { beginning pattern }

  PACKET_LENGTH = 22;

type

  buffer = record
    num : unsigned;
    dat : packed array [1..buf_size] of CHAR ;
  end;

var

  I : integer;
  garbage : BOOLEAN;

  reply_queue : structure_desc;    { SDB for my reply queue }
  handler_queue : structure_desc; { SDB for the handler }

  read_request, write_request,
  enable_request, disable_request : IO$REQ_PKT;

  command_reply : IO$REPLY_PKT;

  write_buffer, read_buffer : buffer;

begin

{*
 * First initialize things
 *}
```

```
  garbage := create_queue_semaphore (NAME := 'RPLYQ ' ) ;

  init_structure_desc ( desc := reply_queue , name := 'RPLYQ ' ) ;
  init_structure_desc ( desc := handler_queue , name := '$XSA  ' ) ;

  WITH enable_request DO BEGIN
    oper := enable_line;
    reply_sem := reply_queue.ID ;
  END ;

  WITH write_request DO BEGIN
    oper := send_frame ;
    reply_sem := reply_queue.ID ;
  END ;

  WITH read_request DO BEGIN
    oper := get_frame ;
    reply_sem := reply_queue.ID ;
  END ;

  write_buffer.dat := start_pattern ;

  SEND (
    VAL_DATA := enable_request,
    VAL_LENGTH := PACKET_LENGTH,
    DESC := handler_queue ) ;

  RECEIVE (
    VAL_DATA := command_reply,
    VAL_LENGTH := PACKET_LENGTH,
    DESC := reply_queue ) ;
{*
 * Now loop here, sending a read and write request, and wait for replies.
 *}
  WHILE TRUE DO BEGIN

    SEND (
      VAL_DATA := read_request,
      VAL_LENGTH := PACKET_LENGTH,
      REF_DATA := read_buffer,
      REF_LENGTH := SIZE ( buffer ) ,
      DESC := handler_queue ) ;

    write_buffer.num := write_request.sequence;

    SEND (
      VAL_DATA := write_request,
      VAL_LENGTH := PACKET_LENGTH,
      REF_DATA := write_buffer,
      REF_LENGTH := SIZE ( buffer ) ,
      DESC := handler_queue ) ;

    FOR I := 1 TO 2 DO RECEIVE (
      VAL_DATA := command_reply,
      VAL_LENGTH := PACKET_LENGTH,
      DESC := reply_queue ) ;

  END;
END.
```

3.4.4 Using the TU58 DECtape II (DD) Device Handler

The DD device handler is similar to the standard MicroPower/Pascal arbiter DD handler except it includes support for SLU2 channel A and SLU2 channel B (multiprotocol communication controller) on the KXT11-C. This lets you connect a TU58 DECtape II subsystem to any of the serial lines on the KXT11-C.

The DD device handler supports logical and physical I/O operations on a TU58 cartridge tape subsystem. The handler also supports read-with-increased-threshold and write-verify options and reports the storage capacity of the device in terms of logical blocks.

Requests for logical read/write functions specify the initial tape address in terms of a 512-byte logical block. Logical block numbers range from Ø to 511.

Requests for physical read/write functions specify the initial tape address in terms of a 128-byte physical record (that is, tape positioning is performed in special address mode). Physical record numbers range from Ø to 2Ø47. Multirecord transfers exceeding 128 bytes read from or write to physically sequential records on the tape.

The functions provided by the handler are described in further detail in the MicroPower/Pascal Runtime Services Manual.

The KXT11-C DD handler is in the KXT11-C handler library, DRVK.OBJ. To use the KXT11-C DD handler, you must edit its prefix file, DDPFXK.MAC, and then use the prefix file to build the KXT11-C DD handler into your application software. Software configuration procedures are described in the MicroPower/Pascal-RT-11 System User's Guide. In the TYPRM parameter of the CTFCF$ macro you must specify the terminal type and bit rate of the KXT11-C line you intend to use. The type parameter can be one of the TT$DMx parameters described in Section 3.4.2.

3.4.5 Using the QD (DMA Transfer Controller) Handler

The QD device handler provides a standard device handler interface to the two-channel DMA transfer controller (DTC) on the KXT11-C. The QD handler moves data from place to place without the assistance of the CPU. Any memory location can serve as a source or destination including I/O device registers (with certain restrictions). However, at least one location, source or destination, must be local to the KXT11-C.

Your programs can access the QD handler from MicroPower/Pascal or MACRO-11 routines by using the standard primitive I/O requests or the MicroPower/Pascal device-handler interface routines listed below. All examples shown in this section use these routines.

$DMA_TRANSFER -- Performs DMA transfer

$DMA_SEARCH -- Performs DMA search

$DMA_SEARCH_TRANSFER -- Performs transfer while searching

$DMA_ALLOCATE -- Allocates DMA channel

$DMA_DEALLOC -- Deallocates DMA channel

$DMA_GET_STATUS -- Gets status of DMA unit

Chapter 4 and Appendix H of the MicroPower/Pascal Runtime Services Manual provide detailed information about the MACRO-11 and MicroPower/Pascal interface to the QD device handler.

You can use the QD handler and its associated interface routines to:

- Transfer data to and from Q-BUS memory.
- Transfer data to and from local memory.
- Search for data.
- Transfer to and from local I/O devices.
- Access the Q-BUS I/O page.
- Assure access to a DMA Channel.

3.4.5.1 **Transferring Between Local Memory and Q-BUS Memory** - The QD handler can transfer data between KXT11-C and system memory. The following program fragment shows such a transfer.

```
CONST
  BUFSIZE = %o'2000';
  QBUSBUF = DMA$ADDRESS ( %O'70000', %O'2', DMA$NOIO, DMA$WAIT_0, DMA$UP,
                          DMA$NOWFR, DMA$NOBYTE, DMA$QBUS );
VAR
  buf1 : PACKED ARRAY[1..BUFSIZE] OF BYTE_RANGE;
  address_1 : DMA$ADDRESS;

BEGIN
      address_1 := DMA$NORM_IBUS_ADDRESS;
      address_1.low := (ADDRESS(BUF1))::UNSIGNED;

      $DMA_TRANSFER (                  { transfer... }
      UNIT := un,                      { on this unit }
        SOURCE := address_1,           { from my local buffer }
      DEST := QBUSBUF,                 { to the Q-BUS buffer }
      COUNT := BUFSIZE )               { this much }
```

3.4.5.2 **Transferring Data Within Local Memory** - The QD handler can transfer data within KXT11-C memory. The calls are similar to the previous example except the destination address is replaced by a local address. In the following example fragment, some of the addressing options are also used. This transfer will invert a local (to the KXT11-C) buffer end for end, in byte mode, which will have the effect of reversing the byte order.

```
VAR
  buf1, buf2 : PACKED ARRAY[1..BUFSIZE] OF BYTE;
  address_1, address_2 : DMA$ADDRESS;

BEGIN
      address_1 := DMA$NORM IBUS_ADDRESS;
      address_1.low := (ADDRESS(BUF1))::UNSIGNED;
      address_2 := DMA$NORM_IBUS ADDRESS;
      address_2.low := (ADDRESS(BUF2))::UNSIGNED+BUFSIZE-1;
      address_2.ws := DMA$WAIT 4;
      address_2.inc := DMA$DOWN;
      address_2.bm := DMA$BYTE;

      $DMA_TRANSFER (                  { transfer... }
      UNIT := un,                      { on this unit }
        SOURCE := address_1,           { from my local buffer }
      DEST := address_2,               { to the other local buffer }
      COUNT := BUFSIZE )               { this much }
```

3.4.5.3 **Using the Search Option** - The QD handler supports two search options, with or without data transfer.

3.4.5.3.1 **Searching with Transfer** - This option can transfer variable-length messages. The search looks for an end-of-message character and terminates the transfer at that point. The following example shows how to copy variable-length messages.

```
VAR
  buf1, buf2 : PACKED ARRAY[1..BUFSIZE] OF BYTE;
  address_1, address_2 : DMA$ADDRESS;

BEGIN
      address_1 := DMA$NORM_IBUS_ADDRESS;
      address_1.low := (ADDRESS(BUF1))::UNSIGNED;
      address_2 := DMA$NORM_IBUS_ADDRESS;
      address_2.low := (ADDRESS(BUF2))::UNSIGNED;

      k := $DMA_SEARCH_TRANSFER (       { transfer... }
      UNIT := un,                       { on this unit }
        SOURCE := address_1,            { from my local buffer }
      DEST := address_2,                { to the other local buffer }
        VAL := 0,                       { looking for a 0... }
        MASK := %o'377',                { in the high byte }
      COUNT := BUFSIZE );               { this much }
```

3.4.5.3.2 **Searching Without Transfer** - This option allows a program to determine the length of a buffer of data by searching for an end-buffer character. Searching can also be used to poll a location.

3.4.5.4 **Transferring to and from Local I/O Devices** - The QD handler can transfer data to and from KXT11-C local I/O devices without the involvement of the processor. The YK handler supports interaction with the QD handler. The SLU2 serial lines can be accessed directly from user-written software in DMA mode, but DIGITAL-supplied software does not support DMA access to the serial ports.

3.4.5.4.1 **Parallel I/O Using DMA** - The YK and QD handlers in conjunction can perform DMA transfers to and from the parallel I/O port. The following example shows a sequence of commands that transfer data from the parallel port to local memory on the KXT11-C. The example assumes the port A request line is connected to DMA channel 1 and that channel is allocated for use by this process.

The %INCLUDE commands in the following example contain references to RT-11 logical device LB:. If your host environment is RSX-11 or VMS, substitute the appropriate logical device. Refer to the MicroPower/Pascal Installation Guide for this information.

MicroPower/Pascal Program Fragment

```
%INCLUDE LB:IODEF.PAS
%INCLUDE LB:QDINC.PAS
%INCLUDE LB:YKINC.PAS
        .
        .
        .
VAR
    YK_REQ : YK_PORT_RQST ;                 { Request Packet for YK handler }
    YK_RPLY_DESC : SEMAPHORE_DESC ;         { Sem for YK reply - assumed
                                              to be already initialized }

FUNCTION PORT_A_DMA_INPUT  (
            YK_DMA_ADR : DMA$ADDRESS ;      { port address in DMA format }
            BUF_DMA_ADR : DMA$ADDRESS ;     { buffer address in DMA format }
            DATA_LENGTH : UNSIGNED          { length of data transfer }
                 ) ; DMA$BYTE_COUNT ;       { function returns number of
                                              bytes actually transferred }

BEGIN
WITH YK_REQ DO
    BEGIN
    oper := DMA_READ ;
    funct_mods := [] ;
    ind_mods := [ FI$SIMPLE_REPLY ] ;
    unit_num := PORT_A ;
    reply_sem := yk_rply_desc .id ;
    END ;

SEND ( NAME := '$YKA  ' ,
       VAL_DATA := YK_REQ ,
       VAL_LENGTH := SIZE (YK_PORT_RQST) ) ;

WAIT ( DESC := YK_RPLY_DESC ) ;

PORT_A_DMA_INPUT := $DMA_TRANSFER  ( SOURCE := YK_DMA_ADR ,
                                     DEST := BUF_DMA_ADR ,
                                     COUNT := DATA_LENGTH ,
                                     UNIT := 1 ) ;

YK_REQ . oper := DMA_COMPLETE ;

SEND ( NAME := '$YKA  ' ,
       VAL_DATA := YK_REQ ,
       VAL_LENGTH := SIZE (YK_PORT_RQST) ) ;

WAIT ( DESC := YK_RPLY_DESC ) ;

END ;
```

3.4.5.4.2 **Reading and Writing Data from/to Serial Line Ports** - You can use the DMA handler to read data from or write data to the serial-line ports SLU2 channel A and SLU2 channel B. This operation must be done by user-supplied software that sends requests to the DMA handler since the XL and XS handlers do not support the operation. You can read or write characters by using the request line hardware on either channel of SLU2.

3.4.5.5 **Accessing the Q-BUS I/O Page** - The QD handler can be used to access the Q-BUS I/O page, allowing I/O processing directly by the KXT11-C, but you cannot connect the Q-BUS DMA request line to the KXT11-C DMA device (which would notify the DMA device that the Q-BUS device is ready to be read or written). However, the QD handler can be used with a Q-BUS device if the device has a silo that will accept new data each time it is written/read.

3.4.5.6 **Assuring Access to a DMA Channel** - To assure that a DMA channel is available when you need it, you can dedicate access to either or both channels to a given process with the DMA_ALLOCATE command. This feature can be used when a transfer must occur within a specific time period following an event. You dedicate the DMA channel before the event so you are assured your transfer request will not be held up by another transfer in progress.

3.4.6 Using the KX (Arbiter-Resident Two-Port RAM) Handler

The KX handler performs the arbiter (master) portion of the KX/KK protocol described in Appendix A. It passes data between the arbiter CPU and up to 14 KXT11-C peripheral processors running on the Q-BUS. The handler communicates with the KXT11-C through the command and status registers of data channel Ø and data channel 1 of the two-port RAM (TPR). It operates in conjunction with the KK handler described in Section 3.4.7.

Each unit is associated with a unique interrupt to the handler to permit fast communication. The controller IDs, unit numbers, associated register base address, and default interrupt vectors for each system ID switch position are shown in Appendix C. The KX handler is in DRVU.OBJ and DRVM.OBJ, not DRVK.OBJ.

The run-time hardware support library (RHSLIB.OBJ) includes two routines, KX_WRITE_DATA and KX_READ_DATA, that can be used to send and receive data via the KX/KK handlers to and from the KXT11-C. These routines and associated data structures are defined by the %INCLUDE file KXINC.PAS.

Refer to Section 3.5 for examples of the use of the KX handler in peripheral processing applications.

3.4.7 Using the KK (KXT11-C-Resident Two-Port RAM) Handler

The KK handler performs the slave portion of the KX/KK protocol described in Appendix A, and operates in conjunction with the KX handler described in Section 3.4.6. The prefix file KKPFXK.PAS configures data channel Ø (4-byte data area) as KK unit Ø and data channel 1 (12-byte data area) as KK unit 1.

The run-time hardware support library (RHSLIB.OBJ) includes two device handler interface routines, KK_WRITE_DATA and KK_READ_DATA, that can be used to send and receive data via the KK/KX handlers to and from the arbiter. These routines and associated data structures are defined by the %INCLUDE file KKINC.PAS.

Refer to Section 3.5 for examples of the use of the KK handler in peripheral processing applications.

3.5 PROGRAMMING THE KXT11-C PERIPHERAL PROCESSOR INTERFACE

MicroPower/Pascal provides the following tools to assist you in programming your peripheral processing application.

- KX and KK device handlers. They provide a two-channel communication path from the arbiter to the KXT11-C and can be used to pass data or commands to and from the KXT11-C under control of the arbiter program.

- QD device handler. It provides a two-channel DMA path that can move interfacing data at a high rate without processor intervention.

- Device handler interface routines and associated %INCLUDE files that simplify the use of the KK, KX and QD device handlers.

3.5.1 Interface Considerations

A peripheral processing application consists of several separate processes that control the peripheral processor and send, receive, and process data associated with it. Some of these processes reside in the arbiter processor and some reside in one or more KXT11-C peripheral processors. MicroPower/Pascal provides low-level interconnections between these processes in the form of semaphores and ring buffers for communication between processes running on a single processor, and KX, KK, and QD handlers for communication between processes running on separate processors. An application using the handlers for communication must contain the following additional protocols or interfacing.

- Commands Necessary to Control the KXT11-C Application -- If you need control information to start, stop, or interrupt the KXT11-C application from the arbiter, or if you need to differentiate between report information and alarm information in messages received from the KXT11-C, you can encode the commands in a message format. For example, you can assign commands as numbers in the first byte of the message followed by five bytes containing the address of the buffer to copy the data into and two bytes containing the length of the buffer.

- Message Passing Mechanisms -- Large blocks of data or frequently sent data are more efficiently passed with the DTC. To initiate transfer of the data you can pass the address and length of the buffer in a command message sent by the KX/KK handlers, then instruct the QD handler to transfer it. Alternatively you can choose a fixed buffer location and a flag location to test (using the DTC) to determine when to write the data. The application in the KXT11-C must inform the arbiter application each time data is written or read. A message through the TPR is the most likely method.

  For shorter or infrequently issued messages, you can send just the data as a message using the KX/KK handlers.

- Arbiter Management of Errors from the KXT11-C Application -- An error message should inform the arbiter application of error or alarm conditions, and appropriate procedures should deal with them. Such management is required when the KXT11-C application cannot access some memory location using the DTC.

- Protection of Shared Memory Areas -- Shared memory areas in the Q-BUS system or the KXT11-C should be protected from simultaneous access. Semaphores will suffice for some areas, but others (for example, buffers that can be used and modified by KXT11-C and arbiter applications) will require semaphores and control messages.

If you have two processes in one KXT11-C using the KX/KK communication path, you can dedicate one channel to each process. The second channel uses more of the two-port RAM (six words at a time instead of two) to pass messages and thus is faster for messages greater than two words in length.

If you have more than two processes contending for use of the KX/KK communication path, you must write two message switching processes that will add a destination address to the message to select the receiving process on the receiving end, with one switching process at each end of the path (the KXT11-C end and the arbiter end). Each switching process should decode the address, strip it from the message, and pass the message on to the receiving process.

3.5.2 KK/KX Interface Example

An example peripheral processor application is in the MicroPower/Pascal kit. Its installation and operation is documented in Appendix A of the MicroPower/Pascal Installation Guide. The following KK/KX interface example draws from this example application and uses sections of the MicroPower/Pascal source files IOPVFY.PAS and ARBVFY.PAS.

The example peripheral processor application is a simulation of an ideal ball bouncing within an ideal rectangular box. It is a peripheral processor application because half of the box is managed by the arbiter program and the other half is managed by the KXT11-C. As the ball leaves the right edge of the screen in the arbiter controlled half, it appears at the left edge of the screen in the KXT11-C controlled half.

The ball is modeled in the programs as a vector containing its current position and velocity. This is shown by the following MicroPower/Pascal data structures.

```
col = Ø..79;
row = Ø..23;

vector = record
  x,y : real;                { X and Y components }
end;

screen_pos = record
  c : col;
  r : row;
end;

Ball = record
  op : vector;               { Old position }
  p : vector;                { Position }
  v : vector;                { Velocity }
  sp : screen_pos;           { Position in screen units }
  osp : screen_pos;          { Old position in screen units }
end;
```

The old position and the screen positions are included for ease of moving the ball around the screen.

When the ball is passed from one system to another the current ball record must be transmitted. This is done with a message like this:

```
message = record
  case m_type : char of      { Message type }
    "b": (b:Ball);
end;
```

The record field m_type distinguishes between different types of messages, of which only one is defined, the m_type "b" for Ball. This field can be used to define "i" for Initialize, "a" for Alarm, and so on.

Now, it is a simple matter of repeatedly moving the ball one v (velocity) from p (the current position) during each time unit and checking to see if it hit a wall or got passed to the other side, as follows.

```
Fly(b);
if (b.p.x < LEFT) then Pass(b);
if (b.p.x > RIGHT) then Bounce(b.p.x,RIGHT,b.v.x);
if (b.p.y < TOP) then Bounce(b.p.y,TOP,b.v.y);
if (b.p.y > BOTTOM) then Bounce(b.p.y,BOTTOM,b.v.y);
Show(b);
```

Here, Fly moves the ball one v from p, Bounce performs whatever is needed to bounce the ball off one of the walls, Show displays the ball on the screen, and Pass takes the current ball vector and passes it to the other side. In this application, once the ball is passed to the other side, there is nothing else for this side to do but wait for the ball to come bouncing back again, and so Pass, shown below, just waits for this to happen. (Your application may very well have other processes performing other tasks.)

```
procedure Pass(var b:Ball);
var
  m : message;
  garbage : io$status;
  garbage2 : unsigned;
begin
  m.m_type := "b";
  m.b := b;
  garbage := KK_write_data(m,size(m),garbage2);
  garbage := KK_read_data(m,size(m),garbage2);
  b := m.b;
end;
```

3.5.3 Determining Physical Addresses

When using the QD handler to move data blocks to and from arbiter memory, you should provide a physical address for a buffer in the arbiter memory. This is done by declaring a process as handler-mapped and using the higher order 3 bits of the buffer address as an index into the memory management unit (MMU) registers. Using the PAR (page address register) of the MMU and the buffer address, you can calculate a physical address.

3.6 DEBUGGING A KXT11-C APPLICATION

This section describes the aspects of program debugging that are unique to KXT11-C applications.

3.6.1 Setting up PASDBG

The following list contains general requirements for setting up the KXT11-C for use with the PASDBG debugger.

1. Configure memory as follows.

   For RAM systems, configure memory just as it will be in the final application (Section 3.3.1.1).

   For ROM systems, configure memory just as it will be in the final application (Section 3.3.1.1, steps 1 to 4) but install the equivalent-size RAM in the user sockets in place of ROM. Set the memory map selection jumpers appropriately for those RAMs (refer to the KXT11-CA User's Guide hardware reference manual).

   When you specify the MEMORY macro parameter TYPE=ROM in the kernel configuration file, the application's pure code and pure data is built into the ROM (or what will eventually be ROM) address space. PASDBG can then load the application into the KXT11-C because the ROM has been replaced with RAM. Keeping the ROM portions of the application separate from the RAM portions helps track the size of each while the program is growing and facilitates your use of a logic analyzer to detect corrupt areas of ROM.

2. Build the debugger service module (DSM) into your application by specifying DEBUG = YES in the kernel configuration file.

3. Build in debugging symbols by using the /D option in the MERGE, RELOC and MIB steps of the application build process. This is performed automatically by the MPBLD procedure when you build a configuration that includes debugging.

4. Set the boot/self-test switch to position 4.

5. Set the baud for SLU1 using the baud jumpers on the KXT11-C circuit board. Do not use the autobaud feature.

6. Install the SLU1 break-enable jumper on the KXT11-C circuit board.

7. Once PASDBG and the target KXT11-C are running and before loading and running your application, use PASDBG in ODT mode to open location 175002. Then set bit 4 (enter console ODT on halt) to 1. This prevents the KXT11-C from trapping through vector 10 if a HLT instruction is executed.

3.6.2 Test System Configuration

You should develop a program specifically to test the operation of your peripheral processor application before integrating with the arbiter section of the application. The test program can be run under RSX-11, RT-11 or MicroPower/Pascal. If you develop the arbiter test

program under RT-11 or RSX-11, you can use all the program development and testing aids that are available with these operating systems to facilitate running your test program and examining the results (for example, interactive terminal interface, indirect command files, file management and compare utilities, and so on). The test program should pass commands to the application in the KXT11-C and verify that the correct results are returned.

3.6.3 Debugging with an RT-11 or RSX-11 Arbiter Test Program

Develop a test program that runs under RT-11 or RSX-11. This program should pass appropriate commands and messages to the KXT11-C application and verify the results that are returned.

Perform the following steps.

1. Connect the host to the KXT11-C console port in the same way you connect a stand-alone system.

2. Load the application into the KXT11-C with PASDBG's LOAD command, then start execution with the GO command.

3. Start your arbiter test program and begin testing.

You can use PASDBG to set breakpoints and watchpoints as in normal debugging operations. You can also use the KK handler debugging features described in Section 3.6.6. You can use the KUI utility program's TRAP command to test trap handling routines in the KXT11-C application.

3.6.4 Debugging with a MicroPower/Pascal Arbiter-Resident Test Program

To debug a KXT11-C application using a MicroPower/Pascal arbiter-resident test program, you use PASDBG to interact with the test program in the arbiter and the application program in the KXT11-C. The methods for accomplishing this are the same as those described in Section 3.6.5 for debugging multiple-processor MicroPower/Pascal applications.

3.6.5 Debugging Multiple-Processor MicroPower/Pascal Applications Simultaneously

If you want to access more than one processor with PASDBG to determine the cause of a malfunction in a peripheral processing application, you can use one of two methods.

One method has a separate PASDBG host system process and terminal for each processor in the target system. This is the most convenient way of debugging in the VMS and RSX-11 host environments. The other method, using a single PASDBG host system process, switches the communication line that PASDBG uses from processor to processor as required. With either method you attach a serial line from the host to the console port of each target system.

To switch back and forth between systems with PASDBG, you exit PASDBG, select the serial line connected to the desired processor, and restart PASDBG. You can simplify this by creating indirect command files or user-defined commands.

3.6.5.1 **Setting up PASDBG as a Single Host Process** - To transfer control back and forth between processors, you should create indirect command files or user-defined DCL commands that will load the appropriate handler and start PASDBG. Examples of command files to configure PASDBG in RT-11, RSX-11 or VMS are given in the following examples.

RT-11 Host Systems

```
.UNLOAD TD
.SET TD CSR aaaaaa
.SET TD VECTOR vvv
.LOAD TD
```

RSX-11 Host Systems

```
>DEA TD:
>ALL TTnn:=TD
```

VMS Host Systems

```
$DEFINE TD:  TTcn:
```

The peripheral processor verification procedure in the MicroPower/Pascal kit includes a command file, PPVFY, which generates two other command files that switch PASDBG between the arbiter and KXT11-C (refer to the applicable MicroPower/Pascal installation guide). Using these files as models, you can create your own command files to switch PASDBG between multiple KXT11-C processors.

3.6.5.2 **Loading and Starting Target Processors** - You must use the PASDBG LOAD command to load each target processor so that the DSM is properly initialized. You cannot use the KXT_LOAD procedure or the KUI utility program to load a KXT11-C from the arbiter processor if you wish to debug it with PASDBG.

Once a target is loaded, you can set breakpoints or watchpoints (Section 3.6.5.3). Thereafter, you can use PASDBG's GO/EXIT command to start the target application. The command starts target execution, then returns control to the host system monitor. You can then connect to another target processor (using an indirect command file) prior to loading and executing the next target application. If you are debugging with multiple host PASDBG processes, use the PASDBG GO command to initiate target execution and continue running with PASDBG.

3.6.5.3 **Setting Breakpoints and Watchpoints** - You can set breakpoints and watchpoints in a system that has several processors by following the steps listed below.

1. Connect to the first processor (see Section 3.6.5).
2. Load symbols (LOAD/SYM command).
3. Set the breakpoints and/or watchpoints.
4. Start up the target application and exit PASDBG with a GO/EXIT command.

The processor is now running with the breakpoint(s) and/or watchpoint(s) set. You can connect to other processors and set other breakpoints and/or watchpoints if desired. You should start all KXT11-C processors before starting the arbiter processor. If your arbiter processor is running a MicroPower/Pascal application, you can start it with PASDBG. If your arbiter is running under RSX-11 or RT-11, you start your application using the appropriate system-specific command from the console terminal.

PASDBG does not save the /AFTER and /PROCESS breakpoint and watchpoint modifiers if control is transferred to another processor. On reconnection to a processor, all breakpoints and watchpoints have modifiers of /AFTER=1 and /PROCESS=ALL.

On reconnection to a processor, if a breakpoint or watchpoint has been triggered since the disconnect, the message UNKNOWN BREAKPOINT AT nnnnnn is displayed at the terminal. To find the symbolic location, load the symbols (LOAD/SYM file.ext) and execute a SHOW TARGET command. The SHOW TARGET will symbolically display the location at which the target processor halted (the breakpoint location). The other processors in the system will continue to execute unless they are waiting for something from the processor that is halted at the breakpoint.

3.6.6 Using the KK Handler Debugging Locations

The KK handler uses two kernel locations, $KXTQW and $KXTQR, which are called (JSR PC instruction) just after the arbiter or KXT11-C places a message in the TPR command register. By setting breakpoints at either or both locations, you can examine the communication sequence between the arbiter and a KXT11-C processor on a step-by-step basis.

3.6.7 Debugging an Application in a KXT11-C with Battery Backup

PASDBG's GO and INIT/RESTART commands start your application directly in the kernel's power-up module, bypassing the native firmware that sets up the power-up flags. Because of this, you must manually set up the flags before issuing these commands.

The following procedures apply when you are using PASDBG in a KXT11-C system configured for battery backup (: POWER = NONVOL specified in KXT11C macro).

To down-line load an application:

1. Use PASDBG in ODT mode to open location 175ØØ2.
2. Set bit 9 (power up without battery backup) to 1 and bit 1Ø (power up with battery backup) to Ø.
3. Load your application with the LOAD command.

To restart a loaded application:

1. Use PASDBG in ODT mode to open location 175ØØ2.
2. Set bit 9 (power up without battery backup) to 1 and bit 1Ø (power up with battery backup) to Ø.
3. Issue the INIT/RESTART command.

To test the power-failure and restart sequence:

1. Use PASDBG in ODT mode to open location 175ØØ2.
2. Set bit 1Ø (power up with battery backup) to 1 and bit 9 (power up without battery backup) to Ø.
3. Set kernel location KXT$PF to a nonzero value.
4. Issue the INIT/RESTART command.

If you attempt to load or restart an application configured for battery backup, without first setting one of the flags, the target will crash. If that happens, enter ODT mode and follow the instructions above to repeat the down-line load.

CHAPTER 4

THE KXT11-C AND YOUR RSX-11 OR RT-11 ARBITER APPLICATION

This chapter describes KXT11-C peripheral processor application development for target systems that use an arbiter application in an RT-11 or RSX-11 system environment. For these target systems, there are three types of application development software:

- MicroPower/Pascal software for programming the KXT11-C
- RT-11 or RSX-11 software to operate on the arbiter system
- KXT11-C/RT-11 Peripheral Processor Tool Kit or KXT11-C/RSX-11 Peripheral Processor Tool Kit

Each tool kit provides a KX device handler and the KUI program for peripheral processor loading and control.

MicroPower/Pascal KXT11-C software makes the KXT11-C look (to an arbiter system) like a traditional Q-BUS I/O device. The KX handler (MicroPower/Pascal, RSX-11 and RT-11 versions) treats the two-port RAM (and the KK handler on the KXT11-C) as a standard I/O device, and the KXT11-C is configured into a system just like any other device. In addition, the DMA transfer controller (QD) device handler gives you a high-speed data path that is similar to other Q-BUS DMA devices.

The design considerations identified in Section 3.5 apply to arbiter applications that use RT-11 and RSX-11. The major differences between these arbiter applications and MicroPower/Pascal arbiter applications are the implementation language and the lack of MicroPower/Pascal features such as semaphores and run-time hardware support routines.

4.1 KUI UTILITY COMMANDS

The KUI program provides the following peripheral processing development support and final application support.

- SET and SHOW commands to allocate a peripheral processor and display its status.
- Indirect file commands @, CLOSE, SUSPEND, and RESUME to manage the use of command file input.
- LOG command to write a file containing the commands entered during a KUI session. The log file is executable as a command file.

- LOAD command to transfer executable images from the arbiter's mass storage device to a KXT11-C.
- EXECUTE command to start the program on the KXT11-C.
- REINIT command to cause the KXT11-C native firmware to execute a bootstrap procedure.
- ODT command to emulate the console ODT of the KXT11-C native firmware.
- TRAP command to cause the execution of a trap handling routine on the KXT11-C.
- SELFTEST command to selectively invoke the self-test routines in the KXT11-C native firmware.
- EXIT command to terminate execution of the KUI program and return to the system's command line interpreter.

4.2 ARBITER APPLICATIONS THAT USE RT-11

4.2.1 Interface Tools

The KX handler supplied with the KXT11-C/RT-11 Peripheral Processor Tool Kit provides the primary interface between application code running under the RT-11 system and application code running on the KXT11-C. The KX handler supports up to four KXT11-Cs where each KXT11-C provides two logical units. Logical unit numbering is determined when you build the handler or by subsequent SET KX CSR (DCL) commands.

To support more than four KXT11-C processors, you must edit, rename, and rebuild the KX handler. The procedure for creating a second or subsequent handler appears in the tool kit documentation.

The KX handler for RT-11 provides .OPEN, .CLOSE, .READ, and .WRITE programmed requests.

- The .OPEN programmed request associates a user-specified channel number with a logical unit number of a KXT11-C.
- The .CLOSE programmed request reverses the effect of the .OPEN programmed request, thus freeing the user's channel for use with another device or file.
- The .READ programmed request transfers data from a peripheral processor to an arbiter buffer with three options: .READ, .READW and .READC.
- The .WRITE programmed request transfers data from an arbiter buffer to a peripheral processor with three options: .WRITE, .WRITW and .WRITC.

The RT-11 arbiter application may also interface the application on the KXT11-C using the KXT11-C's DMA transfer controller (DTC) and QD handler. This interface provides for the exchange of buffers of data between the arbiter and the KXT11-C using direct memory access.

The tool kit also provides the MACRO-11 macro KXTDF$ that defines the registers of the KXT11-C and the commands and responses defined for the KX/KK protocol in Appendix A. This macro can be used when necessary to interface the KXT11-C directly without using the KX handler, or if the application requires access to the command and status registers used by the native firmware on the KXT11-C.

In developing a peripheral processing application using an RT-11 arbiter, you must follow the same steps that were outlined in Chapters 2 and 3. The following are specific considerations that reflect the RT-11 operating system environment.

4.2.2 Calculating Physical Memory Addresses

If you want to use DMA transfers between the RT-11 arbiter's memory and the KXT11-C, you must provide a physical address to the QD handler on the KXT11-C. The RT-11 KX handler provides a special function (.SPFUN) that converts a 16-bit virtual address to a 22-bit physical address.

In the RT-11 SJ and FB monitors, virtual addresses and physical addresses are identical. To facilitate the transportability of programs among all the RT-11 monitors (SJ, FB, and XM), you should use the .SPFUN function in the SJ and FB environments, as well as in the XM environment.

4.2.3 Application Building for Debugging

The same overall considerations apply to debugging an application using an RT-11 arbiter as apply to the MicroPower/Pascal arbiter, as outlined in Chapters 2 and 3. Conventional debugging tools, such as ODT, are available for debugging the arbiter-resident portion of the application. KUI may also be used during the debugging phase of development. The ODT and TRAP commands are particularly useful when debugging. It is helpful to include PASDBG in the KXT11-C application so you can use the two arbiter/KXT11-C debugging locations, $KXTQW and $KXTQR, described in Section 3.6.6.

4.2.4 Final RT-11 Application Configuration

Final application configuration considerations include removing the debugging features, adapting the application to its final loading method, and testing it in its final configuration.

4.2.5 Loading a KXT11-C Peripheral Processor from the RT-11 Arbiter

The KXT11-C application in its final configuration can be loaded using the KUI program, loaded from TU58 DECtape II, or resident in ROM. KUI supports the loading of .SAV, .LDA and .MIM files on RT-11. Loading and starting the application with KUI can be automated by using command files at system start-up. Appendix F describes how a program can detect nonfatal errors reported when issuing the KUI commands REINIT or LOAD to the KXT11-C.

4.3 ARBITER APPLICATIONS THAT USE RSX-11

4.3.1 Interface Tools

The KX handler supplied with the KXT11-C/RSX-11 Peripheral Processor Tool Kit provides the primary interface to the KXT11-C. The KX device (handler) has one unit number associated with each data channel in each KXT11-C two-port RAM area. These units are assigned to their respective data channels at the time the KX handler is generated. This handler is a conventional RSX-11 handler providing the standard requests IO.RVB, IO.WVB, IO.ATT, and IO.DET.

Another interface is through shared memory areas using the DTC and the QD handler on the KXT11-C. This interface is faster than the KX handler, especially for larger data blocks, but is more complex to use and requires special precautions.

The tool kit also provides the MACRO-11 macro KXTDF$ that defines the registers of the KXT11-C and the commands and responses defined for the KX/KK protocol in Appendix A. This macro can be used when necessary to interface the KXT11-C directly without using the KX handler, or if the application requires access to the command and status registers used by the native firmware on the KXT11-C.

In developing a peripheral processing application using an RSX-11 arbiter, you must follow the same steps that were outlined in Chapters 2 and 3. The following are specific considerations that reflect the RSX-11 operating system environment.

4.3.2 Calculating a Physical Memory Address

You must calculate a physical address when transferring memory blocks to and from the arbiter memory space with the KXT11-C using the QD handler. Do this by using the get region parameters directive GREG$ from the task containing the buffer. The GREG$ directive returns the physical address of the beginning of the task region. Use this address plus the offset of the buffer from the beginning of the task to calculate the physical address of the buffer. This address can then be passed to the QD handler on the KXT11-C.

4.3.3 Accessing Shared Memory Areas

You must take precautions when using a shared memory area or a buffer that is accessed by the QD handler and DTC device. In RSX-11 systems the shuffler task can move a task from place to place in memory, so a task issuing a request to a KXT11-C can be moved between the time the request is issued and the time the KXT11-C attempts to move the data, leading to catastrophic results.

You can lock a task in position by queuing an I/O request which increments the task's I/O-in-progress count; RSX-11 will not shuffle a task with I/O in progress. If, however, your application uses the KX handler to pass a buffer transfer request to the QD handler, the KX handler request will complete, freeing the task to be swapped before the QD handler can move the data. This can lead to unpredictable and random failures of the application.

To prevent this problem, manually change the I/O-in-progress count to prevent task shuffling during critical periods, or use data partitions for the shared data areas.

4.3.4 Protecting Shared Data Areas from Simultaneous Access

RSX-11 does not provide semaphore operations, so other means of protecting data areas from simultaneous access must be devised. You should design this protection into the protocol used in communicating with the KXT11-C.

4.3.5 Application Building for Debugging

You will probably include ODT or some other debugging tool in your RSX-11 application while debugging it. You can use the KUI program during the debugging phase of development. The ODT and TRAP commands are particularly useful when debugging. It is helpful to include PASDBG in the KXT11-C application so you can use the two arbiter/KXT11-C debugging locations, $KXTQW and $KXTQR, described in Section 3.6.6.

4.3.6 Final RSX-11 Application Configuration

Final application configuration considerations include removing the debugging features, adapting the application to its final loading method, and testing it in its final configuration.

4.3.7 Loading a KXT11-C Peripheral Processor from the RSX-11 Arbiter

The KXT11-C application in its final configuration can be loaded using the KUI program, or resident in ROM. KUI can load memory image files with the .TSK and .MIM file types. Loading and starting the application with KUI can be automated by using command files at system start-up. Appendix F describes how a program can detect nonfatal errors reported when issuing the KUI commands REINIT or LOAD to the KXT11-C.

APPENDIX A

KX/KK DEVICE HANDLER COMMUNICATION PROTOCOL

This appendix describes the protocol that the KK and KX device handlers use to communicate with one another through the two-port RAM (TPR). It contains information to assist you in designing your own KK or KX device handler so it can communicate with the DIGITAL-supplied KK or KX handler. The KK and KX handlers are used when the KXT11-C is set up for peripheral processor operation.

The protocol provides a master-slave relationship between the arbiter processor and the KXT11-C (do not confuse master-slave with the bus-master/bus-slave hardware concept). The KK handler running on the KXT11-C uses the TPR to emulate a traditional Q-BUS peripheral device. The KX handler running on the arbiter communicates with this device. The protocol implements a request-reply dialog between the arbiter on the Q-BUS and the KXT11-C to assure correct and complete transfer of data between them.

The arbiter is the master in all transactions with the peripheral processor which is the slave (refer to Figure A-1). The peripheral processor must receive a command from the arbiter before it passes any data to the arbiter or before it receives any data from the arbiter.

A.1 COMMUNICATION MECHANISMS

The basic TPR hardware communication mechanisms are the:

- Command register for each data channel -- used by the master to pass commands to the slave
- Status register for each data channel -- used by the slave to pass error and operational status to the master
- Data registers -- 4 bytes for data channel Ø and 12 bytes for data channel 1, used for passing data between the master and the slave
- QIR register in the slave -- used by the slave to interrupt the master

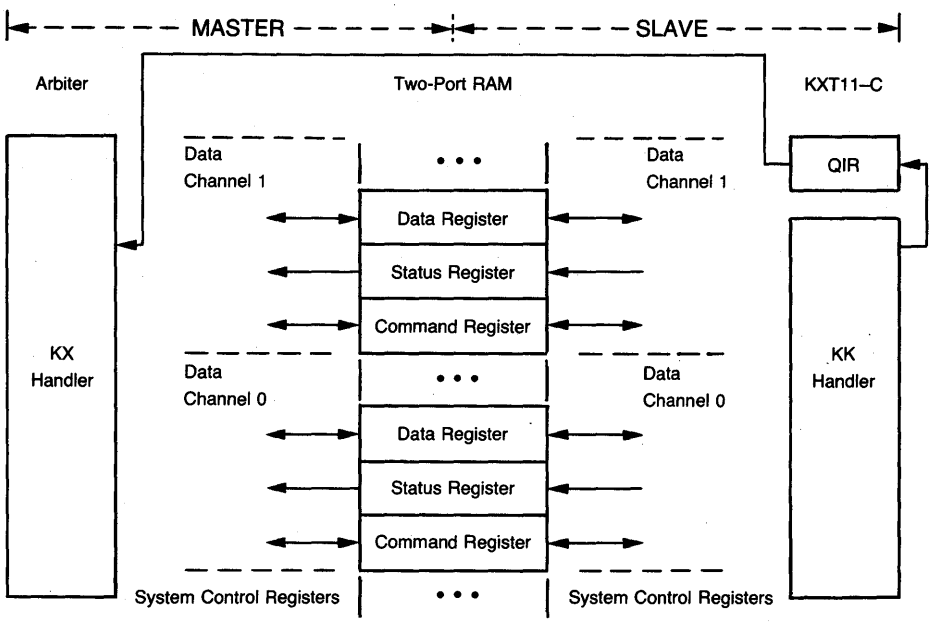


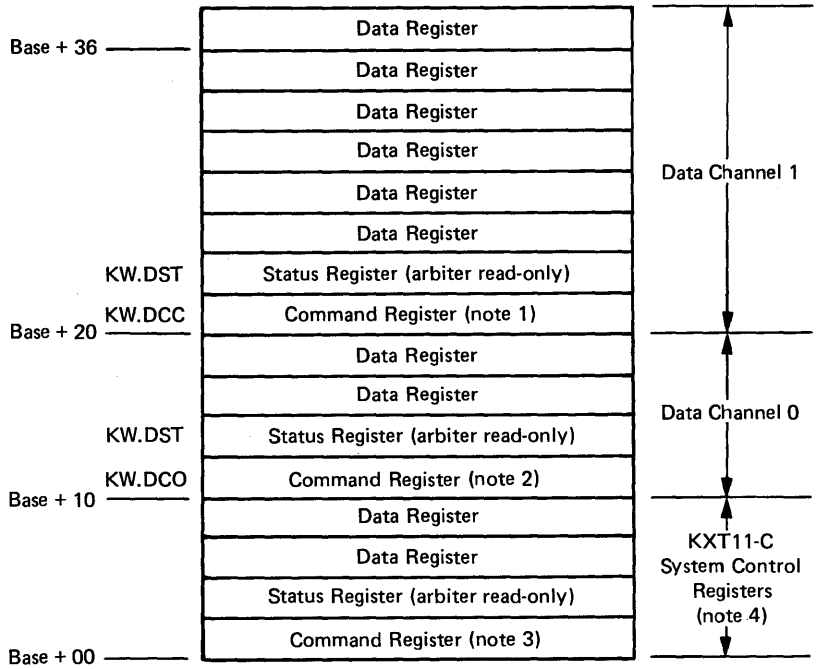
Figure A-1: **KX/KK Device Handler Communication Linkages**

The interface between the arbiter and the KXT11-C consists of layers of software. The lowest layer contains MACRO definitions for the bits in the command and status registers of the TPR. The next layer consists of KX and KK handlers that move data between a KXT11-C process and a process in the arbiter.

The protocol provides the arbiter with commands that cause the:

- Device initialization of the peripheral processor
- Arbiter read request/peripheral processor write reply sequence
- Arbiter write request/peripheral processor read reply sequence
- Enabling and disabling interrupts from the peripheral processor to the arbiter

The following sections describe the special meanings the protocol assigns to the TPR registers in the context of KX/KK handler operations. Figure A-2 shows the TPR's general layout.



Note 1 Writing to this register from the arbiter causes a level 5 interrupt through vector 124 on the KXT11-C.

Note 2 Writing to this register from the arbiter causes a level 5 interrupt through vector 120 on the KXT11-C.

Note 3 Writing to this register from the arbiter causes the KXT11-C to restart the native firmware.

Note 4 The system control registers are not part of the protocol. They are used by the KUI utility program, the KXT_LOAD procedure, diagnostic programs, and user-written programs.

Figure A-2: **TPR Register Layout**

A.2 KX/KK PROTOCOL DEFINITION

In the protocol a data channel's command register (KW.DCO in Figure A-2) controls ownership of the contents of the data channel's data register. If the command register is 0, the KX handler owns the data channel's registers. If the command register is nonzero, the KK handler owns them. All status registers (KW.DST in Figure A-2) are owned by the KK handler.

When the KX handler communicates with the KK handler with interrupts disabled, it must poll the command register (KW.DCO) using it as the ownership flag for the data channel. To poll the KK handler, the KX handler must:

1. Poll the command register until it becomes zero. The zero condition means the KK handler is idle and any previous command has completed.

2. Issue a command by writing it into the command register. This causes the KK handler to perform the command. Once the command register has been written, the KX handler cannot alter the contents of any of the data channel's registers. Consequently the KX handler must write the data being transferred by the command into the data registers before the command is issued.

3. Poll the command register until it becomes zero. At that time, the status register data is valid and the KX handler can issue further commands (as in step 2).

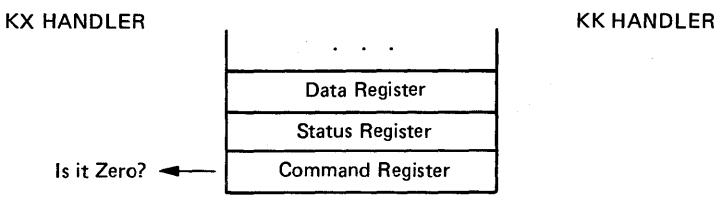
When the KX handler communicates with the KK handler with interrupts enabled, the KK device handler uses the Q-BUS interrupt register (QIR) to signal the KX handler that an operation is complete. The KK handler interrupts the KX handler after a command has completed and the proper status and/or data bits have been set.

By using the interrupt-on-data-available (KC.IDA) and interrupt-on-data-requested (KC.IDR) bits of the enable interrupts command, the KX handler can instruct the KK handler to interrupt the KX handler when the KS.DA and/or KS.DR status register bits change from Ø to 1. This presents a device-like interface analogous to a physical device's receive-buffer-full and transmit-buffer-empty interrupts.

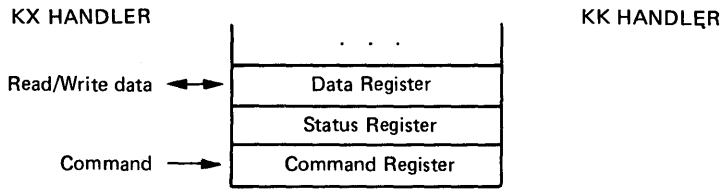
A.2.1 KX and KK Handler Transactions

The transactions between the KX handler and the KK handler are illustrated below.

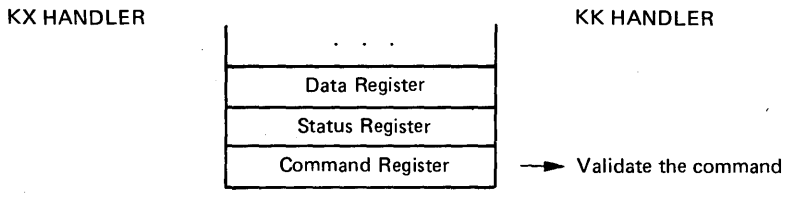
1. The KX handler tests the command register. If it is zero, the handler proceeds to the next step. If nonzero, the handler repeats this step.



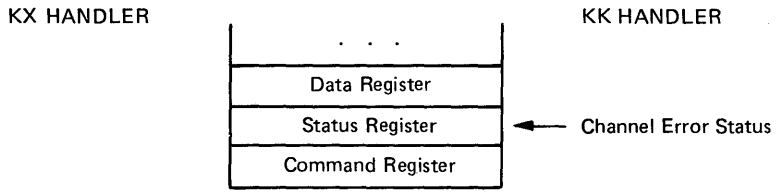
2. Since the command register is zero, the KX handler owns the data channel. It can write data to or read data from the data registers, as implied by the pending command. The KX handler issues the command by writing it to the command register. The act of writing the command to the command register, thereby making it nonzero, switches ownership of the data channel to the KK handler.



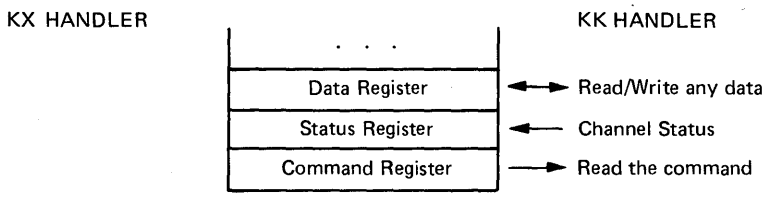
3. The KK handler is interrupted by the KX handler writing to the command register. The KK handler now owns the data channel. It reads and validates the command.



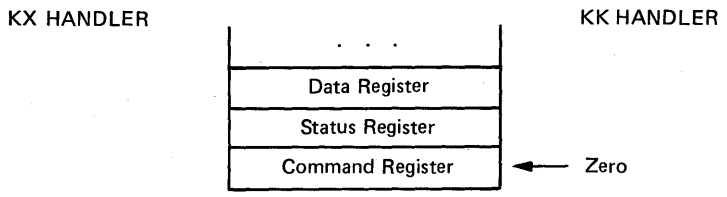
If the KK handler detects an error in the command, it reports this error in the status register and proceeds with step 4.



If it detects no error, the KK handler performs the command, moves any data required by the command into or out of the data registers, and writes the status of the channel into the status register.



4. The KK handler completes the transaction by zeroing the command register, thus transferring ownership of the data channel back to the KX handler. If interrupts are enabled, the KK handler interrupts the KX handler by queuing an interrupt to the Q-BUS interrupt register (QIR).



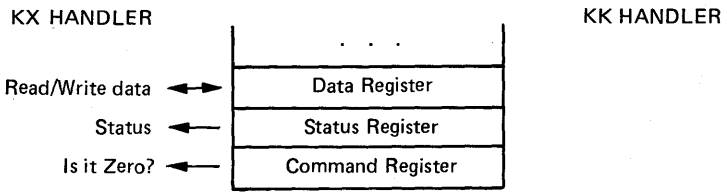
5. Finally, the KX handler regains ownership of the data channel by:

   Polling the command register and testing for zero

   or

   Waiting for an interrupt and testing the command register on being interrupted to find it is zero

   Once it gains ownership, the KX handler checks the status register for error and status information and reads from or writes to the data registers as implied by the pending command. From this point the cycle repeats.



A.2.2 Message Communication Between the KK and KX Handlers

In the transactions between the KX and KK handlers over the TPR (Section A.2.1), the number of bytes in a data transfer is limited by the number of data registers in the channel (4 bytes for data channel Ø and 12 bytes for data channel 1). At the application-program handler level of communication, however, the protocol provides for longer messages by using an end-of-message (EOM) indicator. Thus, a read or write request to the KX handler causes multiple transactions over the TPR if the message is larger than the size of the data channel.

When the KX handler receives a write request from the arbiter program, it performs as many TPR write operations as necessary to send the message. For each TPR write operation, the KK handler completes the transaction by performing a corresponding TPR read operation. On the last write operation, the KX handler sets the EOM indicator in the data channel's status register. This informs the KK handler that all data has been sent. Therefore, the arbiter program's write request is complete.

When the KX handler receives a read request from the arbiter program, it performs as many TPR read operations as necessary to receive the message. For each TPR read operation, the KK handler completes the transaction by performing a corresponding TPR write operation. On the last TPR write operation, the KK handler sets the EOM indicator in the

data channel's status register. This informs the KX handler that all data has been sent. Therefore the program's read request is complete. Accordingly, the number of bytes received can be less than the number of bytes specified in the arbiter program's read request.

A.2.3 Synchronizing KK and KX Device Handler Operation

For a TPR read operation by the KX handler to complete, there must be a corresponding TPR write operation by the KX handler. Similarly, a TPR write operation by the KX handler requires a TPR read operation by the KK handler. If the KX handler issues a TPR read operation and the KK handler has not posted a TPR write operation, the KX handler receives an error indicating no data is available.

To optimize synchronization of TPR operations, the KX and KK handlers use two interrupts: interrupt when data available and interrupt when data requested (Sections A.3.1.2 and A.3.1.3). By using these interrupts the handlers need not continually issue and retry commands because of data-not-requested or data-not-available error conditions.

A.2.3.1 **Interrupting When Data is Available** - The KX handler uses the interrupt-when-data-available as follows.

1. Before issuing a TPR read command it checks for TPR data available.

2. If data is available, it issues the read command and returns, waiting for an interrupt on completion.

   If data is not available, it does not issue the read command. Instead, it saves its context and returns, waiting for an interrupt on data available.

When the KK handler issues a TPR write command, it sets the data-available indicator and requests an interrupt. When the KX handler receives the interrupt, it checks that it has a TPR read command pending and that data is available before issuing its TPR read command.

The KK handler can request an interrupt-when-data-available when it has no read pending. The KX handler ignores these interrupts.

A.2.3.2 **Interrupting When Data is Requested** - The KX handler uses the interrupt-when-data-requested as follows.

1. Before issuing a TPR write command it checks for TPR data requested.

2. If data is requested, it issues the write command and returns, waiting for an interrupt on completion.

   If data is not requested, it does not issue the write command. Instead, it saves its context and returns, waiting for an interrupt on data requested.

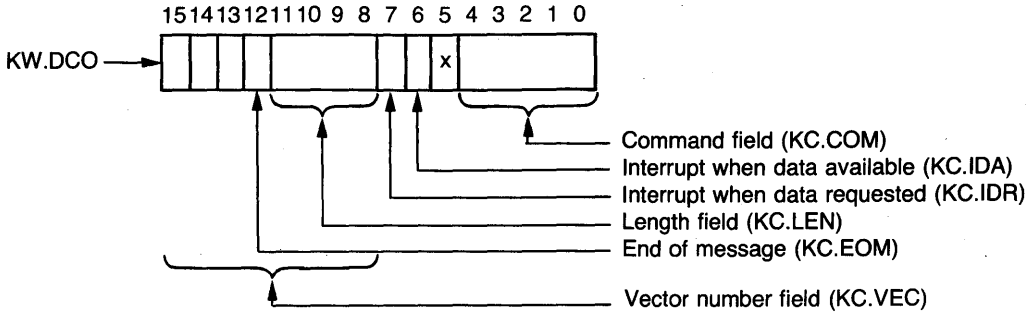
When the KK handler issues a TPR read command, it sets the data-requested indicator and requests an interrupt. When the KX handler receives the interrupt, it checks that it has a TPR write command pending and that data is requested before issuing its TPR write command.

The KK handler can request an interrupt-when-data-requested when it has no write pending. The KX handler ignores these interrupts.

A.3 REGISTER DEFINITIONS

The following sections define the command and status registers as viewed from the Q-BUS (KX handler) side of the interface between the Q-BUS and the KXT11-C.

A.3.1 Command Register Definition



Bit 5 is set to Ø. Bits KC.IDA, KC.IDR, KC.LEN, KC.EOM, and KC.VEC have meaning for specific commands only.

A.3.1.1 **Command Field (KC.COM)** - The command field, KC.COM, contains one of the following commands, issued to the KK handler from the KX handler. The codes are designed to allow your program to index them from a table.

No-Op Command KC$NOP (Code Ø)

A null operation. The KK handler places the data channel's status in the status register (KW.DST) and clears the command register (KW.DCO). If interrupts are enabled (KC$EI command) a Q-BUS interrupt occurs.

Reset KK Handler to KX Handler Command KC$RSM (Code 2)

Resets ownership of the TPR to the KX handler. The KK handler reports the channel's status in the status register (KW.DST) and clears the command register (KW.DCO). If interrupts are enabled (KC$EI command) a Q-BUS interrupt occurs.

Enable Interrupt Command KC$EI (Code 4)

Enables interrupt mode in the KK handler. The address of the vector to use must be in the KC.VEC field (bits 8 to 15) of the command register and must be the vector address divided by four.

The KK handler interrupts the KX handler under any of these conditions.

1. Command completion
2. If the KS.DA bit in the status register changes from Ø to 1 when the interrupt-when-data-available bit (KC.IDA) is set
3. If the KS.DR bit in the status register changes from Ø to 1 when the interrupt-when-data-requested bit (KC.IDR) is set

The KK handler interrupts the KX handler on command completion when the interrupt mode is enabled. Conditions 2 and 3 (above) cause the KK handler to interrupt the KX handler only if the KC.IDA and KC.IDR bits are set when the command is issued (see Sections A.3.1.2 and A.3.1.3). Before interrupting the KX handler, the KK handler places the channel's status in the status register and clears the command register.

Disable Interrupt Command KC$DI (Code 6)

Disables interrupt mode in the KK handler. The KK handler places the channel's status in the status register and clears the command register. Completion of the command never interrupts the KX handler.

Get Status Command KC$GS (Code 8)

Instructs KK handler to place its internal status in the data registers. This status information is currently undefined and reserved by DIGITAL for future use. The command is effectively the same as the no-op command. The KK handler places the channel status in the status register and clears the command register. If interrupts are enabled (KC$EI command) a Q-BUS interrupt occurs.

Read Status Command KC$SS (Code 1Ø)

Instructs KK handler to read the new internal status from the data registers. This status information is currently undefined and reserved by DIGITAL for future use. The command is effectively the same as the no-op command. The KK handler places the channel's status in the status register and clears the command register. If interrupts are enabled (KC$EI command) a Q-BUS interrupt occurs.

Read Data Command KC$RD (Code 12)

Causes KK handler to place bytes of data into the channel's data registers. The maximum number of bytes to transfer is specified by data length field KC.LEN (see Section A.3.1.4). The KK handler must be ready to send the data (as indicated by the data-available KS.DA bit in the status register) or it will return a no-data-available (KE$NDA) error in the status register.

The KK handler:

1. Moves the data into the data registers
2. Sets the number of bytes being transferred in the actual length field (KS.ALN) of the status register
3. Sets the end-of-message bit (KS.EOM) in the status register if this is the last transfer in the message
4. Sets any other status
5. Clears the command register and interrupts the arbiter if interrupts are enabled

If the KX handler buffer being filled with data overflows, the KX handler issues a reset KK-handler-to-KX-handler command (KC$RSM) thereafter. If interrupts are enabled (KC$EI command), a Q-BUS interrupt occurs.

Write Data Command KC$WD (Code 14)

Causes KK handler to accept bytes of data from the channel's data registers. The maximum number of bytes to transfer is specified by data length field KC.LEN (see Section A.3.1.4). The KK handler must be ready to accept the data (as indicated by the data-requested KS.DR bit in the status register) or it will return the no-data-requested (KE$NDR) error in the status register. If the KK handler buffer being filled with data overflows, excess data is discarded and the KK handler returns a data overrun (KE$OVR) error.

The KK handler:

1. Examines the length field (KC.LEN) for the number of bytes being transferred
2. Removes the number of bytes specified by KC.LEN from the data registers
3. Tests for the EOM bit (KC.EOM) to find the last transfer in the message
4. Places its status in the status register
5. Clears the command register and interrupts the arbiter if interrupts are enabled

A.3.1.2 **Interrupt When Data Available Bit (KC.IDA)** - The interrupt-when-data-available bit, KC.IDA, when set to 1 indicates the KK handler should interrupt the KX handler when the data-available bit (KS.DA) of the status register changes from Ø to 1. This bit is meaningful only when used with the KC$EI command.

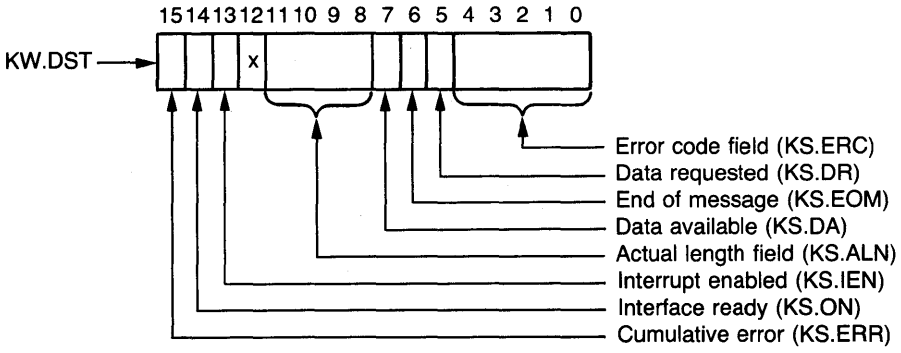
A.3.1.3 **Interrupt When Data Requested Bit (KC.IDR)** - The interrupt-when-data-requested bit, KC.IDR, when set to 1 indicates the KK handler should interrupt the KX handler when the data-requested bit (KS.DR) in the status register changes from Ø to 1. This bit is meaningful only when used with the KC$EI command.

A.3.1.4 **Length Field (KC.LEN)** - The length field, KC.LEN, indicates the maximum number of bytes to be transferred by the read-data (KC$RD) and write-data (KC$WD) commands. This field is meaningful only when used with the KC$RD and KC$WD commands.

A.3.1.5 **End-of-Message Bit (KC.EOM)** - The end-of-message bit, KC.EOM, when set to 1 indicates the last byte in the current transfer that ends the message. This bit is meaningful only when used with the KC$WD command.

A.3.1.6 **Vector Number Field (KC.VEC)** - The vector number field, KC.VEC, specifies the vector number (vector address divided by four) of the interrupt vector being activated by the enable-interrupt command (KC$EI). This field is meaningful only when used with the KC$EI command.

A.3.2 Status Register Definition



Bit 12 is set to Ø.

A.3.2.1 **Error Code Field (KS.ERC)** - The error code field, KS.ERC, contains the following status or one of the following errors after a requested command operation. The codes are designed to allow your program to index them from a table.

Operation Successful Status KE$OK (Code Ø)

The operation previously requested completed without errors.

No Data Available Error KE$NDA (Code 2)

The read data command (KC$RD) was rejected because no data was available.

No Data Requested Error KE$NDR (Code 4)

The write data command (KC$WD) was rejected because no data was requested by the KK handler.

Illegal Command Error KE$ILC (Code 6)

The command specified in the command field (KC.COM) of the command register is invalid.

Illegal Length Error KE$ILL (Code 8)

The number of bytes specified in the length field (KC.LEN) of the command register is invalid.

Illegal Vector Error KE$ILV (Code 1Ø)

The vector number (vector address divided by four) specified in the vector number field (KC.VEC) of the command register is invalid.

KK Handler Buffer Overflow Error KE$OVR (Code 12)

The KK handler buffer being filled by a write data (KC$WD) command overflowed and excess data was discarded.

A.3.2.2 **Data Requested Bit (KS.DR)** - The data-requested bit, KS.DR, when set to 1 indicates the KK handler is requesting data. Thus, a write data (KC$WD) command issued by the KX handler will be accepted by the KK handler.

A.3.2.3 **End-of-Message Bit (KS.EOM)** - The end-of-message bit, KS.EOM, when set to 1 indicates the last byte in the current transfer ends the message.

A.3.2.4 **Data Available Bit (KS.DA)** - The data-available bit, KS.DA, when set to 1 indicates data is available to be read from the KK handler. Thus, the KK handler will accept a read data (KC$RD) command issued by the KX handler.

A.3.2.5 **Actual Length Field (KS.ALN)** - The actual-length field, KS.ALN, is set to the number of bytes to be transferred in response to a read data (KC$RD) command.

A.3.2.6 **Interrupt Enabled Bit (KS.IEN)** - The interrupt-enabled bit, KS.IEN, when set to 1 indicates an enable interrupt (KC$EI) command completed successfully and the KK handler will interrupt the arbiter on the specified interrupt condition.

A.3.2.7 **Interface Ready Bit (KS.ON)** - The interface-ready bit, KS.ON, when set to 1 indicates to the KX handler that the KK handler is ready to perform the protocol.

A.3.2.8 **Cumulative Error Bit (KS.ERR)** - The cumulative error bit, KS.ERR, when set to 1 indicates an error condition exists. The error code is in the error code (KS.ERC) field. When this bit is set to 1, the KS.ALN field is not meaningful and its contents should be ignored.

A.3.3 Interface Initialization

At system start-up the TPR is locked from write access by the KX handler side, and the status and command registers are in a cleared state. The KX handler waits for the KK handler to initialize itself, and indicates its readiness by waiting for the interface-ready bit (KS.ON) in the status register to be set to 1. The KK handler cannot clear the KS.ON bit until it has permanently ceased TPR communication.

APPENDIX B

KXT11-C CSR AND VECTOR ASSIGNMENTS

This appendix lists the interrupt vector assignments for all KXT11-C devices and their associated control status registers (CSRs). Refer to Appendix C for the interrupt vector and CSR assignments for the two-port RAM (TPR).

Vector	CSR Address	Device	Comments
60	177560- 177562	SLU1 console DLART receiver	
64	177564- 177566	SLU1 console DLART transmitter	
70	175700- 175716	SLU2 hardware	Do not specify this vector as an argument to the MicroPower/Pascal DEVICES macro. The kernel routes its interrupts from this vector through vectors 140 to 174.
	175720- 175736	8254 timer 0 and timer 1	Timer 0 and timer 1 on the 8254 device provide timing for SLU2.
100	177520	Line frequency clock	Interrupts through this vector are enabled in the MicroPower/Pascal kernel if CLOCK=ON in the KXT11C macro. The MicroPower/Pascal clock handler enables the interrupt with or without specifying CLOCK=ON. CSR 177520 is KXTCSRA. Bit 6 enables/disables the line frequency clock as in the usual clock CSR at 177546. However, the other bits are allocated to serve other functions.
104	175720- 175736	8254 timer 2	MicroPower/Pascal does not support this timer. You must write your own handler for it. Specify vector 104 in the MicroPower/Pascal DEVICES macro. Timer 2 is enabled by bit 7 in KXTCSRA (address 177520).
110	174400- 174536	DTC channel 0	The native firmware sets up this vector.

114	174400 174536	DTC channel 1	The native firmware sets up this vector.
	175000- 175006	TPR system control	The first four words of the two-port RAM used for KXT11-C native firmware/ arbiter communication. When the arbiter writes to word 0 (TPR command register), the KXT11-C restarts at 173004.
120	175010- 175016	TPR data channel 0	This vector is used when the arbiter writes to TPR word 4 (command register for data channel 0).
124	175020- 175036	TPR data channel 1	This vector is used when the arbiter writes to TPR word 8 (command register for data channel 1).
	177532	QIR	Q-BUS interrupt register. The MicroPower/Pascal KK device handler writes the arbiter's vector address to use for TPR operations.
130	-	Q-BUS interrupt answer-back	This vector is used when the arbiter acknowledges the interrupt requested by the MicroPower/Pascal KK device handler over the QIR.
134	175030- 175036	TPR data channel 2	This vector is used when the arbiter writes to word 12 of the TPR (enabled by a bit in KXTCSRD). Words 12 to 15 of the TPR form data channel 2 which is not supported by the MicroPower/ Pascal KK/KX device handlers. Instead, MicroPower/Pascal uses these locations as the last four words of data channel 1. Do not specify this vector in the MicroPower/Pascal DEVICES macro. Module KSLU2 in MicroPower/Pascal's kernel fans out the interrupts from SLU2, the multiprotocol chip, through the vector at 70 to the vectors 140 to 174.
140	175700- 175736	SLU2 channel A character received	
144	175700- 175736	SLU2 channel A character sent	
150	175700- 175736	SLU2 channel A error	
154	175700- 175736	SLU2 channel A modem control	
160	175700- 175736	SLU2 channel B character received	

164	175700- 175736	SLU2 channel B character sent	
170	175700- 175736	SLU2 channel B error	
174	175700- 175736	SLU2 channel B modem control	
200	177000- 177140	Parallel I/O port A	This vector is set up by the KXT11-C native firmware and used by parallel I/O port A.
204	177000- 177140	Parallel I/O port B	This vector is set up by the KXT11-C native firmware and used by parallel I/O port B.
210	177000- 177140	Parallel I/O timers	This vector is set up by the KXT11-C native firmware and used by parallel I/O port counter-timers.
214	-	Nonvolatile RAM power restore	This vector is used by MicroPower/ Pascal when the KXT11-C, set up for battery backup, performs a power restore operation. You must specify POWER=NONVOL in the KXT11C macro to obtain this service. A user interrupt service routine connected to this vector is called during a power recovery.
220	-	Arbiter RESET	This vector is used when the arbiter's BRESET signal is asserted. BRESET is asserted when the arbiter executes a RESET instruction or its RESTART switch is toggled. Many device interrupts are enabled and disabled by setting bits in the CSRs.

APPENDIX C

SYSTEM ID SWITCH POSITIONS, TWO-PORT RAM CSR AND VECTOR ASSIGNMENTS

This appendix shows the control status register (CSR) and interrupt vector assignments for the two-port RAM (TPR) that are selected by the KXT11-C system ID switch. These registers and vectors appear in the I/O page and vector area of arbiter memory. The table also shows the associated KX device handler logical unit IDs.

The KX device handler passes data between the arbiter CPU and up to 14 KXT11-C peripheral processors running on the Q-BUS. The handler communicates with the KK device handler in the KXT11-C through the command and status registers in the data channel areas of the TPR (see Sections 3.4.6 and 3.4.7 and Appendix A for more information).

ID Switch Position	MicroPower KX Handler ID	TPR Base Address Jumper In	TPR Base Address Jumper Out	Default Vectors MicroPower and RSX-11	Default Vectors RT-11
0	Stand-alone mode				
1	Stand-alone mode				
2	A	17762100	17760100	500,504	340,344
3	B	17762140	17760140	510,514	350,354
4	C	17762200	17760200	520,524	360,364
5	D	17762240	17760240	530,534	370,374
6	E	17762300	17760300	540,544	*
7	F	17762340	17760340	550,554	*
8	G	17777400	17775400	560,564	*
9	H	17777440	17775440	570,574	*
10	I	17777500	17775500	600,604	*
11	J	17777540	17775540	610,614	*
12	K	17777600	17775600	620,624	*
13	L	17777640	17775640	630,634	*
14	M	17777700	17775700	640,644	*
15	N	17777740	17775740	650,654	*

*RT-11 device handlers have no more than eight logical units. Since two units are assigned to each KXT11-C, an RT-11 KX handler can support no more than four KXT11-C peripheral processors. For each four additional KXT11-C processors you want to use on an RT-11 system, you must rename, edit and rebuild the handler. The KXT11-C Software Toolkit/RT Reference Manual describes the procedure to build the second and subsequent KX handlers.

Since RT-11 uses memory above location 500 for passing data during .CHAIN operations, you must restrict all KXT11-C vectors to less than 500, thus allocating vectors to the KX handler that are not in use by other devices in the system.

APPENDIX D

SAMPLE MICROPOWER/PASCAL CONFIGURATION FILE

The following is a copy of the MicroPower/Pascal configuration file that you must edit according to your hardware and software requirements in preparation for building a KXT11-C application.

```
     .enabl     GBL
     .mcall     CONFIGURATION

;+
;
; Module name: CFDKTC.MAC
;
; System: MicroPower/Pascal
;
; Functional Description:
;
; This module describes the hardware and system software configuration on
; which the application system is to run. It must be edited by the user
; and assembled. The resulting object module is used to build the kernel.
; This file has been edited for a typical ROM/RAM system on a KXT11-C,
; including debug symbols and NOT including kernel optimizations.
;
; The following set of macros may be used in this file. The CONFIGURATION
; macro must be the first macro evoked. The ENDCFG macro must be the last.
; A configuration  file must contain at the minimum the CONFIGURATION,
; SYSTEM, PROCESSOR, KXT11C, MEMORY, DEVICES, and ENDCFG macros.
;
; CONFIGURATION {name}
; SYSTEM        optimize[=NO],debug[=NO]
; PROCESSOR     mmu=[NO], fpu{=FP11;=FIS}, type[=L1123], vector[=1000]
; MEMORY        base[=0], size[=0], type[=RAM], parity[=NO], csr[=0]
; DEVICES       v1,v2,...,v6
; RESOURCES     stack[=..KIS],packets[=20.],structures[=3000.],ramtbl[=20.]
; PRIMITIVES    p1[=ALL],p2,p3,p4,p5,p6
; KXT11         nxm[=HALT], break[=TRAP], syshalt[=ODTROM], level7[=TRAP],
;               slu1[=9600], slu2[=9600]
; KXT11C        bhalt=NO, reset=IGNORE, clock=OFF, power=BOOT, map=0
; TRAPS         t1{=ALL;t1},t2,t3,t4,t5,t6,t7,t8
;     .         ALL - TR4, T10, BPT, EMT, and TRP (std. LSI-11 set)
;               TR4 - Trap to 4 (bus timeout)
;               T10 - Trap to 10 (reserved instruction)
;               BPT - Breakpoint instruction trap
;               EMT - EMT instruction trap
;               TRP - TRAP instruction trap
;               MPT - Memory parity error
;
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;       KXT11C Configuration Macro
;
;  This macro defines the KXT11-C power-up configuration.   The macro
;  has the parameters listed below, with the DEFAULTS being the first
;  ones listed:
;
;    BHALT ...  Used to indicate if Q-bus debug traps (B-HALTs) are
;               to cause the routine at $KXTDB to be called.  The user
;               can put a PASDBG breakpoint there.
```

SAMPLE MICROPOWER/PASCAL CONFIGURATION FILE

```
;
;       parameters ... NO       No debug traps are provided on B-halt
;                      YES      Debug traps are provided on B-halt
;
;    RESET ...  Used to indicate the action to take on a Q-bus reset.
;
;       parameters ... IGNORE   Q-bus resets are ignored by the KXT
;                      BOOT     Q-bus resets cause a KXT power-up.
;                      RSTBOT   Q-bus reset causes KXT reset then power-up.
;                      INTRPT   An interrupt is dispatched to vector 220.
;                               The ISR must call or jump to $KXTPU
;                               (KXT11-C power-up routine) when complete.
;
;    CLOCK ...  Used to indicate whether or not to enable the Real-time clock
;               interrupts on the KXT11C through vector 100.
;
;       parameters ... OFF      The real-time clock is not enabled at power-up
;                      ON       The real-time clock is enabled a power-up
;
;    POWER ...  Used to indicate the action to take on a power-fail and
;               subsequent power-up.
;
;       parameters ... BOOT     No power-fail or power-recover processing
;                               will be done.  Every trap through the restart
;                               vector will be treated like a power-up.
;
;    NONVOL ... The non-volatile RAM support routines
;               will be used. When debugging, INIT/RESTARTs
;               by PASDBG present a special problem. Prior
;               to restarting the user must set bit 9 at
;               octal address 175002 for a power-up, or set
;               bit 10 to test power recover.
;
;    MAP ...    Used to set the KXT11C memory configuration.
;
;       parameter ...  0..7     The map parameter must be set to an integer
;                               between 0 and 7 which corresponds to the
;                               memory map jumper settings on the KXT11C.
;                               The parameter is used to select the address
;                               of the power-fail flags which ODT and the
;                               kernel will use.  The last 2 words of native
;                               RAM are used.  Serial ODT uses 60 bytes before
;                               that as its scratch pad.
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

CONFIGURATION

SYSTEM debug=YES, optimize=YES

PROCESSOR mmu=NO, type=KXT11C, vector=224

MEMORY base=<0>, size=<777>, type=RAM, volatile=NO
;
; (note that the highest 64 bytes of the native RAM can not be configured,
;  because they are used by the native firmware)
;
RESOURCES packets=20., structures=2048.

PRIMITIVES ALL

KXT11C bhalt=YES, reset=IGNORE, clock=OFF, power=BOOT, map=0

TRAPS ALL

DEVICES 60,64                   ;Console Serial line
DEVICES 100,104                 ;RTC and Spare Timer

DEVICES 110,114                 ;DMA Vectors
```

SAMPLE MICROPOWER/PASCAL CONFIGURATION FILE

```
DEVICES 120,124,130             ;Two-port RAM arbiter write interrupts

DEVICES 140,144,150,154         ;SLU2 Pseudo-vectors - chan A.
DEVICES 160,164,170,174         ;SLU2 Pseudo-vectors - chan B.

DEVICES 200,204,210             ;PIO and Counter-Timers

DEVICES 214                     ;Power Recover
DEVICES 220                     ;Q-BUS Reset

ENDCFG
```

APPENDIX E

CALCULATING CHECKSUMS FOR PROMS

This appendix tells you how to use the VMS DECprom program to calculate checksums for PROM devices (programmable read-only memories) on the KXT11-C. The checksums calculated by this method can be verified by the ROM checksum test performed by the KXT11-C self-tests in the native firmware. The KXT11-CA User's Guide hardware reference manual describes the algorithm that the ROM checksum test uses to calculate checksums.

DECprom calculates only checksums for PDP-11 processors based on a 16-bit system word. The ROM checksum test in the KXT11-C native firmware expects that each PROM device will contain its own (byte) checksum. Therefore, you must calculate a separate checksum for each PROM device, then load the appropriate checksum value into the last location of each device.

The procedure that follows assumes you know how to use DECprom. Refer to the VAX/VMS DECprom User's Guide for detailed reference information.

1. Using an initialization file that is appropriate for your PROM devices, run DECprom.

2. Set the system word width to 16 bits.

3. Load the PROM devices from the input file type of your choice (.MIM, .LDA, .SAV) but leave the last location of each PROM empty.

4. Exit DECprom.

5. Perform steps 1 and 2 (above) but set the system word width to 8 bits. This allows DECprom to calculate a byte-wide checksum.

6. LIST the contents of each PROM in a binary (.SAV) file. This will produce a file for the low-byte device and a file for the high-byte device.

7. Calculate a separate checksum for each file (CALC_CHECKSUM command).

8. Store the appropriate checksum in the last location of each device (STORE_CHECKSUM command).

APPENDIX F

DETECTING NONFATAL FIRMWARE ERROR CONDITIONS

This appendix tells you how to determine the severity of nonfatal error conditions reported by the automatic self-tests performed by the KXT11-C native firmware. These error conditions can result when the boot/self-test switch is in position 1, 2, 3, 5, 8, 9 or 10 (see Section 3.3.3.2).

Not all failures of the KXT11-C automatic self-tests inhibit application execution. By examining test result data, your application can determine its ability to execute despite the presence of these error conditions.

Automatic self-tests report test results and status information in the status and data registers of the TPR system control area. This data is available to your application only after the tests complete and before other routines or handlers in the application use these registers. The KXT11-CA User's Guide hardware reference manual shows the format of the status registers and tells you how to interpret the test results.

The procedure to use when checking results of these tests depends on your operating system environment. The following sections describe the procedures to follow when using the KXT11-C in stand-alone mode and in peripheral processing mode under MicroPower/Pascal, RT-11 and RSX-11.

F.1 STAND-ALONE MODE

An application initialization routine should test for error conditions that are considered intolerable for the specific application (for example, DMA errors if the DMA controller is used by the application). If an error condition is found that is critical to the operation of the application, the code should take the appropriate action for that particular situation.

F.2 PERIPHERAL PROCESSING MODE

The procedure to use depends on where the application resides.

F.2.1 Application in ROM

This is similar to the stand-alone environment. Once the application code is entered on power-up, it should check for critical error responses from the automatic self-tests and indicate the conditions to the arbiter using your application's communication protocol.

F.2.2 Application Loaded from Arbiter with MicroPower/Pascal

The KXT_LOAD utility procedure will not load the KXT11-C if there are any error conditions. If you load the KXT11-C in some other fashion, you can write a program, as described in Section F.2.3, to check for error conditions.

F.2.3 Application Loaded from Arbiter with RT-11 or RSX-11

You use the KUI utility program to load an application in the RT-11 or RSX-11 environment. Before running KUI, however, you must create a small application program that examines the status registers in the system control area of the TPR for error conditions. This program should directly access these registers and examine the results of the automatic self-tests. To simplify this task, you can run this program and the KUI utility under the control of an RT-11 or RSX-11 command file using applicable parameters.

Your program should perform the following tasks.

1. Poll the system control area's status register for the waiting-for-command condition.
2. Examine the composite error response in the TPR for relevant errors.
3. If there is a relevant error, set a flag that can be passed back to the indirect command file processor.
4. Repeat steps 1, 2 and 3 (above) for each additional KXT11-C in the system.
5. Return to the command file with the error parameters set appropriately.

The command file should then examine the returned parameters and take the appropriate action. If there are no significant automatic self-test failures, the command file should execute KUI to load the peripheral processor(s).

INDEX

HOW TO ORDER ADDITIONAL DOCUMENTATION

From	Call	Write
Chicago	312–640–5612 8:15 A.M. to 5:00 P.M. CT	Digital Equipment Corporation Accessories & Supplies Center 1050 East Remington Road Schaumburg, IL 60195
San Francisco	408–734–4915 8:15 A.M. to 5:00 P.M. PT	Digital Equipment Corporation Accessories & Supplies Center 632 Caribbean Drive Sunnyvale, CA 94086
Alaska, Hawaii	603–884–6660 8:30 A.M. to 6:00 P.M. ET *or* 408–734–4915 8:15 A.M. to 5:00 P.M. PT	
New Hampshire	603–884–6660 8:30 A.M. to 6:00 P.M. ET	Digital Equipment Corporation Accessories & Supplies Center P.O. Box CS2008 Nashua, NH 03061
Rest of U.S.A., Puerto Rico*	1–800–258–1710 8:30 A.M. to 6:00 P.M. ET	
Canada British Columbia	1–800–267–6146 8:00 A.M. to 5:00 P.M. ET	Digital Equipment of Canada Ltd 940 Belfast Road Ottawa, Ontario K1G 4C2 Attn: A&SG Business Manager
Ottawa–Hull	613–234–7726 8:00 A.M. to 5:00 P.M. ET	
Elsewhere	112–800–267–6146 8:00 A.M. to 5:00 P.M. ET	
Elsewhere		Digital Equipment Corporation A&SG Business Manager*

*Prepaid orders from Puerto Rico must be placed with the local DIGITAL subsidiary (call 809–754–7575)

*c/o DIGITAL's local subsidiary or approved distributor

KXT11–C Peripheral Processor
Software User's Guide
AA–Y615A–TK

READER'S COMMENTS

NOTE: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well organized? Please make suggestions for improvement.

Did you find errors in this manual? If so, specify the error and the page number.

Please indicate the type of user/reader that you most nearly represent.

- ___ Assembly language programmer
- ___ Higher-level language programmer
- ___ Occasional programmer (experienced)
- ___ User with little programming experience
- ___ Student programmer
- ___ Other (please specify) _____

Name _____ Date _____

Organization _____ Telephone _____

Street _____

City _____ State _____ Zip Code _____
or Country

Do Not Tear — Fold Here and Tape

digital

No Postage Necessary if Mailed in the United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

SSG/ML PUBLICATIONS, MLO5–5/E45
DIGITAL EQUIPMENT CORPORATION
146 MAIN STREET
MAYNARD, MA 01754

Do Not Tear — Fold Here