8-14-1986

A Multiprocessor Distributed Debugger

David Blackman

Rochester Institute Of Technology
School Of Computer Science and Technology

A Multiprocessor Distributed Debugger

David Blackman

A thesis, submitted to The Faculty of the School of Computer Science and Technology, in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

Approved by:

Professor Roy Czernikowski

Professor John Ellis

Professor Margaret Reek

August 14, 1986

Abstract

This thesis presents the design and implementation of a distributed debugger The debugger was designed to support the debugging of a system containing multiple processors from a single debug console. The debugger implementation consists of host software which runs on a VAX minicomputer and target software which runs on Intel SDK-86 single board computers. The host and targets communicate using an RS-232 channel.

The debugger supports breakpoints, disassembly of target code, symbolic reference of program procedures and variables, and download of Intel Object Module Format binary files.

Table of Contents

Figures

1. Introduction and Background

1.1 Problem Statement

The continually dropping costs and increasing performance of integrated circuits has made the embedded microprocessor system an effective solution for many engineering problems. A board containing a 16 bit microprocessor, RAM, ROM, and I/O can be built for less than $500. These low prices coupled with the growing number of applications which have high computational demands, tight real-time constraints, extensive I/O processing requirements, or are fault-tolerant make multi-microprocessor systems very attractive [1].

Two problems arise when developing software for this type of system [2]. The first is that these computers do not include sufficient hardware to directly develop and debug software that runs on the system. The reason is that the additional hardware for support of a full software development environment would make the resulting system too expensive. The second problem is that current debuggers are designed to debug *single* processor target systems; debugging of a multiprocessor target system is usually done in an ad hoc manner.

The goal of this thesis is to design and implement a debugger which is effective for debugging multiprocessor embedded systems. The debugger should minimize the use of the target system's limited resources and be inexpensive to implement. Although it is being developed for the Intel SDK-86 single board computer, it should be easily adaptable to work with other processors.

A debugger which can support a target system consisting of a hetereogenous set of processors offers the advantage of providing a single uniform user interface to the debugger, rather than a different terminal and command language as an interface to each processor's debugger in the target system. It also offers the possibility of providing features not possible with the single debugger per processor approach such as being able to globally start or stop a system, or to set breakpoints across machine boundaries.

1.2 Previous Work

This section surveys seven methods described in the literature or encountered in experience used to debug software for embedded computer systems.

1.2.1 PROM Monitor

Some systems include a PROM resident debugger on the microprocessor system which allows a user to enter programs and debug commands through a keypad/LED display or terminal I/O. This method is practical for small programs which can be hand assembled or assembled on another machine and typed in each time they are to be debugged. These debuggers generally have cryptic command sequences and no online help, due to the limited amount of program and data space available for the debugger.

1.2.2 PROM Monitor/Communications Channel

An enhancement to the PROM resident debugger is to include a communications channel on the target system used to download object code from a host computer. Programs can then be edited and cross assembled or compiled on the host computer and downloaded to the target system. An example of this idea is implemented on the Intel SDK-86 single board computer [3]. The disadvantages of this solution are the inability of the PROM debugger to use symbol table information generated by compilers or assemblers (either because the debuggers are not sophisticated enough or because there is insufficient memory in the target system to hold the symbol table) and lengthy download times resulting from the limited bandwidth of the communications channel. For example, it takes over two minutes to download a 64 KByte program encoded in Intel Hex Format using a 9600 bps asynchronous serial channel.

1.2.3 Expandable Target Systems

Another method is to design the target system to be expandable so extra hardware can be added to the target to support a full software development environment. For example, the Intel 86/30 CPU board includes a Multibus interface allowing the CPU board, a disk controller, and a memory board to be plugged into a card cage and run the full Intel iRMX-86 operating system/development environment. Once the software is debugged, it is committed to PROM and the extra hardware is removed. This method works only if the target system is designed to be expandable up front. It also

has the disadvantage that once the extra hardware is removed, only the remaining hardware and software may be used to debug the system resulting in the problems described in sections 1.2.1 and 1.2.2.

1.2.4 In Circuit Emulation

In circuit emulation (ICE) is probably the most powerful method of debugging embedded systems. In an in circuit emulation debug environment, the microprocessor of the target system is replaced with a probe that performs real time emulation of the replaced processor, i.e. behaves identically to the microprocessor. In addition to emulation, the ICE system has the capabilities of starting and stopping processor execution, downloading code into the target memory, reading and writing processor memory, registers, and I/O ports, and breakpointing in *hardware*. As in circuit emulation does not rely on target resident code or data for correct operation, it is impervious to the problem of errant software writing over the debugger, rendering systems undebuggable after a crash. ICE systems have three principle disadvantages. Microprocessor development systems/ICE systems are priced greater than $10K. This high unit cost coupled with the fact that most ICE systems can only be used with one target system at a time results in high costs when debugging a multiprocessor system. The timing delays which are introduced by the additional electronics and cabling often require that the ICE to run with more wait states than the emulated processor. This may mask or introduce timing related software problems. Finally, ICE systems are generally available months after a microprocessor is introduced to the market. This means that another method of debugging must be used until the ICE becomes available.

1.2.5 Logic State Analyzers

A logic state analyzer is a device consisting of a high speed trace memory, trigger hardware, and probes which can be clipped onto a microprocessor in a system under debug. When debugging microprocessor software, the analyzer records processor instruction fetches, memory reads and writes, and I/O reads and writes in its trace memory. The trigger hardware is programmed by the user to select a sequence of bus cycles which must occur before the analyzer starts to collect bus cycles and is programmed to select which bus cycles to capture after the trigger sequence is recognized [4]. Many logic state analyzers can timestamp each captured processor bus cycle. As an example, an analyzer could be programmed to trace instruction fetches (i.e. trace program

execution) that preceded the write of an incorrect value to a program variable. Or, it could be programmed to trace execution of an interrupt service routine; linked with the timestamp information, this data could be used to measure interrupt service time.

As with in circuit emulators, the use of logic state analyzers when debugging a multiprocessor system would be quite expensive. Since logic state analyzers are passive devices, they do not allow a user to arbitrarily examine a target's memory or I/O ports after the trigger sequence occurs. Furthermore, they cannot directly monitor events within the processor, e.g. reads and writes of processor registers. Logic state analyzers cannot download target systems; therefore one of the other techniques outlined here is used in conjunction with a logic state analyzer. Usually, logic state analyzers do not use the symbol table information generated by assemblers or compilers which forces the user to deal with collected bus cycles at the machine level.

1.2.6 Simulation

Using the technique of simulation, target system software is executed or emulated on a host computer offering a rich program development and debug environment. If the target system software is written in a high level language supported on the host, it may be compiled with the host's native compiler and debugged on the host. If a native compiler is not available or if the software is written in assembly language, the software is translated to target machine code using a cross compiler or assembler and run using a target simulator/debugger on the host [2]. In both cases, software must be written to handle input and output instructions contained in the target code and to simulate interrupts from external hardware.

Simulation offers the advantage of allowing a programmer to use the host's debugger or a host-resident target simulator/debugger. This environment is more powerful and easier to use than a target resident debugger as the host machine offers a richer execution environment than the target hardware (i.e. more memory, higher performance CPU, disk storage) for a debugger. It also allows code to be developed before or in parallel with the development of a system's hardware. Simulation has the principal disadvantage of not running in real time which results in lengthy simulation runs or the masking of time dependent errors which often occur in real time microprocessor code. This problem would be exacerbated by the simulation of a multi-processor target system. Due to the inability of simulation to flush out hardware, I/O, and timing related bugs, simulation must be followed by a

debug phase using the target hardware and one of the other debugging techniques outlined here.

1.2.7 Distributed Debuggers

In a remote debugger developed for the Alto workstation [5], a small debugger nub (server) runs on the target system, while a debugger host runs on another Alto. The server and host communicate using the Ethernet. The debugger server, which is only 400 bytes of code, implements Ethernet communication primitives and routines to service requests from the host, such as: write/read memory, start/stop processor, proceed from breakpoint, and obtain processor status. The remote debugger has the interesting property that the machine being debugged and the debugger can be physically separated by large distances, limited only by the extent of the network.

The Alto remote debugger does not support simultaneous debugging of multiple processors from a single debug console and is not designed to work with other types of processors.

Joff [6] is a distributed debugger developed at Bell Labs for debug of programs which run on the Blit terminal. It consists of a debugger process that runs on the target system (the Blit terminal) which is 1000 lines of C code and a debugger host which runs on a VAX which is 6000 lines of C code. Here, the debugger server and host communicate using an RS-232 channel. The debugger host has access to the symbol table information generated when the Blit programs are compiled; therefore full symbolic debug at the source level is supported.

DAD (Do All Debugger) [7] is a distributed debugger designed to work in the National Software Works (NSW) network environment. It is used to debug applications programs consisting of processes which are running on a hetergeneous set of machines in a network. Its goals include source level debug and to provide a uniform user interface regardless of the languages the application is written in and the machines the processes are running on. It is also designed to be extensible so new languages and machines can be supported.

DELTA [8] is a multilingual debugger for the Honeywell CP-6 operating system designed to debug programs running locally on a Honeywell DPS 8 mainframe and programs running remotely on a communications front end processor. DELTA is decomposed into three layers. The first layer, which executes on the host DPS 8 computer, contains software which is independent of the machine being debugged. It consists of the debugger user interface, symbol table management, and routines to format and display machine independent data structures (e.g. character data). The second

layer, which also executes on the host, contains machine dependent functions such as formatting of data structures which differ among machines (e.g. pointers), disassembly of code, and display of hardware faults. Layer three executes on the target hardware and communicates with layer two either with direct procedure calls (for local debugging) or via an exchange of messages over a communications link (for remote debugging). This layer contains routines to fetch/store memory, set breakpoints, trace program execution and start/stop execution of a user's program.

The Distributed Incremental Compiling Environment or DICE [9][10] is a distributed programming environment designed to explore the use of incremental compilation for program development. It consists of an editor, incremental compiler, debugger, and data base which run on a host computer—the DEC20 computer and a debug server which runs on a target system—the PDP-11. The DecNet is used as the communications link between the host and target system. The host to target communications primitives used in DICE include halt and resume target execution, store/fetch target memory, store/fetch target registers and block move; the target to host messages are breakpoint/runtime error and requests to perform simulated terminal input/output. In DICE, Pascal statements may be entered into the debugger, incrementally compiled on the host, downloaded to the target, and executed remotely. DICE is an unusual example of an important concept resulting from the use of a distributed architecture. Large and sophisticated tools can be used on a large host machine to develop programs for a modest target. In DICE, the development environment was implemented with 20,000 lines of Interlisp running on the host system and a server running on the target to handle a small set of primitives. As another example, if the debugger host was running on UNIX, tools such as make, sccs, sed, etc. could be used while developing target code [11]. Furthermore, debugger user interfaces and command languages could be quickly prototyped and evaluated using yacc and lex. Due to the size of the programs generated by yacc and lex, this approach is not feasible for debuggers resident on the target system.

1.3 Theoretical and Conceptual Development

The previous section illustrates that most traditional debugging techniques fail to satisfy at least one of the debugger's goals. However, a distributed debugger as described in 1.2.7 is capable of meeting all of the debugger goals, i.e. minimizing the use of target resources, being inexpensive to implement, and being suited for debug of target consisting of a heterogenous set of processors.

A distributed debugger should use approximately the same amount of target's resources (i.e. ROM and RAM) as a conventional debugger. At the expense of adding code to a conventional debugger to handle server-host communications, the user interface, symbol table management, and disk I/O routines can be offloaded to the host. The debugger host, running on a mainframe computer, has access to a large amount of memory and disk storage allowing it to implement features such as symbolic debug, a friendly user interface, online help, logs of debug sessions, a command history mechanism, etc.

To support debug of a heteregenous set of processors, a generic protocol will be used for host-server communications. The protocol should be able to support different machine address spaces (e.g. code, data, microcode, I/O), word widths, data and instruction formats, register complements, and breakpointing capabilities. Although the generality of the protocol will increase the size of the servers in each target system, it eliminates the necessity of modifying large parts of the host software when supporting a new target. Furthermore, any enhancements in the debugger host will immediately be useful for all targets. RFC-909 [12] specifies a Loader/Debugger protocol which satistifes these requirements and will be used for this debugger.

To simplify the future support of other processors, the debugger host will be partitioned into machine dependent and machine independent modules. Each supported processor will require a set of machine dependent modules which perform target dependent functions such as mapping the processor's register names, runtime error types, address formats, etc. into parameters used by the Loader/Debugger protocol.

2. Debugger Specification

2.1 Functional Specification

2.1.1 Functions Performed

The debugger's functions can be divided into three catagories: data transfer, processor control, and miscellaneous functions.

The data transfer commands display target data, and move data between the host and the locations in the targets' address space. The supported target address spaces include memory, I/O and processor registers.

The debugger will support five generic data types which may be displayed or written to the target's address space. The types are byte, character, long, pointer, and word data which are assigned interpretations by the processor dependent modules. The conversion of data received from the target to printable strings or from VAX long words to target format will be performed in the processor dependent personality modules of the debugger.

The data transfer functions include commands to interactively read or write processor memory, I/O ports, or registers, to fill target memory with a pattern, to download target memory from a VMS file, and to disassemble target memory.

The processor control functions include commands to start, single step, or stop the currently selected processor, and commands to set and clear breakpoints. In addition, the debugger will provide commands to start or stop *all* processors in a target system.

The miscellaneous functions include setting the radix for display or input of data (e.g. hex, octal, decimal), and a command to select the current target.

2.1.2 Limitations and Restrictions

The debugger should be able to use symbol table information generated by assemblers and compilers to allow symbolic reference of global variables and program locations. However, it does not support symbolic reference of local (stack resident) variables in block-structured languages such as Pascal or PL/M.

The debugger will download files in machine dependent format from the host to the target system. For Intel 8086 targets, this format will be the Intel Object Module Format [13]. The low transfer rate between the host and target may result in lengthy download times.

The user code must not disable interrupts for longer than one millisecond as not to lose characters in host—target messages.

2.2 System Specification

The distributed structure of the debugger entails the need for a communications channel between the debugger host and server(s). The hardware for support of the channel is the main expense associated with the distributed debugger. Figures 2.1 through 2.4 illustrate possible distributed architectures. A network such as Ethernet or Cheapernet could be used allowing an arbitrary number of target systems to be debugged from one mainframe. Unfortunately, Ethernet hardware is not present on the Intel SDK-86 and the cost of adding an Ethernet interface to each target is greater than $400 (mostly in transceiver costs). Cheapernet has a much lower connection cost ($100) as it eliminates the transceiver and uses a less expensive network medium but has the disadvantage that not many mainframes support it. To work around this problem, a bridge could be used which transfers packets between an Ethernet and a Cheapernet. Such a bridge is currently unavailable.

Due to these considerations, the debugger will use point to point serial connections between the mainframe and the targets; serial ports are readily available on the host system to be used for this thesis and the cost of adding a serial port to the SDK-86 is less than $50. Unfortunately, the use of a serial connection will limit transfer rates between the host and server to 9600 bps, prolonging the time to read or write large blocks of target memory

Figure 2.4 illustrates the layering of the host and target software. The host software consists of the debugger host, the loader/debugger protocol layer, transport and data link layers, and a multiport serial I/O driver. The host operating system, VMS, provides the implementation of the serial I/O driver.

The target software consists of a server for the loader/debugger protocol, transport and data link software, and a low level serial I/O driver which provides the basic transmit and receive primitives. Both sides contain transport and data link software to provide the peer loader/debugger layers with an error free, frame-oriented communications channel. To allow communications with multiple targets, the host system must support multiple transport connections; each target system only needs to support a single transport connection.

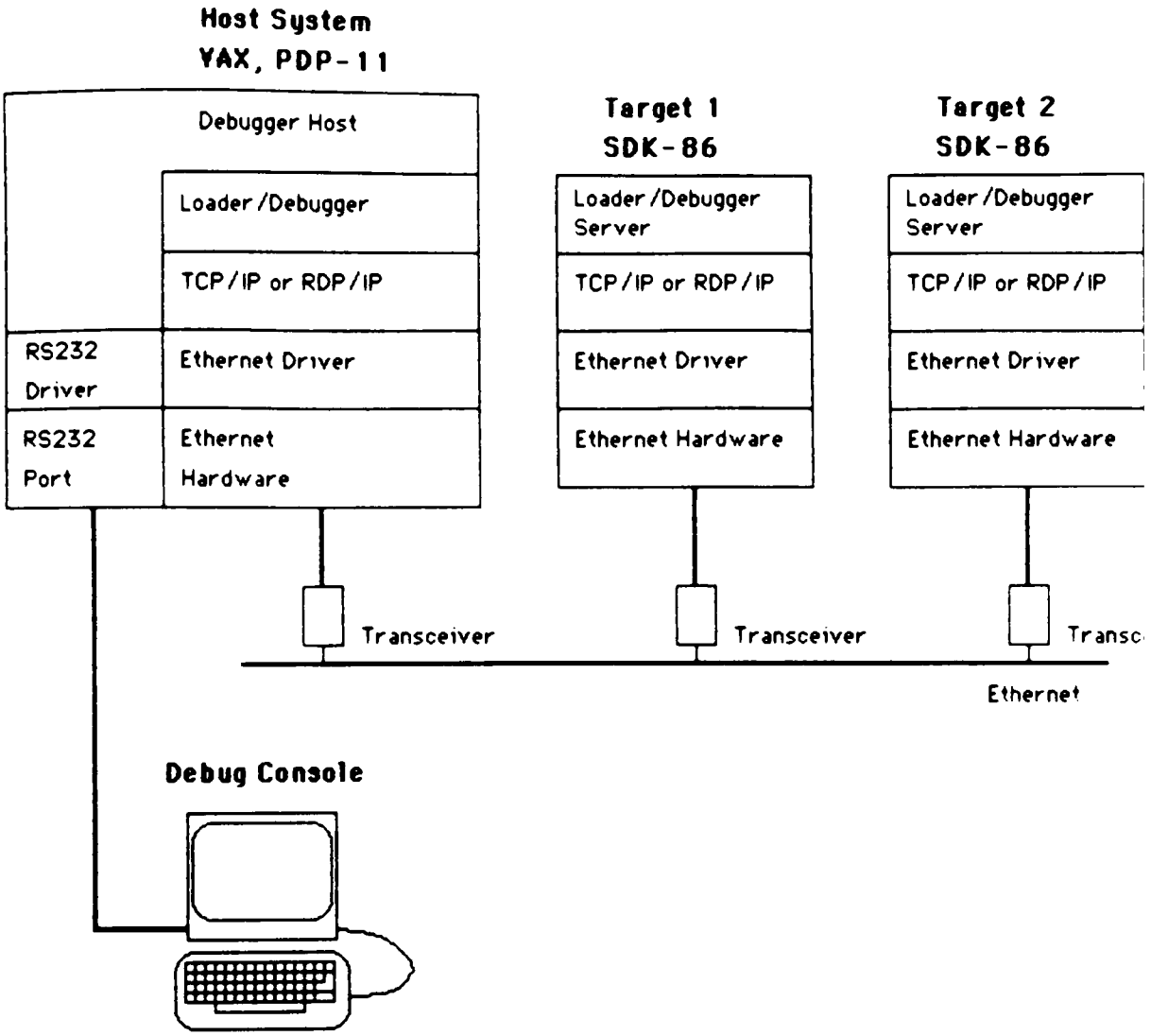


Figure 2.1. Ethernet host-target communication.

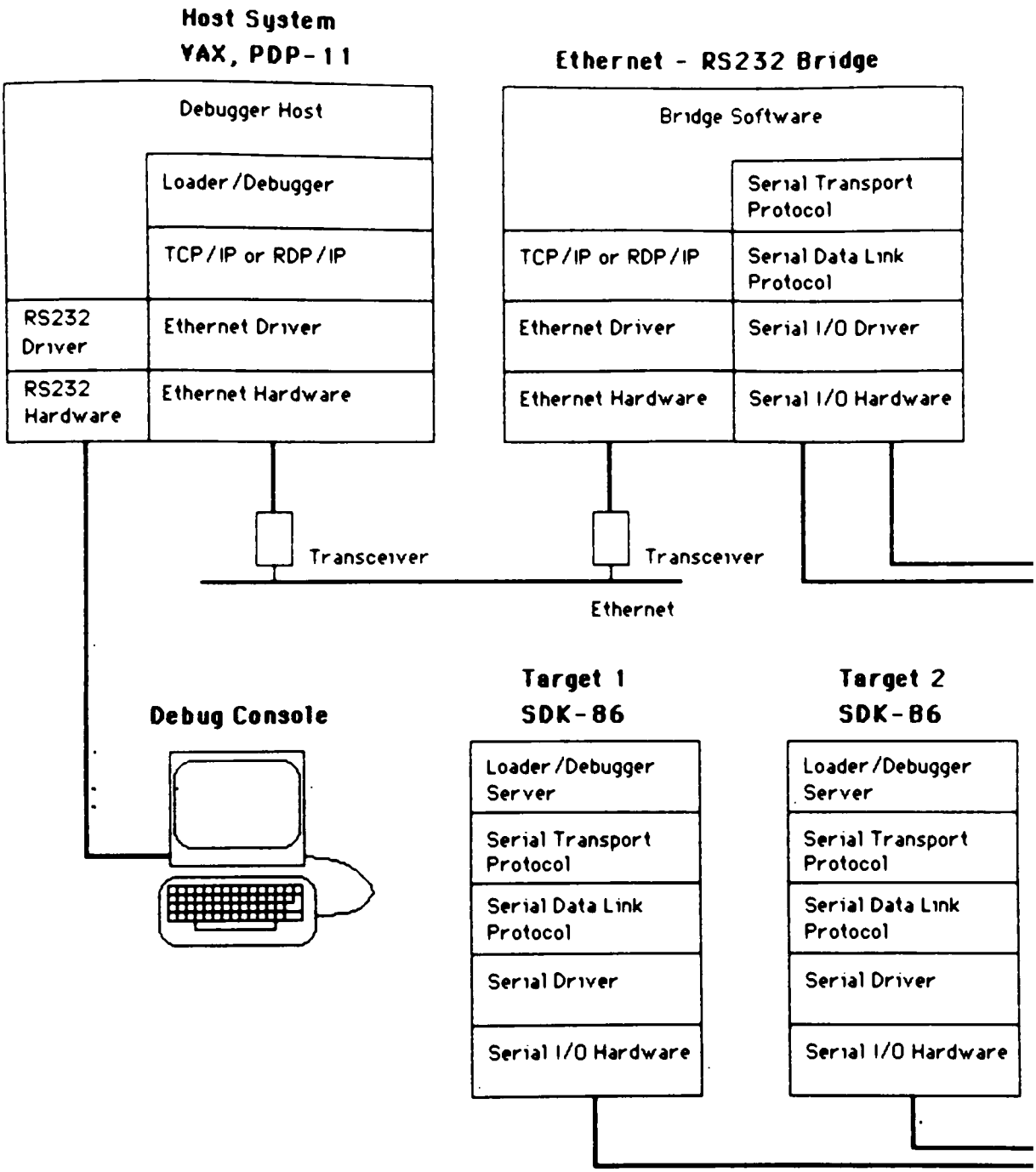


Figure 2.2 Ethernet/Serial host-target communication

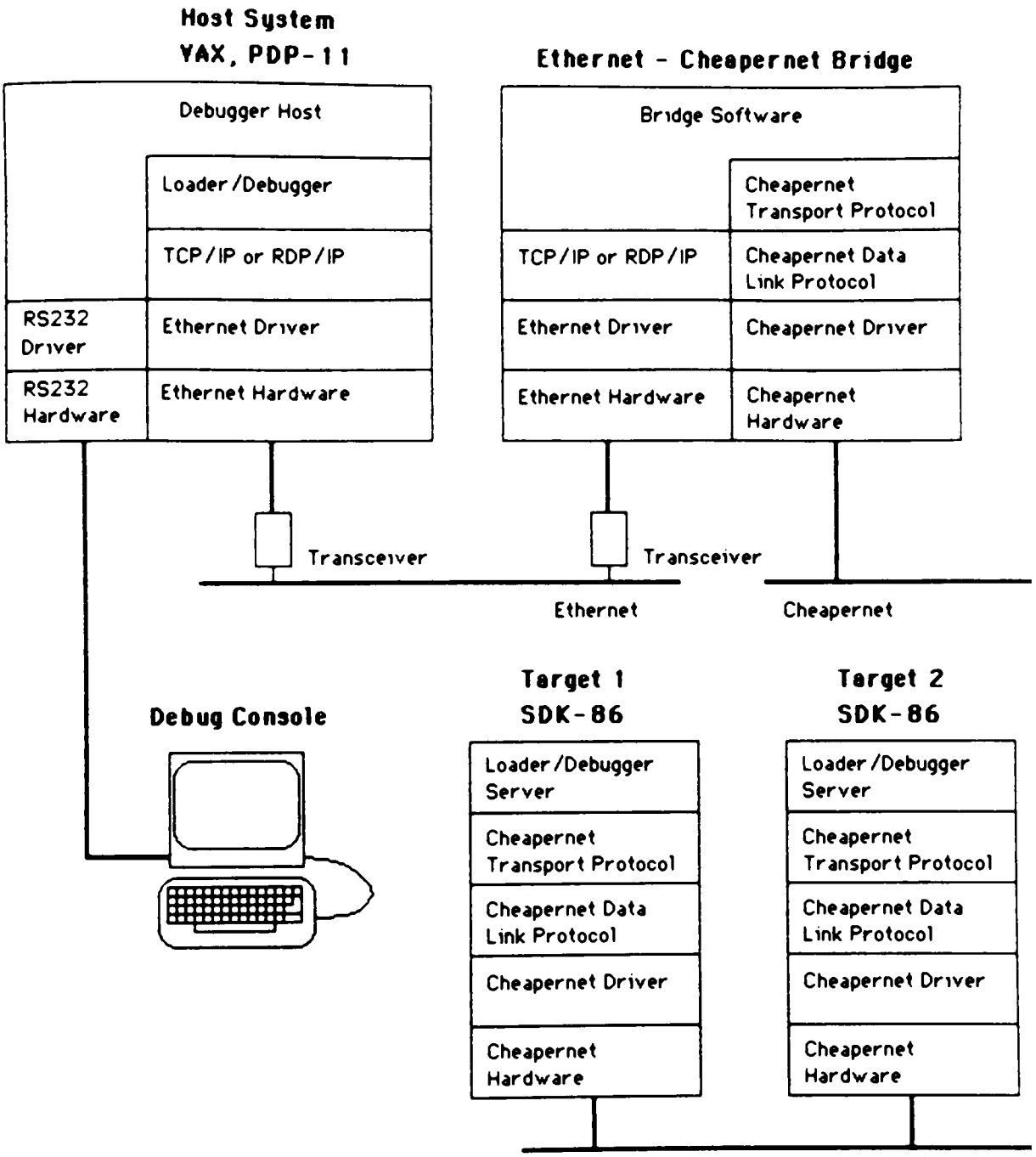


Figure 2.3. Ethernet/Cheapernet host-target communication.

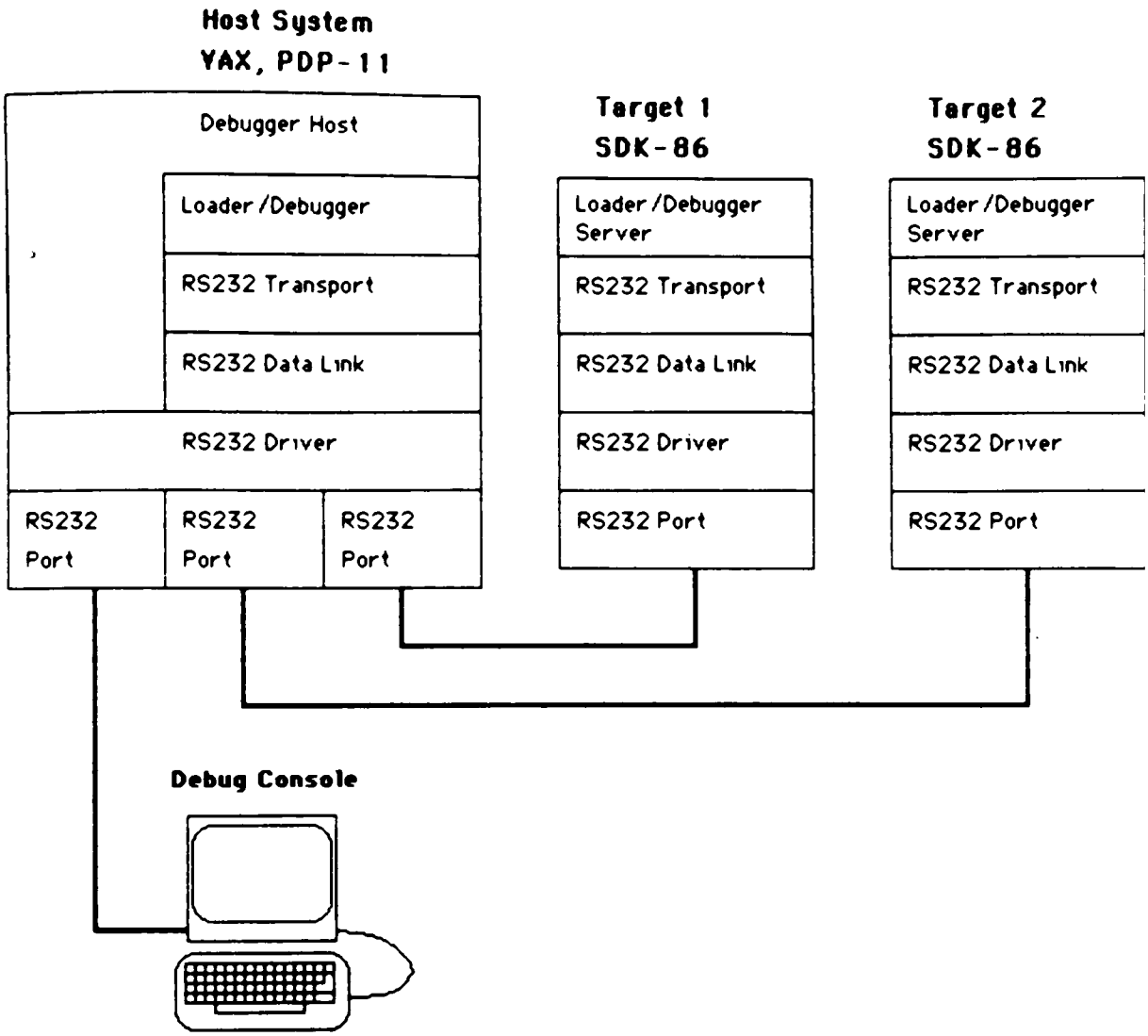


Figure 2.4. Serial host-target communication.

2.2.1 The Debugger Host

The debugger host software can be broken into three functional blocks: a user interface/command parser, command execution, and processor personality modules. The debugger host will run on the VAX/VMS operating system and will be implemented using the C language. VMS was chosen for two reasons. There are a large number of development tools for the 80x86 microprocessor are available for this operating system. These tools include C, Fortran, and Pascal compilers and an assembler for the 8086. This support is not available for UNIX. VMS also provides the operating system routines necessary to implement the communications portion of the debugger such as interprocess communication, timeouts, and critical section protection.

The user interface/command parser is a target independent module which will accept command strings from the debug console, check them for syntactic validity, and parse them into tokens for interpretation by the command execution modules.

The target independent command execution modules are responsible for translating parsed commands into loader/debugger protocol primitives, submitting them for execution through calls to the transport layer, and displaying the results. To convert processor dependent entities (e.g. data and addresses) into loader/debugger parameters, the command execution module will make calls into the processor personality module for the currently selected processor.

For each supported processor, there will be a processor personality module which exports a data structure containing conversion procedures, an optional disassembler procedure, a download procedure, and processor parameters. The conversion procedures perform the mapping between processor dependent strings (e.g. memory addresses, I/O port locations, and register names) and loader/debugger protocol parameters. A conversion routine is also necessary to map target runtime error codes received from the loader/debugger into strings for display on the debugger console. Through the use of a single data structure to hold these procedures and parameters, the debugger will be able to select a new processor simply by setting a pointer to the appropriate personality data structure.

2.2.2 Loader/Debugger Protocol

RFC-909 [12] specifies a loader/debugger protocol (LDP) designed to work with a variety of machine architectures to provide the functions of remote debugging and downloading target systems. It specifies three levels of functionality; in increasing complexity they

are: loader/dumper, basic debugger, and full debugger The loader/dumper level provides protocol management commands such as establishing a connection and reporting protocol errors. It also includes the functions of reading, writing, moving, and filling target memory, starting the target, and a runtime exception reporting mechanism.

The next level, the basic debugger, includes all the loader/dumper commands plus commands to single step the target, set breakpoints, continue from breakpoints, and report status.

The last level, the full debugger, includes a mechanism for setting complex conditional breakpoints using an FSM (finite state machine) model. The FSM is constructed using loader/debugger messages which specify the number of machine states and a set of condition/action pairs for each state. The conditions that can be programmed include tests of processor registers, memory, and debugger maintained counters. The actions include commands to halt execution, continue execution, switch the state of the finite state machine, and to clear and increment the counters. When a breakpoint is hit, the debugger server evaluates each condition for the current state until one is satisfied by the machine state. It then executes the action associated with the true condition. This debugger server and host for this thesis will implement the basic debugger level.

The protocol uses a single address format to refer to target data in a variety of address spaces. The LDP allows addresses to be specified in one of two forms—long or short format. The long address format consists of a 7 bit mode field, a 32 bit ID field and a 32 bit offset. The mode field specifies which address space the ID and offset fields refer to (i.e. system RAM, register, I/O space, etc.). A possible interpretation of the ID, offset pair could be a process number and virtual address for machines supporting multiple address spaces. The short address format consists of a 7 bit mode field and a 32 bit offset. The debugger will use the short format which will decrease the size of the server and make better use of the limited bandwidth between the host and target. The 16 bit segment and 16 bit offset comprising an 80x86 address will be packed into the 32 bit offset field. Appendix G contains a summary of the LDP requests and responses.

2.2.3 RS-232 Transport Protocol

Due to the low channel bandwidth and the desire to minimize the size of the target software, several standard transport and data link protocols were evaluated and rejected because of large and complex encapsulation overheads. For example, TCP/IP has a header consisting of at least 40 bytes. Approximately 15% of the

channel bandwidth would be consumed just by the TCP/IP headers with a 256 byte data field at 9600 bps. For this reason, new transport and datalink protocols were designed for the debugger

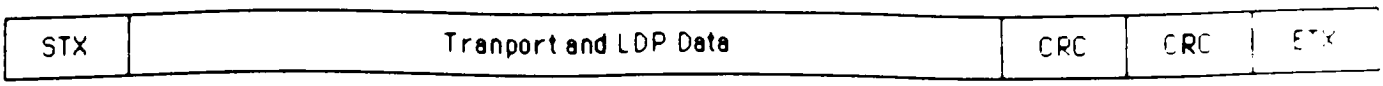
The transport layer will provide an error free transmission facility between the host and server loader/debugger layers. It will use a one byte highly encoded control byte similar to the control field in SDLC [14]. There are three frame formats supported by this scheme—the information, supervisory, and unnumbered formats. Figure 2.5 illustrates the various communications frame formats, which are described below.

Information frames are used for transmission of client data; these frames also allow piggybacking of acknowledgements. The information frame control byte contains a three bit *ns* sequence number, a three bit *nr* sequence number, and one bit used to specify the frame format. The Poll/Final bit of the SDLC protocol will not be used. The *ns* sequence number is incremented for each frame successfully transmitted; the *nr* sequence number is used to acknowledge all frames up to the value of *nr* minus one (modulo eight).

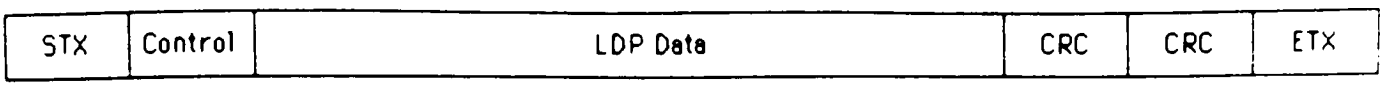
Supervisory frames are used for acknowledgements and flow control. A supervisory frame control byte contains a three bit *nr* sequence number, two bits to specify the supervisory frame format, and two bits to specify the supervisory frame type. The RR (receiver ready) supervisory frame type is used for acknowledging messages; the RNR (receiver not ready) frame type is used to acknowledge a message and close the receive window when receiver buffers are temporarily unavailable. A RR format frame will be sent when the receiver window is reopened after receiver buffers have become available.

Unnumbered frames are used for establishing the transport connection. Their control byte contains a five bit field to specify the unnumbered frame type and two bits to select the unnumbered frame format. A RIM (Request Initialization Mode) format frame will be sent by the debugger host to a target to open the connection; the target will respond with a UA (Unnumbered Ack) format frame.

To simplify the implementation of the debugger server and to decrease the amount of memory needed for buffering unacknowledged transmissions, a one frame sliding window will be used. The tradeoff here is lower transfer rates due to the stop and wait-for-ack nature of the resulting protocol.

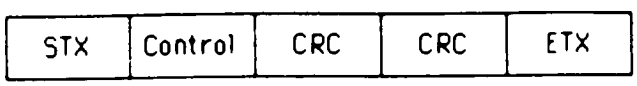


Datalink frame format

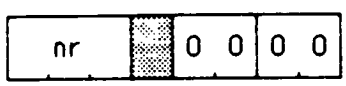


Information format transport frame

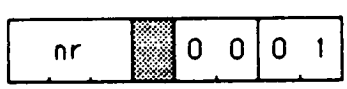
Information frame control byte



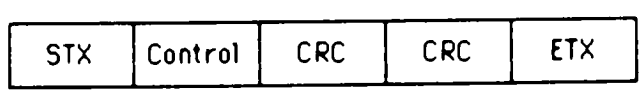
Supervisory format transport frame



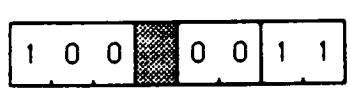
Supervisory RR Control Byte



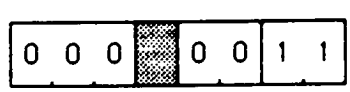
Supervisory RNR Control Byte



Unnumbered format transport frame



Unnumbered RIM Control Byte



Unnumbered UA Control Byte

Figure 2.5. Datalink and Transport frame formats.

2.2.4 RS-232 Data Link Protocol

Generally, the function of a data link protocol is to provide an error free, flow controlled, frame-based transmission facility [15] However, for this application, the data link protocol does not need to provide flow control, retransmissions, or duplicate suppression. Flow control is not required since multiple processes are not competing for the use of a limited pool of receive buffers. Retransmissions and duplicate suppression are not required as these functions will be supplied by the transport layer. Therefore, the data link protocol will provide a frame-based datagram (connectionless) service. Transmission errors will be detected with the 16 bit CCITT CRC. Although the 8251A USART does not include hardware to generate or check the CRC, a table driven algorithm can be used which minimizes CPU bandwidth at the expense of a 512 byte table [16].

The data link level requires that the physical link be able to transmit eight bit characters. This eliminates the need for packing bytes and words into odd sized units (e.g. seven bit characters) and improves performance as less channel bandwidth is used for transmitting start and stop bits.

As in the BISYNC protocol, framing will be accomplished through the use of two reserved byte codes: a start of frame byte (STX), and an end of frame byte (ETX). An escape sequence is used to achieve transparency when data contains a reserved byte code [17]. The ETX character will be a carriage return since many terminal drivers terminate input lines with this character.

The data link format uses four bytes of overhead: one start of frame byte, two CRC bytes, and one end of frame byte. The transport layer has a one byte overhead. At 9600 bps, these overhead bytes require approximately five milliseconds to transmit and result in a datalink/transport overhead less than 3% when using a 256 byte data field.

2.2.5 Target System Hardware Support

Approximately ten integrated circuits will be added to the SDK-86 to support the RS-232 communications channel to the host. The circuit will be wire wrapped on a prototype board and plugged into the SDK-86 bus expansion interface with two 50 wire ribbon cables allowing it to be easily detached when the SDK-86 is to be used in other projects. The additional circuitry consists of four functional blocks: chip select logic, an interrupt controller, the serial I/O interface, and additional ROM/RAM sockets. Appendix A contains schematics for the hardware support.

The debugger server code must be interrupt driven: when user

code has control of the processor, the debugger server would be unable to poll the communications hardware for host requests. Because the SDK-86 does not include an interrupt controller, an Intel i8259A will be included on the prototype board.

The communications interface logic includes a USART, the RS-232 electrical interface, and baud rate generation logic. Although the SDK-86 already has a USART, a second Intel i8251A USART will be added for two reasons. The first is so the debugger does not interfere with target code using the SDK-86 USART. The second reason is that the SDK-86 USART is not wired to support interrupt driven operation. Rather than cutting traces and adding wires to support interrupts, a second USART will be used.

A single Maxim MAX232 integrated circuit will support the RS-232 electrical interface. This chip will drive the RS-232 Transmit Data and Request to Send signals from the 8251A TTL outputs and will convert the host RS-232 Receive Data and Clear To Send signals to 8251A input voltage levels. The MAX232 has the advantage of requiring a single five volt power supply and four external capacitors to generate ±10 volt RS-232 voltage swings; the SDK-86 implements the equivalent function using two power supplies (+5 and -12 volts) and a host of discrete components. An Intel i8254 will be used to derive the serial I/O baud rate clock which can be programmed from 110 to 19,200 bps.

To minimize the parts count, an Intel i8256 MUART was considered for implementation of the serial I/O, baud rate generation and interrupt controller. Unfortunately, the i8256 requires multiplexed address/data bus; the SDK-86 only offers a demultiplexed expansion bus. To re-multiplex the address/data bus would require extra logic and buffers defeating the purpose of using the i8256.

Four ROM/RAM sockets will be on the prototype board to increase the amount of program and data space (8K PROM, 2K RAM expandible onboard to 4K RAM) available for the debugger and applications code. The sockets will support either an additional 32K of RAM or an additional 16K of RAM and 16K of PROM.

3. Debugger Host Software Implementation

3.1 Overview

The debugger host software consists of ten subsystems:

- The Buffer Manager
- Communications software
- Terminal Handler
- LDP support routines
- Buffer Dispatcher
- Command Parser
- Procedures to execute the debugger commands
- Response Handler
- 8086 personality implementation
- Symbol Table support

The software uses several VMS system services to perform I/O, timeouts, and interprocess communication. Appendix F contains a summary of the these system services; detailed documentation for these services is contained in [18].

The Buffer Manager serves as an allocater of free storage organized as Buffer data structures. These structures are used by the debugger command handlers, communications software, and Terminal Handler. The Buffer Manager also exports routines which allow clients to maintain linked lists of buffers. The modules which implement the Buffer Manager are Buffer.c and Queue.c.

The communications software consists of two modules titled Datalink.c and Transport.c. These modules contain code to initialize and finalize serial connections with debugger target systems, transmit frames for the debugger host, and process received frames from target systems using the transport and datalink protocols described in sections 2.2.3 and 2.2.4.

The Terminal Handler is contained in the module Terminal.c. Its function is to queue terminal buffers to the VAX terminal driver for keyboard input from the user, to queue completed buffers to the Buffer Dispatcher subsystem, and to process user abort requests (control C).

The LDP support routines are procedures to generate and parse LDP messages. These include routines to serialize and deserialize LDP command headers, addresses, descriptors, bytes, words, and long words. Also included in these routines is a procedure to dump out LDP frames in symbolic form on the terminal to assist in protocol debugging. These routines are implemented in the modules LDPS.c and LDPP.c.

The Buffer Dispatcher is the main process in the debugger. This

subsystem forwards a completed line of terminal input to the Command Parser and target response messages to the Response Handler. It is contained in the main procedure in module DBA.c.

The Command Parser is responsible for parsing a line of user input and invoking the proper command handler to execute the command line. The output of the parsing phase is a data structure containing a command opcode and parameter strings copied from the command line. The parser is contained in modules DBB.c and DBT.mar.

The debugger command handlers contain the processor independent code that implement debugger commands. By convention, the handlers are procedures whose name starts with 'Exec' (e.g. ExecRead or ExecWrite). The following table lists the location of each command handler:

Module	Command(s)
DBA.c	Radix, Select, Synch
DBC.c	Breakpoint, Control
DBD.c	Read, Write
DBE.c	Download
DBH.c	Disassembler

The Response Handler processes unsolicited responses from the target systems. These are LDP Error messages which are transmitted by the target when a protocol violation is detected and LDP Exception messages which are generated when the target detects a runtime error or interrupt. The Response Handler is contained in the module DBF.c.

The 8086 personality implementation is contained in modules i8086.c, i8086A.c and i8086B.c. i8086.c contains routines to interconvert LDP addresses and strings, a procedure to convert LDP exception codes received from a target to strings, routines to serialize and deserialize data in and out of LDP communications buffers, and a procedure to print data from target memory. The module i8086A.c contains a disassembler for 8086 machine code. i8086B.c contains code to download Intel Object Module Format files into target memory.

The Symbol Table routines support the construction of a program symbol table for each target system and conversion of a symbolic address reference to a target memory address. The program symbol table is constructed while a program is being downloaded to a target. The symbol table is consulted during debugger commands which read or write target memory. The Symbol Table routines are contained in the file Symtab.c.

Figure 3.1 illustrates the flow of data through the debugger. A

completed user command buffer is passed to the Terminal Handler which enqueues the buffer on the terminal linked list. The Buffer Dispatcher dequeues the buffer and calls the Command Parser which parses the command and calls the appropriate command handler.

A message received from a target system is first stored in a communications buffer by the VMS terminal driver. The message is then processed by the datalink and transport communications software. If the message is an LDP frame (rather than a transport control message), it is enqueued on the appropriate target receiver linked list. The message is asynchronously dequeued by either a command handler which is processing a debugger command or by the Buffer Dispatcher which dequeues and processes unsolicited target messages.

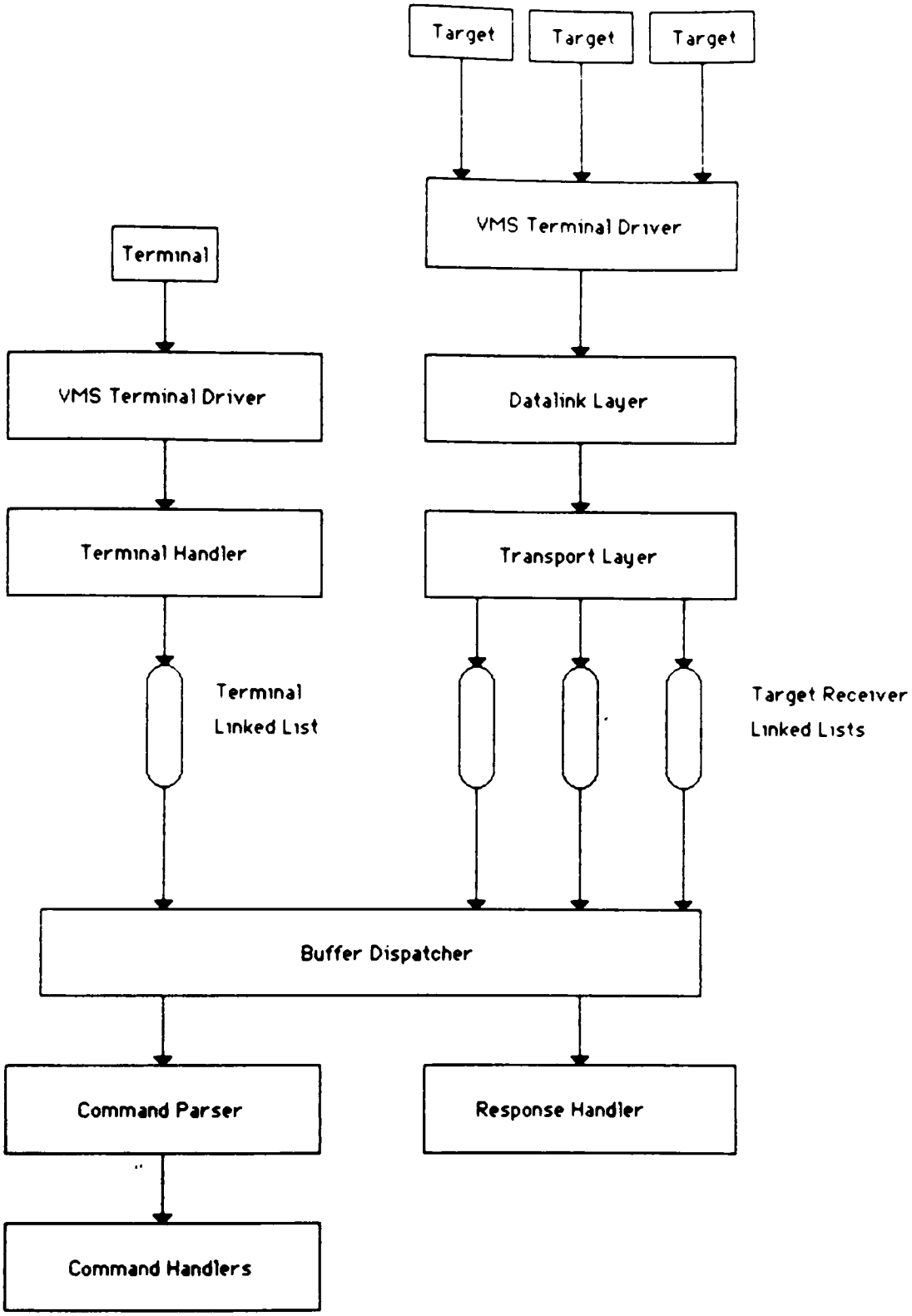


Figure 3.1. Debugger Data Flow

3.2 Procedures and data structures

3.2.1 Buffer Manager

Data Structures:

```
struct Queue_Head { /* in Queue.h */
    unsigned first;
    unsigned last;
    unsigned password;
    };

struct Buffer {        /* in Buffer.h */
    struct Buffer *frontlink;
    struct Buffer *backlink;
    unsigned password;
    struct DataLink_Channel *channel;
    struct IOSB iosb;
    unsigned char state;
    unsigned parseIndex;
    unsigned frameL;
    unsigned maxFrameL,
    unsigned char stx;
    unsigned char frame [1];
    };
```

Exported procedures:

```
struct Queue_Head *Queue_Init ();
Queue_Enqueue (queue, buffer);
struct Buffer *Queue_Dequeue (queue);

Buffer_Init (nBuffers, bufferSize);
struct Buffer *Buffer_Allocate ();
Buffer_Deallocate (buffer);
Buffer_IsBuffer (buffer);
```

The Queue routines are used to manipulate linked lists of Buffer elements. The function of Queue_Init is to create a new linked list. It returns a pointer to a new instance of the Queue_Head structure which contains the head and tail pointers of the linked list and the *password* field. The returned value is the *queue* parameter used in the other Queue routines. The *password* field is used to validate a Queue_Head parameter passed to other Queue routines at runtime.

Queue_Enqueue adds the specified Buffer element to the tail of the designated linked list. Queue_Dequeue dequeues the first buffer on the specified linked list and returns a pointer to it; if the queue is empty, Queue_Dequeue returns a NULL pointer. The Enqueue

and Dequeue routines are implemented using the VMS library routines LIB$INSQTI and LIB$REMQHI and are therefore atomic [19].

The Buffer routines allow a client to allocate and deallocate Buffer elements from a global pool of buffers created when the debugger starts up. This pool is used by the command handlers, the communications software and the Terminal Handler. The Buffer routines use the Queue package to maintain the linked list of free buffers.

The Buffer_Init procedure creates the buffer pool; the number of bytes in each buffer available for LDP messages, communications control messages or terminal input is determined by the *bufferSize* parameter. The total number of buffers initially allocated for the pool is set by the *nBuffers* parameter.

Buffer_Allocate returns a pointer to a free Buffer element; if a free buffer is unavailable, the procedure returns a NULL pointer. Buffer_Deallocate returns the specified Buffer element back to the free pool. Buffer_IsBuffer allows a client to verify that the buffer parameter is a legal Buffer pointer.

The Buffer data structure contains fields used by various debugger subsystems. The *frontLink* and *backLink* fields are used by the Queue routines for linked list maintenance. The *password* field is used by Buffer_IsBuffer to perform simple runtime type checking of buffer pointer parameters. The *state* field maintains the element's allocation state; it is used to verify that buffers passed to the Buffer_Deallocate routine haven't already been deallocated. The *parseIndex* field is used by the LDP serialize and deserialize routines to maintain the position of the next byte to remove or append in the Buffer's *frame* field. The *frameL* field stores the actual number of data bytes in the frame field of a Buffer. The datalink software sets this field when a packet is received; the debugger host software sets this field when building packets to be transmitted to a target system. The *maxFrameL* contains the number of bytes allocated for the frame field; it is initialized by the Buffer_Init routine.

3.2.2 Datalink Layer

Data structures:

```
struct DataLink_Connection {  /* in DataLink.h */
   int password;
   int chanNum;
   int nCRCErrors, nRunts, nRx, nTx;
   char *clientParam;
   char *devName;
   };
```

Exported procedures:

```
struct DataLink_Connection *Datalink_CreateChannel (
   devName, status, clientParam)
Datalink_DeleteChannel (channel)
Datalink_GetMaxBufferSize ()
Datalink_Queue (channel, buffer)
Datalink_Stats (channel)
Datalink_Transmit (channel, buffer)
```

Imported procedures:

```
Transport_RxHandler (clientParam, buffer)
Transport_TxHandler (clientParam, buffer)
```

Private Procedures:

```
RxAST (buffer)
TxAST (buffer)
```

The function of Datalink_CreateChannel is to create a full duplex serial datalink channel. The transport layer calls this procedure, passing the name of the serial channel (e.g. "TXI7:"), a pointer to a completion status word, and the *clientParam* longword which the datalink layer passes back to the transport layer on upcalls for received and transmitted frames. The transport layer uses this parameter as a pointer to transport layer connection information. The status values returned from this call are the same as those returned from the VMS SYS$ASSIGN library call. Datalink_CreateChannel returns a pointer to an instance of the Datalink_Connection structure; this pointer is the channel parameter used in most of the other Datalink layer calls.

Datalink_DeleteChannel cancels pending I/O on the specified channel and returns the storage used for maintenance of the channel. It is invoked when the debugger is finishing up.

Datalink_GetMaxBufferSize returns the number of overhead bytes used by the datalink layer. This routine is used by the transport layer to determine the position of the transport

encapsulation.

Datalink_Queue queues a buffer for frame reception at the specified channel. An arbitrary number of buffers may be queued at a channel. Data will be lost if a frame is received at the VAX and no buffers are in the channel's receive pool.

Datalink_Stats prints out the statistics for the specified datalink channel. Included in the statistics are the number of frames received and transmitted and error counts such as framing errors, number of runt frames and CRC errors.

Datalink_Transmit asynchronously transmits a client's frame on the specified the serial I/O channel. The client's data is copied into a new buffer with DLE stuffing, framing characters, and two CRC bytes, and queued for transmission using the VMS QIO call. On completion of the transmission, the VMS operating system invokes the TxAST procedure using the AST (Asynchronous Service Trap) mechanism. TxAST upcalls the transport layer Transport_TxHandler procedure, passing a pointer to the buffer that was transmitted and the transport layer's clientParam which identifies the transport connection associated with the buffer.

Received frames are handled by the RxAST procedure which is also invoked via the VMS AST mechanism. The RxAST routine verifies that the received frame's size is valid, strips off DLE characters, and checks the frame's CRC. If all is well, the Transport_RxHandler procedure is upcalled with two parameters—a pointer to the received frame and the transport clientParam. If any errors are detected in the frame, the buffer is requeued for frame reception.

3.2.3 Transport Layer

Data structures:

```
struct Transport_Connection { /* in Transport.h */
    unsigned password;
    struct DataLink_Channel *channel;
    unsigned char transportState;
    unsigned char nr, ns;
    unsigned char nUA;
    struct Buffer *txBuffer;
    int txEventFlag;
    unsigned retransmitCount;
    int nRxBuffersQueued;
    int uaFlag;
    char *clientParam;
    };
```

Exported procedures:

```
struct Transport_Connection *Transport_CreateConnection (
    devname, status, clientParam)
int Transport_GetMaxBufferSize ()
Transport_ReturnBuffer (connection, buffer)
Transport_RxHandler (connection, buffer)
Transport_Synch (connection)
boolean Transport_TransmitFrame (connection, buffer)
Transport_TxHandler (connection, buffer)
```

Imported procedures:

```
LDP_Handler (clientParam, buffer)
```

The debugger host calls Transport_CreateConnection to set up a transport layer connection with a target. This procedure is passed a serial channel name string, a pointer to a completion status longword, and a *clientParam* value which is passed back to the debugger's LDP_Handler on received frame upcalls. The debugger host uses the *clientParam* value to associate a transport connection with a data structure maintaining debugger level information for each target.

To establish the connection, Transport_CreateConnection allocates and initializes storage for the Transport_Connection data structure, creates a datalink channel with Datalink_CreateChannel, and queues an initial quota (rxBufferQuota) of receiver buffers obtained from the global buffer pool. Transport_CreateConnection returns a pointer to a new instance of the Transport_Connection; this pointer is the *connection* parameter in the other Transport calls. The debugger host follows a call to Transport_CreateConnection with a

call to Transport_Synch to synchronize the host's sequence numbers with the target's sequence numbers.

Transport_GetMaxBufferSize returns the number of overhead bytes used by both the transport layer and the datalink layer This call is used by the debugger host to determine the portion of the *frame* field of the Buffer structure available for the LDP protocol.

Transport_ReturnBuffer requeues a receive buffer to a tranport connection. When the connection's receive buffer count (nRxBuffersQueued) rises above a threshold (rxBufferThresh), a transport receiver ready message is sent to the target indicating that the host is able to receive target transmissions.

When the datalink layer receives a frame with correct framing and CRC, it upcalls Transport_RxHandler. This procedure is passed the clientParam value which was passed to the DataLink_CreateChannel procedure and a pointer to the received buffer. The clientParam value references the Transport_Connection data structure associated with the datalink channel which received the frame.

Transport_RxHandler selects one of three routines depending on the type of the received frame. For information format frames, the *nr* field of the control byte is processed as an ack reply. This involves comparing the sequence number of the connection's pending transmission buffer with the received *nr* number. If the *nr* number is a valid acknowledgement, the transmission buffer is returned to the buffer pool. Next, depending on the connection's receiver buffer quota, one of three types of acknowledgement messages is returned to the target.

If the buffer quota is below a threshold, a RNR (receiver not ready) message containing the *nr* field of the last successfully received frame is transmitted. This informs the target that the frame was discarded because the host had a transient buffer shortage and the frame should be retransmitted after a delay.

If the reception of this frame drops the connection's receiver buffer quota below a threshold, a RNR acknowledge message is sent. This indicates that the host has successfully received the frame but has no resources to receive another frame.

If the buffer quota is above the threshold, a RR message is transmitted which acknowledges the frame and informs the target that the host is able to receive another frame. Frames which are in sequence are passed to the debugger using an upcall to the LDP_Handler procedure.

When processing a supervisory frame, Transport_RxHandler first verifies that the type of the frame is RR. Then, the *nr* field of the control byte processed as an ack reply.

Unnumbered frames received from the target are verified to be

either UA (Unnumbered Acknowledge) or RIM (Request Initialization Mode) type messages. UA frames are handled by incrementing a count of received unnumbered acks maintained for each connection and setting a VMS event flag. The count and the flag are used to signal processes waiting for UA messages in the Transport_Synch procedure. The transport implementation does not respond to target initiated RIM messages.

Transport_Synch attempts to establish a circuit with a target system. It issues a unnumbered format RIM frame and waits two seconds for a UA from the target system. If successful, Transport_Synch resets the host's sequence numbers and returns TRUE; otherwise it retries two times. If unsuccessful three consecutive times, Transport_Synch returns a FALSE value.

Transport_TransmitFrame asynchronously transmits a frame to the target system. This call encodes an information format control byte containing the connection's current *nr* and *ns* sequence numbers and stores the control byte into the frame's transport layer encapsulation field. Next, a pointer to the frame is stored in the connection's *txBuffer* field. This field maintains a pointer to the connection's unacknowledged transmit buffer. The frame is then transmitted using the Datalink_Transmit call. Transport_TxHandler is upcalled by the datalink layer when the frame transmission is complete. If the frame transmitted is an information format message, an ack timer is set for the buffer; otherwise the frame is returned to the Buffer manager.

If an ack is not received for the transmitted frame in a three second timeout period, private procedure TimerAST is called by the VMS AST mechanism. This procedure bumps the frame's retransmit count and retransmits the transmit buffer. If four retransmissions occur without an ack from the target, the transport connection is closed; it must be explicitly reopened using Transport_Synch.

3.2.4 Terminal Handler

Exported Procedures:
- Terminal_Cleanup ()
- Terminal_Init ()
- Terminal_QueueBuffer (buffer)

Imported Procedures:
- db_EnqueueTerminalBuffer (buffer)
- db_SetAbort ()

Private Procedures:
- Terminal_AST (buffer)
- Terminal_ControlC_AST ()
- Terminal_SetControlC_AST ()

The Terminal Handler implements procedures for queuing Buffer data structures to the VMS terminal driver for user input, queuing completed buffers for interpretation by the debugger, and handling control-C abort requests. All debugger terminal output is performed using the library routines printf, puts, and putc.

Terminal_Init is called by the startup processing of the debugger. It establishes an I/O channel for the terminal input device "SYS$INPUT", queues an initial quota of terminal buffers to the VAX terminal driver, and binds the procedure Terminal_ControlC_AST as the handler for control-C's using the Terminal_SetControlC_AST procedure.

Terminal_Cleanup is called by the cleanup code of the debugger in response to a user quit command. It cancels all pending I/O (including queued input buffers) for the terminal using the VMS SYS$CANCEL system call.

Terminal_QueueBuffer posts the specified Buffer data structure for terminal input using the VMS SYS$QIO system call. When a buffer is queued, the AST handler is set to be the Terminal_AST procedure. The Terminal_AST procedure is called by VMS when the user enters a complete line of terminal input. Terminal_AST appends the NUL character onto the end of the buffer so the frame field of the Buffer conforms to the C convention of string termination and calls db_EnqueueTerminalBuffer to enqueue the buffer for processing by the Buffer Dispatcher.

The Terminal_ControlC_AST is called by the VMS terminal driver when the user types a control-C. When invoked, this procedure notifies the debugger host software of the abort request with a call to db_SetAbort. It then reenables the Terminal_ControlC_AST procedure using the Terminal_SetControlC_AST procedure.

3.2.5 LDP Support Routines

Data structures:

```
struct LDP_Address {  /* in LDP.h */
    unsigned char tag;
    unsigned modeArg;
    unsigned mode;
    unsigned idHi;
    unsigned idLo;
    unsigned offHi;
    unsigned offLo;
    }

struct LDP_Descriptor {  /* in LDP.h */
    unsigned modeArg;
    unsigned mode;
    unsigned idHi;
    unsigned idLo;
    }

struct LDP_CommandHeader {  /* in LDP.h */
    unsigned commandLength;
    unsigned commandClass;
    unsigned commandType;
    }
```

Exported Procedures:

```
LDP_BytesInAddress (ldpAddr)
LDP_CopyAddress (srcAddr, destAddr)
LDP_DeserializeInit (buffer)
unsigned char LDP_DeserializeByte (buffer)
LDP_DeserializeDescriptor (buffer, descrip)
LDP_DeserializeHeader (buffer, descrip)
unsigned LDP_DeserializeLong (buffer)
unsigned LDP_DeserializeWord (buffer)
LDP_PrintFrame (buffer)
LDP_SerializeInit (buffer)
LDP_SerializeAddress (buffer, ldpAddr)
LDP_SerializeByte (buffer, byteValue)
LDP_SerializeDescriptor (buffer, descrip)
LDP_SerializeHeader (buffer, header)
LDP_SerializeLong (buffer, longValue)
LDP_SerializeWord (buffer, wordValue)
```

These procedures serialize debugger data structures into LDP communications buffers and deserialize bytes in received

communications buffers into debugger data structures. They are used when generating LDP messages for transmission to a target and when parsing received frames. These routines are provided to insulate the debugger from the actual format of data in LDP messages. They are also useful because they provide bounds checking when appending data onto a buffer or removing data from a buffer

The LDP_Deserialize procedures remove sequential bytes from the specified buffer's *frame* field to build debugger data structures. The current position in the buffer is maintained with the Buffer *parseIndex* field and must be initialized using the LDP_DeserializeInit procedure before using other LDP_Deserialize procedures.

The LDP_Serialize procedures serialize debugger data structures into LDP format by appending the proper bytes onto the *frame* field of buffer parameter. As with the Deserialize routines, the LDP_SerializeInit procedure must be called to initialize the parseIndex field before calling other LDP_Serialize routines.

The debugger host software uses the LDP_Address structure to represent target memory, register, and I/O port addresses. The LDP_SerializeAddress and LDP_DeserializeAddress routines convert between LDP_Address structures used by the debugger and the encoding of LDP addresses in a LDP message.

The LDP_CommandHeader structure represents the header portion of LDP messages. The structure consists of a *commandLength* field which contains number of bytes in the entire LDP message, a *commandClass* field, and *commandType* field. LDP_SerializeHeader and LDP_DeserializeHeader convert between the LDP_CommandHeader data structures used by the debugger and an LDP command headers in an LDP message.

The LDP_Descriptor data structure represents LDP protocol descriptors on the VAX. This structure is used to refer to breakpoints on the target system. The LDP_SerializeDescriptor and LDP_DeserializeDescriptor routines translate between LDP_Descriptor structures used by the debugger and the format of LDP descriptors in an LDP message.

The LDP_SerializeByte, LDP_DeserializeByte, LDP_SerializeLong, LDP_DeserializeLong, LDP_SerializeWord, LDP_DeserializeWord are used to interconvert between VAX byte, longword, and word data types and bytes in an LDP message.

LDP_PrintFrame prints out a frame's command length, command class, comand type and parameters in symbolic form. This procedure is used when debugging code which generates or processes LDP frames.

3.2.6 Buffer Dispatcher

Imported procedures:
db_Exec (buffer)
db_ResponseHandler (buffer)

Private Procedures:
main ()

The Buffer Dispatcher is the main process of the debugger. Its job is to dequeue terminal buffers with user input from the terminal linked list and pass them to the Command Parser and to dequeue received target messages and pass them to the Response Handler.

The use of a single process serializes the processing of buffers arriving asynchronously from several sources. This helps eliminate critical sections in the debugger and precludes the necessity of protecting resources which could be in demand simultaneously in a multiprocess implementation. For example, in a multiprocess implementation, two processes may try to write to the terminal at once, leading to undecipherable output.

The debugger uses a counter variable called *eventCount* to maintain the number of pending buffers that require processing by the Buffer Dispatcher. *eventCount* is incremented when a terminal buffer is enqueued on the terminal linked list with the db_EnqueueTerminalInput procedure or when a target message is queued on a target receiver linked list by the LDP_Handler; it is decremented each time the Buffer Dispatcher removes a message from a linked list and processes it. Therefore, while *eventCount* is non-zero, the Buffer Dispatcher remains active and dequeues pending messages. When *eventCount* goes to zero, the Buffer Dispatcher process enters a dormant state; VMS event flags are used to awaken the Buffer Dispatcher when a new terminal or target buffer is enqueued.

3.2.7 Command Parser

```
Data structures:
   struct db_ParseData {
      unsigned opcode;
      unsigned flags;
      unsigned nValues;
      char addrString1 [80];
      char addrString2 [80];
      char countString [80];
      char filename [80];
      char targetname [80];
      char values [80];
      struct {
         unsigned char first;
         unsigned char last;
         } vIndex [40];
      };

Exported procedures:
   boolean db_Exec (buffer)
   struct db_ParseData *Parse_GetData ()
```

db_Exec is a procedure contained in the module DBB.c which parses a user command string and calls a debugger command handler procedure to execute the parsed command. db_Exec is called by the Buffer Dispatcher when a line of user input has been dequeued from the terminal linked list. The parse step is performed using the VMS LIB$TPARSE subroutine. The module DBT.mar contains a description of a finite state machine which defines the debugger's command grammer and action procedures which copy command line parameters into appropriate fields of the db_ParseData record. If the call to LIB$TPARSE returns with no errors (i.e. the string is syntactically correct), db_Exec calls the appropriate command handler routine to execute the user's command. Otherwise, db_Exec prints out a syntax error message and prints the portion of the input string which was unparsable. db_Exec returns a TRUE if the debugger quit command was entered; this return value informs the Buffer Dispatcher that the user wishes to exit from the debugger.

The Parse_GetData procedure returns a pointer to the db_ParseData record contained in the DBT module and set up by the finite state machine action procedures during a parse. Debugger command handler procedures call this routine to obtain parsed parameters from the command line.

After a parse, the *opcode* field in the db_ParseData structure is

set to a constant that corresponds to the command that was entered. The valid opcode constants are contained in the file db.h.

The *flags* field of the db_ParseData are interpreted as an array of 32 booleans which contain modifier information for the command opcode. The currently valid modifier flags are in the file db.h. The *flagWrite* bit is set for data transfer commands which write into target memory; it is clear for read data transfer commands. The *flagBPClear*, *flagBPList*, and *flagBPSet* flags are set for breakpoint clear, list and set commands respectively.

The *nValues* field contains the count of numeric data items to be written to target memory for the debugger write memory, I/O port, and register commands. It is used in conjunction with *values* string and the *vIndex* structure also contained in the db_ParseData structure. The *values* string is a copy of the string containing the data items for write commands; *vIndex* is an array of structures delimiting individual data items in the *values* string. For example, for the command "byte 0 = 1 2 3 4":

```
nValues = 4
values string = "1 2 3 4"
vIndex array = ((0, 0) (2, 2) (4, 4) (6, 6))
```

The *addrString1* field receives a copy of the address parameter for debugger commands containing one target address specifier These are the memory, I/O port and register commands and the disassembler command. The *addrString2* field gets a copy of the second address parameter for debugger commands containing two target addresses. Currently, no command is in this format.

The *countString* field receives a copy of the count parameter for the debugger memory, I/O and register commands and the disassembler commands. The *filename* string is set to the VMS filename specified in the debugger's download command. Commands which refer to a target system's name copy the name string into the *targetname* field of the db_ParseData record.

3.2.8 Disassembler Command Execution

```
Data structures:
   struct Stream {
      struct Buffer *b;
      struct db_Target *target;
      int nLeft;
      boolean error;
      }

Exported procedures:
   ExecASM (target)

Private procedures:
   StreamInit (target, sH, startAddr, streamLength)
   StreamGetByte (sH)
   StreamNewBuffer (sH)
   StreamFinish (sH)
```

The module DBG.c is the processor independent portion of the debugger's disassemble command. It exports the ExecASM procedure which is invoked by the Command Parser. ExecASM first sets up the target start address and instruction count by converting the *addrString1* and *countString* fields of the db_ParseData record into an LDP address and integer. Next, a serial byte stream is established using the StreamInit call. The Stream routines allow the debugger to serially read bytes from the target system without regard for packet boundaries. The last set up step is to prime a sixteen code byte buffer using the StreamGetByte routine. Then, for each instruction to be disassembled, ExecASM prints out the current target address, calls the target's disassembler procedure, prints out the disassembled instruction, and replenishes the code buffer. When finished with the loop, the byte stream is closed and ExecASM returns.

3.2.9 Download command execution

Exported procedures:
ExecDownload (target)

Imported procedures:
personality->Download (target, filename)

The module DBE.c contains the ExecDownload procedure which implements the target independent portion of the debugger's download command. This procedure invokes the download procedure for the named target, passing it the target system to download and the name of a VMS file to write to target memory The remainder of the download command is implemented in the target's processor dependent personality module as the format of object files is machine dependent.

3.2.10 Target Read Command Execution

Exported procedures:
db_SendRead (target, ldpAddr, count)
ExecRead (type, space)

Private Procedures:
HandleReadResponse (target, type, space)

The ExecRead procedure implements the debugger's memory, I/O, and register read commands. ExecRead is invoked by db_Exec after a parse of the debugger's *byte*, *char*, *long*, *pointer*, and *word* memory read commands, the *register* read command, and the *port*/*wport* I/O read commands. ExecRead is passed two parameters—the space parameter which specifies memory, register, or I/O reads and the type parameter which specifies how the data is to be displayed (i.e. byte, char, long, pointer, or word).

The first action ExecRead takes is to convert the start address string parameter into an LDP address data structure using the db_StringToLDP_Address utility. Then the count string parameter is converted into an integer using the db_Eval routine. Next, ExecRead computes the number of bytes required to be read from the target using the count and the target's Bits procedure. The Bits procedure is a processor dependent routine which returns the number of bits a processor uses to encode one of the debugger's five generic types. ExecRead then calls the db_Read routine which transmits an LDP Read message requesting the target to return the specified amount of data from memory, processor registers, or I/O ports.

The HandleReadResponse procedure is called after transmission of the LDP Read request. This procedure processes target responses to the host's LDP Read request which consist of LDP ReadData packets containing the requested target data and an LDP ReadDone packet transmitted when all requested data has been sent to the host. ReadData packets are handled by deserializing the data portion of the frame using the processor's Deserialize procedure and printing each datum with the processor's PrintItem procedure.

3.2.11 Target Write Command Execution

Exported procedures:
ExecWrite (type, space)

Private Procedures:
ValueString_Init ()
ValueString_NextValue ()

ExecWrite handles the debugger commands which write command line data into target memory, registers, or I/O ports. These are the *byte*, *char*, *long*, *word*, *register*, *port*, and *wport* commands. ExecWrite is passed a type parameter which indicates how the data on the command line is to be interpreted (i.e. byte, char, long, or word) and a space parameter which specifies which target address space is to be written to (i.e. memory, register, or I/O). In addition, ExecWrite uses the *nValues* and *values* fields of the db_ParseData data structure to access user's command line data.

ExecWrite first computes the number of data items on the command line. This is equal to the length of the *values* string when the command line data is a character string; it is equal to the *nValues* field of the db_ParseData structure when the data is a list of numeric items. Then, using this count and the target's Bits procedure, it computes the number of bytes in an LDP message needed to store the serialized command line data. This number will be used when generating the encapsulation for the LDP message. Following this step, the target start address is converted from string format to an LDP data structure using the db_StringToLDP_Address procedure.

The presence of the count string parameter on the command line implies that the user wishes to fill a block of target memory with the command line data. For this case, ExecWrite generates and serializes the encapsulation for an LDP RepeatData message. This type of LDP message is used to fill target memory with a pattern. If no count string parameter is present, ExecWrite generates and serializes the encapsulation for an LDP WriteData message.

The data to be written into target memory follows the RepeatData or WriteData encapsulation. ExecWrite generates the target data by converting each value substring on the command line into a VAX longword by calling the ValueString_NextValue procedure and appending the longword onto the LDP message using the target's Serialize command. When all the data on the command line has been serialized, it is transmitted to the target using the db_TransmitCurrent utility.

3.2.12 Control Command Execution

Exported procedures:
ExecControl (target, controlType)

Imported procedures:
db_Transmit (target, buffer)

The ExecControl procedure implements the debugger's *go*, *go from*, *step*, and *halt* commands. The controlType parameter used in conjunction with the db_ParseData structure specifies whether the specified target should be started, continued, halted, or single stepped. The values that the controlType parameter may take are the constants *controlTypeGo*, *controlTypeHalt*, and *controlTypeStep* in the file db.h.

For the debugger's *go* and *go from* commands, ExecControl checks the db_ParseData structure for the presence of a target start address entered on the command line. This address string is copied into the *addrString1* field of the db_ParseData structure by the Command Parser. If a start address string is present, it is converted into an LDP Address structure using the db_StringToLDP_Address procedure and serialized into an LDP Start message. If no start address is specified, the debugger assumes that the user wishes to continue the target from its current program counter. In this case, an LDP Continue message is constructed.

For the debugger's step and halt commands, the controlType parameter passed to ExecControl is *controlTypeStep* and *controlTypeHalt*, respectively. For these commands, ExecControl builds an LDP Step or LDP Stop message.

Once the LDP message has been generated, it is transmitted to the target using the db_Transmit utility. Error response messages from the target (e.g. target already started or stopped) are handled by the Response Handler.

3.2.13 Breakpoint Command Execution

Exported procedures:
ExecBreakpoint ()

Imported procedures:
ValueString_Init ()
ValueString_NextValue ()
ValueString_NextString ()

Private Procedures:
BPClear (target)
BPList (target)
BPPrint (target, ldpAddr, descrip)
BPPrintList (target, respB, firstResponse)
BPSendCreate (target, ldpAddr)
BPSendDelete (target, id)
BPSet (target)
struct Buffer *GetDoneMsg (target, commandType)

This group of procedures implement the debugger commands which set, clear, and list target execution breakpoints. After a breakpoint command is parsed, the db_ParseData *opcode* field contains the constant *opBP* (defined in db.h); the parser identifies the type of breakpoint command using bits in the db_ParseData *flags* field. ExecBreakpoint dispatches to BPClear, BPList or BPSet depending on the state of the *flags* field.

The function of BPClear is to clear the breakpoints refered to by the list of id's on the command line. The id's are substrings in the *values* string of the db_ParseData structure; the *vIndex* array of the db_ParseData structure delimits the substrings.

BPClear performs three actions for each id on the command line. First, it converts the next id substring to an integer using the ValueString_NextValue utility. Next, it invokes the BPSendDelete procedure which transmits an LDP Delete message to the target. Finally, BPClear calls the GetDoneMsg procedure which waits for an LDP DeleteDone message acknowledging the removal of the breakpoint.

BPList is responsible for transmitting a message requesting the target to respond with LDP messages containing its current execution breakpoints. It then prints the id, address pairs contained in each target response messages.

BPList first constructs and transmits an LDP ListBPs message using the LDP serialize routines and the db_Transmit utility. Then, it enters a loop containing three steps. First, a target response message is dequeued from the target receiver linked list with using

the db_DequeueRx routine. Then, the message is verified to be a valid response to an LDP ListBPs message. Finally, the BPPrintList procedure is called to print out the address and id of each breakpoint in the response message. The loop is exited when a response message is received that contains a null id, address pair list.

The function of BPSet is to insert execution breakpoints at the addresses on the command line. After the parse, the addresses are substrings in the db_ParseData *values* string; these substrings are delimited by the *vIndex* array in the db_ParseData structure.

BPSet performs the following actions for each address substring in the *values* string. First, the next address substring is copied to a local string using the ValueString_NextString procedure. This string is converted to an LDP_Address data structure using the db_StringToLDP_Address procedure. Next, BPList calls the BPSendCreate procedure which transmits an LDP message requesting the creation of an execution breakpoint. After this message is sent, the GetDoneMsg procedure is invoked to wait for an acknowledgement from the target. If the acknowledgement arrives and is in the proper format, the BPPrint procedure is called to print out the id that the target assigned to the newly created breakpoint.

3.2.14 Miscellaneous Command Handlers

Exported procedures:
ExecRadix (newRadix)
ExecSelect (targetName)
ExecSync (targetName)

Imported procedures:
db_Eval ()
struct db_Target *db_FindTarget ()
Transport_Synch ()

These procedures, which are contained in the module DBA.c, implement the debugger's *radix*, *select*, and *synch* commands.

ExecRadix is passed a numeric string containing the new default debugger radix. If the string is empty, ExecRadix prints out the debugger's current radix. Otherwise, the string is converted to an integer with the db_Eval utility. If the integer is a valid radix (i.e. octal, decimal, or hex), the variable db_DefaultRadix which stores the current default radix is set to the new value; otherwise an error message is printed.

ExecSelect is passed the *targetName* string containing the name of the new default target system. The default target system is maintained in the variable db_CurrentTarget. If the string is empty, ExecSelect prints out the target name, processor type, and serial channel for each target system instantiated when the debugger was started. If *targetName* is not empty, ExecSelect uses the db_FindTarget utility to search the target linked list for the desired target name. If found, db_CurrentTarget is set to the new target system; otherwise an error message is printed.

ExecSync is used to resynchronize a target system with the host. This may be necessary if a target system crashes and is restarted during a debug session. If the *targetName* string is empty, ExecSync calls the Transport_Synch using the default target system's transport connection; otherwise ExecSync calls the db_FindTarget procedure to lookup the specified target name. If the target exists in the linked list, Transport_Synch is called with that target's transport connection. If the target is not found, an error is reported.

3.2.15 Response Handler

Exported procedures:
db_ResponseHandler (target, buffer)

Imported procedures:
LDP_DisplayFrame ()
personality->ExceptionToString ()

The Response Handler is invoked by the Buffer Dispatcher when an unsolicited message is received from a target. The Buffer Dispatcher passes db_ResponseHandler pointers to the target system which transmitted the message and the received message.

The Response Handler only processes LDP Error and LDP Exception messages. All other target responses occur in response to a host request and should be handled by command handler which prompted the target to send a message. If the Response Handler encounters a message not in the Error or Exception format, it will dump it out using the LDP_DisplayFrame routine.

Error messages are handled by printing out a string corresponding to the error code contained in the Error message. The most commonly encountered error messages are listed below:

Message	Cause
Bad ID	Tried to delete a nonexistent breakpoint
Out Of Resources	Tried to set too many breakpoints
Already Created	Tried to set a breakpoint at a location which already had a breakpoint
Already Started	Tried to start a target which was already running
Already Stopped	Tried to stop a target which was already stopped

Any other error message results from a bug in either the host or target software.

Exception messages are handled in three steps. First, the exception address is deserialized from the Exception message. Next, a message reporting that an exception occured is reported for the target. Finally, the target's LDP_ExceptionToString is invoked and the returned exception string is displayed.

3.2.16 Intel 8086 Processor Personality

Data Structures:

```
struct db_Personality {  /* in db.h */
    char *name;
    unsigned nRegisters;
    int (*StringToLDP_Address) ();
    int (*LDP_AddressToString) ();
    int (*LDP_ExceptionToString) ();
    int (*nBits) ();
    int (*Disassemble) ();
    int (*Deserialize) ();
    int (*Download) ();
    int (*PrintItem) ();
    int (*Serialize) ();
    };
```

Exported procedures:

```
struct db_Personality i8086_Instance ()
```

Private Procedures:

```
Bits (type, nItems)
Deserialize (buffer, space, type, ldpAddr)
i8086_Disassemble (codeByteBuffer, outString, ldpAddr)
i8086_Download (target, filename)
LDP_AddressToString (ldpAddr, string, status)
LDP_ExceptionToString (buffer, ldpAddr, string)
PrintItem (value, type)
Serialize (buffer, space, type, value)
StringToLDP_Address (string, space, type, ldpAddr, status)
```

The modules i8086.c, i8086A.c, and i8086B.c implement the debugger's processor dependent routines for the Intel 8086 microprocessor. The routines are accessible through pointers in the db_Personality record which is created with a call to i8086_Instance. In addition to the processor dependent procedures, the db_Personality record contains the *name* and *nRegisters* fields. The *name* field contains the name of the processor; the *nRegisters* field contains the number of processor registers.

To support a new processor, implementations of the private procedures listed above would be coded. In addition, a public procedure to create instances of the db_Personality record for the new processor is necessary. Finally, db_AddTarget in the module DBA.c would be altered to recognize the new processor's name and instantiate a db_Personality record for the new processor. For example, suppose a personality implementation for the 8085

processor was written and the name of the procedure to create an instance of the 8085 personality was "i8085_Instance" db_AddTarget would be altered as follows:

```
...
    if (EquivalentString (processorName, "i8086"))
    newTarget->processor = i8086_Instance ();
else if (EquivalentString (processorName, "i8085"))
    newTarget->processor = i8085_Instance ();
else ...
```

The i8086_Instance procedure is invoked during debugger startup for each i8086 processor in the system to be debugged. It allocates space for the db_Personality structure, fills in the *name* and *nRegisters* field, and sets the procedure pointer fields to the addresses of private procedures in i8086.c, i8086A.c, and i8086B.c. Its return value a pointer to the newly created db_Personality record.

The file i8086.h contains definitions which are used by both the host and target. These include constants used to address processor registers in an LDP Address and constants used to encode 8086 exceptions in LDP Exception messages.

The Bits procedure returns the number of bits the required to encode *nItems* objects of the specified debugger data type in an LDP buffer. This procedure is used by the debugger to compute the number and size of LDP messages required when transmitting data to the target. The following table shows the values returned by the i8086 Bits procedure:

Type	Bits/Item
char	8
byte	8
long	32
pointer	32
word	16

The function of the Deserialize procedure is to return a value of the specified type by unpacking bytes from an LDP buffer. The LDP protocol specifies that data is stored in communications buffers by packing the target's bits from increasing memory locations. The Deserialize procedure serves two functions: to unpack bits from an LDP buffer and to reconstruct a VAX longword from the bits contained in an LDP buffer.

For the i8086 personality, the Deserialize routine uses the LDP Deserialize routines to unpack eight bit units from LDP buffers. For long, pointer, and word data types it permutes the bytes to form

an object in VAX format. This permutation is necessary since the 8086 stores multibyte data structures in memory in a different order than they are stored in an LDP buffer

After the data is unpacked, the Deserialize routine adjusts the LDP Address pointed to by the *ldpAddr* parameter to address the next item in the target's memory.

The function of the i8086_Disassemble routine is to return a string containing the mnemonic form of one target machine instruction. The debugger calls this routine from ExecASM—the device independent portion of the disassemble command. i8086_Disassemble is passed the *codeBytesBuffer* which points an array of target memory bytes to be disassembled, the *outString* which is a pointer to a string to receive the symbolic form of the instruction, and the *ldpAddr* parameter which is a pointer to a target address. On entering the procedure, this is the address of the first byte of the instruction to be disassembled; on returning from i8086_Disassemble, it must be set to the address of first byte of the next instruction in target memory.

The i8086_Disassemble routine is implemented using a table which is indexed by the first byte of the machine instruction. Each table entry contains an index into the *mnemonics* string array and a byte specifying the format of the instruction's operands. i8086_Disassemble first strips off 8086 prefix bytes (e.g. lock, segment prefixes, etc.) and appends their mnemonics to the *outString* array. Next the opcode field is parsed and an instruction mnemonic is appended to the *outString* array. Finally, the operands are converted to symbolic form in the DoOperands procedure.

i8086_Download downloads the specified VMS file into a target's memory. i8086_Download expects the file to be in Intel OMF (Object Module Format).

i8086_Download repeatedly reads OMF records until a MODEND (module end) record is encountered. The records processed by the download procedure are PEDATA (Physical Enumerated Data) and PIDATA (Physical Iterated Data). PEDATA records contain a sequence of bytes to be loaded at a specified address (e.g. code bytes to be downloaded into the target).

The PIDATA records contain data specifying a pattern to be written into target memory. These records are emitted by compilers and assemblers to preset blocks of program memory. To decrease VAX-target communications traffic, the download procedure attempts to convert the PIDATA records into LDP RepeatData messages rather than expanding them into sequences of LDP Write messages. The RepeatData message cannot be used when the pattern's length is greater than the size of an LDP buffer.

LDP_AddressToString is responsible for converting an LDP_Address

data structure into a string. By convention, the 8086 target encodes memory addresses by storing an address's segment in the *offHi* field and the address's offset in the *offLo* field. For these addresses, this procedure creates a string in the form "segment:offset". For register addresses, LDP_AddressToString converts the register number encoded in the *offLo* field of the LDP Address into a register name string. The file i8086.h defines the register number associated with each 8086 register. Finally, for I/O addresses, LDP_AddressToString returns a string containing the port number which is encoded in the LDP_Address *offLo* field.

LDP_ExceptionToString converts a target specific LDP exception message into a string message. This procedure is called by the Response Handler when an LDP Exception message is received from a target. The Response Handler pre-processes the message by deserializing the LDP Address contained in Exception format messages. This address is passed to the LDP_ExceptionToString procedure through the *addr* parameter. The *buffer* parameter points to the received LDP Exception message. The exception string generated by LDP_ExceptionToString is stored in the buffer pointed to by the *string* parameter.

The 8086 target generates five exception codes: divide by zero, single step interrupt, the NMI interrupt, a execution breakpoint, and overflow; strings corresponding to these code are stored in the *string* buffer by LDP_ExceptionToString. The file i8086.h defines constants corresponding to each exception reason.

The PrintItem routine prints out a debugger value of the specified type on the debugger's terminal. It is called with the *value* longword which was obtained from the Deserialize procedure and a *type* parameter. This routine is necessary as several of the debugger's generic types such as char or pointer cannot be printed in a processor independent fashion. For example, the debugger would need to know the unusual format of an 8086 pointer to be able to print out a pointer variable.

The Serialize packs a debugger value of the specified type into an LDP buffer. For single byte types, the LDP_SerializeByte procedure is used to serialize the value. For multibyte memory types, bytes are stored in the LDP buffer least significant byte first so they stored in target memory in the correct order.

StringToLDP_Address converts a string in the specified address space to an LDP Address record. If the string is syntactically correct, the returned status is TRUE; otherwise StringToLDP_Address reports an error and returns a FALSE status.

3.2.17 Symbol Table

```
Data structures:
   struct Bucket {
      struct Bucket *link;
      char *symbolName;
      struct LDP_Address address;
      unsigned char symType;
      };

   struct ModuleSymtab {
      struct ModuleSymtab *link;
      char *moduleName;
      struct Bucket *buckets [nBuckets];
      };

   struct Symtab {
      unsigned int password;
      struct ModuleSymtab *head;
      struct ModuleSymtab *tail;
      struct ModuleSymtab *current;
      };

Exported procedures:

   struct Symtab *Symtab_Create ()
   struct Symtab *Symtab_Destroy ()
   struct Bucket *Symtab_FindSymbol (
      st, moduleName, symbolName, status)
   Symtab_GetCurrentModule (st, moduleName, status)
   Symtab_NewModule (st, moduleName)
   Symtab_NewSymbol (st, symbolName, addr)
   Symtab_PrintError (status)
   Symtab_SetModule (st, moduleName, status)
```

The Symbol Table routines support the maintenance of a symbol table data base for each target system being debugged. This data base is used by the debugger to convert symbolic procedure and variable names to target LDP addresses.

A program's symbol table is organized as a linked list. Each element of the linked list is a module symbol table which stores the symbol/address pairs for a single module within a program. The Symtab data structure is the head of the linked list of module symbol tables. The *head* field points to the first module symbol table in a program; the *tail* field points to the last module symbol

table added to the linked list. The *current* field contains a pointer to the default module for the target. The default module is searched when a symbol not containing an explicit module name is to be mapped to an LDP address

The ModuleSymtab data structure represents the symbol table for a single program module. It contains the *link* field which points to the next module symbol table in a program and the *buckets* field which points to an array of hash buckets. Each element (e.g. buckets[x]) in this array points to a linked list of Bucket structures whose symbol name hashed to the element's array index (e.g. x).

The Bucket data structure is a record containing a symbol's name and its corresponding LDP address. The Bucket's *link* field chains together Buckets whose symbol name hash to the same value. Figure 3.2 illustrates a sample symbol table.

Many of the symbol table routines return a completion code in the *status* parameter. A list of the completion codes is contained in the file Symtab.h. The constant Symtab_OK is returned on the successful completion of a Symbol Table request. When a Symbol Table routine returns an code indicating that an error occured, the debugger invokes the Symtab_PrintError routine to report the error to the user

The SymTab_Create routine creates and initializes a new instance of a Symtab data structure. This procedure returns a pointer to the new Symtab record—this is the *st* parameter passed to many of the other Symbol Table routines. The Symtab_Destroy routine purges a symbol table by returning the table's storage to the VMS operating system. The Symtab_Destroy routine is invoked by the debugger's download command prior to loading a new program. The Symtab_Create routine is then invoked to prepare for the construction of a new symbol table.

The Symtab_NewModule routine adds a new module symbol table to a program's symbol table linked list. This procedure is passed the *moduleName* parameter which specifies the name of the new module to be added to the program's symbol table. This procedure allocates storage for a new instance of the ModuleSymTab data structure and links it onto the tail of the module linked list. This routine is called for each module being downloaded to a target system.

The Symtab_NewSymbol routine adds a new symbol name—LDP address pair to the current module symbol table. Symtab_NewSymbol first allocates and initializes storage for a new Bucket record. It then hashes the symbol name and links the new Bucket record onto the linked list of symbol Buckets which hashed to the same value.

The Symtab_SetCurrentModule procedure sets a target's default module context. This specifies the default module to search for

symbols which do not include an explicit module name. The Symtab_GetCurrentModule returns the name of the current default module. These two procedures are used by the debugger's *module* command.

Symtab_FindSymbol searches the target's symbol table for the specified symbol name. If the *moduleName* parameter is NULL, the default module is searched; otherwise the module symbol table specified by the *moduleName* parameter is searched. The search is performed by first hashing the symbol name to obtain a hash index. Then, the search algorithm linearly searches each Bucket associated with the hash index for the symbol name string. If the specified symbol is found, Symtab_FindSymbol returns a pointer to the symbol's Bucket record. A client of the Symtab_FindSymbol procedure can extract the LDP address from the returned Bucket record.

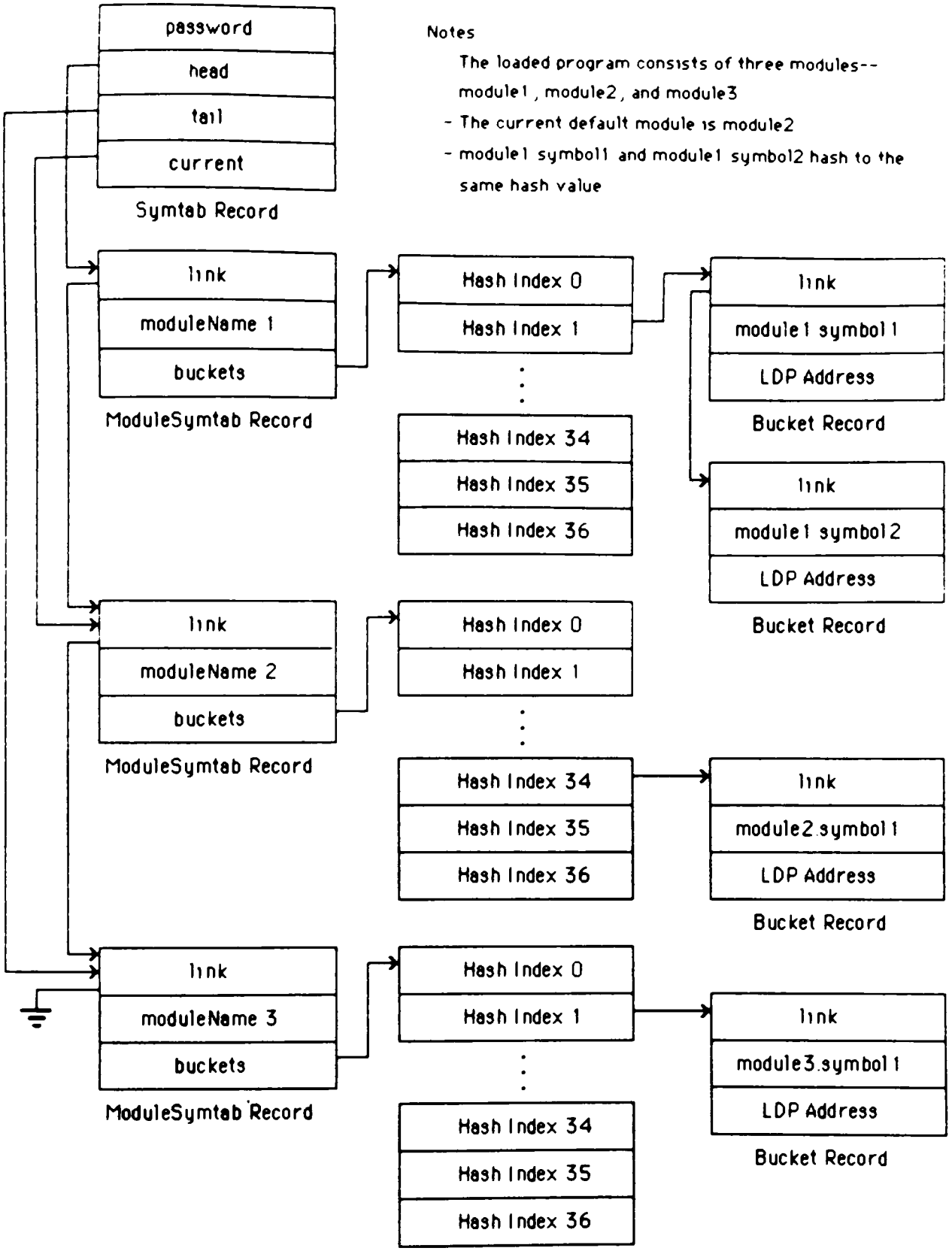


Figure 3 2. Symbol Table Organization

4. i8086 Target Software Implementation

4.1 Overview

The debugger software running on the SDK-86 is responsible for servicing host LDP requests and reporting runtime exceptions encountered by user code. The LDP Server software is coded in C and ASM86 (8086 assembly language); it occupies 12K bytes of PROM code space and 2K bytes of RAM data space. The subsystems comprising the LDP Server are as follows:

- Buffer Manager
- Interrupt Handler
- Runtime Library
- Communications software
- Exception Handler
- LDP Server process
- LDP Command Handlers

The Buffer Manager consists of routines to allocate and free Buffer elements from a statically allocated pool. This pool serves as the source of all communications buffers in the LDP Server.

The Interrupt Handler is responsible for handling the hardware and software generated interrupts on the SDK-86. In addition, the Interupt Handler exports routines to perform context switches between the debugger's LDP Server Process and user code. These routines are invoked to start or stop user code execution.

The Runtime Library contains a collection of procedures which are of general use on the SDK-86. Included in these routines are a subset of the UNIX terminal I/O procedures (e.g. putchar, puts, printf, getchar, gets), procedures to access 8086 I/O ports, and procedures to access the entire 20 bit memory space of the 8086 (which is not fully addressible when using the small option for C compilation).

The communications software consists of the modules CRC.a86, Datalink.c86, Timeout.c86 and Transport.c86. These modules implement a CRC checker/generator, a mechanism for performing timed waits, a device driver for the 8251 USART, and the datalink and transport protocols specified in sections 2.2.3 and 2.2.4.

The Exception Handler is invoked by the Interrupt Handler when the user's code encounters a software interrupt or when a NMI (non maskable interrupt) occurs. Its function is to post the exception reason to the LDP Server process which will report the exception to the host system.

The LDP Server Process is responsible dispatching LDP commands received from the host to the appropriate LDP Command Handler

and reporting runtime exceptions to the host. The LDP Command Handlers are the procedures which carry out the host requests. There is one Command Handler for each LDP command class (i e. Protocol, DataTransfer, Control, Management, etc.).

Figure 4.1 illustrates the flow of control in the LDP Server program. All interrupts in the SDK-86 are initially serviced by the Interrupt Handler. The first step performed by the Interrupt Handler is to save the CPU registers of the process that was running by pushing them on the interrupted process's stack. Then, the Interrupt Handler determines whether user code or the debugger was interrupted using a state variable which tracks the process that has control of the processor. If user code was interrupted, the user stack segment and pointer are saved and the debugger's stack pointer and stack segment are restored. This stack swap minimizes the amount of user stack space used by the debugger. The saved user stack segment and pointer are exported through the global variables *userSS* and *userSP* which allows the debugger to examine and alter the user registers when the user program is stopped.

After saving the state of the interrupted process, the Interrupt Handler invokes an imported procedure to service the interrupt. Timer interrupts are handled by the Timer_ISR procedure in the Timeout module; USART receiver and transmitter interrupts are serviced in the Datalink module. NMI's and software interrupts such as divide by zero traps, breakpoints, overflows, and single step interrupts are handled by the Exception_ISR procedure.

When the routine has completed interrupt processing, it returns control to the Interrupt Handler. The Interrupt Handler examines a variable named *rest* which may be altered to by interrupt service routines to select which process to restart—the debugger or the user. Control of user program execution is accomplished by writing the appropriate value into the *rest* variable during the course of executing LDP requests to start or stop the target. To restart user code, the debugger's stack pointer and stack segment are saved, the user's stack pointer and segment are restored, the user registers are popped off the stack, and the user code is restarted using the return from interrupt instruction.

When the debugger has control of the processor, it runs the LDP Server Process which waits for host LDP requests to be enqueued on the receiver linked list by the LDP_Handler procedure or for exceptions posted by the Exception_ISR. When an LDP request is enqueued, the LDP Process dequeues it, partially parses it to determine its LDP command class and invokes appropriate LDP Command Handler. If an exception has been posted, an LDP Exception message is constructed and transmitted to the host.

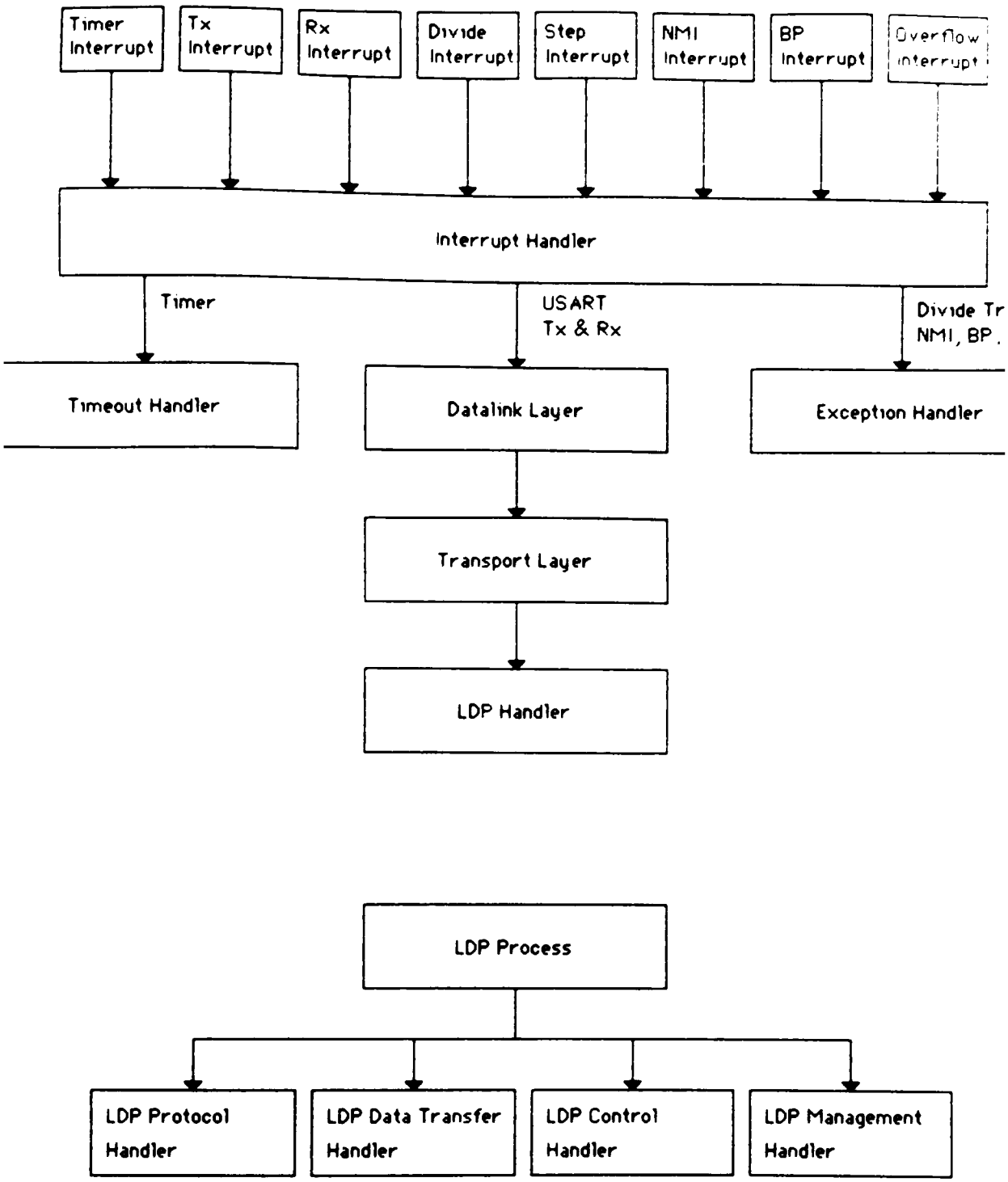


Figure 4.1. Loader/Debugger Server Control Flow

4.2.1 Buffer Manager

```
Constants
    #define Buffer_maxFrameL 160
    #define Buffer_nBuffers 6

Data structures:
    struct Buffer {
        struct Buffer *link;
        unsigned int password;
        unsigned char state;
        unsigned notify;
        unsigned parseIndex;
        unsigned int frameL;
        unsigned char frame [Buffer_maxFrameL];
        }

Exported procedures:
    struct Buffer *Buffer_Allocate ()
    Buffer_Deallocate ()
    Buffer_Init ()
    Buffer_IsBuffer ()

Imported procedures:
    Int_DisableInterrupts ()
    Int_EnableInterrupts ()
```

The Buffer Manager is the allocater of Buffer data structures in the LDP Server. These data structures are employed by the LDP Server to transmit and store received LDP frames. The Buffer data structure in the LDP Server is a simpler version of the host Buffer structure. The purpose of the *link* field is to chain Buffer elements together to form linked lists. Linked lists of buffers are used by the Buffer Manager to maintain the free pool of buffers, by the LDP Process to queue pending LDP host requests, and by the Datalink software to queue transmit requests. The *password* field contains a unique constant used for runtime type checking of buffer parameters. The *state* field contains the allocation state of a Buffer element. It enables the Buffer Manager to detect erroneous buffer allocate/deallocate sequences. The Transport software uses the *notify* field to specify the action to be taken by the Datalink layer after a buffer is transmitted—a buffer may either be returned to the Buffer Manager or marked as transmitted. The *parseIndex* field stores the position of the next byte in the frame field to unpack or

pack for the LDP Deserialize/Serialize routines. The *frameL* field tracks the number of valid bytes in the *frame* field of the buffer; the *frame* field holds the body of the communications messages.

The buffer pool is initialized during the startup of the LDP Server with a call to Buffer_Init. The size and number of buffers is determined at compile time by the constants *Buffer_maxFrameL* and *Buffer_nBuffers*. Currently, the buffers' frame size is 160 bytes. This yields a good tradeoff between target responsiveness and communications bandwidth. Larger buffer sizes result in longer delays for the first reply packet to a host request; shorter buffer sizes result in more channel bandwidth wasted for packet encapsulation and packet acknowledgements.

Buffer_Allocate returns a free Buffer element; if no element is available, it returns the constant *NullBuffer*. Buffer_Deallocate requeues the designated buffer on the free list. Buffer_IsBuffer allows clients to verify that a buffer pointer variable is valid.

The imported procedures Int_DisableInterrupts and Int_EnableInterrupts bracket critical sections in the Buffer Manager implementation.

4.2.2 Interrupt Handler

Exported procedures:
```
Int_DisableInterrupts ()
Int_DisableLevel (level)
Int_EnableInterrupts ()
Int_EnableLevel (level)
Int_EOI (level)
Int_Init ()
boolean Int_IsEnabled (level)
Int_SetReturnContext (context)
```

Exported variables:
```
unsigned userSP;
unsigned userSS;
unsigned excep;
```

Imported procedures:
```
Exception_ISR ()
Rx_ISR ()
Timer_ISR ()
Tx_ISR ()
```

The Interrupt Handler is responsible for processing hardware interrupts generated by the i8259A interrupt controller, NMI's generated by the SDK-86 INTR button, software interrupts, and for performing context switches between user and debugger code.

Int_Init is called during the startup of the LDP Server It initializes the 8086's software interrupt vectors, sets up the i8259A interrupt controller, and initializes the state variables that track which process has the processor.

Int_DisableInterrupts and Int_EnableInterrupts disable and enable all maskable hardware interrupts. These procedures guard critical sections of code in the target system.

Int_DisableLevel and Int_EnableLevel allow the software to selectively disable and enable a particular interrupt level. Int_IsEnabled allows a client to determine if a particular interrupt level is enabled.

Int_EOI must be called after servicing a hardware interrupt. This procedure issues a command to the interrupt controller to rearm the specified interrupt level. The i8259A is programmed to mask out the interrupt level being serviced until this command is issued; this prevents the nested invocation of an interrupt service routine.

The following table lists the interrupt source for each 8259A interrupt level:

Level	Interrupt Source
0..3	Unused
4	8254 Timer -- 100 ms period
5	8251A USART Receiver Interrupt
6	8251A USART Transmitter Interrupt
7	Unused

Int_SetReturnContext selects which process the Interrupt Handler will restart on completion the current interrupt routine. This procedure is called by the LDP Control handler when processing host requests to start or stop user program execution and by the Exception_ISR when handling the case of starting a user's program at a breakpoint. If Int_SetReturnContext is not called during the execution of an interrupt service routine, the Interrupt Handler will restart the process which was originally interrupted.

The Exception_ISR, Rx_ISR, Tx_ISR, and Timer_ISR are procedures imported by the Interrupt Handler to service SDK-86 interrupts. When a software interrupt or an NMI occurs, the Interrupt Handler sets the global variable *excep* to a constant corresponding to the interrupt type and calls the Exception_ISR procedure. The variable *excep* allows the Exception_ISR to determine which interrupt caused its invocation.

The Rx_ISR and Tx_ISR routines are called for 8251 USART receiver and transmitter interrupts. The Timer_ISR is invoked when output of timer #1 of 8254 pulses; the pulse period is programmed to be 100 ms. The timer interrupt is used to detect ack timeouts by the transport layer.

When the user's program is interrupted, the Interrupt Handler pushes the user's registers on the user stack and saves the user's stack pointer and stack segment in the global variables *userSP* and *userSS*. The LDP Server is able to read and write the user's registers through these variables.

4.2.3 Runtime Library

Data structures:

```
struct Pointer {  /* in Pointer.h */
   unsigned offset;
   unsigned segment;
   }
```

Exported procedures:

AsciiToBin (c)
CharTyped ()
Crash (crashCode)
char getchar ()
gethex ()
gets ()
HighByte (w)
isdigit (c)
LP_Fill (addr, length, pattern)
LP_ReadByte (pointer)
LP_ReadWord (pointer)
LP_WriteByte (pointer, byte)
LP_WriteWord (pointer, word)
LowByte (w)
port_input_byte (port)
port_input_word (port)
port_output_byte (port, byte)
port_output_word (port, word)
printf (formatString, v1, v2, ..., v_k)
putchar (c)
puts (s)
start ()
strlen (s)
Timer_WriteTimer (timer, timeConstant)
toupper (c)

The Runtime Library contains a collection of utilities which are suitable for a wide range of SDK-86 applications. Included in the library are terminal I/O routines, data conversion/manipulation routines, procedures to access the full 20 bit address space of the 8086, and procedures to perform reads and writes in the 8086 I/O space.

The Mark Williams Company (supplier of the C compiler) provides a runtime library for C applications which also implements most of these routines. It was not used because it also includes many routines which are not used by the LDP Server. The extra memory requirements of the Mark Williams library rendered

it unsuitable for this application.

The procedures getchar, gets, putchar, and puts are functionally equivalent to their UNIX counterparts. getchar returns a single character from the SDK-86 terminal; gets returns a NUL terminated string from terminal. putchar and puts write an ASCII character or NUL terminated string to the SDK-86 terminal. printf writes a sequence of values to the terminal using a format string passed in the first printf parameter. It is funtionally similar to the Fortran WRITE/FORMAT statements. gethex reads a string from the terminal, converts the string to a number, and returns the string's value. The CharTyped procedure returns a TRUE if a character has been typed at the terminal but not yet read. Although the PROM version of the LDP Server does not use these I/O procedures, they were invaluable when debugging the LDP Server program.

AsciiToBin, HighByte, isdigit, LowByte, strlen and toupper are the data conversion/manipulation routines. AsciiToBin converts an ASCII character to its corresponding numeric value. HighByte and LowByte return the upper or lower byte of a 16 bit word. isdigit returns a TRUE value if its argument is the character code of a hexadecimal number. toupper converts lower case letters to upper case. Finally strlen returns the length of a NUL terminated string.

The LP (Long Pointer) routines allow memory references anywhere in the 20 bit address space of the 8086. These routines are necessary as programs compiled using the C compiler's small option can only address a 64K region of memory. The pointer parameter in the LP routines references a Pointer data structure containing an 8086 segment and offset. LP_ReadByte and LP_ReadWord read a byte or word from the location refered to by the pointer parameter. LP_WriteByte and LP_WriteWord write a byte or word to the specified location. LP_Fill fills a block of memory of specified length with the byte value passed in the *pattern* parameter.

The port procedures allow a client to read or write to the 8086's I/O space. port_input_byte and port_input_word read a byte or word from the specified I/O port; port_output_byte and port_output_word write a byte or word.

The start procedure in the module Startup.a86 bootstraps a C application on the SDK-86. This procedure is the first code which executes in the LDP Server. It disables interrupts, stashes away the initial set of CPU registers, sets up the stack, and calls the procedure *main* which is exported by module LDPA.c86.

The debugger's stack consists of statically allocated memory between the *stkbot* and *stktop* symbols. The stack is filled with a distinctive pattern by the start routine; this facilitates the post mortem detection of a stack overflow by dumping memory in the

stack area.

When main is called, it is passed a pointer to the initial set of CPU registers. The registers may be used to pass parameters to the program being started. For example, a demo clock program written for the 8086 uses the initial registers to set the time.

The Crash procedure is called when an irrecoverable error is detected by the LDP Server. It disables interrupts, displays the *crashCode* parameter on the SDK-86 LED displays, and halts. All LDP Server crashcode values are contained in the file Crash.h. If a crash occurs, search the Crash.h file for the hexadecimal crash code on the LED displays. This will yield a definition in the following form:

```
#define module_id hexCrashCode
```

The "module" part of this definition determines the file that caused the crash. Search this file for the "id" portion of the definition; this will uniquely determine the line of code which reported the crash condition.

4.2.4 Datalink layer

Data structures:

```
struct Datalink_Stats {
   unsigned nCRC, nFE, nOE, nPE, nRx, nTx;
   }
```

Exported procedures:

```
CRC_Generate (oldCRC, byte)  /* in module CRC.a86 */
Datalink_GetMaxBufferSize ()
Datalink_GetStats (statsPtr)
Datalink_Init ()
Datalink_Transmit (buffer)
USART_RxISR ()
USART_TxISR ()
```

Imported procedures:

```
Transport_Handler (buffer)
```

Private procedures:

```
StartTransmitter ()
USART_Init ()
```

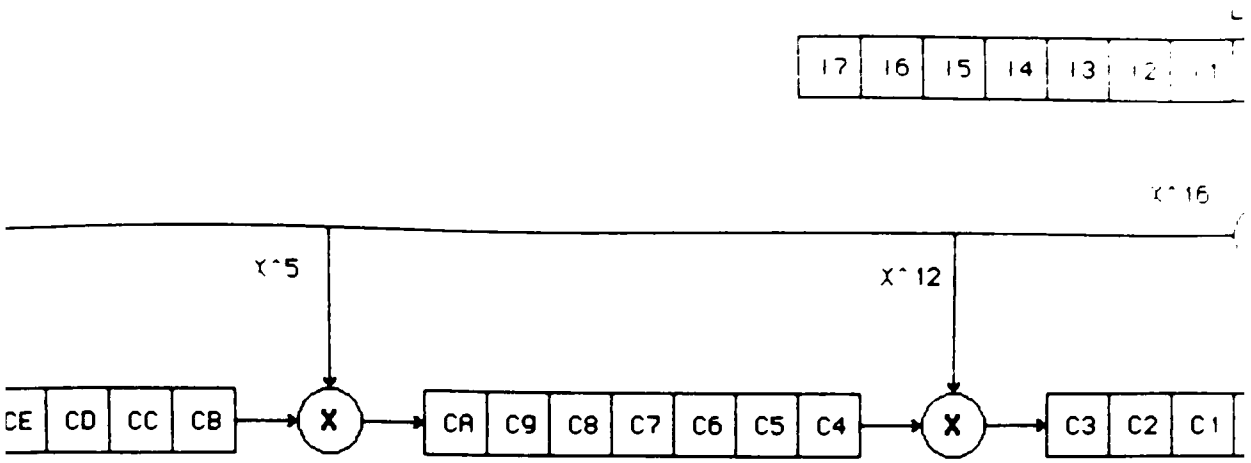
These procedures implement the RS-232C Datalink protocol specified in section 2.2.4. Unlike the host implementation, this code only supports a single datalink connection.

The module CRC.a86 exports the CRC_Generate procedure. The algorithm used by CRC_Generate is based on a method described by Aram Petez [21] for implementing the CRC-16 polynomial. Figure 4.2 illustrates the algorithm employed. The algorithm only uses 55 bytes of code space and computes the CRC for one byte in 65 μS (2.5 MHz 8086 clock). CRC_Generate is used for checking the CRC of incoming frames and for generating the CRC for frames transmitted by the Datalink.

Datalink_GetMaxBufferSize returns the number of bytes in the *frame* field of a buffer less the number of bytes used by Datalink encapsulation. This allows the Transport layer and ultimately the LDP Server to determine the number of free bytes in a buffer.

Datalink_GetStats allows a client to obtain a copy of the communications statistics maintained by the Datalink software. Datalink_GetStats stores these statistics in the Datalink_Stats structure referenced by the *statsPtr* parameter. The statistics include the number of framing errors, overrun errors, CRC errors and the number of frames received and transmitted.

Byte-wise computation of CCITT CRC -- X^16 + X^12 + X^5 + 1



Initial Conditions.

CRC Register = CF CE CD CC CB CA C9 C8 C7 C6 C5 C4 C3 C2 C1 C0

Input Byte = 17 16 15 14 13 12 11 10

After computation of CCITT CRC, the new CRC Register is

MS Bit

C7	C6	C5	C4	C3	C2	C1	C0	CF	CE	CD	CC	CB	CA	C9	C8
C3	C2	C1	C0	13	12	11	10	C4	C3	C2	C1	C0	C6	C5	C4
13	12	11	10		C7	C6	C5	C0	13	12	11	10	C2	C1	C0
17	16	15	14		C3	C2	C1	10				C7	12	11	10
					13	12	11	14				C3	16	15	14
					17	16	15					13			
												17			

Each column is one bit of the crc register The entries in each column are exclusive or'ed together

Substituting Xi = Ci XOR Ii, we obtain:

								CF	CE	CD	CC	CB	CA	C9	C8
X3	X2	X1	XD												
X7	X6	X5	X4	X3	X2	X1	X0								
					X7	X6	X5	X4	X3	X2	X1	X0			
					X3	X2	X1	XD							
												X3	X2	X1	X0
												X7	X6	X5	X4

Figure 4.2 Byte-wise CRC calculation.

Datalink_Init performs the initialization of the Datalink software and hardware. This involves setting up the 8254 timer used as a baud rate generator, initializing the Datalink transmitter linked list, resetting the Datalink statistics, initializing the i8251 USART, and priming the Datalink receiver The baud rate is set to 9600 baud; attempts to use 19,200 baud resulted in many garbled frames.

Datalink_Transmit asynchronously transmits a Buffer to the VAX. The transmit buffer is first added to the linked list of transmit buffers maintained by the Datalink. If the newly added buffer is the only buffer in the list, the transmitter was idle. In this case, the StartTransmitter private procedure is invoked. This sets up a pointer to the frame currently being transmitted, initializes the transmit CRC, and initializes a count of the number of bytes left to transmit. Then, a STX byte is output to the USART. The remainder of the frame is sent by the USART transmitter interrupt service routine.

The Tx_ISR procedure is called by the Interrupt Handler for each 8251 transmitter interrupt. These occur when the transmitter is ready to send another byte. Tx_ISR is responsible for transmitting the frame's bytes (with inserted DLE's), the two CRC bytes, and the ETX character. When a frame has been transmitted, the *notify* field of the Buffer record is examined; it indicates whether the transmitted frame should be returned to the Buffer Manager or marked as transmitted. The Transport layer uses this field to detect when a frame has been completely transmitted and to prevent information format frames from being returned to the free pool—these frames may need to be retransmitted. Also at the completion of a frame, the Tx_ISR examines the transmitter linked list for another frame to transmit; if one is available, the transmit process is restarted with a call to StartTransmitter; otherwise the transmitter enters the idle state.

The Rx_ISR procedure is invoked for each character received by the i8251. This procedure is responsible for checking the framing and CRC of incoming messages, for stripping out inserted DLE bytes, and for accumulating the bytes of a message in a communications buffer. If the frame is received successfully, it is passed to the Transport layer through an upcall to Transport_Handler. Frames containing errors are discarded by resetting the receiver's state and waiting for the next STX byte which marks the beginning of a new frame.

4.2.5 Transport layer

Exported procedures:
- Transport_GetMaxBufferSize ()
- Transport_Handler (buffer)
- Transport_Init ()
- Transport_SendAck ()
- Transport_TransmitFrame (buffer)

Imported procedures:
- LDP_Handler (buffer)
- Datalink_Transmit (buffer)

Private Procedures:
- ProcessInfo (buffer, controlByte)
- ProcessSuper (buffer, controlByte)
- ProcessUnnumbered (buffer, controlByte)

The Transport software is responsible for establishing and maintaining a single transport connection with the host.

The target transport implementation differs from the host implementation in one significant way. The host transport layer automatically generates an acknowledgement for each information format frame received from the target. In the target implementation, the LDP Server is required to either transmit an information format frame or explicitly request the transport layer to transmit an ack for each information format frame received from the host. In the former case, an ack message is piggybacked on the frame by the transport layer. This difference is due to the nature of the traffic between the target and host: most host initiated messages require at least one information format response. Rather than sending both an ack and the response frame, the communications bandwidth is better utilized by piggybacking the ack on the response frame. Although this results in more work for the target software, it yields better response to host requests.

The Transport_GetMaxBufferSize routine returns the number of bytes in the frame field of a Buffer structure used by the Datalink and Transport layers. This allows the LDP Server to determine how much LDP data can be packed into a buffer.

The Transport_Init procedure is called by the LDP Server initialization code. This procedure resets the transport's sequence numbers, initializes the Datalink with the Datalink_Init routine, and attempts to establish a transport connection using the RIM/UA protocol.

The Transport_Handler is upcalled by the Datalink when an

error free frame is received. The frame is handled by ProcessInfo, ProcessSuper or ProcessUnnumbered, depending on the frame type (i.e. information, supervisory, or unnumbered).

An information format frames is handled by storing the incoming *nr* sequence number in the variable *his_nr*, verifying that the frame is in sequence, and forwarding the message to the LDP layer through the LDP_Handler procedure. The stored *nr* sequence number is a potential piggybacked ack; it is polled by Transport_TransmitFrame when waiting for an ack.

Received supervisory frames are processed by incrementing a counter corresponding to the supervisory type (i.e. RR or RNR) and storing the frame's *nr* sequence number in *his_nr*. These counters are also polled by Transport_TransmitFrame when expecting an ack; they allow Transport_TransmitFrame to quickly determine the type of response received from the host.

Received SIM (Set Initialization Mode) unnumbered frames indicate that the host wishes to reset the transport layer connection. The Transport software responds to SIM request by resetting the transport sequence numbers and transmitting an UA (Unnumbered Ack) frame.

Transport_SendAck is called by the LDP Server to explictly send a RR (Receiver Ready) acknowledgement. This procedure allocates a communications buffer, formats a supervisory control byte containing the sequence number of the next frame expected from the host, stuffs the control byte into the transport encapsulation field, and transmits the frame with the Datalink_Transmit routine.

Transport_TransmitFrame synchronously transmits an LDP message to the host. This requires three steps. First, an information format control byte is built and stored in the client's frame. Next, the Transport_TransmitFrame enters a loop which transmits the frame using the Datalink_Transmit routine and waits for an acknowledgement. The loop is terminated when the frame has been acknowledged or too many retransmissions have occured. Last, the buffer is freed and a completion status is returned to the caller. While waiting for an acknowledgement, one of three events may occur—an ack arrives, a timeout occurs, or a RNR message arrives. For timeouts, Transport_TransmitFrame increments a retransmit count and tries again. When the retransmit count exceeds four, the transmission attempt is aborted. If a RNR message arrives, the host is signalling that a temporary receiver buffer shortage exists. In this case, the retransmit count is reset to zero, the frame is retransmitted, and the procedure again waits for an ack. This response to a RNR message causes the target to keep trying to send as long as the host appears to be running.

4.2.6 Exception Handler

Exported procedures:
Exception_ISR

Imported procedures:
Int_SetReturnContext (context)
LDP_PostException (exceptionCode)

Private Procedures:
SaveUserRegs ()
StopTarget (from)

The Exception Handler is invoked by the Interrupt Handler when a software interrupt or NMI interrupt occurs during the execution of user code. The types of software interrupts caught by the LDP Server are the divide by zero error, single step, breakpoint, and overflow.

The Exception Handler plays an important part in controlling the execution of user code. Breakpoints are implemented by inserting the one byte INT opcode at breakpoint locations. Execution of this opcode results in a breakpoint software interrupt. When a user's program is continued from a location containing a breakpoint, it will encounter a breakpoint interrupt before executing any code. To remedy this situation, when continuing user code after a breakpoint, the LDP Server single steps the program once without inserting the INT opcodes. Then, the breakpoint opcode(s) are inserted and the user's code is allowed to run.

When the Exception Handler catches a single step interrupt, it determines if the user code is being stepped around a break opcode by the LDP Server. If so, it reinserts the INT opcodes at the user's breakpoint locations and invokes Int_SetReturnContext to restart the user process on return from the single step interrupt. If a normal single step interrupt is encountered, the Exception Handler performs two actions. First, the SaveUserRegs private procedure is called. This routine pops the user registers from the user stack and stores them in the *userRegisters* array. This restores the user's stack to its state prior to the single step interrupt. Then, Int_SetReturnContext is called to restart the debugger process.

Divide Errors, NMI's, and overflows are handled by calling the private procedure StopTarget and posting the exception reason to the LDP Process using the LDP_PostException routine. StopTarget saves the user registers with the SaveUserRegs private procedure, replaces any inserted INT instructions with the original code bytes, and sets the return context to the debugger process.

Breakpoints are handled in a similar fashion but require one

additional step. After saving the user registers, the IP (Instruction Pointer) is decremented by one. This points the IP at the instruction which was not executed due to the breakpoint.

4.2.7 LDP Server Process

Exported procedures:
LDP_PostException (exceptionCode)

Imported procedures:
Transport_SendAck ()

Private Procedures:
struct Buffer *Dequeue ()
Dispatch ()
PostException (exceptionCode)

The LDP Server Process is the primary task in the target software. While the debugger has control of the processor, this task continually loops waiting for one of two events: LDP frames enqueued on the receiver linked list or exceptions posted by the Exception Handler.

The Transport layer passes received frames to the LDP layer with an upcall to LDP_Handler in the context of a USART receiver interrupt service routine. The LDP_Handler partially deserializes the message to determine its LDP command class and type. This step is required because certain LDP messages are handled immediately in the interrupt context. For example, an LDP Abort request which is sent by the host to abort a command in progress must be processed immediately as the LDP Server Process may be tied up processing a host request. LDP Control messages are also handled in an interrupt context. This allows user or debugger code to easily be restarted by the Interrupt Handler by simply reloading the user or debugger stack pointer, popping saved registers, and executing the IRET (return from interrupt) instruction. Messages which do not need to be handled in an interrupt context are enqueued on the received linked list by the LDP_Handler.

When the LDP Server Process polls the head of the received linked list and detects an enqueued frame, it dequeues the frame and calls the private procedure Dispatch. Dispatch partially parses the frame to determine its LDP command class and calls the LDP Command Handler responsible for the frame's command class.

The Exception Handler posts exception reasons to the LDP Server using the exported procedure LDP_PostException. This procedure stores the exception code in the variable *postedException* which is also polled by the LDP Server Process. When a new exception is detected, the LDP Server calls the private procedure PostException. This procedure builds an LDP Exception frame containing the exception code and user program address at which the exception occured and transmits the frame to the host.

4.2.8 LDP Protocol Handler

Exported procedures:
LDP_ProtocolHandler (header, buffer)
LDP_ReportError (errorCode, errorData)

Private Procedures:
SendHelloReply ()
SendSynchReply ()

The procedure LDP_ProtocolHandler processes LDP Protocol class messages received from the host that are not handled in an interrupt context. These are the LDP Hello and Synch Protocol messages.

The Hello request is sent by the host to determine the target's LDP protocol version, processor type, and LDP options such as the LDP debugger level and supported address formats. The private procedure SendHelloReply transmits a response frame to the HelloRequest containing the current LDP protocol version number, a constant identifying 8086 targets, and a constant indicating that the LDP Server program supports the Basic Debugger level of protocol implementation.

The Synch request establishes a virtual connection between LDP peers. The SendSynchReply private procedure responds to this request with an LDP SynchReply frame.

4.2.9 LDP Data Transfer Handler

Exported procedures:
LDP_DataTransferHandler (header, buffer)

Private procedures:
DoMove (header, buffer)
DoRead (header, buffer)
DoRepeatData (header, buffer)
DoWrite (header, buffer)
LDP_ComputeUnitSize (ldpAddr)
LDP_Fetch (ldpAddr, doIncrement)
LDP_Store (ldpAddr, value, doIncrement)
SendReadDone ()

The LDP_DataTransferHandler processes host requests to read, write, move or fill SDK-86 memory or to read/write user registers or SDK-86 I/O ports.

The LDP_DataTransferHandler is partitioned into a processor dependent and a processor independent module to ease portability of the LDP Server software. The module LDPM.c86 contains the target dependent procedures LDP_ComputeUnitSize, LDP_Fetch and LDP_Store. LDP_ComputeUnitSize returns the number of bytes that each object in the address space designated by the ldpAddr parameter occupies. Given a byte count, the target independent procedures use this procedure to determine the number of target data objects to process. For example, if the LDP_DataTransferHandler receives a message to write six bytes into the target register space, it uses LDP_ComputeUnitSize to calculate that this involves writing three registers (two bytes/register).

LDP_Fetch and LDP_Store perform the processor dependent reads or writes of target data. These procedures are passed an LDP address which contains the *mode* field that specifies the address space to read or write and the *offHi* and *offLo* fields that determine the address. The *doIncrement* parameter notifies LDP_Fetch or LDP_Store to increment the LDP address parameter after performing the read or write—this is useful when reading or writing bytes in a contiguous memory block. LDP_Store is passed the *value* parameter which is a byte or word to write into target memory; LDP_Fetch returns a byte or word from target memory.

The processor independent implementation of the LDP_DataTransferHandler is contained in the file LDPC.c86. The format of an LDP addresses and data is opaque to procedures in this module.

LDP Move messages request the target to move a block a memory to a destination address. These requests are processed by

the DoMove private procedure. After deserializing the source address, destination address, and count, DoMove enters a loop calling LDP_Fetch to read an item from the source block and calling LDP_Store to store the item in the destination range.

LDP Read messages are transmitted by the host to read a block of target data. This is implemented by the DoRead procedure using three steps. First, the start address and count parameters are deserialized from the LDP Read message. Then DoRead transmits the requested data in LDP ReadData response messages. Each response message contains an LDP command header, the LDP address of the first data object in the message, the number of data bytes in the message and the actual target data. More than one response message may be sent due to the size of a communications buffer. When all the requested data has been transmitted, the SendReadDone procedure is called. This procedure sends an LDP ReadDone frame informing the host that the Read request has been completed.

DoRepeatData handles requests from the host to fill target memory with a pattern contained in LDP RepeatData messages. DoRepeatData first deserializes the RepeatData destination address and repeat count parameters. Then, the pattern string in the packet is iteratively written to target memory for the specified repeat count.

LDP Write messages request the target to store message data into the specified address space. DoWrite processes these requests in two steps. First, the destination address is deserialized from the message and the number of objects contained in the message is computed using the number of bytes in the message and LDP_ComputeUnitSize. Then, DoWrite deserializes each data object and stores it to target memory with the LDP_Store procedure.

4.2.10 LDP Control Handler

Exported procedures:
- LDP_ControlHandler (header, buffer)
- LDP_InitControl ()
- LDP_StopTarget ()

Imported procedures:
- BP_AtBreakpoint ()
- BP_InsertBreaks ()
- BP_RemoveBreaks ()
- LDP_FetchUserReg (regID)
- LDP_ReportError (errorCode, errorData)
- LDP_StoreUserReg (regID, value)

Private procedures:
- RestoreUserRegs ()
- SaveUserRegs ()
- SetupRun ()
- SetupStep ()
- StopTarget (from)

The LDP Control Handler processes requests from the host to start, continue, halt, and single step user code. These requests are managed by the LDP_ControlHandler procedure which is called in an interrupt context by the LDP_Handler. By running in an interrupt context, the LDP_ControlHandler can specify resumption of the debugger or the user process when returning from the interrupt. Start, continue, and single step requests are implemented by resuming the user process; the stop request is implemented by resuming the debugger process.

LDP_InitControl runs during the debugger initialization code. This procedure initializes the variable *runState* which maintains the state of user program execution and initializes the *popOrder* array which is a mapping between the order registers are pushed on the stack by the Interrupt Handler and indices into the *userRegisters* array which stores the user registers while the user process is halted.

The Start and Continue messages request the debugger to start execution of user code. The Start request includes an program start address; the Continue request resumes execution from the user's current program counter. These requests are carried out as follows: the *runState* variable is first checked to verify that the user process is halted. If not, an error response indicating that the target is already running is returned to the host using LDP_ReportError. For Start requests, the user's CS (code segment)

and IP (instruction pointer) are loaded using the address deserialized from the Start message. These are written with calls to LDP_StoreUserReg which allows write access to the *userRegisters* array. The next action depends on whether the user's code will be started at a breakpoint location. The SetupRun private procedure is invoked when execution is not started at a breakpoint. This routine restores the user's breakpoints, transfers the user registers from the image maintained in the *userRegisters* array onto the user stack, and sets the interrupt return context to be the user process.

When starting user code from a breakpoint location, the debugger will single step the instruction at the breakpoint location, reinsert the breakpoints and resume user code. The procedure SetupStep is invoked to perform the operations necessary to single step the user process; the Exception Handler performs the remainder of these actions.

Steps are implemented using the 8086 single step interrupt which is enabled by a bit in the 8086 flags registers. After this bit is set, a single step interrupt will occur after execution of one machine instruction. SetupStep sets the step bit in the user's flags register contained in the *userRegister* array and establishes the return context to be the user process. When the Interrupt Handler executes a return from interrupt instruction to resume the user process, the flags register containing the set single step bit will be loaded into the 8086. The processor will then execute one user machine instruction and trap with a single step interrupt. The Exception_Handler then reinserts breakpoints and restarts the user's program.

The LDP Stop message is sent by the host to stop execution of user software. If the user's program is already halted, an error is reported to the host using LDP_ReportError. Otherwise, the StopTarget routine is invoked. This procedure transfers the user registers from the stack to the *userRegisters* array, removes inserted breakpoints, and sets the return context to the debugger process.

The LDP Step message requests the execution of one user machine instruction. This request is processed by the SetupStep procedure described above.

If the debugger receives an LDP request while user code is running, it will halt its execution before processing the request. This action is perfomed by the LDP_Handler routine prior to enqueuing a host request on the receiver linked list using the LDP_StopTarget routine. LDP_StopTarget calls the StopTarget procedure if the *runState* variable indicates that the user program is running. This action is necessary since the LDP Server Process is suspended while the user process is running; in order to service the host request, the LDP Server Process must be restarted.

4.2.11 LDP Management Handler

```
Constants:
   #define breakpointOpcode 0x0CC
   #define maxBreakpoints 5

Data structures:
   struct {
      unsigned ID;
      struct Pointer bpAddr;
      unsigned char opcode
      } bpTable [maxBreakpoints]

   struct LDP_Descriptor {  /* in LDP.h */
      unsigned mode;
      unsigned modeArg;
      unsigned idHi;
      unsigned idLo;
      }

Exported procedures:
   BP_AtBreakpoint ()
   BP_InsertBreaks ()
   BP_RemoveBreaks ()
   LDP_Init_BPTable ()
   LDP_ManagementHandler ()

Private procedures:
   HandleDelete (buffer)
   HandleCreate (buffer)
   HandleList ()
```

The module LDPE.c86 contains the routines which process LDP Management frames. These frames create, delete, or return a list of target breakpoints, watchpoints, processes, or named objects. Watchpoints specify an address to be monitored; target execution is halted when the data at the specified address matches a pattern. Named objects are used to symbolically reference target data structures. Watchpoints, processes and named objects are LDP options and not handled in this implementation.

The host and target reference breakpoints in LDP messages with Object Descriptors in LDP Management frames. These Descriptors contains a *mode* field which selects either a breakpoint, watchpoint, process, or named object and an *id* field. For breakpoints, the *id* field of the Object Descriptor contains an integer referencing a particular breakpoint. The target assigns these id's

for each created breakpoint; the host uses the *id* field to reference a breakpoint to be deleted. The LDP Serialize and Deserialize routines convert between an Object Descriptor in an LDP frame and the LDP_Descriptor data structure which represents Object Descriptors in the host and target software.

The module LDPE.c86 contains the *bpTable* data structure which maintains overhead information for each active breakpoint. The constant *maxBreakpoints* determines the maximum number of breakpoints that may be set at once. Each entry in the *bpTable* contains an *id*, *bpAddr*, and *opcode* field. The *id* field stores the id generated by the target for the breakpoint. The *bpAddr* field is a pointer to the instruction in user code at which the breakpoint was set. The *opcode* field stores the user code byte at a breakpoint location while user code is running.

BP_InsertBreakpoints is called by the LDP Control Handler prior to starting user code. For each active entry in the *bpTable*, this procedure stores the user code byte in the *opcode* field and inserts a breakpoint opcode at the location referenced by the *bpAddr* field. BP_RemoveBreakpoints is called after halting user code. This procedure restores the user code bytes at breakpoint locations. BP_AtBreakpoint returns a TRUE value if the user's current program counter is at a breakpoint location; the Control Handler uses this procedure to determine if it is starting user code at a breakpoint.

The LDP_ManagementHandler procedure is invoked by the LDP Server process for LDP Management frames. This procedure calls HandleCreate, HandleDelete, or HandleList depending on the management frame's command type.

HandleCreate first verifies that the host wishes to create a breakpoint object and that there is a free slot in the *bpTable*. If either of these conditions is false, HandleCreate returns an Error frame to the host. Next, HandleCreate deserializes the breakpoint address in the frame. If a breakpoint is already set at the address, an Error frame is returned. Then, HandleCreate generates a new id for the breakpoint and enters the id and breakpoint address in the *bpTable*. Finally, a CreateDone response is transmitted to the host containing the id of the created breakpoint.

HandleList is responsible for returning a BpList reply to a host ListBPs request. This reply contains the id and LDP Address for each active breakpoint. This is accomplished by allocating a communications buffer, creating and serializing an LDP_Descriptor and LDP_Address for the id and address of each breakpoint, and transmitting the buffer.

HandleDelete deletes the breakpoint referenced by the Object Descriptor in an LDP Delete frame. This is implemented with the following steps. First, the Object Descriptor in the Delete frame is

deserialized into an LDP_Descriptor structure. Then, the *bpTable* is searched for the id contained in the LDP_Descriptor; if the id is not found, an Error response frame is sent to the host. If the id is found, its table entry is deleted by moving the entries which follow it up one slot. When processing of the Delete frame is complete, an LDP DeleteDone frame is transmitted to the host indicating successful completion of the Delete request.

5. Testing and Validation

The testing and validation of the debugger's implementation was performed in five phases:

- Debugger expansion hardware testing
- Datalink/Transport layer testing
- Host/Target module testing
- Runtime library testing
- System testing

The support of multiple target systems is the only part of the host software which was not tested. This testing was not performed because of the unavailability of a second VAX serial line and because only one debugger expansion board was built.

5.1 Debugger Expansion Hardware Tests

The expansion hardware consists of three subsystems which required testing: the serial I/O channel, the interrupt controller, and the memory. These were exercised using three programs: UTest.c86, IntTest.c86, and MemTest.c86. These test programs use the SDK-86 terminal to obtain test parameters from the user and report test results.

UTest.c86 is a program which tests the serial channel. It is run with a loopback plug on the debugger expansion board serial connector. This plug connects the serial data output pin to the input pin. UTest first initializes the i8251A USART and the i8254 timer chip which generates the baud clock. Then, it enters a loop which performs the following:

1. Write a test byte to the USART.
2. Waits up to five milliseconds for the i8251A to receive the test byte through the loopback path.
3. If the test byte is not received, print a timeout message
4. Verify that the received data matches the transmitted test byte. If not, print an error message.
5. Increment the test byte.

IntTest.c86 checks out the i8259A interrupt controller. It first initializes the interrupt controller hardware using routines in the module Int.a86. Next, it sets up the i8254 timer chip to generate an interrupt once a second. For each timer interrupt generated, the test program prints out the character 'T'. By counting the number of characters printed out in a fixed period of time, the

user can verify that the interrupt controller is generating interrupts at the proper frequency.

MemTest.c86 exercises the RAM on the debugger expansion hardware. It first prompts the user for a range of memory to check. Then, it enters a loop which verifies the memory in the user defined memory block using the following test patterns:

1. all ones
2. all zeros
3. 10101010
4. 01010101
5. A pseudo random pattern

Memory errors are reported on the SDK-86 terminal.

5.2 Datalink/Transport Tests

The Datalink and Transport layer implementations for the VAX and the SDK-86 were tested using the programs TLTest.c (VAX) and TLTest.c86 (SDK-86). These programs can run in one of two modes: echo user or echo server mode. In the echo user mode, they generate a test pattern, transmit it using the transport layer, and wait for an echo response from a peer test program. The echo user software verifies that the response contains an exact copy of the transmitted frame. In the echo server mode, the test programs echo received transport messages using the transmit function of the transport layer. During a test run, one computer acts as an echo server; the other acts as an echo user.

To verify the retransmission algorithm, messages were intentionally garbled by disconnecting the serial connection between the VAX and the SDK-86. This test software verified the following functions:

1. Datalink framing
2. Datalink CRC checking and generation
3. Datalink transparency algorithm (the test patterns include data bytes which require insertion and removal of DLE bytes)
4. Transport sequencing
5. Transport acknowledgements/piggybacking
6. Transport retransmissions

5.3 Host/Target Module Tests

Several debugger modules contain test code which is enabled using conditional compilation. The code allows the module to run

standalone interactive tests. Examples of this may be found in the following modules:

Buffer.c	-- exercises buffer allocate/deallocate sequences
Symtab.c	-- exercises the Symbol Table public procedures
i8086A.c	-- allows disassembly of an instruction entered by the user

Other modules were tested using auxillary test programs. The 8086 personality modules were tested using the ProcTest.c program. This program prompts a user for the personality procedure to invoke and for the procedure's parameters. It then calls the requested personality procedure and displays the return values. This program could be used to verify the operation of new personality modules added to the debugger.

5.4 Runtime Library Tests

The Runtime Library was tested using the programs PFTest.c86 and LibTest.c86. These programs exercised the following library functions:

1. CRC generation
2. putchar, puts, printf, getchar, gets
3. strlen, toupper
4. port input and output routines

In addition, the Runtime Library was exercised by the hardware diagnostic programs.

5.5 System tests

To verify the operation of the debugger, four programs were developed using the VMS 8086 development tools and the debugger These programs were called Pat, LEDClock, Dice, and DivZero. In the course of developing these programs, many of the debugger's functions were exercised. LEDClock also helped exercise the SDK-86 Runtime Library.

Pat is an assembly language program which presets SDK-86 memory to a test pattern using the ASM86 *db* and *dw* directives. This program was used to verify that the Object Module Format download command was correctly processing PIData and PEData records and that the debugger's memory read commands (i.e. byte, char, long, pointer, and word) were working properly.

LEDClock is a C program which turns the SDK-86 into a digital clock. It uses the SDK-86 LED display to show the current time.

The time is preset by setting the 8086's AX, BX, CX, and DX registers using the debugger before starting the LEDClock program.

Dice is an assembly language program obtained from the SDK-86 User's Guide. This program rapidly displays the numbers one through six on an SDK-86 LED display. The display is frozen when a button is pressed on the SDK-86 keypad; it is restarted when a second button is pressed. Breakpoint operation was verified by inserting a breakpoint at the instruction which writes a digit to the LED display. On each breakpoint, the LED display incremented by exactly one dice face.

DivZero is a short C program which performs a divide by zero. This tested the runtime exception reporting of divide errors.

6. User Manual

6 1 Getting Started

Before using the debugger, the VAX serial channels used to communicate with the target systems must be reserved. This is accomplished using the VMS *allocate* command. For example, if the serial channel TXI7: was used as a host-target serial line, the following VMS command would be entered:

```
$ alloc TXI7:
```

The *allocate* command only needs to be entered once per login session.

The SDK-86 debugger server is contained in PROMs on the debugger expansion board. It is started using the SDK-86 monitor as follows:

1. Reset the SDK-86
2. Press the Go button, then enter FE00:0 and a period.
3. Reset the SDK-86
4. Press the Go button, then enter F000:0 and a period.

Once the debugger server is running, the debugger host software is invoked by executing the VMS command:

```
$ run db
```

On invocation, the debugger reads the file db.ini to establish the names of the targets the user wishes to debug and the corresponding serial channel names and processor names. The file db.ini must be in the same directory as db.exe—the debugger binary file. The format of each entry in the db.ini file is:

target: <targetname> <processorName> <channelName>

Currently, the only <processorName> parameter supported is i8086. Lines starting with two dashes ("--") are skipped during the processing of the db.ini file. These lines may be used to document the db.ini entries.

The following is a sample db.ini file

```
--
--   File: db.ini
--   Function: Startup file for db.exe, single SDK86 target
--
--   Entry format:
--      target: targetName processorName channelName
--
target: sdk86  i8086   txi7:
```

The debugger will attempt to establish a serial connection for each target in the db.ini file. If it is unable to establish a connection, it prints the following message:

Establishing connection with target *name*... failed.

Should this occur, verify that the target serial connection is intact, that the target is running the LDP Server software, and that the db.ini file specifies the correct serial channel name.

The debugger maintains a default target system which is the system refered to by the commands which do not explicitly name a target. On startup, the default target system is set to the last system listed in the db.ini file. The *select* command is used to change or display the default target system.

When the debugger is ready for a command, it issues the prompt character ">"

6.2 Debugger Commands

The debugger uses the VMS terminal driver to read user command lines. Therefore, the standard VMS line editing keys can be used when entering a command line. For example, the up-arrow key will recall the last input line; Control-C will abort the current command.

For all debugger commands, numeric items may be displayed or entered in octal, decimal, or hex format. The default base for displayed and input data is hexadecimal but may be altered with the *radix* command. For input, the default radix may be overridden by appending a base suffix onto the numeric item. A suffix of 'Q' specifies octal, 'T' specifies decimal, and 'H' specifies hexadecimal.

Most debugger keywords can be abbreviated. In the following section which explains each debugger command, the boldface letters in a keyword are the minimal string that must be entered so the keyword will be recognized by the debugger. The debugger's parser is not case sensitive so either upper or lower case input

may be used.

All debugger memory address parameters may be specified in a processor dependent pointer format or in symbolic format. For the 8086 target, pointers are specified in one of the following formats:

```
<segment>:<offset>  -- <segment>, <offset> are numeric
<absoluteAddress>   -- a twenty bit absolute memory location
```

The following are the symbolic address formats:

```
&moduleName.symbolName
&symbolName         -- uses the target's current default module
```

6.2.1 Disassemble Memory

asm <address> { **length** <count> }

This command displays target machine code in symbolic form. The debugger will print out an error message if the currently selected target does not support the disassembly operation.

The optional <count> parameter is a numeric field specifying how many instructions to disassemble. The default <count> is one instruction.

Examples:

```
asm 0FFFF:0                    -- disassemble one instruction
                               -- starting at address FFFF:0

asm &ledclock.displayTime len 12
                               -- disassemble 12 instructions
                               -- beginning at the procedure
                               -- displayTime in the module
                               -- ledclock
```

6.2.2 Display Memory

byte <address> { **length** <count> }
char <address> { **length** <count> }
long <address> { **length** <count> }
pointer <address> { **length** <count> }
word <address> { **length** <count> }

These commands display target memory in one of five processor dependent formats. The <address> parameter specifies the starting target address to start the display; the optional <count> specifies the number of items to print out.

The i8086 personality interprets these five formats as follows:

byte	eight bit unsigned data
char	ASCII data
long	32 bit unsigned data
pointer	32 bit 8086 pointers (i.e. segment:offset)
word	16 bit unsigned data

Examples:

```
byte 0 length 100H          -- prints out 100 (hex) bytes
                            -- starting at address 0

char &TTY.buffer length 25t
                            -- prints out 25 (decimal) ASCII
                            -- characters starting at symbol
                            -- buffer in module TTY.
```

6.2.3 Modify/Fill Memory

byte <address> { **length** <count> } = <valueString>
char <address> { **length** <count> } = <charString>
long <address> { **length** <count> } = <valueString>
word <address> { **length** <count> } = <valueString>

These commands write data into a target's memory. The <address> parameter specifies the first byte in the target memory to start writing data; the optional <count> field specifies how many data items should be written. The <valueString> parameter is a list of numeric items separated by commas or spaces. The <charString> parameter is a quoted string.

Examples:

```
byte 10:0 = ea 0 0 10 0     -- stores the hex bytes 0EAH, 0,
                            -- 0, 10H, 0 into target locations
                            -- 10:0 through 10:4

byte 100 length 1000H = 90  -- fills target memory starting at
                            -- location 100 for 1000 (hex)
                            -- bytes with 90

char 100 = "RIT"            -- stores characters 'R', 'I', and
                            -- 'T' into target memory
```

6.2.4 Download File Into Target Memory

load <filename>

This commands downloads a VMS resident file into a target system. The <filename> parameter is a VMS format file name.

For Intel 8086 targets, the file must be in Intel Object Module Format generated by the LOC86 utility.

Example:

```
load [dsb8674.sdk86]dice.dat
```

6.2.5 Display Target Registers

registers
registers <registerName>

This command displays target processor registers. If the <registerName> field is omitted, all target registers will be displayed. If present, the <registerName> is a string identifying a register from the current target system to be displayed.

For Intel 8086 targets, the valid <registerName> values are AX, BX, CX, DX, SI, DI, SP, BP, CS, DS, SS, ES, IP, and FL.

Examples:

```
reg                     -- displays all target registers
reg cs                  -- displays 8086 code segment
```

6.2.6 Modify Target Registers

registers <registerName> = <valueString>

This command is used to modify a target register or a group of target registers. The <registerName> parameter is a string identifying a register in the currently selected target. The <valueString> parameter is a list of numeric items which are separated by commas or spaces. If multiple items are specified, they are stored in sequential target registers in a target dependent order

For the 8086 target, the ordering is AX, BX, CX, DX, SP, BP, SI, DI, CS, DS, SS, ES, IP, FL. This is the same order that the registers are displayed using the registers command.

Examples:

```
reg ax = 1000Q          -- stores the constant 1000 (octal)
                        -- into the target's AX register

reg ax = 1 22 34 0      -- stores the values 1, 22, 34, 0
                        -- into registers AX, BX, CX, DX
```

6.2.7 Read I/O Port

port <portName>
wport <portName>

These commands allow a user to read data from the target's I/O space. The <portName> parameter is a numeric value specifying the port to read and display.

For 8086 targets, the port command displays a port from the 8 bit I/O space and the wport command displays a port from the 16 bit I/O space.

Examples:

```
port 100H                -- reads from port 100 (hex)

wport FFEA               -- reads port FFEA (hex) from 16
                         -- bit I/O space
```

6.2.8 Write I/O Port

port <portName> = <valueString>
wport <portName> = <valueString>

These commands allow a user to write data to target I/O port. As with the read I/O port commands, the <portName> parameter identifies which target port to write. The port or wport keyword selects the I/O space that is written to. The <valueString> parameter is a list of numeric items separated by spaces or commas.

Examples:

```
port FFEA = C3           -- writes the value C3 (hex) to
                         -- port FFEA (hex) in the 8 bit
                         -- space

wport FFE8 = 1 2 4 8     -- write the values 1, 2, 4, 8 to
                         -- port FFE8 (hex) in the 16 bit
                         -- space
```

6.2.9 Breakpoint Maintenance

breakpoint list
breakpoint set <address-List>
breakpoint clear <breakpointID-List>

These commands set, clear and list execution breakpoints in a target system. The breakpoint set command inserts execution breakpoints at the specified target memory locations; for each specified address, the debugger responds to the set command with a <breakpointID> number which is used to reference the breakpoint in the other breakpoint commands. The breakpoint clear command removes the breakpoints refered to by the list of breakpoint ID's. The breakpoint list command displays the addresses and ID's of all active breakpoints.

The debugger will report an error if the target has filled its breakpoint table, if the user refers to a nonexistent breakpoint ID, or if the user attempt to set a duplicate breakpoint.

Examples:

```
breakpoint set 10:34        -- inserts an execution break at
                            -- target location 10:34. The
                            -- debugger will respond with this
                            -- breakpoint's ID

breakpoint list             -- prints active breakpoints

breakpoint clear 1          -- clear breakpoint whose ID is 1
```

6.2.10 Processor Control

go
go from <address>
step
halt

These commands control the execution of a target program. The *go* command resumes target execution at the target's current program counter. The *go from* command starts the target at the specified target <address>. The *step* command single steps the target one instruction. Finally, the *halt* command stops target execution. The debugger will report an error if the user attempts to start a target which is already running or stop a target which is halted.

Example:

```
go from 0FFFF:0          -- starts target from 0FFFF.0
```

6.2.11 Miscellaneous Functions

module { <moduleName> }
quit
radix { <base> }
select { <targetName> }
synch { <targetName> }

The *module* command sets or displays the default program module name for the current target system. This name qualifies symbolic addresses that do not contain an explicit module name.

Example:

```
module ledclock          -- sets the default module
                         -- name to ledclock

word &hour               -- displays the word variable
                         -- hour in module ledclock
```

The *quit* command ends a debugger session.

The *radix* command displays or sets the base used when displaying or reading numeric data from the terminal. The valid <base> parameters are 8, 10, and 16 (decimal). If the <base> parameter is omitted, the debugger prints out the current radix.

The *select* comand has two functions. If the <targetName> parameter is not specified, the debugger prints out the name, processor type, and serial I/O channel of each target system in the db.ini file. If the <targetName> parameter is present, the debugger sets the default target to the specified target.

The *synch* command is used to restablish the communications link if a target system crashes during a debug session. The <targetName> parameter specifies which target to resynchronize with; if this parameter is not specified, the debugger attempts to resynchronize with the default target.

6.3 Target Runtime Exceptions

The SDK-86 target will detect and report the following runtime exceptions:

- Divide by zero
- NMI generated when the SDK-86 INTR button is pressed
- Breakpoints
- Overflows

When one of these exceptions occurs, the debugger will print one of the following messages identifying the target which encountered the exception and the exception reason:

Target *targetName* : Divide by zero at *address*
Target *targetName* : Intr Button at *address*
Target *targetName* : Breakpoint at *address*
Target *targetName* : Overflow at *address*

7. Conclusions and Future Work

This thesis presented the design and implementation of a distributed debugger. The design supports the debug of an arbitrary number of target systems connected to the host using RS-232 serial channels. The debugger is used from a single VAX terminal and supports a uniform command language for all target types.

The implementation is extensible so other target systems may be supported by adding a personality module to the debugger host and implementing an LDP Server for the target system. Other Intel processors could most readily be supported due to the similarity of their architectures. For example, an 8085 personality module could use many of the routines written for the 8086; the disassembler and download routines would need to be coded from scratch.

Much effort was spent to make effective use of the communications channel bandwidth. The goals were to provide good response times for user requests which read or write target memory and download a target system. This was accomplished by minimizing the overhead of the communications protocols, using piggybacked acks, and optimizing the size of communications buffers. Download times were cut by a factor of three compared to the previous SDK-86 download method. This is due to the higher communications bit rate, encoding each downloaded byte in eight bits rather than sixteen bits, and compressing records which fill blocks of target memory to a single LDP RepeatData request.

C proved to be an excellent implementation language for the host and target. It allowed most of the target code to be debugged on the host before being checked out on the target. The main problem encountered with C was the lack of a suitable runtime library for the 8086. Now that this has been coded, C could be used for many other SDK-86 applications.

The cost of the prototype target expansion hardware for the SDK-86 was $70 for the integrated circuits, sockets, prototype board, and connectors—this is slightly higher than the $50 goal. This price could be reduced if a printed circuit board was fabricated for the expansion hardware.

A problem arose during implementation of the debugger due to the late delivery of DecServer 1.2 LAT software. The VAX supports two types of serial channels—the forced connection which is a serial line directly wired to the VAX and a LAT connection which is a virtual serial channel implemented using the Ethernet. The majority of the VAX serial channels are implemented using a LAT. The problem is that the current LAT software does not allow direct addressing of a serial port from a program; it will be supported in the next version of LAT software. This makes the LAT serial

connections difficult to use for the debugger application.

A message from Andy Potter [20] indicates that software could be written to allow a user logged in to a LAT serial line to run a program which transfers frames between the LAT serial channel and VMS mailboxes. Once the program is started, the serial line would be switched from the user's terminal to the target system. This program and the debugger process would communicate using the mailboxes; one mailbox would be used to enqueue write requests to the target. A second mailbox would be used to buffer received target frames. This would require modifications to the host Datalink module to use the mailboxes to perform I/O rather than direct VMS I/O requests.

The debugger could be extended or enhanced in many ways. Other processors such as the Intel 8085 or Motorola 68000 could be supported. Other features such as logging of a debug session on a file, a command history mechanism, or a command to compare target memory with a VMS file would also be desirable.

References

1. E. T. Fathi, M. Krieger, "Multiple Processor Systems: What, Why, When", IEEE Computer, March 1983, page 23

2. R. L. Glass, "Real Time: The Lost World of Software Debugging and Testing", CACM 23, 5, May 1980, page 264

3. SDK-86 MCS-86 System Design Kit User's Guide, Intel Corporation, Order Number 9800698-02 Rev B

4. R. Goering, "Logic Analyzers Offer Wide Choice of Performance and Price", Computer Design, September 1, 1985, page 25

5. B. Lampson, M. Paul, H.J. Siegert (Eds.), Distributed Systems -- Architecture and Implementation, Springer-Verlag, 1981, page 363

6. T. A. Cargill, "Implementation of the Blit Debugger", Software -- Practice and Experience, 15, 2, February 1985, Page 153

7 B. Lampson, M. Paul, H.J. Siegert (Eds.), Distributed Systems -- Architecture and Implementation, Springer-Verlag, 1981, page 442

8. C. K. Walter, "DELTA -- The Universal Debugger for CP-6", Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging. SIGPLAN 18, 8, August 1983, page 203

9. P. Fritzon, "A Systematic Approach to Advanced Debugging Through Incremental Compilation", Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging. SIGPLAN 18, 8, August 1983, page 130

10. P. Fritzon, "Preliminary Experience from the DICE system: A Distributed Incremental Compiling Environment", Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, SIGPLAN 19, 5, May 1984, page 113

11. J. R. Ward, "UNIX as an environment for non-UNIX software development, A Case History", SIGSOFT 10, 3, July 1985, page 95

12. C. Welles, W Milliken, "Loader Debugger Protocol, RFC-909", July 1984

13. 8086 Relocatable Object Module Formats, Intel Corporatioin, Order No. 121748-001

14. IBM Synchronous Data Link Control General Information, IBM Corporation, Order No. GA27-3093-2

15. A. S. Tanenbaum, Computer Networks, Prentice Hall Inc., New Jersey

16. T. Ritter, "The Great CRC Mystery", Dr. Dobb's Journal of Software Tools, February 1986, page 26

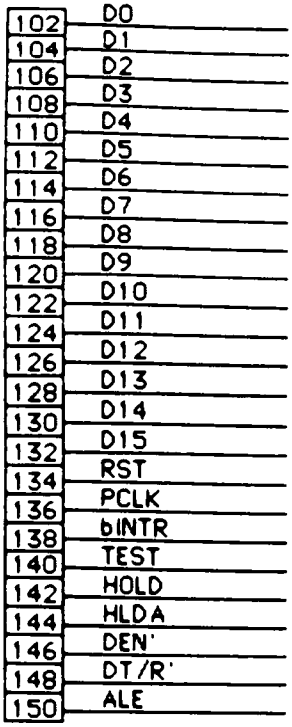
17. H. C. Folts (Ed.), Data Communications Standards, Edition II, McGraw-Hill, New York, page 976

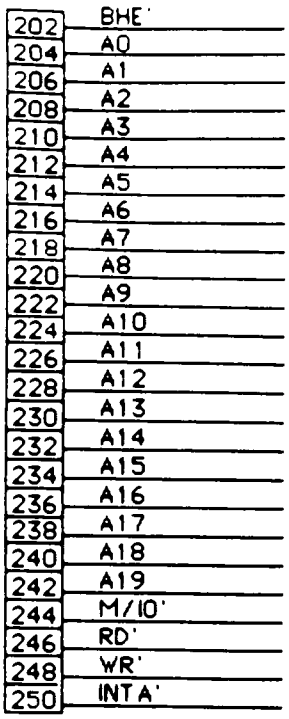
18. VAX/VMS System Services Reference Manual, Digital Equipment Corporation, Order No. AA-Z501B-TE

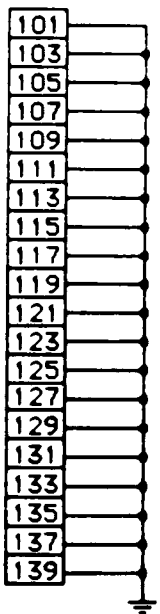
19 VAX/VMS Run-Time Library Routines Reference Manual, Digital Equipment Corporation, Order No. AA-Z502B-TE

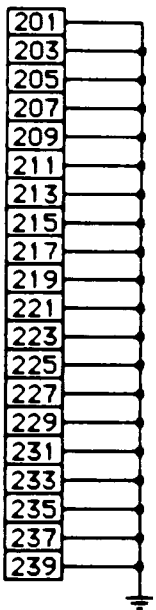
20. Andy Potter, (member of the ISC Staff at Rochester Institute of Technology) personal communication

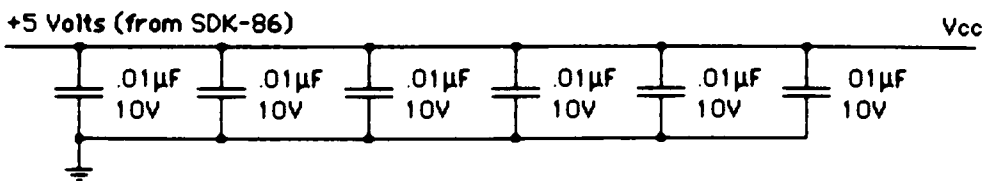
21. A. Petez, "Byte-wise CRC Calculations", IEEE Micro, June 1983, Page 40



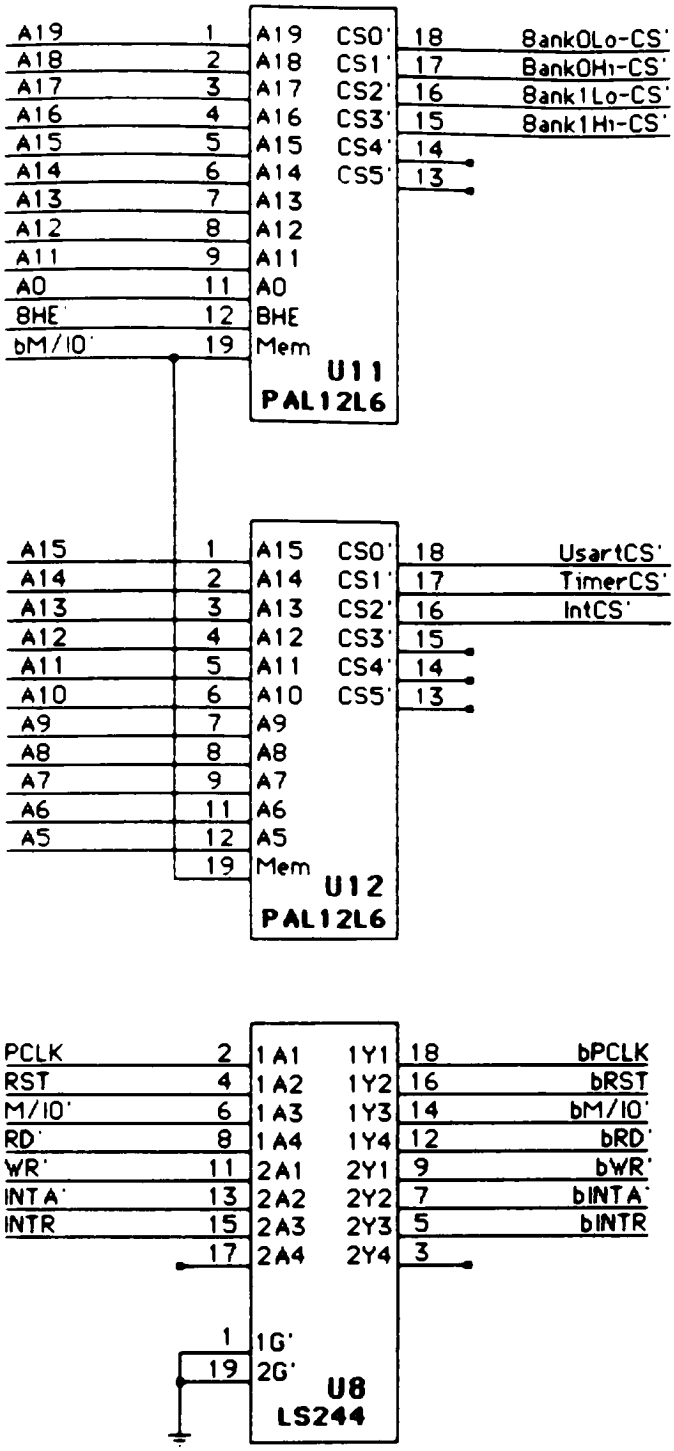




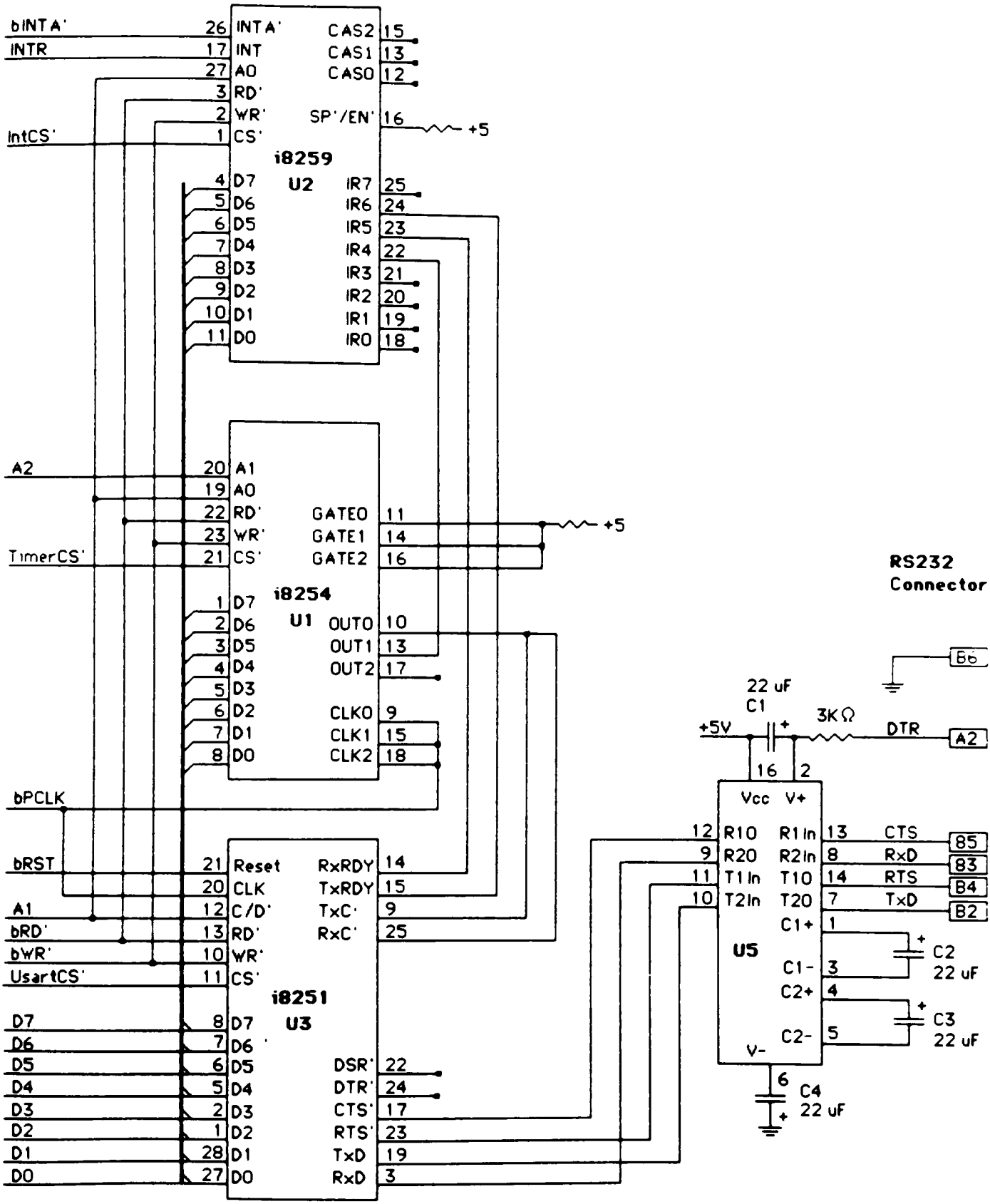




SDK-86 Bus Ex
Interface
Vcc Bypass Cap

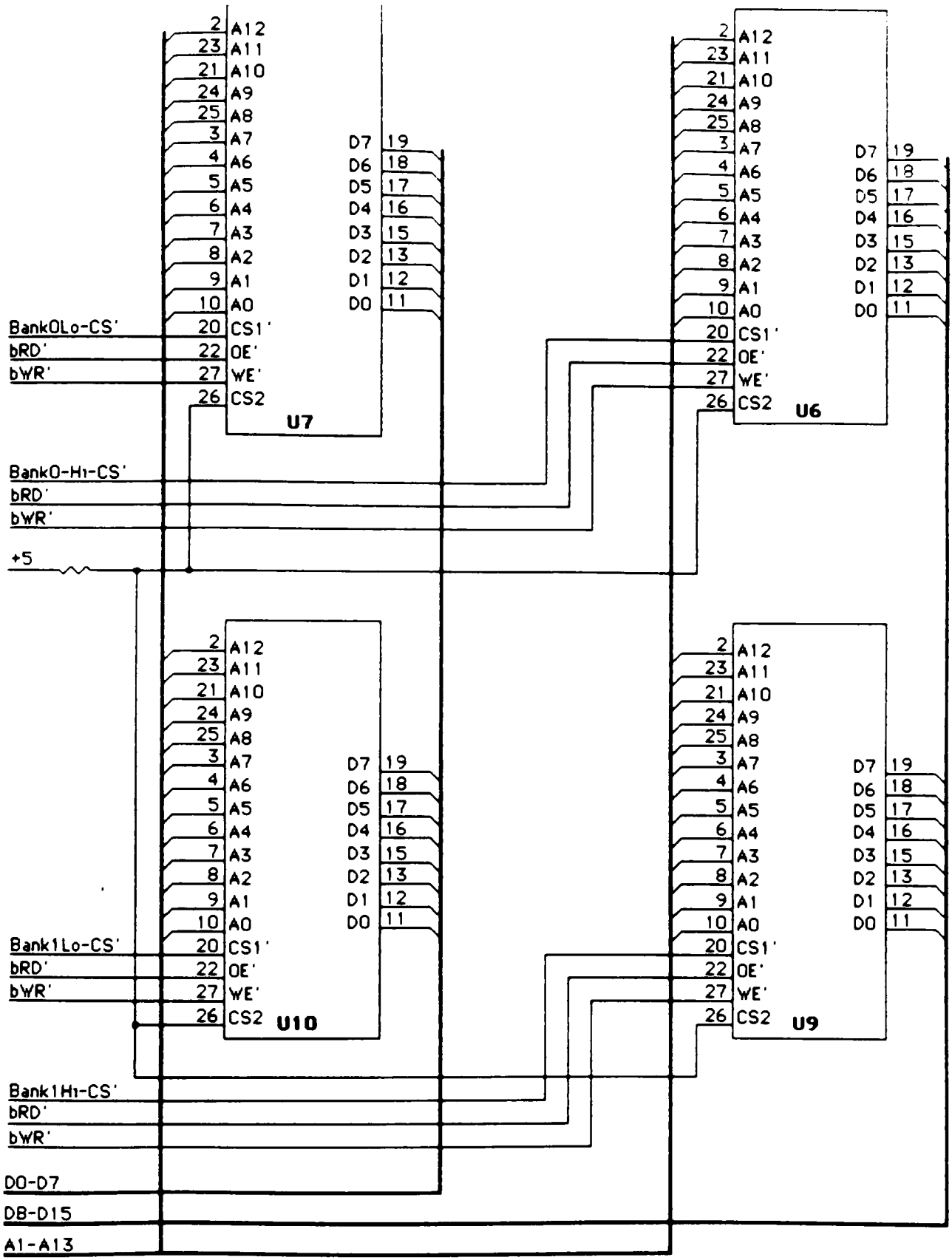


IO Chip Selects
RAM/ROM Chip Selects
Control Signal Buffers



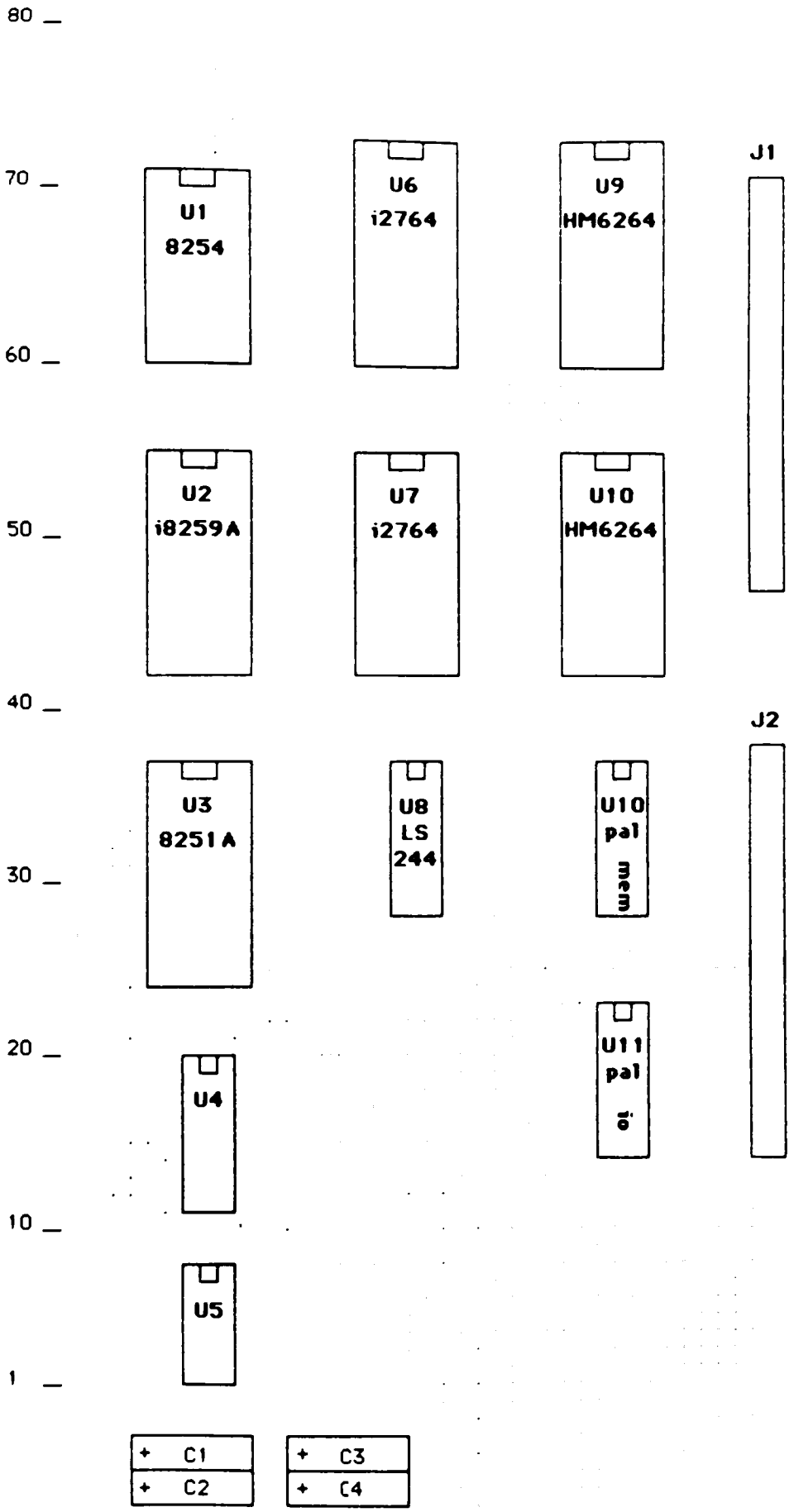
Appendix A

Serial I/O
Baud Rate Generator/Timer
Interruot Controller


U7
U6
U10
U9
A12
A11
A10
A9
A8
A7
A6
A5
A4
A3
A2
A1
A0
CS1'
OE'
WE'
CS2
D7
D6
D5
D4
D3
D2
D1
D0
Bank0Lo-CS'
bRD'
bWR'
Bank0-Hi-CS'
+5
Bank1Lo-CS'
Bank1Hi-CS'
D0-D7
D8-D15
A1-A13

IC Power and Ground Connections

IC	Type	Pins	+5	GND
U1	i8254	24	24	12
U2	i8259A	28	28	14
U3	i8251A	28	26	4
U4	Header	20		
U5	Maxim MAX232	16	16	15
U6	i2764A EPROM	28	28	14
U7	i2764A EPROM	28	28	14
U8	74LS244	20	20	10
U9	Hitachi HM6264	28	28	14
U10	Hitachi HM6264	28	28	14
U11	MMI PAL 12L6	20	20	10
U12	MMI PAL 12L6	20	20	10



Protoboard layout

Appendix A

```
PAL12L6                    PAL DESIGN SPECIFICATION

PAL1                       DAVID BLACKMAN 02/21/86

Memory Chip Select PAL for Debugger Expansion Board

RIT

A19  A18  A17  A16  A15  A14  A13  A12  A11  GND
A0   BHE  /CS5 /CS4 /CS3 /CS2 /CS1 /CS0 MEM  VCC

CS0 = A19*A18*A17*A16*/A15*/A14*/A0*MEM
CS1 = A19*A18*A17*A16*/A15*/A14*/BHE*MEM
CS2 = A19*A18*A17*A16*/A15*A14*/A0*MEM
CS3 = A19*A18*A17*A16*/A15*A14*/BHE*MEM

FUNCTION TABLE

A19 A18 A17 A16 A15 A14 A0 BHE MEM /CS0 /CS1 /CS2 /CS3

;AAAAAA A B M CCCC
;111111 0 H E SSSS
;987654 0 E M 0123 Comments

-------------------------
 HHHHLL L L H LLHH
 HHHHLL L H H LHHH
 HHHHLL H L H HLHH
 HHHHLH L L H HHLL
 HHHHLH L H H HHLH
 HHHHLH H L H HHHL
 XXXXXX X X L HHHH
 LXXXXX X X X HHHH
 XLXXXX X X X HHHH
 XXLXXX X X X HHHH
 XXXLXX X X X HHHH
 -------------------------
```

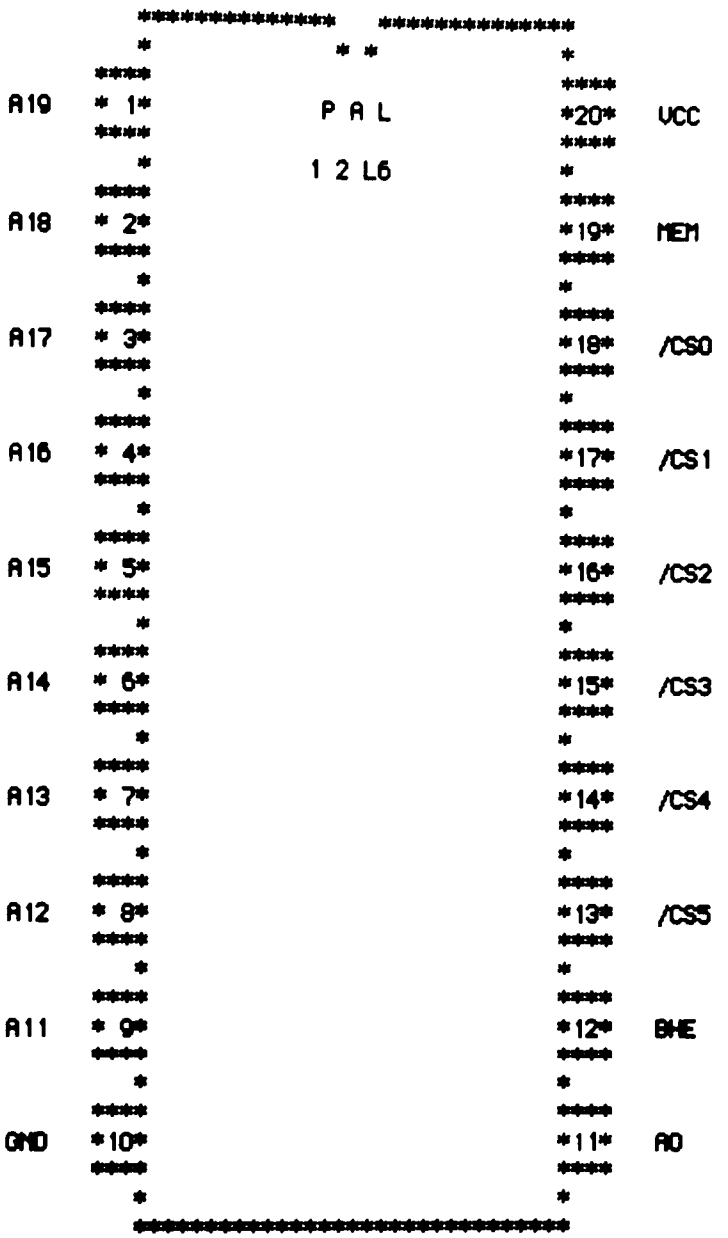
DESCRIPTION

This PAL provides chip select signals to the memory devices on the debugger expansion board. It is set up for decoding two 16K banks of memory. Each 16K bank consists of 8K even (low) address and 8K odd (high) addresses.

Address Range

F0000..F3FFF	Bank0, A0 and BHE' select low and high byte respectively
F4000..F7FFF	Bank1, A0 and BHE' select low and high byte respectively

Memory Chip Select for Debugger Expansion Board



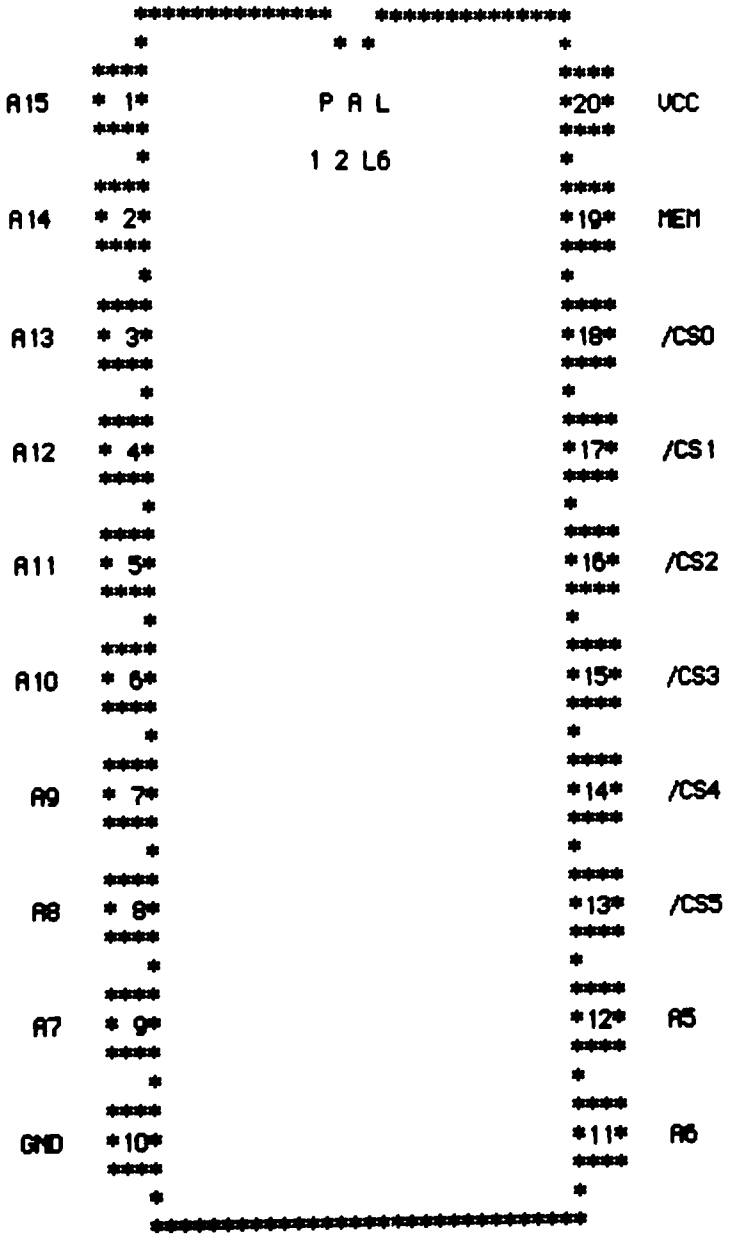
```
PAL12L6                    PAL DESIGN SPECIFICATION

PAL1                       DAVID BLACKMAN 02/21/86

IO Chip Select PAL for Debugger Expansion Board

AIT

A15  A14  A13  A12  A11  A10  A9   A8   A7   GND
A6   A5   /CS5 /CS4 /CS3 /CS2 /CS1 /CS0 MEM  VCC

CS0 = /A15*/A14*/A13*/A12*/A11*/A10*/A9*A8* /A7*/A6*/A5* /MEM
CS1 = /A15*/A14*/A13*/A12*/A11*/A10*/A9*A8* /A7*/A6*A5*  /MEM
CS2 = /A15*/A14*/A13*/A12*/A11*/A10*/A9*A8* /A7*A6*/A5*  /MEM
CS3 = /A15*/A14*/A13*/A12*/A11*/A10*/A9*A8* /A7*A6*A5*   /MEM
CS4 = /A15*/A14*/A13*/A12*/A11*/A10*/A9*A8* A7*/A6*/A5*  /MEM
CS5 = /A15*/A14*/A13*/A12*/A11*/A10*/A9*A8* A7*/A6*A5*   /MEM

FUNCTION TABLE

A15 A14 A13 A12 A11 A10 A9 A8 A7 A6 A5 MEM
/CS0 /CS1 /CS2 /CS3 /CS4 /CS5

;AAAAAAAA AAA M CCCCCC
;11111100 000 E SSSSSS
;54321098 765 M 012345 Comments

---------------------------
 XXXXXXXX XXX H HHHHHH
 LLLLLLLH LLL L LHHHHH
 LLLLLLLH LLH L HLHHHH
 LLLLLLLH LHL L HHLHHH
 LLLLLLLH LHH L HHHLHH
 LLLLLLLH HLL L HHHHLH
 LLLLLLLH HLH L HHHHHL
---------------------------
```

DESCRIPTION

This PAL generates 6 chip selects in the IO space of the 8086. Each chip select enables one device. A4..A0 are used to select a register within a device enabled by one of the 6 chip selects.

IO Address Range	Chip Select Active	Device Selected
100..11F	0	i8251A USART
120..13F	1	i8254 Timer
140..15F	2	i8259A Interrupt Controller
160..17F	3	Not Used
180..19F	4	Not Used
1A0..1BF	5	Not Used

IO Chip Select for Debugger Expansion Board



Appendix B. How to build db.exe

The file db.com contains a script to recompile and relink db.exe. It can recompile all the debugger sources and relink or just relink. The script creates the executable image file 'db exe'

```
$ @db              ! this form just relinks
$ @db c            ! the c parameter specifies recompile & relink
```

The following is a listing of the db.com script:

```
$if p1.eqs."" then goto nocompile
$if p1.eqs."C" then goto compile
$compile:
$!
$!  Debugger host modules.
$!
$write sys$output "dba.. "
$cc dba.c
$write sys$output "dbb..."
$cc dbb.c
$write sys$output "dbc.  "
$cc dbc.c
$write sys$output "dbd..."
$cc dbd.c
$write sys$output "dbe.. "
$cc dbe.c
$write sys$output "dbf..."
$cc dbf.c
$write sys$output "dbg.. "
$cc dbg.c
$write sys$output "dbh..."
$cc dbh.c
$write sys$output "dbi..."
$cc dbi.c
$!
$!  Command parser fsm table
$!
$write sys$output "dbt..."
$macro dbt.mar
```

```
$!
$!   LDP print frame and serialize routines
$!
$write sys$output "ldpp.   "
$cc ldpp.c
$write sys$output "ldps.   "
$cc ldps.c
$!
$!   i8086 personality
$!
$write sys$output "i8086.   "
$cc i8086.c
$write sys$output "i8086a..."
$cc i8086a.c
$write sys$output "i8086b.. "
$cc i8086b.c
$!
$!   Symbol table
$!
$write sys$output "symtab.. "
$cc symtab.c
$!
$!   Terminal handler
$!
$write sys$output "terminal.   "
$cc terminal.c
$!
$!   Buffer and linked list queue handlers
$!
$write sys$output "buffer..."
$cc buffer.c
$write sys$output "queue..."
$cc queue.c
$!
$!   Communications software.
$!
$write sys$output "datalink..."
$cc datalink.c
$write sys$output "transport.. "
$cc transport.c
$!
$write sys$output "crash..."
$cc crash.c
```

```
$!
$nocompile:
$write sys$output "linking db.  "
$link/executable = db.exe                              -
     dba.obj, dbb.obj, dbc.obj, dbd.obj,               -
     dbe.obj, dbf.obj, dbg.obj, dbh.obj,               -
     dbi.obj, dbt.obj,                                 -
     ldpp.obj, ldps.obj,                               -
     i8086.obj, i8086a.obj, i8086b.obj,                -
     symtab.obj, terminal.obj,                         -
     buffer.obj, queue.obj,                            -
     datalink.obj, transport.obj,                      -
     crash.obj,                                        -
     sys$login:options_file/opt
```

Appendix C. How to build the LDP Server

The file ldpserver.com contains a script of VMS commands to rebuild the LDP Server software. It can recompile all the LDP Server sources and relink or relink only:

```
$ @ldpserver          ! relink only
$ @ldpserver c        ! recompile and relink
```

All LDP Server and db.exe sources must be in separate directories due to name conflicts. The ldpserver.com script expects to find the Runtime Library file ldp.lib in the logical directory 'lib:'

On completion of this script, the file ldpserver.hex contains the Intel hex representation of the ldpserver code. The script is set up to locate the code at F4000 hex; the ldpserver's data and stack segment are located at F0000 hex.

The following is a listing of the ldpserver script:

```
$if p1.eqs."" then goto nocompile
$if p1.eqs."C" then goto compile
$write sys$output "Use C parameter to compile LDP software"
$exit
$compile:
$write sys$output "LDPA.. "
$cc86 LDPa.c86 SMALL OPTIMIZE(2)
$write sys$output "LDPB..."
$cc86 LDPb.c86 SMALL OPTIMIZE(2)
$write sys$output "LDPC. "
$cc86 LDPc.c86 SMALL OPTIMIZE(2)
$write sys$output "LDPD. "
$cc86 LDPd.c86 SMALL OPTIMIZE(2)
$write sys$output "LDPE..."
$cc86 LDPe.c86 SMALL OPTIMIZE(2)
$write sys$output "LDPF..."
$cc86 LDPf.c86 SMALL OPTIMIZE(2)
$write sys$output "LDPG..."
$cc86 LDPg.c86 SMALL OPTIMIZE(2)
$write sys$output "LDPM..."
$cc86 LDPm.c86 SMALL OPTIMIZE(2)
$write sys$output "LDPS..."
$cc86 LDPs.c86
$!
```

```
$write sys$output "Buffer   "
$cc86 Buffer.c86 SMALL OPTIMIZE(2)
$write sys$output "CRC.. "
$asm86 crc.a86
$write sys$output "DataLink.  "
$cc86 DataLink.c86 SMALL OPTIMIZE(2)
$write sys$output "Int.. "
$asm86 int.a86
$write sys$output "Timeout.  "
$cc86 Timeout.c86  SMALL OPTIMIZE(2)
$write sys$output "Transport..."
$cc86 Transport.c86 SMALL OPTIMIZE(2)
$!
$!
$nocompile:
$link86 lib:startup.obj,      -
     buffer.obj,              -
     crc.obj,                 -
     datalink.obj,            -
     int.obj,                 -
     LDPA.obj,                -
     LDPB.obj,                -
     LDPC.obj,                -
     LDPD.obj,                -
     LDPE.obj,                -
     LDPF.obj,                -
     LDPG.obj,                -
     LDPM.obj,                -
     LDPS.obj,                -
     timeout.obj,             -
     transport.obj,           -
     lib:LDP.lib to LDPServer.lnk
$write sys$output ""
$loc86 LDPServer.lnk to LDPServer -
    segsize (stack (6)) -
    order (segments (code, ??seg, data, stack, const, memory)) -
    ad (sm (code (0F0000H), data (0F4000H)))
$write sys$output ""
$oh86 LDPServer
```

Appendix D. How to build the Runtime Library

The VMS script file ldplib.com contains the commands which recompile/reassemble the sources in the Runtime Library. All Runtime Library sources are stored in a single directory whose logical name is 'lib:'. After the binary files have been created, they are merged into the file ldp.lib with the following commands:

```
$ lib86
create ldb.lib
add common.obj to ldp.lib
add crash.obj to ldp.lib
add lp.obj to ldp.lib
add portio.obj to ldp.lib
add timer.obj to ldp.lib
add sio.obj to ldp.lib
add sioA.obj to ldp.lib
exit
```

These commands must be entered manually as lib86 does not support input from a script file.

The following is a listing of the file ldplib.com:

```
$cc86     common.c86 SMALL OPTIMIZE(2)
$asm86    crash.a86
$asm86    LP.a86
$asm86    PortIO.a86
$cc86     Timer.c86 SMALL OPTIMIZE(2)
$cc86     sio.c86 SMALL OPTIMIZE(2)
$asm86    sioA.a86
$asm86    startup.a86
```

Appendix E. Using the VMS 8086 Development Tools

A full complement of 8086 development tools are available on the VAX. These tools include C, PL/M-86, and Pascal compilers, an 8086 assembler, and Intel's 8086 utilities.

To use these tools, the following DCL commands must be executed; a good place to put these commands is the login.com file which is run each time a user logs in to VMS:

```
$ASSIGN LIB$DISK:[ACCLIB.INTEL] SYS$MDX
$PLM86 :== "$SYS$MDX:PLM86 PLM86"
$PASCAL86 :== "$SYS$MDX:PASCAL86 PASCAL86"
$ASM86 :== "$SYS$MDX:ASM86 ASM86"
$@LIB$ACC:[CC86]SETUP86C
$LIB86 :== "$SYS$MDX:LIB86 LIB86"
$LINK86 :== "$SYS$MDX:LINK86 LINK86"
$LOC86 :== "$SYS$MDX:LOC86 LOC86"
$OH86 :== "$SYS$MDX:OH86 OH86"
$ASSIGN SYS$OUTPUT: CO:
$ASSIGN SYS$OUTPUT: CI:
$ASSIGN [] F1:
```

This script makes the following commands available:

I. Compiler & assembler

```
$ asm86 programName (controls)
$ plm86 programName (controls)
$ pascal86 programName (controls)
$ cc86 programName (controls)
```

II. Link & Locate

```
$ link86 inputList (to objectFile) (controls)
$ loc86 inputFile (to objectFile) (controls)
```

III. Library maintainence

```
$ lib86
```

IV. File Format Conversion

```
$ oh86 inputFile (to objectFile)
```

The following is an example of the use of the 8086 development tools. This script builds a program which makes the SDK-86 into a clock using the LED's to display the time. The script creates two files: LEDClock.dat which can be downloaded using the debugger and LEDClock.hex which can be downloaded using the SDK-86 PROM monitor.

```
$ cc86 LEDClock.c86 debug
$ asm86 Startup.a86 debug
$ link86 startup.obj, LEDClock.obj, lib:sdk86.lib to LEDClock.lnk
$ loc86 LEDClock.lnk to LEDClock.dat segsize (stack (2)) -
        addresses (segments (code (100H))) noinitcode
$ oh86 LEDClock.dat to LEDClock.hex
```

Appendix F. VMS System Services

SYS$ASCEFC (efn, name, prot, perm)

Associate Common Event Flag Cluster -- causes a named common event flag cluster to be associated with a process and to be assigned a process-local cluster number for use with other event flag services.

SYS$ASCTIM (timlen, timbuf, timadr, cvtflg)

Convert Binary time to ASCII string -- converts an absolute or delta time from 64-bit system format to an ASCII string.

SYS$ASSIGN (devnam, chan, acmode, mbxnam)

Assign I/O Channel -- provides a process with an I/O channel

SYS$BINTIME (timebuf, timadr)

Convert ASCII String to Binary Time -- converts an ASCII string to absolute or delta time value in 64 bit system format. Good for SetTime (SYS$SETIMR) or Schedule Wakeup (SYS$SCHDWK)

SYS$CANCEL (chan)

Cancel I/O on a channel -- cancels all pending I/O requests on a specified channel.

SYS$CANTIM (reqidt, acmode)

Cancel Timer -- cancels all or a selected subset of the Set Timer requests previous issued by the current image executing in a process. The request identification is that which was specified in the Set Timer call.

SYS$CLREF (efn)

Clear Event Flag -- sets an event flag in a local or common event flag cluster to 0.

SYS$DACEFC (efn)

Dissasociate Common Event Flag Cluster -- Releases the calling process's association with a common event flag cluster.

SYS$DASSGN (chan)

Deassign I/O Channel -- releases an I/O channel that was acquired using SYS$ASSIGN

SYS$DLCEFC (name)

Delete Command Event Flag Cluster -- marks a permanent common event flag cluster for deletion. The cluster is actually deleted when no more processes are associated with it.

SYS$QIO[W] (efn, chan, func, iosb, astadr, astprm, p1, p2, p3, p4, p5, p6)

Queue I/O Request -- queues an I/O request for a channel associated with a device. Asynchronous and synchronous calls are supported. The parameters p1, ..., p6 are device and function specific.

SYS$SETEF (efn)

Set Event Flag -- sets an event flag in a local or common event flag cluster. The return value indicates whether the specified flag was previously set or clear. Once an event flag is set, processes waiting for the event flag to be set resume execution.

SYS$SETIMR (efn, daytim, astadr, reqidt)

Set Timer -- sets a timer to expire at a specified time. When the timer expires, an event flag is set and optionally, an AST routine is executed.

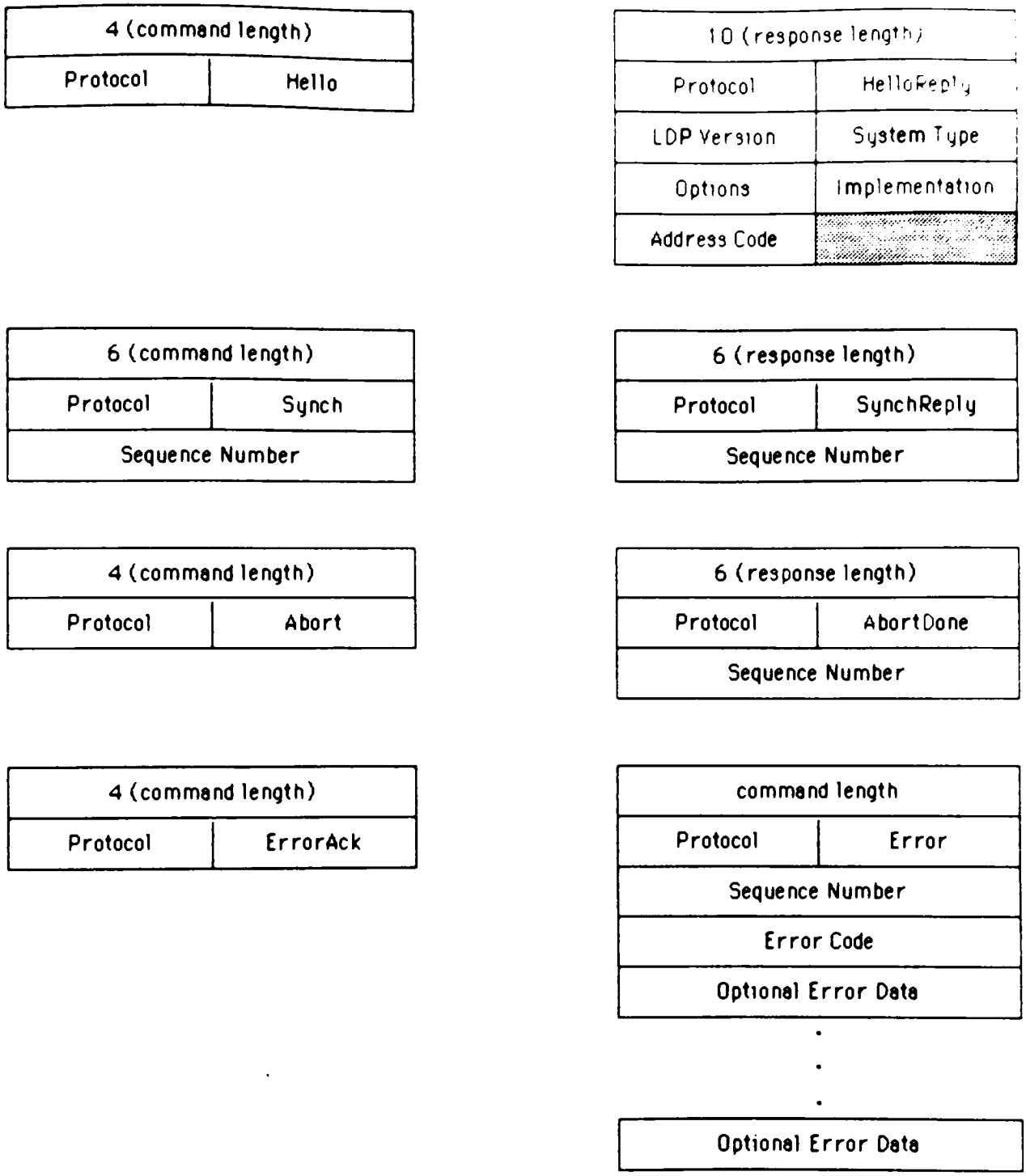
SYS$WAITFR (efn)

Wait for Single Event Flag -- waits until a specified event flag is set.

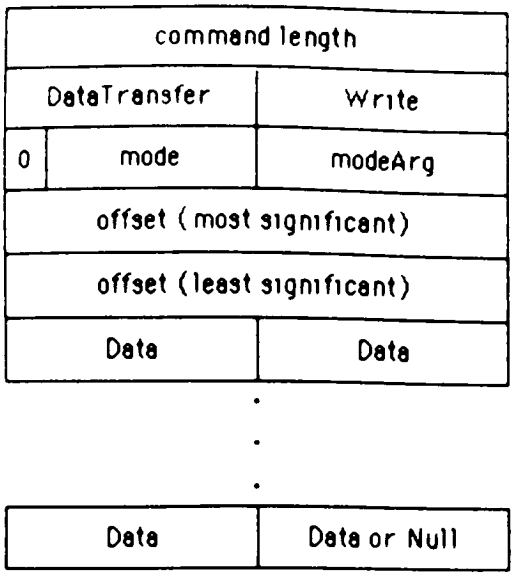
SYS$WFLOR (efn, mask)

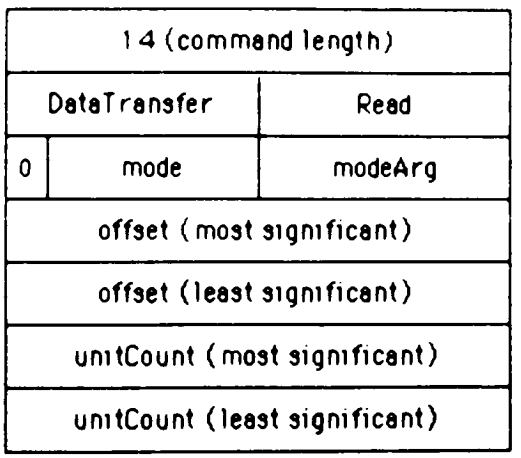
Wait For the Logical OR of Event Flags -- suspends a process until any one of the specified event flags are set.

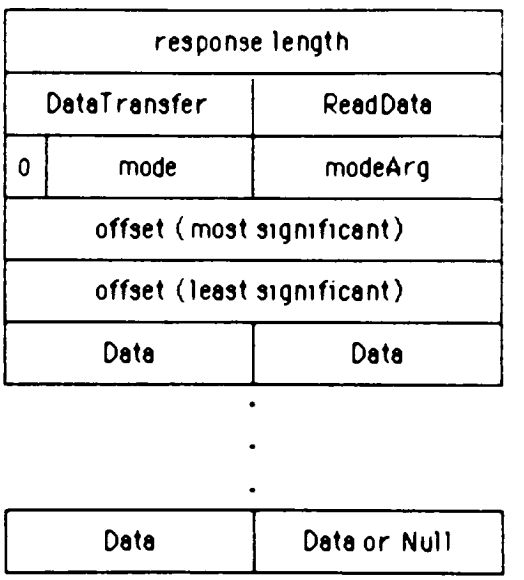
LDP Protocol Class Frame Formats

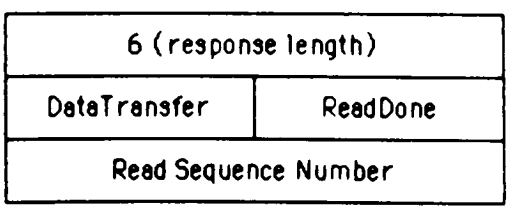


LDP Data Transfer Class Frames



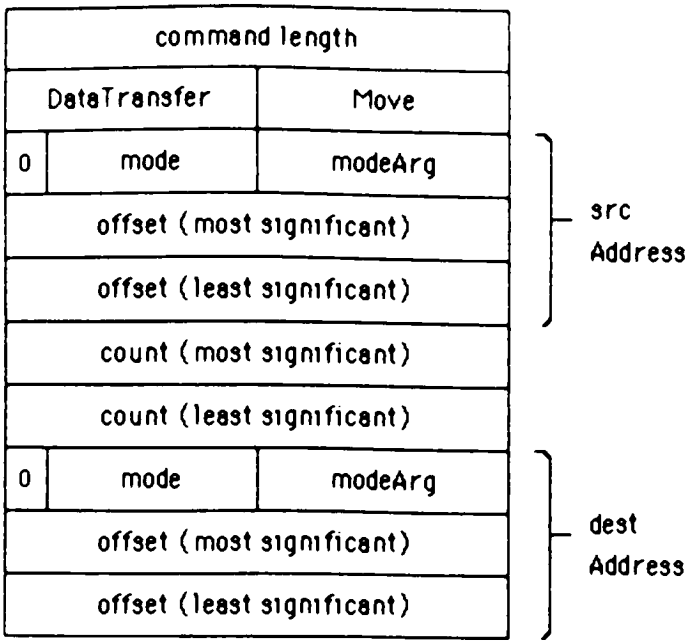


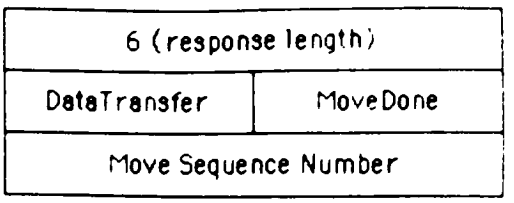


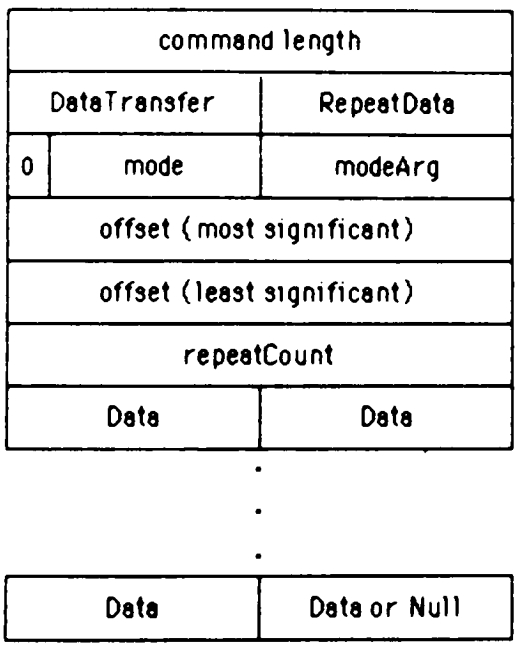


Appendix G

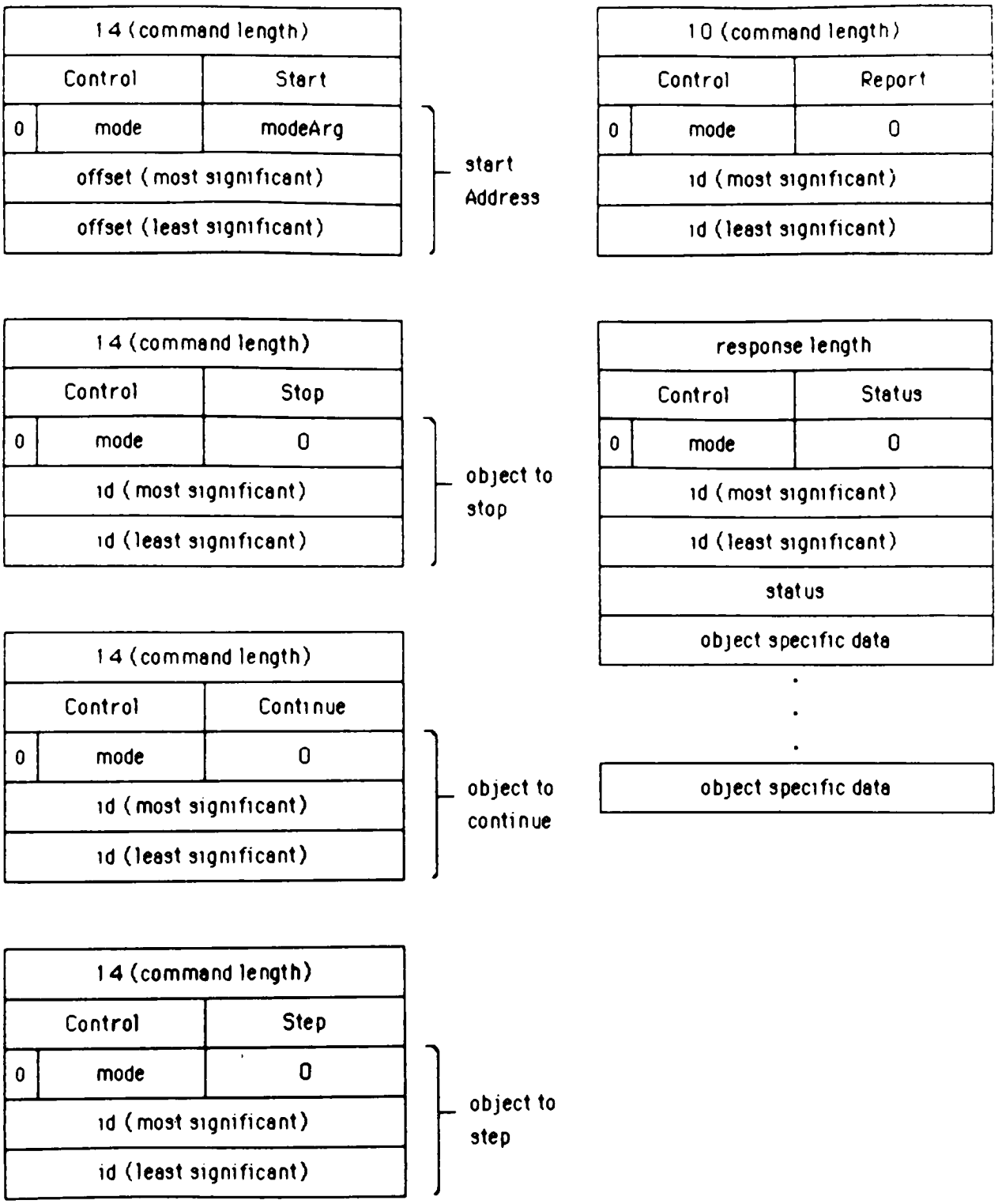
LDP Data Transfer Class Frames





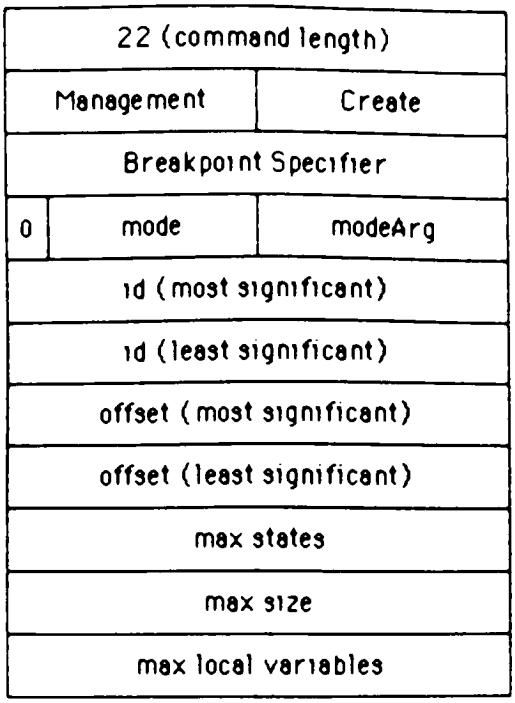


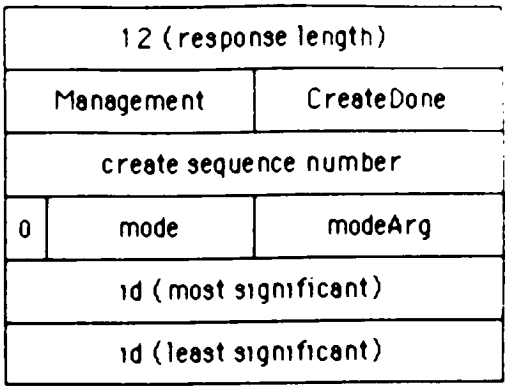
LDP Control Class Frame Formats



Appendix G

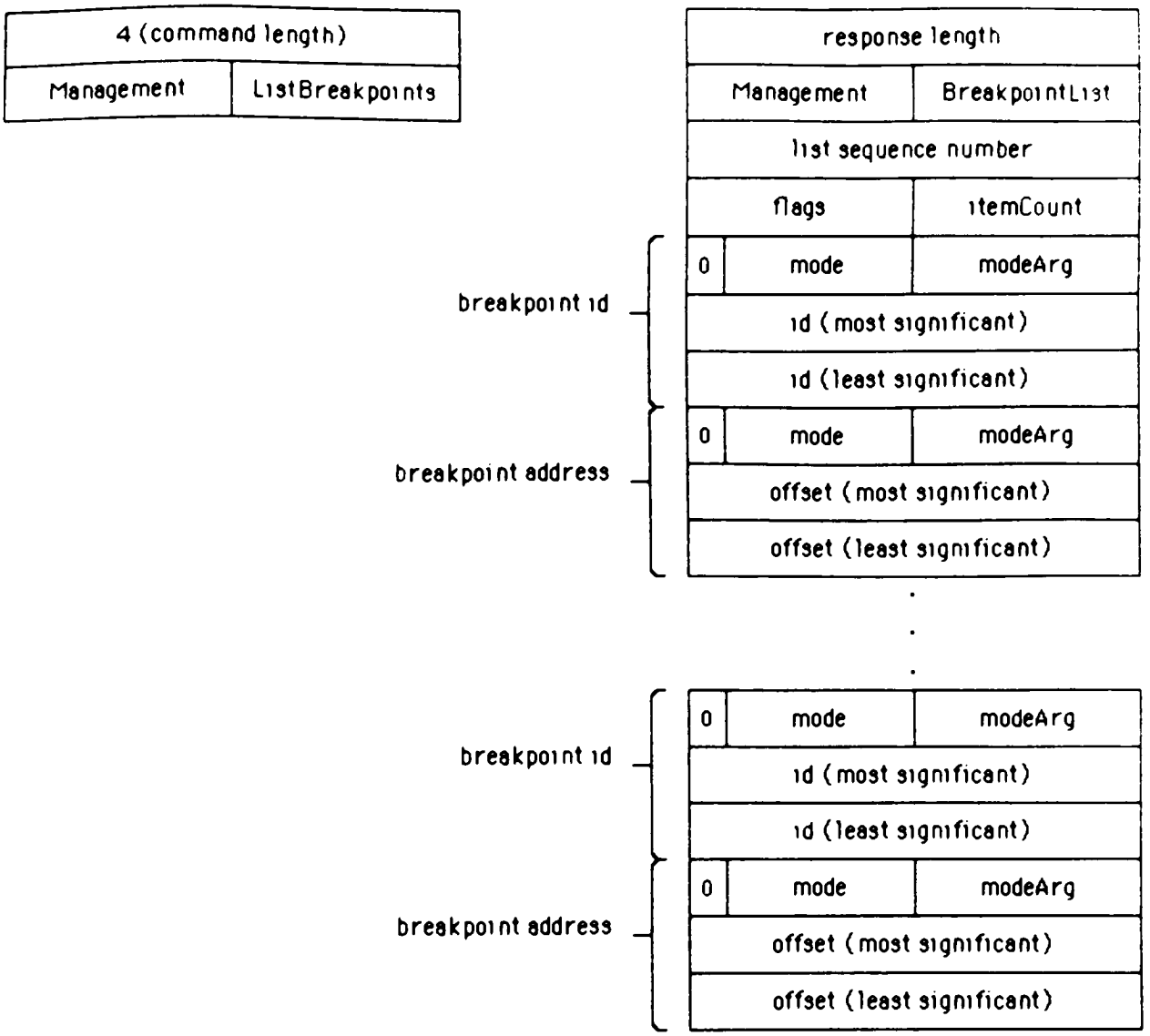
LDP Management Class Frame Formats





Appendix G

LDP Management Class Frame Formats



Appendix G

Appendix H. Adding a New Debugger Target

This appendix describes how to add debugger support for a new target processor The Intel 8085 processor will serve as an example of a target to be added.

Three steps are required to support a new target processor

1. The implementation of the target resident software
2. The implementation of the target's personality module
3. An edit to the host to recognize the new processor name

The target resident software consists of a communications package and a server for Loader/Debugger (LDP) requests. The communications package implements the Datalink and Transport protocols specified in sections 2.2.3 and 2.2.4. The server must handle all LDP Basic Debugger requests described in [12] These include the following host LDP requests:

- Abort
- Continue
- Create (breakpoint)
- Delete (breakpoint)
- Hello
- List Breakpoints
- Move
- Read
- Repeat Data
- Report
- Start
- Step
- Stop
- Synch
- Write

The target should also report user program exceptions and protocol errors with the LDP Exception and Error target reply messages. Exception codes should be placed in a header file which can be shared by the target and the host software.

The second step performed when supporting a new processor is the implementation of the processor's personality module. The debugger host software contains a machine dependent personality module for each type of target processor. Each personality module contains procedures and data which characterize a processor.

The db_Personality data structure bundles together the processor dependent procedures and data. Each personality module must export a procedure which returns instances of the db_Personality structure, initialized with the personality procedures and data. The format of the db_Personality structure is:

```
struct db_Personality {  /* this definition is in db.h */
        char *name;
        unsigned nRegisters;
        int (*StringToLDP_Address) ();
        int (*LDP_AddressToString) ();
        int (*LDP_ExceptionToString) ();
        int (*nBits) ();
        int (*Disassemble) ();
        int (*Download) ();
        int (*PrintItem) ();
        int (*Serialize) ();
        };
```

Table H.1, found at the end of this appendix, contains a brief description of each db_Personality field; refer to section 3.2.16 for detailed information on each field.

The 8086 personality module contains many procedures which would be suitable for the 8085 with minor editing. The principle exceptions are the Download and Disassemble procedures. The program *ProcTest,* found in the debugger host directory, may be used to test personality procedures.

After the personality procedures have been coded and tested, a procedure must be written to return instances of the db_Personality structure. For the 8085 target, this procedure could be named *i8085_Instance*. This procedure would perform the following:

- Allocate storage for the personality data structure
- Initialize the *name* field
- Initialize the *nRegisters* field
- Set up the pointers to the personality procedures
- Return a pointer to the new 8085 instance

The third step required to support a new processor is to modify the host debugger software to recognize the new processor name in db.ini files. For the 8085 example, the procedure db_AddTarget in the module dba.c would be modified as follows:

```
if (EquivalentString (processorName, "i8086"))
        newTarget->processor = i8086_Instance (),
else if (EquivalentString (processorName, "i8085"))
        newTarget->processor = i8085_Instance ();
else ... /* Unknown processor name! */
```

This edit allows the debugger to create an instance of the 8085 personality structure for each "i8085" processor name encountered in the db.ini initialization file.

After these modifications have been performed, the following db.ini file could be used for a system containing one 8085 processor and one 8086 processor:

```
--
--  File: db.ini
--  Function: Startup file for the debugger host. Used for a system
--      consisting of an 8085 processor and an 8086 processor.
--
--
target: sdk86 i8086 txi7:  -- the sdk86 is an 8086 based target
target: sdk85 i8085 txi6:  -- the sdk85 is an 8085 based target
```

Field	Function
name	Pointer to a string containing the name of the target; for the 8085 this could contain "i8085"
nRegisters	A constant containing the number of target CPU registers; for the 8085 this value would be ten (A, B, C, D, E, H, L, PC, SP, Flags).
StringToLDP_Address	A procedure which converts a string representing a memory address, I/O port, or register name into an LDP Address record.
LDP_AddressToString	A procedure which converts an LDP Address data structure into a corresponding string.
LDP_ExceptionToString	A procedure which converts an LDP Exception message into a corresponding string.
nBits	A procedure which returns the number of bits a target uses to encode one of the debugger's five generic data types.
Disassemble	A procedure which returns the mnemonic form of an instruction given a sequence of machine code bytes.
Deserialize	A procedure which deserializes bytes from an LDP communications frame to form a higher level debugger data type.
Download	A procedure to download the target using a machine dependent file format. For the 8085, this may be the Intel Hex format.
PrintItem	A procedure which prints out a datum in one of the debugger's generic formats.
Serialize	A procedure to serialize a debugger generic data type into an LDP communications frame.

Table H.1. Summary of Personality Procedures and Data

Appendix I. The debugger's directory structure

This appendix summarizes the directory structure of the debugger's development account. The debugger root directory is [DSB8674] on device USER3:.

Directory USER3:[DSB8674] -- The root of the debugger's development directory.

DEBUGGER.DIR;1	EDTINI.EDT;2	LOGIN.COM;3	OPTIONS_FILE.OPT;1
PALS.DIR;1	SDK86.DIR;1		

EDTINI.EDT	Initialization file for the VMS editor. Sets up the editor for C program development.
LOGIN.COM	Performs the logical assignments necessary to use the VMS 8086 development tools and to build the host and target debugger binary files.
OPTIONS_FILE.OPT	Used by the VMS linker to specify the linkage of the VMS Runtime Library with C image files.

Directory USER3:[DSB8674.DEBUGGER.COMMON] -- This directory contains header files which are used by both the host and target debugger C software.

I8086.H;7	LDP.H;2

Directory USER3:[DSB8674.DEBUGGER.HOST] -- This directory contains the sources used to build the debugger host software. The file DB.COM is a script which builds the debugger host.

BITCRC.C;1	BUFFER.C;34	BUFFER.H;14	BUFFER.OBJ;1
CRASH.C;2	CRASH.OBJ;1	CRC.C;97	CRC.COM;2
CRC.EXE;77	DATALINK.C;139	DATALINK.H;19	DATALINK.OBJ;1
DB.COM;31	DB.EXE;64	DB.H;44	DB.INI;24
DBA.C;21	DBA.OBJ;3	DBB.C;48	DBB.OBJ;1
DBC.C;49	DBC.OBJ;1	DBD.C;120	DBD.OBJ;1
DBE.C;7	DBE.OBJ;1	DBF.C;27	DBF.OBJ;1
DBG.C;22	DBG.OBJ;1	DBH.C;22	DBH.OBJ;1
DBI.C;16	DBI.OBJ;3	DBT.MAR;16	DBT.OBJ;1
DBT.OLD;1	ECHO.C;20	EDTINI.EDT;2	I8086.C;73
I8086.OBJ;1	I8086A.C;122	I8086A.OBJ;1	I8086B.C;37
I8086B.OBJ;11	IOSB.H;2	LDPP.C;2	LDPP.OBJ;1
LDPS.C;4	LDPS.OBJ;1	LDPSTUBS.C;2	MAKE.COM;13
OMF.H;3	OPTIONS_FILE.OPT;1	PROCTEST.C;10	PROCTEST.COM;4
QUEUE.C;5	QUEUE.H;4	QUEUE.OBJ;1	SYMTAB.C;64
SYMTAB.H;10	SYMTAB.OBJ;2	TERMINAL.C;10	TERMINAL.OBJ;1
TLTEST.C;33	TLTEST.COM;4	TLTEST.EXE;35	TRANSPORT.C;4
TRANSPORT.H;16	TRANSPORT.OBJ;1		

Directory USER3:[DSB8674.PALS] -- This directory contains the PALASM equations for the chip select PAL's on the debugger expansion hardware board.

IO.DAT;1 MEM.DAT;13

IO.DAT	PALASM Equations for I/O chip select PAL
MEM.DAT	PALASM Equations for RAM/ROM chip select PAL

Directory USER3:[DSB8674.SDK86.DEBUGGER] -- This directory contains the sources for the SDK-86 debugger target software. The file LDPSERVER.COM builds the SDK-86 target debugger software.

BUFFER.C86;15	BUFFER.H;27	BUFFER.OBJ;1	CRASH.H;22
CRC.A86;6	CRC.OBJ;2	DATALINK.C86;2	DATALINK.H;7
DATALINK.OBJ;1	DLTEST.C86;10	EDTINI.EDT;2	INT.A86;58
INT.H;5	INT.OBJ;1	LDPA.C86;3	LDPA.OBJ;2
LDPB.C86;5	LDPB.OBJ;2	LDPC.C86;2	LDPC.OBJ;1
LDPD.C86;3	LDPD.OBJ;1	LDPE.C86;3	LDPE.OBJ;1
LDPF.C86;2	LDPF.OBJ;1	LDPG.C86;2	LDPG.OBJ;1
LDPM.C86;2	LDPM.OBJ;1	LDPS.C86;2	LDPS.OBJ;1
LDPSERVER.COM;2	LDPSERVER.HEX;1	LDPSERVER.MP2;1	LDPSTATE.H;2
POINTER.H;2	TEST.COM;2	TIMEOUT.C86;14	TIMEOUT.OBJ;1
TLTEST.C86;26	TLTEST.COM;7	TLTEST.HEX;2	TLTEST.MP2;1
TLTEST.OBJ;1	TRANSPORT.C86;3	TRANSPORT.H;13	TRANSPORT.OBJ;1

Directory USER3:[DSB8674.SDK86.DEMO] -- This directory contains the demo and checkout programs for the SDK-86.

DICE.A86;7	DICE.DAT;8	DIVO.C86;2	DIVO.DAT;1
DIVO.MP2;1	EDTINI.EDT;2	LEDCLOCK.C86;8	LEDCLOCK.COM;13
LEDCLOCK.DAT;10	LEDCLOCK.DOC;1	LEDCLOCK.JOU;1	LEDCLOCK.LNK;9
LEDCLOCK.MP1;9	LEDCLOCK.MP2;8	LEDCLOCK.O86;1	LEDCLOCK.OBJ;11
MEMTEST.C86;31	MEMTEST.O86;1	PAT.A86;7	PAT.DAT;8
SIOANAL.C86;18	SIOANAL.COM;5	SIOANAL.DAT;4	SIOANAL.LNK;1
SIOANAL.MP1;2	SIOANAL.MP2;1	SIOANAL.OBJ;2	STARTUP.A86;42
STARTUP.LST;1	STARTUP.OBJ;2	UTEST.C86;29	

Directory USER3:[DSB8674.SDK86.LIB] -- This directory contains the sources for the SDK-86 Runtime Library.

COMMON.C86;6	COMMON.OBJ;2	CRASH.A86;7	CRASH.OBJ;1
EDTINI.EDT;2	INTTEST.C86;15	LDP.LIB;1	LDPLIB.COM;1
LED.C86;13	LED.OBJ;1	LIBTEST.C86;21	LIBTEST.COM;2
LP.A86;8	LP.OBJ;1	MAKE86.COM;18	PFTEST.C86;13
PORTID.A86;7	PORTIO.OBJ;1	SDK86.LIB;15	SDK86LIB.COM;13
SIO.C86;26	SIO.OBJ;2	SIOA.A86;40	STARTUP.A86;43
STARTUP.OBJ;4	TIMER.C86;5	TIMER.H;4	TIMER.OBJ;2

LDPLIB.COM	Recompiles the Runtime Library sources
LDP.LIB	The LDP Server Runtime Library