The Alpha Operating System: Program Maintenance Manual

Samuel E. Shipman
J. Duane Northcutt
Donald C. Lindsay

TR #88071

Archons Project Technical Report

July 15, 1988

The Alpha Operating System: Program Maintenance Manual

Samuel E. Shipman
J. Duane Northcutt
Donald C. Lindsay

July 15, 1988

Archons Project Technical Report #88071

This research was supported in part by the USAF Rome Air Development Center under contract number F30602-85-C-0274. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of RADC or the U.S. Government.

Table of Contents

Abstract

This report provides information necessary to understand, maintain, and extend the Release 1 version of the Alpha operating system's kernel. It is intended to be a "road-map" to guide a maintainer through the body of code that is Alpha. It collects in one place the details of the system's construction. This report and the actual source code provide all of the documentation needed to do low-level work.

This report also explains the programming conventions that were followed in producing the system. It documents conventions pertaining to directory structures, file contents, file formats, coding style, and module structure.

The implementation structure of the kernel proper and its major subsystems is described in detail. Each of the routines in each individual file is described, in terms of its interface and functionality.

The programming tools used to support the coding conventions are also described.

1 Introduction

This report provides information necessary to understand, maintain, and extend the Release 1 version of the Alpha operating system's kernel. It is intended to be a "road-map" to guide a maintainer through the body of code that is Alpha. It collects in one place the details of the system's construction. This report and the actual source code provide all of the documentation needed to do low-level work.

The structure of this report parallels the logical and physical structure of the system implementation (i.e., it is organized by kernel subsystems). The structure defined in this report is the implementation structure and does not necessarily reflect the logical structure of the system. Also, the entire report is concerned exclusively with the implementation of Release 1. All logical specifications appear in other documents (see [Northcutt 88a] and [Northcutt 88b]). No inferences concerning the conceptual aspects of Alpha should be drawn from this document.

Chapter 2 explains the programming conventions that were followed in producing the system. It documents conventions pertaining to directory structures, file contents, file formats, coding style, and module structure. Adhering to conventions made it easier for several programmers to work cooperatively on the same software. It also improved the readability and uniformity of the resulting code.

Chapter 3 describes in detail the implementation structure of the kernel proper and its major subsystems. Each of the routines in each individual file is described, in terms of its interface and functionality.

The programming tools used to support the coding conventions are described in an appendix. These consist largely of automated support for editing and formatting code that adheres to the coding conventions documented in Chapter 2.

2 Programming Conventions

This chapter documents the programming conventions used in building Alpha. It addresses the organization of files (Section 2.1), their contents (Section 2.2), organization, formatting, and coding conventions (Section 2.3), and the organization of program modules (Section 2.4).

2.1 Canonical Program Structure

The Release 1 implementation is divided into collections. Within each of these collections are files, organized by file type. At the top level, files are organized according to which major kernel subsystem they pertain (specifically, the kernel proper, the scheduling subsystem, the communications subsystem, or the storage management subsystem).

Each of these major subsystems is known as a *module*, and all of the modules that make up the operating system consist of a number of files that are organized according to a common structure. This structure is supported by the tools which make up the programming environment of Release 1 Alpha [Northcutt 88c]. These tools (such as md and acp) expect and maintain this structure.

An Alpha standard programming module consists of: assembly code files (**asm**), source header files (**include**), C source code files (**src**), and executable files (**bin**).

Most source files are written in the C language, and placed in the **src** (for source code) or **include** (for included headers) directories. A small number of files are written in the assembly language of the Motorola MC68010, and these are placed in the **asm** directories.

For improved manageability, directories that contain source files should not be polluted with temporary files and executable files. For this reason, such files are placed in a separate directory, usually named **bin** (for binary files). The conventions and tools are designed to allow source-file directories and executable-file directories to be decoupled, rather than requiring them to be in a fixed (e.g., adjacent) location.

2.2 File Contents

This section defines the conventions concerning the contents of each type of file: assembly code, header, and C source code.

2.2.1 Assembly Code

In the Alpha programming environment, there are two types of assembly files: plain assembly files, and extended-form assembly files. The format of the plain assembly files is defined by the UNIX assembler utility (as), which was specifically designed to discourage assembly language programming. The plain assembly files have names ending with '.s', whereas the extended form files have names ending with '.a'. No plain assembly language files are used in the kernel of Alpha; only the extended form is used. The extended form files are processed by the Alpha tools before they can be assembled. The *makefile* contains the necessary commands to convert extended form assembly files to plain assembly files. The processing consists largely of passing the extended form file through a preprocessor (the C language's preprocessor, cpp). The major benefit of this step is the ability to allow the inclusion of "header" files, to bind various parameter values defined by such header files, to allow the use of macros, and to allow the use of compile-time constant expressions (using arithmetic and logical operations not provided by as). Source files are never included in this manner: this would be a violation of Alpha standards.

2.2.2 Source Header Code (Include Files)

Header files contain non-executable data-generating definitions that may be shared among multiple source files. There are two major types of statements that appear in header files: ones that define constants and macros, and ones that define data types. These are typically segregated into separate files. Constant and macro definitions are contained in **cons** files and type definitions are contained in header files.

cons files are included only by an associated header file (but this principle is not strictly adhered to in that some source files include **cons** files directly). The filename of a **cons** file begins with a word or acronym in all capitals that indicates the function of the file, followed by 'cons.h' (e.g., VMcons.h, SUNcons.h, etc.).

By including the necessary header files from within other header files, an attempt is made to eliminate concerns about the ordering of **#include** statements within the source files. A common C-language practice is followed, whereby the body of an include file is made conditional on the non-definition of a symbol, which that body defines. In this manner, an include file may legally be included many times into a single source file, without causing multiple instantiations of its body. This results in a property of idempotence that allows file inclusion ordering problems to be largely treated as compile-time efficiency concerns, rather than as errors which must be prevented, or detected and dealt with.

2.2.3 C Language Source Code

The source files used in Alpha are cross-compiled using the standard C compiler of the Sun-3 workstation (using the "-68010" switch). Also, source files should pass most lint checks (lint is a UNIX tool that examines C programs for error-prone usage). Machine dependent features, such as **asm** statements, are avoided as much as possible in human-generated code. They are sometimes generated by macros defined in include files. Also, the C optimizer sometimes generates illegal instructions for the MC68010, and **asm** statements are sometimes used to work around that problem.

2.3 File Formats

Each type of file has a common format, regardless of the major module that it is a part of, i.e., the format of file types does not change from one module to another. Also, file change logs are all maintained in the same format, regardless of the type of the file. This section defines the file format for each type of file discussed in the previous section.

2.3.1 Assembly Code

This describes the format of extended-form assembly language files ('.a' files). The first item in the file is a boxed header comment containing a title describing the general function of file.

Following that is a change-log (also called a HISTORY comment, because the HISTORY keyword is used to indicate the beginning of the change-log so that an automated tool can generate timestamp information automatically). Each entry in the change-log gives the date and the user who made the entry, followed by one or more lines of commentary on the change being documented. Many of these history comments indicate a new release, and are added regardless of whether there were any changes between releases. The end of each change-log contains a copyright notice for the file.

The next item is comment stating "Include Files" followed by a list of **#include** statements, which define all the include files needed by this file.

The next item is a comment stating “Import/Export Business” followed by **.globl** directives for global labels, broken into two categories based on whether they are logically imported to the file or defined and exported in the file. Any other required assembler directives are given after the imports and exports.

Following this are the definitions of any global data defined in the file (other than global labels for routine names). This is indicated by a comment stating “Data Declarations.”

After this preamble comes a comment stating “Routines.” This is followed by the routines defined in the file. Each routine has a one-line banner comment that states its function succinctly. Following this is the label definition for its name. Next is a list of symbols defining offsets used in accessing arguments from the stack, if any. Following that is a header comment that explains in more detail what the routine does. If appropriate, this includes a definition of the interface to the routine from the C-language. Individual statements in the routine may have an end-of-line comment, delimited at the beginning by a semicolon and at the end by the end-of-line.

This format is illustrated in the example assembly language source file layout shown in Figure 1.

```
/************************************************************************
*
*                    Example Assembly Language File Layout
*
\************************************************************************/

/*
* HISTORY
* 17-Sep-88  Alpha Kernel (alpha) at Carnegie Mellon University
*       Release 0.6
*/

/*
* Include Files
*/

#include    "../include/example1.h"
#include    "../include/example2.h"

/*
* Imports/Exports
*/

        /* imports */
        .globl    _Import
        /* exports */
        .globl    _Export

/*
* Routines
*/

        .even
        .text

/* This is an example assembly language routine */

_Example:
        offset = 0x4
/*
* This example routine takes an argument with stack offset of 4.
* If this routine did anything, this comment would explain what it did.
* C Interface:
*       void    Example(offset)
*               int offset;
*/
        moveml    sp@(offset), d0              ; get parameter (a statement)
        rts                                    ; return to caller
```

Figure 1: Assembly Language File Layout

2.3.2 Source Header Code

Header files begin with a boxed header comment and a change log, as described for '.a' files. Following that are the statements:

```
#ifndef <filename>_HDR
#define <filename>_HDR
```

and at the end of the file is the statement:

```
#endif <filename>_HDR
```

These statements ensure that a header file is included only once for a given source file, as described in Section 2.2.2. Note that "<filename>" signifies the basename of the include file, rendered in all capitals. For example, the macro used for the file VMcons.h would be VMCONS_HDR.

Next there is comment stating "Include Files" followed by **#include** directives, if any are needed.

At this point, if the file is a **cons** file, it will contain only macro definitions for constants. By convention, the names of macros that define constants are in all capitals, using underscores if necessary to indicate word divisions. A macro or group of macros typically has a comment explaining its meaning.

If the file is not a **cons** file, then at this point it can contain type definitions and macro definitions. We discuss type definitions first.

All types used in Alpha (except built-in types in C) are defined using a **typedef** statement. A type name usually is rendered in all lower case letters, except sometimes when acronyms are used as part of the type name, in which case the acronym is often given in upper case. If the type being named is a pointer type, the typename ends in "ptr". If the type is declared as a **struct**, it ends in "str" (but this name is not used in declaring variables of the type; rather, another name is **typedef**ed, which indicates the structure). For example:

```
typedef struct abheadstr {
    abid        id;           /* control block's ABID */
    abheadptr   bktchain;     /* bucket chain in dictionary */
} abhead;
```

This declares the type abhead, which is a C structure; the **struct** is named abheadstr. Usually, pointers to such types are declared before the types are declared, as follows:

```
typedef struct abheadstr       *abheadptr;       /* pointer to abhead */
```

This takes advantage of the rules of C that allow a pointer to a **struct** to be declared before the **struct** is declared, but do not allow a pointer to a **typedef** name to be declared before that **typedef** name is defined. This allows other **struct** declarations to use the pointer types before the base types are declared. Note that no variable is declared to be of type **struct** abheadstr; rather, the type abhead is used.

Note that when data structures are defined, Alpha's data layout convention is followed. This dictates that each name is introduced on a separate line, so that there will be a logical place (to its right) for any comments. Name lists are not used, in the sense that

```
int a,b,c;
```

is a list of length three. This is intended to reduce rework when a line or lines is moved around or copied elsewhere.

Enumeration declarations are handled similarly, i.e., each element of the enumeration is defined on a separate line.

After the type definitions come macro definitions, if any. Typically, all macros that define constants are defined in **cons** files, so the macros in a non-**cons** file usually define fragments of source code.

If a macro defines a construct that evaluates to some value, and that value is not a compile-time constant, then the macro is defined and referenced as a function, even if the macro has no parameters, e.g.:

```
#define     RUN_TIME_VALUE()      (variable / 2)
```

The layout of a **cons** header file is illustrated in Figure 2. The layout of a non-**cons** header file is illustrated in Figure 3.

```
/*****************************************************************************
*
*                         Example Constants Header File Layout
*
\*****************************************************************************/

/*
* HISTORY
* 17-Sep-88  Alpha Kernel (alpha) at Carnegie Mellon University
*          Release 0.6
*/

#ifndef EXCONS_HDR
#define EXCONS_HDR

/*
* Include Files
*/

#include       "../include/example1.h"
#include       "../include/example2.h"

/*
* Example Constants
*/

/* First example constant */
#define        EXAMPLE              32

/* Second example constant */
#define        EXAMPLE_TWO      64

#endif EXCONS_HDR
```

Figure 2: Layout of Header File for Constants

```
/************************************************************************
*
*                         Example Header File Layout
*
\************************************************************************/

/*
* HISTORY
* 17-Sep-88  Alpha Kernel (alpha) at Carnegie Mellon University
*          Release 0.6
*/

#ifndef EXAMPLE_HDR
#define EXAMPLE_HDR

/*
* Include Files
*/

#include      "../include/example1.h"
#include      "../include/example2.h"

/*
* Structure Predeclarations
*/

typedef struct firstexamplestr                  *firstexampleptr;
typedef struct secondexamplestr                 *secondexampleptr;

/*
* Structure Type Definitions
*/

typedef struct firstexamplestr {                        /* this is a first example */
     int                         count;           /* 1, 2, 3, 4, ... */
     secondexampleptr            septr;           /* pointer to a second example */
} firstexample;

typedef struct secondexamplestr {                       /* this is a second example */
     int                         count;           /* 1, 2, 3, 4, ... */
     firstexampleptr             feptr;           /* pointer to the first example */
} secondexample;

/*
* Macros
*/

#define       EXAMPLE_MACRO(arg)              ((arg) / 2)

#endif EXAMPLE_HDR
```

Figure 3: Layout of Header File

2.3.3 C Language Source Code

C language source files begin with a boxed header comment, a change log, and include statements, as described in previous sections.

Following that is a comment stating “Procedure Predeclarations”, and then a list of declarations of all the procedures (C functions) logically exported procedures (i.e., global or non-static functions). This provides a convenient listing of the procedural interface to the source file. Note that procedure names begin with an upper case letter, e.g., Procedure(). Also, if the procedure name consists of multiple words, each “word” is capitalized as well, e.g., ProcedureReturningValue(). (Like most of the other naming conventions, this is not an absolute rule, but a guideline).

After the procedure predeclarations is a comment stating “Data Declarations”, followed by the data declarations for the source file. These consist exclusively of the defining declarations of global and static variables. No **extern** declarations are allowed here. Note that variable names typically begin with a lower case letter. Notable exceptions to this rule are variables that begin with acronyms; the acronym is sometimes capitalized. As with procedure names, multiple words after the first in a variable name are sometimes capitalized.

After the data are declared, there is a comment stating “Routines”, followed by the code for the routines in the file. Figure 4 illustrates the layout of C language source files up to this point.

When this point in the file has been reached, the remainder of the file contains zero or more routine definitions.

Each routine is preceded by a one-line "banner" comment, which summarizes it. The routine’s "signature" follows this—that is, the routine’s returned type, its name, and the names and types of its formal arguments. After the signature is a block comment, giving the specifications of the routine. These typically contain the information needed for a user of the routine to understand its interface and function. This comment follows the routine signature so that the specifications become integral to the routine and stay with it if subsequent editing moves it around.

Following the block comment is the opening brace of the function, followed by declarations of local variables and external variables. It is standard practice in Alpha code to declare all variables and functions that are used in the routine, both local and external, within the routine. This provides better documentation of the variables and routines referenced by each routine.

Code is generally formatted as shown in the example. This adheres to the principle of having one relatively significant meaningful unit per line, and indenting in multiples of four spaces in structured constructs. Note also that for any C statement that allows a statement as part of itself (e.g., a **while** statement), that statement must be a compound statement. This is for uniformity, readability, and ease of adding code. Often a semicolon is added following compound statements (where permitted by the language) for uniformity.

```
/*****************************************************************************
*
*                    Example C Source File Layout (preamble)
*
\*****************************************************************************/

/*
* HISTORY
* 17-Sep-88  Alpha Kernel (alpha) at Carnegie Mellon University
*       Release 0.6
*/

/*
* Include Files
*/

#include    "../include/example1.h"
#include    "../include/example2.h"

/*
* Procedure Predeclarations
*/

void            Example();
boolean         IsExample();
int             NumExample();

/*
* Data Declarations
*/

/* this variable illustrates a variable declaration */
firstexample        example1;           /* a variable of type firstexample */

/* the array declared below has the number of elements defined by the constant SIZE */
secondexample       example2[SIZE];     /* an array of elements of type firstexample */

int                 numExamples;        /* a global integer variable */

static int          privateNumber;      /* an integer variable private to this file */


/*
* Routines
*/
```

Figure 4: Layout of C Language Source File (Preamble)

Alpha code does not take undue advantage of the C semantics that state that a non-zero value is considered to signify TRUE, and zero signifies FALSE. Alpha code uses an enumerated type for boolean variables and direct comparisons with the enumeration values are performed. An expression is explicitly compared to zero (or NULL, a constant defined as 0, if a pointer value) to determine whether its value is equal to zero.

Figure 5 shows the format of a routine definition.

```
/*
* Routines
*/

/* This example routine illustrates some of the Alpha coding conventions */

boolean Example(arg);
    int         arg;
/*
* This routine takes an integer argument and returns TRUE if the argument is even,
* FALSE if it is odd. It contains other extraneous code that illustrates some conventions.
*/
{
    int             local;              /* a local variable */
    boolean         awake = TRUE;       /* a local boolean */
    extern int      goldenRatio;        /* an external global variable */

    /* the if statement below does nothing useful */
    if (awake == TRUE) {
        awake = FALSE;
        for (local = 0; local < 10; local++) {
            /* note that the indentation here is a constant four spaces per level;
             * the proportional spacing of this font may obscure that fact. */
            local = local * 2;
        };
    };
    if (arg == (arg/2)*2) {             /* check whether argument is even */
        return(TRUE);
    }
    else {
        return(FALSE);
    };
}
```

Figure 5: Layout of C Language Source File (Routine Definition)

2.4 Building Modules

It is an Alpha convention to use the md program and its associated standard *makefiles* to build any software consisting of more than a few source files. This requires that a module specification for md be created and kept up to date for each module in the system. md uses this specification to derive dependencies (based on **#include** directives in the source and header files) and generate *makefiles* that express these dependencies and define make variables representing the files required to build the module. This specification consists primarily of a list of source files for the module. An example module specification (for the communication subsystem) is shown in Figure 6.

```
module comm:              ; communication subsystem
    in comm/asm:
        crt0.a            ; C startup code (@stand)
        excep.a           ; assembly language utilities (@kernel)

    in comm/src:
        cvm.c             ; main state machine routines
        cvm_buffer.c      ; packet buffer management routines
        cvm_debug.c       ; communications virtual machine debugger
        cvm_event.c       ; timed event routines
        cvm_itimer.c      ; interval timer routines
        cvm_memory.c      ; memory management routines
        cvm_packet.c      ; packet manipulation routines
        cvm_set.c         ; set manipulation routines
        cvm_start.c       ; communications virtual machine startup routines
        cvm_token.c       ; state machine token manipulation routines
        cvm_uart.c        ; UART handling routines
        ext_proto.c       ; External (UNIX) interface protocol routines
        ipi.c             ; interprocessor interface (@kernel)
        man_proto.c       ; Monitor Active Nodes protocol routines
        printf.c          ; 'printf()' routine (@kernel)
        pt_proto.c        ; Page Transfer protocol routines
        rbi_APintf.c      ; RBI coprocessor interface routines
        rbi_proto.c       ; Reliable Basic Invoke protocol routines
        rm_proto.c        ; Reliable Message Protocol routines
        se_debug.c        ; sun ethernet controller debugging routines
        se_filter.c       ; ethernet multicast packet filter routines
        se_intf.c         ; ethernet interface routines
        se_intrnl.c       ; ethernet internal routines
        stimer.c          ; event logging timer routines (@kernel)
        tmar_proto.c      ; Thread Maintenance and Repair protocol routines
```

Figure 6: Example Module Specification

Using md with the appropriate makefiles allows a user to copy a single makefile to an empty directory and type "make" to build the entire module. Use of md is documented in [Northcutt 88c].

3 Kernel Files

The kernel is formed from several collections of files, grouped by module, i.e., kernel proper, scheduling subsystem, communications subsystem, and storage management subsystem. The files that make up each of these collections are further grouped according to file type, i.e., assembler, include, and source files.

3.1 Kernel Proper

The kernel-module files make up the kernel proper, i.e., the part of the kernel that coexists with the applications code on the applications processor.

3.1.1 Assembler Files

These files implement the low-level machine-specific actions.

An attempt was made to minimize the amount of assembly code that is in the kernel. Only those functions that have to be done in assembly code are. These routines provide low-level interfaces to the hardware (e.g., UART, MMU, and interrupt priority) and encapsulate the standard C code that is invoked by the hardware's exception handling mechanisms (e.g., traps, interrupts, kernel invocations, etc.). The routines that are provided to the client use the standard C calling interface. Also included are routines that must be efficient (e.g., memory block copy/fill, and an event logging facility).

All interface routines use a common preamble.

3.1.1.1 Bus Error Trap Handler — berr.a

This routine invokes the C routine that handles hardware bus errors that are generated as a result of memory access protection failures, memory parity failures, and non-existent memory accesses.

Routine:

```
void    BusErr(stkdump)
        berr    stkdump;
```

This routine decides if this trap is to be handled by my code, if so, call 'VMFault()', else go to the monitor's trap handler. Pass the faulting address and the special status word to 'VMFault()' (both of them are taken from the stack barf on the SS). When the call to 'VMFault()' returns, restore everything to it's previous state, and continue with the faulting instruction. This handler uses the standard exception handling preamble.

3.1.1.2 Assembly Constants — cons.a

This file contains a collection of constants to be used by assembly language code.

These constants are defined in this file in order to cope with the manner in which the C programming language and the assembler interact. All of these constants are (or could be) defined in terms of other constants or macros from the include files in "../include/." Those include file definitions are used by both C and assembly language programs. However, a second definition is required when a symbol is to be used in an 'asm' statement in a C program (e.g., "*asm(*" movw #DUMMY_VOR,sp@-"*);*"). Since the argument to the "asm" statement is a single string, the C compiler doesn't expand macros within the string. This file supplies the definitions of the constants that are used in such statements.

Constants:

DUMMY_VOR
SUPER_MSK
INTR_STK
MON_STK
BC_REG
FC_REG
BMOST_REG
FMOST_REG
BMANY_REG
FMANY_REG
IN_PBP
OUT_PBP
SUPER_MODE
USER_MODE
MIN_PRIORITY
MAX_PRIORITY
USER_MIN_PRI
LO
L1
L2
L3
L4
L5
L6
L7
CLIENTCALL
EXIT_TRP
CO_RET_TRP
RET_MON_TRP
KERN_TRP
TRUE_VALUE
FALSE_VALUE

3.1.1.3 Spurious Interrupt Handler Stub — intr.a

Routine:

```
void    Interrupt()
```

Routine for handling unexpected interrupts. Obviously, there are no explicit arguments or returned value.

This code gets the vector number from the stack, calls the C routine 'IntrHdlr()' (after saving the C scratch registers), and passes the vector number as a parameter to 'IntrHdlr()'. When 'IntrHdlr()' completes this routine returns from the exception. This disables other interrupts on entry and reenables them on return.

3.1.1.4 Interprocessor Interrupt Handler Stub — ipiintr.a

Routine:

```
void    IPIintr()
```

Routine for handling interrupts from the "IP" (InterProcessor) hardware. (These are interrupts concerning messages between the processors within a node.) Control comes here on interrupt level 4, in supervisor state and in whatever virtual-memory context it was in at the time.

Saves all relevant "C" scratch registers and calls the routine 'IPIrequest()' to handle the request.

3.1.1.5 Sun-2 MMU Access Routines — mmu.a

This routine includes routines that are used to access the Sun-2s MMU tables. This includes routines for reading and writing the processor's context register, segment map cache, and page map cache.

Routine:

```
void    SetUserCtx(ctxnum)
        ctx     ctxnum;
```

This routine writes the low-order byte of 'ctxnum' into the MMU's User Context Register. It does this by first selecting the MMU Space function code and then using this code to access the user portion of the context register, leaving the Supervisor Context Register unchanged.

Routine:

```
void    SetSuperCtx(ctxnum)
        ctx     ctxnum;
```

This routine writes the low-order byte of 'ctxnum' into the MMU's Supervisor Context Register. It does this by first selecting the MMU Space function code and then using this code to access the supervisor portion of the context register, leaving the User Context Register unchanged.

Routine:

```
void    SetCtxs(suctx, usctx)
        ctx     suctx;
        ctx     usctx;
```

This routine writes the low-order byte of 'suctx' into the MMU's Supervisor Context Register and the low-order byte of 'usctx' into the MMU's User Context Register. This is done by first selecting the MMU Space function code and then using this code to access both the supervisor and the user portions of the context register.

Routine:

```
void    GetUserCtx(ctxnum)
        ctx     ctxnum;
```

This routine returns the low-order byte of the MMU's User Context Register. This is done by first selecting the MMU Space function code and then using this code to access the user portion of the context register.

Routine:

```
void    GetSuperCtx(ctxnum)
        ctx     ctxnum;
```

This routine returns the low-order byte of the MMU's Supervisor Context Register. This is done by first selecting the MMU Space function code and then using this code to access the supervisor portion of the context register.

Routine:

```
void    GetCtxs(suctx, usctx)
        ctx     suctx;
        ctx     usctx;
```

This routine returns the low-order byte of the MMU's Supervisor Context Register in the location pointed at by 'suctx' and the low-order byte of the MMU's User Context Register in the location pointed at by 'usctx.' This is done by first selecting the MMU Space function code and then using this code to access both the supervisor and the user portions of the context register.

Routine:

```
void    LoadSMC(ctxnum, seg, segcnt, psrc)
        ctx             ctxnum;
        segindx         seg;
        int             segcnt;
        segdescrip      *psrc;
```

Takes the context in which this is to be done ('ctxnum'), an index into the SMC that indicates which SD to start from ('seg'), a count of how many SD's are to be loaded ('segcnt'), and pointer to the array of SD's to be loaded ('psrc'). Loads the array of SD's into the SMC, at the specified location. This routine is "safe" in the sense that it works even in the face of NMI's (unlike the monitor's SET_SEG() routine).

Routine:

```
void    LoadPMC(ctxnum, seg, psrc)
        ctx             ctxnum;
        segindx         seg;
        segdescrip      *psrc;
```

Takes the context in which this is to be done ('ctxnum'), an index into the PMC that indicates which segment is to be loaded ('seg'), and pointer to where in the kernel's memory the PD's to be loaded can be found ('psrc').

Then, loads an entire page map into the PMC from the array of PD's in the kernel's memory.

This routine is "safe" in the sense that it works even in the face of NMI's.

Routine:

```
void    UnloadPMC(ctxnum, seg, pdest)
        ctx          ctxnum;
        segindx      seg;
        segdescrip   *pdest;
```

Takes the context in which this is to be done ('ctxnum'), an index into the PMC that indicates which segment is to be loaded ('seg'), and pointer to where in the kernel's memory the PD's to be unloaded should be put ('pdest').

Then, unloads an entire page map of PD's from the PMC into the kernel's memory.

This routine is "safe" in the sense that it works even in the face of NMI's.

3.1.1.6 Utility Routines for Objects — objutil.a

Routine:

```
void    ClientCall()
```

An assembly language routine for entering a client object from the kernel and handling the return to the kernel when the operation is done.

This is entered by an 'rte' from 'InvokeCO' and the top of the stack holds a pointer to the entry point in the object being called.

This routine sets up the parameters on the client's stack for a making a "C" call to the called object. When the called object returns, the client's stack is cleaned up, and this code returns to 'InvokeCO'.

This modifies 'a0' but it is called by (and calls) a "C" routine and so it shouldn't matter.

Routine:

```
void    AbortBlock()
```

This routine traps to the "AbortTrapIn" handler.

Routine:

```
void    LogEventIntf(type, parm)
        long    type;
        long    parm;
```

A generic kernel event is placed in the log. This is done by trapping to the "LogEventIn" handler.

Routine:

```
void    UserEventIntf(type, parm)
        long    type;
        long    parm;
```

This routine traps to the "LogEventIn" handler.

Routine:

void TimingEventIntf(type, parm)
long type;
long parm;

This routine traps to the "LogEventIn" handler.

3.1.1.7 Trap Handlers — trap.a

Routine:

void ClientTrapIn()

Trap entry vector for entering the kernel on an invocation from a client. Just calls the C ’ClientInvoke()’ routine, and the returns to the object that made the invocation. Uses the standard exception handler preamble.

Routine:

void KernelTrapIn()

This is a handler for the trap entry vector for entering the kernel on an invocation from a kernel object. This handler is always entered from a kernel object as a result of an invocation, and the parameter that is to be passed to ’KernelInvoke()’ (i.e., a pointer to the outgoing parameter block) is always in register A4. This parameter is passed by pushing it onto the kernel stack. It calls the C ’KernelInvoke()’ routine, and the returns to the object that made the invocation.

This routine uses the standard exception handler preamble.

Routine:

void ExecTrapIn()

This is for entering the kernel from ’CThreadStub’, which short circuits the normal methods for entering the kernel (through ’KernelInvoke()’ or ’ClientInvoke()’), but still uses the standard conventions for entering the kernel.

It calls the C ’ExecInvoke()’ routine, and the returns to the object that made the invocation.

This routine assumes that the trapper was in supervisor mode before the trap was executed. (This is not too unreasonable since only objects that are special in some way should ever use this trap, and such objects should never be client objects (read "user mode").)

This routine uses the standard exception handler preamble.

Routine:

void LogEventIn()

Trap entry vector for entering the kernel to log an event (it doesn’t matter if you are already there, we do it this way for uniformity). Have to determine which mode the trap was called from, put the parameters onto the supervisor stack, call the routine, clean up the stack, and then return from the trap.

We are now letting the compiler push the parms onto the stack, and so there is also a ’jsr’ frame on the calling stack following the parameters.

Interrupts are disabled at the start and reenabled on ’rte’.

This doesn’t use the standard exception preamble.

Routine:

```
void    AbortTrapIn(u_type, s_type)
        opindex     u_type;
        opindex     u_type;
```

This trap handler is entered whenever an abort block operation is to be executed from an object. The invocation of this routine is always accompanied by a parameter that indicates the specific type of operation that is to be performed. This operation type parameter is pushed on the stack by the invoking routine (i.e., the 'AbortTrap()' routine).

The D1 register is set with a flag to indicate if this handler was entered from a client or kernel object, and the appropriate routine called based on the value of the operation type parameter. It is assumed in this routine (as well as all of the abort handling code) that the C scratch registers can be used at will (i.e., A0, A1, D0, and D1).

This code propagates back any return value passed back by the routines it calls.

Note that this is just a demux'ing stub and that the real code is in the various operation handling routines.

The types of operations handled by this routine are:

BEGIN_GAB — begin a generic abort block

BEGIN_DAB — begin a deadline abort block

END_GAB — end a generic abort block

END_DAB — end a deadline abort block

END_ABORT — end an abort block's code section

STD_ABORT — standard (system default) "on abort" handling code

STD_DL_ABORT — standard "on deadline abort" handling code

This uses the standard exception handling preamble.

3.1.1.8 Serial I/O Interrupts — uart.a

Routine:

```
void    _UARTintr()
```

Come here on UART interrupt (Level 5 on Sun 1.5s, Level 6 on Sun 2.0s), in supervisor state and in whatever context it was in at the time.

First the "C" scratch registers are saved and then the UART is read to see if the interrupt was due to a Tx or Rx event (i.e., signalling that the transmit buffer is now empty, or that the receive buffer is now full).

The appropriate TTY handler routine is called based on the result of the Tx/Rx test.

This handler uses the standard exception handling preamble.

3.1.1.9 Utility Routines for the Kernel — util.a

Routine:

```
u_short     SetPriority(newpri)
        int     newpri
```

Masks to ensure only priority bits are set. Takes and returns a priority value aligned as it appears in the status register.

Makes a point of not lowering priority to zero before changing it.

Leaves the trace bit as it was, and leaves the processor in the same mode.

Routine:

```
u_short     GetPriority()
```

Returns the priority value aligned as it appears in the status register.

Routine:

```
void  BlkMove(src, dst, cnt)
      char *      src;
      char *      dst;
      int         cnt;
```

Moves 'cnt' bytes from 'src' to 'dst'.

There are 3 possible byte alignments that must be dealt with here — 'src' and 'dst' are even, 'src' and 'dst' are odd, and 'src' and 'dst' are odd/even or even/odd. And in any event, the 'cnt' can be either odd or even. All of these possibilities must be taken into account in order to make this an efficient, and still generally useful, routine. If both the 'src' and 'dst' are even, a long word, loop mode instruction sequence can be used. If both the 'src' and 'dst' are odd, the first byte can be moved and then it looks like both of the addresses are even. If one address is odd and the other even, then a byte-by-byte, loop mode instruction sequence must be used. In the case where long word moves can be done, the 'cnt' is rounded down by zero'ing the last two bits, and any remaining bytes are moved individually.

Routine:

```
void  BlkFill(val, dst, cnt)
      short       val;
      char *      dst;
      int         cnt;
```

Moves the value into the array of short's starting at 'dst' and continues until 'cnt' moves have been done.

If the 'dst' address is odd, have to do byte moves, otherwise can do word moves.

Routine:

```
void  BlkFillByte(val, dst, cnt)
      short       val;
      char *      dst;
      int         cnt;
```

Moves the value into the array of bytes starting at 'dst' and continues until 'cnt' moves have been done.

Routine:

```
void   BlkFillLong(val, dst, cnt)
       long        val;
       char *      dst;
       int         cnt;
```

Moves the value into the array of long's starting at 'dst' and continues until 'cnt' moves have been done.

If the 'dst' address is odd, have to do byte moves, otherwise can do word moves.

Routine:

```
void   TestAndSet(semaptr)
       char *      semaptr;
```

Perform Test and Set operation on a byte size semaphore pointed by 'semaptr'. It will Return only when the semaphore is free, otherwise it will wait spinning on the semaphore.

3.1.2 Include Files

The following files are part of the source of the kernel. However, they define constants and data types which are parameters of the kernel, or which are shared between the various compilations which must be performed.

3.1.2.1 Abort Block Constants — ABcons.h

This file contains constants used by the abort block maintenance facility. This includes aborty block nesting limits and abort trap operation codes.

Constants:

```
MAX_ABLKS
MAX_ABLK_LEVEL
BEGIN_GAB
END_GAB
BEGIN_DAB
END_DAB
END_ABORT
STD_ABORT
STD_DL_ABORT
```

3.1.2.2 Alphabit Identifiers Constants — AIcons.h

This file contains constants related to Alphabit identifiers.

Constants:

```
ABID_SSO
ABID_IO
ABID_IRO
ABID_ERO
ABID_WK
ABID_WKSSO
```

ABID_NODE_MASK
ABID_TYPE_MASK
ABID_INSTANCE_MASK
ABID_COUNT_MASK
ABID_REP_NODE_MASK
ABID_NODE_SIZE
CAPA_SIZE
FIRST_KRN_WKC_ABID_CT
SSO_TYPE_SIZE
SSO_TYPE_MASK
SSO_INSTANCE_SIZE
SSO_INSTANCE_MASK

3.1.2.3 Application Processor Interface Constants — APIcons.h

This file contains constants related to the application processor interface.

Constants:

C_INV
C_INVC
C_ADD
C_REM
C_BEGIN_ABRT
C_ABRT_DONE
C_KERNEL_READY
C_SEND_PAGE
C_SEND_INFO
C_WRITE_SYNC
C_PAGE_INSTLLD
C_INFO_INSTLLD
C_DUMP_RQST
C_EXT_MSG_SEND
S_ADD
S_SADD
S_DADD
S_REM
S_KILL
S_PSH_SA
S_POP_SA
S_CHG_SA
S_DP_SI
S_UP_SI
S_SET_A_SI

S_GET_A_SI
S_PRMT_CF
S_IPRMT_CF
S_STATS_ON
S_STATS_OFF
AP_R_INV
AP_R_INVC
AP_TO_CP_ADD
AP_TO_CP_REMOVE
AP_BEGIN_ABORT
AP_ABORT_DONE
AP_KERNEL_READY
AP_SEND_PAGE
AP_SEND_INFO
AP_WRITE_SYNC
AP_PAGE_INSTLLD
AP_INFO_INSTLLD
AP_DUMP_RQST
AP_EXT_MSG_SEND
AP_ADD
AP_SADD
AP_DADD
AP_REM
AP_KILL
AP_PSH_SA
AP_POP_SA
AP_CHG_SA
AP_DP_SI
AP_UP_SI
AP_SET_A_SI
AP_GET_A_SI
AP_PRMT_CF
AP_IPRMT_CF
AP_SP_STATS_ON
AP_SP_STATS_OFF

3.1.2.4 Bus Error Decoding Constants — BEDcons.h

This file contains constants related to the decoding of bus errors.

Constants:

BE_AOB
BE_VOR

BE_SSW
BE_PC
C_REG
SSW_FUNC
SSW_INST
SSW_RW
SSW_RMW
MMMM
L7_INTR_HDLR
MON_REFR
BUSERR
ADDERR
FUNC_USER
FUNC_DATA
FUNC_PROG
FUNC_SUPER
MAKE_SUPER

3.1.2.5 Coprocessor Interface Constants — CPIcons.h

This file contains constants related to the coprocessor interface.

Constants:

C_RINV_RQ
C_RINV_C
C_TRIM_T
C_STRT_ABRT
C_END_ABRT
C_PG_SENT
C_PG_RCVD
C_INFO_SENT
C_INFO_RCVD
C_SYNC_READ
C_NEWSTATUS
C_DUMP_RPLY
C_EXT_MSG_RCVD
S_PRMT
S_SID
S_PRMT_SID
S_ABRT
S_ABRTSI_RPLY
S_DPUPSI_DONE
SYNCHRONIZE

CP_INV_RQ
CP_INV_C
CP_TRIM_T
CP_START_ABORT
CP_END_ABORT
CP_SYNC
CP_PG_SENT
CP_PG_RCVD
CP_INFO_SENT
CP_INFO_RCVD
CP_SYNC_READ
CP_NEWSTATUS
CP_DUMP_RPLY
CP_EXT_MSG_RCVD
SP_PRMT
SP_SID
SP_PRMT_SID
SP_ABRT
SP_ABRTSI_RPLY
SP_DPUPSI_DONE
SP_SYNC

3.1.2.6 Communication Virtual Machine Protocol-Specific Constants — CVMPcons.h

This file contains constants related to the Virtual Machine protocol.

Constants:

RQST_TYPE
RPLY_TYPE
INV_RQST
INV_RPLY
AP_ADD_ABID
AP_REMOVE_ABID
AP_OUTRQST
AP_OUTRPLY
AP_INRQST
AP_INRPLY
OBI_PTYPE
TT_PTYPE
RBI_PTYPE
RM_PTYPE
BB_PTYPE
TMAR_PTYPE

PT_PTYPE
MAN_PTYPE
EXT_PTYPE
OBI_PROTOCOL
TIMER_TEST_PROTOCOL
RBI_PROTOCOL
RM_PROTOCOL
BB
TMAR_PROTOCOL
PT_PROTOCOL
MAN_PROTOCOL
EXT_PROTOCOL
AP_ADD_ABID_TKN
AP_REMOVE_ABID_TKN
AP_OUTRQST_TKN
AP_OUTRPLY_TKN
AP_INRQST_TKN
AP_INRPLY_TKN
RQST_HDR_SIZE
RPLY_HDR_SIZE
PARMSIZE
CAPASIZE

3.1.2.7 CVM Standard Protocol Constants — CVMSPcons.h

This file contains constants related to the Communication Virtual Machine Standard Protocol.

Constants:

STARTUP
MY_ENET_ADDR
NULL_TKN_TYPE
MAX_PREDEF_NULL_TKN_TYPES
MAX_PREDEF_UART_TKN_TYPES
MAX_PREDEF_RCVDPKT_TKN_TYPES
MAX_PREDEF_SENTPKT_TKN_TYPES
MAX_PREDEF_TEVENT_TKN_TYPES
MAX_PREDEF_APINTF_TKN_TYPES
MAX_PREDEF_INTERNAL_TKN_TYPES

3.1.2.8 Alphabit Dictionary Constants — Dcons.h

This file contains constants related to the Alphabit Dictionary.

Constants:

DICT_SIZE

3.1.2.9 Event Constants — EVcons.h

This file contains constants related to the Event mechanism.

Constants:

MAX_EVENTS
EV_BUF_START
EV_BUF_END
EVENT_BUF

3.1.2.10 File Subsystem Constants — Fcons.h

This file contains constants related to the File subsystem.

Constants:

MAX_KERN_EXT
MAX_OTYPE_EXT
MAX_PINST_EXT
MAX_OINST_EXT
MAX_DATA_EXT
INSTALL_EXTENT
OT_CODE_EXT
OT_DATA_EXT
PI_CSTK_EXT
OI_CODE_EXT
OI_DATA_EXT
OI_HEAP_EXT
OI_MAP_EXT
K_HEAP_EXT
K_PHEAP_EXT
D_FILE_EXT
MAX_INS_EXT_PGS

3.1.2.11 Heap Constants — Hcons.h

This file contains constants related to the kernel heap.

Constants:

HEAP_SIZE
MIN_BLOCK_SIZE
NUM_BLOCK_DESCS
NUM_BLOCK_SIZES

3.1.2.12 Interprocessor Interrupt Generator Constants — IGENcons.h

This file contains constants related to the Interprocessor Interrupt Generator.

Constants:

APIPI_INTRGEN_VA
IGEN
CPIPI_INTRGEN_VA
IGEN
MODBIT_MASK
INTRGEN_MASK
IGEN_SET
IGEN_CLEAR
INTR_AP_MASK
INTR_CP_MASK
SET_INTR
CLEAR_INTR

3.1.2.13 Interprocessor Interface Constants — IPIcons.h

This file contains constants related to the Interprocessor Interface.

Constants:

CHAN_TYPE_MASK
CHAN_SYNC_MASK
CHAN_INTR_MASK
CHAN_OPNUM_MASK
SP_CHAN
CP_CHAN
DP_CHAN
SP
CP
DP
SYNC
ASYNC
IPI_L0
IPI_L1
IPI_L2
IPI_L3
IPI_L4
IPI_OP_TYPE
IPI_OP_SYNC
IPI_OP_INTR_LEVEL
IPI_OP_OPNUM
CHAN_SIZE_IN_PG
CHAN_SIZE_IN_BYTES
CHAN_LOG_SIZE_IN_BYTES

STREAM_SIZE_IN_BYTES
STREAM_CMD_SIZE_IN_BYTES
STREAM_QUEUE_SIZE
STREAM_HDR_SIZE
MAX_CHANS
OUT_STRM_OFFSET
IN_STRM_OFFSET
CHAN_LOG_OFFSET
APIPI_BASE_SEG
APIPI_BASE_VA
APIPI_LIMIT_VA
APIPI_LAST_SEG
IPI_CB_BASE
CPIPI_BASE_SEG
CPIPI_BASE_VA
CPIPI_LIMIT_VA
CPIPI_LAST_SEG
IPI_CB_BASE
VCP_TRAP

3.1.2.14 Invocation Parameter Structure Constants — IPcons.h

This file contains constants related to the Invocation Parameter Structure.

Constants:

SUCCESS
FAILURE
SYSTEM_FAILURE
BAD_CAPA
BAD_OBJECT
LOCAL_BAD_PASSED_CAPA
REMOTE_BAD_PASSED_CAPA
LOCAL_CLIST_FULL
REMOTE_CLIST_FULL
THREAD_BREAK
RQST_PARM_OVH
RPLY_PARM_OVH
PARM_OVH
PARM_MAX

3.1.2.15 Kernel Constants — Kcons.h

This file contains constants related to the kernel.

Constants:

KERN_TEXT
KERN_DATA
KCODE_PPS
KCODE_SEGS
KDATA_PPS
KDATA_SEGS
INTR_STACK
MAX_INV_DEPTH
MAX_CAPA
CT_CSTK_EXT
EXTENTS_PER_CT
CO_CODE_EXT
CO_DATA_EXT
CO_HEAP_EXT
CO_MAPFIL_EXT
EXTENTS_PER_CO
DEFAULT_EXTENTS_PER_CO
KERNEL_VA_P
CT_SEG_INDX
CO_SEG_INDX
THREAD_VA_P
OBJ_VA_P
KERN_SMS
MON_PPS
LOWCORE_PPS
KERN_PPS
LOWCORE_PM
FIRST_K_CODE_PM
LAST_K_CODE_PM
FIRST_K_DATA_PM
LAST_K_DATA_PM
INVALID_PM
FAULT_PM
FIRST_KIO_DEV_PM
LAST_KIO_DEV_PM
MONITOR_PM
KERN_PMS
KERN_HEAP_EXT
KERN_PHEAP_EXT

NUM_KERN_EXTENTS
MAX_MEM_EXTENTS

3.1.2.16 Multibus Page VM Mapping Constants — MBMcons.h

This file contains constants related to the Multibus Page VM Mapping.

Constants:

MBM_SIZE
MBM_ALL_ALLOCD
MBM_NONE_ALLOCD

3.1.2.17 Memory Management Unit Constants — MMUcons.h

This file contains constants related to the Machine Dependent Memory Management, for both the Sun1 and the Sun2 MMU.

Constants:

US_DATA_FC
US_PROG_FC
MMU_FC
SU_DATA_FC
SU_PROG_FC
CPU_FC
PMC_SIZE
SMC_SIZE
SDS_PER_SM
PDS_PER_PM
PMS_PER_PMC
SMS_PER_SMC
PAGE_SIZE
SEG_SIZE
NUM_PHYS_PGS
NUM_PHYS_PGS
NUM_PHYS_PGS
LO_CTX_NUM
HI_CTX_NUM
LO_SEG_NUM
HI_SEG_NUM
INVALID_PME
VALID_PME
UNUSED_PG
SU_R
SU_W
SU_X

US_R
US_W
US_X
RWXRWX
RWXRW_
RWXR_X
RWXR__
RWX_WX
RWX_W_
RWX__X
RWX___
RW_RWX
RW_RW_
RW_R_X
RW_R__
RW__WX
RW__W_
RW___X
RW____
R_XRWX
R_XRW_
R_XR_X
R_XR__
R_X_WX
R_X_W_
R_X__X
R_X___
R__RWX
R__RW_
R__R_X
R__R__
R___WX
R___W_
R____X
R_____
_WXRWX
WXRW
_WXR_X
_WXR__
_WX_WX

_WX_W_
_WX__X
_WX___
_W_RWX
_W_RW_
_W_R_X
_W_R__
_W__WX
_W__W_
_W___X
_W____
__XRWX
__XRW_
__XR_X
__XR__
__X_WX
__X_W_
__X__X
__X___
___RWX
___RW_
___R_X
___R__
____WX
____W_
_____X
______
OBMEM
NXM
MBMEM
MBIO
OB_MEM
OB_IO
MB_MEM
MB_IO
__
_M
A_
AM
__

_M
A_
AM
MB_MEM_BASE
MB_MEM_LIMIT
PMC_ADDR
SMC_ADDR
CTX_REG
PARALLEL_PORT
P2_PHYS_MEM
BW_FRAME_BUF
KB_MOUSE_UART
VIDEO_CTL_REG
EPROM
ENCRYPT_PROC
IPI_INTR_GEN
PARALLEL_PORT
SERIAL_PORT
TIMER
REAL_TIME_CLK
P1_DVMA_BASE
P1_MEM
P1_IO
PG_MAP
SEG_MAP
CTX_REGS
SU_CTX_REG
US_CTX_REG
ID_PROM
DIAG_REG
BUS_ERR_REG
SYS_ENABLE_REG
US_CTX_MASK
SU_CTX_MASK
SEG_CTX_MASK
SEG_PROT_MASK
SEG_INDX_MASK
PG_CTL_MASK
PG_TYPE_MASK
PG_INDX_MASK

PG_VALID_MASK
PG_PROT_MASK
PG_TYPE_MASK
PG_CTL_MASK
PG_INDX_MASK
FAULT_MASK
FAULT_TYPE_MASK
PG_VALID_SHIFT
PG_PROT_SHIFT
PG_TYPE_SHIFT
PG_CTL_SHIFT
SEG_MASK
PG_MASK
PD_MASK
SD_MASK
PG_SHIFT
SEG_SHIFT
SET_CTX
LOAD_SD
LOAD_PD
SET_SD
SET_PD
GET_CTX
GET_SD
GET_PD
GET_SEG_NUM
GET_PAGE_NUM
SEG_CTX
SEG_PROT
SEG_INDX
PG_CTL
GET_SEG_NUM
GET_PAGE_NUM
SEG_INDX
PG_VALID
PG_PROT
PG_CTL
LOWEST_VA_IN_SEG
HIGHEST_VA_IN_SEG
LOWEST_VA_IN_PAGE

HIGHEST_VA_IN_PAGE
LOWEST_PG_INDEX_IN_SEG
HIGHEST_PG_INDEX_IN_SEG
PG_TYPE
PG_INDX
EVEN_UP
EVEN_DOWN
ROUND_LONG
INVALID_SD_INDX
FAULT_SD_INDX
INVALID_SD
FAULT_SD
INFILE_PD
SCRATCH_PD
PB_SCRATCH_PD
ILLEGAL_PD
TEMP_PG_PD
INFILE_PD
SCRATCH_PD
PB_SCRATCH_PD
ILLEGAL_PD
CSTK_OVERFLOW_PD
FAULT_PD
MONITOR_PD
TEMP_PG_PD
PB_PROT
HEAP_CTL_PROT
HEAP_DATA_PROT
PG_HEAP_PROT
CLIENT_STACK_PROT

3.1.2.18 *ObjectManager* Constants — OMcons.h

This file contains constants related to the ObjectManager System Service Object.

Constants:

OM_CREATE
OM_DELETE
OM_REGISTER_WKABID
OM_REPLICATED_CREATE
OM_CREATE_BACKUP
OM_SELECT_BACKUP
OM_ACTIVATE_BACKUP

OM_MONITOR_BACKUP
OM_ANCHOR
OBJECT_MANAGER_OBJ_ID
OM_UNAVAILABLE_OR_UNDEFINED_OBJECT_TYPE
OM_UNABLE_TO_CREATE_CO
OM_ALPHABIT_NOT_AN_OBJECT_TYPE
OM_WKABID_LIST_FULL
OM_ALPHABIT_NOT_AN_OBJECT_INST
OM_ALPHABIT_NOT_IN_LOCAL_DICT
OM_UNDEFINED_OPERATION
OM_BAD_REPLICATION_FACTOR
OM_UNABLE_TO_CREATE_BACKUP
OM_UNABLE_TO_CREATE_MONITOR_THREAD
OM_OBJECT_IS_PRIMARY_COPY
OM_UNABLE_TO_ACTIVATE_BACKUP
OM_BACKUP_ALREADY_EXISTS

3.1.2.19 Virtual Memory Paging Constants — PGcons.h

This file contains constants related to the Virtual Memory Paging.

Constants:

PAGER_EVENT
PG_HWM
PP_LWM
PP_NWM
PP_HWM

3.1.2.20 Kernel Page Heap Constants — PHcons.h

This file contains constants related to the Kernel Page Heap.

Constants:

FULL_PH_SEG
EMPTY_PH_SEG
REAL_PG
DUMMY_PG
MAX_PH_SEGS
ABS_MAX_PH_SEGS

3.1.2.21 Object Replication Constants — REPcons.h

This file contains constants related to Object Replication.

Constants:

MAX_BACKUP_COPIES

3.1.2.22 Semaphore Constants — SScons.h

This file contains constants related to the Semaphore System Service Object.

Constants:

MAX_PRIVATE_SEMAPHORES
ALLOCSEM
DEALLOCSEM
PSEM
VSEM
VALLSEM
SEM_MGR_OBJ_ID

3.1.2.23 SUN Hardware Related Constants — SUNcons.h

This file contains constants related to the hardware (of the Sun 1.5 and Sun 2.0 boards).

Constants:

NUM_REGS
TRACE_MASK
SUPER_MASK
SUPER
USER
PRI_MASK
NOT_PRI_MASK
VEC_OFFSET
LEVEL7
LEVEL6
LEVEL5
LEVEL4
LEVEL3
LEVEL2
LEVEL1
LEVEL0
INT1
INT2
INT3
INT4
INT5
INT6
INT7
MAX_PRI
TIMR_PRI
TIMR_PRI

UART_PRI
UART_PRI
ENET_PRI
MIN_PRI
S_D0
S_D1
S_D2
S_D3
S_D4
S_D5
S_D6
S_D7
S_A0
S_A1
S_A2
S_A3
S_A4
S_A5
S_A6
S_A7
R_D0
R_D1
R_D2
R_D3
R_D4
R_D5
R_D6
R_D7
R_A0
R_A1
R_A2
R_A3
R_A4
R_A5
R_A6
R_A7
ALL_REG
B_MOST_REG
F_MOST_REG
B_MANY_REG

F_MANY_REG
B_SOME_REG
F_SOME_REG
B_C_REG
F_C_REG
NO_REG
MONITOR_STK

3.1.2.24 System-Wide Constants — SYScons.h

This file contains constants that have system-wide significance.

Constants:

NULL
NIL
DISABLE
ENABLE
SUPER_MODE
USER_MODE
BYTE_MASK
SHORT_MASK
INT_MASK
LONG_MASK
BYTE_SIZE
SHORT_SIZE
INT_SIZE
LONG_SIZE

3.1.2.25 Timer Object Constants — TIMEcons.h

This file contains constants related to the time System Service Object.

Constants:

GET_TIME
SLEEP
SPIN

3.1.2.26 On-Board Timer Constants — TMRcons.h

This file contains constants related to the on-board timer.

Constants:

TC_Reg
TC_Cmd
TC_Reg
TC_Cmd
TMR_Reg

TMR_Cmd
TMR_MASK
TC_Master
TC_Mode
TC_Load
TC_Hold
TC_bin
TC_BCD
TC_EnbInc
TC_DisInc
TC_8bit
TC_16bit
TC_FoutOn
TC_FoutOff
TC_DisCmp2
TC_EnbCmp2
TC_DisCmp1
TC_EnbCmp1
TC_DisTOD
TC_LdDPtr
TC_Arm
TC_Ld
TC_LdArm
TC_DarmSav
TC_Hld
TC_Disarm
TC_SetOut
TC_ClrOut
TC_Step
TC_SMM14
TC_SMM12
TC_SMM13
TC_CMM14
TC_CMM12
TC_CMM13
TC_Reset
TC_GateMsk
TC_NoGate
TC_Rise
TC_Fall

TC_TCN_1
TC_Src1
TC_F1
TC_F2
TC_F3
TC_F4
TC_F5
TC_DisSpG
TC_RldLd
TC_CntRep
TC_BinCnt
TC_CntDown
TC_CntUp
TC_OutLow
TC_OutHighP
TC_OutTog
TC_OutHighZ
TC_OutLowP
STDL
STDH
STDBM
STDHhold
STDLhold
STDHmode
STDLmode
STDHload
STDLload
STDChld
STDCclr
STDClarm
STIMER_INTR
ITIMER_INTR
ST
STBM
STmode
STload
SThold
STClarm
STChld
STCclr

STCdarm
STDL
STDH
STDBM
STDHhold
STDLhold
STDHmode
STDLmode
STDHload
STDLload
STDChld
STDCclr
STDClarm
STIMER_INTR
ITIMER_INTR
IT
ITBM
IThold
ITmode
ITload
ITChld
ITCclr
ITClarm
ITCdarm

3.1.2.27 *ThreadManager* Object Constants — TMcons.h

This file contains constants related to the ThreadManager System Service Object.

Constants:

TM_CREATE
TM_DELETE
THREAD_MANAGER_OBJ_ID
TM_UNDEFINED_OPERATION

3.1.2.28 Resource Tracking Constants — TRKcons.h

This file contains constants related to resource tracking.

Constants:

TRACK_TABLE_SIZE
UNDOALLOC
UNDOPV
MAX_RECORD

3.1.2.29 Trap Related Constants — TRPcons.h

This file contains constants related to traps.

Constants:

EXIT_TRAP
I_INV_TRAP
KO_INV_TRAP
CO_INV_TRAP
CO_RET_TRAP
LOG_EV_TRAP
ABLK_TRAP
RET_MON_TRAP

3.1.2.30 Thread Section Trimming Constants — TSECcons.h

This file contains constants related to Thread Section Trimming.

Constants:

SECTION_TABLE_SIZE
MAX_SEC_HASH

3.1.2.31 TempServices Object Constants — TScons.h

This file contains constants related to the TempServices System Service Object.

Constants:

TS_ADD_WKCAPA
TS_MAKE_READY
TS_SET_IMPORTANCE
TS_STATS_ON
TS_STATS_OFF
TEMP_SERVICES_OBJ_ID
TS_TOO_FEW_ARGUMENTS
TS_UNABLE_TO_ADD_CAPA
TS_ALPHABIT_NOT_IN_LOCAL_DICT
TS_NOT_A_KT
TS_NO_SCHEDID
TS_UNDEFINED_SERVICE

3.1.2.32 Console Related Constants — TTYcons.h

This file contains constants related to the TTY.

Constants:

TTY_CHAN_BUF_SIZE
UART_A_BASE
UART_A_DATA
UART_A_CTL
UART_A_BASE

UART_A_DATA
UART_A_CTL
UART_B_BASE
UART_B_DATA
UART_B_CTL
A_BASE
A_DATA
A_CTL
A_BASE
A_DATA
A_CTL
B_BASE
B_DATA
B_CTL
BREAK_ABORT
TX_UNDERRUN_EOM
CTS
SYNC_HUNT
DCD
TX_BUF_EMPTY
INTR_PENDING
RX_BUF_FULL
END_OF_FRAME
CRC_FRAMING_ERROR
RX_OVERRUN_ERROR
PARITY_ERROR
RESIDUE_CODE_0
RESIDUE_CODE_1
RESIDUE_CODE_2
ALL_SENT
NULL_CODE
RESET_RX_CRC_CHECK
RESET_TX_CRC_GEN
RESET_TX_UNDERRUN_EOM
SEND_ABORT
RESET_EXT_STATUS_INTRS
CHANNEL_RESET
ENABLE_INTR_ON_NXT_RX
RESET_TX_INTR_PENDING
ERROR_RESET

RESET_HIGHEST_IUS
UART_R0
UART_R1
UART_R2
UART_R3
UART_R4
UART_R5
UART_R6
UART_R7
WAIT_ENABLE
WAIT_DISABLE
WAIT_ON_RX
WAIT_ON_TX
RX_INTR_DISABLE
RX_INTR_ON_FIRST_CHAR
RX_INTR_ON_ALL_CHARS1
RX_INTR_ON_ALL_CHARS2
VARIABLE_INTR_VECTOR
FIXED_INTR_VECTOR
TX_INTR_ENABLE
TX_INTR_DISABLE
EXT_INTR_ENABLE
EXT_INTR_DISABLE
PIN_10_IS_SYNDET
PIN_10_IS_RTS
VECTORED_INTR
NON_VECTORED_INTR
_8085_MODE_1
_8085_MODE_2
_8086_8088_MODE
PRIORITY_1
PRIORITY_2
A_AND_B_INTR
A_DMA_B_INTR
A_AND_B_DMA
RX_5_BIT_CHARS
RX_7_BIT_CHARS
RX_6_BIT_CHARS
RX_8_BIT_CHARS
AUTO_ENABLES

ENTER_HUNT_MODE
RX_CRC_ENABLE
ADDR_SRCH_MODE
SYNC_CHAR_LOAD_INHIBIT
RX_ENABLE
X1_CLK_MODE
X16_CLK_MODE
X32_CLK_MODE
X64_CLK_MODE
_8_BIT_SYNC_CHAR
_16_BIT_SYNC_CHAR
SDLC_HDLC_MODE
EXT_SYNC_MODE
SYNC_MODES_ENABLE
_1_STOP_BIT
_1_AND_HALF_STOP_BITS
_2_STOP_BITS
EVEN_PARITY
ODD_PARITY
PARITY_ENABLE
PARITY_DISABLE
DTR
TX_5_BIT_CHARS
TX_7_BIT_CHARS
TX_6_BIT_CHARS
TX_8_BIT_CHARS
SEND_BREAK
TX_ENABLE
SDLC_CRC16
RTS
TX_CRC_ENABLE
UART_RESET
UART_DEFAULTS
ENABLE_TX_RX_INTRS
DISABLE_TX_RX_INTRS
RX_CONFIG
TX_CONFIG
ENB_TRI
DIS_TRI
ENB_R8A

SB1_C16
R_D_T8
RXBF
TXBE
BREAK_ABORT
TX_UNDERRUN_EOM
CTS
SYNC_HUNT
DCD
TX_BUF_EMPTY
ZERO_COUNT
RX_BUF_FULL
END_OF_FRAME
CRC_FRAMING_ERROR
RX_OVERRUN_ERROR
PARITY_ERROR
RESIDUE_CODE_0
RESIDUE_CODE_1
RESIDUE_CODE_2
ALL_SENT
RR2_UART_A
RR2_STATUS_MASK
RR2_TX_EMPTY
RR2_EXT_OR_STATUS
RR2_RCVD_CHAR
RR2_SPECIAL
CHAN_A_RX_IP
CHAN_A_TX_IP
CHAN_A_EXT_STATUS_IP
CHAN_B_RX_IP
CHAN_B_TX_IP
CHAN_B_EXT_STATUS_IP
ONE_CLK_MISSING
TWO_CLKS_MISSING
LOOP_SENDING
ON_LOOP
BREAK_ABORT_IE_RO
TX_UNDERRUN_EOM_IE_RO
CTS_IE_RO
SYNC_HUNT_IE_RO

DCD_IE_RO
ZERO_COUNT_IE_RO
NULL_CODE
RESET_RX_CRC_CHECK
RESET_TX_CRC_GEN
RESET_TX_UNDERRUN_EOM
SELECT_HIGH_REG_GROUP
RESET_EXT_STATUS_INTRS
SEND_ABORT
ENABLE_INTR_ON_NXT_RX
RESET_TX_INTR_PENDING
ERROR_RESET
RESET_HIGHEST_IUS
UART_R0
UART_R1
UART_R2
UART_R3
UART_R4
UART_R5
UART_R6
UART_R7
UART_R8
UART_R9
UART_R10
UART_R11
UART_R12
UART_R13
UART_R14
UART_R15
WAIT_DMA_REQUEST_ENABLE
WAIT_DMA_REQUEST_FUNC
WAIT_DMA_REQUEST_RX_TX
RX_INTR_DISABLE
RX_INTR_ON_FIRST_CHAR
RX_INTR_ON_ALL_CHARS
RX_INTR_ON_SPECL_ONLY
PARITY_IS_SPECL_COND
TX_INTR_ENABLE
TX_INTR_DISABLE
EXT_INTR_ENABLE

EXT_INTR_DISABLE
RX_5_BIT_CHARS
RX_7_BIT_CHARS
RX_6_BIT_CHARS
RX_8_BIT_CHARS
AUTO_ENABLES
ENTER_HUNT_MODE
RX_CRC_ENABLE
ADDR_SRCH_MODE
SYNC_CHAR_LOAD_INHIBIT
RX_ENABLE
X1_CLK_MODE
X16_CLK_MODE
X32_CLK_MODE
X64_CLK_MODE
_8_BIT_SYNC_CHAR
_16_BIT_SYNC_CHAR
SDLC_MODE
EXT_SYNC_MODE
SYNC_MODES_ENABLE
_1_STOP_BIT
_1_AND_HALF_STOP_BITS
_2_STOP_BITS
EVEN_PARITY
ODD_PARITY
PARITY_ENABLE
PARITY_DISABLE
DTR
TX_5_BIT_CHARS
TX_7_BIT_CHARS
TX_6_BIT_CHARS
TX_8_BIT_CHARS
SEND_BREAK
TX_ENABLE
SDLC_CRC16
RTS
TX_CRC_ENABLE
NO_RESET
RESET_CHAN_B
RESET_CHAN_A

FORCE_HARDWARE_RESET
STATUS_HIGH
STATUS_LOW
MIE
DLC
NV
VIS
CRC_PRESET_TO_1
CRC_PRESET_TO_0
NRZ
NRZI
FM1
FM0
GO_ACTIVE_ON_ROLL
MARK_FLAG_IDLE
ABORT_FLAG_ON_UNDERRUN
LOOP_MODE
_6_BIT_SYNC
_8_BIT_SYNC
RTXC_XTAL
NO_XTAL
RCLK_IS_RTXC_PIN
RCLK_IS_TRXC_PIN
RCLK_IS_BR_GEN_OUT
RCLK_IS_DPLL_OUT
TCLK_IS_RTXC_PIN
TCLK_IS_TRXC_PIN
TCLK_IS_BR_GEN_OUT
TCLK_IS_DPLL_OUT
TRXC_IS_OUTPUT
TRXC_IS_INPUT
TRXC_OUT_IS_XTAL_OUT
TRXC_OUT_IS_TCLK
TRXC_OUT_IS_BR_GEN_OUT
TRXC_OUT_IS_DPLL_OUT
NULL_COMMAND
ENTER_SEARCH_MODE
RESET_MISSING_CLK
DISABLE_DPLL
SET_SRC_IS_BR_GEN

SET_SRC_IS_RTXC
SET_FM_MODE
SET_NRZI_MODE
LOCAL_LOOPBACK
AUTO_ECHO
DTR_REQUEST_FUNCTION
BR_GEN_SRC
BR_GEN_ENABLE
BREAK_ABORT_IE
TX_UNDERRUN_EOM_IE
CTS_IE
SYNC_HUNT_IE
DCD_IE
ZERO_COUNT_IE
ENABLE_TX_RX_INTRS
DISABLE_TX_RX_INTRS
RX_CONFIG
TX_CONFIG
CR
LF

3.1.2.33 Virtual Memory Management Constants — VMcons.h

This file contains constants related to the Virtual Memory Management Facility.

Constants:

FIRST_K_SEG
LAST_K_SEG
LOWCORE_SEG
FIRST_K_CODE_SEG
NUM_K_CODE_SEGS
LAST_K_CODE_SEG
FIRST_K_DATA_SEG
LAST_K_DATA_SEG
FIRST_K_HEAP_CTL_SEG
FIRST_K_HEAP_DATA_SEG
FIRST_K_HEAP_SEG
LAST_K_HEAP_SEG
FIRST_K_PGHEAP_SEG
LAST_K_PGHEAP_SEG
FIRST_INT_VEC_PG
LAST_INT_VEC_PG
INT_STK_PG

TEMP_PG
FIRST_EV_BUF_PG
LAST_EV_BUF_PG
SCRATCH_PG
LIMIT_PGHEAP_PG
KERN_LIMIT_PG
INT_STK_VA
KERN_BASE_VA
KERN_LIMIT_VA
KERN_STAT_DATA_VA
KERN_PGHEAP_BASE_VA
KERN_PGHEAP_LIMIT_VA
KERN_HEAP_BASE_VA
KERN_HEAP_LIMIT_VA
K_HEAP_DATA_BASE_VA
K_HEAP_CTL_BASE_VA
TEMP_STK_PG_VA
TEMP_PG_VA
SCRATCH_PG_VA
HIGHEST_KERN_VA
FIRST_CT_SEG
LAST_CT_SEG
MAX_CT_STK_SEGS
LAST_USED_CT_SEG
FIRST_USED_CT_SEG
LAST_CT_STK_SEG
FIRST_CT_STK_SEG
BIUPARMBLK_PG
INPARMBLK_PG
OUTPARMBLK_PG
CT_STK_UDFL_PG
CT_STK_FIRST_PG
BASE_CT_VA
LIMIT_CT_VA
INPARMBLK_VA
OUTPARMBLK_VA
CT_STK_BASE_VA
CT_STK_LIMIT_VA
FIRST_CO_SEG
LAST_CO_SEG

FIRST_CO_CODE_SEG
LAST_CO_CODE_SEG
FIRST_CO_DATA_SEG
LAST_CO_DATA_SEG
FIRST_CO_HEAP_SEG
FIRST_CO_MAPFIL_SEG
LAST_CO_HEAP_SEG
MIN_CO_MAPFIL_SEG
CO_HEAP_BASE_VA
CO_CODE_BASE_VA
CO_DATA_BASE_VA
BASE_CO_VA
LIMIT_CO_VA
FIRST_KIO_SEG
LAST_KIO_SEG
SCRATCH_SEG
FIRST_MBM_SEG
SMM_SIZE_IN_BYTES
LAST_MBM_SEG
EXCEPT_VEC_TAB_SEG
FIRST_USED_KIO_SEG
FIRST_KIO_DEV_SEG
LAST_KIO_DEV_SEG
FIRST_MONITOR_SEG
LAST_MONITOR_SEG
FIRST_MULTIBUS_P1_SEG
LAST_MULTIBUS_P1_SEG
FIRST_DVMA_SEG
LAST_DVMA_SEG
EXCEPT_VEC_TAB_PG
TIMER_PG
RTC_PG
DES_PG
PARALLEL_PORT_PG
SERIAL_PORT_PG
EXCEPT_VEC_TAB_VA
TIMER_VA
RTC_VA
DES_VA
PARALLEL_PORT_VA

SERIAL_PORT_VA
MAX_WIRED_PGS
MAX_WIRED_PGS
SDS_PER_CO
SDS_PER_CT
NUM_STICKY
STICK
NONSTICK
STICK_PM
UNSTICK_PM

3.1.2.34 Well-Known-Capabilities Constants — WKCcons.h

This file contains constants related to the Well-Known Capabilities.

Constants:

MAX_WKABIDS
OBJECT_MANAGER_WKABID
THREAD_MANAGER_WKABID
SEMAPHORE_MANAGER_WKABID
TEMP_SERVICES_WKABID
TIME_WKABID
CTHREADSTUB_OTID
CTSTUB_OP
CTRISTUB_OP
INITIALIZATIONOBJECT_OTID
INITAPPLICATION_OP

3.1.2.35 Alphabit Identifier Header — abitid.h

This file contains the header information for Alphabit Identifiers. This includes the definition of the Alphabit ID and macros for manipulating these structures and their internal representation.

Data Structures:

abid
abidptr

Macros:

ID_MATCH
ID_COPY
ID_SWAP
IS_SSO
IS_IO

IS_IRO
IS_RO
IS_ERO
IS_WK
IS_WKSSO
IS_LOCAL
IS_REMOTE

3.1.2.36 Abort Block Header — abort.h

This file relates to the Abort Block Header.

Data Structures:

ablkregsetptr
abortparmptr
ablkregsetstr
abortparmstr

3.1.2.37 AP Interprocessor Interface Header — apipi.h

This file relates to the AP Interprocessor Interface.

Data Structures:

RemInvRqstptr
RemInvRqststr

Macros:

MULTIBUS_MEM_START_1PT5
VADDR_IN_CP_VAS

3.1.2.38 Bit Vector Header — bitvector.h

This file relates to bit vector manipulation.

Data Structures:

BV

Macros:

BV_LENGTH_IN_LONGS
BIT_VECTOR
EXTERN_BIT_VECTOR
BIT_VECTOR_PTR
EXTERN_BIT_VECTOR_PTR
BIT_VECTOR_ARRAY
EXTERN_BIT_VECTOR_ARRAY
SELECT_BIT_VECTOR
BINDX
LINDX
SET_BIT

CLEAR_BIT
BIT_VALUE
BIT_SET
BIT_CLEAR
CLEAR_BV
COPY_BV
FIRST_CLEAR_BIT

3.1.2.39 Capabilities Header — capa.h

This file relates to capabilities.

Data Structures:

capa
capaptr

3.1.2.40 Client Thread Skeleton Pool Header — ctspool.h

This file relates to the Client Thread Skeleton Pool

Constants:

CTSP_HWM

3.1.2.41 Communications Virtual Machine Timer Header — cvm_token.h

This file relates to the Communication Virtual Machine Timer Facility.

Data Structures:

tokenptr
tknsrcdest
enettkntype_str
tokenhdrstr
tokenstr

Macros:

TKN_INSERT
TKN_REMOVE
TOKEN_BODY_SIZE
TOKEN_BODY_ADDR
AP_INTF
ENET_INTF
UART_INTF
TIMER_INTF
EXCEPTION_HDLR
MAX_TKN_DATA_ITEMS

3.1.2.42 Dictionary Header — dict.h

This file relates to the Dictionary.

Macros:

DICTHASH

3.1.2.43 Event Logging Header — event.h

This file relates to the Event Logging Service

Data Structures:

eventptr
eventstr

3.1.2.44 External Protocol Interface Header — ext_proto_if.h

This file relates to the External Protocol Interface.

Data Structures:

extmsgptr
extmsgbodyptr
EXTmsgtkn_ptr
EXTmsgshorttkn_ptr
EXTtimeouttkn_ptr
extmsgstr
extmsgbodystr
EXTmsgtkn_str
EXTmsgshorttkn_str
EXTtimeouttkn_str

Macros:

EXT_SENDMSG_TKN
EXT_MSGRCVD_TKN
EXT_TIMEOUT_TKN
MAX_EXT_MSG_LEN
EXT_MSG_RCVD_CMD

3.1.2.45 Secondary Storage Subsystem Header — file.h

This file relates to the Secondary Storage Subsystem.

Data Structures:

FCBptr
memextent
installFCBrqp
installFCBrqpptr
installFCBrpp
installFCBrppptr
installFCBpb
installFCBpbp
fileattr

filetype
FCBstr

3.1.2.46 Kernel Heap Header — heap.h

This file relates to the Kernel Heap.

Data Structures:

blkdescstr
blkdescptr
blktblentrystr
blktbl

3.1.2.47 Invocation Parameter Structure Header — invparms.h

This file relates to the Invocation Parameter Structure.

Data Structures:

parmblkptr
rqstpblkptr
rplypblkptr
altrplyhdrptr
rqsthdrstr
rplyhdrstr
altrplyhdrstr
rqstpblkstr
rplypblkstr

3.1.2.48 Interprocessor Interface Header — ipi.h

This file relates to the Interprocessor Interface

Data Structures:

cmdheaderptr
strmhdrptr
chandescptr
ipiCBptr
cmdheaderstr
strmhdrstr
ipicmdbuf
ipicmdbufptr
chandescstr
ipiCBstr

3.1.2.49 Kernel Per-Se Header — kernel.h

This file relates to the kernel per-se.

Data Structures:

abheadptr

KCBptr
COTCBptr
COCBptr
CTCBptr
KTCBptr
KOTCBptr
KOCBptr
WKCBptr
time
abtype
threadstate
abheadstr
KCBstr
COTCBstr
COCBstr
CTCBstr
KOTCBstr
KOCBstr
KTCBstr
WKCBstr

3.1.2.50 Kernel Tables Header — ktables.h

This file relates to the kernel tables.

Data Structures:

KOTentryptr
KOTentrystr

3.1.2.51 Kernel Thread Skeleton Pool Header — ktspool.h

This file relates to the Kernel Thread Skeleton Pool

Macros:

KTSP_HWM

3.1.2.52 Load Module Data Structure Header — loadmodule.h

This file relates to the Load Module Data Structure.

Data Structures:

lmhdrptr
COTlmptr
LMtype
LMid
lmhdrstr
COTlmstr

lmhdr

3.1.2.53 Monitor Active Nodes Protocol Header — man_proto_if.h

This file relates to the Monitor Active Nodes Protocol.

Constants:

MAN_MAX_NODES
MAN_MAX_NODENUM
MAN_NEWSTATUS_TKN
MAN_DUMP_RQST_TKN
MAN_DUMP_RPLY_TKN
MAN_DUMP_RPLY_SHORT_TKN
MAN_TIMER_TKN

Data Structures:

MANnewstatustkn_ptr
MANdumprqsttkn_ptr
MANdumprplytkn_ptr
MANdumprplyshorttkn_ptr
MANtimertkn_ptr
MANnewstatustkn_str
MANdumprqsttkn_str
MANdumprplytkn_str
MANdumprplyshorttkn_str
MANtimertkn_str

3.1.2.54 Multibus Page VM Mapping Header — mbmapping.h

This file relates to the Multibus Page VM Mapping

Data Structures:

shortBV
MBMcellstr

Macros:

MBMAPPING_HDR
MBM_BV_BINDX
MBM_BV_WINDX

3.1.2.55 Virtual Memory Header — mmu.h

This file relates to the virtual memory.

Data Structures:

ctx
segd
pgd

3.1.2.56 Network I/O Object Header — netio.h

This file relates to the Network IO Object.

Data Structures:

EXTqueueptr
EXTqueuestr

Constants:

EXT_Q_SIZE

3.1.2.57 Virtual Memory Paging Header — paging.h

This file relates to the Virtual Memory Paging.

Data Structures:

pagestate

3.1.2.58 Kernel Page Heap Header — pgheap.h

This file relates to the Kernel Page Heap

Data Structures:

listclass
PHentry
pageheap

3.1.2.59 Page Transfer Protocol Header — pt_proto_if.h

This file relates to the Page Transfer Protocol

Constants:

PT_MAX_DESTS
PT_CONV
PT_SEND_PAGE_TKN
PT_SEND_PAGE_SHORT_TKN
PT_PAGE_SENT_TKN
PT_RCVD_PAGE_TKN
PT_SEND_INFO_TKN
PT_INFO_SENT_TKN
PT_RCVD_INFO_TKN
PT_KERN_RPLY_TKN
PT_WRITE_SYNC_TKN
PT_SYNC_READ_TKN
PT_TIMEOUT_TKN

Data Structures:

destinfoptr
PTsendpagetkn_ptr
PTsendpageshorttkn_ptrPTpagesenttkn_ptr
PTrcvdpagetkn_ptr

PTsendinfotkn_ptr
PTinfosenttkn_ptr
PTrcvdinfotkn_ptr
PTwritesynctkn_ptr
PTkernrplytkn_ptr
PTsyncreadtkn_ptr
PTtimertkn_ptr
MBaddr
destinfostr
PTsendpagetkn_str
PTsendpageshorttkn_str
PTpagesenttkn_str
PTrcvdpagetkn_str
PTsendinfotkn_str
PTinfosenttkn_str
PTrcvdinfotkn_str
PTkernrplytkn_str
PTwritesynctkn_str
PTsyncreadtkn_str
PTtimertkn_str

3.1.2.60 Reliable Basic Invoke Protocol Header — rbi_proto_if.h

This file relates to the Reliable Basic Invoke Protocol.

Constants:

RBI_INV_RQST
RBI_INV_RPLY
RBI_RQST_TSYNC
RBI_RPLY_TSYNC
RBI_CONV
RBI_ADD_ABID_TKN
RBI_REMOVE_ABID_TKN
RBI_OUTRQST_TKN
RBI_OUTRPLY_TKN
RBI_INRQST_TKN
RBI_INRPLY_TKN
RBI_ABORT_THREAD_TKN
RBI_TIMEOUT_TKN

Data Structures:

RBIaddabidtkn_ptr
RBIrmvabidtkn_ptr

RBIoutrqsttkn_ptr
RBIoutrplytkn_ptr
RBIinrqsttkn_ptr
RBIinrplytkn_ptr
RBIabortthreadtkn_ptr
RBItimertkn_ptr
RBIaddabidtkn_str
RBIrmvabidtkn_str
RBIoutrqsttkn_str
RBIoutrplytkn_str
RBIinrqsttkn_str
RBIinrplytkn_str
RBIabortthreadtkn_str
RBItimertkn_str

3.1.2.61 Object Replication Header — rep.h

This file relates to Object Replication.

Data Structures:

chkptinforecptr
chkptinforecstr

3.1.2.62 Scheduler Header — sched.h

This file relates to the Scheduler.

Data Structures:

commonSPBptr
timcritSPBptr
initialSPBptr
dinitialSPBptr
schedhdrptr
SIPBptr
initSPIBptr
dinitSPIBptr
TCSPIBptr
SIIBptr
oSIIBptr
abstime
commonSPBstr
timcritSPBstr
initialSPBstr
dinitialSPBstr
schedhdrstr

SIPBstr
initSPIBstr
dinitSPIBstr
TCSPIBstr
SIIBstr
oSIIBstr

3.1.2.63 Scheduler Statistics Header — schedstats.h

This file relates to the Scheduling statistics.

Data Structures:

FIFOstatsptr
RRstatsptr
SPRIstatsptr
DLstatsptr
SPTstatsptr
SLKstatsptr
VDstatsptr
BEstatsptr
schedstatsptr
FIFOstats_info_str
RRstats_info_str
SPRIstats_info_str
DLstats_info_str
SPTstats_info_str
SLKstats_info_str
VDstats_info_str
BEstats_info_str
schedstats_str

3.1.2.64 SUN PROM Code Header — sunromvec.h

This file relates to the features supported by the Sun ROM code.

Constants:

romp
RomVecPtr
v_SunRev
v_MemorySize
v_EchoOn
v_InSource
v_OutSink
v_InitGetkey
v_KeybId
v_Keybuf

v_FBAddr
v_FontTable
v_message
v_KeyFrsh
AbortEnt
v_RefrCnt
v_GlobPtr
v_KRptInitial
v_KRptTick
v_ExitOp
v_fwrstr
v_linbuf

Macros:

GetChar
PutChar
MayGet
MayPut
GetLine
GetOne
PeekChar
GetHexByte
GetNum
Message
PrintHex
GetHexWord
GetHexAddr
GetHexLong
EchoOn
InSrc
OutSink
LineBuffer
LineBufPtr
LineLength
MemorySize
SET_SEG
GET_SEG

Data Structures:

sunromvec

3.1.2.65 Kernel Synchronization Header — sync.h

This file relates to Kernel Synchronization.

Data Structures:

usrsyncptr
semaphoreptr
semaphorestr
usrsyncstr

3.1.2.66 System Service Object Header — sysserv.h

This file relates to the System Service Object.

Data Structures:

syssrv

Macros:

GET_SSO_TYPE
GET_SSO_INSTANCE

3.1.2.67 System Types Header — system.h

This file relates to the system types.

Data Structures:

u_char
u_short
u_int
u_long
vaddr
string
boolean
routptr
broutptr
sroutptr
iroutptr

Macros:

IN_PORT_VA
GET_NODE_ID
MIN
MAX
DELAY
VERBOSE
ASM
ASM1
INTR_VEC

TRAP_VEC
DO_TRAP
BUS_ERR_VEC
LOAD_SP
LABEL
G_LABEL
OLDLOGEVENT
LOGEVENT
USEREVENT
TIMINGEVENT

3.1.2.68 Thread Maintenance and Repair Protocol Header — tmar_proto_if.h

This file relates to the Thread Maintenance and Repair Protocol.

Macros:

TMAR_CONV
TMAR_POLLING_TIMER_TKN
TMAR_REFRESH_TIMEOUT_TKN
TMAR_BCAST_RETRY_TKN
TMAR_RI_ABORT_TKN
TMAR_ABORT_DONE_TKN
TMAR_START_ABORT_TKN
TMAR_START_ABORT_SHORT_TKN
TMAR_END_ABORT_TKN
TMAR_BEGIN_ABORT_TKN
TMAR_BEGIN_ABORT_SHORT_TKN

Data Structures:

TMARrefreshTOtkn_ptr
TMARbcastretrytkn_ptr
TMARRIaborttkn_ptr
TMARbeginaborttkn_ptr
TMARbeginabortshorttkn_ptr
TMARstartaborttkn_ptr
TMARstartabortshorttkn_ptr
TMARabortdonetkn_ptr
TMARendaborttkn_ptr
TMARsparmsrqsttkn_ptr
TMARsparmstkn_ptr
TMARrefreshTOtkn_str
TMARbcastretrytkn_str
TMARRIaborttkn_str

TMARbeginaborttkn_str
TMARbeginabortshorttkn_str
TMARstartaborttkn_str
TMARstartabortshorttkn_str
TMARabortdonetkn_str
TMARendaborttkn_str
TMARsparmsrqsttkn_str
TMARsparmstkn_str

3.1.2.69 Usage Resource Tracking Header — track.h

This file relates to the Usage Resource Tracking.

Data Structures:

gentrackptr
heaptrackstr
sematrackptr
tracktableptr
tracktype
semopr
gentrackstr
heaptrackstr
sematrackstr
trackrecstr
tracktablestr

3.1.2.70 Thread Section Header — tsec.h

This file relates to the Thread Section logic.

Data Structures:

sectiontableptr
sectionrecstr
sectiontablestr

3.1.2.71 Console Interface Header — tty.h

This file relates to the UART.

Data Structures:

ttyctlparam
ttyctlval
ttyqueuestr
ttyqueueptr
ttystr

3.1.2.72 Virtual Memory Header — vm.h

This file relates to the Virtual Memory logic.

Data Structures:

EDptr
pmapptr
PPentrystr
EDstr
SMentrystr
pmapstr
PMentrystr

3.1.3 Source Files

3.1.3.1 Global System Initialization — init.c

This file contains the top-level files used to initialize the Alpha operating system's kernel. This file contains the main entry point for the entire program.

Routine:

void main()

Come here in supervisor state, in context 0, at an indeterminant interrupt level.

First have to initialize the hardware — set up the kernel stack pointer, set up the trap vectors, set up the bus error trap handler, and disable interrupts.

At this point, everything is identity mapped, the kernel has the first SM, the first "N" PM's and the first "M" PP's set aside for the kernel's use.

This sets up the virtual memory for the kernel. Loads the kernel's segments and pages, and sets their protection.

Initial segments and pages for the Kernel Heap and Kernel Page Heap are not preallocated.

This initializes the Kernel Code Region and the Kernel Static Data Region (in that order). It does each region by loading a SD in the Kernel Context that points to the next reserved PM, then load the given PM with PD's of the next "K" reserved PP's.

The system timer is initialized and started running, and the event logging facility is initialized.

The phantom thread is initialized (to give the initialization code a current thread) and it is made to be the currentthread and the idlethread is set to NULL.

When all of this is done, the 'InitKern()' routine is called to start up the system.

Routine:

void InitKern()

Initialize the kernel by calling init. routines for individual components, assigning values to global variables, and starting up some of the system daemons.

This routine is called after the idle thread is created, it dispatches the next thread ready, and goes away (never to return).

Get here after the 'main()' routine has set up the memory mappings, loaded the exception vectors, and put the system stack pointer on the interrupt stack. Interrupts are disabled, and we are in supervisor state at this point.

Note that interrupts are enabled when the first client thread begins running.

Routine:

void InitExceptTab()

In order to write-protect the processor's exception vector table, it must be moved out of the lowcore segment (where it is located by default). (It can't be write-protected there because protection is controlled on a per-page basis, and the page in which it is located also contains R/W data.) This routine allocates a new page, allocates a new page map, installs the page in the page map, copies the current exception vector table from the lowcore segment into this page, sets the new table to be read-only, and modifies the processor's Vector Base Register to use the new table.

NOTE: The current definitions for setting interrupt and exception vectors writes them into the table in lowcore. This routine is called after these vectors have been initialized, so there is no problem. However, if a vector is to be altered after this routine has been called, new definitions must be created to make the update in the new table. (Alternately, this routine could be called earlier and all exception vectors could be set afterwards with a single (new) macro definition.)

Routine:

void InitKOTs()

The kernel contains a number of Kernel Object Types (KOTs). Each KOT is listed in the KOTtable, and this information is used to create a KOTCB for each of them; the KOTs are also entered in the local Dictionary (using the (pseudo-well-known) Alphabit ID contained in the KOTtable for each entry).

Routine:

void InitCOTs()

The load module is (or at least can be) composed of a collection of smaller load modules; the first is the kernel itself; this is followed by others which may contain Client Object Type definitions (or other things); this routine scans all of the smaller portions of the load module following the kernel itself; for each COT it finds, an elaborated FCB is created for the memory resident COT information and a COTCB is created and entered in the local Dictionary.

Routine:

void RegisterSSOs()

This routine registers well-known capabilities for system-service objects.

Routine:

void InitObjs()

This routine constructs objects that are intended for the kernel's user and enters them in the local node's Dictionary; (there is only one such object at this time — i.e., "CThreadStub").

Routine:

void InitThreads()

This routine sets up the scheduler's global variables. This routine is obsolete.

Routine:

void InitAppl()

Use 'ExecInvoke()' to create the 'InitializationObject,' and to give the object its well-known capabilities (for now; this will be improved later). It uses 'ExecInvoke()' again to place the 'InitializationThread' in this object, and adds the 'InitializationThread' to the ready list.

3.1.3.2 Client Thread Startup Object — CThreadStub.C

This object is used to start a client thread's execution and clean up when the thread's execution has completed.

Kernel Object:

CThreadStub

Operation:

CTStub()

This is a "special" KT in that it can directly manipulate kernel data structures. This is prohibited by the compiler in the case of user-defined kernel processes.

It is expected that this KT was built with a NULL incoming parmblk and the outgoing parmblk is the initial parmblk page for the invoked CO.

The KT's incoming parmblk has been set up by the creator with all of the necessary information to invoke the CO.

This makes an invocation on the initial object with 'ExecInvoke()' and when the invocation returns it checks if the KT was working as a result of a remote invocation and sends a response to the CP if one is needed.

Finally, the 'KillThread()' routine is called to delete the thread's control blocks, and it is responsible for returning any of the paramblk's allocated for the KT.

Operation:

CTRIStub()

This is a "special" KT in that it can directly manipulate kernel data structures. This is prohibited by the compiler in the case of user-defined kernel processes.

It is expected that this KT was built with a NULL incoming parmblk and the outgoing parmblk is the initial parmblk page for the invoked CO.

The KT's incoming parmblk has been set up by the creator with all of the necessary information to invoke the CO.

This makes an invocation on the initial object with 'InternalInvoke()' and when the invocation returns it checks if the KT was working as a result of a remote invocation and sends a response to the CP if one is needed.

Finally, the 'KillThread()' routine is called to delete the thread's control blocks, and it is responsible for returning any of the paramblk's allocated for the KT.

3.1.3.3 Communications Processor Interface Object — CommProcIntf.C

This object implements an interface to the communications processor.

Kernel Object:

CommProcIntf

Operation:

ServiceCP()

Initialize the queue that is shared with the APIPI interrupt routine and the semaphore they use for coordination. Enter a loop in which service requests from the CP are handled one at a time. (This operation should never terminate.)

3.1.3.4 Network Access Object — NetIO.C

This kernel object talks to the external communications protocol in the CP to send short (<= 256 byte) messages across the ethernet to a unix daemon that speaks the EXT protocol, and receive them.

Kernel Object:

NetIO

Operation:

Init()

Initializes the object.

Operation:

Send(IN extmsg msg)

The given message is queued.

Operation:

Receive(OUT extmsg msg)

A message is dequeued and returned. The operation waits if no message is available immediately.

3.1.3.5 Object for Object Control — ObjectManager.C

This system-service object allows creation and deletion of client objects.

Kernel Object:

ObjectManager

Operation:

Create(IN CAPA objectType, OUT CAPA instance)

Each invocation of Create returns a capa to new object of that type.

Operation:

Delete(IN CAPA instance)

Invocation of Delete will destroy the indicated object.

Operation:

RegisterWellknown(IN CAPA regularCapa, IN CAPA wellknownCapa)

This operation creates an alias, so that the ‘wellknown’ capa refers to the same thing as the ‘regular’ capa already refers to.

Operation:

```
ReplicatedCreate(IN CAPA objectType,
        IN u_long desiredReplicationFactor,
        OUT CAPA instance,
        OUT u_long actualReplicationFactor)
```

This operation resembles Create. However, it will create multiple instances of the new object, each on a different node. The instance CAPA applies to the entire set, since conceptually each object is all of the objects.

The number which are created is not necessarily the number requested, since there can be at most one replica per node. See further the section on file 'objmgmt.c'.

3.1.3.6 Object for Semaphore Usage — Semaphore.C

Kernel Object:

Semaphore

Operation:

```
PSem()
```

This is the classical 'P' operation, applied to the semaphore object which is invoked.

Operation:

```
VSem()
```

This is the classical 'V' operation, applied to the semaphore object which is invoked.

Operation:

```
VAllSem()
```

This operation unblocks all threads which may be waiting on the semaphore object. The semaphore count is reset to zero.

3.1.3.7 Object for Semaphore Creation — SemaphoreManager.C

Kernel Object:

SemaphoreManager

Operation:

```
AllocSem(IN u_long count,
        IN boolean pvundo,
        IN boolean allocundo,
        OUT CAPA Semaphore semaphore)
```

Allocates a private semaphore for an object, and returns a capa for it.

Operation:

```
DeallocSem(IN CAPA Semaphore semaphore)
```

Deallocates a semaphore.

3.1.3.8 Catchall Service Object — TempServices.C

Assorted services are performed by this object. If the implementation were larger, groups would eventually have split off into more well-defined places.

Kernel Object:

TempServices

Operation:

AddWellknown(IN CAPA instance, IN CAPA wellknown)

The ‘wellknown’ capa is bound into the C-list of the ‘instance’..

Operation:

MakeReady(IN CAPA thread)

A newly created thread is unblocked and made runnable.

Operation:

SetThreadImportance(IN CAPA thread, IN u_long new_importance,
OUT u_long old_importance)

The thread (which should be runnable) is given a new importance.

Operation:

SchedStatsOn()

Turns on the scheduling statistics gathering.

Operation:

SchedStatsOff()

The scheduler will stop recording statistics.

3.1.3.9 Clocking Object — TimeServices.C

This object offers time-related services to user threads.

Kernel Object:

TimeServices

Operation:

GetTime(OUT u_long timeval)

This operation reads the system clock, and returns an unsigned integer number which is the current time (in units of 200 nanoseconds).

Operation:

Sleep(IN u_long delay)

This operation suspends the execution of the invoking thread for at least ‘delay’ milliseconds.

3.1.3.10 Thread Creation Object — ThreadManager.C

Kernel Object:

ThreadManager

Operation:

Create(IN CAPA instance, IN u_long operation, OUT CAPA thread)

Create a new thread. It comes into existence blocked.

Operation:

```
Delete(IN CAPA thread)
```

The indicated thread is removed from the system.

3.1.3.11 Alphabit ID Management — abitid.c

This C source file contains logic for Alphabit ID management.

Routine:

```
void    ID_Init()
```

Initialize the Alphabit ID Generator

This routine lears the volatile counter and the non-volatile counter.

Routine:

```
void    ID_Gen(idptr, type)
        abidptr     idptr;
        u_long      type;
```

Generate a new Alphabit ID and put in '*idptr'. Use the passed type information for the type field, read the unique node id (from the parallel port), and get the global counters.

Routine:

```
void    ID_Roll()
```

Increment the 'instance' part of the Alphabit ID Generator. (Increment the non-volatile counter and clear the volatile one.)

3.1.3.12 Thread Abort — abort.c

This C source file contains logic for Thread Abort.

Routine:

```
void    ThreadAbort(KTptr, abortlevel)
        KTCBptr     KTptr;
        u_short     abortlevel;
```

Abort a thread to a given level.

This routine is responsible for aborting the specified thread to the given level. This is done by jamming the thread's initial kernel stack frame with the address of either the 'DoAbort()' routine (in the kernel), or the address of the return point in the invocation code.

This routine begins by adjusting the abort level to its proper value (i.e., the lowest abort level). The 'aborttolevel' variable is set in such a way as to ensure that the most extreme abort is performed if multiple aborts were issued before the thread gets to its abort code (i.e., with a MIN function).

Then the code checks if there is already an abort pending/in progress, and if there is, it just returns.

If there is currently no pending abort, the target thread's KTCB is marked to indicate that there is an abort pending on it by setting the 'abortpending' flag 'TRUE'. Then, a pointer to the initial kernel entry stack frame is obtained (either from a blocked thread's KTCB or from the global variable 'kentrystkptr' in case the thread to be aborted is the current thread). Then the thread's initial kernel stack frame is jammed with a pointer to the 'DoAbort()' routine.

This code acts as an "rte extender", i.e., it puts the 'DoAbort()' code in-line with the target thread's return to its object, regardless of what state the thread is in at the time. When the jammed frame is rte'd, the 'DoAbort()' routine is executed in supervisor mode and with interrupts disabled.

Note that the scheduling parameters are adjusted by the scheduling processor if necessary as part of the abort (this means that the thread will execute under the next level up's scheduling parameters following the execution of this routine).

Routine:

void DoAbort()

Abort a thread to its next abort block level.

This routine actually performs the aborting of a thread to the next abort block level in an object. This routine is entered via an 'rte' from a jammed stack (that was done by the 'ThreadAbort()' routine), from a thread with abort pending, or it is called at the end of the 'EndAbort()' routine. In either case, it executes in supervisor mode, and at max priority.

This code is entered right before a thread is to return to the object it is currently executing in. First a check is made to see if there is an abort block in the current object, and if there is the address of the saved abort block registers is obtained, the saved registers are restored, the kernel stack is modified, the return value is set to be 'TRUE', and the code does an 'rte' back to the current object.

Note that the thread will think it is back at the same point where it did its block begin trap from, but this time the trap routine returns a TRUE (last time it was a FALSE).

If there is no abort block in the object currently being executed, then the invocation is rolled back to the object that invoked this object. This is done by restoring the system stack pointer and by jumping back to either the client or kernel object invocation code. Not only is this routine entered in an unusual way (i.e., via an 'rte' from a dummied trap frame), but it also leaves in a funny way (i.e., via another 'rte' off another dummied trap frame or a jump). Despite the fact that the kernel stack is used in a non-standard way, everything should work out all right and there should not be any inconsistencies.

Routine:

boolean BeginGAB(saveregs)

Mark the start of a Generic Abort Block.

This routine marks the beginning of a generic abort block and enters a new abort block level.

The code checks for pending aborts and if there is one it returns a 'TRUE', else it pushes an abort block level (i.e., increments the abort level and saves the registers), and returns a 'FALSE'.

The trap handler that invokes this routine passes the necessary register block (with the kernel stack pointer adjusted to take its own trap frame into account).

This routine is typically called by the 'BeginGABTrap' trap handler.

All registers are saved except the scratch registers (i.e., A0, A1, D0, D1) — including the PC, SR, SSP, and USP.

Note that this routine executes on the thread's kernel stack, and therefore it can be preempted at any time.

Routine:

```
void    EndGAB()
```

Mark the end of an Generic Abort Block.

This routine decrements the abort block depth count (to go back to the previous abort register set).

This routine is typically called by the ’EndGABTrap’ trap handler.

Note that this does not have to be made atomic with respect to interrupts or preemption.

Routine:

```
boolean        BeginDAB(saveregs, SIBptr)
        ablkregset    saveregs;
        TCSPIBptr     SIBptr;
```

Mark the start of a Deadline Abort Block.

This routine marks the beginning of a deadline, changes the scheduling parameters for the thread, and enters a new abort block level. The code pushes an abort block level (i.e., increments the abort level and saves the registers), sends an IPI command block to change the thread’s scheduling parameters, and returns a ’FALSE’.

The trap handler that invokes this routine passes the necessary register block (with the kernel stack pointer adjusted to take its own trap frame into account), along with the deadline type, and a pointer to the scheduling parameter block.

The command is sent to the scheduling processor regardless of the values of the scheduling parameters. (Note that if a tighter time constraint is currently in effect, the SP treats the command as a NOP.)

If OLDSCHED is defined, the block is treated as a generic abort block.

This routine is typically called by the ’BeginDABTrap’ trap handler. All registers are saved except the scratch registers (i.e., A0, A1, D0, D1) — including the PC, SR, SSP, and USP. Note that this routine executes on the thread’s kernel stack, and therefore it can be preempted at any time.

Routine:

```
void    EndDAB()
```

Mark the end of a Deadline Abort Block.

This routine decrements the abort block depth count (to back to the previous abort register set) and sends an IPI command to the scheduling processor to pop the scheduling parameters for this thread.

If OLDSCHED is defined, the block is treated as a generic abort block.

This routine is typically called by the ’EndDABTrap’ trap handler.

Note that this does not have to be made atomic with respect to interrupts or preemption.

Routine:

```
void    EndAbort()
```

Mark the end of the execution of an abort code block.

This routine marks the end of the execution of a block of abort code (that was initiated by ’DoAbort()’ as a result of a call to ’ThreadAbort()’) by decrementing the abort block depth count. If there are still further levels of aborts to go, the 'DoAbort()' routine is jammed on the stack and an abort is done to another level when this routine returns to the object. In this way, aborts continue until the desired level is reached.

If this marks the end of the last level of aborts to be done, this is indicated in the 'abortpending' and 'aborttolevel' fields of its KTCB.

This routine is called by the 'EndAbortTrap' trap handler.

Note that this does not have to be made atomic with respect to interrupts or preemption.

Routine:

```
void    StdAbort()
```

This is the Standard (System-Provided) Abort Routine.

This is the "on-abort" routine provided by the system if the user doesn't specify one. This code performs the appropriate rollback operations on semaphores that belong to the given abort level.

3.1.3.13 Interprocessor Interface for AP — apipi.c

This C source file contains logic for the Applications Processor to handle requests from the IPI.

Routine:

```
void    IPIrequest()
```

Common routine to handle requests coming via the Interprocessor interface. Since only one interrupt line (level 4) is used by all CPs to get the AP's attention, polling of each channel for requests are carried out here.

Routine:

```
void    CPrequest(opcode, toknptr)
        u_short     opcode;
        tokenptr    toknptr;
```

This routine handles IPI commands from the Communications Processor.

For each command, an entry is made in the 'cpqueue', which is shared with the 'CommProcIntf' thread. After this entry is added to the queue, the 'CommProcIntf' thread is signalled by means of a global semaphore belonging to the 'CommProcIntf' object.

Special provision has been made to make certain that the 'CommProcIntf' thread exists and that there is an available slot in the 'cpqueue'. If either of these conditions is not met, the IPI channel is cleared and the command is discarded. A notification is made whenever a command is discarded after the 'CommProcIntf' thread has been started.

Some IPI commands are long enough that they cannot be placed in the 'cpqueue' in their entirety. These are translated into a form that can be placed into the queue, typically by copying some fields to a buffer outside the queue and then including a pointer to this buffer in the queue. (The maximum size command that can be placed in the 'cpqueue' is 'sizeof(token)'.)

Routine:

```
void    SPrequest(opcode)
        u_short     opcode;
```

Service a Scheduling Processor IPI request.

This routine handles commands from the scheduling processor. It is given the command's opcode and a pointer to the scheduling information parameter block for the command. The commands recognized by this handler from the SP are:

SID — return an identifier from the scheduler for a thread that has been registered with the scheduler. This ID is used to make it easier for the scheduler to find thread control blocks.

PRMT — preempt the currently running thread and start running the one indicated in the command. The 'preemptthread' global variable is set to point to the KTCB of the new

thread to be run, and then a check is made to see if preemption is currently deferred. If it is, a preempt attempt is signalled, and if it is not deferred a real preempt is done by calling the 'Preempt()' routine.

ABRT — abort a thread because of a deadline expiration. The indicated thread is aborted to the indicated abort level. The 'ThreadAbort()' routine sets things up so that the thread will start running abort code the next time it gets dispatched. Next it is determined if the deadline originated on this node or came from another. If it is a local deadline abort, it must further be determined if the thread is blocked or not, and if it is, whether it is

blocked on a remote invocation. In the case of a local deadline where the thread is blocked on a remote invocation, the thread's new scheduling parameters must be given to the CP for distribution to the orphaned part of the thread. This is done by sending a "request scheduling parameter dump" command to the SP, which will then respond by sending the desired scheduling information in a reply command. The thread is not unblocked at this point in time, but rather it waits until the CP indicates that thread repair has been done and unblocks the thread in the usual way. If the thread is blocked on something other than a remote invocation, it must be unblocked to carry out the abort sequence and nothing more needs to be done. When the abort is due to the expiration of a deadline established on a remote node, and the affected thread is blocked, this code unblocks it so that it may begin executing its abort code. The aborted thread runs without time constraints until the CP provides the SP with a new set of timing constraints with a "push abort timing constraints" command. Threads that are aborted with local deadlines run under the next most constraining deadline, while threads that are aborted due to remote deadlines run without any time constraints until the CP provides it with new scheduling parameters (that it gets from the node that issues this command). Threads are aborted properly regardless of the order in which the CP and SP signal the abortion (note that 'ThreadAbort()' only sets up the abort to be done the next time the thread gets to run, and it is idempotent).

DUMP — provide the current scheduling parameters for an aborting thread. This command receives the scheduling interface parameters from the SP for an aborting thread in response to a scheduling parameter dump request, and sends them in a "begin abort" command to the CP.

UPDATE/DUMP NOTIFICATION — notify the AP that the UPDATE or DUMP request had been done. This handshake is part of the remote invocation mechanism where requested scheduling information is dumped directly into the invocation parmblk to be propagated. Similarly, scheduling info changes propagated back on return from remote invocation are also updated by accessing those changes directly from the invocation parmblk.

The SP is somewhat of a special case since there are cases where it does not return to do an 'rte'. There are also cases where further interaction between the SP and AP is carried out. Due to its special nature, it is required that a 'SynchGetCB()' or an 'AsynchGetCB()' must be called as the first thing in each leg of the outermost case statement (i.e., once you know what the command is, hence its size). The opcode can be discarded following the GetCB command because it was already obtained by the Preview command.

3.1.3.14 Client Object Management — clientobj.c

This C source file contains logic for Client Object management.

Routine:

```
void    InitCOSP()
```

Initialize the COSP.

Routine:

```
COCBptr CreateCO(COTptr)
        COTCBptr    COTptr;
```

Create a new Client Object from a CO skeleton.

This routine takes a pointer to the COTCB of the client object type to be instantiated. From this COTCB, it extracts the abid of a file for the client object type to be created, gets a CO skeleton from the pool, and turns the skeleton into a client object by (1) associating the file's FCBid with each of the client object's extent descriptors, (2) initializing the pmap entries for each of the CO extents, and (3) generating an abid for the client object and entering it in the dictionary.

A pointer to the client object's COCB is returned by this routine.

A skeleton pool is used to make the CreateCO operation fast. This is an important consideration since this routine is executed each time a kernel object invokes an operation on a client object (which occurs, for example, whenever a remote invocation arrives).

Routine:

```
void    DeleteCO(COptr)
```

Delete a Client Object .

This routine removes the client object's entry from the dictionary and returns the CO skeleton to the COS pool.

Routine:

```
COCBptr     GetCO()
```

Get a client object skeleton from the COSP.

Get a CO skeleton. Attempt to get it from the pool first; otherwise, allocate a new one. Check the water marks and signal the pool manager thread if the pool is outside of the water mark limits.

Routine:

```
void    PutCO(COptr)
        COCBptr     COptr;
```

Return a client object skeleton to the COSP. (Put a CO skeleton back in the pool.)

Check the water marks and signal the pool manager thread if the pool is outside of the water mark limits.

Routine:

```
COCBptr AllocCO()
```

Create a CO skeleton (i.e., allocate and connect the major data structures for a client object).

Allocate the major data structures for a client object. This includes allocating a COCB, the extent descriptors for the CO's extents, and one 'pmap' for each extent. All of these structures are linked together and initialized as appropriate.

A pointer to the skeleton's COCB is returned to the caller.

Routine:

```
void    DeallocCO(COptr)
        COCBptr     COptr;
```

Routine to deallocate a Client Object Skeleton.

Return the skeleton to its elemental form (by deallocating its pmaps, its extent descriptors, and its COCB). This is only done when the pool exceeds its high water mark.

3.1.3.15 Client Thread Management — clientthread.c

This C source file contains logic for Client Thread management.

Routine:

```
void    InitCTSP()
```

Initialize the skeleton pool.

Routine:

```
CTCBptr     CreateCT(FID, KTptr)
        abid        FID;
        KTCBptr     KTptr;
```

Create a new Client Thread.

Given the ID of an instance file ('FID'), and a pointer to a KT for the CT, this routine will get a CT skeleton from the CT skeleton pool, will associate the CT and the KT with each other, and will fill in various fields of the CT (including the client stack pointer and the instance file field of the CTs extents).

An SMC slot (a context) will be allocated and loaded for the CT.

A PMC slot is allocated and loaded for the CTs first client stack extent .

The CT created by this routine will get it's initial incoming parmblk from the KT it is attached to, and the outgoing parmblk should be a scratch page.

An abid is generated for the CT and entered in the dictionary.

A pointer to the CTCB of the created client thread is returned to the caller.

Routine:

```
void    DeleteCT(CTptr)
        CTCBptr     CTptr;
```

Delete a Client Thread.

This routine deletes the CT's abid from the dictionary, clears the fields in the CTCB that were originally set up by CreateCT, and returns the CT skeleton to the CTS pool.

Note that this is (and must be) always called from the target CT's context.

Routine:

```
CTCBptr     GetCT()
```

Get a CT Skeleton from the CT Skeleton Pool.

This routine removes a CT skeleton from the CT skeleton pool if one is available. If none are available, then 'AllocCT()' is called to create a new skeleton.

The CT skeleton pool manager maintains the pool at an acceptable level. The caller must only wait for a CT skeleton to be constructed in the (hopefully rare) case that the pool manager is not scheduled to do its job in time.

Routine:

```
void   PutCT(CTptr)
       CTCBptr     CTptr;
```

Return a CT skeleton to the CT skeleton pool.

Check if the CTS pool is full, and if so deallocate the CT skeleton, else put the CTS into the pool.

Routine:

```
CTCBptr     AllocCT()
```

Allocate a client thread skeleton (i.e., allocate and connect the major data structures for a CT).

Allocate the major data structures that must be added to a KT in order to produce a CT. This includes allocating a CTCB, the extent descriptors for the CT's extents, and the 'pmap's for each extent. All of these structures are linked together and initialized as appropriate.

The pmaps for the client stack extent are set up as part of this initialization.

A physical page is allocated and initialized for the client stack.

Routine:

```
void   DeallocCT(CTptr)
       CTCBptr     CTptr;
```

Routine to deallocate a client thread skeleton.

Deallocate the major data structures (CTCB, extent descriptors, and pmaps) that are added to the KT to produce the CT. Also, deallocate the physical page that is used for the client stack.

3.1.3.16 Kernel Debugging Aids — debug.c

This C source file contains logic for the kernel debugging aids.

Routine:

```
void   KDBug()
```

Main kernel debugging routine.

This routine provides a common user interface to a collection of debugging routines.

The main debugging aid provided by this routine involves the printing of various data structures. This routine also provides access to the routine that prints out the events in the log area.

The interface is menu-driven and an attempt is made to make the printing be "screen-oriented". At first the page size is given and a "more"-like function is provided: it prints a screen, prompts and waits for a response. Currently legal responses are: 'q' to quit printing, 'CR' for one more line, and any other character to print another full screen. Also, an 's' stops the paging (streaming the output), and any character typed at this point puts it back into paged mode.

3.1.3.17 Dictionary Access — dict.c

This C source file contains logic for Dictionary access.

Routine:

```
void    InitDict()
```

Initialize the Dictionary.

Routine:

```
abheadptr    LookupInDict(id)
        abid    id;
```

Look up an entry in the Dictionary.

This routine searches the dictionary for the control block associated with the ABID 'id'. If such an object is found, this routine returns a pointer to the control block, otherwise a NULL is returned.

Serializability of access to the dictionary is not guaranteed, and a return value of NULL just means we couldn't find it right now.

Routine:

```
void    EnterInDict(abhptr)
        abheadptr    abhptr;
```

Add an entry into the Dictionary.

Add the control block pointed to by 'abhptr' to the dictionary. The hash used to generate the dictionary index is based on 'abhptr->id'.

Routine:

```
void    RemoveFromDict(abhptr)
        abheadptr    abhptr;
```

Remove an entry from the dictionary.

Remove the control block pointed to by 'abhptr' from the dictionary. It is a fatal error if the requested control block is not found in the dictionary.

A well-known abid is treated as a pseudonym of the actual abid to which it corresponds; therefore, when an actual alphabit or any of its WK abid's is removed from the 'Dictionary,' all of its 'Dictionary' entries are removed.

Routine:

```
void    RemoveDictEntry(abitid)
        abid    abitid;
```

Remove an entry from the dictionary.

Remove the control block for alphabit identifier 'abitid' from the dictionary.

It is a fatal error if the requested control block is not found in the dictionary.

3.1.3.18 Interprocessor Memory Management — dvma.c

This C source file contains logic for managing DVMA space from the AP.

Routine:

```
void    InitDVMA()
```

Initialize DVMA region.

Allocate PMs at the start of FIRST_DVMA_SEG to cover the whole 256K DVMA space.

Routine:

```
vaddr         MapPageIntoDVMA(pgaddr)
        vaddr         pgaddr;
```

Map a page into the DVMA space.

Map the PD found in the specified page address into DVMA space.

The address in DVMA space where this PD is mapped will be returned.

Routine:

```
void    UnMapPageFromDVMA(pgaddr)
        vaddr         pgaddr;
```

Unmap the PD from DVMA space specified by 'dvmapgvaddr'.

3.1.3.19 Event Logging — event.c

This C source file contains logic for event logging.

Routine:

```
void    InitEvent()
```

Initialize the event logging area, initialize the event timer, and log these actions as the first event.

It is assumed that the event timer is initialized elsewhere.

The event buffer logging area is in low core between the monitor stack and the start of kernel code.

Routine:

```
int     LogEvent(id, parm)
        u_char        id;
        u_char        parm;
```

This routine is used to log an event. It uses the 'eventcount' global variable as an index into the event log buffer, it gets the current value of the event timer and puts it and the given parameter into the event record indexed by 'eventcount'.

This routine is called by the 'LogEventIn()' trap handler. Interrupts are disabled by the trap handler. Interrupts must be disabled during this routine because we have to maintain the ordering of the events in the log (an interrupt could result in another event being logged).

Note that an 'int' is returned in order to make the assembly interface work out right.

3.1.3.20 File System Interface — file.c

This C source file contains logic for the file system interface.

Routine:

```
abid    CreateFCB(type)
        filetype      type;
```

Create an FCB for a (non-wired) file .

Routine:

```
void    DeleteFCB(FID)
        abid    FID;
```

Destroy an FCB for a (non-wired) file.

Routine:

```
abid    CreateWFCB(type, memext)
        filetype      type;
        memextent     memext[];
```

Create an FCB for a Wired File.

Routine:

```
void    DeleteWFCB(FID)
        abid    FID;
```

Destroy an WFCB for a wired file.

Routine:

```
void    InvokeFCB(fptr, pbp)
        FCBptr        fptr;
        parmblkptr    pbp;
```

Invoke an Operation on a File Object.

This routine handles all operations on "file objects." The request portion of the parmblk is examined to determine which operation is being invoked on the "file object" and makes a call to the appropriate routine to handle the requested operation.

Routine:

```
void    InstallExtent(fptr, pbp)
        FCBptr        fptr;
        parmblkptr    pbp;
```

Install an extent from an FCB into a set of contiguous pages of physical memory.

This routine receives the index of the extent which is to be installed in memory. This index is validated, and the pages of that extent are placed into contiguous pages of memory (for now).

The invoker receives the page index of the first physical page in the installed extent, as well as the total number of pages in the extent.

3.1.3.21 Kernel Global Variables — globals.c

This C source file contains global variables that weren't logically associated with some other file.

Variables:

```
timeofday
```

BV
inkernel
kentrystkptr
intstackdepth
tsduringint
storedsp
storedctx

3.1.3.22 Kernel Dynamic Memory Allocation — heap.c

This C source file contains logic for Kernel Heap Storage.

Routine:

```
void    InitHeap()
```

Initialize the kernel heap. Allocate PP's for the initial control and data segments and put them in the heap extent of the KernCB. Set up all the needed control structures and initialize them.

Get the initial 'pmap's from the global data segment, when all done, use 'alloc()' to get space for the 'pmap' for the next expansion segment.

For now, we alloc real PP's, later we could choose to get 'SCRATCH' pages.

At the end, allocate a 'pmap's worth of space for the next expansion segment, and put it in the static pointer 'freepmptr'.

Routine:

```
vaddr           alloc(size)
        u_int   size;
```

Allocate "size" bytes of memory from the kernel heap.

Routine:

```
void    dealloc(addr)
        vaddr           addr;
```

This routine returns a block of memory to the kernel heap. It checks the address of the block being returned, and if it is reasonable, it returns the space to the heap. All possible merge operations to combine buddies in order to form larger blocks of free space are performed when the new space is added into the free table.

3.1.3.23 Spurious Interrupts Handler — intrhdlr.c

This C source file contains logic for handling spurious interrupts.

Routine:

```
void    IntrHdlr(vector)
        u_int   vector;
```

This routine handles unexpected interrupts. It takes as a parameter the interrupt vector number, prints a message on the console indicating the level of the interrupt that was caught, and returns.

3.1.3.24 Interprocessor Interface — ipi.c

This C source file contains logic for the Interprocessor Interface (for both SUN 2.0 and SUN 1.5 Boards).

Routine:

```
void    InitAPI()
```

This routine initializes the interprocessor interface for an application processor.

This is done by mapping the interrupt generator into on-board I/O space to the first page of the first multibus P1 segment to be accessible via DVMA, and also to the first page of the AP's IPI base segment. Then memory is allocated for the IPI control blocks and its channels, starting from the second page of the AP's IPI base segment.

Also, the AP's IPI control block is set up at the start of the AP's IPI base segment and all of its channel descriptors are initialized. Note that the allocation of shared MB memory space must be contiguous to ensure APs and CPs have identical mappings. Also it should be noted that 'InitVM()' must be called prior to this routine, since virtual memory management routines are used here. Furthermore, this code assumes that there is shared MB memory available, in sufficient quantity to meet the needs of IPI.

Routine:

```
void    SendCB(opcode, CBptr, CBsize)
        u_short     opcode;
        vaddr       CBptr;
        u_short     CBsize;
```

Send a command block, pointed to by 'CBptr', via the channel defined by 'opcode'. Note that all information about what channel to send, if the transfer should be synchronous or asynchronous, and which interrupt line to yank is encoded in the 'opcode' parameter.

Routine:

```
void    SynchSendCB(opcode, CBptr, CBsize)
        u_short     opcode;
        vaddr       CBptr;
        u_short     CBsize;
```

Send a command block, pointed to by 'CBptr', via the channel defined by 'opcode' and wait for the command block to be processed before returning.

Note that all information about what channel to send, if the transfer should be synchronous or asynchronous, and which interrupt line to yank is encoded in the 'opcode' parameter.

WARNING: Think carefully before using a synchronous send. There is serious risk of deadlock; this routine should probably never be used.

Routine:

```
u_short SynchGetCB(chantype, CBptr, CBsize)
        u_short     chantype;
        vaddr       CBptr;
        u_short     CBsize;
```

Receive a command/reply block via a channel specified by 'chantype'.

This is a synchronous operation which is called right after the 'SendCB()' routine. The command/reply block from the IN stream of the channel in question is then copied into 'CBptr'.

Only 'CBsize' bytes will be copied.

It is up to the AP and CP software to decide on a common structure for the command and reply blocks.

Routine:

```
u_short AsynchGetCB(chantype, CBptr, CBsize)
        u_short     chantype;
        vaddr       CBptr;
        u_short     CBsize;
```

Receive a command/reply block via a channel determined by the 'chantype'. This is an asynchronous operation.

The command/reply block from the IN stream of the channel in question is then copied into 'CBptr'. Note that 'CBsize' bytes will be copied.

It is up to the AP and CP software to decide on a common structure for the command and reply blocks. The opcode in the command/reply extracted will be returned.

A null opcode will be returned if no command block is present.

Clears the interrupt: it is due to the fact that this interrupt line is yanked by all CPs to get AP's attention.

Routine:

```
u_short PreviewCB(chntype, CBptr)
        u_short     chntype;
        vaddr       *CBptr;
```

Called by the APs and CPs.

Allows for the previewing the next command block by returning the opcode sent with the command block along with a pointer to the command block (returned in the parameter 'CBptr').

Note that the command block in the channel is not copied out and cleared in this operation. A call to 'AsynchGetCB()' will be required to clear the channel.

Routine:

```
void    ChanAttn(opcode)
        u_short     opcode;
```

Generate an interrupt to get the attention of the processor associated with 'chntype'.

For CPs, a real interrupt line will always be asserted by way of the interprocessor interrupt generator hardware.

For VCPs, a software interrupt (i.e., a trap) will be generated.

The interrupt mask field in the opcode is examined to determine whether a *real* interrupt line or a software trap is to be generated. If it is non-zero, the interrupt mask is used to set the interrupt generator, else a software trap corresponding to the value of the 'chntype' parameter is used.

There is a performance tradeoff as compared to a simple conditional compilation for CP/VCP, but this tradeoff allows us more flexibility in configuring system that has a mixture of CP and VCPs thru usage of different opcode sets.

Routine:

```
void    ReadChanLog(chantype, Lbufptr, Lbufsize)
        u_short     chantype;
        vaddr       Lbufptr;
        u_short     Lbufsize;
```

Read the log area from the channel specified by ’chantype’ into a buffer pointed to by ’Lbufptr’.

Note that the first byte of the channel log area is used as a synchronization semaphore operated by a ’test and set’ primitive.

Routine:

```
void    WriteChanLog(chantype, Lbufptr, Lbufsize)
        u_short     chantype;
        vaddr       Lbufptr;
        u_short     Lbufsize;
```

Write the contents of the buffer pointed by ’Lbufptr’ into the log area of the channel specified by ’chantype’.

Note that the first byte of the channel log area is used as a synchronization semaphore operated by a ’test and set’ primitive.

3.1.3.25 Kernel Object Manipulation — kernobj.c

This C source file contains logic for Kernel Object manipulation.

Routine:

```
void    InitKOSP()
```

Initialize the KO skeleton pool. For now, the pool is initially empty, so all skeletons will be allocated dynamically on demand.

Routine:

```
KOCBptr CreateKO(KOTptr)
        KOTCBptr    KOTptr;
```

Build a kernel object instance of a given kernel object type, enter it in the Dictionary, and return a pointer to the KO structure created.

Routine:

```
void    DeleteKO(KOptr)
        KOCBptr     KOptr;
```

Delete a kernel object. This includes removing the KO’s entry (as well as those for each of its well-known abids) from the Dictionary and returning the KO skeleton to the KOS pool.

Routine:

```
KOCBptr GetKO()
```

This routine removes a KO skeleton from the KO skeleton pool if one is available. If none are available, then 'AllocKO()' is called to create a new skeleton.

The KO skeleton pool manager should maintain the pool at an acceptable level. You only wait for a KO skeleton to be constructed in the (hopefully rare) case that the pool manager is not scheduled to do its job in time.

Routine:

```
void    PutKO(KOptr)
        KOCBptr     KOptr;
```

Check if the KOS pool is full, and if so deallocate it, else put the KO into the pool.

Routine:

```
KOCBptr     AllocKO()
```

Allocate the major data structures necessary to create a KO, fill in most of the fields, get an ID for it, and enter it in the dictionary.

Routine:

```
void    DeallocKO(KOptr)
        KOCBptr     KOptr;
```

Deallocate the KOCB associated with the KO skeleton pointed at by 'KOptr.'

3.1.3.26 Kernel Thread Management — kernthread.c

This C source file contains logic for Kernel Thread Management.

Routine:

```
void    InitKTSP()
```

Initialize the KT skeleton pool. For now, the pool is initially empty, so all skeletons will be allocated dynamically on demand.

Routine:

```
KTCBptr CreateKT(KOptr, entrypt, ipbp, opbp)
        KOCBptr     KOptr;
        u_short     entrypt;
        parmblkptr  ipbp;
        parmblkptr  opbp;
```

Create a new KT from a pointer to the KT's initial KO (i.e., 'KOptr'), the index of the initial entry point within the KO, and an incoming and outgoing parmblk. This routine returns a pointer to a complete KT. All that remains is to make the thread ready to run (via a call to 'MakeReady') and then for 'LoadThread()' to do an 'rte' to start

the execution of the KT at the indicated entry point within the indicated object, with the given incoming and outgoing parmblks.

This routine gets a KT skeleton from the KTSP, get an ID for it, enters it in the dictionary, and then sets up and initializes the stack. To initialize the stack, this routine must derive the address of the entry point within the initial object.

Note that no error checking is done on the parameters. (This is a kernel function and we had better get things right ourselves.)

This routine assigns the ID of the KTCB to the threads 'threadid' field. If this is supposed to be a surrogate thread, this must be overwritten with the remote thread's threadid.

Routine:

```
void    DeleteKT(KTptr)
        KTCBptr     KTptr;
```

Delete a Kernel Thread by removing it from the dictionary and returning its skeleton to the KTSP.

Routine:

```
KTCBptr GetKT()
```

This routine removes a KT skeleton from the KT skeleton pool if one is available. If none are available, then 'AllocKT()' is called to create a new skeleton.

The KT skeleton pool manager should maintain the pool at an acceptable level. You only wait for a KT skeleton to be constructed in the (hopefully rare) case that the pool manager is not scheduled to do its job in time.

Routine:

```
void    PutKT(KTptr)
        KTCBptr     KTptr;
```

Return a KT skeleton to the KT skeleton pool.

Check if the KTS pool is full, and if so deallocate it, else put the KT into the pool.

Routine:

```
KTCBptr     AllocKT()
```

Allocate the major data structures necessary to create a KT and fill in most of the fields. Also, a real page of memory is allocated for the KTs kernel stack page.

Routine:

```
void    DeallocKT(KTptr)
        KTCBptr     KTptr;
```

Deallocate and disconnect the major data structures for a kernel thread skeleton. This includes returning the kernel stack page to the system.

3.1.3.27 Multibus Page Virtual Memory — mbmapping.c

This C source file contains logic for Multibus Page Virtual Memory Mapping.

Routine:

```
void    InitMBPageMap()
```

Initialize the data structures for mapping Multibus memory pages.

Incoming remote invocation parmblks are placed in Multibus shared memory by the Communications Coprocessor; these parmblks are accessed in place (in Multibus memory) by threads; other information and parameters for the kernel may also arrive from the Communications Coprocessor and reside in shared Multibus memory; the 'MBmap[]' handles the mapping of these parmblks and any other Multibus page into the kernel's VAS; in order to avoid statically allocating enough page maps to map all of the Multibus shared memory at once, each Multibus shared memory page is only mapped when it is actually in use; it is later released when it is no longer needed.

The 'MBmap[]' is an array which could potentially map all of the shared Multibus memory at once; each 'MBmap[]' slot that is active contains an index for a page map and a bit vector indicating which slots in the page map are currently mapped; in addition there is another field which serves to link the array slots together; this field is used to construct two lists — one contains 'MBmap[]' slots that have a page map allocated (this list is pointed to by 'firstusedMBMslot') and the other contains the 'MBmap[]' slots that don't have a page map associated with them (this list is pointed to by 'firstunusedMBMslot').

This routine initializes the data structures used to perform the mapping of Multibus memory pages into the kernel's VAS; it initializes the 'MBmap[]' and creates the two lists of 'MBmap[]' slots; initially, there is a single 'used' 'MBmap[]' slot; all the others are unused; the 'used' 'MBmap[]' slot is allocated a page map and this page map is initialized.

Routine:

```
vaddr  MapMBPage(smmoffset)
       u_long smmoffset;
```

Map a shared Multibus memory page into the kernel's VAS.

This routine is passed a Multibus address (physical Multibus address); it finds a location in the kernel's VAS to map the page located at that Multibus address, carries out the required mapping, and returns the address (in the kernel's VAS) of the page.

In order to find a spot in the kernel's VAS for the page, this routine first searches the list of page maps that are already in use; if any of these has a page descriptor slot that has not yet been allocated, then that slot is used to map the page; if all of the page maps that have already been used are full, then a new page map is allocated and initialized; it is entered into the 'MBmap[]' and one of its page descriptors is used to map the page into the kernel's VAS.

Routine:

```
void   UnMapMBPage(kernvaddr)
       vaddr        kernvaddr;
```

This routine receives a pointer to a page in Multibus shared memory; the pointer is actually a virtual address in the kernel's VAS; the indicated page is removed from the kernel's VAS and the 'MBmap[]' is updated accordingly.

3.1.3.28 Node Status Monitoring — nodestat.c

This C source file contains logic for Node Status Monitoring.

Routine:

```
void   InitNodeStat()
```

Initialize the Node Status Array.

Initially, the local node is marked active and all other nodes are marked inactive.

Routine:

```
void    UpdateNodeStat(nodenum, active)
        u_short      nodenum;
        boolean      active;
```

Update an Entry in the Node Status Array.

Set 'nodenum's status to be as indicated by 'active'.

Routine:

```
void    UpdateAllNodeStat(nodeinfo)
        boolean      nodeinfo[];
```

Update All Entries in the Node Status Array.

Set each nodes status as indicated.

Routine:

```
boolean      ActiveNode(nodenum)
        u_short      nodenum;
```

Return an Entry from the Node Status Array.

Return the current (believed) status of node 'nodenum'.

3.1.3.29 Client Object Management — objmgmt.c

This C source file contains logic for Client Object Management.

Routine:

```
void    ObjectMgrSrv(rqst, rply)
        rqstpblkptr  rqst;
        rplypblkptr  rply;
```

Service an invocation on the ObjectManager SSO.

Demultiplex the ObjectManager SSO operations; find out which operation is being requested and call a routine to provide it.

This routine provides the object interface for this SSO.

Routine:

```
void    CreateObject(rqst, rply)
        rqstpblkptr  rqst;
        rplypblkptr  rply;
```

Create an object instance of the specified object type and return that instance's abid; also add a capability for the newly created object to the creator's capa list.

Routine:

```
void    DeleteObject(rqst, rply)
        rqstpblkptr  rqst;
        rplypblkptr  rply;
```

Delete an object instance.

Routine:

```
void    RegisterWKABID(rqst, rply)
        rqstpblkptr   rqst;
        rplypblkptr   rply;
```

Register a well-known alphabit identifier (WKABID).

This operation takes two capas as arguments: the alphabit identifier of an existing alphabit (object, in this case) and another (well-known) alphabit identifier by which this alphabit should be known. The existing alphabit is assumed to already in the local 'Dictionary,' and a new entry is made in the local 'Dictionary' for the well-known abid. The new 'Dictionary' entry points to the same alphabit control block as the previously existing alphabit identifier. (This registry facility is not used to register SSOs under their well-known names.)

Routine:

```
void    ReplicatedCreate(rqst, rply)
        rqstpblkptr   rqst;
        rplypblkptr   rply;
```

Create a replicated object instance.

Synopsis:

```
OPERATION ReplicatedCreate(
        IN CAPA objectType,                    rqstcapas[0]
        IN u_long replicationFactor,           rqst->parms[0]
        OUT CAPA replicatedObject,             rplycapas[0]
        OUT u_long actualReplicationFactor)    rply->parms[0]
```

Create a replicated instance of type 'objectType' that has 'replicationFactor' replicas (including the primary copy). 'replicatedObject' is a capa for the primary copy (which doesn't change, even when a new primary copy takes over).

For various reasons, we may have to create fewer replicas than the user specified, so 'actualReplicationFactor' is returned. For implementation reasons, this may be inaccurate (i.e., it may indicate fewer than the actual number of replicas, but never more).

Routine:

```
void    CreateBackup(rqst, rply)
        rqstpblkptr   rqst;
        rplypblkptr   rply;
```

Create a backup copy for a replicated object instance.

Synopsis:

```
OPERATION CreateBackup(
        IN CAPA objectType,            rqstcapas[0]
        IN CAPA objectPrimaryCopy,     rqstcapas[1]
        IN CAPA backupCopy,            rqstcapas[2]
        IN u_long numBackups)          rqst->parms[0]
```

This operation is invoked by the system to create the initial backups for a replicated object, or to replace a failed backup. There is only one backup for a given replicated object on a particular node. The node on which the backup resides is encoded in its alphabit id, and we take great advantage of this fact.

Routine:

```
void    SelectBackup(rqst, rply)
        rqstpblkptr    rqst;
        rplypblkptr    rply;
```

Coronate a backup copy as the successor primary copy.

Synopsis:

```
OPERATION SelectBackup(
        IN CAPA replicatedObject)       rqstcapas[0]
```

This operation is invoked when the primary copy of a replicated object can't be found by an invoke. It selects a backup copy to promote to primary copy status. The canonical algorithm for doing this assumes only knowledge of the primary copy's alphabit id.

Routine:

```
void    ActivateBackup(rqst, rply)
        rqstpblkptr    rqst;
        rplypblkptr    rply;
```

Activate a backup copy of a replicated object to act as primary copy.

Synopsis:

```
OPERATION ActivateBackup(
        IN CAPA BackupCopy);            rqstcapas[0]
```

This routine is actually called as a result of the receipt of an ROsync packet from the communications processor. At this time, the local (backup) copy of a replicated object should be transformed into the active primary copy for the indicated replicated object.

For the time being, the node's index number is encoded into the instance id of each copy. However, the primary copy's id is kept constant throughout the lifetime of the replicated object.

Routine:

```
void    MonitorBackup(rqst, rply)
        rqstpblkptr    rqst;
        rplypblkptr    rply;
```

Keep tabs on a backup, and try to create a replacement if it dies.

Synopsis:

```
OPERATION MonitorBackup(
        IN CAPA replicatedObject,       rqstcapas[0]
        IN CAPA backupCopy)             rqstcapas[1]
```

For each backup copy of a replicated object, a thread is created that invokes this operation to monitor the status of the backup.

Basically, this operation invokes an operation on the backup's node that just holds onto the thread (Anchor). If that operation ever returns, we know that the node died (or at least we assume that), and we assume that the backup dies iff it's node dies. Then we try to create another backup to replace the dead one.

Routine:

```
void    Anchor(rqst, rply)
        rqstpblkptr    rqst;
        rplypblkptr    rply;
```

Block on a semaphore and do nothing.

Routine:

```
short   AddCapa(abhptr, val)
        abheadptr      abhptr;
        capa           val;
```

Add a capability to an object.

Put a value (or capa) into the clist of the object specified by 'abhptr', at the first available slot in the object's clist.

Also increments the count of the number of capa's that the object has.

This routine returns the clist index for the added capa, or a 'NIL' if the clist is full.

Routine:

```
void    RemoveCapa(abhptr, cindx)
        abheadptr      abhptr;
        u_short        cindx;
```

Remove a capability from an object.

This routine takes a pointer to an object, and an indx into its clist removes/invalidates that capa, and decrements the count of the number of capabilities that the object has.

Routine:

```
void CheckPoint(COptr)
        COCBptr COptr;
```

Do a checkpoint for a replicated object.

Routine:

```
boolean CopyCOInfo(COid, numbackups, backupid)
        abid           COid;
        u_short        numbackups;
        abidptr        backupid;
```

Copy significant info from primary replicated object copy.

Routine:

```
void    UpdateCOInfo(COptr, infoptr)
        COCBptr             COptr;
        chkptinforecptr     infoptr;
```

Update a backup copy's CO information for a replicated object.

3.1.3.30 Operation Invocation — oprinv.c

This C source file contains logic for Operation Invocation.

Routine:

```
void    ClientInvoke()
```

Client invocation routine.

A call from client context first fills in the 'objindex', 'oprindex', and 'numparms' fields of the outgoing parmblk and traps to the 'ClientTrapIn' trap handling routine in the kernel. This code sets up pointers to the request and reply parts of the caller's outgoing parmblk, fills in the reply part with default values, and begins the translation and validation of the capabilities passed in the request part of the parmblk. This involves the translation of the 'objindex' from the invocation into an alphabit identifier, and the same thing is done for any capabilities passed as parameters. This routine then goes on to fill in the 'recipientid', 'senderid', and 'threadid' fields in the outgoing parmblk. If any of the capabilities are found to be invalid, an error indication is returned to the invoking thread.

Next, the parmblk stack must be rolled around to save the caller's old incoming parmblk, make the new incoming parmblk (of the called object) be the caller's outgoing parmblk, and allocate a new (scratch) page for the called object's outgoing parmblk. Once this is done, the 'ExecInvoke()' kernel routine is called and passed a pointer to the caller's outgoing parmblk (which is also, the called object's incoming parmblk).

When the call to 'ExecInvoke' returns, the parmblk stack must be rotated back around which involves making the called object's incoming parmblk be the caller's outgoing parmblk again, deallocating the called object's outgoing parmblk, and paging back in the caller's incoming parmblk.

Also upon completion, a return is made to the 'ClientTrapIn' trap handling code, where an 'rte' must be executed to return to the calling object (in User state).

Routine:

```
void    KernelInvoke(pbp)
        parmblkptr    pbp;
```

Routine for Invoking an Operation from a KT/KO.

This routine is entered by a trap from a kernel thread/object when it makes an invocation. A pointer to the invoker's outgoing paramblk is passed into this routine by the trap handler, from the invoking kernel object.

KT's parmblk pages aren't really ever rolled out, the invocation depth limits and the limits on the number of KT's that can exist are sufficient to keep down the number of physical pages tied up in KT paramblk's.

Note that the invocation depth is incremented on entering the kernel on an invocation, and is decremented on reentering the kernel after completing an operation invocation.

Also on completion, this routine returns to the 'KernelTrapIn' trap handling code, where an 'rte' is done to return to the kernel object.

Routine:

```
void    InternalInvoke(pbp)
        parmblkptr    pbp;
```

Routine for invoking an operation from an entity within the kernel.

This routine is a stub that takes the place of the 'KernelInvoke()' and 'ClientInvoke()' routines that are normally called by objects on invokes. The point of this routine is to simulate a normal invocation (with respect to the kernel entry information needed by the abort code) for invocations that are done from within the kernel.

All that the trap handler does is the normal kernel entry procedure and then call 'ExecInvoke()'.

Currently, this is only called from the 'CThreadStub' object.

Routine:

```
void    ExecInvoke(pbp)
        parmblkptr    pbp;
```

Routine for invoking an operation from within the kernel.

This routine determines whether the invoked object is a system service object. If it is an SSO, it is determined (from the ABID) whether it is local or remote. If the destination object is not an SSO, the object is looked up in the dictionary to see if it is local. In any event, 'RemoteInvoke()' is called if the invoked object is remote, 'SystemInvoke()' is called for a local SSO, and 'LocalInvoke()' is called in the case of a local object.

This routine is either called from 'KernelInvoke()' (in the case of the invocation coming from the kernel), or from the 'ClientInvoke()' (when the invocation originates in a client object). In either case, this routine is passed a pointer (in the kernel's region of the VAS) to the outgoing parmblk of the calling object.

Routine:

```
void    LocalInvoke(abhptr, pbp)
        abheadptr     abhptr;
        parmblkptr    pbp;
```

Invoke an Operation on a Local Object.

This routine handles all of the invocations of operations on objects that exist on the local node. It is responsible for determining the specific type of alphabit being called and for carrying out the actions necessary to invoke an operation on that particular type of alphabit. There is an invocation routine for each type of alphabit that exists (these routines include 'InvokeCO()', 'InvokeKO()', 'InvokeCT()', 'InvokeKT()', and 'InvokeFCB()').

This routine is also responsible for the expansion of kernel threads to client threads (through the aquisition of a CTCB by the existing KTCB) when the called alphabit is a client object, and the calling alphabit is a kernel entity.

On the way back from a local call, this routine must remove any client threads that it added to kernel threads on the way into a local call. (The CTCB's used for this purpose are a part of a special, preallocated pool intended specially for this purpose, and are put back there when they are no longer useful).

Routine:

```
void    RemoteInvoke(pbp)
        parmblkptr    pbp;
```

Invoke an operation on a remote object.

Routine to handle remote invocation. The parameter page 'pbp' is first mapped into DVMA space. A command block containing the virtual address of the mapped parameter page is then sent to the communication co-processor. The current thread is then blocked until the remote invocation is completed.

Before the interprocessor command is issued to the CP, a check is made to see if there is an abort pending on the current thread. If there is, the command is not issued and the thread returns to the object where this invocation was done from.

Have to consider making part of this atomic to ensure that we don't get a permanently blocked and aborted thread.

Map in the page containing the paramblk into DVMA space.

Note that we are assuming that the extent of paramblk does not cross page boundary.

Routine:

```
void    SystemInvoke(pbp)
        parmblkptr  pbp;
```

Invoke an operation of a system service object.

This routine handles all (local) system calls. When the invocation mechanism detects that an invocation is destined to a system service object (i.e., the invocation is a system call), this routine is called which handles the demultiplexing of the system calls.

Routine:

```
void    InvokeCO(COptr, ipbp, opbp)
         COCBptr      COptr;
        parmblkptr    ipbp;
        parmblkptr    opbp;
```

Invoke an Operation on a Local Client Object.

This routine handles the activation of the invoked object and the deactivation of the invoking object (whenever it is necessary).

This routine is passed a pointer to the invoked object's control block (a COCB), the invoked object's incoming parmblk (i.e., the invoking object's outgoing parmblk) and the invoked object's outgoing parmblk (a newly alloc'd PB scratch page), all of which are pointers in the kernel's region of the VAS.

This code increments the thread's invocation depth counter and kills the thread if the depth is too great. It first determines if the invoker is a CO, or a KT or KO. If the invoker is a CO, it must be deactivated (i.e., mapped out of the invoker's context).

Next, the invoked CO must be activated (i.e., mapped into the invoker's context), and all of the associated control structures must be updated to indicate this change.

Following this, all passed capabilities have to be added to the invoked CO's clist and translated into local form in the incoming parmblk. At this point, the kernel can be exited and the validity of the given entry point must be checked (it should have been checked at compile time so only a bounds check is done and a fatal error is generated if this check fails).

Now interrupts must be disabled in order to ensure atomicity until the target object is entered, then a check is made to see if an abort is pending and if there is, interrupts are reenabled and the invocation is ended and a return is begun. If no abort is pending, the status register and the kernel entry stack pointer is saved on the system stack.

This routine then dummies up a stack frame, and does an 'rte' to enter the 'ClientCall' assembly stub in the invoked CO. The 'ClientCall' code is linked into all objects and is passed the target object's entry point on the kernel stack. The 'ClientCall' routine then does a call the given operation in the target client object.

When the operation is complete, the invoked CO traps back to the 'ClientTrapOut' label and the stack is cleaned up, the kernel is reentered, interrupts are reenabled, the abort variables are restored, all capabilities passed back from the invocation are translated and validated, the invoked CO is deactivated, and if the invoker was a CO, it is reactivated.

Routine:

```
void    InvokeKO(KOptr, ipbp, opbp)
        KOCBptr       KOptr;
        parmblkptr    ipbp;
        parmblkptr    opbp;
```

Invoke an operation on a local kernel object.

This routine invokes an operation on the specified KO. It starts by incrementing the invocation depth counter and then gets the starting address of the kernel object from the 'KOCB' (given by 'abhptr'), validates that the operation is within the bounds of the called object, and then gets the address of the entry point in the called object.

State is saved to be used in case of an abort, and then the object is called and passed in the given incoming and outgoing parmblks.

This routine is also responsible for translating and validating all capa's passed in parmblks (both to and from the invoked object).

Routine:

```
void    InvokeCT(CTptr, pbp)
        CTCBptr       CTptr;
        parmblkptr    pbp;
```

Invoke an Operation on a Local Client Thread.

Routine:

```
void    InvokeKT(KTptr, pbp)
        KTCBptr       KTptr;
        parmblkptr    pbp;
```

Invoke an Operation on a Local Kernel Thread.

3.1.3.31 Virtual Memory Paging — paging.c

This C source file contains logic for Virtual Memory Paging.

Routine:

```
void    InitPaging()
```

Initialize the paging system.

3.1.3.32 Parameter Block Page Manipulation — pbpaging.c

This C source file contains logic for Parameter Block Page manipulation.

Routine:

```
void    InitPBRoll()
```

Initialize the PB rolling service. Clear all of the lists and zero all of the counts.

Routine:

```
vaddr  RollOut()
```

Roll out a parameter block page.

This routine is called by 'InvokeCO()' to rotate out the parameter pages when an invocation is made. When called, this routine puts the invoking object's invocation depth into the given parameter page, and links the physical page associated with the current outgoing invocation parameter page into the 'pageout' list.

This routine also moves the old outgoing parmblk to the incoming slot, and alloc's a dummy page for the new outgoing parmblk page.

Upon completion, this routine returns the address (relative to the KVAS) of the old incoming parmblk page that was rolled out.

This routine is called in kernel context.

As an optimization, could detect small paramblk's and copy them into a page and keep compressing them into a page until the page fills up and only then page it out. This would cut down the number of times you'd have to go to disk, especially for small paramblk's.

For the time being, just unmap the page and link it into the 'pageout' list. The pager thread doesn't run yet, so we don't have to worry about it getting paged out, and the pages are not in the "free" or "inuse" list so they won't get taken away. Just leave them in the list until someone asks for them. (this is the same as a very large HWM for the pool).

This only works within the current context (i.e., the kernel context). We can't afford to do 'alloc()' each time a 'RollOut()' is done, this is done in INVOKE and so it must be efficient.

Routine:

```
void    RollIn(pageaddr)
        vaddr       pageaddr;
```

Roll a parameter page in.

This routine dealloc's the outgoing parmblk, moves the incoming parmblk into the outgoing slot, and then brings back in the last paramblk page that was paged out by the current thread and puts it in incoming slot. The address of the page to bring back in is given by the 'pageaddr' argument (and the invoking object's invocation depth).

If the specified page is 'RESIDENT', it is removed from the 'pageout' list.

This ignores the invocation depth because all pages are in order and there is no ambiguity. Currently only works for on board memory.

3.1.3.33 Kernel Page Heap — pgheap.c

This C source file contains logic for the Kernel Page Heap.

Routine:

```
void    InitPageHeap()
```

Initialize the kernel page heap structures. Load the initial SD and PD's, initialize the variables 'firstfreePH' and 'numPHsegs', and link all of the 'PHstate[]' entries togeather with the first one at the head of the list.

When 'firstfreePH' is 'NIL', the page heap is full and cannot be expanded further; this is a fatal error.

When the 'inuse' field of a PH entry is equal to 'PH_FULL', there are no more pages in that segment and another is alloc'd and wired down.

The page heap starts off with one segment (without any physical pages bound to it), allocates the physical pages when needed, and gets a new segment when it runs out of free pages in the segments it already has.

Routine:

```
vaddr   AllocVP(type)
        boolean     type;
```

Allocate a virtual page (for the kernel) from the kernel page heap.

Returns a pointer (in the kernel's region of the VAS) to a page of memory. If 'type' is 'REAL_PG', then a PP is allocated, and if the 'type' argument is 'DUMMY_PG', then a scratch page is allocated, a page fault will occur if it is referenced, and a real page will be allocated on demand.

There are four sets of segment descriptors kept; "unused", "inuse and empty", "inuse and not full", and "inuse and full". Each of these is a doubly-linked list that is rooted in the 'PHctl' structure.

The physical pages allocated are "wired" (if the caller wants to unwire them, he must do it himself).

This is used primarily to get kstk's and parmblk's for KT's.

The page heap starts off with one segment (without any physical pages bound to it), and expands as needed (provided there is room).

Now, dummy pages aren't zero'd but they will be.

Routine:

```
void    DeallocVP(addr)
        vaddr       addr;
```

Deallocate a virtual page (for the kernel) from the kernel page heap.

Takes the address of a page in the address space of the kernel's page heap, validates that it in fact is a pointer to a previously allocated page, and returns it to the virtual page heap.

It is a fatal error to try to deallocate a page that is outside of the kernel page heap, or a page that has not been allocated.

Routine:

```
vaddr           AllocKstk()
```

Allocate contiguous pages in VM for Kstk from the kernel page heap.

Returns a pointer (in the kernel's region of the VAS) to the last (highest VA) page in a newly allocated kernel stack. This routine searches the kernel heap for the first available block of 'kstkpgs_const' (a constant) contiguous, unallocated pages in a single page map. (The routine must be altered to find such pages that span page maps. Since the page maps are searched in the order in which they are grouped (the 'notfull' list, the 'empty' list, and the 'unused' list), rather than according to ascending/descending virtual address, it seems to make sense to handle it this way and not attempt to span page maps.)

The physical pages allocated are "wired" (if the caller wants to unwire them, he must do it himself).

This is used to get kstk's for KT's.

Routine:

```
void    DeallocKstk(kstkbase)
        vaddr       kstkbase;
```

Deallocate contiguous pages that form a Kstk.

To deallocate a kernel stack, all that is required is to deallocate each of the pages that comprise the stack. (The allocation of a kstack is more difficult because it requires that several consecutive pages be found in the page heap; if consecutive 'AllocVP()'s were used to accomplish this, there would be no guarantee that consecutive pages would be obtained; so special attention is needed for this problem. The deallocation of a kernel stack does not have to be done atomically and so is much easier.)

3.1.3.34 Formatted Console Printing — printf.c

This C source file contains logic for

Routine:

```
void    printf();
```

This is a simple, portable version of the standard C-library routine.

Calls only putchar.

Escape sequences implemented:

%nd	1-prec decimal, signed
%nD	2-prec decimal, signed
%no	1-prec octal, unsigned
%n)	2-prec octal, unsigned
%nx	1-prec hex, unsigned
%nX	2-prec hex, unsigned
%e	float, 6 digits precision
%ns	string
%c	character
%%	percent

3.1.3.35 Kernel Interface to Scheduler — sched.c

This C source file contains logic for procesor scheduling. Actually, it is an interface to the scheduling subsystem proper.

Routine:

```
void    PushPreemption()
```

Push preemption state.

This routine is used to defer preemption for the current thread.

Routine:

```
void    PopPreemption()
```

This routine is used to pop the preemption disable stack. If a preempt attempt was made during the deferral interval, it it honored now by calling 'Resume()' to save the current thread's state and run the preempting thread.

If no preempt attempt is outstanding or preemption is still disabled, this routine just pops to the next level in the preemption disable stack and returns.

If the preemption is honored, a preemption confirmation is sent to the AP.

Routine:

```
void    MakeReady(KTptr)
        KTCBptr     KTptr;
```

This routine adds a thread to the ready queue.

This is the case where the scheduling coprocessor is used, so a check is made to see if this is a new thread, and if so a command is sent to the SP with the thread's initial scheduling parameters, otherwise a command is sent to simply add the thread to the SP's mix.

Note that we can't really tell whether the CPU is idling or in the middle of preemption sequence initiated by the scheduling co-processor. We have to wait for the SP to tell the AP what to do.

Routine:

```
void    Block()
```

This routine removes the current running thread from the ready queue, marks it as being in the blocked state, and gives up the processor.

This is the scheduling co-processor version.

This routine sends a command to the SP indicating that the thread is to be removed from the ready list and blocked. The 'Resume()' routine is then called to save the thread's state and to give up the processor to the next ready thread.

This routine is (currently) only called from the 'P()' synchronization operation, and is done with interrupts disabled. Interrupts must be disabled regardless from where the routine is called.

If a preempt is pending when this routine begins, a 'Resume()' is done straight away. This ensures that the preempt is done correctly, and gives the processor away the next time the thread is run. This makes it look like the preempt got in first.

Routine:

```
void    UnBlock(KTptr)
        KTCBptr     KTptr;
```

This routine removes threads from semaphore queues that they are blocked on.

It starts by removing the thread from the semaphore's blocked queue, makes the thread ready, and does a V on the semaphore.

Note that this forces a thread to be unblocked and can cause serious synchronization problems if used improperly. This routine is intended to be used primarily in the process of aborting a thread.

This routine is atomic with respect to interrupts, and doesn't do anything if the thread is not blocked.

Routine:

```
void    Sleep(delay)
        u_long        delay;
```

This routine blocks the current thread and restarts it after a given number of microseconds.

This routine checks if the delay is within epsilon of the current time, and if so, it does not block the thread, but just spins a little and returns.

Routine:

```
void    Suicide()
```

This routine is called by a thread that wants to kill itself.

If the scheduling coprocessor is being used, it must be informed of the thread's death. This is done by sending a command to the SP. This routine makes the current thread be the idle thread and then begins execution of the next thread that is ready to be run.

Upon receiving a KILL request, the AP will be idling.

This routine is called by the thread management object, and is called with interrupts disabled (they will be enabled in 'NextReady()'). The reason why this is done with interrupts disabled (instead of only disabling preemption), and no check is made for pending preemptions is that the KTCB has been dealloced and we want to get this over with as soon as possible.

Routine:

```
void    Resume()
```

The calling thread gives up the processor, and the next thread ready to run takes over.

With the uniprocessor scheduler, this routine does not add the calling thread back to the ready run queue — that is the responsibility of the caller (or somebody else) and is done with the 'MakeReady()' routine.

When the scheduling coprocessor is used, threads are not removed from the ready queue by 'NextReady()' — instead 'NextReady()' just returns the preempting thread (when there is one) and the SP manages the ready queue.

For the case where the scheduling co-processor is present, the next ready thread to run will always be the thread that is preempting, and if no preemption attempt has been made, the idle loop will be executed in 'NextReady()' to wait for further instructions from the SP.

This routine does not change the state of the thread it is saving nor the new being run. The running thread's state must be changed before calling this code, and the 'Run()' routine changes the new thread to the running state.

When the calling thread resumes, it will pick up at the end of this routine and return from this call.

The next thread to run sets up it's interrupt state to what it was when it was descheduled, and when the calling thread is reactivated, it resumes with the same interrupt state that it had upon entering this routine.

Routine:

```
u_short LoadThread(KTptr)
        KTCBptr     KTptr;
```

Set up a Thread's Context so That it Can be Run .

Take a pointer to the KT part of the thread that is to be started, decide if it is a client or kernel thread, and set up its context If it is a kernel thread, unstick the exiting thread's KSTK segment, and stick down the new thread's KSTK segment (i.e., the segment in which the thread's kernel stack page exists).

If it is a client thread, see if the thread already has a context loaded, if not, aquire a context to run in and set up the context. Then, unstick the exiting thread's KSTK segment, and stick down the new thread's KSTK segment (i.e., the segment in which the thread's kernel stack page exists).

Routine:

```
KTCBptr NextReady()
```

Get the Next Thread that is Ready to be Run.

This routine gets the next thread to be run. This is the scheduling coprocessor version.

There are two cases to consider:

- Preemption has been attempted, and so a pointer to the KTCB of the preempting thread is returned.
- No preemption was attempted, and so the idle loop is executed.

This routine only returns when it has a ready thread to pass back. If there isn't a ready one, the idle thread becomes the current thread and this routine spins (i.e., it is the idle loop). In this case the preempting thread is always the one returned.

This routine resets the preemption in progress flag — if a preemption confirmation is required, it has to be done before this routine is called.

Note that this routine assumes that interrupts are disabled by the caller.

Routine:

```
void    Run(KTptr)
        KTCBptr     KTptr;
```

Start a new thread running.

This routine is called whenever a new thread is to be run and the previously executing thread has been saved (if it wasn't the idle thread, which doesn't need to have its state saved). This code loads the specified thread and starts it running.

This routine checks if a preemption confirmation is required, and if so sends a command to the SP to confirm the preemption request (a different confirmation is sent depending on whether a thread was running at the time or the AP was idle).

The given thread's saved state is restored and execution is begun with an 'rte' instruction.

Routine:

```
void    Preempt()
```

Preempt the currently running thread.

This routine suspends the currently executing thread and starts the new thread indicated by the 'preemptthread' global variable.

It is not necessary to save the thread's state when preempting the idle thread, the new thread is run immediately without doing a resume. Otherwise, a resume must be done to save the current thread's state and run the next ready thread.

If the currently executing thread is not the idle thread, and if preemption is currently deferred, this routine indicates that a preemption has been attempted and just returns. The 'PopPreemption()' routine detects that the preemption was attempted and calls 'Resume()' at that point. If preemption is not deferred, 'Resume()' is called to save the current thread's state and give up the processor. The call to 'Resume()' puts the preempted thread back into the ready state, and the new thread into the running state.

A confirmation is sent to the SP, whenever a new thread is run, regardless of whether the AP was running the idle thread or not.

This routine is only called from the IPI interrupt handler that deals with interrupts from the SP. This routine either calls 'Run()' and doesn't return, calls 'Resume()' which calls 'Run()' and doesn't return, or it returns normally.

This routine is only called from the IPI interrupt handler.

3.1.3.36 Thread Section Manipulations — sections.c

This C source file contains logic for Thread Section manipulation.

Routine:

```
void    InitSectionTable(stptr)
        sectiontableptr     stptr;
```

Initialize a Section Table.

Make the free list take up the whole of the table, and also make sure all the start of chain pointers refer to empty lists.

Routine:

```
void    TrimThread(tid, sec, schedparmptr)
        abid            tid;
        u_short         sec;
        SIIBptr         schedparmptr;
```

Trim a thread back to a certain section on this node.

Look up the head of the section list for this thread id. Search down the list, aborting all the threads with a section greater than or equal to the one given in the parameter.

This routine is only called from the 'CommProcIntf' object/thread.

Routine:

```
void    AddSection(KTptr)
        KTCBptr         KTptr;
```

Add a threadID/KTCBptr pair to this node's section structures.

This routine uses the node's section table structure to discover the KTCB of the given threadid that has the highest section number. The given KTCB will form the head of a linked list of KTCBs with the same threadid (stored in descending section order).

This routine assumes that KTCBs to be registered are always of later section than the latest one already registered, and puts the new one at the head of the list.

Routine:

```
void    RemoveSection(KTptr)
        KTCBptr       KTptr;
```

Remove a thread section from the mapping table.

This takes a pointer to a KTCB and removes it from the mapping table.

The tables is searched for the list of sections of the given thread, it returns without doing anything if this thread does not appear in the table, otherwise, it is removed from the table.

3.1.3.37 Semaphore Services — semsrv.c

This C source file contains logic for Semaphore System Service Objects.

Routine:

```
void    SemMgrSrv(rqst, rply)
        rqstpblkptr   rqst;
        rplypblkptr   rply;
```

Demultiplex the Semaphore Manager SSO Operations.

Finds out which operation is being requested, and invokes it.

Provides the object interface for this SSO (i.e., packs/unpacks parameters from the request/reply side of the incoming parmblk).

Routine:

```
void    SemSrv(rqst, rply)
        rqstpblkptr   rqst;
        rplypblkptr   rply;
```

Demultiplex the Semaphore SSO operations; find out which operation is being requested, and invoke it.

Provides the object interface for this SSO (i.e., packs/unpacks parameters from the request/reply side of the incoming parmblk).

Routine:

```
capa    AllocPrivateSem(cnt, pvundo, allocundo)
        int           cnt;
        boolean       pvundo;
        boolean       allocundo;
```

Allocate a Private Semaphore for an Object and return a capability for it.

Routine:

```
u_short DeallocPrivateSem(semcapa)
        capa          semcapa;
```

Deallocate a Private Semaphore for an Object.

3.1.3.38 System Timer — stimer.c

This C source file contains logic for dealing with the System Timer.

Routine:

```
void    InitSTimer()
```

Initialize the system timer and start it running.

Typically, the AMD9513's timer 3 and 4 are cascaded to form a 32 bit system timer. For a SUN 1.5 that is short of timers, timer 4 is used as the low part of the 32 bit system timer.

In the case of being short of timers, timer 4 is set up to generate an interrupt on each terminal count (TC). This interrupt is used to update a 16 bit value in memory that contains the high 16 bits of the (logically 32 bit) system clock.

Routine:

```
u_int   GetSTime()
```

This routine returns the current time from the system timer.

Routine:

```
void    STimerIntrHdlr()
```

This routine handles interrupts generated by the interval timer.

The purpose is to increment the high part of the system timer that is kept in software.

3.1.3.39 Kernel Synchronization — sync.c

This C source file contains logic for Kernel synchronization.

Routine:

```
semaphoreptr AllocSem(cnt)
        u_short         cnt;
```

Allocate and initialize an empty semaphore, and return a pointer to it (initialization — 'count' = cnt, 'next' = NULL.)

Take a pointer to the semaphore, set its next pointer to NULL, and set its count to the specified initial value.

Routine:

```
void    DeallocSem(sem)
        semaphoreptr        sem;
```

Deallocate a semaphore.

Routine:

```
void    InitSem(semptr, cnt)
        semaphoreptr        semptr;
        u_short             cnt;
```

Initialize a semaphore,

Take a pointer to the semaphore, set its next pointer to NULL, and set its count to the specified initial value.

Routine:

```
void    P(semptr)
        semaphoreptr        semptr;
```

Do a P operation on ’sem’, which is a pointer to a counting semaphore.

This is done by decrementing the token count and checking if we got one. If we got a token this routine just returns, and if it didn’t get one it gets put on the semaphore’s queue and blocks.

The ’Block()’ routine is used to give up the processor.

This does a resume to save the thread’s state and informs the SP of the processor’s release when the scheduling coprocessor is used.

Routine:

```
void    V(semptr)
        semaphoreptr        semptr;
```

Do a V operation on ’semptr’, which is a pointer to a counting semaphore.

This routine adds a token back to the semaphore and checks if there are any threads waiting for the token. If there are waiting threads, one is chosen to get the token that was just returned, and it is started up.

Currently the oldest thread is chosen as the next (i.e., a FIFO queueing policy), but this will be changed so the scheduler chooses the next one based on its current policy.

The chosen thread is made ready to run by a call to the ’MakeReady()’ routine that puts the thread into the ready state.

Routine:

```
void    VAll(semptr)
        semaphoreptr        semptr;
```

V operation on ’sem’, which is a pointer to a counting semaphore.

This operation releases all the threads waiting on this semaphore.

This operation is equivalent to executing ’V()’ operations until there are no more threads blocked on this semaphore. Rather than actually calling the ’V()’ routine that often, an optimization is used: each blocked thread is made ready to run and the semaphore’s count field is set to zero (to indicate that there are no more threads blocked on it). (Note that the count field and the number of threads blocked on the semaphore may not be consistent due to the execution of some number of ’UnBlock()’s on threads that had been blocked on the semaphore.)

Routine:

```
void    AdjustSems(semptr)
        semaphoreptr        semptr;
```

This routine adjusts the information associated with each of the semaphores belonging to a replicated object. This adjustment is only intended to be performed when a backup copy of the replicated object is promoted to be the primary copy (following the failure of the previous primary copy). At that moment, the backup copy’s ’privatesem[]’ array contains a snapshot of the previously checkpointed state of the primary copy’s array. In order to guarantee a smooth transition to the backup copy, the semaphore information must be transformed by:

(1) eliminating all pointers contained in the semaphores (these are now dangling references since they are KTCBptr’s from a presumably failed node);

(2) setting the ’count’ field of each semaphore to be a non-negative number; that is, any semaphore with a positive ’count’ field is not altered, while any count that is less than zero will be set to zero; (threads are blocked when the ’count’ becomes negative; if a thread had been blocked in the primary object at the time of the failure and that semaphore’s ’count’ had been transferred to the backup copy due to another thread’s checkpoint operation, a deadlock situation might arise; by resetting the ’count’ to zero this deadlock situation should be avoided).

3.1.3.40 System Wide Error Function — system.c

This C source file contains logic for system difficulties.

Routine:

```
void    Error(str)
        char    *str;
```

Print an error message on the console, and then croak.

3.1.3.41 TempServices Object Support — tempsrv.c

This C source file contains logic for temporary services.

Routine:

```
void    TempServicesSrv(rqst, rply)
        rqstpblkptr    rqst;
        rplypblkptr    rply;
```

Service an invocation on the TempServices SSO.

Demultiplex the TempServices SSO operations; find out which operation is being requested and call a routine to provide it.

This routine provides the object interface for this SSO (i.e., it packs/unpacks parameters from the request/reply side of the incoming parmblk).

Routine:

```
u_long          SetImportance(KTptr, new_importance)
        KTCBptr        KTptr;
        u_long         new_importance;
```

Set scheduling importance of a thread.

3.1.3.42 Time Object Support — timemgmt.c

This C source file contains logic to demultiplex time object requests.

Routine:

```
void    TimeSrv(rqst, rply)
        rqstpblkptr    rqst;
        rplypblkptr    rply;
```

Demultiplex the Time SSO operations.

The defined operations are:

GET_TIME: Which reads the system clock and returns it to the caller. The time is returned in terms of an unsigned integer number of system clock ticks (currently 200ns).

SLEEP: Which suspends the execution of the thread’s execution for a time (greater than or equal to) the given parameter. The delay parameter is given in milliseconds.

3.1.3.43 Thread Management — threadmgmt.c

This C source file contains logic for managing threads.

Routine:

```
void    ThreadMgrSrv(rqst, rply)
        rqstpblkptr    rqst;
        rplypblkptr    rply;
```

Service an invocation on the ThreadManager SSO.

Demultiplex the ThreadManager SSO operations; find out which operation is being requested and call a routine to provide it.

This routine provides the object interface for this SSO (i.e., it packs/unpacks parameters from the request/reply side of the incoming parmblk).

Routine:

```
void    KillThread()
```

This routine destroys the current thread.

A check is made to determine if the thread to be killed is a root or a surrogate and its paramblks are deallocated accordingly.

By the time this routine is called, it is expected that any client thread structures will have been deleted and all that remains is the kernel thread whose KTCB is pointed to by 'currentthread'.

The 'Suicide()' routine is called to kill off the current thread and run the next thread. It is not necessary to enable interrupts because the current KTCB is gone for good, we are in the idle thread, and 'Suicide()' calls 'NextReady()' and 'Run()'.

Do not push preemption before calling this routine. Since the pop is never done, a preemption may not be confirmed.

3.1.3.44 Resource Tracking — track.c

This C source file contains logic for resource tracking.

Routine:

```
void    InitTrackTable(ttabptr)
        tracktableptr          ttabptr;
```

Initialize a Resource Tracking Table.

Routine:

```
u_short SemRegister(semptr, opr)
        semaphoreptr           semptr;
        semopr         opr;
```

Record a Semaphore Operation.

This routine records for later use information on how to undo the semaphore operation "opr" in the resource tracking table of the current thread. If this thread gets rolled back to an earlier level, data from its tracking table can be retreived on a per-level basis and used to undo its actions. Only P's, V's, and allocations can be undone, as there is no meaningful way to undo a dealloc.

Routine:

```
void    EndLevel(KTptr)
        KTCBptr      KTptr;
```

Indicate the End of an Abort Level.

Necessary actions that must be performed when a thread leaves a level are performed here. All the resource information associated with the current level is moved down to the previous level, to await completion or abortion there.

Routine:

```
void    SemRollBack(KTptr, level)
        KTCBptr      KTptr;
        u_short      level;
```

Undo the effects of Semaphore Operations to a given level.

Tries to undo all the semaphore operations done by the KTptr thread beginning with the given level. Uses the thread's tracking table to find out what must be done to roll back the operations.

3.1.3.45 Console Interface — tty.c

This C source file contains logic for the console TTY interface.

Routine:

```
void    InitTty()
```

Initialize the tty, allocate and set up the semaphores and queues, and then initialize the UART itself. This sets the default mode to 'RAW', and is called once at system initialization, this means that you don't have to disable interrupts, because none will be active.

Routine:

```
void    TtyCtl(param, val)
        ttyctlparam  param;
        ttyctlval    val;
```

Set the control parameters of TTY interface.

Assign the value contained in 'val' to the interface parameter specified by 'param'. Also, carry out any cleaning up that must be done as a result of the parameter change.

Routine:

```
void    TtyRead(bufaddr, bufsize, lenaddr)
        vaddr        bufaddr;
        u_int        bufsize;
        u_int        *lenaddr;
```

Perform a read operation on TTY object.

Read a character string from the TTY and place it in the buffer pointed to by 'bufaddr'. 'bufsize' is the maximum string length. (NOTE: strings returned by 'TtyRead()' are not NULL terminated.) 'lenaddr' points to the number of characters returned by the read operation — typically, 1 for "RAW" mode, or the number of characters read until a carriage return (inclusive) in "COOKED" mode (up to a maximum of 'bufsize').

Routine:

```
void    TtyWrite(bufaddr, count)
        vaddr       bufaddr;
        u_int       count;
```

Write a character string to the TTY. 'count' specifies the number of characters to be written. (NOTE: the string does not have to be NULL terminated.)

Routine:

```
char    ReadChar()
```

This routine is used to actually get a character from the incoming queue. It does a 'P()' to wait until the queue is non-empty, and then takes the next character from the queue.

3.1.3.46 Console Interrupts — ttyintr.c

This C source file contains logic for the Console TTY Interrupt Handler.

Routine:

```
void    TtyInIntr()
```

This is the UART Receive buffer full interrupt handler.

This is called by the assembly language TTY interrupt handler (in uart.a) when the UART's receive buffer has a character in it. This routine is responsible for taking the character from the UART and putting it into the incoming queue.

This routine first checks to see if the incoming character is an extraneous null caused by a break condition. If so, it throws it out.

It then determines the mode in which the tty currently operating.

In "RAW" mode it begins by getting the character from the UART's receive buffer, puts it in the incoming queue, and does a 'V()' operation to signal an awaiting thread that a character has arrived.

In "COOKED" mode the incoming character is examined; if it is a specially recognized control character (e.g., ^H, ^U, ^R, etc.) the appropriate action is taken, if the character is not a special control character and is not a CR, then the character is added to the incoming queue and the routine returns, if the character is a CR, it is added to the incoming queue, a 'V()' operation is issued to signal a waiting thread, and then the routine returns.

If the incoming queue becomes full, the oldest character in the queue is overwritten to ensure that the most recent "N" characters are always in a full queue.

Routine:

```
void    TtyOutIntr()
```

Transmit buffer empty interrupt handler.

This is called by the assembly language TTY interrupt handler (uart.a) when the UART's transmit buffer is empty. This routine is responsible for putting characters buffered in the outgoing queue into the UART.

This routine starts by checking the outgoing queue for characters, if the queue is not empty, the next character is put into the UART's transmit buffer.

When the queue empties, the uart is told to stop sending interrupts.

Routine:

void StartTtyOut()

Start Tty output.

Called by the user each time a character is put into the outgoing queue. First checks to see if the UART transmit buffer is empty. If so get the next character from the outgoing queue, put it in the UART, signal waiting thread with a 'V()', and return.

This does not run under interrupt, it is called by the tty kernel object and so it has to disable UART interrupts .

3.1.3.47 Virtual Memory Management — vm.c

This C source file contains logic for Virtual Memory Management (For the Sun-2 Memory Management Unit).

The 'PPstate' structure indicates which physical pages are in use, what 'pmap's' they belong to, and what page this is within the 'pmap'.

There are three fields in the 'PPstate' structure — a pointer to the 'pmap' that owns the indexed page (and a 'NULL' if it is not owned or if it is owned by the kernel), a page index of this page within the 'pmap', the current invocation depth of the thread this page belongs to, and an index to the next page in the free physical page list (or a 'WIRED' if the page is "wired").

If the indexed page is not in use (i.e., it is in the free list), then the 'next' field contains an integer index indicating the next free physical page. Otherwise the 'owner' field points to the 'pmap' that this page is a part of, and the 'pmapindx' field indicates which page it is within the given 'pmap'.

The 'nextfree' field is used to chain together the pages that make up the free page list (that is rooted at the global variable 'firstfreePP'), and the variable 'PPcycle' is used (currently) to do a simple LRU replacement policy.

The variable 'wiredPPcount' is used to keep track of how many PP's are currently "wired".

The 'SMCstate' structure indicates whether a SMC slot is in use, and if so, by which Client Thread.

If the indexed slot is not in use (i.e., in the free list) then the 'next' field contains an integer index indicating the next free SM entry in the SMC. If the SMC slot is in use by a Client Thread, the 'threadowner' field points to the 'CTCB' of the client currently loaded in that context, and the 'objowner' field points to the 'COCB' of the object that the thread is currently active in (if it is active in one). The context used by the kernel is not made available for replacement, and has a NULL in both the 'threadowner' and 'objowner' fields, and a 'WIRED' in the 'next' field.

The 'next' field is used to chain together the SM's that make up the free list (that is rooted at the global variable 'firstfreeSM'), and a rotating pointer (the global variable 'SMCcycle') is used to select candidates for replacement.

The 'PMCstate' structure indicates which slot's are in use, and by which 'pmap's'.

There are two fields in the 'PMstate' structure — a pointer to a 'pmap' when the indexed PMC entry is in use by a thread, (and a NULL otherwise), and a index to the next slot in the free PMC slot list. If the indexed slot is not in use (i.e., is in the free list), the 'next' field contains an integer index indicating the next free PM entry in the PMC. If the PMC slot is in use by the kernel, it does not appear in the free list, it's 'owner' field contains a 'NULL', and it's 'next' field contains a 'WIRED'.

The ’next’ field is used to chain together the PM’s that make up the free list (that is rooted at the global variable ’firstfreePM’), and a rotating pointer (i.e., global variable ’PMCcycle) is used to select candidates for replacement.

The variable called ’stickyPMs’ is a bit vector that indicates which of the PM’s are currently unavailable for replacement by ’AllocPM()’. (this is for things like the current thread’s kernel stack, and any PM that is being updated by a VM routine and can’t be taken away)

Routine:

```
void    InitVM()
```

Initialize the VM global variables, clear the VM state tables, and set up the KCB structure.

All free lists are set up and terminated with ’NIL’ values, furthermore all unused fields in the state table structures are initalized to ’NULL’.

Context 0 in the SMC is reserved for the kernel and is unused, but remains in the array so that the table indicies correspond directly to the SMC entries.

This only initializes the VM structures and set’s aside the pages that are always wired down for the kernel.

A VM structure’s ’owner’ field points to the owner if it’s in use, a NULL if it’s not in use, and a NULL if it is in use by the kernel. Also, the ’next’ field points to the next entry if it is a part of the free list (or a NIL if it is at the end of the free list), and has a ’WIRED_VM’ in it if it is in use by the kernel.

Assume that "cycle" variables are always preincremented by the replacement code and wrap around is done back to the initial point.

The static parts of the KCB structure are linked together, and the ’KernCB’ and all of its FCB’s are entered in the Dictionary.

Routine:

```
u_short AllocPP()
```

Allocate a Physical Page from the pool of free pages.

Returns index of a physical page and wires the page down. It also decrements the count of free pages (this count is kept to assist in the activation of the page pool manager thread).

This routine is called from the kernel to get wired pages for its own use, and to get wired pages for the kernel page heap (both immediately and when a "dummy" page fault occurs). The page heap is used for parmblk and kernel stack pages, and wired pages are appropriate for both of these uses.

The count of the number of wired pages is incremented and if the limit is exceeded, a fatal error has occurred.

This takes pages off the head of the free page list, and sets the page’s ’next’ field to ’NIL’ (this means it’s not in either the ’inuse’ or the ’free’ page lists).

Should the free list be empty, it is either a system error or a page is dynamically allocated (and the caller is made to block while the page is freed).

This routine just gets a free page and returns its index in the array of physical pages.

It is up to the caller to install the page in some VAS and to write the appropriate stuff in the right PD and ’pmap’ (and it can even put the right stuff in the page’s ’PPstate[]’ entry).

If an unwired page is needed, the caller is responsible for unwiring it (i.e., put it in a 'pmap' and some page list). (The 'InstallPP()' routine will do most types of page installation, and calls this routine itself.)

Routine:

```
void    DeallocPP(pageindx)
        u_short      pageindx;
```

Deallocate a Physical Page and return it to the free page pool.

Return a page (given by its page index) to the list of free physical pages. This puts pages back at the head of the free page list, and sets the 'pmap', 'depth, and 'indx' fields to NULL.

This routine permits the kernel to release pages that belong to a client.

It is up to the caller to invalidate the page descriptor and update all of the VM structures for the given page (including decrementing 'wiredPPcount' if it was wired).

Routine:

```
void    InstallPP(addr, protection, pmptr)
        vaddr        addr;
        u_long       protection;
        pmapptr      pmptr;
```

Install a physical page into a VAS.

A physical page is allocated, is installed in the 'pmap' pointed to by 'pmptr', at the page index derived from the destination virtual address. The protection assigned to the page is given by the 'protection' parameter.

If the segment is currently loaded, the proper PD within the segment's PMC slot is updated.

In the indicated 'pmap', the 'PMimage[]' field is loaded with a page descriptor (the same as what gets put into the PMC), and the 'PGstate[]' field is set to 'RESIDENT'.

The global 'PPstate[]' structure is updated to reflect the installation of the page into a 'pmap'. This only works when called from the context that the page is to be installed in. The order in which things are done is significant in order to reduce the amount of time spent with interrupts disabled.

Routine:

```
void    UnInstallPP(addr, pmptr)
        vaddr        addr;
        pmapptr      pmptr;
```

Remove a physical page from a VAS.

Deallocate the page at the given address ('addr'), and remove it from all of the virtual memory structures.

If the segment the page is in is currently loaded, the proper PD within the segment's PMC slot is updated. In the indicated 'pmap', the 'PMimage[]' field is loaded with an invalid page descriptor, and the 'PGstate[]' field is set to 'ILLEGAL'.

This only works when called from the context that the page is to be installed in.

Currently this routine expects only wired physical pages.

Routine:

```
u_short AllocPM()
```

Allocate a Page Map entry in the Page Map Cache.

Similar to 'AllocPP', but maintains state of the PMC. If no free PMC slot is available, take one away. This is done by a cycling through the PMC, skipping wired and stuck PM's, and unloading a victim PM into its 'pmap'. All SD's that reference the removed PM must be invalidated in the SMC too.

The allocated PM's are always stuck down, it is up to the caller to unstick it when the time is right.

In order to access a PM, temporarily map the segment into the kernel space and then restore it when done. This routine just returns an index to an empty PM slot, the calling routine is responsible for loading the returned PM. This code must write the victim PM back into its 'pmap', or the dirty and referenced bits will be lost.

The kernel's PM's are wired down and are not eligible for replacement (this is handled by wiring the first "N" PM's and skipping them in the replacement cycle).

This routine also skips the reserved PM's and the PM's that are currently stuck down.

Routine:

```
void    DeallocPM(PMCslot)
        u_short     PMCslot;
```

Deallocate a Page Map entry in the Page Map Cache.

Release a PM indexed by 'PMCslot', (i.e., a relative index into 'PMCstate' or the PMC itself), increment the count of number of free PM's, and link the given one in at the head of the free list.

Assume that no further references to the PM exist. Allows kernel to deallocate any page map it choses.

Routine:

```
u_short AllocSM(CTptr, COptr)
        CTCBptr     CTptr;
        COCBptr     COptr;
```

Allocate a Segment Map Cache Slot (i.e., a context).

Find an empty cache slot (aquire one if necessary) and return an index to it. Invalidates the 'SMCslot' fields of threads and objects that were using the aquired context.

This routine just returns an index, it is up to the caller to load the indexed SMC slot.

Routine:

```
void    DeallocSM(SMCslot)
        u_short     SMCslot;
```

Deallocate a Segment Map Cache Slot (i.e., a context).

Return an empty cache slot to the list of free SM slots.

Invalidates the 'SMCslot' fields of threads and objects that were using the returned context.

Routine:

```
boolean CopyPage(srcCO, srcvaddr, destvaddr, destcnt, destCO)
        abid        srcCO;
        vaddr       srcvaddr;
        vaddr       destvaddr;
        u_short     destcnt;
        abid        *destCO;
```

Copy a page from one COs address space to other COs' address spaces.

Copy the page at address 'srcvaddr' in CO 'srcCO' to address 'destvaddr' in the 'destcnt' copies (possibly backup copies) listed in the 'destCO[]' list. 'TRUE' is returned iff all copies to all destinations occur successfully.

Routine:

```
boolean InstallPage(pgaddr, CO, destaddr)
        vaddr       pgaddr;
        abid        CO;
        vaddr       destaddr;
```

Install a page into a CO's address space.

Install the page currently at address 'pgaddr' in 'CO' at address 'destaddr'. Since this routine is called in order to install pages that have been received by the Communications Processor, it is necessary to copy the contents of the page, rather than just mapping

the page into the indicated CO's address space. (Otherwise, the kernel might hold on to a large number of pages in shared memory that is actually managed by another processor.) 'TRUE' is returned iff the installation is completed successfully — i.e., the CO is found, the 'destaddr' is legal, and so on.

3.1.3.48 Virtual Memory Fault Handler — vmfault.c

This C source file contains logic for the virtual memory fault handler.

Routine:

```
void    VMFault(addr, ssw, pc, sr)
        u_long      addr;
        u_short     ssw;
        u_long      pc;
        u_short     sr;
```

Handler for VM faults (called on Bus Error Trap).

This code is called by the (assembler) bus error trap handler when it has been determined that the cause of the fault is in fact a VM fault, and not a fatal system error, that is handled by the monitor code (or at least it thinks so at this point).

The trap handler passes in the faulting address and the special status word.

This code runs in supervisor state. Also, interrupts are disabled by the caller (i.e., 'BusErr()').

Check if the thread causing the fault is the idle thread, and if so, generate a fatal error — the idle thread had better not generate any memory faults because it should be wired down.

This routine determines if it is a segment or page fault (by looking at the faulting segment descriptor). If it is a segment fault, the index field of the SD is examined to determine if it was an anticipated segment fault, or an invalid segment error. If it is an error a message to that effect is printed on the console before trapping to the monitor. There are two types of error: an unused segment is accessed or a used segment is accessed in an illegal manner (without the appropriate permissions for the attempted access). The special status word ('ssw') contains the information needed to check if the reference was legal or not.

If it wasn't an error the 'pmap' for the faulted segment is located, a PMC slot is allocated, the slot is loaded with information from the 'pmap's image, the SD index field is set to the proper PMC slot, and this code returns to the faulting instruction.

If the fault was a page fault, the page descriptor is examined to determine which of the "N" different varieties of page faults it is.

If 'Scratch', it gets a real page from the free page pool, installs it in place of the dummy page, and returns.

If 'PBScratch', it gets a real page from the free page pool, installs it in place of the dummy page, ensures that it is doubly mapped (for both the kernel and the client), and returns.

If 'PagedOut', it calls 'PageIn()' and if it returns the page, installs it then returns, otherwise it must wait for the page to come in off disk and so it blocks the thread (on a semaphore), and resumes the next thread on the ready list (i.e., gives up the processor).

If 'InFile', it calls 'ReadFile()' and if it returns the page, installs it then returns, otherwise it must wait for the page to come in off disk and so it blocks the thread (on a semaphore), and resumes the next thread on the ready list (i.e., gives up the processor).

If 'CstkOverflow', it checks if the stack extent is full, and if so prints an error message before trapping to the monitor, and if not extends the stack, installs the necessary segments/pages, and returns.

If 'CstkUnderflow', 'Resident', 'Illegal', or 'Wired', prints an error message on the console, and traps to the monitor.

When this routine returns successfully, preemption is restored and it returns to the caller.

This code tries to do things much as is done in the 'bedecode()' routine in the monitor (mogul@Stanford).

3.2 Scheduling Subsystem

The scheduling subsystem is linked separately from the kernel proper, and runs on the scheduling processor (SP). However, it shares some code with other subsystems. The descriptions below will only deal with the files which are unique to the scheduling subsystem.

3.2.1 Assembler Files

The files crt0.a, excep.a and util.a are used by this subsystem but are defined elsewhere.

3.2.1.1 Floating Point Support — fcrt.a

This file contains support for software-based floating point arithmetic. It is basically a trap handler.

3.2.2 Include files

3.2.2.1 Best Effort Scheduling Policy Constants — BEcons.h

This file contains constants related to the Best-Effort scheduling policy.

Constants:

MAX_INT
LOWEST_PROB

3.2.2.2 Clock Frequency Constants — CLKFcons.h

This file contains constants related to the clock frequency.

Constants:

MSEC_IN_SPTICKS
SPCLK_EPSILON
MAX_CLOCK_VAL
USECS_TO_F2TICKS
SI_PROPT_OH
MAX_T_EVENTS
TE_COMP_TIME
TE_MIN_INTERVAL
START_FREQ
FREQ_UNIT
MAX_FREQ
SYSTIMER_IS_F1_MASK

Macros:

F2TICK_TO_SPTICKS
SPTICK_TO_F2TICKS
MSEC_TO_SPTICKS
USEC_TO_SPTICKS
MSEC_TO_F2TICKS
USEC_TO_F2TICKS

3.2.2.3 Math Library Constants — MATHcons.h

This file contains constants related to the math library.

Constants:

HUGE

3.2.2.4 Event Logging Constants — SAEVcons.h

This file contains constants related to event logging.

Constants:

LE_BUF_SIZE
MAX_L_EVENTS

3.2.2.5 Event Queues Constants — SEVcons.h

This file contains constants related to scheduling event queues.

Constants:

SEVQ_BACKLOG
SEVQ_CATCH_UP

3.2.2.6 Value Density Scheduling Policy Constants — VDcons.h

This file contains constants related to Value Density scheduling policy.

Constants:

MAX_INT
LOWEST_PROB

3.2.2.7 Best Effort Scheduling Policy Header — be_pol.h

This file defines data structures related to the Best Effort scheduling policy.

Data Structures:

BEtcbptr
TCspcbptr
TCspcbstr
BEtcbstr

3.2.2.8 Deadline Scheduling Policy Header — dl_pol.h

This file defines data structures related to the Deadline scheduling policy.

Data Structures:

DLtcbptr
DLspcbptr
DLtcbstr
DLspcbstr

3.2.2.9 Fixed-Size Storage Allocator Header — falloc.h

This file defines data structures related to the fixed-size storage allocator.

Data Structures:

fallocptr
fallocun

3.2.2.10 FIFO Scheduling Policy Header — fifo_pol.h

This file defines data structures related to the FIFO scheduling policy.

Data Structures:

FIFOtcbptr
FIFOtcbstr

3.2.2.11 Normal Distribution Table — normal.h

This file defines data structures and values which constitute a lookup table for the Normal (Bell) distribution. (This file is largely machine-generated.)

Data Structures:

normtabptr

normtabstr

Variables:

normaltable

Constants:

NORMTABLEDELTA
NORMTABLESIZE
TABLEDELTAON2
ONEONTABLEDELTA

3.2.2.12 Queue Header — queue.h

This file defines data structures related to the queues.

Data Structures:

qptr
qendsptr
qstr
qendstr

Constants:

MINKEY
MAXKEY
HVKEY_F
LVKEY_F
Q_NULL

Macros:

QUEUE_HEADPTR
QUEUE_TAILPTR
QUEUE_HEAD
QUEUE_TAIL
QUEUE_EMPTY
QUEUE_NEMPTY
QUEUE_SIZE
QUEUE_ITEM
QUEUE_NEXT
QUEUE_PREV
QUEUE_KEY
QUEUE_INIT
QPTR_ITEM
QPTR_NEXT
QPTR_PREV
QPTR_KEY

QPTR_INIT
QUEUE_ISHEAD
QUEUE_ISTAIL

3.2.2.13 Round Robin Scheduling Policy Header — rr_pol.h

This file defines data structures related to the Round Robin scheduling policy.

Data Structures:

RRtcbptr
RRtcbstr

Constants:

QUANTUM

3.2.2.14 Scheduling Statistics Header — schedstats.h

This file defines data structures related to scheduling statistics.

Data Structures:

FIFOstatsptr
RRstatsptr
SPRIstatsptr
DLstatsptr
SPTstatsptr
SLKstatsptr
VDstatsptr
BEstatsptr
schedstatsptr
FIFOstats_info_str
RRstats_info_str
SPRIstats_info_str
DLstats_info_str
SPTstats_info_str
SLKstats_info_str
VDstats_info_str
BEstats_info_str
schedstats_str

3.2.2.15 Scheduler Header — scheduler.h

This file defines data structures related to the scheduler.

Data Structures:

sevptr
sibptr
sibstrptr
sevstrptr

sevenum
sibstr
sevstr

3.2.2.16 Interval Timer Header — sp_itimer.h

This file defines data structures related to the scheduling processor's interval timer.

Data Structures:

timedeventptr
timedeventstr

3.2.2.17 Debug Indirection Table Header — spdit.h

This file defines data structures related to the Scheduling Policy Debug Indirection Table.

Data Structures:

spdpptr
spdpstr

Macros:

SP_PRINTRQ
SP_PRINTTCB
SP_PRINTHWM

3.2.2.18 Scheduling Policy Indirection Header — spit.h

This file defines data structures related to the Scheduling Policy Indirection Table.

Data Structures:

sppptr
sppstr

Macros:

SP_INIT_INTR
SP_ADD_INTR
SP_REMOVE_INTR
SP_CHANGE_INTR
SP_MISC_INTR
SP_TIMER_INTR
SP_RESCHED_INTR
SP_ACCSI_INTR
SP_ABORT_INTR
SP_INIT
SP_ADD
SP_REMOVE
SP_CHANGE
SP_MISC
SP_TIMER

SP_RESCHED
SP_ACCSI
SP_ABORT

3.2.2.19 Static Priority Scheduling Policy Header — spri_pol.h

This file defines data structures related to the Static Priority scheduling policy.

Data Structures:

SPRItcbptr
SPRItcbstr

3.2.2.20 Best Effort Value Function Header — vf.h

This file defines data structures related to the value functions used by Best Effort scheduling.

Data Structures:

VFptr
VFstr
valuefunctype

Constants:

VFCT

3.2.2.21 Exponential Function Table Header — xdist.h

This file defines data structures and values relating to the Exponential table.

Data Structures:

xdisttabptr
xdisttabstr

Variables:

xdisttable

Constants:

DISTTABLEDELTA
DISTTABLESIZE
DISTDELTAON2
ONEONDISTDELTA
THALFAREA

3.2.3 Source Files

The files falloc.c, ipi.c and printf.c are used by this subsystem, but are defined elsewhere.

3.2.3.1 Best Effort Scheduling Policy Debug Support — be_debug.c

This C source file contains logic for the Best Effort Scheduling Policy debugger.

Routine:

```
void BEPrintHWM()
```

Prints the high water mark information gathered by the Best Effort policy.

Routine:

```
void   BEPrintRQ()
```

Print the contents of the ready queues.

Routine:

```
void   BEPrintTCB(TCBptr)
       BEtcbptr      TCBptr;
```

Print the contents of a thread control block.

3.2.3.2 Best Effort Scheduling Policy Math — be_math.c

This C source file contains logic for the math routines of the BE scheduler.

Routine:

```
double        ndist(t, mean, stddev)
       float   t;
       float   mean;
       float   stddev;
```

Returns probability that a normally distributed random variable (of given mean and standard deviation) is > t.

Routine:

```
double        xdist(t, mean, stddev)
       double        t;
       double        mean;
       double        stddev;
```

This routine returns the area of ’x*phi(x)’, integrated from -infinity to t.

Note: the integral is -.4 at 0, 0 at both infinities, and is symmetrical about 0.

Routine:

```
float   dist(t, mean, stddev2)
       float   t;
       float   mean;
       float   stddev2;
```

This routine returns probability that a normally distributed random variable (of given mean and variance) is <= t.

Routine:

```
double      denf(t, mean, stddev)
        double      t;
        double      mean;
        double      stddev;
```

This routine returns value of normal distribution function phi(t) for that mean and standard deviation.

3.2.3.3 Best Effort Scheduling Policy — be_pol.c

This C source file contains logic for the Best Effort scheduling policy.

Routine:

```
void  BEInit()
```

Scheduler Initialization Routine.

This routine initializes the queues, the interval timer, and global variables. Also, print a start-up banner.

The statistics kept by the scheduler are auto-initialized.

Routine:

```
long  BEAbort(SEVptr)
      sevptr      SEVptr;
```

Main-loop-level abort handling routine.

This routine is called when a thread's hard time constraint expires or when BEResched has determined that a soft deadline has expired. Soft time constraint aborts only occur when a soft deadline is the outermost one. Therefore, only the outermost time constraint need be aborted in this case. It is also possible that an outer hard time constraint expired and was queued before this request, in which case the abort block level recorded in the 'SEV' structure will be greater than that in the outermost TC. In that case, nothing need be done. Otherwise, the TC block is removed and deallocated, and the AP will be asked to abort that time constraint.

For hard time constraints, first check to see if the innermost hard time constraint has expired. If it has not occured, assume that the time constraint was removed before the time event could be canceled, and just print a warning message. Otherwise, remove the first hard time constraint and any normal time constraints that occured after it was established. Propagate the computation time statistics for each of these time constraints forward and deallocate the blocks. Finally, ask the AP to abort to the abort block level given in the time constraint whose timer expired. If there are no time constraints left, move the thread to the importance queue. Otherwise, remove the thread from the deadline queue and replace it at the new point.

This routine also queues a reschedule when it is done.

Routine:

```
long  BEAddIntr(SEVptr)
      sevptr      SEVptr;
```

Interrupt-level add task routine.

If this is a delayed add (i.e., DADD), pass it to the interval timer system to be delivered later. Otherwise, queue the 'Sev' up to be delivered to the main loop service process.

Both regular adds and surrogate add (i.e., SADD) commands are queued for the main loop.

Routine:

```
long    BEAdd(SEVptr)
        sevptr        SEVptr;
```

Main-Loop-Level add task routine.

This routine adds a new thread to the SP's ready mix. It results from a call to 'MakeReady()' in the kernel, and can be either a regular add command (i.e., ADD), or a command to add a surrogate thread (i.e., SADD) — i.e., a thread created on another node.

If this thread is new to the SP, allocate a TCB for it.

If, in addition to being a new thread, it is also a surrogate thread, pick up the inherited scheduling information from the scheduling information block contained in the 'Sev' (i.e., scheduler event block). If this surrogate thread is under a time constraint, this picks up the scheduling information for the thread, and sets a timed event for the innermost deadline.

If the thread being added is not new to the scheduler, get the pointer to the thread's TCB from the 'Sev'. It is not necessary to set any timed events for time constraints in this case, because they are set when the TC is originally pushed. At this point, we have a valid, properly initialized TCB.

If the thread to be added is not under a time constraint, queue it in the importance queue (using its importance as the key), otherwise, queue it in the deadline queue (using its critical time as the key).

Finish by checking if a reschedule event has been queued (by examining a global flag), and if one is not queued up, then set the flag to indicate that one is queued, and queue up a reschedule event.

Routine:

```
long    BERemoveIntr(SEVptr)
        sevptr        SEVptr;
```

Interrupt-level remove task routine.

All types of remove commands are handled in the main loop, so just queue this command block up to be handled by the main loop.

Routine:

```
long    BERemove(SEVptr)
        sevptr        SEVptr;
```

Main-Loop-Level remove task routine.

This routine removes a thread from the ready mix. This class of command can only come from the AP's current thread, but there are two possibilities to consider. The first case is the simple one — i.e., the currently executing thread has decided to block, so we just remove it from the ready queue(s) that it appears in, and dispatch the next thread to be run.

The other case to consider is when the current thread has decided to block before it received a recently issued preempt command. In this situation the SP has indicated that a preempt is in progress and has set the 'preeptthreadptr' variable to the preempting thread, but has not yet set the 'currentthreadptr' variable to what it thinks is the new running thread. The AP thinks the previous thread is the current thread.

If this routine is called and the thread to be removed is not the current one, we need to be cautious about blindly trying to remove the thread; it may or may not still be in a ready queue (because whatever caused it to be preempted, i.e., 'Resched()', may have removed it from the queues already). This means that we use 'QKill2()' so it won't complain if a thread isn't in a queue.

In either of these two cases, the main thing that happens here is that the indicated thread is removed from all of the ready queues that it appears in, and the next thread to run is dispatched. Also, we update the blocking thread's ending run-time statistics. This means that we make note of the time at which the thread quit running (as defined by when the remove command was received), and update the necessary statistics.

There are two variants of this routine — in one case, the thread's TCB is just removed from the ready queue(s) that it appears in, and allowed to float around in the ether until it is added back to the ready mix, or killed. The AP keeps a pointer to the TCB around and it includes it in the 'Sev' when the thread is added back. This event is indicated by the REM command, and is initiated by calling the kernel's 'Block()' and 'Sleep()' routines.

In the other case, the TCB is removed from the ready mix in the same way that is done in the previous case, and, at the end, the all of the data structures associated with the thread are deallocated. This case is indicated by the KILL command, and is initiated by calling the kernel's 'Suicide()' routine.

After the given thread is removed, this routine performs a dispatch operation to run the next thread in the BE queue (if there is one), and queues a reshedule (if necessary).

Routine:

```
long    BEResched(SEVptr)
        sevptr          SEVptr;
```

Main-Loop-Level new schedule generation routine.

This routine takes the threads that appear in the IM and DL queues and generates the BE queue (from which threads are dispatched), based on the best-effort algorithm.

If both the DL and BE queues are empty, then this routine does nothing but dealloc its SEV.

If the IM queue is not empty, its contents are copied (in order) onto the head of the BE queue, and then this routine is done.

If there is nothing in the IM queue, and there is something in the DL queue, we generate a new schedule. This involves looping through the entire DL queue, adding threads to the BE queue in the order they are encountered in the DL queue, until an overload is detected.

For each thread in the DL queue, we check if it has a soft time constraint that has dropped to within epsilon of zero (we only check when it is the current time constraint, which can be detected by noting that the first thing in the TC list is not a hard TC). If we find that a soft time constraint has expired, we queue up a soft abort command and go on to the next thread.

For each thread in the DL queue, we compute how much time it needs, we keep a running total of the amount of slack time we have left, and we compute a probability that we have an overload at this point. If the probability of overload exceeds a given threshold, we do a load-shed, that drops some of the things we've put into the BE queue so far, and then goes on.

Before we do the load computation, we check if the thread is a problem thread, in that it represents a thread that is unlikely to be completed on time. If we decide that this is an overload case all by itself (i.e., it is asking for the whole CPU until its critical time), we pre-shed it — i.e., we just skip it, unless there is nothing else to run right now.

Routine:

```
long    BEChangeIntr(SEVptr)
        sevpttr         SEVptr;
```

Interrupt-Level change scheduling parameters routine.

All types of changes are presently handled in the main loop, so just queue the command up.

Routine:

```
long    BEChange(SEVptr)
        sevptr          SEVptr;
```

Main-Loop-level change scheduling attributes routine.

This routine handles all requests that involve the manipulation of a thread's scheduling-related attributes — i.e., time constraint pushing (PUSH_SA), popping (POP_SA), and simple change requests (CHG_SA).

Requests to push or pop time constraints are handled by their own routines, defined separately in 'be_util.c'.

'CHANGE_SEV' requests are solely used to change the importance of a thread. This request is handled by removing the thread from any queues it might be in, changing its importance, and reinserting it in any queues it was in originally. A reschedule is also necessary after this.

Requests to push abort scheduling information are handled by removing the present set of time constraints (the TC blocks), removing the thread from any queues it might be in, changing the scheduling parameters of the thread (including possibly allocating new TC blocks and chaining them), requeueing the thread if appropriate, and enqueueing a reschedule event.

It is the responsibility of the called routines to dealloc the SEV (if necessary).

Routine:

```
long    BEMiscIntr(SEVptr)
        sevptr          SEVptr;
```

Interrupt-level routine to handle miscellaneous commands.

This routine handles all of the misc commands associated with this specific policy. For this policy, the commands include the preempt confirmation commands (PRMT_CF and IPRMT_CF).

The preempt confirmation commands indicate that a requested preemption has been completed. This is a signal that the SP's notion of current thread is now valid and that thread is running. This command is needed because the AP can take time to respond to a

preempt command (due to such things as deferred preemption). The confirmation of a preemption is a signal that the current thread is now running and it's starting run-time statistics should be updated.

In the case when the confirmation indicates that the AP preempted a thread other than the idle thread (i.e., PRMT_CF), the ending run-time statistics for the preempted thread must be updated (along with the starting run-time statistics for the preempting thread). This must be done because there is no REMOVE command associated with this activity,

and it is that command that is usually responsible for updating the ending run-time statistics for a thread that is stopped.

The starting run-time statistics consist of the time that the thread was dispatched. The ending run-time statistics consist of the thread's total accumulated execution time, the time left until the thread's (currently defined) critical time, and the total computation time accumulated under the current time constraint. (Note that the latter two statistics need be gathered only if the thread is executing under a time constraint).

The other miscellaneous command (i.e., the statistics on/off commands (STATS_ON/STATS_OFF) are handled by the interrupt handler routine).

In all of these cases, global variables are set and nothing has to be done at the main-loop-level. Care should be taken in all of the main-loop routines that access these global variables (i.e., make sure to synchronize when it is necessary).

Routine:

```
long    BETimerIntr(SEVptr, TEtoken)
        sevptr      SEVptr;
        u_long      TEtoken;
```

Interrupt-level timer event routine.

This routine handles all of the interrupts generated by the interval timer package.

Timer interrupts are generated to indicate delayed add commands (i.e., ADD), and to indicate that the critical time has been reached for a thread within a hard deadline (i.e., ABRT).

If the timer event signals a delayed add, then a standard add is queued up to be handled at the main-loop level.

If the event indicates a critical time has been reached, an abort event is queued up.

In any event, this routine takes the 'Sev' given by the interval timer and queues up the event indicated by the token (this means that the oken is always a pointer to a 'Sev').

Routine:

```
long    BEAccessSIIntr(SEVptr)
        sevptr      SEVptr;
```

Interrupt-level access scheduling information routine.

All types of access commands are presently handled in the main loop, so just queue the command up.

Routine:

```
long    BEAccessSI(SEVptr)
        sevptr      SEVptr;
```

User-level access scheduling information routine.

This routine allows the scheduling information kept for a thread to be accessed.

Basically, there are two types of accesses that are done here — reads and writes. In addition, there are normal and abort accesses.

This routine just demultiplexes the commands and calls the proper routine.

It is the responsibility of the called routine to dealloc the SEV.

3.2.3.4 Best Effort Scheduling Policy Utility Routines — be_util.c

This C source file contains logic for utility routines for the BE scheduler.

Routine:

```
double      BEValue(TCptr, t)
        TCspcbptr   TCptr;
        double      t;
```

Compute the completion value of the thread at a particular point in time (as given in the value of 't', which is in SPTICKS).

Routine:

```
double      BEEstimatedValue(TCptr, t)
        TCspcbptr   TCptr;
        double      t;
```

Compute estimated completion value of thread.

This is done by computing the probability that a random variable is greater than the total computation time received so far by the thread, and seeing if that probability is small.

If it is, then it is assumed that the thread will not need any more cycles to complete and the estimated value is equal to that accrued if the thread completed now.

If there is a significant probability that the thread will take more cycles to complete, the estimated value is computed by computing the area under the value function within the range of possible completion times, and it is scaled against the probability densities of these completion times based on the computation time distribution.

This routine assumes that the given thread begins/continues execution at time 't' (in SPTICKS).

Routine:

```
u_long      BEEarliestTime(TCptr, val)
        TCspcbptr   TCptr;
        double      val;
```

Routine to compute the earliest time at which this thread should be executed to get a value at least 'val'.

Only 'Vearly' function is considered here: this is monotonically increasing, or else constant.

Routine:

```
void   DoLoadShed(curtime, nkeptptr, tslackptr, tslackvarptr, ctleftptr)
       abstime      curtime;
       int          *nkeptptr,
                    *tslackptr,
                    *tslackvarptr;
       float        *ctleftptr;
```

This routine removes threads from the BE queue, starting with the one of smallest value density and moving up, until the probability of overload has gone below the overload probability threshold.

Routine:

```
void   Dispatch()
```

Dispatch the next thread to run.

This routine looks at the ready queues to find the next thread to be run and, if it finds one, it dispatches it.

The general approach here is to take the thread on the top of the BE queue and run it. The reschedule routine ensures that the BE list is kept in the proper order, with the right members.

In the event that this routine is called and the the BE queue is empty, we check the DL queue, just in case there are more threads that have been load shed. When both of these queues are empty, this routine does nothing, except set the idle time counter and then return.

The actual setting of the global 'currentthreadptr' value is deferred until the dispatch is acknowledged. This routine only sets the 'preemptthread' global variable and the 'preemptinprog' flag, 'currentthreadptr' is set by a preemption confirmation command, and is handled under interrupt.

If there is a preemption in progress when we try to do a dispatch, this routine spins until the preemption completion is signalled (at the interrupt level).

Also, if the thread chosen to be dipatched is the currently running one, then this routine does nothing.

Routine:

```
void   ChangeTC(SEVptr, TCBptr)
       sevptr       SEVptr;
       BEtcbptr     TCBptr;
```

Change Time Constraints.

This was originally intended to provide a general-purpose way of changing a thread's time constraints. However, at present all that may be changed is the thread's importance.

All that is necessary is for the thread's importance to be changed, and for it to be removed from the importance queue and reinserted with the new value, if it was found there. A reschedule is always necessary afterwards.

Routine:

```
void  PushTC(SEVptr, TCBptr)
      sevptr      SEVptr;
      BEtcbptr    TCBptr;
```

Push a time constraint.

In the case of pushing time constraints, we check to see whether we are acting under abort time constraints or the new time constraint is not stricter than the current one. In these two cases the new time constraints are irrelevant; nothing else need be done. Otherwise, allocate a new TC for the new time constraints, fill it out, place it into the appropriate thread queue (importance or deadline), and queue a reschedule.

Routine:

```
void  PopTC(SEVptr, TCBptr)
      sevptr      SEVptr;
      BEtcbptr    TCBptr;
```

Pop a time constraint.

Popping time constraints involves first checking to make sure that the abort level known by the SP is the same as that provided by the AP in the scheduling information block. If they are not the same, then signal a fatal error. Popping time constraints could have made the thread not have any time constraints at all, in which case it should be placed in the importance queue. It could also back to a previous time constraint, in which case it must be placed into the deadline queue at a new place. If the previous time constraint was a hard deadline, the timer event that was queued for it should be canceled.

Queues a reschedule when finished.

Routine:

```
void  DumpSI(SEVptr, TCBptr)
      sevptr      SEVptr;
      BEtcbptr    TCBptr;
```

Disgorge a thread's scheduling information into a buffer in the AP.

This routine dumps the SP's scheduling information into a buffer so that the AP can send it off to another node along with a remote invocation. The AP provides the buffer and is responsible for mapping it into DVMA space so that the SP can access it. The interrupt handler has already taken care of translating the pointer to the dump block into the SP's address space.

This code begins by dumping the parts of the scheduling parameters that are common to all threads. Then it checks if there is an active time constraint block, and if so, this is dumped too.

Since this can only be done by the the currently executing thread, we have to get the latest update of the accumulated computation time for the thread. This is used to give the surrogate thread good estimates of the deadline and accumulated execution time. The actual run-time stats are accumulated when the thread actually blocks (it's still running at the point it requests this info).

If there is an active time constraint, the routine dumps the needed information from the TC into the given buffer.

The routine then sends a command to the AP to indicate that the dump has been completed.

Routine:

```
void   UpdateSI(SEVptr, TCBptr)
       sevptr       SEVptr;
       BEtcbptr     TCBptr;
```

Load a thread's scheduling information from a buffer in the AP.

This routine is used to update a thread's scheduling information from another its (surrogate) execution on another node. The information it uses is the result of a Dump operation done on another node's SP.

This is done when a thread returns from a remote invocation.

This routine updates the thread's time dependent information — i.e., the amount of computation time it received on a remote node, and, if the thread is under a time constraint, the amount of time remaining in the current time constraint.

The thread stays in its ready queue position if it is under a time constraint. If it is not, then we check if the importance has changed, and if it has we put it in its proper place in the importance queue, and then queue up a reschedule.

Routine:

```
void   GetAbortSI(SEVptr, TCBptr)
       sevptr       SEVptr;
       BEtcbptr     TCBptr;
```

Get a thread's scheduling information for an abort.

This routine dumps the SP's scheduling information into a buffer that is given to other thread sections doing aborts so that they can do their clean-up with the proper scheduling parameters. The buffer is part of the incoming command block (SEV) and the scheduling information is passed back in the SEV via the IPI channel.

This code begins by dumping the parts of the scheduling parameters that are common to all threads. Then it checks if there is an active time constraint block, and if so, this is dumped too.

Since this can only be done by the the currently executing thread, we have to get the latest update of the accumulated computation time for the thread. This is used to give the surrogate thread good estimates of the deadline and accumulated execution time. The actual run-time stats are accumulated when the thread actually blocks (it's still running at the point it requests this info).

If there is an active time constraint, the routine dumps the needed information from the TC into the given buffer.

The routine then sends a command to the AP to indicate that the dump has been completed.

Routine:

```
void   SetAbortSI(SEVptr, TCBptr)
       sevptr       SEVptr;
       BEtcbptr     TCBptr;
```

Load a thread's scheduling information for an abort.

This routine handles the loading of scheduling parameters for a thread that is in the process of aborting.

This routine begins by determining if the thread in question has any time constraints associated with it. If it does, the thread's TC list is deallocated, and then the thread is removed from any of the queues that it is in (i.e., DL or BE, because it is under a TC).

If the thread does not have any time constraints, it is removed from the IM queue (if it is in there).

This routine then updates the thread's common scheduling information, and then checks if there is a time constraint to inherit. If there is no TC to update, then the thread is put into the IM queue, if the thread is ready, and a reschedule is queued up.

If there is a TC as part of the update, a TC block is allocated and filled out with the information passed in the SEV. If it is a hard TC, then the new TC is also linked into the thread's hard TC list.

Then the thread is added to the DL queue, if it was already ready, and a reschedule is queued up. If the thread was not already in the ready queue, then a timed event is started for the new deadline and the thread is left as it is.

Routine:

```
static double       FunctionArea(treal, tcompute, tleft, vfptr)
        double      treal;
        double      tcompute;
        double      tleft;
        VFptr       vfptr;
```

The integral from 0 to 'tleft' of (VF(treal+t) * p(tcompute+t)) dt.

Returns (for the given thread) the area which is the integral from 0 to 'tleft' of valuefunction(-treal+t) * probability(tcompute+t) dt.

Result must be scaled by the 'yasf * value'.

'treal' must fall in either early f'n or late f'n, and is since the 'reqtime' or the 'crittime'.

Routine:

```
static double       ExpCase(treal, tcompute, tleft, valfunc)
        double      treal;
        double      tcompute;
        double      tleft;
        double      (*valfunc)();  either 'QuadValue()' or 'ExponValue()'
```

Value of integral of valuefunction(treal+t) * p(tcompute+t) dt.

Returns (for the given thread) the area which is the integral from 0 to 'tleft' of valuefunction(-treal+t) * probability(tcompute+t) dt.

Result must be scaled by the 'yasf * value'.

'treal' must fall in either early f'n or late f'n, and is relative to either 'reqtime' or 'crittime'.

Uses a general method for any time and function. (Accuracy: low to fair) .

Routine:

```
static double       QuadValue(t)
        double      t;
```

Compute the completion value of the thread.

’t’ is time relative to ’reqtime’ or ’crittime’ (in SPTICKS).

Assumes the global variable ’globalVFptr’ points to parms for a quadratic.

Returns value needing to be y-scaled.

Routine:

```
static double      ExponValue(t)
         double    t;
```

Compute the completion value of the thread.

’t’ is time relative to ’reqtime’ or ’crittime’ (in SPTICKS).

Assumes that the global variable ’globalVFptr’ points to parameters for an exponential.

Returns value needing to be y-scaled.

3.2.3.5 Deadline Scheduling Policy Debug Support — dl_debug.c

This C source file contains logic for Deadline Scheduling Policy Debugging.

Routine:

```
void   DLPrintRQ()
```

Print contents of ready queue.

Routine:

```
void   DLPrintTCB(TCBptr)
       DLtcbptr     TCBptr;
```

Print contents of thread control block.

Routine:

```
void DLPrintHWM()
```

Print High Water Marks.

3.2.3.6 Deadline Scheduling Policy — dl_pol.c

This C source file contains logic for the Deadline Scheduling Policy.

Routine:

```
void   DLInit()
```

Scheduler Initialization Routine.

This routine initializes the queues, the interval timer, and global variables. Also, print a start-up banner.

The statistics kept by the scheduler are auto-initialized.

Routine:

```
long   DLAbort(SEVptr)
       sevptr       SEVptr;
```

Main-loop-level abort handling routine.

This routine is called when a thread’s hard time constraint expires. It must be aborted in this case. It is also possible that an outer hard In the case where the abort block level recorded in the ’SEV’ structure is greater than that in the outermost TC. In that case, nothing need be done.

Otherwise, the TC block is removed and deallocated, and the AP will be asked to abort that time constraint.

For hard time constraints, first check to see if the innermost hard time constraint has expired. If it has not occurred, assume that the time constraint was removed before the timed event could be cancelled, and just print a warning message. Otherwise, remove the first hard time constraint that occurred after it was established.

Propagate the computation time statistics for each of these time constraints forward and deallocate the blocks. Finally, ask the AP to abort to the abort block level given in the time constraint whose timer expired.

If there are no time constraints left, move the thread to the importance queue. Otherwise, remove the thread from the deadline queue and replace it at the new point.

Finally, a new thread is dispatched if the aborted thread has gone to the top of the ready queue.

Routine:

```
long    DLAddIntr(SEVptr)
        sevptr        SEVptr;
```

Interrupt-level add task routine.

If this is a delayed add (i.e., DADD), pass it to the interval timer system to be delivered later. Otherwise, queue the 'Sev' up to be delivered to the main loop service process.

Both regular adds and surrogate add (i.e., SADD) commands are queued for the main loop.

Routine:

```
long    DLAdd(SEVptr)
        sevptr        SEVptr;
```

Main-Loop-Level add task routine.

This routine adds a new thread to the SP's ready mix. It results from a call to 'MakeReady()' in the kernel, and can be either a regular add command (i.e., ADD), or a command to add a surrogate thread (i.e., SADD) — i.e., a thread created on another node.

If this thread is new to the SP, allocate a TCB for it.

If, in addition to being a new thread, it is also a surrogate thread, pick up the inherited scheduling information from the scheduling information block contained in the 'SEV' (i.e., Scheduler Event Block). If this surrogate thread is under a time constraint, this picks up the scheduling information for the thread, and sets a timed event for the innermost deadline.

If the thread being added is not new to the scheduler, get the pointer to the thread's TCB from the 'SEV'. It is not necessary to set any timed events for time constraints in this case, because they are set when the TC is originally pushed.

At this point, we have a valid, properly initialized TCB.

If the thread to be added is not under a time constraint, queue it in the importance queue (using its importance as the key), otherwise, queue it in the deadline queue (using its critical time as the key).

Finally, a new thread is dispatched if the added thread has gone to the top of the ready queue.

Routine:

```
long    DLRemoveIntr(SEVptr)
        sevptr        SEVptr;
```

Interrupt-level remove task routine.

All types of remove commands are handled in the main loop, so just queue this command block up to be handled by the main loop.

Routine:

```
long    DLRemove(SEVptr)
        sevptr        SEVptr;
```

Main-Loop-Level remove task routine.

This routine removes a thread from the ready mix. This class of command can only come from the AP's current thread, but there are two possibilities to consider. The first case is the simple one — i.e., the currently executing thread has decided to block, so we just remove it from the ready queue(s) that it appears in, and dispatch the next thread to be run.

The other case to consider is when the current thread has decided to block before it received a recently issued preempt command. In this situation the SP has indicated that a preempt is in progress and has set the 'preeptthreadptr' variable to the preempting thread, but has not yet set the 'currentthreadptr' variable to what it thinks is the new running thread. The AP thinks the previous thread is the current thread.

If this routine is called and the thread to be removed is not the current one, we need to be cautious about blindly trying to remove the thread; it may or may not still be in a ready queue (because whatever caused it to be preempted, may have removed it from the queues already). This means that we use 'QKill2()' so it won't complain if a thread isn't in a queue.

In either of these two cases, the main thing that happens here is that the indicated thread is removed from all of the ready queues that it appears in, and the next thread to run is dispatched. Also, we update the blocking thread's ending run-time statistics. This means that we make note of the time at which the thread quit running (as defined by when the remove command was received), and update the necessary statistics.

There are two variants of this routine — in one case, the thread's TCB is just removed from the ready queue(s) that it appears in, and allowed to float around in the ether until it is added back to the ready mix, or killed. The AP keeps a pointer to the TCB around and

it includes it in the 'SEV' when the thread is added back. This event is indicated by the REM command, and is initiated by calling the kernel's 'Block()' and 'Sleep()' routines.

In the other case, the TCB is removed from the ready mix in the same way that is done in the previous case, and, at the end, the all of the data structures associated with the thread are deallocated. This case is indicated by the KILL command, and is initiated by calling the kernel's 'Suicide()' routine.

After the given thread is removed, this routine performs a dispatch operation to run the next thread in the ready queues (if there is one).

Routine:

```
long    DLChangeIntr(SEVptr)
        sevptr        SEVptr;
```

Interrupt-Level change scheduling parameters routine.

All types of changes are presently handled in the main loop, so just queue the command up.

Routine:

```
long    DLChange(SEVptr)
        sevptr        SEVptr;
```

Main-Loop-level change scheduling attributes routine.

This routine handles all requests that involve the manipulation of a thread's scheduling-related attributes — i.e., time constraint pushing (PUSH_SA), popping (POP_SA), and simple change requests (CHG_SA).

Requests to push or pop time constraints are handled by their own routines, defined separately in 'dl_util.c'.

'CHANGE_SEV' requests are solely used to change the importance of a thread. This request is handled by removing the thread from any queues it might be in, changing its importance, and reinserting it in any queues it was in originally. A dispatch is also necessary after this to see if another thread should be run.

Requests to push abort scheduling information are handled by removing the present set of time constraints (the TC blocks), removing the thread from any queues it might be in, changing the scheduling parameters of the thread (including possibly allocating new TC blocks and chaining them), requeueing the thread if appropriate, and enqueueing a reschedule event. It is the responsibility of the called routines to dealloc the SEV (if necessary).

Routine:

```
long    DLMiscIntr(SEVptr)
        sevptr        SEVptr;
```

Interrupt-level routine to handle miscellaneous commands.

This routine handles all of the misc commands associated with this specific policy. For this policy, the commands include the preempt confirmation commands (PRMT_CF and IPRMT_CF).

The preempt confirmation commands indicate that a requested preemption has been completed. This is a signal that the SP's notion of current thread is now valid and that thread is running.

This command is needed because the AP can take time to respond to a preempt command (due to such things as deferred preemption).

The confirmation of a preemption is a signal that the current thread is now running and it's starting run-time statistics should be updated.

In the case when the confirmation indicates that the AP preempted a thread other than the idle thread (i.e., PRMT_CF), the ending run-time statistics for the preempted thread must be updated (along with the starting run-time statistics for the preempting thread). This must be done because there is no REMOVE command associated with this activity, and it is that command that is usually responsible for updating the ending run-time statistics for a thread that is stopped.

The starting run-time statistics consist of the time that the thread was dispatched. The ending run-time statistics consist of the thread's total accumulated execution time, the time left until the thread's (currently defined) critical time, and the total computation time accumulated under the current time constraint. (Note that the latter two statistics need be gathered only if the thread is executing under a time constraint).

The other miscellaneous command (i.e., the statistics on/off commands (STATS_ON/STATS_OFF) are handled by the interrupt handler routine).

In all of these cases, global variables are set and nothing has to e done at the main-loop-level.

Care should be taken in all of the main-loop routines that access these global variables (i.e., make sure to synchronize when it is necessary).

Routine:

```
long    DLTimerIntr(SEVptr, TEtoken)
        sevptr        SEVptr;
        u_long        TEtoken;
```

Interrupt-level timer event routine.

This routine handles all of the interrupts generated by the interval timer package.

Timer interrupts are generated to indicate delayed add commands (i.e., ADD), and to indicate that the critical time has been reached for a thread within a hard deadline (i.e., ABRT).

If the timer event signals a delayed add, then a standard add is queued up to be handled at the main-loop level.

If the event indicates a critical time has been reached, an abort event is queued up.

In any event, this routine takes the 'Sev' given by the interval timer and queues up the event indicated by the token (this means that the token is always a pointer to a 'Sev').

Routine:

```
long    DLAccessSIIntr(SEVptr)
        sevptr        SEVptr;
```

Interrupt-level access scheduling information routine.

Alll types of access commands are presently handled in the main loop, so just queue the command up.

Routine:

```
long    DLAccessSI(SEVptr)
        sevptr        SEVptr;
```

User-level access scheduling information routine.

This routine allows the scheduling information kept for a thread to be accessed.

Basically, there are two types of accesses that are done here — reads and writes. In addition, there are normal and abort accesses. This routine just demultiplexes the commands and calls the proper routine.

It is the responsibility of the called routine to dealloc the SEV.

3.2.3.7 Deadline Scheduling Policy Utility Routines — dl_util.c

This C source file contains logic for the Utility Routines for the Deadline Scheduler.

Routine:

```
void   DLDispatch()
```

Dispatch the next thread to run — use the deadline policy.

This routine looks at the ready queues to find the next thread to be run and, if it finds one, it dispatches it.

The general approach here is to take the thread on the top of the DL queue and run it. The add routine ensures that the DL list is kept in the proper order, with the right members. In the event that this routine is called and the the DL queue is empty, this routine does nothing, except set the idle time counter and then return.

The actual setting of the global 'currentthreadptr' value is deferred until the dispatch is acknowledged. This routine only sets the 'preemptthread' global variable and the 'preemptinprog' flag, 'currentthreadptr' is set by a preemption confirmation command, and is handled under interrupt.

If there is a preemption in progress when we try to do a dispatch, this routine spins until the preemption completion is signalled (at the interrupt level).

Also, if the thread chosen to be dipatched is the currently running one, then this routine does nothing.

Routine:

```
void   DLChangeTC(SEVptr, TCBptr)
       sevptr          SEVptr;
       Ltcbptr         TCBptr;
```

Change Time Constraints.

This was originally intended to provide a general-purpose way of changing a thread's time constraints. However, at present all that may be changed is the thread's importance.

All that is necessary is for the thread's importance to be changed, and for it to be removed from the importance queue and reinserted with the new value, if it was found there. A dispatch is always necessary afterwards.

Routine:

```
void   DLPushTC(SEVptr, TCBptr)
       sevptr          SEVptr;
       DLtcbptr        TCBptr;
```

Push a time constraint.

In the case of pushing time constraints, we check to see whether we are acting under abort time constraints or the new time constraint is not stricter than the current one. In these two cases the new time constraints are irrelevant; nothing else need be done. Otherwise, allocate a new TC for the new time constraints, fill it out, place it into the appropriate thread queue (importance or deadline), and do a dispatch to run another thread (if necessary).

Routine:

```
void   DLPopTC(SEVptr, TCBptr)
       sevptr          SEVptr;
       DLtcbptr        TCBptr;
```

Pop a time constraint.

Popping time constraints involves first checking to make sure that the abort level known by the SP is the same as that provided by the AP in the scheduling information block. If they are not the same, then signal a fatal error. Popping time constraints could have made the thread not have any time constraints at all, in which case it should

be placed in the importance queue. It could also back to a previous time constraint, in which case it must be placed into the deadline queue at a new place. If the previous time constraint was a hard deadline, the timer event that was queued for it should be canceled.

This routine also performs a dispatch to run a new thread (if necessary).

Routine:

```
void   DLDumpSI(SEVptr, TCBptr)
       sevptr       SEVptr;
       DLtcbptr     TCBptr;
```

Disgorge a thread's scheduling information into a buffer in the AP.

This routine dumps the SP's scheduling information into a buffer so that the AP can send it off to another node along with a remote invocation.

The AP provides the buffer and is responsible for mapping it into DVMA space so that the SP can access it. The interrupt handler has already taken care of translating the pointer to the dump block into the SP's address space.

This code begins by dumping the parts of the scheduling parameters that are common to all threads. Then it checks if there is an active time constraint block, and if so, this is dumped too.

Since this can only be done by the the currently executing thread, we have to get the latest update of the accumulated computation time for the thread. This is used to give the surrogate thread good estimates of the deadline and accumulated execution time. The actual run-time stats are accumulated when the thread actually blocks (it's still running at the point it requests this info).

If there is an active time constraint, the routine dumps the needed information from the TC into the given buffer.

The routine then sends a command to the AP to indicate that the dump has been completed.

Routine:

```
void   DLUpdateSI(SEVptr, TCBptr)
       sevptr       SEVptr;
       DLtcbptr     TCBptr;
```

Load a thread's scheduling information from a buffer in the AP.

This routine is used to update a thread's scheduling information from another its (surrogate) execution on another node. The information it uses is the result of a Dump operation done on another node's SP.

This is done when a thread returns from a remote invocation.

This routine updates the thread's time dependent information — i.e., the amount of computation time it received on a remote node, and, if the thread is under a time constraint, the amount of time remaining in the current time constraint.

The thread stays in its ready queue position if it is under a time constraint. If it is not, then we check if the importance has changed, and if it has we put it in its proper place in the importance queue, and then queue up a reschedule.

Routine:

```
void    DLGetAbortSI(SEVptr, TCBptr)
        sevptr      SEVptr;
        DLtcbptr    TCBptr;
```

Get a thread's scheduling information for an abort.

This routine dumps the SP's scheduling information into a buffer that is given to other thread sections doing aborts so that they can do their clean-up with the proper scheduling parameters.

The buffer is part of the incoming command block (SEV) and the scheduling information is passed back in the SEV via the IPI channel.

This code begins by dumping the parts of the scheduling parameters that are common to all threads. Then it checks if there is an active time constraint block, and if so, this is dumped too.

Since this can only be done by the the currently executing thread, we have to get the latest update of the accumulated computation time for the thread. This is used to give the surrogate thread good estimates of the deadline and accumulated execution time. The actual run-time stats are accumulated when the thread actually blocks (it's still running at the point it requests this info).

If there is an active time constraint, the routine dumps the needed information from the TC into the given buffer.

The routine then sends a command to the AP to indicate that the dump has been completed.

Routine:

```
void    DLSetAbortSI(SEVptr, TCBptr)
        sevptr      SEVptr;
        DLtcbptr    TCBptr;
```

Load a thread's scheduling information for an abort.

This routine handles the loading of scheduling parameters for a thread that is in the process of aborting.

This routine begins by determining if the thread in question has any time constraints associated with it. If it does, the thread's TC list is deallocated, and then the thread is removed from any of the queues that it is in (i.e., DL, because it is under only a TC).

If the thread does not have any time constraints, it is removed from the IM queue (if it is in there).

This routine then updates the thread's common scheduling information, and then checks if there is a time constraint to inherit. If there is no TC to update, then the thread is put into the IM queue, if the thread is ready, and a new thread is dispatched (if necessary).

If there is a TC as part of the update, a TC block is allocated and filled out with the information passed in the SEV. If it is a hard TC, then the new TC is also linked into the thread's hard TC list.

Then the thread is added to the DL queue, if it was already ready, and 'DLDispatch()' is called. If the thread was not already in the ready queue, then a timed event is started for the new deadline and the thread is left as it is.

3.2.3.8 FIFO Scheduling Policy Debug Support — fifo_debug.c

This C source file contains interactive debugging routines which are enabled when the FIFO scheduling policy has been selected.

Routine:

```
void   FIFOPrintRQ()
```

Print the contents of the ready queue.

Routine:

```
void   FIFOPrintTCB(TCBptr)
       FIFOtcbptr    TCBptr;
```

Print the contents of a thread control block.

3.2.3.9 FIFO Scheduling Policy — fifo_pol.c

Routine:

```
void   FIFOInit()
```

This routine initializes the ready queue, the run-time statistics, and the timed events.

Routine:

```
long   FIFOAddIntr(SEVptr)
       sevptr        SEVptr;
```

Interrupt-level add task routine.

This routine checks if the add command is immediate or delayed. If it is, it queues the event to be handled in background, otherwise a timed event is started up and the Sev is passed to the interval timer as a token.

Routine:

```
long   FIFOAdd(SEVptr)
       sevptr        SEVptr;
```

User-Level add task routine. This does the actual add of a TCB to the Ready Queue.

This first checks if this is a new thread, and if so creates a new data structure for it, else, it uses the exisiting one. Then it adds the thread's TCB to the Ready Queue and checks if this is the only thing in it (i.e., it was empty before we added the thread), and if so, do a preempt to start this thread up.

This also sends a command back to the AP indicating the added TCB's identifier (on the SP).

Routine:

```
long   FIFORemoveIntr(SEVptr)
       sevptr        SEVptr;
```

Interrupt-level remove task routine.

Queues event to be handled in background.

Routine:

```
long    FIFORemove(SEVptr)
        sevptr        SEVptr;
```

This routine removes a task from the ready queue, at user level.

It checks to see if the task being removed is the one currently running, and if it is, see if there is another one to run, and if there is, do a preempt and run it.

Routine:

```
long    FIFOTimerIntr(SEVptr, TEtoken)
        sevptr        SEVptr;
        u_long        TEtoken;
```

Interrupt-level timer event routine.

This routine gets delayed commands from the interval timer and queues them up for the background processor

3.2.3.10 Interrupt Handler — intrhdlr.c

This C source file contains logic for the interrupt handler.

Routine:

```
void    IntrHdlr(level)
        u_long        level;
```

This routine is called by the assembly language generic interrupt handler, and passed the level of the interrupt. Based on this information, we handle it.

3.2.3.11 Queue Package — queue.c

This C source file contains logic for the queue package.

Implements queues as doubly-linked lists. Each queue has a head and a tail. The queue goes from head to tail, with either the higher-valued integer keys towards the tail or towards the head.

Routine:

```
void    QInit(pQe, pHeadItem, pTailItem, sw)
        qendsptr      pQe;
        vaddr         pHeadItem;
        vaddr         pTailItem;
        int           sw;
```

Initialize a queue.

Routine:

```
long    QEmpty(pQe)
        qendsptr      pQe;
```

Determine if the queue is empty.

Routine:

```
void   NQHead(pQe, pQElt)
       qendsptr    pQe;
       qptr        pQElt;
```

Enqueue an item at the head of a queue .

Routine:

```
void   NQTail(pQe, pQElt)
       qendsptr    pQe;
       qptr        pQElt;
```

Enqueue an item at the tail of a queue.

Routine:

```
qptr   DQHead(pQe)
       qendsptr    pQe;
```

Dequeue an item from the head of a queue.

Routine:

```
qptr   DQTail(pQe)
       qendsptr    pQe;
```

Dequeue an item from the tail of a queue.

Routine:

```
void   QInsertB(pQe, pQElt, sw)
       qendsptr    pQe;
       qptr        pQElt;
       int         sw;
```

Insert item in queue (before other equal keys).

Routine:

```
void   QInsertA(pQe, pQElt, sw)
       qendsptr    pQe;
       qptr        pQElt;
       int         sw;
```

Insert item in queue (after other equal keys).

Routine:

```
qptr   QDelete(pQHead, item)
       qptr        pQHead;
       int         item;
```

Delete an item (with a unique key) and return an item.

Routine:

```
void  QKill(pQe, pQElt)
      qendsptr   pQe;
      qptr       pQElt;
```

Delete an item from a queue.

Routine:

```
long  QKill2(pQe, pQElt)
      qendsptr   pQe;
      qptr       pQElt;
```

Same as QKill() but return a status indicating success or failure.

Routine:

```
void  QSwap(pQ1, pQ2)
      qptr  pQ1;
      qptr  pQ2;
```

Swap the positions of two queue items.

Routine:

```
void  QAdjSwap(pQ1, pQ2)
      qptr  pQ1;
      qptr  pQ2;
```

Swap the positions of two (adjacent) queue items.

Routine:

```
void  QPush(pQe, pQ)
      qendsptr   pQe;
      qptr       pQ;
```

Push an item (for the Q-based stack implementation).

Routine:

```
qptr  QPop(pQe)
      qendsptr   pQe;
```

Pop an item (for the Q-based stack implementation).

Routine:

```
qptr  QTos(pQe)
      qendsptr   pQe;
```

Return the top of the stack (for the Q-based stack implementation).

Routine:

```
int   QSize(pQe)
      qendsptr   pQe;
```

Return the size of the queue.

3.2.3.12 Round Robin Scheduling Policy Debug Support — rr_debug.c

This C source file contains logic for the Round Robin Scheduling Policy Debugger.

Routine:

```
void   RRPrintRQ()
```

Print the contents of the ready queue.

Routine:

```
void   RRPrintTCB(TCBptr)
       RRtcbptr      TCBptr;
```

Print the contents of a thread control block.

3.2.3.13 Round Robin Scheduling Policy — rr_pol.c

This C source file contains logic for the Round Robin scheduling policy.

Routine:

```
void   RRInit()
```

This routine initializes the ready queue, the run-time statistics, and the timed events.

Also, start the time-slice interval timer.

Routine:

```
long   RRAddIntr(SEVptr)
       sevptr        SEVptr;
```

Interrupt-level add task routine.

This routine checks if the add command is immediate or delayed. If it is, it queues the event to be handled in background, otherwise a timed event is started up and the Sev is passed to the interval timer as a token.

Routine:

```
long   RRAdd(SEVptr)
       sevptr        SEVptr;
```

User-Level add task routine.

This does the actual add of a TCB to the Ready Queue. This first checks if this is a new thread, and if so creates a new data structure for it, else, it uses the exisiting one. Then it adds the thread's TCB to the Ready Queue and checks if this is the only thing in it (i.e., it was empty before we added the thread), and if so, do a preempt to start this thread up.

This also sends a command back to the AP indicating the added TCB's identifier (on the SP).

Routine:

```
long   RRRemove(SEVptr)
       sevptr        SEVptr;
```

User-Level remove task routine.

This routine removes a task from the ready queue.

It checks to see if the task being removed is the one currently running, and if it is, see if there is another one to run, and if there is, do a preempprempt and run it.

There are two variants of this command — the normal remove, and the kill, which removes all of the SP's state for the thread.

Routine:

```
long  RRTimerIntr(SEVptr, TEtoken)
      sevptr       SEVptr;
      _long        TEtoken;
```

Interrupt-level timer event routine.

This handles the interrupts from the interval timer.

This just queues up an event to be handled in the background This timer interrupt can be a result of the periodic preemption timer, or from a delayed add.

If it is the periodic preempt timer, the interrupt Sev is queued up, if it is a delayed add, the Sev passed by the token is queued up.

If the event is from the periodic timer, another timed event is started up. This is done under interrupt in an attempt to keep the intervals somewhat uniform.

It is expected that the init routine starts up the first periodic timed event.

Routine:

```
long  RRTimer(SEVptr)
      sevptr       SEVptr;
```

User level handler for timer events.

This checks if there is more than one thread in the ready queue, and if so moves the currently executing thread to the end of the ready queue and preempts its execution on the AP, replacing it with the new head of the ready queue.

The init and timer interrupt handling routines make sure that the periodic timer is kept running.

3.2.3.14 Scheduler Main Program — scheduler.c

This C source file contains logic for the main program of the Scheduling processor.

Routine:

```
void  SAInit()
```

Initialize the scheduling subsystem.

This routine is called from the system initialization routine, with interrupts disabled.

This is the main program for the scheduling processor. First, it initializes the scheduler storage allocator and then calls the appropriate init routine for the current scheduling policy used.

Once initialization is done, it will enable interrupts and wait in an infinite loop until an event occurs, and then service the interrupt by calling the appropriate scheduling routines.

The events are obtained from the scheduling event queue which is used by interrupt handling routines to schedule an event occurrence.

Note that the stacking of scheduling events by the interrupt handling routines are entirely up to each individual scheduling policy. Of course, in case of trivial scheduling policies like FIFO, one can simply do everything at the interrupt level and this routine simply ends up behaving like a idle loop.

Routine:

```
void   HandleEvent(SEVptr)
       sevptr       SEVptr;
```

Handle scheduling events in the foreground.

Do the non-interrupt level portion of the processing dictated by the scheduling policy for each event that got queued.

Routine:

```
void   SevNQ(SEVptr)
       sevptr       SEVptr;
```

Add an event to the scheduling event queue.

Routine:

```
sevptr       SevDQ()
```

Remove an event from the scheduling event queue.

3.2.3.15 Interactive Debugger — sdebug.c

This C source file contains logic for debugging the Scheduling Subsystem.

Routine:

```
void   SDBug()
```

This routine provides a common, interactive user interface to a collection of application-specific debugging routines.

The main debugging aid provided by this routine involves the printing of various data structures.

This routine also provides access to the routine that prints out the events in the log area.

The interface is menu-driven and an attempt is made to make the printing be "screen-oriented". At first the page size is given and a "more"-like function is provided: it prints a screen, prompts and waits for a response. Currently legal responses are: 'q' to quit printing, 'CR' for one more line, and any other character to print another full screen. Also, an 's' stops the paging (streaming the output), and any character typed at this point puts it back into paged mode.

This is not intended to be a real debugger. It is also a machine-level debugger, since this is available on the ROMs at all times.

Routine:

```
void   PrintEvents()
```

Print all events in the event log buffer onto the console.

Each event is printed out on a separate line starting with the event number, followed by the event name (a string), the event type and parameter (both single byte integers), the event's timestamp, and finally the delta off of the last event (in decimal microseconds, with the event logging overhead deducted).

The event buffer wraps around, so it is possible that the first event printed is not event #0.

This routine uses the monitor's busy-wait I/O functions.

Routine:

```
void   PrintSEV()
```

Debug print routine, to be called from debugger.

Print out the scheduling event queue statistics.

3.2.3.16 Event Logging — sp_event.c

This C source file contains logic for Event Logging.

Routine:

```
void   InitEvents()
```

Initialize the event logging area, initialize the event timer, and log these actions as the first event. The event buffer logging area is in the segment at the top of physical memory.

Routine:

```
void   LogEventIntf(id, parm)
       u_char       id;
       u_char       parm;
```

This routine is used to log an event. It uses the 'eventcount' global variable as an index into the event log buffer, it gets the current value of the event timer and puts it, and the given parameter, into the event record indexed by 'eventcount'.

Interrupts must be disabled during this routine because we have to maintain the ordering of the events in the log (an interrupt could result in another event being logged).

3.2.3.17 Interval Timer — sp_itimer.c

This C source file contains logic for the Scheduling Processor Interval Timer.

Routine:

```
void   InitTEvents()
```

This routine initializes the timed event facility, by clearing the event structures and setting up the initial interval timer values.

Routine:

```
short  CreateTEvent(interval, token)
       u_long       interval;
       u_long       token;
```

This routine creates a timed event. After the given interval of time has elapsed, the specified token is delivered to the timer interrupt handler.

The returned timed event ID can be used with the 'CancelTEvent()' routine to remove events that have not already occurred.

If no new timed events can be created, a fatal error occurs.

If the event is to be set for an interval less than the minimum value allowed, the event is generated immediately, and this routine returns a NIL.

Note that it is assumed that the 'interval' parameter is given in SPTICKS (whatever that is), however, the internal parts of the interval timer works on F2TICKS.

Routine:

```
void   CancelTEvent(event)
       short          event;
```

This routine cancels the timed event indicated by the ’event’ argument.

The validity of the event ID is checked before any thing is done, and a fatal error is generated if it isn’t valid.

Nothing happens if the indicated event is not active at the time.

This routine uses F2 ticks exclusively.

Routine:

```
void   ITimerIntrHdlr()
```

This routine handles interrupts generated by the interval timer.

It reads the current time, checks if this is just a counter roll-over event, and if not, it issues tokens for the timed events scheduled for times within a given window of the current time, removes them from the timed event list, and sets up the interval timer for the next event in the list.

This routine uses F2 ticks exclusively.

Routine:

```
u_int   GetF2Time()
```

Read current time from a clock running at TC_F2 frequency.

This routine maintains a clock running at the TC_F2 frequncy of the timer chip. It derives the current time from the system clock and knowledge of the system timer’s clock frequency. If the system clock runs at a slower frequency than TC_F2, a message is printed stating that the frequency is too slow and garbage is returned.

(Alternatively, the slower clock’s time could be shifted, providing a course-grained timer for some (fairly short) length of time.)

3.2.3.18 Policy Indirection Table to Debuggers — spdit.c

This C source file contains an indirection table used for accessing the debugger functions of different scheduling policies. The table is constructed statically, and empty slots by default point to an error-message function.

Variables:

```
spdit:
```

3.2.3.19 Policy Indirection Table to Schedulers — spit.c

This C source file contains an indirection table used for accessing different scheduling policies. The table is constructed statically, and empty slots by default point to an error-message function.

Variables:

```
ispit
spit
```

3.2.3.20 Statis Priority Scheduling Policy Debug Support — spri_debug.c

This C source file contains logic for the Static Priority Scheduling Policy Debugger.

Routine:

```
void   SPRIPrintRQ()
```

Print contents of ready queue.

Routine:

```
void   SPRIPrintTCB(TCBptr)
       SPRItcbptr    TCBptr;
```

Print contents of thread control block.

3.2.3.21 Static Priority Scheduling Policy — spri_pol.c

This C source file contains logic for the Static Priority Scheduling Policy.

Routine:

```
void   SPRIInit()
```

This routine initializes the ready queue, the run-time statistics, and the timed events.

Routine:

```
long   SPRIAddIntr(SEVptr)
       sevptr          SEVptr;
```

This routine checks if the add command is immediate or delayed.

If it is, it queues the event to be handled in background, otherwise a timed event is started up and the Sev is passed to the interval timer as a token.

Routine:

```
long   SPRIAdd(SEVptr)
       sevptr          SEVptr;
```

This does the actual add of a TCB to the Ready Queue.

This first checks if this is a new thread, and if so creates a new data structure for it, else, it uses the exisiting one. Then it adds the thread's TCB to the Ready Queue.

The head of the ready queue is checked and if the newly added TCB appears at the head (this means that the queue was empty before we added the thread or that the new thread is now the most important), then we do a preempt to start this thread up.

This also sends a command back to the AP indicating the added TCB's identifier (on the SP).

Routine:

```
long   SPRIRemove(SEVptr)
       sevptr          SEVptr;
```

This routine removes a task from the ready queue.

It checks to see if the task being removed is the one currently running, and if it is, see if there is another one to run, and if there is, do a preempt and run it.

Routine:

```
long   SPRIChangeIntr(SEVptr)
              sevptr          SEVptr;
```

Interrupt-Level change scheduling parameters routine.

Queues event to be handled in background.

Routine:

```
long   SPRIChange(SEVptr)
       sevptr        SEVptr;
```

User-level change scheduling information routine.

This routine changes the importance of an existing thread. If the indicated thread is currently running and its importance is reduced so that it is no longer at the top of the ready list, a preempt is done and the new head of the ready queue is run.

A preempt is also done if the indicated thread's priority is raised to the point where it now becomes the head of the ready queue.

This routine only responds to requests to change the importance outright — pushes and pops of time constraints are ignored.

An error is indicated if the indicated thread does not exist in the ready queue.

Routine:

```
long   SPRITimerIntr(SEVptr, TEtoken)
       sevptr        SEVptr;
       u_long        TEtoken;
```

Interrupt-level timer event routine.

This routine gets delayed commands from the interval timer and queues them up for the background processor.

Routine:

```
long   SPRIAccessSIIntr(SEVptr)
       sevptr        SEVptr;
```

Interrupt-level access scheduling information routine.

Just queue it up to be handled in the background.

Routine:

```
long   SPRIAccessSI(SEVptr)
       sevptr        SEVptr;
```

User-level access scheduling information routine.

This routine allows the information kept in the common scheduling block to be accessed.

3.2.3.22 System Timer — stimer.c

This C source file contains logic for the System Timer (For SUN 2.0 and 1.5).

Routine:

```
void   InitSTimer()
```

Initialize the system timer and start it running.

Typically, the AMD9513's timer 3 and 4 are cascaded to form a 32 bit system timer. For a SUN 1.5 that is short of timers, timer 4 is used as the low part of the 32 bit system timer.

In the case of being short of timers, timer 4 is set up to generate an interrupt on each terminal count (TC). This interrupt is used to update a 16 bit value in memory that contains the high 16 bits of the (logically 32 bit) system clock.

Routine:

u_int GetSTime()

This routine returns the current time from the system timer.

Routine:

void STimerIntrHdlr()

Handle a system timer interrupt.

This routine handles interrupts generated by the interval timer.

The purpose is to increment the high part of the system timer that is kept in software.

3.2.3.23 Best Effort Scheduling Policy Value Function Support — vf.c

This C source file contains variables defining the value function templates

Variables:

VFEarlyDb
VFLateDb

3.3 Communications Subsystem

The communications subsystem runs on a separate processor, referred to as the CP. Therefore, it contains startup code, and must initialize timers, etc. As a consequence, many of its files are in fact shared with other subsystems, and are described elsewhere.

3.3.1 Assembler Files

The communications subsystem shares the file "crt0.a" with the standalone code, and shares the file "util.a" with the kernel proper.

3.3.1.1 Exception Handler Stubs — excep.a

This file contains stubs for interrupt handlers.

Routines:

_APIIntr	API Interrupt Handler Stub (L2)
_EnetIntr	Ethernet Controller Interrupt Handler Stub (L3)
_UARTIntr	UART Interrupt Handler Stub (L5)
_STimerIntr	System Timer Interrupt Handler Stub (L4/L5)
_ITimerIntr	Interval Timer Interrupt Handler Stub (L6/L5)
_DummyIntr	Dummy interrupt handler

3.3.2 Include Files

3.3.2.1 Memory Management Constants — CVMMcons.h

This include file defines the constants for the Communication Virtual Machine Memory Management.

Constants:

SMM_BASE

SMM_SIZE
SMM_LIMIT
NUM_BLK_SIZES
MIN_BLK_INDX
MAX_BLK_INDX
MIN_BLK_SIZE
MAX_BLK_SIZE
MAX_BLK8_SIZE
MAX_BLK16_SIZE
MAX_BLK32_SIZE
MAX_BLK64_SIZE
MAX_BLK128_SIZE
MAX_BLK256_SIZE
MAX_BLK512_SIZE
MAX_BLK1024_SIZE
MAX_BLK2048_SIZE
MAX_BLK2048_SIZE
MAX_BLK4096_SIZE
MAX_BLK4096_SIZE
ZERO
NOZERO
MEM_ALLOC
MEM_BAD_SIZE

Macros:

BV_MAX_SIZE
BV_WORD_INDX
BV_BIT_INDX

3.3.2.2 Protocol-Specific Constants — CVMPcons.h

This include file defines the Protocol-Specific constants for the Communication Virtual Machine.

Constants:

RQST_TYPE
RPLY_TYPE
INV_RQST
INV_RPLY
AP_ADD_ABID
AP_REMOVE_ABID
AP_OUTRQST
AP_OUTRPLY

AP_INRQST
AP_INRPLY
OBI_PTYPE
TT_PTYPE
RBI_PTYPE
RM_PTYPE
BB_PTYPE
TMAR_PTYPE
PT_PTYPE
MAN_PTYPE
EXT_PTYPE
RQST_HDR_SIZE
RPLY_HDR_SIZE
PARMSIZE
CAPASIZE

Macros:

OBI_PROTOCOL
TIMER_TEST_PROTOCOL
RBI_PROTOCOL
RM_PROTOCOL
BB
TMAR_PROTOCOL
PT_PROTOCOL
MAN_PROTOCOL
EXT_PROTOCOL
AP_ADD_ABID_TKN
AP_REMOVE_ABID_TKN
AP_OUTRQST_TKN
AP_OUTRPLY_TKN
AP_INRQST_TKN
AP_INRPLY_TKN

3.3.2.3 Standard Protocol Specific Constants — CVMSPcons.h

This include file defines the Standard Protocol constants for the Communication Virtual Machine.

Constants:

STARTUP
MY_ENET_ADDR
NULL_TKN_TYPE
MAX_PREDEF_NULL_TKN_TYPES

MAX_PREDEF_UART_TKN_TYPES
MAX_PREDEF_RCVDPKT_TKN_TYPES
MAX_PREDEF_SENTPKT_TKN_TYPES
MAX_PREDEF_TEVENT_TKN_TYPES
MAX_PREDEF_APINTF_TKN_TYPES
MAX_PREDEF_INTERNAL_TKN_TYPES

3.3.2.4 Set Facilty Constants — CVMScons.h

This include file defines the Set Facility constants for the Communication Virtual Machine.

Constants:

MAX_SET_SIZE
MAX_EXPSET_SIZE

3.3.2.5 Timed Event Constants — CVMTcons.h

This include file defines the Communication Virtual Machine Timed Event Constants.

Constants:

MAX_T_EVENTS
TE_COMP_TIME
TE_MIN_INTERVAL
TMR_CREATE
TMR_CANCEL
START_FREQ
FREQ_UNIT
MAX_FREQ
SYSTIMER_IS_F1_MASK

3.3.2.6 Communications Virtual Machine Constants — CVMcons.h

This include file defines the constants for the Communication Virtual Machine.

Constants:

TQ_MAX_SIZE
DIR_MAX_SIZE
TS_MAX_SIZE
FATAL
API_MECH
BUF_MECH
FLT_MECH
MAP_MECH
MEM_MECH
PKT_MECH
TMR_MECH
TKN_MECH

3.3.2.7 Multicast Address Filter Constants — SEFcons.h

This include file defines the constants for the Multicast Address Filter.

Constants:

MAX_MCA_KEYS
MAX_MCA_PENTRIES
MAX_MCA_SENTRIES
MAX_MCA_ENTRIES
FLT_ADD_MCA

3.3.2.8 SUN Ethernet Controller Constants — SEcons.h

This include file defines the constants for the Low-level Sun Ethernet Controller.

Constants:

EN_MINLEN
EN_MAXLEN
EN_HDRLEN
MIN_PKT_SIZE
MAX_PKT_SIZE
O_NIL
A_NIL
SEB_PAGE_SIZE
SEB_MEM_SIZE
SEB_PM_SIZE
SEB_PROM_SIZE
SEB_NUM_PAGES
SEB_MEM_SEGS
SEB_REG_SEGS
SEB_MEM
SEB_REG
SEB_MEM_BASE
SEB_REG_BASE
SEB_MEM_PM_BASE
SEB_REG_PM_BASE
SEB_MEM_MB_BASE_INDX
SEB_REG_MB_BASE_INDX
SEB_PM_SWAP
SEB_PM_P2MEM
SEB_PM_INDEX
SEB_RESET
SEB_LOOPBACK
SEB_CHAN_ATTN

SEB_INTR_ENBL
SEB_PRTY_ENBL
SEB_PARITY
SEB_INTERRUPT
SEB_EXPN_ENBL
SEB_OBM_SIZE
SEB_MB_BASE
NUM_CB
NUM_TB
TB_SIZE
NUM_RFD
NUM_RB
RB_SIZE
SYSBUS16
SYSBUS8
ISCP_BUSY
SCB_STAT
SCB_CUS
SCB_RUS
STAT_CX
STAT_FR
STAT_CNA
STAT_RNR
CUS_IDLE
CUS_SUSP
CUS_ACTV
RUS_IDLE
RUS_SUSP
RUS_NORES
RUS_READY
CMD_ACK
CMD_CUC
CMD_RESET
CMD_RUC
ACK_CX
ACK_FR
ACK_CNA
ACK_RNR
CUC_NOP
CUC_START

CUC_RESUME
CUC_SUSPEND
CUC_ABORT
RUC_NOP
RUC_START
RUC_RESUME
RUC_SUSPEND
RUC_ABORT
MC_LIST_SIZE
CODE_NOP
CODE_IASETUP
CODE_CONFIGURE
CODE_MCSETUP
CODE_TRANSMIT
CODE_TDR
CODE_DUMP
CODE_DIAGNOSE
C
B
OK
A
EL
S
I
CMD
NOP_C
NOP_B
NOP_OK
NOP_EL
NOP_S
NOP_I
IASETUP_C
IASETUP_B
IASETUP_OK
IASETUP_A
IASETUP_EL
IASETUP_S
IASETUP_I
CONF_C
CONF_B

CONF_OK
CONF_A
CONF_EL
CONF_S
CONF_I
CONF1_ELB
CONF1_ILB
CONF1_PRMB_LEN
CONF1_AL_LOC
CONF1_ADDR_LEN
CONF1_SAV_BF
CONF1_SRDY
CONF1_PLEN2
CONF1_PLEN4
CONF1_PLEN8
CONF1_PLEN16
CONF1_AL_INBUF
CONF1_ALEN1
CONF1_ALEN2
CONF1_ALEN3
CONF1_ALEN4
CONF1_ALEN5
CONF1_ALEN6
CONF1_ALEN7
STD_CONF1
CONF2_INTF_SPC
CONF2_BOF_MET
CONF2_ACR
CONF2_LIN_PRIO
STD_CONF2
CONF3_RET_NUM
CONF3_SLT_TIM
STD_CONF3
CONF4_CDT_SRC
CONF4_CDTF
CONF4_CRS_SRC
CONF4_CRSF
CONF4_PAD
CONF4_BT_STF
CONF4_CRC16

CONF4_NCRC_INS
CONF4_TONO_CRS
CONF4_MAN_NRZ
CONF4_BC_DIS
CONF4_PRM
STD_CONF4
MCSETUP_C
MCSETUP_B
MCSETUP_OK
MCSETUP_A
MCSETUP_EL
MCSETUP_S
MCSETUP_I
MC_COUNT
TRANS_C
TRANS_B
TRANS_OK
TRANS_A
TRANS_S10
TRANS_S9
TRANS_S8
TRANS_S7
TRANS_S6
TRANS_S5
TRANS_MAX_COLL
TRANS_EL
TRANS_S
TRANS_I
TDR_C
TDR_B
TDR_OK
TDR_EL
TDR_S
TDR_I
TDR_1_LINK_OK
TDR_1_XCVR_PRB
TDR_1_ET_OPN
TDR_1_ER_SRT
TDR_1_TIME
DUMP_C

DUMP_B
DUMP_OK
DUMP_EL
DUMP_S
DUMP_I
DIAG_C
DIAG_B
DIAG_OK
DIAG_FAIL
DIAG_EL
DIAG_S
DIAG_I
TBD_EOF
TBD_COUNT
RFD_C
RFD_B
RFD_OK
RFD_S11
RFD_S10
RFD_S9
RFD_S8
RFD_S7
RFD_S6
RFD_EL
RFD_S
RBD_EOF
RBD_F
RBD_ACT_CNT
RBD_EL
RBD_SIZE
CBA_BASE
BUF_BASE
CMD_INPROG
RCV_INPROG

3.3.2.9 Packet Buffering Constants — cvm_buffer.h

This include file defines the Communication Virtual Machine Packet Buffering.

Constants:

SBM_SIZE
NUM_SBM_PAGES
MAX_SBM_SIZE

SBM_BASE_ADDR
BUF_ALLOC
BUF_RETURN

3.3.2.10 Event Logging Header — cvm_event.h

This include file defines the Communication Virtual Machine Event Logging types.

Data Structures:

eventptr
eventstr

3.3.2.11 Intervale Timer Header — cvm_itimer.h

This include file is related to the Communication Virtual Machine Interval Timer.

Data Structures:

timedeventptr
timedeventstr

3.3.2.12 Memory Management Header — cvm_memory.h

This include file is related to the Communication Virtual Machine Memory Management.

Data Structures:

heapctlptr
heapctlstr
heapstatstr

3.3.2.13 Packet Manipulation Header — cvm_packet.h

This include file is related to the Communication Virtual Machine Packet Manipulation.

Constants::

PKT_TAIL
PKT_HEAD
PKT_MAKE
PKT_EXTRACT

3.3.2.14 Set Facility Header — cvm_set.h

This include file is related to the Communication Virtual Machine Set Facility.

Data Structures:

setptr
expsetptr
elemhdrptr
elemhdrstr
setstr
expsetptr
expsetstr
expsetptr

3.3.2.15 Timer Facilty Header — cvm_token.h

This include file is related to the Communication Virtual Machine Timer Facility.

Constants:

TKN_INSERT
TKN_REMOVE
TOKEN_BODY_SIZE
TOKEN_BODY_ADDR
AP_INTF
ENET_INTF
UART_INTF
TIMER_INTF
EXCEPTION_HDLR
MAX_TKN_DATA_ITEMS

Data Structures:

tokenptr
tknsrcdest
enettkntype_str
tokenhdrstr
tokenstr

3.3.2.16 Console Header — cvm_uart.h

This include file is related to the Communication Virtual Machine UART.

Data Structures:

cqueueptr
cqueuestr

3.3.2.17 External Protocol Header — ext_proto.h

This include file is related to the External Protocol.

Constants:

EXT_ACK_TIMER
EXT_MAX_SEQ
EXT_NUM_BUFS
EXT_MAX_SEQ
EXT_NUM_BUFS
EXT_MAX_MSG_LEN
EXT_SEND_Q_SIZE
EXT_TIMEOUT_IN_SECS
EXT_TIMEOUT_INT
EXT_ACK_TIMEOUT_INT
MIN_EN_PKT_LEN

IP_TYPE
IP_HDR_LEN
UDP_TYPE
UDP_HDR_LEN
UDP_DEST_PORT_NUM
IP_DEST_ADDR
EXT_ENET_DEST_ADDR
EXT_ENET_DEST_ADDR
EXT_DATA
EXT_ACK
EXT_NAK

Data Structures:

extqueueptr
IPaddrptr
enetpktptr
inetpktptr
udppktptr
extpktptr
extnetpktptr
seqnum
framekind
bufnum
extqueuestr
IPaddrstr
enetpktstr
inetpktstr
udppktstr
extpktstr
extnetpktstr

3.3.2.18 External Protocol Interface Header — ext_proto_if.h

This include file is related to the External Protocol Interface.

Constants:

EXT_SENDMSG_TKN
EXT_MSGRCVD_TKN
EXT_TIMEOUT_TKN
MAX_EXT_MSG_LEN
EXT_MSG_RCVD_CMD

Data Structures:

extmsgptr

extmsgbodyptr
EXTmsgtkn_ptr
EXTmsgshorttkn_ptr
EXTtimeouttkn_ptr
extmsgstr
extmsgbodystr
EXTmsgtkn_str
EXTmsgshorttkn_str
EXTtimeouttkn_str

3.3.2.19 Monitor Active Nodes Protocol Header — man_proto.h

This include file is related to the Monitor Active Nodes Protocol.

Constants:

MAN_INACTIVE_THRESHOLD
MAN_IM_ALIVE
MAN_TIMER_INT_IN_SECS
MAN_TIMER_INT

Data Structures:

MAN_noderec_ptr
MAN_data_ptr
MAN_PKT_ptr
MAN_noderec_str
MAN_data_str
MAN_PKT_str

3.3.2.20 Monitor Active Nodes Protocol Interface Header — man_proto_if.h

This include file is related to the interface of the Monitor Active Nodes protocol.

Constants:

MAN_MAX_NODES
MAN_MAX_NODENUM
MAN_NEWSTATUS_TKN
MAN_DUMP_RQST_TKN
MAN_DUMP_RPLY_TKN
MAN_DUMP_RPLY_SHORT_TKN
MAN_TIMER_TKN

Data Structures:

MANnewstatustkn_ptr
MANdumprqsttkn_ptr
MANdumprplytkn_ptr
MANdumprplyshorttkn_ptr

MANtimertkn_ptr
MANnewstatustkn_str
MANdumprqsttkn_str
MANdumprplytkn_str
MANdumprplyshorttkn_str
MANtimertkn_str

3.3.2.21 Page Transfer Protocol Header — pt_proto.h

This include file is related to the Page Transfer Protocol.

Constants:

PT_NODE_ID_SHIFT
PT_PROTO_SHIFT
PT_COUNT_SHIFT
PT_SEQNUM_NODE_ID_MASK
PT_SEQNUM_PROTO_MASK
PT_SEQNUM_COUNT_MASK
PT_SEQNUM_DESTOBJ_MASK
PT_MAX_SEQNUM_COUNT
PT_MAX_SEQNUM_DESTOBJ
PT_PGTRANS
PT_INFOTRANS
PT_SYNC
PT_LOWER_HALF_PG
PT_UPPER_HALF_PG
PT_TIMEOUT_INT_IN_SECS
PT_TIMEOUT_INT

Data Structures:

PT_data_ptr
PT_dest_ptr
PT_AIrec_ptr
PT_AOrec_ptr
PT_PGTRANS_PKTHDR_ptr
PT_PGTRANS_PKT_ptr
PT_INFOTRANS_PKTHDR_ptr
PT_SYNC_PKT_ptr
PT_data_str
PT_dest_str
PT_AIrec_str
PT_AOrec_str
PT_PGTRANS_PKTHDR_str

PT_PGTRANS_PKT_str
PT_INFOTRANS_PKTHDR_str
PT_SYNC_PKT_str

3.3.2.22 Page Transfer Protocol Interface Header — pt_proto_if.h

This include file is related to the Page Transfer Protocol Interface.

Constants:

PT_MAX_DESTS
PT_CONV
PT_SEND_PAGE_TKN
PT_SEND_PAGE_SHORT_TKN
PT_PAGE_SENT_TKN
PT_RCVD_PAGE_TKN
PT_SEND_INFO_TKN
PT_INFO_SENT_TKN
PT_RCVD_INFO_TKN
PT_KERN_RPLY_TKN
PT_WRITE_SYNC_TKN
PT_SYNC_READ_TKN
PT_TIMEOUT_TKN

Data Structures:

destinfoptr
PTsendpagetkn_ptr
PTsendpageshorttkn_ptr
PTpagesenttkn_ptr
PTrcvdpagetkn_ptr
PTsendinfotkn_ptr
PTinfosenttkn_ptr
PTrcvdinfotkn_ptr
PTwritesynctkn_ptr
PTkernrplytkn_ptr
PTsyncreadtkn_ptr
PTtimertkn_ptr
MBaddr
destinfostr
PTsendpagetkn_str
PTsendpageshorttkn_str
PTpagesenttkn_str
PTrcvdpagetkn_str
PTsendinfotkn_str

PTinfosenttkn_str
PTrcvdinfotkn_str
PTkernrplytkn_str
PTwritesynctkn_str
PTsyncreadtkn_str
PTtimertkn_str

3.3.2.23 Reliable Basic Invoke Protocol Header — rbi_proto.h

This include file is related to the Reliable Basic Invoke Protocol.

Constants:

RBI_RQST_PHASE
RBI_RPLY_PHASE
RBI_TSYNC_INT_IN_SECS
RBI_TSYNC_TIMEOUT_INT
RBI_RQST_HDR_SIZE
RBI_RPLY_HDR_SIZE
RBI_PARMSIZE
RBI_CAPASIZE

Data Structures:

RqstBlkHdr_ptr
RplyBlkHdr_ptr
RBI_data_ptr
AIrec_ptr
AOrec_ptr
RBI_threadrec_ptr
RBI_PKT_ptr
RBI_RQST_PKT_ptr
RBI_RPLY_PKT_ptr
abitID
abitID_ptr
RqstBlkHdr_str
RplyBlkHdr_str
RBI_data_str
AIrec_str
AOrec_str
RBI_threadrec_str
RBI_PKT_str
RBI_RQST_PKT_str
RBI_RPLY_PKT_str

3.3.2.24 Reliable Basic Invoke Header — rbi_proto_if.h

This include file is related to the Reliable Basic Invoke Protocol Interface.

Constants:

RBI_INV_RQST
RBI_INV_RPLY
RBI_RQST_TSYNC
RBI_RPLY_TSYNC
RBI_CONV
RBI_ADD_ABID_TKN
RBI_REMOVE_ABID_TKN
RBI_OUTRQST_TKN
RBI_OUTRPLY_TKN
RBI_INRQST_TKN
RBI_INRPLY_TKN
RBI_ABORT_THREAD_TKN
RBI_TIMEOUT_TKN

Data Structures:

RBIaddabidtkn_ptr
RBIrmvabidtkn_ptr
RBIoutrqsttkn_ptr
RBIoutrplytkn_ptr
RBIinrqsttkn_ptr
RBIinrplytkn_ptr
RBIabortthreadtkn_ptr
RBItimertkn_ptr
RBIaddabidtkn_str
RBIrmvabidtkn_str
RBIoutrqsttkn_str
RBIoutrplytkn_str
RBIinrqsttkn_str
RBIinrplytkn_str
RBIabortthreadtkn_str
RBItimertkn_str

3.3.2.25 Reliable Message Protocol Header — rm_proto.h

This include file is related to the Reliable Message Protocol.

Constants:

RM_MAX_RETRIES
RM_TIMEOUT_IN_SECS
RM_TIMEOUT_INT

RM_NUM_NODES
RM_RCVD_BV_SIZE
RM_NODE_ID_SHIFT
RM_PROTO_SHIFT
RM_CONVNUM_NODE_ID_MASK
RM_CONVNUM_PROTO_MASK
RM_CONVNUM_COUNT_MASK
RM_MAX_CONVNUM_COUNT
RM_ANCIENT_PKT

Data Structures:

RM_PKT_ptr
RM_data_ptr
RMrec_ptr
RM_data_str
RMrec_str
RM_PKT_str

3.3.2.26 Reliable Message Protocol Interface Header — rm_proto_if.h

This include file is related to the Reliable Message Protocol Interface.

Constants:

RM_CONV
RM_SENDMSG_TKN
RM_MSGDONE_TKN
RM_TIMEOUT_TKN
RM_MSGRCVD_TKN

Data Structures:

sendRMtkn_ptr
RMsenttkn_ptr
RMrcvdtkn_ptr
RMtimeouttkn_ptr
ENAddrstr
sendRMtkn_str
RMsenttkn_str
RMrcvdtkn_str
RMtimeouttkn_str

3.3.2.27 Packet Filtering Header — se_filter.h

This include file is related to the Communication Virtual Machine Packet Filtering.

Macros:

SE_FILTER_HDR

IS_BC_ENA
ISNOT_BC_ENA
IS_MC_ENA
ISNOT_MC_ENA

Data Structures:

MCAtabptr
MCAkeyptr
MCAtabstr
MCAkeystr

3.3.2.28 Ethernet Interface Header — se_if.h

This include file is related to the Ethernet Interface.

Data Structures:

enettkn_ptr
enettkn_str

3.3.2.29 Ethernet Controller Header — sunenet.h

This include file is related to the Low-level Sun Ethernet Controller.

Macros:

RMP
MCBP
SWAB
CHAN_ATTN
SCB_CMD_WAIT
IN_PORT

Data Structures:

pktdescptr
pfdescptr
pkttmplptr
ENaddrptr
ENpacketptr
regmapptr
scpptr
iscpptr
scbptr
nopptr
iasetupptr
configureptr
mcsetupptr

transmitptr
tdrptr
dumpptr
diagnoseptr
tbdptr
rfdptr
rbdptr
gcbptr
cbaptr
rfaptr
mcbptr
ptypes_str
pktdescstr
pfdescstr
pkttmplstr
ENaddrstr
ENpacketstr
regmapstr
iaddr
ioffset
ioffsetptr
scpstr
iscpstr
scbstr
nopstr
iasetupstr
configurestr
mcsetupstr
transmitstr
tdrstr
gcbptr
dumpstr
diagnosestr
tbdstr
rfdstr
rbdstr
gcbstr
cbastr
rfastr
mcbstr

3.3.2.30 Thread Maintenance and Repair Protocol Header — tmar_proto.h

This include file is related to the Thread Maintenance and Repair Protocol.

Constants:

TMAR_MAX_THREADS
TMAR_MAX_THREAD_SECTS
TMAR_TBREAK_THRESHHOLD
TMAR_MAX_MISSING_CNT
TMAR_POLL_INT_IN_SECS
TMAR_POLL_INT
TMAR_REFRESH_TO_INT_IN_SECS
TMAR_REFRESH_TO_INT
TMAR_BCAST_RETRY_INT_IN_SECS
TMAR_BCAST_RETRY_INT
TMAR_POLL
TMAR_RESPONSE
TMAR_REFRESH
TMAR_REPAIRTHREAD
TMAR_ACTIVATETHREAD
TMAR_SPARMS_RQST
TMAR_SPARMS_RPLY
TMAR_BEGIN_ABORT_RQST
TMAR_ACK
TMAR_TSECTS_ABORTED
TMAR_POLLNUM_NODE_ID_MASK
TMAR_POLLNUM_PROTO_MASK
TMAR_POLLNUM_COUNT_MASK
TMAR_MAX_POLL_COUNT
TMAR_SPECIAL_POLL_NUM

Data Structures:

TMAR_threadindxptr
TMAR_threadrecptr
TMAR_data_ptr
TMAR_ATrecptr
TMAR_Tinfoptr
TMAR_refreshrecptr
TMAR_TRinfoptr
TMAR_TAinfoptr
TMAR_PKThdr_ptr
TMAR_POLL_PKT_ptr
TMAR_REFRESH_PKT_ptr

TMAR_RESPONSE_PKT_ptr
TMAR_FREEZEHEAD_PKT_ptr
TMAR_HEADFROZEN_PKT_ptr
TMAR_REPAIRTHREAD_PKT_ptr
TMAR_ACTIVATETHREAD_PKT_ptr
TMAR_SPARMS_RQST_PKT_ptr
TMAR_SPARMS_RPLY_PKT_ptr
TMAR_BEGIN_ABORT_RQST_PKT_ptr
TMAR_ACK_PKT_ptr
TMAR_TSECTS_ABORTED_PKT_ptr
abitID
abitIDptr
TMAR_threadindxstr
TMAR_threadrecstr
TMAR_data_str
TMAR_ATrecstr
TMAR_Tinfostr
TMAR_refreshrecstr
TMAR_TRinfostr
TMAR_TAinfostr
TMAR_PKThdr_str
TMAR_POLL_PKT_str
TMAR_REFRESH_PKT_str
TMAR_RESPONSE_PKT_str
TMAR_FREEZEHEAD_PKT_str
TMAR_HEADFROZEN_PKT_str
TMAR_REPAIRTHREAD_PKT_str
TMAR_ACTIVATETHREAD_PKT_str
TMAR_SPARMS_RQST_PKT_str
TMAR_SPARMS_RPLY_PKT_str
TMAR_BEGIN_ABORT_RQST_PKT_str
TMAR_ACK_PKT_str
TMAR_TSECTS_ABORTED_PKT_str

3.3.2.31 Thread Maintenance/ Repair Protocol Interface Header — tmar_proto_if.h

This include file is related to the Thread Maintenance and Repair Protocol Interface.

Constants:

TMAR_CONV
TMAR_POLLING_TIMER_TKN
TMAR_REFRESH_TIMEOUT_TKN
TMAR_BCAST_RETRY_TKN

TMAR_RI_ABORT_TKN
TMAR_ABORT_DONE_TKN
TMAR_START_ABORT_TKN
TMAR_START_ABORT_SHORT_TKN
TMAR_END_ABORT_TKN
TMAR_BEGIN_ABORT_TKN
TMAR_BEGIN_ABORT_SHORT_TKN

Data Structures:

TMARrefreshTOtkn_ptr
TMARbcastretrytkn_ptr
TMARRIaborttkn_ptr
TMARbeginaborttkn_ptr
TMARbeginabortshorttkn_ptr
TMARstartaborttkn_ptr
TMARstartabortshorttkn_ptr
TMARabortdonetkn_ptr
TMARendaborttkn_ptr
TMARsparmsrqsttkn_ptr
TMARsparmstkn_ptr
TMARrefreshTOtkn_str
TMARbcastretrytkn_str
TMARRIaborttkn_str
TMARbeginaborttkn_str
TMARbeginabortshorttkn_str
TMARstartaborttkn_str
TMARstartabortshorttkn_str
TMARabortdonetkn_str
TMARendaborttkn_str
TMARsparmsrqsttkn_str
TMARsparmstkn_str

3.3.3 Source Files

3.3.3.1 Communications Virtual Machine — cvm.c

This C source file contains logic for the Communication Virtual Machine.

Routine:

void StartCVM()

This is the startup for the communication virtual machine code.
This is called after the startup routine has initialized everything.

Routine:

```
void   RunCM()
```

The main communications machine run cycle.

This is the routine that executes the state machines.

The loop should never end, but a termination mechanism is provided for debugging purposes — some errors can result in turning off the execution loop.

Routine:

```
void   CVMerror(string, src, type)
       char    *string;
       u_short     src;
       u_short     type;
```

This handles fatal errors from the Communications Virtual Machine.

It prints a message and then traps to the monitor (if in debug mode) or forces the current SM into its exception state (if in production mode).

In addition to taking a string to be printed, this routine takes an integer (i.e., 'src') that indicates which mechanism had an exception and a word (i.e., 'type') that indicates the specific type of error that occurred. These parameters are used (in production mode) to let the user-defined exception handlers know the cause of the exception.

In production mode, if there is not a user-specified exception handling state, the system kills the current SM.

The exception 'type' is packed into the token 'ptr' field with the state the SM was in when the error occurred.

If the 'src' is "FATAL", then we trap to the monitor, even in production mode.

3.3.3.2 Packet Buffering — cvm_buffer.c

This C source file contains logic for the Communication Virtual Machine Packet Buffering.

Routine:

```
void   InitBuf()
```

Initialize the shared Multibus memory manager.

This routine initializes the shared Multibus memory bit vector ('SBM[]'); it marks all of the pages that are not available to be allocated; these include the IPICBs for the AP and the CPs and the communication channels; in addition, only a certain amount of the Multibus memory is actually used.

Routine:

```
mbaddr       BufAlloc()
```

Allocate a buffer in the shared memory space.

This routine allocates a range of (contiguous) memory from the node's shared memory — i.e., the memory that can be accessed via the Multibus P1 bus, by the applications and communications processors.

This routine allocates physical memory in units of pages, and returns the (physical) Multibus address of the allocated page. This address can then be converted into an appropriate virtual address in order to access the page. (This is done since the allocated page is potentially shared by multiple processors with differing virtual address layouts; each virtual address space can reference pages located in the Multibus address and the appropriate translation is left to each of them.)

A bit vector is used to indicated the state of each page (i.e., in use or free). This is a global structure that is initialized to indicate reserved pages.

Routine:

```
void   BufReturn(pgaddr)
       mbaddr       pgaddr;
```

Return a shared memory buffer to the free pool.

This routine takes the address of a page in Multibus memory space (i.e., 'pgaddr') and puts the page back into the pool of free pages. This is done by clearing the bit corresponding to the page in the allocation bit vector.

3.3.3.3 Debug Support — cvm_debug.c

This C source file contains logic for the Communication Virtual Machine Debugger.

Routine:

```
void   Debug()
```

This routine is used to print out various data structures for debugging.

Routine:

```
void   PrintEvents()
```

Print all events in the event log buffer onto the console.

Each event is printed out on a separate line starting with the event number, followed by the event name (a string), the event type and parameter (both single byte integers), the event's timestamp, and finally the delta off of the last event (in decimal microseconds,

with the event logging overhead deducted).

The event buffer wraps around, so it is possible that the first event printed is not event #0.

This routine starts up by printing the compensation time, the size of the event log buffer, and the number of events logged so far. It then prompts the user for event to start printing with, and the types of events to show or hide.

When running the "more" banner comes up after each screen full of events, the following commands are accepted in more mode:

' ' — next page

'^M' — next line

'q' — stop printing

's' — enter stream mode

'/n' — skip n events

'.n' — goto event number n

In stream mode the events are printed non-stop until the end of the buffer is reached or any character is typed (at which point it returns to more mode).

3.3.3.4 Event Logging — cvm_event.c

This C source file contains logic for the Communication Virtual Machine Event Logging.

Routine:

```
void   InitLEvents()
```

Initialize the event logging area, initialize the event timer, and log these actions as the first event.

The event buffer logging area is in the segment at the top of physical memory.

It is assumed that the event timer is initialized elsewhere.

It is assumed that the MMU has the identity mapping.

Routine:

```
void   LogEventIntf(id, parm)
       u_char       id;
       u_char       parm;
```

This routine is used to log an event. It uses the 'eventcount' global variable as an index into the event log buffer, it gets the current value of the event timer and puts it, and the given parameter, into the event record indexed by 'eventcount'.

Interrupts must be disabled during this routine because we have to maintain the ordering of the events in the log (an interrupt could result in another event being logged), and the accuracy of the event loggin overhead must be maintained.

Routine:

```
u_long      GetSTimer()
```

This routine reads the current value of the 32 bit, free running, system timer and returns it to the caller.

3.3.3.5 Interval Timer — cvm_itimer.c

This C source file contains logic for the Communication Virtual Machine Interval Timer.

Routine:

```
void   InitTEvents()
```

This routine initializes the timed event facility, by clearing the event structures and setting up the initial interval timer values.

Routine:

```
u_short     CreateTEvent(interval, tkndest, tkntype, tkndataptr)
       u_int        interval;
       tknsrcdest   tkndest;
       u_short      tkntype;
       u_long       *tkndataptr;
```

This routine creates a timed event. After the given interval of time has elapsed, the specified token is delivered to the indicated SM.

An identifier for the newly created timed event is returned. If no new timed events can be created, a fatal error occurs.

The returned timed event ID can be used with the 'CancelTEvent()' routine to remove events that have not already occurred.

Note that it is assumed that the 'interval' parameter is given in ticks of the clock at F2 (i.e., 3.2 usec).

Routine:

```
void   CancelTEvent(event)
       short          event;
```

This routine cancels the timed event indicated by the 'event' argument.

The validity of the event ID is checked before any thing is done, and a fatal error is generated if it isn't valid.

Routine:

```
void   ITimerIntrHdlr()
```

This routine handles interrupts generated by the interval timer.

It reads the current time, checks if this is just a counter roll-over event, and if not, it issues tokens for the timed events scheduled for times within a given window of the current time, removes them from the timed event list, and sets up the interval timer for the next event in the list.

Routine:

```
u_int   GetF2Time()
```

Read current time from a clock running at TC_F2 frequency.

This routine maintains a clock running at the TC_F2 frequncy of the timer chip. It derives the current time from the system clock and knowledge of the system timer's clock frequency. If the system clock runs at a slower frequency than TC_F2, a message is printed stating that the frequency is too slow and garbage is returned.

(Alternatively, the slower clock's time could be shifted, providing a course-grained timer for some (fairly short) length of time.)

3.3.3.6 Memory Management — cvm_memory.c

This C source file contains logic for the Communication Virtual Machine Memory Management.

Routine:

```
void   InitHeap()
```

This routine initializes the heap data structures.

Routine:

```
vaddr  AllocBlk(size, zero)
       u_short        size;
       boolean        zero;
```

This routine allocates a block of free memory and returns a pointer to it. The size of the block is at least as big as the number of bytes specified in the 'size' parameter.

If the 'zero' parameter is TRUE, the allocated block is set to all zero's (otherwise the previous, undefined, contents remain in the block).

If there are no buffers of the desired size, this routine cycles through the remaining sizes, looking for free buffers of the next larger sizes. An error occurs if too big a buffer is requested, or if there is no more memory.

Because the sizes of blocks are guaranteed to powers of 2, we can take short-cuts. Also, we must tune the number of each size of buffers to use memory effectively.

Because interrupts result only in TQinsert operations, and all actions are non-preemptible, there is no need to worry about synchronization and so interrupts are not disabled.

Routine:

```
void   DeallocBlk(blkptr)
       vaddr          blkptr;
```

This routine deallocates a block of free memory pointed to by the 'addr' parameter.

The size of the block is determined by it's starting address, this is made possible by ensuring that the buffer arrays are laid out in memory in smallest-to-largest block order.

Because interrupts result only in TQinsert operations, and all actions are non-preemptible, there is no need to worry about synchronization and so interrupts are not disabled.

3.3.3.7 Packet Manipulation — cvm_packet.c

This C source file contains logic for Communication Virtual Machine Packet Manipulation.

Routine:

```
vaddr  PktCreate(PTptr, PDptr)
       pkttmplptr     PTptr;
       pktdescptr     PDptr;
```

This routine creates a packet from a given packet template, and returns a pointer to the packet buffer.

The packet created is a copy of the packet template pointed to by the 'PTptr' parameter.

A pointer to a packet descriptor is passed to this routine in the 'PDptr' parameter, which is filled out when the packet is created and is used to alter it as well as to send the packet.

A packet control block is allocated along with the necessary number of transmit buffers, and the given template is copied into the allocated buffers.

Routine:

```
void   PktDelete(PDptr)
       pktdescptr     PDptr;
```

This routine deletes the packet indicated by the 'PDptr' parameter.

This routine determines if the packet is a receive or transmit packet, and deletes it in the fashion appropriate for the type.

A received packet's RFD is deallocated and all of its RBD's and RB's are returned to their free lists.

A transmit packet's command block is deallocated (if it is not fixed), and all of the TBD's and TB's returned to their free lists.

Routine:

```
void   PktFragMake(PFDptr, addr, size)
       pfdescptr     PFDptr;
       vaddr         addr;
       u_short       size;
```

This routine takes a pointer to a packet fragment descriptor (i.e., 'PFDptr'), along with an indication of a region of memory (as defined by a starting address in 'addr' and the number of bytes in 'size'), and creates a new packet body.

Transmit buffers are allocated within the ethernet controller, the given region of memory is copied into the buffers, and the given packet descriptor is filled out.

Routine:

```
void   PktFragAdd(pos, PDptr, PFDptr)
       boolean       pos;
       pktdescptr    PDptr;
       pfdescptr     PFDptr;
```

This routine adds a packet body part to the head or tail of an existing packet. The 'pos' argument defines if the addition should go at the head or tail, the 'PDptr' argument is a

pointer to an existing packet's descriptor and the 'PFDptr' argument is a pointer to a descriptor for the packet fragment that is to be added.

The 'PktMake()' routine allocates the additional transmit buffers needed for the packet body, and this routine links these buffers into the existing packet's transmit control block.

Note that this routine assumes that the only type of packets that are manipulated in this way are transmit packets.

Routine:

```
void   PktExtract(PDptr, start, size, dstaddr)
       pktdescptr    PDptr;
       u_short       start;
       u_short       size;
       vaddr         dstaddr;
```

This routine extracts a region of a packet (described by the 'PDptr' argument) and copies it into a contiguous region of memory (given by the 'dstaddr' argument).

The region to be extracted is given by a pair of offsets — 'start' which is the number of bytes into the packet where the region begins, and 'size' which indicates how long the region is in number of bytes.

A 'size' argument of 0 is used to indicate that the region should extend to the end of the packet, regardless of its actual size.

This routine works on both received packets and packets to be transmitted.

Routine:

```
boolean PktCompare(src, dst, mask)
        vaddr       src;
        vaddr       dst;
        u_short     mask[];
```

This routine is used to compare a packet against a template.

The array pointed to by the 'src' variable is compared to the 'dst' array. The 'mask' variable is used to indicate the size of the arrays to compare.

The mask is an array of shorts that indicate the number of bytes of each segment of the packet. The mask array starts with the number of bytes to compare, followed by the number of bytes to skip. The values in the mask continue to alternate until a zero is found, except if the mask array starts with a zero, in which case it skips the next number of bytes and terminates when it finds another zero.

Routine:

```
void   PktSend(PDptr, tkndest, tknparm)
        pktdescptr   PDptr;
        tknsrcdest   tkndest;
        u_long       tknparm;
```

This routine sends the packet specified by the given packet descriptor (i.e., 'PDptr').

When the packet has been successfully transmitted, an interrupt is generated. This routine takes as parameters a token destination (usually, a protocol number) and a parameter to pass in a token sent to that destination, 'tkndest' and 'tknparm' respectively. When this packet is sent, the specified token is inserted into the token queue. The type of the token generated is system-defined and constant.

If the 'tkndest' parameter is NIL, no token is generated when the "packet sent" interrupt occurs.

If the packet to be sent is smaller than the minimum size allowed by the ethernet, this routine pads the packet.

This routine assumes that packets are properly byte-aligned, and that the transmit buffers are at least as big as the smallest possible ethernet packet. It further assumes that the minimum buffer size is greater than (or equal to) the minimum packet size. This is to ensure that there will always be room in the last buffer to pad the packet if it is too short.

3.3.3.8 Set Manipulation — cvm_set.c

This C source file contains logic for the Communication Virtual Machine Set Manipulatin.

Routine:

```
setptr         SetCreate(elemsize)
        u_short     elemsize;
```

This routine creates a set which is to contain elements of a size specified by the 'elemsize' parameter, and a pointer to the new set is returned.

The fact that 'NULL' and 'FALSE' are binary zero's makes it possible to initialize the set by alloc'ing a zero'd block from the heap.

Routine:

```
void   SetDelete(sptr)
       setptr        sptr;
```

This routine takes a pointer to a set and deletes it.

This is done by simply dealloc'ing the block's making up the set (starting with the last expansion block and moving back to the initial set block).

Routine:

```
void   SetInsert(sptr, key, elemptr)
       setptr        sptr;
       u_long        key;
       vaddr         elemptr;
```

This routine is used to insert an element into the specified set, indexed with the given key. The element to insert is given by the the pointer in the 'elem' parameter, the key is a 32-bit unsigned integer given in the 'key' parameter, and the 'sptr' parameter is a pointer to the given set.

If an element with the given 'key' is already in the given set, this routine inserts elements at the end of bucket chain so you always get the oldest one if there are more than one elements in the set with the same key.

The 'sptr' parameter is not checked, so it better really be pointing to a valid set.

Routine:

```
void   SetRemove(sptr, key)
       setptr        sptr;
       u_long        key;
```

This routine is use to remove the element indexed by the 'key' parameter from the specified set.

If the removal of an element causes an expansion set to become empty, it is dealloc'd.

If an element is removed from the main table, the element at the head of the bucket chain (if there is one for the given slot) is moved into the main table.

If no element with the given 'key' value is not found in the set, this routine has no effect.

Routine:

```
vaddr  SetLookup(sptr, key)
       setptr        sptr;
       u_long        key;
```

Lookup an element in a set.

This routine is used to find an element indexed by the 'key' parameter in the given set. If an element with the given 'key' is found in the set, a pointer to it is returned, otherwise a NULL value is returned.

Routine:

```
u_short      SetSize(sptr)
     setptr       sptr;
```

This routine takes a pointer to a set and returns an indication of the number of elements currently in the set.

3.3.3.9 Initial Startup — cvm_start.c

This C source file contains logic for starting up the Communications subsystem on a Sun 1.5 board.

Routine:

```
void  main()
```

This routine sets up a Sun 1.5 processor board for standalone use.

Starts by disabling interrupts and printing a banner. It then maps multibus memory into the processors address space and invalidates the unused parts of the space. Next, the interprocessor interface is initialized along with the processor's system timer. Following this, the event logging facility is initialized, a message is printed with the address of the standalone debug routine, and the user's standalone program is called.

Interrupts are disabled and stack pointers are set up by 'crt0.a'.

Routine:

```
void  Error(string)
      char*         string;
```

Fatal Error reporting routine. Prints the error msg and halts.

3.3.3.10 Token Handling — cvm_token.c

This C source file contains logic for Communication Virtual Machine Token Handling.

Routine:

```
void  TQinsert(tkndest, tknsrc, tkntype, numargs, arg)
      tknsrcdest    tkndest;
      tknsrcdest    tknsrc;
      u_short       tkntype;
      u_short       numargs;
      u_long        arg[MAX_TKN_DATA_ITEMS];
```

Insert a token into the input queue.

This routine takes a token destination, a token source, a token type, and the data associated with the token. From this information, a token is created and added to the tail of the input token queue, 'TQ[]'.

A fatal error occurs if the token queue is full.

Routine:

```
void  TQremove()
```

Remove a token from the input queue.

This routine removes the token at the head of the input token queue. A fatal error occurs if the specified queue is empty.

Because interrupts only result in tokens being added, and adding tokens only modifies the TQ's tail pointer, there is no need to lock out interrupts for synchronization.

Routine:

```
tokenptr    nextTQ()
```

Get next token from the input queue.

This routine removes the token at the head of the input token queue. Because interrupts only result in tokens being added, and adding tokens only modifies the TQ's tail pointer, there is no need to lock out interrupts for synchronization.

In the current implementation, this is where the CVM spends its idle time.

3.3.3.11 UART (Console) Handling — cvm_uart.c

This C source file contains logic for UART Handling.

Routine:

```
void  InitUART()
```

This routine initializes the UART.

See the appropriate data sheets to see what's going on.

This also initializes the character output queue, and sets up the UART's interrupt vector.

Routine:

```
void  UARTIntrHdlr()
```

This is the UART interrupt handling routine. When the UART generates an interrupt, the source of interrupt is determined. If the interrupt was generated as a result of receiving a character, a token is created, the character is put in the 'ptr' field, and the token is placed into the appropriate SM's input token queue.

If the interrupt is indicating that the UART is empty, the output character queue is examined and if it is not empty, the next character is removed and transmitted.

UART_A_CTL is the command register and UART_A_DATA is the data register. Only the A channel of the UART is used here.

Routine:

```
void  PutC(chr)
      u_char     chr;
```

This routine puts a byte of data into the output buffer ('outbuf'). If the buffer is empty, the character is placed directly into the UART, provided that it is not busy, in which case the character is added to the buffer.

If the buffer isn't empty, many special cases must be considered. First, if the buffer has recently been filled to its limit, incoming data is ignored until the buffer empties out to the "low water mark". If the buffer hasn't overflowed, we check to see if it is full and will overflow. If so, a flag is set, and if not, the character is put into the buffer.

3.3.3.12 External Message Protocol — ext_proto.c

This C source file contains logic for the External Message Protocol.

Routine:

```
void InitEXT()
```

Initialize the EXT protocol.

Routine:

```
void   EXT_TknHdlr(toknptr)
       tokenptr      toknptr;
```

This routine handles tokens for the EXT protocol.

Routine:

```
void SendMsg_EXT_action(shortmsg)
        EXTmsgshorttkn_ptr shortmsg;
```

This routine sends a message using the EXT protocol.

Routine:

```
void MsgRcvd_EXT_action(enetmsg)
        enettkn_ptr enetmsg;
```

This routine receives an EXT protocol message.

Routine:

```
void SendFrame(kind, frame)
        framekind   kind;
        seqnum      frame;
```

This routine sends an EXT protocol frame packet.

Routine:

```
int     CheckSum(p, amt)
        u_short       *p;
        u_int         amt;
```

The checksum mechanisms work by summing the data 16 bits at a time. If there are an odd number of bytes for a UDP checksum, either the last byte must be followed by a zero, or the equivalent of this must occur, for the calculation to be correct. Here, the data is summed 2 bytes at a time. If there is an odd amount of data, the last byte pretends to be followed by a zero.

Routine:

```
void PrintPacket(p)
        extnetpktptr p;
```

This routine prints the indicated packet.

Routine:

```
u_long NewTimeoutId()
```

Generate "unique" id for keeping track of timeouts.

3.3.3.13 Monitor Active Nodes Protocol — man_proto.c

This C source file contains logic for the Monitor Active Nodes Protocol.

Routine:

```
void   InitMAN()
```

Initialize MAN private data structures, including Ethernet address, a node status array, and a packet to be broadcast periodically to the other Alpha nodes. In addition, the first transmission of the broadcast packet and check of active nodes is scheduled.

This protocol was created to support the straightforward replication scheme first used in Alpha (which featured a master copy on which operations were performed and a number of back-up copies that were updated at the completion of each operation).

Routine:

```
void   MAN_TknHdlr(toknptr)
       tokenptr      toknptr;
```

Token handler for tokens intended for the MAN protocol. Based on the source and type of the token, an action routine is selected to handle the token.

Routine:

```
void PktRcvd_MAN_action(toknptr)
       enettkn_ptr toknptr;
```

Process an "I'm alive" packet received from another node. First, the packet is checked to verify that it is an "I'm alive" packet. If it is not, this fact is logged. Otherwise, the status for the node that sent the packet is updated to indicate that the node is active, and a watchdog timer is set in case no further "I'm alive" packets are received for a while. If the node had been considered inactive, the kernel on the AP is notified of the change in node status.

Routine:

```
void PktSent_MAN_action(toknptr)
       enettkn_ptr toknptr;
```

The "I'm alive packet for this node has been sent. (The Ethernet controller signals the completion of the transmission.) Indicate that the transmission has completed.

Routine:

```
void Main_MAN_action(toknptr)
       MANtimertkn_ptr toknptr;
```

Check node status of other nodes and send next "I'm alive" packet.

This routine carries out almost all of the work for the MAN protocol.

First, if the "I'm alive" packet that was scheduled to be sent during the last execution of this routine has been sent, another "I'm alive" packet is scheduled to be sent. Otherwise, the fact that the previous packet has not yet been sent is logged.

Next, every other active node in the system has an associated counter that is incremented. These counters act as watchdog timers for the other nodes; if a counter ever gets too high, the corresponding node is declared inactive and the kernel (on the AP) is notified of the node's assumed change in state.

Finally, the next execution of this routine is scheduled.

Routine:

```
void DumpRqst_MAN_action()
```

Send the (assumed) state of the world to the kernel on the AP. Also, indicate that the kernel should be notified of any state changes in the future.

3.3.3.14 Page Transfer Protocol — pt_proto.c

This C source file contains logic for the Page Transfer Protocol.

Routine:

```
void   InitPT()
```

Initialize PT private data structures, including the PT sequence number and sets describing ongoing page transfers to ('activeout') and from ('activein') other nodes.

This protocol was created to support the straightforward replication scheme first used in Alpha (which featured a master copy on which operations were performed and a number of backup copies that were updated at the completion of each operation).

Routine:

```
void   PT_TknHdlr(toknptr)
       tokenptr      toknptr;
```

Token handler for tokens intended for the PT protocol. Based on the source and type of the token, an action routine is selected to handle the token.

Routine:

```
void   SendPage_PT_action(toknptr)
       PTsendpageshorttkn_ptr   toknptr;
```

Initiate a page transfer to one or more target objects.

This routine begins to transfer a page to a number of target objects. After checking the parameters, a number of packets are created — two for each object that is to receive a copy of the page. Each packet contains half of the page (due to Ethernet packet limitations, as described below). The actual transfer is begun by sending the first half of the page to each target.

This routine and 'PktSent_PT_action()' transmit pages, half a page at a time due to Ethernet packet length restrictions. Unfortunately, the only reliable message facility that now exists is a point-to-point protocol, not a broadcast protocol. Since reliable transfers must be guaranteed and since time is of the essence in implementing this protocol, it is best to use that protocol for the time being.

However, the mechanisms currently available to the protocol writer do not allow packet fragments to be reused once they have been added to a packet. Since this protocol uses the reliable message protocol and since that protocol adds information to the packet to be sent, there is no way to reuse the packet that we are trying to send AND use the underlying support protocol. The dismal implication of all of this is that separate packets must be made from scratch (including copying the page of data) for each of the objects in the destination object list.

Routine:

```
void   PktSent_PT_action(toknptr)
       RMsenttkn_ptr       toknptr;
```

This routine completes the page transfer to one of the destination objects.

After the first half of the page has been reliably sent to the destination object (via the RM protocol), this protocol is notified. It then sends the second half of the page to the object.

After the second half of the page has been reliably sent to the destination object (via the RM protocol), this protocol is again notified. It checks to see if transfers to any of the other destination objects are still in progress. If not, it sends a result status to the originator of the page transfer request.

A successful transfer is indicated only if every destination object received both halves of the page successfully (according to the RM protocol).

Routine:

```
void   PageRcvd_PT_action(toknptr)
              RMrcvdtkn_ptr      toknptr;
```

This routine is executed each time a packet containing half a page is received from another node.

If the packet contains the first half of the page, a page of shared Multibus memory is allocated, the packet data is copied into the page, a timer is set to go off if the second page is not received quickly enough, and a record is created for this page transfer.

If the packet contains the second half of the page, the page transfer record is accessed, the timer event is canceled, the packet data is copied into the Multibus memory page, and the kernel is notified of the arrival of the new page.

In each case the packet is deleted at the end of the routine, and any exception conditions are logged.

Routine:

```
void   PageRcvd_Timeout_PT_action(toknptr)
       PTtimertkn_ptr      toknptr;
```

Handle expiration of timer (indicating second page half is late).

This routine is executed if the timer set to bound the waiting time for the second page half expires. In that case, the page transfer record is accessed and deleted after the page of Multibus memory has been deallocated.

If the page transfer record cannot be found, then the page transfer has already completed (presumably after the timer expired, but before this routine was executed).

Routine:

```
void   SendInfo_PT_action(toknptr)
       PTsendinfotkn_ptr   toknptr;
```

This routine transfers object state information to one or more target objects. This is quite similar to sending a data page.

After checking the parameters, a number of packets are created — one for each object that is to receive a copy of the object state information. The transfer is begun by sending these packets to their respective targets (via the RM protocol).

Routine:

```
void  InfoSent_PT_action(toknptr)
          RMsenttkn_ptr     toknptr;
```

This routine completes the transfer of object state information to one of the destination objects.

After the object state information has been reliably sent to the destination object (via the RM protocol), this protocol is notified. It checks to see if transfers to any of the other

destination objects are still in progress. If not, it sends a result status to the originator of the information transfer request.

A successful transfer is indicated only if every destination object received the information successfully (according to the RM protocol).

Routine:

```
void  InfoRcvd_PT_action(toknptr)
      RMrcvdtkn_ptr     toknptr;
```

Process packet containing object state info from another node.

This routine is executed each time a packet containing half a page is received from another node. In response, a page of shared Multibus memory is allocated, the packet data is copied into the page, the kernel is notified of the information's arrival, and a record is created for this information transfer.

The packet is deleted at the end of the routine, and any exception conditions are logged.

Routine:

```
void  KernRply_PT_action(toknptr)
      PTkernrplytkn_ptr  toknptr;
```

Handle the kernel's response to a PT protocol notification. This response always indicates that the kernel has gotten the information it needs and the Multibus page allocated by PT can be deallocated.

In addition, the transfer record can also be deleted.

Routine:

```
void  WriteSync_PT_action(toknptr)
      PTwritesynctkn_ptr      toknptr;
```

Place a reference mark (a sync token) in the token queue.

Routine:

```
void  SyncRead_PT_action(toknptr)
          PTsyncreadtkn_ptr       toknptr;
```

Notify the kernel on the AP that a reference mark (a sync token) has been reached.

3.3.3.15 Interface to AP — rbi_APintf.c

This C source file contains logic for the Communication Virtual Machine AP Interface.

Routine:

```
void   SignalAP(command, tknsrc, tkntype, tknptr, tknsize)
       u_short        command;
       tknsrcdest     tknsrc;
       u_short        tkntype;
       tokenptr       tknptr;
       u_short        tknsize;
```

Signal the AP.

This routine takes a token type and a pointer to a partially filled out token structure; it fills out the rest of the token and passes it to the AP via the IPI interface (specifying the desired 'command').

Routine:

```
void   APIIntrHdlr()
```

API Interrupt Handler.

This routine handles interrupts from the interprocessor interface.

3.3.3.16 Reliable Basic Invoke Protocol — rbi_proto.c

This C source file contains logic for the Reliable Basic Invoke Protocol.

Routine:

```
void   InitRBI()
```

Initialize RBI private data structures, including sets to monitor both incoming ('activein') and outgoing ('activeout') invocations, as well as the invocations that correspond to each thread.

Routine:

```
void   RBI_TknHdlr(toknptr)
       tokenptr       toknptr;
```

Token handler for tokens intended for the RBI protocol. Based on the source and type of the token, an action routine is selected to handle the token.

Routine:

```
void   AddAbit_RBI_action(toknptr)
       RBIaddabidtkn_ptr   toknptr;
```

Add an alphabit identifier to the set of logical addresses for this node. Subsequently, Ethernet packets with that logical (multicast) address will be received by this node.

Routine:

```
void   RemoveAbit_RBI_action(toknptr)
       RBIrmvabidtkn_ptr   toknptr;
```

Remove an alphabit identifier from the set of logical addresses recognized by this node. Subsequently, Ethernet packets with that logical (multicast) address will not be received by this node.

Routine:

```
void   OutRequest_RBI_action(toknptr)
       RBIoutrqsttkn_ptr    toknptr;
```

Initiate an (outgoing) operation invocation. This is done by encapsulating the invocation parameter block in an RBI packet and using the RM protocol to send it reliably to its destination object. In addition, a record of this conversation is created, and this invocation is added to the list of invocations for this thread. Finally, if this is the root section of the thread, thread maintenance is initiated.

The processing of the invocation will continue when the RM protocol responds with the status of the reliable message delivery.

Routine:

```
void   OutRequest_Ack_RBI_action(toknptr)
       RMsenttkn_ptr        toknptr;
```

Proceed with next stage of outgoing operation invocation. At this point, the RM protocol has successfully sent the invocation to the destination object on a remote node. Now control (the head of the thread) must be explicitly transferred to the invoked node. A timer is set in case the transfer message (TSYNC) is not received.

Routine:

```
void   OutRequest_Fail_RBI_action(toknptr)
       RMsenttkn_ptr        toknptr;
```

Initial phase of outgoing operation invocation failed. Since it is not safe to assume that the transmission really failed, this action routine acts as if it had succeeded. The normal

thread maintenance and repair mechanism will determine whether the first stage of the invocation occurred or not, and trim the thread if it did not succeed.

Routine:

```
void   OutRequest_Tsync_RBI_action(toknptr)
       RMrcvdtkn_ptr        toknptr;
```

The TSYNC packet has been received from the remote node (on which the invocation's destination object is located). Therefore, transfer the head of the invoking thread to that remote node. Also, attempt to cancel the timer event associated with the receipt of this message.

Routine:

```
void   OutRequest_Timeout_RBI_action(toknptr)
       RBItimertkn_ptr toknptr;
```

Transfer control (TSYNC) packet not received in time.

The watchdog timer for the receipt of the transfer of control (head of thread) packet has expired. Given communication queues, delays, and errors, it is difficult to know the true situation. Therefore, transfer control to the remote node — that is, stop reporting to TMAR that the thread's head is local. If the remote node has not assumed responsibility for the thread's head, TMAR will detect that the thread's head has died and will repair the thread.

Routine:

```
void OutReply_RBI_action(toknptr)
        RBIoutrplytkn_ptr    toknptr;
```

Initiate a reply for a previously received invocation. Normally, this action routine will encapsulate the reply portion of the invocation parameter block in an RBI packet and use the RM protocol to send that packet reliably to the invoking node. In addition, this node will stop reporting that the thread's head is local for purposes of thread maintenance.

In the event that this thread section should be aborted, the reply is not carried out. Rather, a cleanup of the invocation's data and data structures is carried out.

Routine:

```
void    OutReply_Ack_RBI_action(toknptr)
        RMsenttkn_ptr        toknptr;
```

Proceed with next stage of the invocation reply. At this point, the RM protocol has successfully sent the reply to the invoking node. A synchronization message (TSYNC) from the invoking node will complete the invocation sequence. A timer is set in case this message is not received.

Routine:

```
void    OutReply_Fail_RBI_action(toknptr)
        RMsenttkn_ptr        toknptr;
```

Initial phase of invocation reply failed. Since it is not safe to assume that the transmission really failed, this action routine acts as if it had succeeded. The normal thread maintenance and repair mechanism will determine whether the first stage of the invocation occurred or not, and trim the thread if it did not succeed. This routine will clean up as if the reply had succeeded and the subsequent TSYNC synchronization message had been received.

Routine:

```
void    OutReply_Tsync_RBI_action(toknptr)
        RMrcvdtkn_ptr        toknptr;
```

Invocation reply is complete — clean up invocation records.

The TSYNC packet has been received from the remote node (from which the invocation originated). Therefore, the reply sequence has been completed, and the records for this invocation should be removed.

The watchdog timer for the receipt of the TSYNC packet is canceled. In case the watchdog timer expired before the TSYNC packet was received, the cleanup will have already been done.

Routine:

```
void    OutReply_Timeout_RBI_action(toknptr)
        RBItimertkn_ptr toknptr;
```

Thread synchronization (TSYNC) packet not received in time.

The watchdog timer for the receipt of the thread synchronization packet has expired. Given communication queues, delays, and errors, it is difficult to know the true situation. Therefore, clean up the invocation records as if the reply had been completed successfully.

Routine:

```
void InRequest_RBI_action(toknptr)
        RMrcvdtkn_ptr     toknptr;
```

Handle an incoming operation invocation from another node. The request side of the invocation parameter block is extracted from the incoming packet and placed in shared Multibus memory. The AP is notified of the arrival of the new invocation, and this node begins to participate in the thread maintenance of the new thread section (also indicating that the head of the thread is local to this node). A thread synchronization (TSYNC) packet is sent to the invoking node to indicate that this node has accepted responsibility for reporting the location of the thread's head. (The TSYNC packet is sent via the RM protocol.)

Finally, necessary bookkeeping is done to facilitate the tracking of this invocation.

Routine:

```
void InReply_RBI_action(toknptr)
        RMrcvdtkn_ptr     toknptr;
```

Handle invocation reply from another node. This is a reply to a previous invocation request. If the invocation has not been aborted, the reply is processed. In particular, for purposes of thread maintenance, the thread's head has now officially returned to the invoking node (this node). Also, a thread synchronization (TSYNC) message is composed and sent (via the RM protocol) to the replying node to indicate that the transfer of responsibility for the head of the thread has occurred. Finally, the reply side of the invocation parameter block is extracted and placed in the original invocation parameter block passed to the CP to initiate the operation invocation.

Routine:

```
void InReply_Done_RBI_action(toknptr)
        RMsenttkn_ptr     toknptr;
```

Complete the reply phase of an operation invocation. At this point, the thread synchronization (TSYNC) message has been successfully sent to the replying node. If the thread section is not being aborted, then the reply processing is completed now. Specifically, the AP is signalled that the invocation has completed, and the necessary bookkeeping is done to wrap up this invocation.

Routine:

```
void  AbortInvoke_RBI_action(toknptr)
        TMARRIaborttkn_ptr     toknptr;
```

Terminate an invocation by sending a 'THREAD_BREAK' status to the invoking object. The invocation record for the designated invocation is accessed and a 'THREAD_BREAK' status is stuffed into the corresponding invocation parameter block. The AP is then signalled that the invocation has completed. Normal bookkeeping is performed to remove the RBI protocol's records for the invocation.

Routine:

```
void   AbortThread_RBI_action(toknptr)
       RBIabortthreadtkn_ptr      toknptr;
```

Delete a thread section's invocation records and data (as part of the processing that aborts that thread section). The ongoing invocations for the thread are examined in order to find those for the designated section. Once found, they are eliminated and any memory used for them is deallocated. After this has been done for both the incoming and outgoing invocations for the section (if both are present), the TMAR protocol — which coordinates the trimming of threads — is notified that this cleanup has completed.

Routine:

```
void   CleanupOutRequest_RBI_proc(threadid, sectnum)
       abitID        threadid;
       u_short       sectnum;
```

Remove the outgoing invocation record for the specified thread section. This is done by accessing the thread's record and finding the entry for the specified section. That entry is removed and the other RBI protocol record concerning the outgoing invocation for that section is removed.

Routine:

```
void   Delete_RBI_action(toknptr)
       RMrcvdtkn_ptr      toknptr;
```

Delete the packet that has been received.

Routine:

```
SIPBptr GetSchedParms_RBI_proc(threadid, sectnum, timestampptr)
       abitID        threadid;
       u_short       sectnum;
       u_int         *timestampptr;
```

Return a pointer to scheduling parms that were used for the outgoing operation invocation by section 'sectnum' of thread 'threadid'.

3.3.3.17 Reliable Message Protocol — rm_proto.c

This C source file contains logic for the Reliable Message Protocol.

Routine:

```
void   InitRM()
```

Initialize RM private data structures, including Ethernet address, a set of records to track individual messages, and various statistics.

Routine:

```
void   RM_TknHdlr(toknptr)
       tokenptr      toknptr;
```

Token handler for tokens intended for the RM protocol. Based on the source and type of the token, an action routine is selected to handle the token.

Routine:

```
void    SendMsg_RM_action(toknptr)
        sendRMtkn_ptr      toknptr;
```

Send a reliable message to a specified (logical) Ethernet address.

This is done by constructing a record to track the reliable message transmission, encapsulating the packet to be sent inside a reliable message frame/packet. A copy of the packet is kept in case it must be retransmitted, and a watchdog timer is set in case an acknowledgement is not received within a reasonable time.

Routine:

```
void MsgRcvd_RM_action(toknptr)
        enettkn_ptr toknptr;
```

Handle reliable message that was just received. First, acknowledge receipt of message. Then check to see if the packet is a duplicate of one already received. If so, discard it. Otherwise, create a token for the protocol that is using RM and enter the token in the token queue. In all cases, update relevant statistics.

Routine:

```
void AckRcvd_RM_action(toknptr)
        enettkn_ptr        toknptr;
```

Handle reliable message acknowledgement. Normally, this means that the originator of the message is notified that the transmission was completed successfully, and information about the message is removed since it is no longer needed. Under exceptional conditions, the acknowledgement may have arrived too late (that is, after the watchdog timer for the acknowledgement has expired and removed the message record). In either case, relevant statistics are updated as required.

Routine:

```
void Timeout_RM_action(toknptr)
        RMtimeouttkn_ptrtoknptr;
```

Handle expiration of watchdog timer. This indicates that the acknowledgement has not been successfully received yet. Therefore, either the message should be retransmitted (if the maximum number of retries has not been reached) or no further attempts should be made. In the latter case, the originator of the message should be notified that the reliable message exchange was not completed. In each case the message record and the RM statistics are updated as required.

Routine:

```
boolean RMDuplicatePkt(convnum, delta)
        u_long      convnum;
        long        delta;
```

Return ’TRUE’ iff the packet is a duplicate of one that has already been received. This is done by checking to see if the sequence number of the reliable message (on the transmitting node) is one that has been received previously. A sliding window is conceptually slid along an infinite bit vector where each bit indicates a sequence number. The window is advanced as new packets are received, and it can be made wide enough to detect duplicates from a node with arbitrarily high probability.

Routine:

```
long   RMConvnumDelta(convnum)
       u_long        convnum;
```

Determine the relative position in the conversation number bit vector of the current conversation. The offset returned, 'delta', indicates:

- (RM_RCVD_BV_SIZE) < delta <= 0, conv is in bit vector;

0 < delta < (RM_MAX_CONVNUM_COUNT - RM_RCVD_BV_SIZE), else;

Routine:

```
void   RMRecordPkt(convnum, delta)
       u_long        convnum;
       long          delta;
```

Update the conversation bit vector to reflect the receipt of the packet with conversation number 'convnum' and relative offset 'delta' within the bit vector. Note that there is a separate bit vector for each node. Only one vector is changed by this routine for any given conversation.

This routine will "shift" the window that views part of the conversation number bit vector if the conversation reported indicates that this is a new packet.

3.3.3.18 Ethernet Controller Debug Support — se_debug.c

This C source file contains logic for the Sun Ethernet Controller debugger.

Routine:

```
void   PrintSECstrs()
```

This routine is used to print out various data structures.

Routine:

```
void   PrintENA(ENAptr)
       ENaddrptr     ENAptr;
```

Print an Ethernet Address.

3.3.3.19 Packet Filtering — se_filter.c

This C source file contains logic for Communication Virtual Machine Packet Filtering

Routine:

```
void   InitRcvMCA()
```

This routine sets up the MC address filter.

Routine:

```
void   RcvAddMCA(MCAptr)
       ENaddrptr     MCAptr;
```

This routine takes a pointer to an Ethernet address and adds it to the local list of multicast addresses.

If an entry already exists for this address, this routine has no effect.

If there is not enough space to add another mapping, a fatal error occurs.

Routine:

```
void   RcvRemoveMCA(MCAptr)
       ENaddrptr    MCAptr;
```

This routine takes a pointer to an Ethernet address and removes it from the local list of multicast addresses.

If no entry exists for the given address, this routine has no effect.

Routine:

```
boolean RcvLookupMCA(MCAptr)
       ENaddrptr    MCAptr;
```

This routine takes a pointer to an Ethernet multicast address and compares it against the list of currently active multicast addresses.

If the given address is in the list, TRUE is returned, otherwise it returns FALSE.

It returns a FALSE immediately if the given address is not a MCA.

Routine:

```
void   RcvSetupENA(addrptr)
       ENaddrptr    addrptr;
```

This routine takes a pointer to an Ethernet address and sets the local ethernet controller's specific address equal to it.

Routine:

```
void   RcvSetPrmsc()
```

This routine sets the node's ethernet controller into promiscous receive mode (i.e., it receives all packets on the net).

This routine takes advantage of the fact that the configuration command block has been set up in the 'InitSECB()' routine, and just changes the conf word with the promiscuous receive bit in it.

Routine:

```
void   RcvClearPrmsc()
```

This routine takes the node's ethernet controller out of promiscous receive mode (i.e., it receives only those packets that are addressed specifically to the node, or in its multicast receive set).

Routine:

```
void   SetMCA()
```

Set up the MC addresses to be recognized by the IEC.

This routine uses the hash key table to create a setup MC command block and issues the command.

MC addresses with non-zero use counts in the hash key table are used to compose the MC command block.

The command block used is a static command block located in the MCB (and uses the array of ENaddrs directly following it).

Routine:

```
u_short     GenHashKey(MCAptr)
        ENaddrptr   MCAptr;
```

This routine takes a pointer to a MC address and returns the CRC hash key value for it.

This uses the same CRC-32 polynomial as used in the IEC.

3.3.3.20 Ethernet Controller Interface — se_intf.c

This C source file contains logic for the Sun Ethernet Controller Interface.

Routine:

```
void  InitSECB()
```

Master initialization routine.

This initializes the Sun Ethernet Controller Board (i.e, SECB) and the Intel Ethernet Controller IC (i.e., IEC).

This routine starts out by initializing the controller board, then it initializes the controller chip. The interrupt vector is set up, and the receive unit is started up. This is followed by enabling the interrupts from the board, and then issuing a configuration command

followed by a command to set up the node's individual address. The board is taken out of loopback mode and put into normal mode.

If an error should occur at any point, a message is printed and the routine aborts to the monitor.

More effort is made in this routine than others to ensure that the chip's status is cleared (i.e., it limits the number of times the code spins in the test loops).

Following the successful completion of this reset routine, the built-in configuration and individual address setup commands are set up, as well as the node's individual address (all in the MCB). These command blocks can be reused by "soft-reset" routines.

Also, a loopback test is carried out on the controller board. A minimum length packet of random data is sent, the code spins until the command is complete, then tries to receive the packet. The packet type is chosen to ensure that a SM is not available, and so that the packet is passed to bit bucket SM which disposes of the packet.

Routine:

```
void  ClearSECB()
```

This routine clears and disables the ethernet controller board.

Routine:

```
void  EnetIntrHdlr()
```

Ethernet interrupt handler routine. This is invoked when the IEC generates an interrupt. It deals with each of the four possible interrupt sources separately, by ack'ing the event to clear the interrupt and then carrying out the desired function.

When a command completion event occurs, the first command block in the CBL is checked for the status of the completion, removed from the CBL and added to the free list. If there are any other commands waiting in the CBL, the next command block is linked in place and the CU is started up.

When the interrupt signals that a packet has arrived, the packet is linked out of the RFA and put into a special received packet list, a token is generated with a pointer to the new packet, and the token is passed to the receive packet handling SM specified by the packet's type field.

If a SM can't be found for a given type of packet, the packet and token are passed to the system-defined, catch-all, "bit-bucket" SM.

When the interrupt signals that a packet has been sent, the command block is recycled, and the token associated with the transmit block is passed to the indicated SM (provided that a valid SMptr is given).

Routine:

```
void   MapEC()
```

Map the Sun Ethernet Controller into the processor's VAS.

All mappings are done at the top of the machine's address space and are done in integral multiples of segments.

Routine:

```
void   MapDev()
```

Map the devices (Timer/UART/InPort) into the virtual address space.

Put things into their default locations.

This routine is only used for Sun-2 CPU's.

3.3.3.21 Ethernet Controller — se_intrnl.c

This C source file contains logic for the Sun Ethernet Controller internals.

Routine:

```
void   InitSEB()
```

This routine initializes the Sun Ethernet controller board.

It starts by setting up the board's page map, and clears the controller's on-board memory.

The page map is set up to identity map all of the physical pages, to doubly map the first and last pages, and to invalidate all of the remaining pages (by making them go to non-existent memory).

Routine:

```
void   InitIEC()
```

This routine initializes the Intel Ethernet controller IC.

It starts by setting up the root initialization structures (i.e., the SCP, ISCP, and SCB), it then sets up the Control Block Area (i.e., the generic Command Blocks, Transmit Buffer Descriptors, and Transmit Buffers), followed by the setting up of the Receive Buffer Area (i.e., Receive Frame Descriptors, Receive Buffer Descriptors and Receive

Buffers).

Finally, it sets the board's loopback mode to 'normal' to allow packets to be received from the net.

Routine:

```
vaddr  InitCBA(CBaddr, numCB, TBaddr, numTB, TBsize)
       vaddr      CBaddr;
       u_short    numCB;
       vaddr      TBaddr;
       u_short    numTB;
       u_short    TBsize;
```

This initializes the command block area by creating, initializing, and linking together a given number of generic command blocks.

Generic command blocks are large enough to contain the largest command block (i.e., CONFIGURE), and have common headers added to them.

All command blocks are initialized as NOP commands, have their last block flag set, are marked as reusable, and are all linked together.

The transmit buffer descriptors and transmit buffers are bound to each other, initialized, and all of these pairs are linked together.

The 'CBA' global data structure also contains pointers to the head and tail of both the command block and the transmit buffer lists. The command blocks and the TBD's start at the location in the Ethernet controller board's memory given by 'CBaddr', and the TB's start at the address given by 'TBaddr'.

This routine returns a pointer to the address following the end of the command block area.

Note that the counts must be greater than zero and the address must be valid (we don't do error checking here).

Routine:

```
vaddr  InitRFA(RFDaddr, numRFD, RBaddr, numRB, RBsize)
       vaddr      RFDaddr;
       u_short    numRFD;
       vaddr      RBaddr;
       u_short    numRB;
       u_short    RBsize;
```

This initializes the receive frame area.

This is done by creating, initializing, and linking together a number of frame descriptors and receive buffer descriptor and receive buffer pairs.

All of the RFD's and RBD/RB's are linked together, rooted in the SCB. The RBD/RB list is linked to the first RFD. The RFD's and RBD's start at the location in the Ethernet controller board's memory given by 'RFDaddr", and the RB's start at the address given by 'RBaddr'.

This routine returns the address of the end of the command block area.

Note that a bug in the IEC requires that you have an extra RFD at the end of the RFD list. We put an extra one at the end that always stays there (unless the RFA is depleted). Also note that the sizes must be legal because we don't check them here.

Routine:

```
void    IssueCmd(cmdptr, type)
        gcbptr      cmdptr;
        u_short     type;
```

Issue a command to the IEC.

This routine takes a pointer to a command block and an indication of the type of command that is to be issued, sets up the command block, and issues the command.

This routine only adds the given command to the end of the active command list and starts up the CU (if necessary).

The IEC will generate an interrupt at the completion of the command.

This uses a static command list (i.e., only one command is in the CBL at a time). While there can be many commands in the active list, each has its end-of-list bit set so the IEC sees only one command at a time.

The interrupt handler detects when a command has been completed, puts the CB at the end of the list and activates the next command (if there is an active one).

Routine:

```
gcbptr      GetCmdBlk(size)
        u_short     size;
```

Get a command block from the free list.

Takes a command block off of the head of the free list, gets the number of TBD/TB's needed to send the specified number of bytes, and returns a pointer to the allocated command block.

If a 'size' of zero is given, no transmit buffers are allocated for it. If the 'size' is greater than zero, the appropriate number of transmit descriptor/buffer pairs are allocated and linked to the allocated command block. If the transmit buffers are exhausted, an

error is indicated.

Within critical sections, interrupts from the SECB are inhibited to prevent interference from the interrupt handler.

The last of the aquired TBD's has its 'EOF' flag set and its 'linkoffset' set to nil. The actual count in each TBD must be set by the caller after the buffer is filled in, and before the CB is linked into the CBL.

Routine:

```
void    PutCmdBlk(CBptr)
        gcbptr      CBptr;
```

This routine takes a pointer to a command block and returns it to the free list.

This also returns any transmit descriptor/buffer pairs to the TBD free list.

Routine:

```
void    PutRcvBuf(RFDptr)
        rfdptr      RFDptr;
```

Return a receive block to the RFA.

This routine take a pointer to an RFD (with attached RBD's and RB's) and returns them to the end of the RFA.

RFD's are added to the end of the RFD list, and RBD's are added to the end of the RBD list.

When the RBD list is empty the RU is shut down, and if this is the case when this routine is called, the RU must be restarted. Also, interrupts from the SECB must be disable to ensure synchronization constraints are met.

The one extra RFD is always maintained in place.

3.3.3.22 Ethernet Utility Routines — se_util.c

This C source file contains logic for Low-level Sun Ethernet control.

Routine:

```
void    ENerror(msg)
        char          *msg;
```

This routine is called when a fatal error is detected in the Ethernet code.

The given message is printed and it traps to the monitor.

Routine:

```
iaddr   MaddrIaddr(addr)
        vaddr         addr;
```

Convert a Motorola address into Intel format.

Takes a standard virtual address, and converts it to an address which is relative to the Sun Ethernet Controller Board's memory and is in Intel byte order.

Routine:

```
vaddr   IaddrMaddr(addr)
        iaddr         addr;
```

Convert an Intel address into Motorola format.

Takes an address relative to the Sun Ethernet Controller Board's memory in Intel byte order, converts standard virtual address, and returns the virtual address.

Routine:

```
ioffset MaddrIoffset(addr)
        vaddr         addr;
```

Convert a Motorola address into an Intel offset.

Takes a standard virtual address, converts it to an offset which is relative to the base address in the System Configuration Pointer and is in Intel byte order, and returns the offset.

Routine:

```
vaddr   IoffsetMaddr(offset)
        ioffset       offset;
```

Convert an Intel offset into a Motorola address.

Takes an offset relative to the base address in the System Configuration Pointer and in Intel byte order, converts it into a standard virtual address, and returns the virtual address.

Routine:

```
void   GetENaddr(ENbuffer)
       ENAddrptr          ENbuffer;
```

Get this node's physical Ethernet address.

This is based on a random number with the last two bytes of the IP address.

Routine:

```
void   ENaddrCopy(addr1, addr2)
       ENaddrptr    addr1;
       ENaddrptr    addr2;
```

Copy an Ethernet address, 'addr1' into 'addr2'.

Routine:

```
boolean ENaddrCmp(addr1, addr2)
       ENaddrptr    addr1;
       ENaddrptr    addr2;
```

Compare a pair of Ethernet addresses.

Compare 'addr1' with 'addr2' and return 'TRUE' if they are the same, and 'FALSE' if not.

3.3.3.23 Thread Maintenance and Repair Protocol — tmar_proto.c

This C source file contains logic for the Thread Maintenance and Repair Protocol.

Routine:

```
void   InitTMAR()
```

Initialize TMAR private data structures, including a copy of the Ethernet address, a set of active thread records, and a list of all threads that are rooted on this node and also have sections on other nodes. In addition, a timer event is set up to trigger the first polling cycle.

Routine:

```
void   TMAR_TknHdlr(toknptr)
       tokenptr     toknptr;
```

Token handler for tokens intended for the TMAR protocol. Based on the source and type of the token, an action routine is selected to handle the token.

Routine:

```
void   PollNodes_TMAR_action()
```

Refresh rooted threads and poll for current status.

This action routine refreshes the unbroken portion of those threads that are rooted at this node and also polls other nodes for the current status of all of the sections of threads that are rooted locally.

The first portion of the routine determines the current state of each thread that is rooted locally. Specifically, it checks to see that all of the non-local thread sections from the root to the head, inclusive, have responded to a polling packet recently. If so, then the thread is unbroken, and the entire thread is refreshed. Otherwise, the point of the break is determined, and only the unbroken portion of the thread is refreshed.

A refresh packet is constructed in two phases. First, the number of unbroken threads is included in the packet, followed by the list of unbroken threads. Next, the number of broken threads is appended to the packet, along with the identifiers and required information concerning the break (the point of the break and the scheduling parameters to be used to schedule the thread repair, if known). The refresh packet is broadcast to all of the nodes in the system.

Any local thread sections that must be aborted as a result of a thread break will be identified, and the kernel will be notified to abort them.

After the refresh packet has been sent, a new polling packet is constructed. It contains a list of all of the threads that are rooted locally. It is broadcast to all of the nodes in the system, and its purpose is to solicit information concerning these threads from the other nodes.

Finally, the next execution of this action routine is scheduled by setting an interval timer.

Routine:

```
void   NodeResponse_TMAR_action(toknptr)
       enettkn_ptr        toknptr;
```

This action routine processes a response to a polling packet sent by this node. Initially, the response is checked to verify that it corresponds to the current polling cycle. If it does not, then it is discarded. Otherwise, the information in the packet is extracted. Then each thread included in the response is examined.

The thread sections included in the response, including the head section, are marked to indicate that they are still active. Also, if the response indicates that scheduling parameters are present for a thread, then these scheduling parameters are recorded and timestamped. (This happens following a thread break. The break is broadcast to the system via a refresh packet. When the node containing the thread section that will be the head of the repaired thread processes the refresh packet, it will send a response packet that contains the scheduling parameters of the trimmed head section, as well as the usual thread section info. These scheduling parameters will then be included in future refresh packets for that thread — until the repair is completed or the scheduling parameters are superceded for some other reason.) In this last case, the identity of the node that sent the scheduling parameters for the thread is also recorded for future use.

Routine:

```
void   PollReceipt_TMAR_action(toknptr)
       enettkn_ptr        toknptr;
```

Respond to a polling packet from another node. Extract the list of threads from the polling packet and check to see if any sections of those threads are on this node. If any sections are local, a response packet is composed containing the requested information for each such thread (i.e., the sections on this node and whether or not the head is local). Finally, the response packet is sent to the polling node.

Routine:

```
void   RefreshReceipt_TMAR_action(toknptr)
       enettkn_ptr        toknptr;
```

Process a refresh packet. The packet was sent by the node on which each thread referred to in the packet is rooted. The packet has two parts — first, a list of unbroken threads for which every section should be refreshed; and, second, a list of broken threads for which only those sections preceding the thread break (that is, those sections that lie between the thread's root and the point of the break) should be refreshed.

This action routine begins by extracting the list of unbroken threads from the refresh packet. For each thread in this list that has at least one section on this node, the thread is refreshed. (This means that the interval timer event that was acting as a watchdog timer for the thread is superceded by a new interval timer event that is further in the future.)

Next, the list of broken threads is extracted from the refresh packet. For each of these threads that have sections on the local node, only the sections preceding the break are refreshed. If the local node has any sections beyond the break (i.e., between the break and the head), these sections must be aborted. The kernel is notified if any sections must be aborted. (Note: it is only notified once for any given section. TMAR keeps track of those sections that the kernel is aborting, and only sends additional notifications if additional sections must be aborted.) The kernel notifies the TMAR protocol after completing the abortion of any given thread section. If all of the necessary sections have been aborted, this routine sends the TMAR protocol at the node coordinating the thread trimming a reliable message (using the RM protocol) indciating that this has happened.

If a broken thread's trimmed head will resume its execution at this node, then this node knows the scheduling parameters by which the trimmed thread should be scheduled. These same parameters are used to schedule the abortion of any sections that must be trimmed following a thread break. Therefore, this routine will send a response packet to

the coordinator node listing these scheduling parameters for all of the broken threads whose trimmed heads will resume execution on this node, but whose scheduling parameters were not included in the refresh packet. (Until they receive these parameters, the thread sectionaborts are scheduled using null parameters. This allows the possibility that some work on the abort may be done before the proper parameters arrive.)

Routine:

```
void    RefreshTO_TMAR_action(toknptr)
        TMARrefreshTOtkn_ptr     toknptr;
```

Abort (presumably) rootless thread.

All threads should have their sections refreshed regularly. If this timer goes off, then one of two situations exist: either (1) the timer is no longer valid (it has been superceded by another timer event due to a thread refresh), in which case this timeout should be ignored; or (2) the timer is still valid, in which case the thread has not been refreshed in a

good while and must now be presumed rootless. In the later case, the local thread sections must be aborted. The kernel is notified of this fact, and the CP's records for the thread are deallocated.

Notice that there is a trade-off made in the TMAR protocol: the price for being more certain that a thread break has occurred, or in this case that a thread's root has died, is slower responsiveness to these events. If you are willing to occasionally abort threads unnecessarily (e.g., due to a lost refresh message), then you can respond more quickly to breaks when they do occur.

Routine:

```
void   BcastRetry_TMAR_action(toknptr)
       TMARbcastretrytkn_ptr      toknptr;
```

This action routine retransmits a multicast message, if necessary.

Initially, it checks to see whether the message — which has already been sent at least once — has been acknowledged. If it has, then no retransmission is required, resources are deallocated, and the routine is exited. Otherwise, the packet is recomposed (it can't simply be resent as is because scheduling parameters contained in the packet must be adjusted to take into account the passage of time since the last transmission) and resent, and a timer event is declared to act as a watchdog in case an acknowledgement is not received "soon".

Note: This routine is one of a set that comprise a potentially useful facility. For now they have been constructed to meet the requirements of the TMAR protocol and have been included as part of that protocol, but in the future they could be generalized to be more widely applicable.

Routine:

```
void RepairThread_TMAR_action(toknptr)
       RMrcvdtkn_ptr         toknptr;
```

This action routine processes a repair thread packet. The packet contains information about threads that have been trimmed and are now ready to resume execution. Each thread in the packet should have its trimmed head section on this node. If that is not the case, an error condition is signaled. Each repaired thread resumes execution by returning a 'THREAD_BREAK' return status to the invoker at the head of the repaired thread. TMAR queues a token cause the RBI protocol to terminate the invocation with a thread break indication for each repaired thread in the repair packet.

This routine is similar to 'ActivateThread_TMAR_action()'. Both handle packets that indicate that one or more threads have been trimmed. However, one of the is used when the thread trim was performed as a result of a thread break ('RepairThread_TMAR_ action()'), while the other is used when the thread trim was performed as a result of an AP-initiated abort command ('ActivateThread_TMAR_action()')—due to a missed deadline, for example.

Routine:

```
void ActivateThread_TMAR_action(toknptr)
       RMrcvdtkn_ptr         toknptr;
```

This action routine processes an activate thread packet. The packet contains information about threads that have been trimmed at the request of the AP and are now ready to resume execution. Each thread in the packet should have its trimmed head section on this node. If that is not the case, an error condition is signaled. The outgoing invocation for each repaired thread is discarded, and the AP is signaled that the abort has completed and that normal execution can continue.

This routine is similar to 'RepairThread_TMAR_action()'. Both handle packets that indicate that one or more threads have been trimmed. However, one of the is used when the thread trim was performed as a result of a thread break ('RepairThread_TMAR_ action()'), while the other is used when the thread trim was performed as a result of an AP-initiated abort command ('ActivateThread_TMAR_action()')—due to a missed deadline, for example.

Routine:

```
void BeginAbortRqst_TMAR_action(toknptr)
    enettkn_ptr        toknptr;
```

Process a broadcast packet requesting a thread abort. Since the packet was broadcast, a check is first made to determine if the thread is rooted locally. If it is, then a second test is done to verify that this is a request to trim the thread further back than any other thread repair that is already in progress. If the abort request passes this test, then the trim parameters are recorded, the kernel is signaled if any local thread sections must be aborted as a result of the abort request, and an explicit acknowledgement is sent to the originator of the thread abort packet being processed. In addition, if TMAR optimizations are enabled, a refresh packet is sent out indicating the section to which the thread should be trimmed along with the appropriate scheduling parameters for the trimmed thread.

Routine:

```
void Ack_TMAR_action(toknptr)
    enettkn_ptr        toknptr;
```

Record the fact that a broadcast packet, sent on behalf of a thread, has been acknowledged by the desired node.

Note: This routine is one of a set that comprise a potentially useful facility. For now they have been constructed to meet the requirements of the TMAR protocol and have been included as part of that protocol, but in the future they could be generalized to be more widely applicable.

Routine:

```
void TSectsAborted_TMAR_action(toknptr)
    RMrcvdtkn_ptr      toknptr;
```

Process a packet that reports newly aborted thread sections. If the thread is rooted locally — which it should be since the packet should have been explicitly addressed to the coordinator (root) node — then the newly aborted thread sections are recorded in the thread's record.

If TMAR optimizations are enabled and these sections are the last that need to be aborted, then a repair thread packet is sent immediately to allow the trimmed thread to execute.

Routine:

```
void BeginAbort_TMAR_action(toknptr)
    TMARbeginabortshorttkn_ptr      toknptr;
```

Process a command from the AP requesting a thread abort. There are two cases to consider. First, if the thread is rooted locally, then this is the node that will coordinate the abort and processing can begin. Specifically, for locally rooted threads, a test is done to verify that this is a request to trim the thread further back than any other thread repair that is already in progress. If it is, then the trim parameters are recorded and the kernel is signaled if any local thread sections must be aborted as a result of the abort request. In addition, if TMAR optimizations are enabled, a refresh packet is sent out indicating the section to which the thread should be trimmed along with the appropriate scheduling parameters for the trimmed thread.

In the second case, the thread is not rooted locally, so another node — the thread's root node — must coordinate the thread abort. Therefore, a packet is broadcast requesting that the thread be trimmed as specified in the packet. (The packet also includes the necessary scheduling parameters for the abort processing.)

Routine:

```
void AbortDone_TMAR_action(toknptr)
        TMARabortdonetkn_ptr    toknptr;
```

Indicate that a local thread section has been aborted. All of the work is actually done by a subroutine called by this action routine.

Routine:

```
void   Participate_TMAR_action(threadid, sectnum)
       bitID          threadid;
       u_short        sectnum;
```

Indicate that a thread section is local and participate in the TMAR refresh protocol to keep it alive. There are two major cases to consider — the thread is rooted locally or the thread is rooted remotely.

For locally rooted threads, there are two more cases to consider.

First, this might be a request naming the root section of the thread. In that case, an entry must be made for the section in the ’rootedT[]’ data structure and initialized to indicate that the thread has only one section, it is the head section, it is local, and it is healthy. An entry for the thread is also made in the ’Tindx’ set so that the thread’s ’rootedT[]’ entry can be found using the thread’s id as a key.

Secondly, the parameter ’sectnum’ is a section of a locally rooted thread, other than the root section. In that case, the ’rootedT[]’ entry for the thread is updated to show the presence of the new head section on the local node.

For remotely rooted threads, the same two subcases must be considered.

First, if the node is not currently participating in the TMAR refresh protocol for the thread (because there are no other sections of the thread on the local node now), then a new record is inserted in the ’activethreads’ set describing the thread. Also, a timer must be set to act as a watchdog for this thread. The timer will be used to insure that the thread is refreshed often enough.

Finally, if thread ’threadid’ is rooted remotely, but already has one or more sections on the local node, then the existing ’activethreads’ record is updated to include the new section. Note that no additional timer event is required. Each thread, not each thread section, has its own watchdog timer for thread refreshes.

Routine:

```
void   Withdraw_TMAR_action(threadid, sectnum)
       abitID         threadid;
       u_short        sectnum;
```

Indicate thread section is no longer local and stop participating in the TMAR refresh protocol for that section. The cases to be considered are the same as for ’Participate_TMAR_action()’.

If the section to be withdrawn is the root section of a locally rooted thread, then the ’rootedT[]’ record and ’Tindx’ set entry for that thread are removed. If the section is a non-root section of a locally rooted thread, then the section is marked non-local and the head is also indicated to be non-local.

If the section is the only section of the given, remotely-rooted thread to on the local node, then the 'activethreads' record for the thread is removed. For all other sections of remotely-rooted threads, the 'activethreads' record for the thread is updated to indicate that the section is and the thread's head are both non-local.

Routine:

```
void   RecordTHeadLocal(threadid, sectnum)
       abitID       threadid;
       u_short      sectnum;
```

Indicate that a thread's head is local — that is, active on this node. This routine determines whether the thread is rooted locally or remotely in order to determine which data structure to examine for the given thread. The thread record is then accessed and marked to indicate that the thread's head is local and the head is executing in section 'sectnum'.

Routine:

```
void   RecordTHeadRemote(threadid)
       abitID       threadid;
```

Indicate that a thread's head is remote — that is, not on this node.

This routine determines whether the thread is rooted locally or remotely in order to determine which data structure to examine for the given thread. The thread record is then accessed and marked to indicate that the thread's head is non-local.

Routine:

```
void   SectAbortDone_TMAR_proc(threadid, sectnum)
        bitID       threadid;
       u_short      sectnum;
```

Indicate that a thread section has been aborted. All of the work is actually done by another subroutine.

Routine:

```
void   MarkSectAborted_TMAR_proc(threadid, sectnum)
       abitID       threadid;
       u_short      sectnum;
```

Indicate that a thread section has been aborted. If the thread is rooted locally, the aborted section is recorded. In addition, if TMAR optimizations are enabled and this is the last section that must be trimmed to complete the thread repair, then the thread is allowed to resume execution at this point. (Of course, the scheduler must still select it to run, but it is made ready at this point, conceptually.)

If the thread is rooted remotely, then the aborted section is recorded. If it is the final thread section on this node that must be aborted during this thread trim operation and if TMAR optimizations are enabled, then a packet is immediately sent to the thread's trim coordinator (root node) indicating that these sections have aborted.

(Of course, at least one refresh packet must have been received indicating that the thread was broken in order for the node to know whether the necessary sections had all been aborted.)

Routine:

```
void   ResumeThread_TMAR_proc(indx)
       u_short        indx;
```

Resume execution of a thread that has been trimmed; this routine is called only on the root node of the trimmed thread; 'indx' is the index of the thread's record in the 'rootedT[]' array; this routine consults that record to determine whether the trimmed thread should be repaired (in the case of a thread break) or activated (in the case of an abort); it handles all of the book-keepping associated with the change of status from "trim in progress" to "normal."

Routine:

```
boolean        intactT(headfound, headsect, sects)
       boolean        headfound;
       short          headsect;
       BIT_VECTOR_PTR(sects);
```

Determine if a thread is intact (that is, unbroken). The thread is broken iff it has no head or at least one section is missing (between the root and the head sections of the thread, inclusive).

Routine:

```
boolean        TrimmingSect_TMAR_proc(threadid, sectnum)
       abitID         threadid;
       u_short        sectnum;
```

This routine looks up the thread's TMAR record and determines whether or not this section is currently being trimmed by examining the "triminprog' and 'headsect/newheadsect' fields of the applicable thread record.

Routine:

```
boolean        TrimDone_TMAR_proc(indx)
       u_short        indx;
```

This routine looks up the thread's TMAR record and determines whether or not this thread has been completely trimmed; this routine is called only on the root node of the trimmed thread; 'indx' is the index of the thread's record in the 'rootedT[]' array.

Routine:

```
void   UpdateSchedParms_TMAR_proc(sipbptr, timestamp)
       SIPBptr        sipbptr;
       u_int          timestamp;
```

Adjust scheduling parameters to compensate for passed time.

The scheduling parameters passed into this routine 'sipbptr' were valid at time 'timestamp'. The system clock is read to learn the current time and to determine the amount of time that has passed since 'timestamp' (in scheduler clock ticks). All of the fields in the scheduling parameter block that are relative to some future time are adjusted for this passage of time.

It is probably easiest to call this routine with a pointer to a copy of a timestamped sipb. In this way, the information in the original sipb is not corrupted in case subsequent adjustments must be made, and the timestamp associated with the original sipb never has to change.

Routine:

```
u_long          GetTimerID_TMAR_proc()
```

Get a unique identifier for an interval timer event.

This routine generates a set of unique (for all practical purposes) timer event identifiers. Each TMAR action or routine calls this routine prior to scheduling an interval timer event. The value returned by this routine is then passed to the interval timer as a parameter that is returned upon expiration of the timed interval.

By comparing the value returned by the timer with a stored value indicating which timer(s) are currently in effect, each action or routine can reliably determine whether or not to execute a timeout action.

This routine will never return NIL.

4 Standalone Program Files

The standalone software is used as a basic skeleton for all of the major subsystems, and for all of the tools (such as diagnostics) which must run on the testbed. Essentially, the standalone files contain low-level information about starting up a high-level language program on a set of bare circuit boards. Much of this material deals specifically with low-level hardware specifics, such as setting up an initial mapping into the memory management hardware.

4.1 Assembler Files

4.1.1 C Language Run-Time — crt0.a

This file contains the initial startup code, which is entered directly from the code on the ROM. It expects that it is starting up a C program, which begins at routine "main()" (in the normal manner of C programs).

Routine:

entry

Cold start begins at this address. Assumes the initial conditions described in the documentation for the Sun ROMS.

4.1.2 Exception Handler Stub — excep.a

This file contains an interrupt handler stub.

Routine:

_Interrupt

This is a generic interrupt handler.

It pushes the registers on the stack, disables interrupts, and calls "_IntrHdlr", which should be written in high level language, and would therefore be without the ability to do these things. The interrupt hardware's vector offset is passed in register D0, so the high level software is able to distinguish the different possible interrupts.

When the call returns, the registers are popped, and return-from-interrupt is done.

4.2 Include Files

4.2.1 Event Logging Constants — SAEVcons.h

This file defines constants related to event logging.

Constants:

LE_BUF_SIZE
MAX_L_EVENTS
NUM_L_EVENT_TYPES

4.2.2 Standalone Code Constants — SAcons.h

This file defines memory-related constants.

Constants:

MB_MEM_BASE
MB_MEM_LIMIT
MAX_SYS_STACK_SIZE

MAX_USER_STACK_SIZE

4.2.3 Fast Memory Allocator Header — falloc.h

This file defines data structures which support a standalone fast, fixed-size storage allocator.

Data Structures:

fallocptr
fallocun

4.2.4 Event Logging Header — sa_event.h

This file defines data structures which support standalone event logging.

Data Structures:

eventptr
eventstr

4.3 Source Files

4.3.1 Fast Memory Allocator — falloc.c

This C source file contains logic for the Standalone, Fast, Fixed-Size, Storage Allocator.

Routine:

```
void    FallocInit()
```

Initialize the storage allocator.

This is a simple storage allocator for standalone programs.

It operates in fixed-sized 'falloc' storage unit defined by the SA programmer as union of all possible structures to be allocated. It is extremely fast both in allocation and deallocation since it deals with fixed size units. (yes, speed is the key word here!!!)

The routine first calculate the amount of free memory left and size them up into 'falloc' units. These storage are then linked up into a linked list of free storage units to be used by 'alloc()'.

Routine:

```
vaddr  Falloc()
```

Routine that return a pointer to a free block.

A 'NULL' pointer will be returned if there are no available free blocks.

Routine:

```
void    Fdealloc(fptr)
        fallocptr      fptr;
```

Deallocate a storage unit and return it to the free list.

4.3.2 Example Interrupt Handler — intrhdlr.c

This C source file contains logic for the example interrupt handler.

Routine:

```
void    IntrHdlr(vector)
        char   vector;
```

Generic interrupt handler routine.

Handle interrupts from the on-board interval timer.

Routine:

```
void    IPIIntrHdlr(level)
        u_long      level;
```

IPI interrupt handler routine.

This routine handles a whole range of interrupt levels, demultiplexes them and shows what can be done for IPI interrupts.

This is called by the assembly language generic interrupt stub, is passed the level of the interrupt, and it is handled based on this information.

4.3.3 Example Debugger — sa_debug.c

This C source file contains logic for the standalone debugger.

Routine:

```
void    SDBug()
```

This routine provides a common user interface to a collection of debugging routines.

The main debugging aid provided by this routine involves the printing of various data structures.

This routine also provides access to the routine that prints out the events in the log area.

The interface is menu-driven and an attempt is made to make the printing be "screen-oriented". At first the page size is given and a "more"-like function is provided: it prints a screen, prompts and waits for a response. Currently legal responses are: 'q' to quit printing, 'CR' for one more line, and any other character to print another full screen. Also, an 's' stops the paging (streaming the output), and any character typed at this point puts it back into paged mode.

Routine:

```
void    PrintEvents()
```

Print all events in the event log buffer onto the console.

Each event is printed out on a separate line starting with the event number, followed by the event name (a string), the event type and parameter (both single byte integers), the event's timestamp, and finally the delta off of the last event (in decimal microseconds,

with the event logging overhead deducted).

The event buffer wraps around, so it is possible that the first event printed is not event #0.

This routine starts up by printing the compensation time, the size of the event log buffer, and the number of events logged so far.

It then prompts the user for event to start printing with, and the types of events to show or hide.

When running the "more" banner comes up after each screen full of events, the following commands are accepted in more mode:

' ' — next page

'^M' — next line

'q' — stop printing

's' — enter stream mode

’/n’ — skip n events

’.n’ — goto event number n

In stream mode the events are printed non-stop until the end of the buffer is reached or any character is typed (at which point it returns to more mode).

4.3.4 Event Logger Example — sa_event.c

This C source file contains logic for the standalone event logger.

Routine:

```
void    InitEvents()
```

Initialize the event logging area, initialize the event timer, and log these actions as the first event.

The event buffer logging area is in the segment at the top of physical memory.

It is assumed that the system timer is initialized elsewhere.

It is assumed that the MMU is set to the identity map.

Routine:

```
void    LogEvent(id, parm)
        u_char        id;
        u_char        parm;
```

This routine is used to log an event. It uses the ’eventcount’ global variable as an index into the event log buffer, it gets the current value of the event timer and puts it, and the given parameter, into the event record indexed by ’eventcount’.

Interrupts must be disabled during this routine because we have to maintain the ordering of the events in the log (an interrupt could result in another event being logged).

4.3.5 Example Standalone Program — sa_prog.c

This C source file contains an example standalone program.

Routine:

```
void    SAInit()
```

This program just prints a message and returns.

4.3.6 Initialization Code — startup.c

This C source file contains logic for startup of a standalone program, such as sa_prog.c.

Routine:

```
void    main()
```

This routine sets up a Sun 1.5 processor board for standalone use.

Starts by disabling interrupts and printing a banner. It then maps multibus memory into the processors address space and invalidates the unused parts of the space. Next, the interprocessor interface is initialized along with the processor’s system timer. Following this, the event logging facility is initialized, a message is printed with the address of the standalone debug routine, and the user’s standalone program is called.

Interrupts are disabled and stack pointers are set up by ’crt0.a’.

Routine:

```
void    Error(string)
        char *string;
```

Fatal Error reporting. Prints and then traps to ROM.

5 Acknowledgments

Duane Northcutt developed most of the programming conventions described herein. He created the initial Emacs mocklisp support for code templates and formatting, which involved writing new code and adapting existing code for Alpha. He did the modifications to Emacs and YAPP described in the Appendix.

Sam Shipman made some minor refinements to the initial Emacs mocklisp support for code templates and formatting, and added the comment formatting support. He also experimented with Indent to find the set of options in Figure 7.

References

[Northcutt 88a] Northcutt, J. D.
The Alpha Operating System: Requirements and Rationale.
Archons Project Technical Report #88011, Department of Computer Science, Carnegie-Mellon University, January, 1988.

[Northcutt 88b] Northcutt, J. D. and Clark, R. K.
The Alpha Operating System: Programming Model
Archons Project Technical Report #88021, Department of Computer Science, Carnegie-Mellon University, February, 1988.

[Northcutt 88c] Northcutt, J. D. and Shipman, S. E.
The Alpha Operating System: Programming Utilities.
Archons Project Technical Report #88041, Department of Computer Science, Carnegie-Mellon University, April, 1988.

[Shipman 88] Shipman, S. E.
The Alpha Operating System: Programming Language Support.
Archons Project Technical Report #88042, Department of Computer Science, Carnegie-Mellon University, April, 1988.

Appendix I: Coding Convention Support Tools

There are some tools that help us with our coding conventions and take advantage of these conventions to give us info. They are: emacs mocklisp support, for generating code according to our conventions; indent options for formatting code according to our conventions; and yapp modifications for generating formatted source code hardcopies.

I.A Emacs

Emacs is a powerful screen-oriented editor that allows user customization by writing editing functions in an interpreted programming language. The variety of Emacs used in developing Alpha, known as Gosling Emacs (written by James Gosling), provides an extension language called mocklisp. In addition to the standard mocklisp code provided with Emacs that allows rapid edit/compile/debug cycles, we have created some Alpha-specific functions that:

- automatically controls indenting, paranthesis/braces matching, etc. on input/ editing of source files
- inserts personalized timestamps (for comments)
- maintains history list (automatically prepends changes, generates timestamped and personalized header)
- generates templates (file headers, comments, section dividers, routine templates, etc.)
- generates different comment formats (single-line, multi-line, block, header, temporary, etc.) and allows editing and regeneration of comments (e.g., block reformatting)

In addition, we have also created a device driver to allow Gosling Emacs to be used with the (native frame-buffer of) Sun-2s and Sun-3s. The modifications include:

- testing if the framebuffer is already being used to run a window manager (in order to refrain from blindly seizing the display as it did previously)
- saving the current contents of the screen prior to running Emacs and then restoring it whenever the program is halted (also works for pausing the program)
- allowing user-selectable choice of black on white, or white on black display
- allowing user-selectable font
- enabling the Meta key (replacing the ESC hack)
- providing the maximum number of lines possible with the given font size

This results in greatly increased performance when using the display directly (over using the built-in UNIX *curses*-based terminal emulator).

I.B Indent

The Indent program is a C language code formatter that is sufficiently flexible that it can be configured to format source code to the specifications of our coding standards. Indent is used by mocklisp code to allow an Emacs user to format individual routines while editing. The options to Indent are explained in Figure 7. They are documented in such detail here because they provide details of our code formatting conventions.

-bap	*force a blank line after every procedure body*
-nbacc	*do not force blank lines around every conditional compilation block*
-bad	*force a blank line after every block of declarations*
-nbbb	*do not force a blank line before every block comment*
-bc	*force a newline after each comma in a declaration*
-br	*line up compound statements as follows:* *if (...) {* *code* *}*
-nbs	*disable forcing a blank after sizeof (treat as procedure call, not operator)*
-c41	*starting column for comments on code*
-cd41	*starting column for comments on declarations*
-cdb	*enables placement of comment delimiters on blank lines (Note: this program wants to format comments; these options are set to minimize interference with the Alpha standard commenting conventions.)*
-nce	*disable forcing "else" to immediately follow preceding "}"*
-ci4	*continuation line indentation relative to first line of statement*
-cli1	*indent case labels 1 tab stop relative to containing switch statement*
-d0	*line up comments not to the right of code directly with code (no indent)*
-di16	*indentation (in characters) from declaration keyword to identifier*
-eei	*indent continuation lines of expression part of if() or while() an extra level*
-nfc1	*disable formatting of comments starting in column 1*
-i4	*number of spaces for one indentation level*
-ip	*indent declarations of function parameters from the left margin*
-l75	*maximum length of an output line*
-lc75	*line length for block comments*
-lp	*line up code surrounded by parenthesis in continuation lines*
-npcs	*do not insert a space between the procedure name and the "(" in calls*
-psl	*name of procedure being defined goes in column 1, type on previous line*
-sc	*enable placement of asterisks at the left edge of all comments*
-nsob	*do not delete optional blank lines*

Figure 7: Indent Options

I.C YAPP

YAPP is a program in our local environment that generates listings that are nicely formatted for laser printers. It can produce landscape-mode double-column listings using proportionally spaced fonts with language keywords in boldface, comments in italics, etc. Alpha personnel modified the existing program to fix some bugs that prevented it from working correctly for our application. Additionally, it was modified to recognize AOL keywords [Shipman 88] and format them correctly, and to generate PostScript output. This program is used to generate standard listings of Alpha code.