EXHIBIT 13

Docket No. 375246US91RX

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

IN RE REEXAMINATION OF: Richard Michael NEMES

CONTROL NO:	90/011,426	GAU:	3992
FILED:	January 10, 2011	EXAMINER:	ANDREW NALVEN
FOR:	METHODS AND APPARATUS FOR INFORMATION STORAGE AND RETRIEVAL USING A HASHING TECHNIQUE WITH EXTERNAL CHAINING AND ON-THE-FLY REMOVAL OF EXPIRED DATA		

INFORMATION DISCLOSURE STATEMENT UNDER 37 CFR 1.555

COMMISSIONER FOR PATENTS
ALEXANDRIA, VIRGINIA 22313

SIR:

Patent holder(s) wishes to disclose the following information.

REFERENCES

■ Patent holder(s) wishes to make of record the reference(s) listed on the attached form PTO-1449 and/or accompanying documents from a corresponding foreign application. Copies of the listed reference(s) are attached, where required, as are either statements of relevancy or any readily available partial or full English translations of pertinent portions of any non-English language reference(s).

☐ Credit card payment is being made online (if electronically filed), or is attached hereto (if paper filed), in the amount required under 37 CFR §1.17(p).

RELATED CASES

☐ Attached is a list of patent holder's pending application(s), published application(s) or issued patent(s) which may be related to the present application. In accordance with the waiver of 37 CFR 1.98 dated September 21, 2004, copies of the cited pending applications are not provided. Cited published and/or issued patents, if any, are listed on the attached PTO form 1449.

☐ Credit card payment is being made online (if electronically filed), or is attached hereto (if paper filed), in the amount required under 37 CFR §1.17(p).

Respectfully submitted,

OBLON, SPIVAK, McCLELLAND,
MAIER & NEUSTADT, L.L.P.

Scott A. McKeown
Registration No. 42,866

Customer Number
22850
Tel. (703) 413-3000
Fax. (703) 413-2220
(OSMMN 02/10)

Form PTO 1449 (Modified) U.S. DEPARTMENT OF COMMERCE PATENT AND TRADEMARK OFFICE LIST OF REFERENCES CITED BY APPLICANT	ATTY DOCKET NO. 375246US91RX	CONTROL NO. 90/011,426
	INVENTOR(S) Richard Michael NEMES	
	FILING DATE January 10, 2011	GROUP 3992

U.S. PATENT DOCUMENTS

EXAMINER INITIAL		DOCUMENT NUMBER	DATE	NAME	CLASS	SUB CLASS	FILING DATE IF APPROPRIATE
	AA						
	AB						
	AC						
	AD						
	AE						
	AF						
	AG						
	AH						
	AI						
	AJ						
	AK						
	AL						
	AM						
	AN						

FOREIGN PATENT DOCUMENTS

		DOCUMENT NUMBER	DATE	COUNTRY	TRANSLATION YES	TRANSLATION NO
	AO					
	AP					
	AQ					
	AR					
	AS					
	AT					
	AU					
	AV					

OTHER REFERENCES (Including Author, Title, Date, Pertinent Pages, etc.)

	AW	BAWDEN, Alan, et al., LISP Machine Progress Report - Massachusetts Institute of Technology Artificial Intelligence Laboratory - Memo No. 444; August 1977; 29 pgs.
	AX	Copyright 1995 by Bao Phan, et al.; Key Management Engine for BSD; 09/28/1995; Pgs. 1-29; (DEF00007942-DEF00007970)
	AY	Copyright 1995 by Bao Phan, et al.; Declarations and Definitions for Key Engine for BSD; 09/28/1995; Pgs. 1-4; (DEF00007971-DEF00007974)
	AZ	

Examiner	Date Considered

*Examiner: Initial if reference is considered, whether or not citation is in conformance with MPEP 609; Draw line through citation if not in conformance and not considered. Include copy of this form with next communication to applicant.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

Memo No. 444 August 1977

LISP Machine Progress Report

by the Lisp Machine Group

Alan Bawden
Richard Greenblatt
Jack Holloway
Thomas Knight
David Moon
Daniel Weinreb

ABSTRACT

This informal paper introduces the LISP Machine, describes the goals and current status of the project, and explicates some of the key ideas. It covers the LISP machine implementation, LISP as a system language, input/output, representation of data, representation of programs, control structures, storage organization, garbage collection, the editor, and the current status of the work.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0643.

INTRODUCTION:

The LISP Machine is a new computer system designed to provide a high performance and economical implementation of the LISP programming language.

The LISP language is used widely in the artificial intelligence research community, and is rapidly gaining adherents outside this group. Most serious LISP usage has historically been on the DEC PDP-10 computer, and both "major" implementations (InterLisp at BBN/XEROX and Maclisp at M.I.T.) were originally done on the PDP-10. Our personal experience has largely been with the Maclisp dialect of LISP, which was originally written in 1965.

Over the years, dramatic changes have taken place in the Maclisp implementation. At a certain point, however, modification and reimplementation of a language on a given machine can no longer efficiently gloss over basic problems in the architecture of the computer system. We, and many others, believe this is now the case on the PDP-10 and similar time-shared computer systems.

Time sharing was introduced when it became apparent that computers are easier to use in an interactive fashion than in a batch system, and that during an interactive session a user typically uses only a small fraction of the processor and memory available; often during much of the time his process is idle or waiting, and so the computer can be multiplexed among many users while giving each the impression that he is on his own machine.

However, in the Lisp community there has been a strong trend towards programs which are very highly interactive, very large, and use a good deal of computer time; such programs include advanced editors and debuggers, the MACSYMA system, and various programming assistants. When running programs such as these, which spend very significant amounts of time supporting user interactions, time sharing systems such as the PDP-10 run into increased difficulties. Not only is the processor incapable of providing either reasonable throughput or adequate response time for a reasonable number of users, but the competition for main memory results in large amounts of time being spent swapping pages in and out (a condition known as "thrashing"). Larger and larger processors and memory, and more and more complex operating systems, are required, with more than proportionally higher cost, and still the competition for memory remains a bottleneck. The programs are sufficiently large, and the interactions sufficiently frequent, that the usual time-sharing strategy of swapping the program out of memory while waiting for the user to interact, then swapping it back in when the user types something, cannot be successful because the swapping cannot happen fast enough.

The Lisp Machine is a personal computer. Personal computing means that the processor and main memory are not time-division multiplexed, instead each person gets his own. The personal computation system consists of a pool of processors, each with its own main memory, and its own disk for swapping. When a user logs in, he is assigned a processor, and he has exclusive use of it for the duration of the session. When he logs out, the processor is returned to the pool, for the next person to use. This way, there is no competition from other users for memory; the pages the user is frequently referring to remain in core, and so swapping overhead is considerably reduced. Thus the Lisp Machine solves a basic problem of time sharing Lisp systems.

The user also gets a much higher degree of service from a Lisp machine than from a timesharing system, because he can use the full throughput capacity of the processor and the disk. Although these are quite inexpensive compared to those used in PDP-10 timesharing systems, they are comparable in speed. In fact, since disk access times are mainly limited by physical considerations, it often turns out that the disk used in a personal computer system is less expensive simply because of its smaller size, and has fully comparable throughput charactistics to the larger disk used by a timesharing system.

In a single-user machine, there is no penalty for interactiveness, since there are no competing users to steal a program's memory while it is waiting for its user to type. Thus the Lisp machine system, unlike time sharing systems, encourages highly interactive programs. It puts service to the user entirely ahead of efficiency considerations.

Another problem with the PDP-10 Lisp implementations is the small address space of the PDP-10 processor. Many Lisp systems, such as MACSYMA and Woods's LUNAR program, have difficulty running in an 18-bit address space. This problem is further compounded by the inefficiency of the information coding of compiled Lisp code; compilers for the PDP-10 produce only a limited subset of the large instruction set made available by the hardware, and usually make inefficient use of the addressing modes and fields provided. It is possible to design much more compact instruction sets for Lisp code. Future programs are likely to be quite a bit bigger; intelligent systems with natural language front ends may well be five or ten times the size of a PDP-10 address space.

The Lisp Machine has a 24 bit virtual address space and a compact instruction set, described later in this paper. Thus much larger programs may be used, without running into address space limitations. Since the instruction set is designed specifically for the Lisp language, the compiler is much simpler than the PDP-10 compiler, providing faster and more reliable compilation.

The Lisp machine's compact size and simple hardware construction are likely to make it more reliable than other machines, such as the PDP-10; the prototype machine has had almost no hardware failures.

Much of the inspiration for the Lisp Machine project comes from the pioneering research into personal computing and display-oriented systems done by Xerox's Palo Alto Research Center.

THE LISP MACHINE IMPLEMENTATION:

Each logged in user of the Lisp Machine system has a processor, a memory, a keyboard, a display, and a means of getting to the shared resources. Terminals, of course, are placed in offices and various rooms; ideally there would be one in every office. The processors, however, are all kept off in a machine room. Since they may need special environmental conditions, and often make noise and take up space, they are not welcome office companions. The number of processors is unrelated to the number of terminals, and may be smaller depending on economic circumstance.

The processor is implemented with a microprogrammed architecture. It is called the CONS Machine, designed by Tom Knight [CONS]. CONS is a very unspecialized machine with 32-bit data paths and 24-bit address paths. It has a large microcode memory (16K of 48-bit words) to accommodate the large amount of specialized microcode to support Lisp. It has hardware for extracting and depositing arbitrary fields in arbitrary registers, which substitutes for the specialized data paths found in conventional microprocessors. It does not have a cache, but does have a "pdl buffer" (a memory with hardware push-down pointer) which acts as a kind of cache for the stack, which is where most of the memory references go in Lisp.

Using a very unspecialized processor was found to be a good idea for several reasons. For one thing, it is faster, less expensive, and easier to debug. For another thing, it is much easier to microprogram, which allows us to write and debug the large amounts of microcode required to support a sophisticated Lisp system with high efficiency. It also makes feasible a compiler which generates microcode, allowing users to microcompile some of their functions to increase performance.

The memory is typically 64k of core or semiconductor memory, and is expandable to about 1 million words. The full virtual address space is stored on a 16 million word disk and paged into core (or semiconductor) memory as required. A given virtual address is always located at the same place on the disk. The access time of the core memory is about 1 microsecond, and of the disk about 25 milliseconds. Additionally, there is an internal 1K buffer used for holding the top of the stack (the PDL buffer) with a 200ns access time (see [CONS] for more detail).

The display is a raster scan TV driven by a 1/4 Mbit memory, similar to the TV display system now in use on the Artificial Intelligence Lab's PDP-10. Characters are drawn entirely by software, and so any type or size of font can be used, including variable width and several styles at the same time. One of the advantages of having an unspecialized microinstruction processor such as CONS is that one can implement a flexible terminal in software for less cost than an inflexible, hardwired conventional terminal. The TV system is easily expanded to support gray scale, high resolution, and color. This system has been shown to be very useful for both character display and graphics.

The keyboard is the same type as is used on the Artificial Intelligence Lab TV display system; it has several levels of control/shifting to facilitate easy single-keystroke commands to programs such as the editor. The keyboard is also equipped with a speaker for beeping, and a pointing device, usually a mouse [MOUSE].

The shared resources are accessed through a 10 million bit/sec packet switching network with completely distributed control. The shared resources are to include a highly reliable file system implemented on a dedicated computer equipped with state of the art disks and tapes, specialized I/O devices such as high-quality hardcopy output, special-purpose processors, and connections to the outside world (e.g. other computers in the building, and the ARPANET).

As in a time sharing system, the file system is shared between users. Time sharing has pointed up many advantages of a shared file system, such as common access to files, easy inter-user communication, centralized program maintenance, centralized backup, etc. There are no personal disk packs to be lost, dropped by users who are not competent as operators, or to be filled with copies of old, superseded software.

The complete LISP Machine, including processor, memory, disk, terminal, and connection to the shared file system, is packaged in a single 19" logic cabinet, except for the disk which is free-standing. The complete machine would be likely to cost about $80,000 if commercially produced. Since this is a complete, fully-capable system (for one user at a time), it can substantially lower the cost of entry by new organizations into serious Artificial Intelligence work.

LISP AS A SYSTEM LANGUAGE:

In the software of the Lisp Machine system, code is written in only two languages (or "levels"): Lisp, and CONS machine microcode. There is never any reason to hand-code macrocode, since it corresponds so closely with Lisp; anything one could write in macrocode could be more easily and clearly written in the corresponding Lisp. The READ, EVAL, and PRINT functions are completely written in Lisp, including their subfunctions (except that APPLY of compiled functions is in micro-code). This illustrates the ability to write "system" functions in Lisp.

In order to allow various low-level operations to be performed by Lisp code, a set of "sub-primitive" functions exist. Their names by convention begin with a "%", so as to point out that they are capable of performing unLispy operations which may result in meaningless pointers. These functions provide "machine level" capabilities, such as performing byte-deposits into memory. The compiler converts calls to these sub-primitives into single instructions rather than subroutine calls. Thus Lisp-coded low-level operations are just as efficient as they would be in machine language on a conventional machine.

In addition to sub-primitives, the ability to do system programming in Lisp depends on the Lisp machine's augmented array feature. There are several types of arrays, one of which is used to implement character strings. This makes it easy and efficient to manipulate strings either as a whole or character by character. An array can have a "leader", which is a little vector of extra information tacked on. The leader always contains Lisp objects while the array often contains characters or small packed numbers. The leader facilititates the use of arrays to represent various kinds of abstract object types. The presence in the language of both arrays and lists gives the programmer more control over data representation.

A traditional weakness of Lisp has been that functions have to take a fixed number of arguments. Various implementations have added kludges to allow variable numbers of arguments; these, however, tend either to slow down the function-calling mechanism, even when the feature is not used, or to force peculiar programming styles. Lisp-machine Lisp allows functions to have optional parameters with automatic user-controlled defaulting to an arbitrary expression in the case where a corresponding argument is not supplied. It is also possible to have a "rest" parameter, which is bound to a list of the arguments not bound to previous parameters. This is frequently important to simplify system programs and their interfaces.

A similar problem with Lisp function calling occurs when one wants to return more than one value. Traditionally one either returns a list or stores some of the values into global variables. In Lisp machine Lisp, there is a multiple-value-return feature which allows multiple values to be returned without going through either of the above subterfuges.

Lisp's functional orientation and encouragement of a programming style of small modules and uniform data structuring is appropriate for good system programming. The Lisp machine's micro-coded subroutine calling mechanism allows it to also be efficient.

Paging is handled entirely by the microcode, and is considered to be at a very low level (lower level than any kind of scheduling). Making the guts of the virtual memory invisible to all Lisp code and most microcode helps keep things simple. It would not be practical in a time sharing system, but in a one-user machine it is reasonable to put paging at the lowest level and forget about it, accepting the fact that sometimes the machine will be tied up waiting for the disk and unable to run any Lisp code.

Micro-coded functions can be called by Lisp code by the usual Lisp calling mechanism, and provision is made for micro-coded functions to call macro-coded functions. Thus there is a uniform calling convention throughout the entire system. This has the effect that uniform subroutine packages can be written, (for example the TV package, or the EDITOR package) which can be called by any other program. (A similar capability is provided by Multics, but not by ITS nor TENEX).

Many of the capabilities which system programmers write over and over again in an ad hoc way are built into the Lisp language, and are sufficiently good in their Lisp-provided form that it usually is not necessary to waste time worrying about how to implement better ones. These include symbol tables, storage management, both fixed and flexible data structures, function-calling, and an interactive user interface.

Our experience has been that we can design, code, and debug new features much faster in Lisp-machine programs than in PDP-10 programs, whether they are written in assembler language or in traditional "higher-level" languages.

INPUT/OUTPUT:

Low level:

The Lisp Machine processor (CONS) has two busses used for accessing external devices: the "XBUS", and the "UNIBUS". The XBUS is 32 bits wide, and is used for the disk and for main memory. The UNIBUS is a standard PDP-11 16-bit bus, used for various I/O devices. It allows commonly available PDP-11 compatible devices to be easily attached to the Lisp Machine.

Input/output software is essentially all written in Lisp; the only functions provided by the microcode are %UNIBUS-READ and %UNIBUS-WRITE, which know the offset of the UNIBUS in physical address space, and refer to the corresponding location. The only real reason to have these in microcode is to avoid a timing error which can happen with some devices which have side effects when read. It is Lisp programs, not special microcode, which know the location and function of the registers in the keyboard, mouse, TV, and cable network interfaces. This makes the low-level I/O code just as flexible and easy to modify as the high level code.

There are also a couple of microcoded routines which speed up the drawing of characters in the TV memory. These do not do anything which could not be done in Lisp, but they are carefully hand-coded in microcode because we draw an awful lot of characters.

High level:

Many programs perform simple stream-oriented (sequential characters) I/O. In order that these programs be kept device-independent, there is a standard definition of a "stream": a stream is a functional object which takes one required argument and one optional argument. The first argument is a symbol which is a "command" to the stream, such as "TYI", which means "input one character, and return it" and "TYO", which means "output one character". The character argument to the TYO command is passed in the second argument to the stream. There are several other standard optional stream operations, for several purposes including higher efficiency. In addition particular devices can define additional operations for their own purposes.

Streams can be used for I/O to files in the file system, strings inside the Lisp Machine, the terminal, editor buffers, or anything else which is naturally represented as sequential characters.

For I/O which is of necessity device-dependent, such as the sophisticated operations performed on the TV by the editor, which include multiple blinkers and random access to the screen, special packages of Lisp functions are provided, and there is no attempt to be device-independent. (See documentation on the TV and network packages).

In general, we feel no regret at abandoning device independence in interactive programs which know they are using the display. The advantages to be gained from sophisticated high-bandwidth display-based interaction far outweigh the advantages of device-independence. This does mean that the Lisp machine is really not usable from other than its own terminal; in particular, it cannot be used remotely over the ARPANET.

REPRESENTATION OF DATA:

A Lisp object in Maclisp or InterLisp is represented as an 18 bit pointer, and the datatype of the object is determined from the pointer; each page of memory can only contain objects of a single type. In the Lisp machine, Lisp objects are represented by a 5 bit datatype field, and a 24 bit pointer. (The Lisp machine virtual address space is 24 bits). There are a variety of datatypes (most of the 32 possible codes are now in use), which have symbolic names such as DTP-LIST, DTP-SYMBOL, DTP-FIXNUM, etc.

The Lisp machine data types are designed according to these criteria: There should be a wide variety of useful and flexible data types. Some effort should be made to increase the bit-efficiency of data representation, in order to improve performance. The programmer should be able to exercise control over the storage and representation of data, if he wishes. It must always be possible to take an anonymous piece of data and discover its type; this facilitates storage management. There should be as much type-checking and error-checking as feasible in the system.

Symbols are stored as four consecutive words, each of which contains one object. The words are termed the PRINT NAME cell, the VALUE cell, the FUNCTION cell, and the PROPERTY LIST cell. The PRINT NAME cell holds a string object, which is the printed representation of the symbol. The PROPERTY LIST cell, of course, contains the property list, and the VALUE CELL contains the current value of the symbol (it is a shallow-binding system). The FUNCTION cell replaces the task of the EXPR, SUBR, FEXPR, MACRO, etc. properties in Maclisp. When a form such as (FOO ARG1 ARG2) is evaluated, the object in FOO's function cell is applied to the arguments. A symbol object has datatype DTP-SYMBOL, and the pointer is the address of these four words.

Storage of list structure is somewhat more complicated. Normally a "list object" has datatype DTP-LIST, and the pointer is the address of a two word block; the first word contains the CAR, and the second the CDR of the node.

However, note that since a Lisp object is only 29 bits (24 bits of pointer and 5 bits of data-type), there are three remaining bits in each word. Two of these bits are termed the CDR-CODE field, and are used to compress the storage requirement of list structure. The four possible values of the CDR-CODE field are given the symbolic names CDR-NORMAL, CDR-ERROR, CDR-NEXT, and CDR-NIL. CDR-NORMAL indicates the two-word block described above. CDR-NEXT and CDR-NIL are used to represent a list as a vector, taking only half as much storage as usual; only the CARs are stored. The CDR of each location is simply the next location, except for the last, whose CDR is NIL. The primitive functions which create lists (LIST, APPEND, etc.) create these compressed lists. If RPLACD is done on such a list, it is automatically changed back to the conventional two-word representation, in a transparent way.

The idea is that in the first word of a list node the CAR is represented by 29 bits, and the CDR is represented by 2 bits. It is a compressed pointer which can take on only 3 legal values: to the symbol NIL, to the next location after the one it appears in, or indirect through the next location. CDR-ERROR is used for words whose address should not ever be in a list object; in a "full node", the first word is CDR-NORMAL, and the second is CDR-ERROR. It is important to note that the cdr-code portion of a word is used in a different way from the data-type and pointer portion; it is a property of the memory cell itself, not of the cell's contents. A "list object" which is represented in compressed form still has data type DTP-LIST, but the cdr code of the word addressed by its pointer field is CDR-NEXT or CDR-NIL rather than CDR-NORMAL.

Number objects may have any of three datatypes. "FIXNUMs", which are 24-bit signed integers, are represented by objects of datatype DTP-FIX, whose "pointer" parts are actually the value of the number. Thus fixnums, unlike all other objects, do not require any "CONS"ed storage for their representation.

This speeds up arithmetic programs when the numbers they work with are reasonably small. Other types of numbers, such as floating point, BIGNUMs (integers of arbitrarily high precision), complex numbers, and so on, are represented by objects of datatype DTP-EXTENDED-NUMBER which point to a block of storage containing the details of the number. The microcode automatically converts between the different number representations as necessary, without the need for explicit declarations on the programmer's part.

There is also a datatype DTP-PDL-NUMBER, which is almost the same as DTP-EXTENDED-NUMBER. The difference is that pdl numbers can only exist in the pdl buffer (a memory internal to the machine which holds the most recent stack frames), and their blocks of storage are allocated in a special area. Whenever an object is stored into memory, if it is a pdl number its block of storage is copied, and an ordinary extended number is substituted. The idea of this is to prevent intermediate numeric results from using up storage and causing increased need for garbage collection. When the special pdl number area becomes full, all pdl numbers can quickly be found by scanning the pdl buffer. Once they have been copied out into ordinary numbers, the special area is guaranteed empty and can be reclaimed, with no need to garbage collect nor to look at other parts of memory. Note that these are not at all the same as pdl numbers in Maclisp; however, they both exist for the same reason.

The most important other data type is the array. Some problems are best attacked using data structures organized in the list-processing style of Lisp, and some are best attacked using the array-processing style of Fortran. The complete programming system needs both. As mentioned above, Lisp Machine arrays are augmented beyond traditional Lisp arrays in several ways. First of all, we have the ordinary arrays of Lisp objects, with one or more dimensions. Compact storage of positive integers, which may represent characters or other non-numeric entities, is afforded by arrays of 1-bit, 2-bit, 4-bit, 8-bit, or 16-bit elements.

For string-processing, there are string-arrays, which are usually one-dimensional and have 8-bit characters as elements. At the microcode level strings are treated the same as 8-bit arrays, however strings are treated differently by READ, PRINT, EVAL, and many other system and user functions. For example, they print out as a sequence of characters enclosed in quotes. The characters in a character string can be accessed and modified with the same array-referencing functions as one uses for any other type of array. Unlike arrays in other Lisp systems, Lisp machine arrays usually have only a single word of overhead, so the character strings are quite storage-efficient.

There are a number of specialized types of arrays which are used to implement other data types, such as stack groups, internal system tables, and, most importantly, the refresh memory of the TV display as a two-dimensional array of bits.

An important additional feature of Lisp machine arrays is called "array leaders." A leader is a vector of Lisp objects, of user-specified size, which may be tacked on to an array. Leaders are a good place to remember miscellaneous extra information associated with an array. Many data structures consist of a combination of an array and a record (see below); the array contains a number of objects all of the same conceptual type, while the record contains miscellaneous items all of different conceptual types. By storing the record in the leader of the array, the single conceptual data structure is represented by a single actual object. Many data structures in Lisp-machine system programs work this way.

Another thing that leaders are used for is remembering the "current length" of a partially-populated array. By convention, array leader element number 0 is always used for this.

Many programs use data objects structured as "records"; that is, a compound object consisting of a fixed number of named sub-objects. To facilitate the use of records, the Lisp machine system includes a standard set of macros for defining, creating, and accessing record structures. The user can choose whether the actual representation is to be a Lisp list, an array, or an array-leader. Because this is done with macros, which translate record operations into the lower-level operations of basic Lisp, no other part of the system needs to know about records.

Since the reader and printer are written in Lisp and user-modifiable, this record-structure feature could easily be expanded into a full-fledged user-defined data type facility by modifying read and print to support input and output of record types.

Another data type is the "locative pointer." This is an actual pointer to a memory location, used by low-level system programs which need to deal with the guts of data representation. Taking CAR or CDR of a locative gets the contents of the pointed-to location, and RPLACA or RPLACD stores. It is possible to LAMBDA-bind the location. Because of the tagged architecture and highly-organized storage, it is possible to have a locative pointer into the middle of almost anything without causing trouble with the garbage collector.

REPRESENTATION OF PROGRAMS:

In the Lisp Machine there are three representations for programs. Interpreted Lisp code is the slowest, but the easiest for programs to understand and modify. It can be used for functions which are being debugged, for functions which need to be understood by other functions, and for functions which are not worth the bother of compiling. A few functions, notably EVAL, will not work interpreted.

Compiled Lisp ("macrocode") is the main representation for programs. This consists of instructions in a somewhat conventional machine-language, whose unusual features will be described below. Unlike the case in many other Lisp systems, macrocode programs still have full checking for unbound variables, data type errors, wrong number of arguments to a function, and so forth, so it is not necessary to resort to interpreted code just to get extra checking to detect bugs. Often, after typing in a function to the editor, one skips the interpretation step and requests the editor to call the compiler on it, which only takes a few seconds since the compiler is always in the machine and only has to be paged in.

Compiled code on the Lisp Machine is stored inside objects called (for historical reasons) Function Entry Frames (FEFs). For each function compiled, one FEF is created, and an object of type DTP-FEF-POINTER is stored in the function cell of the symbol which is the name of the function. A FEF consists of some header information, a description of the arguments accepted by the function, pointers to external Lisp objects needed by the function (such as constants and special variables), and the macrocode which implements the function.

The third form of program representation is microcode. The system includes a good deal of hand-coded microcode which executes the macrocode instructions, implements the data types and the function-calling mechanism, maintains the paged virtual memory, does storage allocation and garbage collection, and performs similar systemic functions. The primitive operations on the basic data types, that is, CAR and CDR for lists, arithmetic for numbers, reference and store for arrays, etc. are implemented as microcode subroutines. In addition, a number of commonly-used Lisp functions, for instance GET and ASSQ, are hand-coded in microcode for speed.

In addition to this system-supplied microcode, there is a feature called micro compilation. Because of the simplicity and generality of the CONS microprocessor, it is feasible to write a compiler to compile user-written Lisp functions directly into microcode, eliminating the overhead of fetching and interpreting macroinstructions. This can be used to boost performance by microcompiling the most critical routines of a program. Because it is done by a compiler rather than a system programmer, this performance improvement is available to everyone. The amount of speedup to be expected depends on the operations used by the program; simple low-level operations such as data transmission, byte extraction, integer arithmetic, and simple branching can expect to benefit the most. Function calling, and operations which already spend most of their time in microcode, such as ASSQ, will benefit the least. In the best case one can achieve a factor of about 20. In the worst case, maybe no speedup at all.

Since the amount of control memory is limited, only a small number of microcompiled functions can be loaded in at one time. This means that programs have to be characterized by spending most of their time in a small inner kernel of functions in order to benefit from microcompilation; this is probably true of most programs. There will be fairly hairy metering facilities for identifying such critical functions.

We do not yet have a microcompiler, but a prototype of one was written and heavily used as part of the Lisp machine simulator. It compiles for the PDP-10 rather than CONS, but uses similar techniques and a similar interface to the built-in microcode.

In all three forms of program, the flexibility of function calling is augmented with generalized LAMBDA-lists. In order to provide a more general and flexible scheme to replace EXPRs vs. FEXPRs vs. LEXPRs, a syntax borrowed from Muddle and Conniver is used in LAMBDA lists. In the general case, there are an arbitrary number of REQUIRED parameters, followed by an arbitrary number of OPTIONAL parameters, possibly followed by one REST parameter. When a function is APPLIED to its arguments, first of all the required formal parameters are paired off with arguments; if there are fewer arguments than required parameters, an error condition is caused. Then, any remaining arguments are paired off with the optional parameters; if there are more optional parameters than arguments remaining, then the rest of the optional parameters are initialized in a user-specified manner. The REST parameter is bound to a list, possibly NIL, of all arguments remaining after all OPTIONAL parameters are bound. To avoid CONSing, this list is actually stored on the pdl; this means that you have to be careful how you use it, unfortunately. It is also possible to control which arguments are evaluated and which are quoted.

Normally, such a complicated calling sequence would entail an unacceptable amount of overhead. Because this is all implemented by microcode, and because the simple, common cases are special-cased, we can provide these advanced features and still retain the efficiency needed in a practical system.

We will now discuss some of the issues in the design of the macrocode instruction set. Each macroinstruction is 16 bits long; they are stored two per word. The instructions work in a stack-oriented machine. The stack is formatted into frames; each frame contains a bunch of arguments, a bunch of local variable value slots, a push-down stack for intermediate results, and a header which gives the function which owns the frame, links this frame to previous frames, remembers the program counter and flags when this frame is not executing, and may contain "additional information" used for certain esoteric purposes. Originally this was intended to be a spaghetti stack, but the invention of closures and stack-groups (see the control-structure section), combined with the extreme complexity of spaghetti stacks, made us decide to use a simple linear stack. The current frame is always held in the pdl buffer, so accesses to arguments and local variables do not require memory references, and do not have to make checks related to the garbage collector, which improves performance. Usually several other frames will also be in the pdl buffer.

The macro instruction set is bit-compact. The stack organization and Lisp's division of programs into small, separate functions means that address fields can be small. The use of tagged data types, powerful generic operations, and easily-called microcoded functions makes a single 16-bit macro instruction do the work of several instructions on a conventional machine such as a PDP-10.

The primitive operations which are the instructions which the compiler generates are higher-level than the instructions of a conventional machine. They all do data type checks; this provides more run-time error checking than in Maclisp, which increases reliability. But it also eliminates much of the need to make declarations in order to get efficient code. Since a data type check is being made, the "primitive" operations can dynamically decide which specific routine is to be called. This means that they are all "generic", that is, they work for all data types where they make sense.

The operations which are regarded as most important, and hence are easiest for macrocode to do, are data transmission, function calling, conditional testing, and simple operations on primitive types, that is, CAR, CDR, CADR, CDDR, RPLACA, and RPLACD, plus the usual arithmetic operations and comparisons. More complex operations are generally done by "miscellaneous" instructions, which call microcoded subroutines, passing arguments on the temporary-results stack.

There are three main kinds of addressing in macrocode. First, there is implicit addressing of the top of the stack. This is the usual way that operands get from one instruction to the next.

Second, there is the source field (this is sometimes used to store results, but I will call it a source anyway). The source can address any of the following: Up to 64 arguments to the current function. Up to 64 local variables of the current function. The last result, popped off the stack. One of several commonly-used constants (e.g. NIL) stored in a system-wide constants area. A constant stored in the FEF of this function. A value cell or a function cell of a symbol, referenced by means of an invisible pointer in the FEF; this mode is used to reference special variables and to call other functions.

Third, there is the destination field, which specifies what to do with the result of the instruction. The possibilities are: Ignore it, except set the indicators used by conditional branches. Push it on the stack. Pass it as an argument. Return it as the value of this function. Cons up a list.

There are five types of macroinstructions, which will be described. First, there are the data transmission instructions, which take the source and MOVE it to the destination, optionally taking CAR, CDR, CAAR, CADR, CDAR, or CDDR in the process. Because of the powerful operations that can be specified in the destination, these instructions also serve as argument-passing, function-exiting, and list-making instructions.

Next we have the function calling instructions. The simpler of the two is CALL0, call with no arguments. It calls the function indicated by its source, and when that function returns, the result is stored in the destination. The microcode takes care of identifying what type of function is being called, invoking it in the appropriate way, and saving the state of the current function. It traps to the interpreter if the called function is not compiled.

The more complex function call occurs when there are arguments to be passed. The way it works is as follows. First, a CALL instruction is executed. The source operand is the function to be called. The beginnings of a new stack frame are constructed at the end of the current frame, and the function to be called is remembered. The destination of the CALL instruction specifies where the result of the function will be placed, and it is saved for later use when the function returns. Next, instructions are executed to compute the arguments and store them into the destination NEXT-ARGUMENT. This causes them to be added to the new stack frame. When the last argument is computed, it is stored into the destination LAST-ARGUMENT, which stores it in the new stack frame and then activates the call. The function to be called is analyzed, and the arguments are bound to the formal parameters (usually the arguments are already in the correct slots of the new stack frame). Because the computation of the arguments is introduced by a CALL instruction, it is easy to find out where the arguments are and how many there are. The new stack frame becomes current and that function begins execution. When it returns, the saved destination of the CALL instruction is retrieved and the result is stored. Note that by using a destination of NEXT-ARGUMENT or LAST-ARGUMENT function calls may be nested. By using a destination of RETURN the result of one function may become the result of its caller.

The third class of macro instructions consists of a number of common operations on primitive data types. These instructions do not have an explicit destination, in order to save bits, but implicitly push their result (if any) onto the stack. This sometimes necessitates the generation of an extra MOVE instruction to put the result where it was really wanted. These instructions include: Operations to store results from the pdl into the "source". The basic arithmetic and bitwise boolean operations. Comparison operations, including EQ and arithmetic comparison, which set the indicators for use by conditional branches. Instructions which set the "source" operand to NIL or zero. Iteration instructions which change the "source" operand using CDR, CDDR, 1+, or 1- (add or subtract one). Binding instructions which lambda-bind the "source" operand, then optionally set it to NIL or to a value popped off the stack. And, finally, an instruction to push its effective address on the stack, as a locative pointer.

The fourth class of macro instructions are the branches, which serve mainly for compiling COND. Branches contain a self-relative address which is transferred to if a specified condition is satisfied. There are two indicators, which tell if the last result was NIL, and if it was an atom, and the state of these indicators can be branched on; there is also an unconditional branch, of course. For branches more than 256 half-words away, there is a double-length long-branch instruction. An interesting fact is that there are not really any indicators; it turns out to be faster just to save the last result in its entirety, and compare it against NIL or whatever when that is needed by a branch instruction. It only has to be saved from one instruction to the immediately following one.

The fifth class of macro instructions is the "miscellaneous function." This selects one of 512 microcoded functions to be called, with arguments taken from results previously pushed on the stack. A destination is specified to receive the result of the function. In addition to commonly-used functions such as GET, CONS, CDDDDR, REMAINDER, and ASSQ, miscellaneous functions include sub-primitives (discussed above), and instructions which are not as commonly used as the first four classes, including operations such as array-accessing, consing up lists, un-lambda-binding, special funny types of function calling, etc.

The way consing-up of lists works is that one first does a miscellaneous function saying "make a list N long". One then executes N instructions with destination NEXT-LIST to supply the elements of the list. After the Nth such instruction, the list-object magically appears on the top of the stack. This saves having to make a call to the function LIST with a variable number of arguments.

Another type of "instruction set" used with macrocode is the Argument Description List, which is executed by a different microcoded interpreter at the time a function is entered. The ADL contains one entry for each argument which the function expects to be passed, and for each auxiliary variable. It contains all relevant information about the argument: whether it is required, optional, or rest, how to initialize it if it is not provided, whether it is local or special, datatype checking information, and so on. Sometimes the ADL can be dispensed with if the "fast argument option" can be used instead; this helps save time and memory for small, simple functions. The fast-argument option is used when the optional arguments and local variables are all to be initialized to NIL, there are not too many of them, there is no data-type checking, and the usage of special variables is not too complicated. The selection of the fast-argument option, if appropriate, is automatically made by the system, so the user need not be concerned with it. The details can be found in the FORMAT document.

CONTROL STRUCTURES:

Function calling. Function calling is, of course, the basic main control structure in Lisp. As mentioned above, Lisp machine function calling is made fast through the use of microcode and augmented with optional arguments, rest arguments, multiple return values, and optional type-checking of arguments.

CATCH and THROW. CATCH and THROW are a Maclisp control structure which will be mentioned here since they may be new to some people. CATCH is a way of marking a particular point in the stack of recursive function invocations. THROW causes control to be unwound to the matching CATCH, automatically returning through the intervening function calls. They are used mainly for handling errors and unusual conditions. They are also useful for getting out of a hairy piece of code when it has discovered what value it wants to return; this applies particularly to nested loops.

Closures. The LISP machine contains a data-type called "closure" which is used to implement "full funarging". By turning a function into a closure, it becomes possible to pass it as an argument with no worry about naming conflicts, and to return it as a value with exactly the minimum necessary amount of binding environment being retained, solving the classical "funarg problem". Closures are implemented in such a way that when they are not used the highly speed- and storage-efficient shallow binding variable scheme operates at full efficiency, and when they are used things are slowed down only slightly. The way one creates a closure is with a form such as:

```
(CLOSURE '(FOO-PARAM FOO-STATE)
         (FUNCTION FOO-BAR))
```

The function could also be written directly in place as a LAMBDA-expression, instead of referring to the externally defined FOO-BAR. The variables FOO-PARAM and FOO-STATE are those variables which are used free by FOO-BAR and are intended to be "closed". That is, these are the variables whose binding environment is to be fixed to that in effect at the time the closure is created. The explicit declaration of which variables are to be closed allows the implementation to have high efficiency, since it does not need to save the whole variable-binding environment, almost all of which is useless. It also allows the programmer to explicitly choose for each variable whether it is to be dynamically bound (at the point of call) or statically bound (at the point of creation of the closure), a choice which is not conveniently available in other languages. In addition the program is clearer because the intended effect of the closure is made manifest by listing the variables to be affected.

Here is an example, in which the closure feature is used to solve a problem presented in "LAMBDA - The Ultimate Imperative" [LAMBDA]. The problem is to write a function called GENERATE-SQRT-OF-GIVEN-EXTRA-TOLERANCE, which is to take one argument, which is the factor by which the tolerance is to be increased, and return a function which takes square roots with that much more tolerance than usual, whatever "usual" is later defined to be. You are given a function SQRT which makes a free reference to EPSILON, which is the tolerance it demands of the trial solution. The reason this example presents difficulties to various languages is that the variable EPSILON must be bound at the point of call (i.e.

dynamically scoped), while the variable FACTOR must be bound at the point of creation of the function (i.e. lexically scoped). Thus the programmer must have explicit control over how the variables are bound.

```
(DEFUN GENERATE-SQRT-OF-GIVEN-EXTRA-TOLERANCE (FACTOR)
    (CLOSURE '(FACTOR)
             (FUNCTION
               (LAMBDA (X)
                 ((LAMBDA (EPSILON) (SQRT X))
                  (* EPSILON FACTOR))))))
```

The function, when called, rebinds EPSILON to FACTOR times its current value, then calls SQRT. The value of FACTOR used is that in effect when the closure was created, i.e. the argument to GENERATE-SQRT-OF-GIVEN-EXTRA-TOLERANCE.

The way closures are implemented is as follows. For each variable to be closed an "external value cell" is created, which is a CONSed up free-storage cell which contains the variable's value when it is at that level of binding. Because this cell is CONSed up, it can be retained as long as necessary, just like any other data, and unlike cells in a stack. Because it is a cell, if the variable is SETQed the new value is seen by all the closures that should see it. The association between the symbol which is the name of the variable and this value cell is of the shallow-binding type, for efficiency; an invisible pointer (see the storage organization section) in the normal (internal) value cell supplies the connection, eliminating the overhead of searching stack frames or a-lists. If at the time the closure is created an external value cell already exists for a variable, that one is used instead of creating a new one. Thus all closures at the same "level of binding" use the same value cell, which is the desired semantics.

The CLOSURE function returns an object of type DTP-CLOSURE, which contains the function to be called and, for each variable closed over, locative pointers to its internal and external value cells.

When a closure is invoked as a function, the variables mentioned in the closure are bound to invisible pointers to their external value cells; this puts these variables into the proper binding environment. The function contained in the closure is then invoked in the normal way. When the variables happen to be referred to, the invisible pointers are automatically followed to the external value cells. If one of the closed variables is then bound by some other function, the external value cell pointer is saved away on the binding stack, like any saved variable value, and the variable reverts to normal nonclosed status. When the closed function returns, the bindings of the closed variables are restored just like any other variables bound by the function.

Note the economy of mechanism. Almost all of the system is completely unaffected by and unaware of the existence of closures; the invisible pointer mechanism takes care of things. The retainable binding environments are allocated through the standard CONS operation. The switching of variables between normal and "closed" status is done through the standard binding operation. The operations used by a closed function to access the closed variables are the same as those used to access ordinary variables; closures are called in the same way as ordinary functions. Closures work just as well in the interpreter as in the compiler. An important thing to note is the minimality of CONSing in closures. When a closure is created, some CONSing is done; external value cells and the closure-object itself must be created, but there is no extra "overhead". When a closure is called, no CONSing happens.

One thing to note is that in the compiler closed variables have to be declared "special". This is a general feature of the Maclisp and Lisp machine compilers, that by default variables are local, which means that they are lexically bound, only available to the function in which they are bound, and implemented not with atomic symbols, but simply as slots in the stack. Variables that are declared special are implemented with shallow-bound atomic symbols, identical to variables in the interpreter, and have available either dynamic binding or closure binding. They are somewhat less efficient since it takes two memory references to access them and several to bind them.

Stack groups. The stack group is a type of Lisp object useful for implementation of certain advanced control structures such as coroutines, asynchronous processes, and generators. A stack group is similar to a process (or fork or job or task or control-point) in a time-sharing system; it contains such state information as the "regular" and "special" (binding) PDLs and various internal registers. At all times there is one stack group running on the machine.

Control may be passed between stack groups in several ways (not all of which exist yet on our prototype machine). A stack-group may be called like a function; when it wants to return it can do a %STACK-GROUP-RETURN which is different from an ordinary function return in that the state of the stack group remains unchanged; the next time it is called it picks up from where it left off. This is good for generator-like applications; each time %STACK-GROUP-RETURN is done, a value is emitted from the generator, and as a side-effect execution is suspended until the next time the generator is called. %STACK-GROUP-RETURN is analogous to the ADIEU construct in CONNIVER.

Control can simply be passed explicitly from one stack group to another, coroutine-style. Alternatively, there can be a scheduler stack-group which invokes other stack groups when their requested scheduling conditions are satisfied.

Interrupts cause control of the machine to be transferred to an interrupt-handler stack group. Essentially this is a forced stack group call like those calls described above. Similarly, when the microcode detects an error the current stack group is suspended and control is passed to an error-handling stack group. The state of the stack group that got the error is left exactly as it was when the error occurred, undisturbed by any error-handling operations. This facilitates error analysis and recovery.

When the machine is started, an "initial" stack group becomes the current stack group, and is forced to call the first function of Lisp.

Note that the same scheduler-driven stack-group switching mechanism can be used both for user programs which want to do parallel computations, and for system programming purposes such as the handling of network servers and peripheral handlers.

Each stack group has a call-state and a calling-stack-group variable, which are used in maintaining the relations between stack groups. A stack group also has some option flags controlling whether the system tries to keep different stack groups' binding environments distinct by undoing the special variable bindings of the stack group being left and redoing the bindings of the stack group being entered.

Stack groups are created with the function MAKE-STACK-GROUP, which takes one main argument, the "name" of the stack group. This is used only for debugging, and can be any mnemonic symbol. It returns the stack group, i.e., a Lisp object with data type DTP-STACK-GROUP. Optionally the sizes of the pdls may be specified.

The function STACK-GROUP-PRESET is used to initialize the state of a stack group: the first argument is the stack group, the second is a function to be called when the stack group is invoked, and the rest are arguments to that function. Both PDLs are made empty. The stack group is set to the AWAITING-INITIAL-CALL state. When it is activated, the specified function will find that it has been called with the specified arguments. If it should return in the normal way, (i.e. the stack group "returns off the top", the stack group will enter a "used up" state and control will revert to the calling stack group. Normally, the specified function will use %STACK-GROUP-RETURN several times; otherwise it might as well have been called directly rather than in a stack group.

One important difference between stack groups and other means proposed to implement similar features is that the stack group scheme involves no loss of efficiency in normal computation. In fact, the compiler, the interpreter, and even the runtime function-calling mechanism are completely unaware of the existance of stack groups.

STORAGE ORGANIZATION:

Incremental Garbage Collection. The Lisp machine will use a real-time, incremental, compacting garbage collector. Real-time means that CONS (or related functions) never delay Lisp execution for more than a small, bounded amount of time.

This is very important in a machine with a large address space, where a traditional garbage collection could bring everything to a halt for several minutes. The garbage collector is incremental, i.e. garbage collection is interleaved with execution of the user's program; every time you call CONS the garbage collection proceeds for a few steps. Copying can also be triggered by a memory reference which fetches a pointer to data which has not yet been copied. The garbage collector compactifies in order to improve the paging characteristics.

The basic algorithm is described in a paper by Henry Baker [GC]. We have not implemented it yet, but design is proceeding and most of the necessary changes to the microcode have already been made. It is much simpler than previous methods of incremental garbage collection in that only one process is needed; this avoids interlocking and synchronization problems, which are often very difficult to debug.

Areas. Storage in the Lisp machine is divided into "areas." Each area contains related objects, of any type. Since unlike PDP-10 Lisps we do not encode the data type in the address, we are free to use the address to encode the area. Areas are intended to give the user control over the paging behavior of his program, among other things. By putting related data together, locality can be greatly increased. Whenever a new object is created, for instance with CONS, the area to be used can optionally be specified. There is a default Working Storage area which collects those objects which the user has not chosen to control explicitly.

Areas also give the user a handle to control the garbage collector. Some areas can be declared to be "static", which means that they change slowly and the garbage collector should not attempt to reclaim any space in them. This can eliminate a lot of useless copying. All pointers out of a static area can be collected into an "exit vector", eliminating any need for the garbage collector to look at that area. As an important example, an English-language dictionary can be kept inside the Lisp without adversely affecting the speed of garbage collection. A "static" area can be explicitly garbage-collected at infrequent intervals when it is believed that that might be worthwhile.

Each area can potentially have a different storage discipline, a different paging algorithm, and even a different data representation. The microcode will dispatch on an attribute of the area at the appropriate times. The structure of the machine makes the performance cost of these features negligible; information about areas is stored in extra bits in the memory mapping hardware where it can be quickly dispatched on by the microcode. These dispatches usually have to be done anyway to make the garbage collector work, and to implement invisible pointers.

Invisible Pointers. An invisible pointer is similar to an indirect address word on a conventional computer except the indirection is specified in the data instead of in the instruction. A reference to a memory location containing an invisible pointer is automatically altered to use the location pointed to by the invisible pointer. The term "invisible" refers to the fact that the presence of such pointers is not visible to most of the system, since they are handled by the lowest-level memory-referencing operations. The invisible pointer feature does not slow anything down too much, because it is part of the data type checking that is done anyway (this is one of the benefits of a tagged architecture). A number of advanced features of the Lisp machine depend upon invisible pointers for their efficient implementation.

Closures use invisible pointers to connect internal value cells to external value cells. This allows the variable binding scheme to be altered from normal shallow binding to allocated-value-cell shallow binding when closures are being used, without altering the normal operation of the machine when closures are not being used. At the same time the slow-down when closures are used amounts to only 2 microseconds per closed-variable reference, the time needed to detect and follow the invisible pointer.

Invisible pointers are necessary to the operation of the cdr-coded compressed list scheme. If an RPLACD is done to a compressed list, the list can no longer be represented in the compressed form. It is necessary to allocate a full 2-word cons node and use that in its place. But, it is also necessary to preserve the identity (with respect to EQ) of the list. This is done by storing an invisible pointer in the original location of the compressed list, pointing to the uncompressed copy. Then the list is still represented by its original location, preserving EQ-ness, but the CAR and CDR operations follow the invisible pointer to the new location and find the proper car and cdr.

This is a special case of the more general use of invisible pointers for "forwarding" references from an old representation of an object to a new one. For instance, there is a function to increase the size of an array. If it cannot do it in place, it makes a new copy and leaves behind an invisible pointer.

The exit-vector feature uses invisible pointers. One may set up an area to have the property that all references from inside that area to objects in other areas are collected into a single exit-vector. A location which would normally contain such a reference instead contains an invisible pointer to the appropriate slot in the exit vector. Operations on this area all work as before, except for a slight slow-down caused by the invisible pointer following. It is also desirable to have automatic checking to prevent the creation of new outside references; when an attempt is made to store an outside object into this area execution can trap to a routine which creates a new exit vector entry if necessary and stores an invisible pointer instead. The reason for exit vectors is to speed up garbage collection by eliminating the need to swap in all of the pages of the area in order to find and relocate all its references to outside objects.

The macrocode instruction set relies on invisible pointers in order to access the value cells of "special" (non-local) variables and the function cells of functions to be called.

Certain system variables stored in the microcode scratchpad memory are made available to Lisp programs by linking the value cells of appropriately-named Lisp symbols to the scratchpad memory locations with invisible pointers. This makes it possible not only to read and write these variables, but also to lambda-bind them. In a similar fashion, invisible pointers could be used to link two symbols' value cells together, in the fashion of MicroPlanner but with much greater efficiency.

THE EDITOR:

The Lisp machine system includes an advanced real-time display oriented editor, which is written completely in Lisp. The design of this editor drew heavily on our experience with the EMACS editor (and its predecessors) on the PDP-10. The high-speed display and fast response time of the Lisp machine are crucial to the success of the editor.

The TV display is used to show a section of the text buffer currently being edited. When the user types a normal printing character on the keyboard, that character is inserted into his buffer, and the display of the buffer is updated; you see the text as you type it in. When using an editor, most of the user's time is spent in typing in text; therefore, this is made as easy as possible. Editing operations other than the insertion of single characters are invoked by control-keys, i.e. by depressing the CONTROL and/or META shift keys, along with a single character. For example, the command to move the current location for typein in the buffer (the "point") backward is Control-B (B is mnemonic for Backward); the command to move to the next line is Control-N. There are many more advanced commands, which know how to interpret the text as words or as the printed representation of Lisp data structure; Meta-F moves forward over an English word, and Control-Meta-F moves forward over a Lisp expression (an atom or a list).

The real-time display-oriented type of editor is much easier to use than traditional text editors, because you can always see exactly what you are doing. A new user can sit right down and type in text. However, this does not mean that there can be no sophisticated commands and macros. Very powerful operations are provided in the Lisp machine editor. Self-documentation features exist to allow the user to ask what a particular key does before trying it, and to ask what keys contain a given word in their description. Users can write additional commands, in Lisp, and add them to the editor's command tables.

The editor knows how much a line should be indented in a Lisp program in order to reflect the level of syntactic nesting. When typing in Lisp code, one uses the Linefeed key after typing in a line to move to the next line and automatically indent it by the right amount. This serves the additional purpose of instantly pointing out errors in numbers of parentheses.

The editor can be used as a front end to the Lisp top level loop. This provides what can be thought of as very sophisticated rubout processing. When the user is satisfied that the form as typed is correct, he can activate it, allowing Lisp to read in the form and evaluate it. When Lisp prints out the result, it is inserted into the buffer at the right place. Simple commands are available to fetch earlier inputs, for possible editing and reactivation.

In addition to commands from the keyboard, the mouse can be used to point to parts of the buffer, and to give simple editing commands. The use of mice for text editing was originated at SRI, and has been refined and extended at XEROX-PARC.

The character-string representation of each function in a program being worked on is stored in its own editor buffer. One normally modifies functions by editing the character-string form, then typing a single-character command to read it into Lisp, replacing the old function. Compilation can optionally be requested. The advantage of operating on the character form, rather than directly on the list structure, is that comments and the user's chosen formatting of the code are preserved; in addition, the editor is easier to use because it operates on what you see on the display. There are commands to store sets of buffers into files, and to get them back again.

The editor has the capability to edit and display text in multiple fonts, and many other features too numerous to mention here.

CURRENT STATUS (August 1977)

The original prototype CONS machine was designed and built somewhat more than two years ago. It had no memory and no I/O capability, and remained pretty much on the back burner while software was developed with a simulator on the PDP-10 (the simulator executed the Lisp machine macro instruction set, a function now performed by CONS microcode.) Microprogramming got under way a little over a year ago, and in the beginning of 1977 the machine got memory, a disk, and a terminal.

We now have an almost-complete system running on the prototype machine. The major remaining "holes" are the lack of a garbage collector and the presence of only the most primitive error handling. Also, floating-point and big-integer numbers and microcompilation have been put off until the next machine. The system includes almost all the functions of Maclisp, and quite a few new ones.

The machine is able to page off of its disk, accept input from the keyboard and the mouse, display on the TV, and do I/O to files on the PDP-10. The display editor is completely working, and the compiler runs on the machine, so the system is quite usable for typing in, editing, compiling, and debugging Lisp functions.

As a demonstration of the system, and a test of its capabilities, two large programs have been brought over from the PDP-10. William Woods's LUNAR English-language data-base query system was converted from InterLisp to Maclisp, thence to Lisp machine Lisp. On the Lisp machine it runs approximately 3 times as fast as in Maclisp on the KA-10, which in turn is 2 to 4 times as fast as in InterLisp. Note that the Lisp machine time is elapsed real time, while the PDP-10 times are virtual run times as given by the operating system and do not include the delays due to timesharing.

Most of the Macsyma symbolic algebraic system has been converted to the Lisp machine; nearly all the source files were simply compiled without any modifications. Most of Macsyma works except for some things that require bignums. The preliminary speed is the same as on the KA-10, but a number of things have not been optimally converted. (This speed measurement is, again, elapsed time on the Lisp machine version versus reported run time on the KA-10 time sharing system. Thus, paging and scheduling overhead in the KA-10 case are not counted in this measurement.)

LUNAR (including the dictionary) and Macsyma can reside together in the Lisp machine with plenty of room left over; either program alone will not entirely fit in a PDP-10 address space.

The CONS machine is currently being redesigned, and a new machine will be built soon, replacing our present prototype. The new machine will have larger sizes for certain internal memories, will incorporate newer technology, will have greatly improved packaging, and will be faster. It will fit entirely in one cabinet and will be designed for ease of construction and servicing. In late 1977 and early 1978 we plan to build seven additional machines and install them at the MIT AI Lab. During the fall of 1977 we plan to finish the software, bringing it to a point where users can be put on the system. User experience with the Lisp machine during 1978 should result in improvement and cleaning up of the software and documentation, and should give us a good idea of the real performance to be expected from the machine. At that time we will be able to start thinking about ways to make Lisp machines available to the outside world.

REFERENCES:

CONS: Steele, Guy L. "Cons", not yet published. This is a revision of
Working paper 80, CONS by Tom Knight

GC: Baker, Henry, "List Processing in Real Time on a Serial Computer",
Working Paper 139

LAMBDA: Steele, Guy L. "LAMBDA - The Ultimate Imperative", Artificial Intelligence
Memo 353

MOUSE: See extensive publications by Englebart and group at SRI.

```
/*----------------------------------------------------------------------
  key.c :         Key Management Engine for BSD

  Copyright 1995 by Bao Phan,  Randall Atkinson, & Dan McDonald,
  All Rights Reserved.  All Rights have been assigned to the US
  Naval Research Laboratory (NRL).  The NRL Copyright Notice and
  License governs distribution and use of this software.

  Patents are pending on this technology.  NRL grants a license
  to use this technology at no cost under the terms below with
  the additional requirement that software, hardware, and
  documentation relating to use of this technology must include
  the note that:
        This product includes technology developed at and
    licensed from the Information Technology Division,
    US Naval Research Laboratory.

--------------------------------------------------------------------*/
/*---------------------------------------------------------------------
#   @(#)COPYRIGHT   1.1a (NRL) 17 August 1995

COPYRIGHT NOTICE

All of the documentation and software included in this software
distribution from the US Naval Research Laboratory (NRL) are
copyrighted by their respective developers.

This software and documentation were developed at NRL by various
people.  Those developers have each copyrighted the portions that they
developed at NRL and have assigned All Rights for those portions to
NRL.  Outside the USA, NRL also has copyright on the software
developed at NRL. The affected files all contain specific copyright
notices and those notices must be retained in any derived work.

NRL LICENSE

NRL grants permission for redistribution and use in source and binary
forms, with or without modification, of the software and documentation
created at NRL provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
3. All advertising materials mentioning features or use of this software
   must display the following acknowledgement:

    This product includes software developed at the Information
    Technology Division, US Naval Research Laboratory.

4. Neither the name of the NRL nor the names of its contributors
   may be used to endorse or promote products derived from this software
   without specific prior written permission.

THE SOFTWARE PROVIDED BY NRL IS PROVIDED BY NRL AND CONTRIBUTORS ``AS
IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL NRL OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

```

```
The views and conclusions contained in the software and documentation
are those of the authors and should not be interpreted as representing
official policies, either expressed or implied, of the US Naval
Research Laboratory (NRL).

----------------------------------------------------------------------*/

#include <sys/types.h>
#include <sys/param.h>
#include <sys/proc.h>
#include <sys/mbuf.h>
#include <sys/socket.h>
#include <sys/socketvar.h>
#include <sys/time.h>
#include <sys/kernel.h>
#include <net/raw cb.h>
#include <net/if.h>
#include <net/if types.h>
#include <net/if dl.h>
#include <net/route.h>
#include <netinet/in.h>
#include <netinet/in var.h>
#include <netinet/if_ether.h>

#include <netinet6/in6.h>
#include <netinet6/in6 var.h>
#include <netinet6/ipsec.h>
#include <netinet6/key.h>
#include <netinet6/in6_debug.h>

#define MAXHASHKEYLEN (2 * sizeof(int) + 2 * sizeof(struct
sockaddr_in6))

/*
 *  Not clear whether these values should be
 *  tweakable at kernel config time.
 */
#define KEYTBLSIZE 61
#define KEYALLOCTBLSIZE 61
#define SO2SPITBLSIZE 61

/*
 *  These values should be tweakable...
 *  perhaps by using sysctl
 */

#define MAXLARVALTIME 240;    /* Lifetime of a larval key table entry */
#define MAXKEYACQUIRE 1;      /* Max number of key acquire messages sent
*/
                              /*   per destination address
                              */
#define MAXACQUIRETIME 15;    /* Lifetime of acquire message */

/*
 *  Key engine tables and global variables
 */

struct key tblnode keytable[KEYTBLSIZE];
struct key allocnode keyalloctbl[KEYALLOCTBLSIZE];
struct key_so2spinode so2spitbl[SO2SPITBLSIZE];

struct keyso cb keyso cb;
struct key tblnode nullkeynode  = { 0, 0, 0, 0, 0 };
struct key registry *keyregtable;
struct key acquirelist *key acquirelist;
u long maxlarvallifetime = MAXLARVALTIME;
int maxkeyacquire = MAXKEYACQUIRE;
```

```
u_long maxacquiretime = MAXACQUIRETIME;

extern void dump_secassoc();


/*----------------------------------------------------------------------
 * (temporary) Dump a data buffer
 ---------------------------------------------------------------------*
 /

void
dump_buf(buf, len)
     char *buf;
     int len;
{
  int i;

  printf("buf=0x%x len=%d:\n", buf, len);
  for (i = 0; i < len; i++) {
    printf("0x%x ", (u_int8)*(buf+i));
  }
  printf("\n");
}


/*----------------------------------------------------------------------
 * (temporary) Dump a key tblnode structrue
 ---------------------------------------------------------------------*
 /

void
dump_keytblnode(ktblnode)
     struct key_tblnode *ktblnode;
{
  if (!ktblnode) {
    printf("NULL key table node pointer!\n");
    return;
  }
  printf("solist=0x%x ", ktblnode->solist);
  printf("secassoc=0x%x ", ktblnode->secassoc);
  printf("next=0x%x\n", ktblnode->next);
}


/*----------------------------------------------------------------------
 * key secassoc2msghdr():
 *      Copy info from a security association into a key message buffer.
 *      Assume message buffer is sufficiently large to hold all security
 *      association information including src, dst, from, key and iv.
 ---------------------------------------------------------------------*
 /
int
key_secassoc2msghdr(secassoc, km, keyinfo)
     struct ipsec assoc *secassoc;
     struct key msghdr *km;
     struct key_msgdata *keyinfo;
{
  char *cp;
  DPRINTF(IDL_GROSS_EVENT, ("Entering key_secassoc2msghdr\n"));

  if ((km == 0) || (keyinfo == 0) || (secassoc == 0))
    return(-1);

  km->type = secassoc->type;
  km->state = secassoc->state;
  km->label = secassoc->label;
  km->spi = secassoc->spi;
```

```
  km->keylen = secassoc->keylen;
  km->ivlen = secassoc->ivlen;
  km->algorithm = secassoc->algorithm;
  km->lifetype = secassoc->lifetype;
  km->lifetime1 = secassoc->lifetime1;
  km->lifetime2 = secassoc->lifetime2;

  /*
   *  Stuff src/dst/from/key/iv in buffer after
   *  the message header.
   */
  cp = (char *)(km + 1);

#define ROUNDUP(a) \
  ((a) > 0 ? (1 + (((a) - 1) | (sizeof(long) - 1))) : sizeof(long))
#define ADVANCE(x, n) \
    { x += ROUNDUP(n); }

  DPRINTF(IDL_FINISHED, ("sa2msghdr: 1\n"));
  keyinfo->src = (struct sockaddr *)cp;
  if (secassoc->src.sin6_len) {
    bcopy((char *)&(secassoc->src), cp, secassoc->src.sin6_len);
    ADVANCE(cp, secassoc->src.sin6_len);
  } else {
    bzero(cp, sizeof(struct sockaddr_in6));
    ADVANCE(cp, sizeof(struct sockaddr_in6));
  }
  DPRINTF(IDL_FINISHED, ("sa2msghdr: 2\n"));

  keyinfo->dst = (struct sockaddr *)&(secassoc->dst);
  if (secassoc->dst.sin6_len) {
    bcopy((char *)&(secassoc->dst), cp, secassoc->dst.sin6_len);
    ADVANCE(cp, secassoc->dst.sin6_len);
  } else {
    bzero(cp, sizeof(struct sockaddr_in6));
    ADVANCE(cp, sizeof(struct sockaddr_in6));
  }
  DPRINTF(IDL_FINISHED, ("sa2msghdr: 3\n"));

  keyinfo->from = (struct sockaddr *)cp;
  if (secassoc->from.sin6_len) {
    bcopy((char *)&(secassoc->from), cp, secassoc->from.sin6_len);
    ADVANCE(cp, secassoc->from.sin6_len);
  } else {
    bzero(cp, sizeof(struct sockaddr_in6));
    ADVANCE(cp, sizeof(struct sockaddr_in6));
  }
  DPRINTF(IDL_FINISHED, ("sa2msghdr: 4\n"));

  keyinfo->key = cp;
  keyinfo->keylen = secassoc->keylen;
  if (secassoc->keylen) {
    bcopy((char *)(secassoc->key), cp, secassoc->keylen);
    ADVANCE(cp, secassoc->keylen);
  }

  DPRINTF(IDL_FINISHED, ("sa2msghdr: 5\n"));
  keyinfo->iv = cp;
  keyinfo->ivlen = secassoc->ivlen;
  if (secassoc->ivlen) {
    bcopy((char *)(secassoc->iv), cp, secassoc->ivlen);
    ADVANCE(cp, secassoc->ivlen);
  }

  DDO(IDL_FINISHED,printf("msgbuf(len=%d):\n",(char *)cp - (char *)km));
  DDO(IDL_FINISHED,dump_buf((char *)km, (char *)cp - (char *)km));
  DPRINTF(IDL_FINISHED, ("sa2msghdr: 6\n"));
```

```
  return(0);
}


/*-----------------------------------------------------------------
 * key msghdr2secassoc():
 *      Copy info from a key message buffer into an ipsec_assoc
 *      structure
 -----------------------------------------------------------------*
 /
int
key_msghdr2secassoc(secassoc, km, keyinfo)
     struct ipsec assoc *secassoc;
     struct key msghdr *km;
     struct key_msgdata *keyinfo;
{
  DPRINTF(IDL_GROSS_EVENT, ("Entering key_msghdr2secassoc\n"));

  if ((km == 0) || (keyinfo == 0) || (secassoc == 0))
    return(-1);

  secassoc->len = sizeof(*secassoc);
  secassoc->type = km->type;
  secassoc->state = km->state;
  secassoc->label = km->label;
  secassoc->spi = km->spi;
  secassoc->keylen = km->keylen;
  secassoc->ivlen = km->ivlen;
  secassoc->algorithm = km->algorithm;
  secassoc->lifetype = km->lifetype;
  secassoc->lifetime1 = km->lifetime1;
  secassoc->lifetime2 = km->lifetime2;

  if (keyinfo->src)
    bcopy((char *)(keyinfo->src), (char *)&(secassoc->src),
      keyinfo->src->sa_len);

  if (keyinfo->dst)
    bcopy((char *)(keyinfo->dst), (char *)&(secassoc->dst),
      keyinfo->dst->sa_len);

  if (keyinfo->from)
    bcopy((char *)(keyinfo->from), (char *)&(secassoc->from),
      keyinfo->from->sa_len);

  /*
   *  Make copies of key and iv
   */
  if (secassoc->ivlen) {
    K Malloc(secassoc->iv, caddr_t, secassoc->ivlen);
    if (secassoc->iv == 0) {
      DPRINTF(IDL_CRITICAL,("msghdr2secassoc: can't allocate mem for
      iv\n"));
      return(-1);
    }
    bcopy((char *)keyinfo->iv, (char *)secassoc->iv, secassoc->ivlen);
  } else
    secassoc->iv = NULL;

  if (secassoc->keylen) {
    K Malloc(secassoc->key, caddr_t, secassoc->keylen);
    if (secassoc->key == 0) {
      DPRINTF(IDL_CRITICAL,("msghdr2secassoc: can't allocate mem for
      key\n"));
      if (secassoc->iv)
    KFree(secassoc->iv);
      return(-1);
```

```
    }
    bcopy((char *)keyinfo->key, (char *)secassoc->key,
    secassoc->keylen);
  } else
    secassoc->key = NULL;
  return(0);
}


/*---------------------------------------------------------------------
 * addrpart_equal():
 *      Determine if the address portion of two sockaddrs are equal.
 *      Currently handles only AF_INET and AF_INET6 address families.
 ---------------------------------------------------------------------*
 /
int
addrpart_equal(sa1, sa2)
     struct sockaddr *sa1;
     struct sockaddr *sa2;
{

  if ((sa1->sa_family == sa2->sa_family))
    switch(sa1->sa_family) {
    case AF_INET:
      if (((struct sockaddr_in *)sa1)->sin_addr.s_addr ==
      ((struct sockaddr_in *)sa2)->sin_addr.s_addr)
    return(1);
      break;
    case AF_INET6:
      if (IN6_ADDR_EQUAL(((struct sockaddr_in6 *)sa1)->sin6_addr,
             ((struct sockaddr_in6 *)sa2)->sin6_addr))
    return(1);
      break;
    }
  return(0);
}


/*---------------------------------------------------------------------
 * my_addr():
 *      Determine if an address belongs to one of my configured
 interfaces.
 *      Currently handles only AF_INET and AF_INET6 addresses.
 ---------------------------------------------------------------------*
 /
int
my_addr(sa)
     struct sockaddr *sa;
{
  extern struct in6_ifaddr *in6_ifaddr;
  extern struct in_ifaddr *in_ifaddr;
  struct in6_ifaddr *i6a = 0;
  struct in_ifaddr *ia = 0;

  switch(sa->sa_family) {
  case AF_INET6:
    for (i6a = in6_ifaddr; i6a; i6a = i6a->i6a_next) {
      if (IN6_ADDR_EQUAL(((struct sockaddr_in6 *)sa)->sin6_addr,
             i6a->i6a_addr.sin6_addr))
    return(1);
    }
    break;
  case AF_INET:
    for (ia = in_ifaddr; ia; ia = ia->ia_next) {
      if (((struct sockaddr_in *)sa)->sin_addr.s_addr ==
       ia->ia_addr.sin_addr.s_addr)
    return(1);
```

```
    }
    break;
  }
  return(0);
}


/*----------------------------------------------------------------------
 * key inittables():
 *      Allocate space and initialize key engine tables
 ----------------------------------------------------------------------*
 /
void
key_inittables()
{
  struct key_tblnode *keynode;
  int i;

  K Malloc(keyregtable, struct key_registry *, sizeof(struct
  key registry));
  if (keyregtable == 0)
    panic("key inittables");
  bzero((char *)keyregtable, sizeof(struct key registry));
  K_Malloc(key acquirelist, struct key acquirelist *,
       sizeof(struct key_acquirelist));
  if (key acquirelist == 0)
    panic("key inittables");
  bzero((char *)key acquirelist, sizeof(struct key_acquirelist));
  for (i = 0; i < KEYTBLSIZE; i++)
    bzero((char *)&keytable[i], sizeof(struct key_tblnode));
  for (i = 0; i < KEYALLOCTBLSIZE; i++)
    bzero((char *)&keyalloctbl[i], sizeof(struct key_allocnode));
  for (i = 0; i < SO2SPITBLSIZE; i++)
    bzero((char *)&so2spitbl[i], sizeof(struct key_so2spinode));
}


/*----------------------------------------------------------------------
 * key gethashval():
 *      Determine keytable hash value.
 ----------------------------------------------------------------------*
 /
int
key_gethashval(buf, len, tblsize)
     char *buf;
     int len;
     int tblsize;
{
  int i, j = 0;

  /*
   * Todo: Use word size xor and check for alignment
   *       and zero pad if necessary.  Need to also pick
   *       a good hash function and table size.
   */
  if (len <= 0) {
    DPRINTF(IDL_CRITICAL,("key_gethashval got bogus len!\n"));
    return(-1);
  }
  for(i = 0; i < len; i++) {
    j ^=  (u_int8)(*(buf + i));
  }
  return (j % tblsize);
}


/*----------------------------------------------------------------------
```

```
 * key_createkey():
 *      Create hash key for hash function
 *      key is: type+src+dst if keytype = 1
 *              type+src+dst+spi if keytype = 0
 *      Uses only the address portion of the src and dst sockaddrs to
 *      form key.  Currently handles only AF_INET and AF_INET6 sockaddrs
 ----------------------------------------------------------------------*
 /
int
key_createkey(buf, type, src, dst, spi, keytype)
     char *buf;
     u_int type;
     struct sockaddr *src;
     struct sockaddr *dst;
     u_int32 spi;
     u_int keytype;
{
  char *cp, *p;

  DPRINTF(IDL_FINISHED,("Entering key_createkey\n"));

  if (!buf || !src || !dst)
    return(-1);

  cp = buf;
  bcopy((char *)&type, cp, sizeof(type));
  cp += sizeof(type);

  /*
   * Assume only IPv4 and IPv6 addresses.
   */
#define ADDRPART(a) \
    ((a)->sa_family == AF_INET6) ? \
    (char *)&(((struct sockaddr_in6 *)(a))->sin6_addr) : \
    (char *)&(((struct sockaddr_in *)(a))->sin_addr)

#define ADDRSIZE(a) \
    ((a)->sa_family == AF_INET6) ? sizeof(struct in_addr6) : \
    sizeof(struct in_addr)

  DPRINTF(IDL_GROSS_EVENT,("src addr:\n"));
  DDO(IDL_GROSS_EVENT,dump_smart_sockaddr(src));
  DPRINTF(IDL_GROSS_EVENT,("dst addr:\n"));
  DDO(IDL_GROSS_EVENT,dump_smart_sockaddr(dst));

  p = ADDRPART(src);
  bcopy(p, cp, ADDRSIZE(src));
  cp += ADDRSIZE(src);

  p = ADDRPART(dst);
  bcopy(p, cp, ADDRSIZE(dst));
  cp += ADDRSIZE(dst);

#undef ADDRPART
#undef ADDRSIZE

  if (keytype == 0) {
    bcopy((char *)&spi, cp, sizeof(spi));
    cp += sizeof(spi);
  }

  DPRINTF(IDL_FINISHED,("hash key:\n"));
  DDO(IDL_FINISHED, dump_buf(buf, cp - buf));
  return(cp - buf);
}
```

```
/*-------------------------------------------------------------------
 * key sosearch():
 *      Search the so2spi table for the security association allocated
 to
 *      the socket.  Returns pointer to a struct key_so2spinode which
 can
 *      be used to locate the security association entry in the
 keytable.
 -------------------------------------------------------------------*
 /
struct key so2spinode *
key_sosearch(type, src, dst, so)
     u int type;
     struct sockaddr *src;
     struct sockaddr *dst;
     struct socket *so;
{
  struct key_so2spinode *np = 0;

  if (!(src && dst)) {
    DPRINTF(IDL CRITICAL,("key_sosearch: got null src or dst
    pointer!\n"));
    return(NULL);
  }

  for (np = so2spitbl[((u_int32)so) % SO2SPITBLSIZE].next; np; np = np->
  next) {
    if ((so == np->socket) && (type == np->keynode->secassoc->type)
    && addrpart equal(src,
               (struct sockaddr *)&(np->keynode->secassoc->src))
    && addrpart equal(dst,
               (struct sockaddr *)&(np->keynode->secassoc->dst)))
      return(np);
  }
  return(NULL);
}


/*-------------------------------------------------------------------
 * key sodelete():
 *      Delete entries from the so2spi table.
 *        flag = 1  purge all entries
 *        flag = 0  delete entries with socket pointer matching socket
 -------------------------------------------------------------------*
 /
void
key_sodelete(socket, flag)
     struct socket *socket;
     int flag;
{
  struct key so2spinode *prevnp, *np;
  int s = splnet();

  DPRINTF(IDL MAJOR_EVENT,("Entering keysodelete w/so=0x%x flag=%d\n",
  socket,flag));
  if (flag) {
    int i;

    for (i = 0; i < SO2SPITBLSIZE; i++)
      for(np = so2spitbl[i].next; np; np = np->next) {
    KFree(np);
      }
    splx(s);
    return;
  }

  prevnp = &so2spitbl[((u_int32)socket) % SO2SPITBLSIZE];
```

```
  for(np = prevnp->next; np; np = np->next) {
    if (np->socket == socket) {
      struct socketlist *socklp, *prevsocklp;

      (np->keynode->alloc_count)--;

      /*
       * If this socket maps to a unique secassoc,
       * we go ahead and delete the secassoc, since it
       * can no longer be allocated or used by any other
       * socket.
       */
      if (np->keynode->secassoc->state & K UNIQUE) {
    if (key delete(np->keynode->secassoc) != 0)
      panic("key_sodelete");
    np = prevnp;
    continue;
      }

      /*
       * We traverse the socketlist and remove the entry
       * for this socket
       */
      DPRINTF(IDL FINISHED,("keysodelete: deleting from socklist..."));
      prevsocklp = np->keynode->solist;
      for (socklp = prevsocklp->next; socklp; socklp = socklp->next) {
    if (socklp->socket == socket) {
      prevsocklp->next = socklp->next;
      KFree(socklp);
      break;
    }
    prevsocklp = socklp;
      }
      DPRINTF(IDL FINISHED,("done\n"));
      prevnp->next = np->next;
      KFree(np);
      np = prevnp;
    }
    prevnp = np;
  }
  splx(s);
}


/*----------------------------------------------------------------------
 * key deleteacquire():
 *      Delete an entry from the key acquirelist
 ---------------------------------------------------------------------*
 /
void
key_deleteacquire(type, target)
     u int type;
     struct sockaddr *target;
{
  struct key_acquirelist *ap, *prev;

  prev = key acquirelist;
  for(ap = key acquirelist->next; ap; ap = ap->next) {
    if (addrpart equal(target, (struct sockaddr *)&(ap->target)) &&
    (type == ap->type)) {
      DPRINTF(IDL MAJOR EVENT,("Deleting entry from acquire list!\n"));
      prev->next = ap->next;
      KFree(ap);
      ap = prev;
    }
    prev = ap;
  }
```

```
}


/*-------------------------------------------------------------------
 * key search():
 *      Search the key table for an entry with same type, src addr, dest
 *      addr, and spi.  Returns a pointer to struct key_tblnode if found
 *      else returns null.
 -------------------------------------------------------------------*
 /
struct key tblnode *
key_search(type, src, dst, spi, indx, prevkeynode)
     u int type;
     struct sockaddr *src;
     struct sockaddr *dst;
     u int32 spi;
     int indx;
     struct key_tblnode **prevkeynode;
{
  struct key_tblnode *keynode, *prevnode;

  if (indx > KEYTBLSIZE || indx < 0)
    return (NULL);
  if (!(&keytable[indx]))
    return (NULL);

#define sec type keynode->secassoc->type
#define sec spi keynode->secassoc->spi
#define sec src keynode->secassoc->src
#define sec_dst keynode->secassoc->dst

  prevnode = &keytable[indx];
  for (keynode = keytable[indx].next; keynode; keynode = keynode->next)
  {
    if ((type == sec type) && (spi == sec spi) &&
    addrpart equal(src, (struct sockaddr *)&(sec src))
    && addrpart_equal(dst, (struct sockaddr *)&(sec_dst)))
      break;
    prevnode = keynode;
  }
  *prevkeynode = prevnode;
  return(keynode);
}


/*-------------------------------------------------------------------
 * key addnode():
 *      Insert a key_tblnode entry into the key table.  Returns a
 pointer
 *      to the newly created key tblnode.
 -------------------------------------------------------------------*
 /
struct key tblnode *
key_addnode(indx, secassoc)
     int indx;
     struct ipsec_assoc *secassoc;
{
  struct key_tblnode *keynode;

  DPRINTF(IDL GROSS EVENT,("Entering key addnode w/indx=%d
  secassoc=0x%x\n",indx, (u_int32)secassoc));

  if (!(&keytable[indx]))
    return(NULL);
  if (!secassoc) {
    panic("key_addnode: Someone passed in a null secassoc!\n");
  }
```

```

  K Malloc(keynode, struct key_tblnode *, sizeof(struct key_tblnode));
  if (keynode == 0)
    return(NULL);
  bzero((char *)keynode, sizeof(struct key_tblnode));

  K Malloc(keynode->solist, struct socketlist *, sizeof(struct
  socketlist));
  if (keynode->solist == 0) {
    KFree(keynode);
    return(NULL);
  }
  bzero((char *)(keynode->solist), sizeof(struct socketlist));

  keynode->secassoc = secassoc;
  keynode->solist->next = NULL;
  keynode->next = keytable[indx].next;
  keytable[indx].next = keynode;
  return(keynode);
}


/*----------------------------------------------------------------------
 * key add():
 *      Add a new security association to the key table.  Caller is
 *      responsible for allocating memory for the struct ipsec_assoc as

 *      well as the buffer space for the key and iv.  Assumes the
 security
 *      association passed in is well-formed.
 ----------------------------------------------------------------------*
 /
int
key_add(secassoc)
     struct ipsec_assoc *secassoc;
{
  char buf[MAXHASHKEYLEN];
  int len, indx;
  int inbound = 0;
  int outbound = 0;
  struct key tblnode *keynode, *prevkeynode;
  struct key_allocnode *np;
  int s;

  DPRINTF(IDL_GROSS_EVENT, ("Entering key_add w/secassoc=0x%x\n",
  secassoc));

  if (!secassoc) {
    panic("key_add: who the hell is passing me a null pointer");
  }

  /*
   *  For storage purposes, the two esp modes are
   *  treated the same.
   */
  if (secassoc->type == SS ENCRYPTION NETWORK)
    secassoc->type = SS_ENCRYPTION_TRANSPORT;

  /*
   * Should we allow a null key to be inserted into the table ?
   * or can we use null key to indicate some policy action...
   */

  /*
   *  For esp using des-cbc or tripple-des we call
   * des_set_odd_parity.
   */
```

```
  if (secassoc->key && (secassoc->type == SS_ENCRYPTION_TRANSPORT) &&
      ((secassoc->algorithm == IPSEC_ALGTYPE_ESP_DES_CBC) ||
       (secassoc->algorithm == IPSEC_ALGTYPE_ESP_3DES)))
    des_set_odd_parity(secassoc->key);

  /*
   *  Check if secassoc with same spi exists before adding
   */
  bzero((char *)&buf, sizeof(buf));
  len = key_createkey((char *)&buf, secassoc->type,
               (struct sockaddr *)&(secassoc->src),
               (struct sockaddr *)&(secassoc->dst),
               secassoc->spi, 0);
  indx = key_gethashval((char *)&buf, len, KEYTBLSIZE);
  DPRINTF(IDL_GROSS_EVENT,("keyadd: keytbl hash position=%d\n", indx));
  keynode = key_search(secassoc->type, (struct sockaddr *)&(secassoc->
  src),
                (struct sockaddr *)&(secassoc->dst),
                secassoc->spi, indx, &prevkeynode);
  if (keynode) {
    DPRINTF(IDL_MAJOR_EVENT,("keyadd: secassoc already exists!\n"));
    return(-2);
  }

  inbound = my_addr((struct sockaddr *)&(secassoc->dst));
  outbound = my_addr((struct sockaddr *)&(secassoc->src));
  DPRINTF(IDL_FINISHED,("inbound=%d outbound=%d\n", inbound, outbound));

  /*
   * We allocate mem for an allocation entry if needed.
   * This is done here instead of in the allocaton code
   * segment so that we can easily recover/cleanup from a
   * memory allocation error.
   */
  if (outbound || (!inbound && !outbound)) {
    K_Malloc(np, struct key_allocnode *, sizeof(struct key_allocnode));
    if (np == 0) {
      DPRINTF(IDL_CRITICAL,("keyadd: can't allocate allocnode!\n"));
      return(-1);
    }
  }

  s = splnet();

  if ((keynode = key_addnode(indx, secassoc)) == NULL) {
    DPRINTF(IDL_CRITICAL,("keyadd: key_addnode failed!\n"));
    if (np)
      KFree(np);
    splx(s);
    return(-1);
  }
  DPRINTF(IDL_EVENT,("Added new keynode:\n"));
  DDO(IDL_GROSS_EVENT, dump_keytblnode(keynode));
  DDO(IDL_GROSS_EVENT, dump_secassoc(keynode->secassoc));

  /*
   *  We add an entry to the allocation table for
   *  this secassoc if the interfaces are up and
   *  the secassoc is outbound.  In the case
   *  where the interfaces are not up, we go ahead
   *  and do it anyways.  This wastes an allocation
   *  entry if the secassoc later turned out to be
   *  inbound when the interfaces are ifconfig up.
   */
  if (outbound || (!inbound && !outbound)) {
    len = key_createkey((char *)&buf, secassoc->type,
            (struct sockaddr *)&(secassoc->src),
```

```
            (struct sockaddr *)&(secassoc->dst),
            0, 1);
    indx = key gethashval((char *)&buf, len, KEYALLOCTBLSIZE);
    DPRINTF(IDL_GROSS_EVENT,("keyadd: keyalloc hash position=%d\n",
    indx));
    np->keynode = keynode;
    np->next = keyalloctbl[indx].next;
    keyalloctbl[indx].next = np;
  }
  if (inbound)
    secassoc->state |= K_INBOUND;
  if (outbound)
    secassoc->state |= K_OUTBOUND;

  key deleteacquire(secassoc->type, (struct sockaddr
  *)&(secassoc->dst));

  splx(s);
  return 0;
}


/*----------------------------------------------------------------------
 * key get():
 *      Get a security association from the key table.
 ----------------------------------------------------------------------*
 /
int
key_get(type, src, dst, spi, secassoc)
     u int type;
     struct sockaddr *src;
     struct sockaddr *dst;
     u int32 spi;
     struct ipsec_assoc **secassoc;
{
  char buf[MAXHASHKEYLEN];
  struct key tblnode *keynode, *prevkeynode;
  int len, indx;

  /*
   *  For storage purposes, the two esp modes are
   *  treated the same.
   */
  if (type == SS ENCRYPTION NETWORK)
    type = SS_ENCRYPTION_TRANSPORT;

  bzero(&buf, sizeof(buf));
  *secassoc = NULL;
  len = key createkey((char *)&buf, type, src, dst, spi, 0);
  indx = key gethashval((char *)&buf, len, KEYTBLSIZE);
  DPRINTF(IDL GROSS EVENT,("keyget: indx=%d\n",indx));
  keynode = key search(type, src, dst, spi, indx, &prevkeynode);
  if (keynode) {
    DPRINTF(IDL_EVENT,("keyget: found it! keynode=0x%x",keynode));
    *secassoc = keynode->secassoc;
    return(0);
  } else
    return(-1);  /* Not found */
}


/*----------------------------------------------------------------------
 * key dump():
 *      Dump all valid entries in the keytable to a pf key socket.  Each
 *      security associaiton is sent one at a time in a pf_key message.
 A
 *      message with seqno = 0 signifies the end of the dump
```

```
 transaction.
 ----------------------------------------------------------------------*
 /
int
key_dump(so)
     struct socket *so;
{
  int len, i;
  int seq = 1;
  struct mbuf *m;
  struct key msgdata keyinfo;
  struct key msghdr *km;
  struct key tblnode *keynode;
  extern struct sockaddr key src;
  extern struct sockaddr key_dst;

  /*
   * Routine to dump the key table to a routing socket
   * Use for debugging only!
   */

  DPRINTF(IDL_GROSS_EVENT,("Entering key_dump()"));
  /*
   * We need to speed this up later.  Fortunately, key dump
   * messages are not sent often.
   */
  for (i = 0; i < KEYTBLSIZE; i++) {
    for (keynode = keytable[i].next; keynode; keynode = keynode->next) {

      /*
       * We exclude dead/larval/zombie security associations for now
       * but it may be useful to also send these up for debugging
       purposes
       */
      if (keynode->secassoc->state & (K_DEAD | K_LARVAL | K_ZOMBIE))
    continue;

      len = (sizeof(struct key msghdr) +
         ROUNDUP(keynode->secassoc->src.sin6 len) +
         ROUNDUP(keynode->secassoc->dst.sin6 len) +
         ROUNDUP(keynode->secassoc->from.sin6_len) +
         ROUNDUP(keynode->secassoc->keylen) +
         ROUNDUP(keynode->secassoc->ivlen));
      K Malloc(km, struct key_msghdr *, len);
      if (km == 0)
    return(ENOBUFS);
      if (key secassoc2msghdr(keynode->secassoc, km, &keyinfo) != 0)
    panic("key dump");
      km->key msglen = len;
      km->key msgvers = KEY VERSION;
      km->key msgtype = KEY DUMP;
      km->key pid = curproc->p_pid;
      km->key seq = seq++;
      km->key errno = 0;
      MGETHDR(m, M WAIT, MT DATA);
      m->m len = m->m_pkthdr.len = 0;
      m->m next = 0;
      m->m nextpkt = 0;
      m->m pkthdr.rcvif = 0;
      m copyback(m, 0, len, (caddr_t)km);
      KFree(km);
      if (sbappendaddr(&so->so_rcv, &key_src, m, (struct mbuf *)0) == 0)
    m_free(m);
      else
    sorwakeup(so);
    }
  }
```

```
  K Malloc(km, struct key_msghdr *, sizeof(struct key_msghdr));
  if (km == 0)
    return(ENOBUFS);
  bzero((char *)km, sizeof(struct key msghdr));
  km->key msglen = sizeof(struct key_msghdr);
  km->key msgvers = KEY VERSION;
  km->key msgtype = KEY DUMP;
  km->key pid = curproc->p_pid;
  km->key seq = 0;
  km->key errno = 0;
  MGETHDR(m, M WAIT, MT DATA);
  m->m len = m->m_pkthdr.len = 0;
  m->m next = 0;
  m->m nextpkt = 0;
  m->m pkthdr.rcvif = 0;
  m copyback(m, 0, km->key_msglen, (caddr_t)km);
  KFree(km);
  if (sbappendaddr(&so->so_rcv, &key_src, m, (struct mbuf *)0) == 0)
    m free(m);
  else
    sorwakeup(so);
  DPRINTF(IDL_GROSS_EVENT,("Leaving key_dump()\n"));
  return(0);
}

/*----------------------------------------------------------------------
 * key delete():
 *      Delete a security association from the key table.
 ---------------------------------------------------------------------*
 /
int
key_delete(secassoc)
     struct ipsec_assoc *secassoc;
{
  char buf[MAXHASHKEYLEN];
  int len, indx;
  struct key tblnode *keynode = 0;
  struct key tblnode *prevkeynode = 0;
  struct socketlist *socklp, *deadsocklp;
  struct key so2spinode *np, *prevnp;
  struct key_allocnode *ap, *prevap;
  int s;

  DPRINTF(IDL_GROSS_EVENT,("Entering key_delete w/secassoc=0x%x\n",
  secassoc));

  if (secassoc->type == SS ENCRYPTION NETWORK)
    secassoc->type = SS_ENCRYPTION_TRANSPORT;

  bzero((char *)&buf, sizeof(buf));
  len = key_createkey((char *)&buf, secassoc->type,
             (struct sockaddr *)&(secassoc->src),
             (struct sockaddr *)&(secassoc->dst),
             secassoc->spi, 0);
  indx = key gethashval((char *)&buf, len, KEYTBLSIZE);
  DPRINTF(IDL_GROSS_EVENT,("keydelete: keytbl hash position=%d\n",
  indx));
  keynode = key_search(secassoc->type, (struct sockaddr *)&(secassoc->
  src),
               (struct sockaddr *)&(secassoc->dst),
               secassoc->spi, indx, &prevkeynode);

  if (keynode) {
    s = splnet();
    DPRINTF(IDL EVENT,("keydelete: found keynode to delete\n"));
    keynode->secassoc->state |= K_DEAD;

```

```
    if (keynode->ref count > 0) {
      DPRINTF(IDL MAJOR EVENT,("keydelete: secassoc still held, marking
      for deletion only!\n"));
      splx(s);
      return(0);
    }

    prevkeynode->next = keynode->next;

    /*
     *  Walk the socketlist and delete the
     *  entries mapping sockets to this secassoc
     *  from the so2spi table.
     */
    DPRINTF(IDL GROSS EVENT,("keydelete: deleting socklist..."));
    for(socklp = keynode->solist->next; socklp; ) {
      prevnp = &so2spitbl[((u int32)(socklp->socket)) % SO2SPITBLSIZE];
      for(np = prevnp->next; np; np = np->next) {
    if ((np->socket == socklp->socket) && (np->keynode == keynode)) {
      prevnp->next = np->next;
      KFree(np);
      break;
    }
    prevnp = np;
      }
      deadsocklp = socklp;
      socklp = socklp->next;
      KFree(deadsocklp);
    }
    DPRINTF(IDL_GROSS_EVENT,("done\n"));
    /*
     * If an allocation entry exist for this
     * secassoc, delete it.
     */
    bzero((char *)&buf, sizeof(buf));
    len = key createkey((char *)&buf, secassoc->type,
            (struct sockaddr *)&(secassoc->src),
            (struct sockaddr *)&(secassoc->dst),
            0, 1);
    indx = key gethashval((char *)&buf, len, KEYALLOCTBLSIZE);
    DPRINTF(IDL_GROSS_EVENT,("keydelete: alloctbl hash position=%d\n",
    indx));
    prevap = &keyalloctbl[indx];
    for (ap = prevap->next; ap; ap = ap->next) {
      if (ap->keynode == keynode) {
    prevap->next = ap->next;
    KFree(ap);
    break;
      }
      prevap = ap;
    }

    if (keynode->secassoc->iv)
      KFree(keynode->secassoc->iv);
    if (keynode->secassoc->key)
      KFree(keynode->secassoc->key);
    KFree(keynode->secassoc);
    if (keynode->solist)
      KFree(keynode->solist);
    KFree(keynode);
    splx(s);
    return(0);
  }
  return(-1);
}
```

```
/*-------------------------------------------------------------------
 * key flush():
 *      Delete all entries from the key table.
 -----------------------------------------------------------------*
 /
void
key_flush(void)
{
  struct key_tblnode *keynode;
  int i;

  /*
   * This is slow, but simple.
   */
  DPRINTF(IDL_FINISHED,("Flushing key table..."));
  for (i = 0; i < KEYTBLSIZE; i++) {
    while (keynode = keytable[i].next)
      if (key delete(keynode->secassoc) != 0)
    panic("key_flush");
  }
  DPRINTF(IDL_FINISHED,("done\n"));
}


/*-------------------------------------------------------------------
 * key getspi():
 *      Get a unique spi value for a key management daemon/program.  The

 *      spi value, once assigned, cannot be assigned again.
 -----------------------------------------------------------------*
 /
int
key_getspi(type, src, dst, spi)
      u int type;
      struct sockaddr *src;
      struct sockaddr *dst;
      u_int32 *spi;
{
  struct ipsec assoc *secassoc;
  struct key_tblnode *keynode, *prevkeynode;
  int count, done, len, indx;
  int maxcount = 1000;
  u int32 val;
  char buf[MAXHASHKEYLEN];
  int s;


  DPRINTF(IDL MAJOR EVENT,("Entering getspi w/type=%d\n",type));
  if (!(src && dst))
    return(-1);

  /*
   *  For storage purposes, the two esp modes are
   *  treated the same.
   */
  if (type == SS ENCRYPTION NETWORK)
    type = SS_ENCRYPTION_TRANSPORT;

  done = count = 0;
  do {
    count++;
    /*
     *  Currently, valid spi values are 32 bits wide except for
     *  the value of zero.  This need to change to take into
     *  account more restrictive spi ranges.
     *
     *  TODO:  Kebe says to allow key mgnt daemon to specify range
```

```
     *         of valid spi to get.
     */
    val = random();
    DPRINTF(IDL_FINISHED,("%u ",val));
    if (val) {
      DPRINTF(IDL_FINISHED,("\n"));
      bzero(&buf, sizeof(buf));
      len = key createkey((char *)&buf, type, src, dst, val, 0);
      indx = key gethashval((char *)&buf, len, KEYTBLSIZE);
      if(!key search(type, src, dst, val, indx, &prevkeynode)) {
    s = splnet();
    K Malloc(secassoc, struct ipsec_assoc *, sizeof(struct
    ipsec assoc));
    if (secassoc == 0) {
      DPRINTF(IDL_CRITICAL,("key_getspi: can't allocate memory\n"));
      splx(s);
      return(-1);
    }
    bzero((char *)secassoc, sizeof(struct ipsec_assoc));

    DPRINTF(IDL FINISHED,("getspi: indx=%d\n",indx));
    secassoc->len = sizeof(struct ipsec_assoc);
    secassoc->type = type;
    secassoc->spi = val;
    secassoc->state |= K LARVAL;
    if (my addr((struct sockaddr *)&(secassoc->dst)))
      secassoc->state |= K INBOUND;
    if (my addr((struct sockaddr *)&(secassoc->src)))
      secassoc->state |= K_OUTBOUND;

    bcopy((char *)src, (char *)&(secassoc->src), src->sa len);
    bcopy((char *)dst, (char *)&(secassoc->dst), dst->sa_len);
    secassoc->from.sin6 family = AF INET6;
    secassoc->from.sin6_len = sizeof(struct sockaddr_in6);

    /*
     * We need to add code to age these larval key table
     * entries so they don't linger forever waiting for
     * a KEY UPDATE message that may not come for various
     * reasons.  This is another task that key_reaper can
     * do once we have it coded.
     */
    secassoc->lifetime1 = time.tv_sec + maxlarvallifetime;

    if (!(keynode = key addnode(indx, secassoc))) {
      DPRINTF(IDL_CRITICAL,("key_getspi: can't add node\n"));
      splx(s);
      return(-1);
    }
    DPRINTF(IDL_FINISHED,("key_getspi: added node 0x%x\n",keynode));
    done++;
    splx(s);
      }
    }
  } while ((count < maxcount) && !done);
  DPRINTF(IDL FINISHED,("getspi returns
  w/spi=%u,count=%d\n",val,count));
  if (done) {
    *spi = val;
    return(0);
  } else {
    *spi = 0;
    return(-1);
  }
}


```

```
/*----------------------------------------------------------------------
 * key update():
 *      Update a keytable entry that has an spi value assigned but is
 *      incomplete (e.g. no key/iv).
 ----------------------------------------------------------------------*
 /
int
key_update(secassoc)
     struct ipsec_assoc *secassoc;
{
  struct key tblnode *keynode, *prevkeynode;
  struct key allocnode *np = 0;
  u int8 newstate;
  int len, indx, inbound, outbound;
  char buf[MAXHASHKEYLEN];
  int s;

  /*
   *  For storage purposes, the two esp modes are
   *  treated the same.
   */
  if (secassoc->type == SS ENCRYPTION NETWORK)
    secassoc->type = SS_ENCRYPTION_TRANSPORT;

  bzero(&buf, sizeof(buf));
  len = key_createkey((char *)&buf, secassoc->type,
                  (struct sockaddr *)&(secassoc->src),
                  (struct sockaddr *)&(secassoc->dst),
                  secassoc->spi, 0);
  indx = key gethashval((char *)&buf, len, KEYTBLSIZE);
  if(!(keynode = key search(secassoc->type,
                    (struct sockaddr *)&(secassoc->src),
                    (struct sockaddr *)&(secassoc->dst),
                    secassoc->spi, indx, &prevkeynode))) {
    return(ESRCH);
  }
  if (keynode->secassoc->state & K_DEAD)
    return(ESRCH);

  /* Should we also restrict updating of only LARVAL entries ? */

  s = splnet();

  inbound = my addr((struct sockaddr *)&(secassoc->dst));
  outbound = my_addr((struct sockaddr *)&(secassoc->src));

  newstate = keynode->secassoc->state;
  newstate &= ~K_LARVAL;
  if (inbound)
    newstate |= K_INBOUND;
  if (outbound)
    newstate |= K_OUTBOUND;

  if (outbound || (!inbound && !outbound)) {
    K Malloc(np, struct key_allocnode *, sizeof(struct key_allocnode));
    if (np == 0) {
      DPRINTF(IDL_CRITICAL,("keyupdate: can't allocate allocnode!\n"));
      splx(s);
      return(ENOBUFS);
    }
  }

  /*
   *  We now copy the secassoc over. We don't need to copy
   *  the key and iv into new buffers since the calling routine
   *  does that already.
   */
```

```

    *(keynode->secassoc) = *secassoc;
    keynode->secassoc->state = newstate;

    /*
     * Should we allow a null key to be inserted into the table ?
     * or can we use null key to indicate some policy action...
     */

    if (keynode->secassoc->key &&
         (keynode->secassoc->type == SS_ENCRYPTION TRANSPORT) &&
         ((keynode->secassoc->algorithm == IPSEC ALGTYPE ESP DES_CBC) ||
      (keynode->secassoc->algorithm == IPSEC ALGTYPE_ESP_3DES)))
        des_set_odd_parity(keynode->secassoc->key);

    /*
     *  We now add an entry to the allocation table for this
     *  updated key table entry.
     */
    if (outbound || (!inbound && !outbound)) {
      len = key createkey((char *)&buf, secassoc->type,
              (struct sockaddr *)&(secassoc->src),
              (struct sockaddr *)&(secassoc->dst),
              0, 1);
      indx = key gethashval((char *)&buf, len, KEYALLOCTBLSIZE);
      DPRINTF(IDL_FINISHED,("keyupdate: keyalloc hash position=%d\n",
      indx));
      np->keynode = keynode;
      np->next = keyalloctbl[indx].next;
      keyalloctbl[indx].next = np;
    }

    key deleteacquire(secassoc->type, (struct sockaddr
    *)&(secassoc->dst));

    splx(s);
    return(0);
  }

  /*----------------------------------------------------------------------
   * key register():
   *      Register a socket as one capable of acquiring security
   associations
   *      for the kernel.
   ----------------------------------------------------------------------*
   /
  int
  key_register(socket, type)
       struct socket *socket;
       u_int type;
  {
    struct key registry *p, *new;
    int s = splnet();

    DPRINTF(IDL MAJOR_EVENT,("Entering key_register w/so=0x%x,type=%d\n",
    socket,type));

    if (!(keyregtable && socket))
      panic("key_register");

    /*
     * Make sure entry is not already in table
     */
    for(p = keyregtable->next; p; p = p->next) {
      if ((p->type == type) && (p->socket == socket)) {
        splx(s);
        return(EEXIST);
```

```
    }
  }

  K Malloc(new, struct key_registry *, sizeof(struct key_registry));
  if (new == 0) {
    splx(s);
    return(ENOBUFS);
  }
  new->type = type;
  new->socket = socket;
  new->next = keyregtable->next;
  keyregtable->next = new;
  splx(s);
  return(0);
}

/*----------------------------------------------------------------------
 * key unregister():
 *       Delete entries from the registry list.
 *          allflag = 1 : delete all entries with matching socket
 *          allflag = 0 : delete only the entry matching socket and type
 ---------------------------------------------------------------------*
 /
void
key_unregister(socket, type, allflag)
     struct socket *socket;
     u int type;
     int allflag;
{
  struct key registry *p, *prev;
  int s = splnet();

  DPRINTF(IDL MAJOR EVENT,("Entering key unregister
  w/so=0x%x,type=%d,flag=%d\n",socket, type, allflag));

  if (!(keyregtable && socket))
    panic("key register");
  prev = keyregtable;
  for(p = keyregtable->next; p; p = p->next) {
    if ((allflag && (p->socket == socket)) ||
    ((p->type == type) && (p->socket == socket))) {
      prev->next = p->next;
      KFree(p);
      p = prev;
    }
    prev = p;
  }
  splx(s);
}


/*----------------------------------------------------------------------
 * key acquire():
 *       Send a key acquire message to all registered key mgnt daemons
 *       capable of acquire security association of type type.
 *
 *       Return: 0 if succesfully called key mgnt. daemon(s)
 *               -1 if not successfull.
 ---------------------------------------------------------------------*
 /
int
key_acquire(type, src, dst)
     u int type;
     struct sockaddr *src;
     struct sockaddr *dst;
{
  struct key_registry *p;
```

```
  struct key acquirelist *ap, *prevap;
  int success = 0, created = 0;
  struct socket *last = 0;
  struct mbuf *m = 0;
  u int etype;
  extern struct sockaddr key_src;

  DPRINTF(IDL_MAJOR_EVENT,("Entering key_acquire()\n"));

  if (!keyregtable || !src || !dst)
    return (-1);

  /*
   * We first check the acquirelist to see if a key_acquire
   * message has been sent for this destination.
   */
  etype = type;
  if (etype == SS ENCRYPTION NETWORK)
    etype = SS ENCRYPTION TRANSPORT;
  prevap = key acquirelist;
  for(ap = key acquirelist->next; ap; ap = ap->next) {
    if (addrpart equal(dst, (struct sockaddr *)&(ap->target)) &&
    (etype == ap->type)) {
      DPRINTF(IDL MAJOR EVENT,("acquire message previously sent!\n"));
      if (ap->expiretime < time.tv sec) {
    DPRINTF(IDL_MAJOR_EVENT,("acquire message has expired!\n"));
    ap->count = 0;
    break;
      }
      if (ap->count < maxkeyacquire) {
    DPRINTF(IDL MAJOR_EVENT,("max acquire messages not yet
    exceeded!\n"));
    break;
      }
      return(0);
    } else if (ap->expiretime < time.tv_sec) {
      /*
       *  Since we're already looking at the list, we may as
       *  well delete expired entries as we scan through the list.
       *  This should really be done by a function like key reaper()
       *  but until we code key_reaper(), this is a quick and dirty
       *  hack.
       */
      DPRINTF(IDL_MAJOR_EVENT,("found an expired entry...deleting
      it!\n"));
      prevap->next = ap->next;
      KFree(ap);
      ap = prevap;
    }
    prevap = ap;
  }

  /*
   * Scan registry and send KEY ACQUIRE message to
   * appropriate key management daemons.
   */
  for(p = keyregtable->next; p; p = p->next) {
    if (p->type != type)
      continue;

    if (!created) {
      struct key_msghdr *km;
      int len;

      len = sizeof(struct key_msghdr) + ROUNDUP(src->sa_len) +
    ROUNDUP(dst->sa len);
      K_Malloc(km, struct key_msghdr *, len);
```

```
        if (km == 0) {
      DPRINTF(IDL_CRITICAL,("key_acquire: no memory\n"));
      return(-1);
        }
        DPRINTF(IDL_FINISHED,("key_acquire/created: 1\n"));
        bzero((char *)km, len);
        km->key_msglen = len;
        km->key_msgvers = KEY_VERSION;
        km->key_msgtype = KEY_ACQUIRE;
        km->type = type;
        DPRINTF(IDL_FINISHED,("key_acquire/created: 2\n"));
        /*
         * This is inefficient and slow.
         */

        /*
         * We zero out sin_zero here for AF_INET addresses because
         * ip_output() currently does not do it for performance reasons.
         */
        if (src->sa_family == AF_INET)
      bzero((char *)(&((struct sockaddr_in *)src)->sin_zero),
            sizeof(((struct sockaddr_in *)src)->sin_zero));
        if (dst->sa_family == AF_INET)
      bzero((char *)(&((struct sockaddr_in *)dst)->sin_zero),
            sizeof(((struct sockaddr_in *)dst)->sin_zero));

        bcopy((char *)src, (char *)(km + 1), src->sa_len);
        bcopy((char *)dst, (char *)((int)(km + 1) + ROUNDUP(src->sa_len)),
          dst->sa_len);
        DPRINTF(IDL_FINISHED,("key_acquire/created: 3\n"));
        MGETHDR(m, M_WAIT, MT_DATA);
        m->m_len = m->m_pkthdr.len = 0;
        m->m_next = 0;
        m->m_nextpkt = 0;
        m->m_pkthdr.rcvif = 0;
        m_copyback(m, 0, len, (caddr_t)km);
        KFree(km);
        DPRINTF(IDL_FINISHED,("key_acquire/created: 4\n"));
        DDO(IDL_FINISHED,dump_mchain(m));
        created++;
      }
      if (last) {
        struct mbuf *n;
        if (n = m_copy(m, 0, (int)M_COPYALL)) {
      if (sbappendaddr(&last->so_rcv, &key_src, n, (struct mbuf *)0) == 0)
        m_freem(n);
      else {
        sorwakeup(last);
        success++;
      }
        }
        DPRINTF(IDL_FINISHED,("key_acquire/last: 1\n"));
      }
      last = p->socket;
    }
    if (last) {
      if (sbappendaddr(&last->so_rcv, &key_src, m, (struct mbuf *)0) == 0)
        m_freem(m);
      else {
        sorwakeup(last);
        success++;
      }
      DPRINTF(IDL_FINISHED,("key_acquire/last: 2\n"));
    } else
      m_freem(m);

    /*
```

```
     *  Update the acquirelist
     */
    if (success) {
      if (!ap) {
        DPRINTF(IDL MAJOR EVENT,("Adding new entry in acquirelist\n"));
        K Malloc(ap, struct key_acquirelist *, sizeof(struct
        key acquirelist));
        if (ap == 0)
      return(success ? 0 : -1);
        bzero((char *)ap, sizeof(struct key acquirelist));
        bcopy((char *)dst, (char *)&(ap->target), dst->sa_len);
        ap->type = etype;
        ap->next = key acquirelist->next;
        key_acquirelist->next = ap;
      }
      DPRINTF(IDL_EVENT,("Updating acquire counter and expiration
      time\n"));
      ap->count++;
      ap->expiretime = time.tv_sec + maxacquiretime;
    }
    DPRINTF(IDL MAJOR EVENT,("key_acquire: done! success=%d\n",success));
    return(success ? 0 : -1);
  }

  /*----------------------------------------------------------------------
   * key alloc():
   *      Allocate a security association to a socket.  A socket
   requesting
   *      unique keying (per-socket keying) is assigned a security
   assocation
   *      exclusively for its use.  Sockets not requiring unique keying
   are
   *      assigned the first security association which may or may not be
   *      used by another socket.
   ---------------------------------------------------------------------*
   /
  int
  key_alloc(type, src, dst, socket, unique_key, keynodep)
       u int type;
       struct sockaddr *src;
       struct sockaddr *dst;
       struct socket *socket;
       u int  unique key;
       struct key_tblnode **keynodep;
  {
    struct key tblnode *keynode;
    char buf[MAXHASHKEYLEN];
    struct key allocnode *np, *prevnp;
    struct key_so2spinode *newnp;
    int len;
    int indx;

    DPRINTF(IDL GROSS EVENT,("Entering key_alloc w/type=%u!\n",type));
    if (!(src && dst)) {
      DPRINTF(IDL_CRITICAL,("key_alloc: received null src or dst!\n"));
      return(-1);
    }

    /*
     *  We treat esp-transport mode and esp-tunnel mode
     *  as a single type in the keytable.
     */
    if (type == SS ENCRYPTION NETWORK)
      type = SS_ENCRYPTION_TRANSPORT;

    /*
     * Search key allocation table
```

```
  */
 bzero((char *)&buf, sizeof(buf));
 len = key createkey((char *)&buf, type, src, dst, 0, 1);
 indx = key_gethashval((char *)&buf, len, KEYALLOCTBLSIZE);

#define np type np->keynode->secassoc->type
#define np state np->keynode->secassoc->state
#define np src (struct sockaddr *)&(np->keynode->secassoc->src)
#define np_dst (struct sockaddr *)&(np->keynode->secassoc->dst)

 prevnp = &keyalloctbl[indx];
 for (np = keyalloctbl[indx].next; np; np = np->next) {
  if ((type == np type) && addrpart_equal(src, np_src) &&
  addrpart equal(dst, np_dst) &&
  !(np_state & (K LARVAL | K_DEAD | K_UNIQUE))) {
   if (!(unique_key))
  break;
   if (!(np_state & K_USED))
  break;
  }
  prevnp = np;
 }

 if (np) {
  struct key so2spinode *newnp;
  struct socketlist *newsp;
  int s = splnet();

  DPRINTF(IDL MAJOR EVENT,("key_alloc: found node to allocate\n"));
  keynode = np->keynode;

  K Malloc(newnp, struct key_so2spinode *, sizeof(struct
  key so2spinode));
  if (newnp == 0) {
   DPRINTF(IDL_CRITICAL,("key_alloc: Can't alloc mem for so2spi
   node!\n"));
   splx(s);
   return(ENOBUFS);
  }
  K Malloc(newsp, struct socketlist *, sizeof(struct socketlist));
  if (newsp == 0) {
   DPRINTF(IDL CRITICAL,("key_alloc: Can't alloc mem for
   socketlist!\n"));
   if (newnp)
  KFree(newnp);
   splx(s);
   return(ENOBUFS);
  }

  /*
   * Add a hash entry into the so2spi table to
   * map socket to allocated secassoc.
   */
  DPRINTF(IDL GROSS_EVENT,("key_alloc: adding entry to so2spi
  table..."));
  newnp->keynode = keynode;
  newnp->socket = socket;
  newnp->next = so2spitbl[((u_int32)socket) % SO2SPITBLSIZE].next;
  so2spitbl[((u int32)socket) % SO2SPITBLSIZE].next = newnp;
  DPRINTF(IDL_GROSS_EVENT,("done\n"));

  if (unique_key) {
   /*
    * Need to remove the allocation entry
    * since the secassoc is now unique and
    * can't be allocated to any other socket
    */
```

```
      DPRINTF(IDL MAJOR EVENT,("key alloc: making keynode unique..."));
      keynode->secassoc->state |= K_UNIQUE;
      prevnp->next = np->next;
      KFree(np);
      DPRINTF(IDL_MAJOR_EVENT,("done\n"));
    }
    keynode->secassoc->state |= K USED;
    keynode->secassoc->state |= K_OUTBOUND;
    keynode->alloc_count++;

    /*
     * Add socket to list of socket using secassoc.
     */
    DPRINTF(IDL GROSS EVENT,("key_alloc: adding so to solist..."));
    newsp->socket = socket;
    newsp->next = keynode->solist->next;
    keynode->solist->next = newsp;
    DPRINTF(IDL_GROSS EVENT,("done\n"));
    *keynodep = keynode;
    splx(s);
    return(0);
  }
  *keynodep = NULL;
  return(0);
}


/*----------------------------------------------------------------------
 * key free():
 *      Decrement the refcount for a key table entry.  If the entry is
 *      marked dead, and the refcount is zero, we go ahead and delete
 it.
 ---------------------------------------------------------------------*
 /
void
key_free(keynode)
     struct key_tblnode *keynode;
{
  DPRINTF(IDL MAJOR EVENT,("Entering key_free
  w/keynode=0x%x\n",keynode));
  if (!keynode) {
    DPRINTF(IDL_CRITICAL,("Warning: key_free got null pointer\n"));
    return;
  }
  (keynode->ref count)--;
  if (keynode->ref count < 0) {
    DPRINTF(IDL CRITICAL,("Warning: key_free decremented refcount to
    %d\n",keynode->ref_count));
  }
  if ((keynode->secassoc->state & K_DEAD) && (keynode->ref_count <= 0))
  {
    DPRINTF(IDL MAJOR EVENT,("key free: calling key_delete\n"));
    key_delete(keynode->secassoc);
  }
}

/*----------------------------------------------------------------------
 * getassocbyspi():
 *      Get a security association for a given type, src, dst, and spi.
 *
 *      Returns: 0 if sucessfull
 *               -1 if error/not found
 *
 *      Caller must convert spi to host order.  Function assumes spi is
 *      in host order!
 ---------------------------------------------------------------------*
```

```
 /
int
getassocbyspi(type, src, dst, spi, keyentry)
     u int type;
     struct sockaddr *src;
     struct sockaddr *dst;
     u int32 spi;
     struct key_tblnode **keyentry;
{
  char buf[MAXHASHKEYLEN];
  int len, indx;
  struct key_tblnode *keynode, *prevkeynode = 0;

  DPRINTF(IDL_GROSS_EVENT,("Entering getassocbyspi w/type=%u spi=%u\n",
  type,spi));

  /*
   *  We treat esp-transport mode and esp-tunnel mode
   *  as a single type in the keytable.
   */
  if (type == SS ENCRYPTION NETWORK)
    type = SS_ENCRYPTION_TRANSPORT;

  *keyentry = NULL;
  bzero(&buf, sizeof(buf));
  len = key createkey((char *)&buf, type, src, dst, spi, 0);
  indx = key gethashval((char *)&buf, len, KEYTBLSIZE);
  DPRINTF(IDL FINISHED,("getassocbyspi: indx=%d\n",indx));
  DDO(IDL_FINISHED,dump sockaddr(src);dump sockaddr(dst));
  keynode = key search(type, src, dst, spi, indx, &prevkeynode);
  DPRINTF(IDL GROSS EVENT,("getassocbyspi: keysearch
  ret=0x%x\n",keynode));
  if (keynode && !(keynode->secassoc->state & (K DEAD | K LARVAL))) {
    DPRINTF(IDL EVENT,("getassocbyspi: found secassoc!\n"));
    (keynode->ref count)++;
    *keyentry = keynode;
  } else {
    DPRINTF(IDL MAJOR_EVENT,("getassocbyspi: secassoc not found!\n"));
    return (-1);
  }
  return(0);
}


/*----------------------------------------------------------------------
 * getassocbysocket():
 *      Get a security association for a given type, src, dst, and
 socket.
 *      If not found, try to allocate one.
 *      Returns: 0 if successfull
 *              -1 if error condition/secassoc not found (*keyentry =
 NULL)
 *               1 if secassoc temporarily unavailable (*keynetry =
 NULL)
 *                 (e.g., key mgnt. daemon(s) called)
 ----------------------------------------------------------------------*
 /
int
getassocbysocket(type, src, dst, socket, unique_key, keyentry)
     u int type;
     struct sockaddr *src;
     struct sockaddr *dst;
     struct socket *socket;
     u int unique key;
     struct key_tblnode **keyentry;
{
  struct key_tblnode *keynode = 0;
```

```
  struct key so2spinode *np;
  int len, indx;
  u int32 spi;
  u_int realtype;

  DPRINTF(IDL GROSS EVENT,("Entering getassocbysocket w/type=%u
  so=0x%x\n",type,socket));

  /*
   *  We treat esp-transport mode and esp-tunnel mode
   *  as a single type in the keytable.  This has a side
   *  effect that socket using both esp-transport and
   *  esp-tunnel will use the same security association
   *  for both modes.  Is this a problem?
   */
  realtype = type;
  if (type == SS ENCRYPTION NETWORK)
    type = SS_ENCRYPTION_TRANSPORT;

  if (np = key sosearch(type, src, dst, socket)) {
    if (np->keynode && np->keynode->secassoc &&
    !(np->keynode->secassoc->state & (K DEAD | K LARVAL))) {
      DPRINTF(IDL FINISHED,("getassocbysocket: found secassoc!\n"));
      (np->keynode->ref count)++;
      *keyentry = np->keynode;
      return(0);
    }
  }

  /*
   * No secassoc has been allocated to socket,
   * so allocate one, if available
   */
  DPRINTF(IDL EVENT,("getassocbyso: can't find it, trying to
  allocate!\n"));
  if (key_alloc(realtype, src, dst, socket, unique_key, &keynode) == 0)
  {
    if (keynode) {
      DPRINTF(IDL EVENT,("getassocbyso: key_alloc found secassoc!\n"));
      keynode->ref count++;
      *keyentry = keynode;
      return(0);
    } else {
      /*
       * Kick key mgnt. daemon(s)
       * (this should be done in ipsec output policy() instead or
       * selectively called based on a flag value)
       */
      DPRINTF(IDL FINISHED,("getassocbyso: calling key mgnt
      daemons!\n"));
      *keyentry = NULL;
      if (key acquire(realtype, src, dst) == 0)
    return (1);
      else
    return(-1);
    }
  }
  *keyentry = NULL;
  return(-1);
}
```

```
/*----------------------------------------------------------------------
 * key.h :     Declarations and Definitions for Key Engine for BSD.
 *
 * Copyright 1995 by Bao Phan, Randall Atkinson, & Dan McDonald,
 * All Rights Reserved.  All rights have been assigned to the US
 * Naval Research Laboratory (NRL).  The NRL Copyright Notice and
 * License Agreement governs distribution and use of this software.
 *
 *  Patents are pending on this technology.  NRL grants a license
 *  to use this technology at no cost under the terms below with
 * the additional requirement that software, hardware, and
 * documentation relating to use of this technology must include
 * the note that:
 *      This product includes technology developed at and
 *      licensed from the Information Technology Division,
 *  US Naval Research Laboratory.
 *
 ----------------------------------------------------------------------*/
/*----------------------------------------------------------------------
#   @(#)COPYRIGHT   1.1a (NRL) 17 August 1995

COPYRIGHT NOTICE

All of the documentation and software included in this software
distribution from the US Naval Research Laboratory (NRL) are
copyrighted by their respective developers.

This software and documentation were developed at NRL by various
people.  Those developers have each copyrighted the portions that they
developed at NRL and have assigned All Rights for those portions to
NRL.  Outside the USA, NRL also has copyright on the software
developed at NRL. The affected files all contain specific copyright
notices and those notices must be retained in any derived work.

NRL LICENSE

NRL grants permission for redistribution and use in source and binary
forms, with or without modification, of the software and documentation
created at NRL provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
3. All advertising materials mentioning features or use of this software
   must display the following acknowledgement:

    This product includes software developed at the Information
    Technology Division, US Naval Research Laboratory.

4. Neither the name of the NRL nor the names of its contributors
   may be used to endorse or promote products derived from this software
   without specific prior written permission.

THE SOFTWARE PROVIDED BY NRL IS PROVIDED BY NRL AND CONTRIBUTORS ``AS
IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL NRL OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

```

```
The views and conclusions contained in the software and documentation
are those of the authors and should not be interpreted as representing
official policies, either expressed or implied, of the US Naval
Research Laboratory (NRL).

----------------------------------------------------------------------*/


/*
 * PF_KEY messages
 */

#define KEY ADD          1
#define KEY DELETE       2
#define KEY UPDATE       3
#define KEY GET          4
#define KEY ACQUIRE      5
#define KEY GETSPI       6
#define KEY REGISTER     7
#define KEY EXPIRE       8
#define KEY DUMP         9
#define KEY_FLUSH        10

#define KEY VERSION      1
#define POLICY_VERSION   1

/*
 * Security association state
 */

#define K USED           0x1    /* Key used/not used */
#define K UNIQUE         0x2    /* Key unique/reusable */
#define K LARVAL         0x4    /* SPI assigned, but sa incomplete */
#define K ZOMBIE         0x8    /* sa expired but still useable */
#define K DEAD           0x10   /* sa marked for deletion, ready for reaping */
#define K_INBOUND        0x20   /* sa for inbound packets, ie. dst=myhost */
#define K OUTBOUND       0x40   /* sa for outbound packets, ie. src=myhost */

/*
 * Structure for key message header.
 * PF KEY message consists of key msghdr followed by
 * src sockaddr, dest sockaddr, from sockaddr, key, and iv.
 * Assumes size of key message header less than MHLEN.
 */

struct key msghdr {
  u short key msglen;   /* length of message including src/dst/from/key/iv */
  u char  key msgvers;  /* key version number */
  u char  key msgtype;  /* key message type, eg. KEY ADD */
  pid_t   key pid;      /* process id of message sender */
  int     key seq;      /* message sequence number */
  int     key errno;    /* error code */
  u int8  type;         /* type of security association */
  u int8  state;        /* state of security association */
  u int8  label;        /* sensitivity level */
  u int32 spi;          /* spi value */
  u int8  keylen;       /* key length */
  u int8  ivlen;        /* iv length */
  u int8  algorithm;    /* algorithm identifier */
  u int8  lifetype;     /* type of lifetime */
  u int32 lifetime1;    /* lifetime value 1 */
  u_int32 lifetime2;    /* lifetime value 2 */
};
```

```

struct key msgdata {
  struct sockaddr *src;     /* source host address */
  struct sockaddr *dst;     /* destination host address */
  struct sockaddr *from;    /* originator of security association */
  caddr t iv;               /* initialization vector */
  caddr t key;              /* key */
  int ivlen;                /* key length */
  int keylen;               /* iv length */
};

struct policy msghdr {
  u short policy msglen;    /* message length */
  u char  policy msgvers;   /* message version */
  u char  policy msgtype;   /* message type */
  int     policy seq;       /* message sequence number */
  int     policy_errno;     /* error code */
};


#ifdef KERNEL

/*
 * Key engine table structures
 */

struct socketlist {
  struct socket *socket;      /* pointer to socket */
  struct socketlist *next;    /* next */
};

struct key tblnode {
  int alloc count;               /* number of sockets allocated to
  secassoc */
  int ref_count;                 /* number of sockets referencing secassoc
  */
  struct socketlist *solist;     /* list of sockets allocated to secassoc
  */
  struct ipsec assoc *secassoc;  /* security association */
  struct key_tblnode *next;      /* next node */
};

struct key allocnode {
  struct key tblnode *keynode;
  struct key_allocnode *next;
};

struct key so2spinode {
  struct socket *socket;         /* socket pointer */
  struct key_tblnode *keynode;   /* pointer to tblnode containing secassoc
  */
                 /*  info for socket  */
  struct key_so2spinode *next;
};

struct key registry {
  u int8 type;             /* secassoc type that key mgnt. daemon can
  acquire */
  struct socket *socket;  /* key management daemon socket pointer */
  struct key_registry *next;
};

struct key acquirelist {
  u int8 type;                   /* secassoc type to acquire */
  struct sockaddr_in6 target;   /* destination address of secassoc */
  u int32 count;                 /* number of acquire messages sent */
  u_long expiretime;             /* expiration time for acquire message */
```

```
  struct key_acquirelist *next;
};

struct keyso cb {
  int ip4 count;          /* IPv4 */
  int ip6 count;          /* IPv6 */
  int any_count;          /* Sum of above counters */
};

#endif

/*
 * Useful macros
 */

#ifndef KERNEL
#define K Malloc(p, t, n) (p = (t) malloc((unsigned int)(n)))
#define KFree(p) free((char *)p);
#else
#define K Malloc(p, t, n) (p = (t) malloc((unsigned long)(n), M_SECA,
M DONTWAIT))
#define KFree(p) free((caddr_t)p, M_SECA);
#endif /* KERNEL */

#ifdef KERNEL
void   key init  _P((void));
void   key cbinit   P((void));
void   key inittables   P((void));
int    key_secassoc2msghdr __P((struct ipsec_assoc *, struct key_msghdr
*,
                struct key_msgdata *));
int    key_msghdr2secassoc __P((struct ipsec_assoc *, struct key_msghdr
*,
                struct key_msgdata *));
int    key add   P((struct ipsec assoc *));
int    key delete   P((struct ipsec assoc *));
int    key_get   P((u int, struct sockaddr *, struct sockaddr *, u_int32,
            struct ipsec assoc **));
void   key flush   P((void));
int    key dump  _P((struct socket *));
int    key_getspi   P((u int, struct sockaddr *, struct sockaddr *,
               u int32 *));
int    key update  _P((struct ipsec assoc *));
int    key register  _P((struct socket *, u_int));
void   key unregister   P((struct socket *, u int, int));
int    key acquire  _P((u int, struct sockaddr *, struct sockaddr *));
int    getassocbyspi   P((u int, struct sockaddr *, struct sockaddr *,
                  u int32, struct key tblnode **));
int    getassocbysocket   P((u_int, struct sockaddr *, struct sockaddr *,
                 struct socket *, u int, struct key_tblnode **));
void   key free  _P((struct key tblnode *));
int    key output   P((struct mbuf *, struct socket *));
int    key_usrreq __P((struct socket *, int, struct mbuf *, struct mbuf
*,
               struct mbuf *));
#endif
```