Lisp-Machine Data Types

The purpose of this chapter is to categorize and define all the objects that occur in I-machine memory, both visible and invisible. The categorization of a storage object is done according to its data type as specified by its type code. The definitions are presented in order by Lisp-object type.

The essence of I-machine architecture is its support of the execution of the Lisp language *at the hardware level.* This dictates the salient features of individual architectural components. In particular, I-machine data representations reflect the fact that, in a Lisp machine, every datum is a Lisp object. Every word in memory therefore contains either a Lisp object reference or part of the stored representation of a Lisp object. (The only exceptions are forwarding pointers and special markers. "Invisible" to ordinary Lisp code, these are used primarily for system memory management, including garbage collection.)

I-machine architecture is fully type coded: every word in memory has a data-type field. The function of the data-type encoding, to be described in this chapter, is to allow I-machine hardware to discriminate between the types of data it is operating on in order to handle each appropriately. More information in how I-machine instructions use different types of data is contained in another chapter. See the section "Macroinstruction Set".

The chapter first introduces the I-machine's basic storage unit. It then lists the different ways that a Lisp object can be stored in memory and describes the components of these representations. Note the interrelation between object references and stored representations of objects: while a stored representation is the target of an object reference, it can also *contain* object references as part of its structure. This relationship reflects the nature of the Lisp language.

As part of its introduction to stored representations, the chapter discusses those stored objects that are *not* object references, including those that are invisible. This includes forwarding pointers, which are used when list or structure objects are moved. These are discussed here, despite the fact that the structures they are used in have not yet been defined. The general overview of data types encountered in I-machine memory makes forward references to some structures necessary. The reader can make use of the cross references supplied to help clarify these sections.

After the introduction, the body of the chapter describes and defines the structure of each of the Lisp objects that the I-machine architecture accommodates with a specifically assigned data type. The concluding section summarizes the data-type information.

Introduction to Lisp-Machine Objects

Memory Words

Length and Format

Words are the basic unit of storage on the I machine. Every item in memory, including *object references* and *object representations*, is made up of one or more words. Whenever we refer to an address, it is the address of some word. More information on addresses is available elsewhere. See the section "Memory Layout and Addressing".

A word contains 40 bits, which are assigned to the following fields:

Position	*Length*	*Field Name*
<39:38>	2 bits	Cdr Code
<37:32>	6 bits	Data Type
<31:0>	32 bits	Address or Immediate Data

```
+---+--------+-------------------------------+
|CDR| TYPE   |       ADDRESS/DATA            |
+---+--------+-------------------------------+
39  37       31                              0
```

Fields

The *data-type field* indicates what kind of information is stored in a word. Each Lisp object referenced by its own assigned data type is explained in detail in the data-type section. See the section "Data-type Definitions". The functions of data types that do not serve as Lisp object references are described in an introductory section. See the section "Components of Stored Representations".

The *address or immediate data field* is interpreted according to the data type of the word. This field contains either the address of the stored representation of an object, or the actual representation of an object. This is explained in the sections covering the individual data types.

The *cdr-code field* is used for various purposes. For header data types, the cdr-code field is used as an extension of the data-type field. For stored representations of lists, the contents of this field indicate how the data that constitute the list are stored. Other uses of the cdr-code field are for instruction sequencing and for storing information about displaced arrays. Use of the cdr code is explained in the sections on lists, arrays, headers, and compiled functions.

Classes of Stored Object Representations

Figure OBJECT-REPRESENTATIONS illustrates the ways in which objects are represented.

[Figure caption: Classes of stored object representations.]

The storage structures for Lisp objects are introduced here so that the reader will be able to see how the various data types function within them.

There are three fundamentally different ways that Lisp objects are stored in memory. An object can be stored

- as a list,
- as immediate data,
- or as a structure.

A *list* object is an object built out of one or more conses. Refer to the *Reference Guide to Symbolics-Lisp* for the definition of a cons. The representation consists of a block of memory words strung together by means of the cdr codes. Often the block consists of only one or two words, so it is important to avoid the overhead of having an extra header word: this is why list representation and structure representation are different. The following types of objects have list representations:

conses,
lists,
big ratios,
double-precision floating-point numbers,
complex numbers,
dynamic closures,
lexical closures, and
generic functions

Note that there is a difference between the concept of a list as a type of structure and the concept of the data type **user::dtp-list**. All the above data types use list structure, including cdr coding (described later). Only the object references to lists and conses use the data type **user::dtp-list**. (There is no dtp-cons.)

An *immediate* object does not require any additional memory words for its representation. Instead the entire object representation is contained right in the object reference. To be an immediate object, an object type must not be subject to side-effects, must have a small representation, and must have a need for very efficient allocation of new objects of that type. The following types of objects have immediate representations:

small integers (fixnums),
single-precision floating-point numbers,
small ratios,
characters, and
packed instructions.

A *structure* object is represented as a block of memory words. The first word contains a header with a special data type code. Usually all words after the first contain object references. The header contains enough information to determine the size of the object's representation in memory. Further, it contains enough information about the type of the object so that a legal object reference designating this object can be constructed. Structure representation is designed to work for large objects. Some attention is also paid to minimizing overhead for small objects, but there is always at least one word of overhead. The objects represented as structures are:

symbols,
instances,
bignums,
arrays, and
compiled functions.

The stored representation of a list or structure object is contained in some number of consecutive words of memory. Each memory word within the structure may contain

an *object reference*,
a *header*,
a *forwarding pointer*, or
a *special marker*.

The data-type code identifies the word type. For example, an array is represented as a header word, which contains such information as the length of the array, and, following the header, memory words that contain the elements of the array. An object reference to the array contains the address of the first memory word in the stored representation of the array.

Components of Stored Representations

The components of the stored representations to be found in Lisp machine memory are either object references, headers, forwarding (invisible) pointers, or special markers.

Object References

Object references are the mechanism by which one refers to an object. The object reference is the fundamental form of data in this and any Lisp system. Object references are similar in function to the "pointers" of other languages. As noted before, an object reference can both point to the representation of a Lisp object and be a component part of such a representation.

There are three types of object references:

object references *by address*
immediate object references, and
pointers.

Figure OBJECT-REFERENCES illustrates the three types of object references.

[Figure caption: Three types of object references.]

Object references by address are implemented by a memory word whose address field contains the virtual address of the stored representation of the object. Such memory words are categorized as pointer data. Examples of this type of object reference are symbols, lists, and arrays.

Immediate object references are implemented by memory words that directly contain the entire representation of the object. These are implemented by memory words that contain the object in the 32-bit immediate data field. Examples of this type of object reference are small integers (fixnums) and single-precision floating-point numbers.

Pointers are implemented in the same way as object references by address. The difference between these two types is that pointers contain the virtual addresses of locations that do not contain objects: they point instead to locations *within* objects -- for example, to the value cell of a symbol. Pointers are also categorized as pointer data.

Headers

A header is the first word in the stored representation of structure objects. The header marks the boundary between the stored representations of two objects and contains information about the object that it heads. This information is either immediate data, when the header type is **user::dtp-header-i,** or it is the address of some descriptive data, when the header type is **user::dtp-header-p**. The header-i format contains object-specific immediate data in bits <31:0>. The header-p format contains the address of an object-specific item in bits <31:0>. Object references usually use the address of an object's header as the address of the object. (The only exceptions are the object reference to a compiled function and the object reference to an array with a leader, which reference contains the address of the prefix header.)

The cdr-code field of a header word is used as the header-type field: it distinguishes what kind of object the structure represents. The four header types for each type of header format are:

DTP-HEADER-P

Header Type	*Symbolic Name*	*Object Type*
0	**user::%header-type-symbol**	Symbol
1	**user::%header-type-instance**	Instance
2	**user::%header-type-leader**	Array leader
3		*Reserved*

DTP-HEADER-I

Header Type	*Symbolic Name*	*Object Type*
0	**user::%header-type-compiled-function**	Compiled Function
1	**user::%header-type-array**	Array
2	**user::%header-type-number**	Number
3		*Reserved*

It is possible to change the memory location of an object represented by a structure. In this case, the object's header is moved to a new location and the object's old location is filled with a word of data type **user::dtp-header-forward,** an invisible pointer that contains the address of the new location of the reference. The object references in the locations of the old structure are all replaced with pointers of the type **user::dtp-element-forward**, which contain the addresses of the new locations of the objects. This arrangement allows all existing references to the object to continue to work. Refer to Figure MOVED-ARRAY. Forwarding pointers are described more fully in the next section. See the section "Forwarding (Invisible) Pointers".

[Figure caption: Use of forwarding pointers to move an array.]

Forwarding (Invisible) Pointers

A forwarding pointer specifies that a reference to the location containing it should be redirected to another memory location, just as in postal forwarding. These are also called invisible pointers. They are used for a number of internal bookkeeping purposes by the storage management software, including the implementation of extendible arrays.

The data types of the forwarding pointers are:

user::dtp-external-value-cell-pointer
user::dtp-one-q-forward
user::dtp-header-forward
user::dtp-element-forward

An external-value-cell pointer is used to link a symbol's value cell to a closure or instance value cell. It is not invisible to binding and unbinding. See the section "Binding Stack".

A one-q-forward pointer forwards only the cell that contains it, that is, it indicates that the required data is contained at the address specified in the address field of the **user::dtp-one-q-forward** word and that the cdr-code of the required data is the cdr code of the **user::dtp-one-q-forward** word. This pointer is used to link a symbol value or function cell to a wired cell or a compiled-function's function cell, as well for as many other applications.

A header-forward pointer is used when a whole structure is forwarded. This word marks where the header used to be, and contains the address of where the header is now. When an array with a leader is forwarded, **user::dtp-header-forward** pointers replace both the prefix header and the leader header. The other words of the structure are forwarded with **user::dtp-element-forward** pointers. The address field of an element-forward pointer contains the new address of the word that used to be there. The cdr code of the required data is stored with the relocated data -- the cdr code of the header-forward pointer is ignored. Every word of the structure except the headers contains an element-forward pointer.

A header-forward pointer is also used in connection with list representation. List representation is explained fully in another section. See the section "Representation of Lists". When a one-word cons must be expanded to a two-word cons by **rplacd**, a new two-word cons is allocated and the old one-word cons is replaced by a header-forward pointer containing the address of the new cons. (The cdr code of the header-forward pointer is always **user::cdr-nil** for garbage-collection purposes.) The cdr code in the location containing the forwarding pointer is ignored. This is one difference between a header-forward pointer and a one-q-forward pointer: the cdr code in the location containing a one-q-forward pointer is used rather than ignored. See Figure EXPANDING-CONS. This figure illustrates how a cons whose car contains a reference to a fixnum and whose cdr is **nil** is changed when an **rplacd** instruction changes its cdr to another fixnum.

[Figure caption: Use of forwarding pointers to expands a cons.]

*
****

Notes

In the paragraph about one-q-forward pointers the phrase "an internal processor register," was removed. We can reinsert it later when we decide exactly how references to the stack buffer are going to be handled.
****

Special Markers

A special marker indicates that the memory location containing it does not currently contain an object reference. An attempt to use the contents of that location signals an error. The address field of a special marker is used by error-handling software to determine what kind of error should be reported. (The hardware does not use the special-marker address field.)

The data types of the special markers are:

user::dtp-null
user::dtp-monitor-forward
user::dtp-gc-forward

A null special marker is placed in the value cell or function cell of a symbol or in the instance-variable value cell in an instance, in those cases when no value has been assigned. The address field of the null marker contains the address of the name of the variable. This makes it possible for an error handler to return the name of the offending variable when an attempt to use the value of an unbound variable is detected.

A null special marker is also used to initialize a freshly-created virtual memory page in case it is accidentally accessed before an object is created in it. The address field contains the word's own address.

The encoding of the null-special-marker data type is zero. Memory that is initialized to all bits zero thus contains all null words, which will cause a trap if referenced.

The monitor special marker is intended for use with a debugging feature that will allow modifications of a particular storage location to be intercepted. See the section "Exception Handling".

A marker of type **user::dtp-gc-forward** is used by the garbage collector and may only appear in oldspace. When an object is evacuated from oldspace, each word of the object's former representation contains a **user::dtp-gc-forward** that points to the new location of that word. It is categorized here as a special marker, rather than as a pointer, since it is visible only to the garbage-collecting system, never to Lisp code.

Operand-Reference Classification

Immediate data	**user::dtp-fixnum, user::dtp-small-ratio, user::dtp-single-float, user::dtp-character, user::dtp-packed-instruction, user::dtp-spare-immediate-1, user::dtp-spare-immediate-2** (22 type codes)
Pointer data	**user::dtp-double-float, user::dtp-bignum, user::dtp-big-ratio, user::dtp-complex, user::dtp-spare-number, user::dtp-instance, user::dtp-list-instance, user::dtp-array-instance, user::dtp-string-instance, user::dtp-nil, user::dtp-list, user::dtp-array, user::dtp-string, user::dtp-symbol, user::dtp-locative, user::dtp-lexical-closure, user::dtp-dynamic-closure, user::dtp-compiled-function, user::dtp-generic-function, user::dtp-spare-pointer-1, user::dtp-spare-pointer-2, user::dtp-spare-pointer-3, user::dtp-spare-pointer-4, user::dtp-even-pc, user::dtp-odd-pc, user::dtp-call-compiled-even, user::dtp-call-compiled-odd, user::dtp-call-indirect, user::dtp-call-generic, user::dtp-call-compiled-even-prefetch, user::dtp-call-compiled-odd-prefetch, user::dtp-call-indirect-prefetch, user::dtp-call-generic-prefetch** (33 type codes)
Null	**user::dtp-null** (1 type code)
Immediate Header	**user::dtp-header-i** (1 type code)
Pointer Header	**user::dtp-header-p** (1 type code)
HFWD	**user::dtp-header-forward** (1 type code)
EFWD	**user::dtp-element-forward** (1 type code)
1FWD	**user::dtp-one-q-forward** (1 type code)
EVCP	**user::dtp-external-value-cell-pointer** (1 type code)
GC	**user::dtp-gc-forward** (1 type code)
Monitor	**user::dtp-monitor-forward** (1 type code)
Data	The union of immediate data and pointer data (55 type codes)
Header	The union of immediate header and pointer header (2 type codes)
Immediate	The union of immediate data and immediate header (21 type codes)
Pointer	The union of pointer data, null, pointer header, HFWD, EFWD,

1FWD, EVCP, and monitor (42 type codes)

Data-Type Descriptions

This section defines how each type of object is represented in storage and explains how the stored representations make use of type-coded objects.

Representations of Symbols

The object reference to a symbol is a word of data type **user::dtp-symbol** or **user::dtp-nil**. The address field of this word contains the address of a header of type **user::dtp-header-p**. The header is followed by four words. The header's header-type field equals **user::%header-type-symbol** and the address field of the header contains the address of the symbol's name cell. The five words that constitute a symbol object, in order, are:

0	SYMBOL-NAME-CELL	address of the symbol's name
1	SYMBOL-VALUE-CELL	the value, or an unbound marker
2	SYMBOL-FUNCTION-CELL	the definition, or an unbound marker
3	SYMBOL-PROPERTY-CELL	the property list
4	SYMBOL-PACKAGE-CELL	the home package, or NIL

See Figure SYMBOL-REPRESENTATION.

[Figure caption: Structure of a symbol object.]

The special symbols **nil** and **t** reside in fixed memory locations: (**nil** at zone1+0, address 1000000000; and **t** at zone1+5, address 1000000005).

Representations of Instances and Related Data Types

The data types described in this section are used by the flavor system, which deals with flavors, instances, instance variables, generic functions, and message passing. A flavor describes the behavior of a family of similar instances. An instance is an object whose behavior is described by a flavor. An instance variable is a variable that has a separate value associated with each instance. A generic function is a function whose implementation dispatches on the flavor of its first argument and selects a method that gets called as the body of the generic function. Generic functions are described in the section on function data types. See the section "Representation of Functions and Closures". In message passing, an instance is called as a function; the function's first argument, known as the message name, is a symbol that is dispatched upon to select a method that gets called.

See the Lisp documentation for more information about flavors, instances, instance variables, and messages.

Flavor Instances

The object reference to an instance is a word of data type **user::dtp-instance** whose address field points to the instance structure. The stored representation of an instance consist of a header with type **user::dtp-header-p,** whose header-type field equals **user::%header-type-instance.** The words following the header of the instance are the value cells of the instance variables. They contain either object references or an unbound marker. The cdr codes are not used. The address field of the header contains the address of the hash-mask field of a flavor-description structure. This description structure is called a *flavor*.

A flavor contains information shared by all instances of that flavor. The architecturally defined fields of a flavor are:

- the array header, part of the packaging of the structure (It must be a short-prefix array format, but is not checked.)
- the named-structure symbol, part of the packaging of the structure
- the size of an instance, used by the garbage collector and by the instance referencing instructions (**user::%instance-ref** and the like)
- the name of the flavor, used by the **type-of** function
- the hash mask, used by the hardware for method lookup
- the handler hash table address, used by the hardware for method lookup
- additional fields known only to the flavor system

A handler table is a hash table that maps from a generic function or a message to the function to be invoked and a parameter to that function. Typically, the function is a method and the parameter is a mapping table used by that method to access instance variables. The mapping table is a simple, short-prefix ART-Q array. For speed, the format of handler tables is architecturally defined and known by hardware. Handler hash tables are packaged inside arrays, but this is software dependent, not hardware or architecture dependent.

A handler table consists of a sequence of three-word elements. The address of the first word of the first element is in the flavor. Each element consists of:

the key	This is a generic function (**user::dtp-generic-function**), a message name (**user::dtp-symbol**), or **nil**, which is a default that matches everything (**user::dtp-nil**).
the method	This is a program-counter value (**user::dtp-even-pc** or **user::dtp-odd-pc**) addressing the instruction at which the compiled function corresponding to the method is to be entered, or it is a fixnum. A fixnum is used as a special accelerator for instance-variable access: the sign of the fixnum is 0 to get or 1 to set the instance variable and the remaining bits are the offset in the instance of the slot to be accessed.
the parameter	This is a parameter that gets passed from the function or message to the method as an extra argument. If the parameter in the handler table is **nil**, the generic function or message is used as the parameter.

Method entries are normally of type **user::dtp-even-pc** or **user::dtp-odd-pc.** An interpreted method traps to a special entry point to the Lisp interpreter; this is implemented by storing the interpreter (a **user::dtp-compiled-function**) as the method handler and storing the actual method as the parameter.

Each unused three-word slot in the handler hash table, plus a fence slot at the end of the table, is filled with **nil**, a default method function, and **nil**. The default method function takes care of rehashing after a garbage collection, default handling, and error signalling.

Figure INSTANCE-REPRESENTATION illustrates the structure of an instance object, an instance descriptor, and a handler table. Refer to the chapter on function calling to see how instances, methods, and generic functions are applied. See the section "Handler Table".

[Figure caption: The structure of an instance.]

*
****

Notes

The fixnum accelerator for instance variable access could go away. The order of entries in the handler table should be whatever is easiest for the hardware. The header of an instance could point either at the hash mask in the middle of the flavor or at the beginning of the flavor.
****

List Instances

The object reference to a list instance is a word of data type **user::dtp-list-instance** whose address field points to an instance structure. The instance structure for a list instance is the same as that for an ordinary instance. Trap handlers written in Lisp enable other list-manipulation instructions to operate in a generic manner on objects of the list-instance data type. See the section "Flavor Instances".

Array Instances

The object reference to an array instance is a word of data type **user::dtp-array-instance** whose address field points to an instance structure. The instance structure for an array instance is the same as that for an ordinary instance. Trap handlers written in Lisp enable other array-manipulation instructions to operate in a generic manner on objects of the array-instance data type. See the section "Flavor Instances".

String Instances

The object reference to a string instance is a word of data type **user::dtp-string-instance** whose address field points to an instance structure. The instance structure for a string instance is the same as that for an ordinary instance. Trap handlers written in Lisp enable other string-manipulation instructions to operate in a generic manner on objects of the string-instance data type. See the section "Flavor Instances".

Representation of Characters

The object reference to a character is an immediate object of data type **user::dtp-character**, which contains the following fields in its data field:

Position	*Symbolic Name*	*Description*
<31:28> (4 bits)	%%CHAR-BITS	Control, Meta, Super, Hyper bits
<27:16> (12 bits)	%%CHAR-STYLE	Italic, large, bold, ...
<15:8> (8 bits)	%%CHAR-CHAR-SET	Character set
<7:0> (8 bits)	%%CHAR-SUBINDEX	Index within this character set

```
+--+------+----+----------+---------+--------+
|CC| TYPE |BITS|   STYLE  |CHAR-SET |SUBINDEX|
+--+------+----+----------+---------+--------+
39 37     31   27         15        7       0
```

Note that the fields in a character object are *not* used by the hardware. They may change in future software.

*

****

Notes

Note that character format is currently invisible to the hardware.

****

Representations of Numbers

Fixnum Representation

A fixnum is represented by an immediate object whose data field contains a 32-bit, two's-complement integer. Its data type is **user::dtp-fixnum.**

Bignum Representation

The object reference to a bignum is a word of data type **user::dtp-bignum**, whose address field points to the bignum structure. The header word of the structure contains data type **user::dtp-header-i,** with the header-type field equal to **user::%header-type-number**, and **user::%header-subtype-bignum**. (Note that fifteen values of the 4-bit header subtype field are available for expansion.) See Figure BIGNUM-REPRESENTATION. The following fields in the header word are specific to bignums:

Position	*Symbolic Name*	*Description*
<31:28>	%%HEADER-SUBTYPE-FIELD	0 for a bignum
<27>	%%BIGNUM-SIGN	0 for a positive number, 1 for a

```
                                    negative number
 <26:0>    %%BIGNUM-LENGTH          the number of fixnums that follow
```

Note that the hardware does not make use of these header-word fields. Following the header is a sequence of fixnums that make up the bignum. The least-significant part of the bignum is stored in the first fixnum. The fixnums are two's complement and use all 32 bits for each digit. The bignum sign bit is the value of all the most significant bits not explicitly stored in the bignum. Therefore, -1_32. would occupy 2 words: the header with sign 1 and length 1, and a fixnum of 0. (The notation -1_31 stands for a two's complement -1 that has been multiplied by 2^31, that is, shifted left 31 places.)

```
+-------------------------------------------+
|NM|HEADER-I|BIGN|1000000000000000000000000001|
+-------------------------------------------+
39                                         0

+-------------------------------------------+
|CC| FIXNUM |00000000000000000000000000000000|
+-------------------------------------------+
39                                         0
```

1_31. would also occupy 2 words: the header with sign 0 and length 1, and a fixnum that happens to be -1_31.

```
+-------------------------------------------+
|NM|HEADER-I|BIGN|0000000000000000000000000001|
+-------------------------------------------+
39                                         0

+-------------------------------------------+
|CC| FIXNUM |10000000000000000000000000000000|
+-------------------------------------------+
39                                         0
```

[Figure caption: Structure of an object of type **user::dtp-bignum**]

*
****
Notes:

See SYS:SYS2;BIGDEFS.LISP, BIGNUM.LISP. Essentially the same code should run on the I Machine. --Moon
****

Small-Ratio Representation

A small ratio is represented by an immediate object of data type

user::dtp-small-ratio. The data field is divided into two subfields as follows:

```
 Position                     Description
  <31:16>              form a two's-complement numerator.  0 is an
                       illegal value.
  <15:0>               is an unsigned denominator.  0 and 1 are
                       illegal values.
  +--+------+----------------+-----------------+
  |CC|SM-RAT|  NUMERATOR     |  DENOMINATOR    |
  +--+------+----------------+-----------------+
  39 37     31               15                0
```

The illegal values are so because of either division by zero, or because the number is an integer and should be represented as such. Note that the hardware does not make use of the fields of the small ratio.

Big-Ratio Representation

The object reference to a big ratio is a word of data type **user::dtp-big-ratio**, whose address field points to a cons pair. The car of the cons contains the numerator of the ratio, and the cdr contains the denominator. See Figure BIG-RATIO.

[Figure caption: Representation of a big ratio.]

Single-precision Floating-point Representation

A single-precision floating-point number is represented as an immediate object of data type **user::dtp-single-float** whose data field contains a 32-bit IEEE single basic floating-point number. The following fields are defined:

```
Position   Symbolic Name            Description
 <31>     %%SINGLE-SIGN      0 for positive numbers, 1 for negative numbers
 <30:23>  %%SINGLE-EXPONENT  excess-127 exponent
 <22:0>   %%SINGLE-FRACTION  positive fraction, with hidden 1 on the left
   +--+------+-+---------+----------------------+
   |CC|SNG-FL|S| EXPONENT|     FRACTION         |
   +--+------+-+---------+----------------------+
   39 37     31          22                     0
```

*
****

Notes

Same as the 3600, except for 6-bit data-type field.

Do we want to reference the IEEE standard?
****

Double-precision Floating-point Representation

The object reference to a double-precision floating-point number is a word of data type **user::dtp-double-float**. The address field of the double-float word contains the address of a cons pair. See Figure DOUBLE-FLOAT. The data fields in the words of the cons pair hold two fixnums, containing the sign, exponent, and fraction as packed fields. The most-significant word is stored first, violating normal byte-order conventions. The second fixnum contains the low 32 bits of the fraction. The first fixnum contains the following fields:

Position	*Symbolic Name*	*Description*
<31>	%%DOUBLE-SIGN	0 for a positive number, 1 for a negative number
<30:20>	%%DOUBLE-EXPONENT	excess-1023. exponent
<19:0>	%%DOUBLE-FRACTION-HIGH	top 20 bits of fraction (excluding the hidden bit)

```
+--+------+-+----------+---------------------+
|CC| FXNM |S| EXPONENT |    FRACTION-HIGH    |
+--+------+-+----------+---------------------+
39 37      31          19                    0
```

The second fixnum contains one field:

Position	*Symbolic Name*	*Description*
<31:0>	%%FRACTION-LOW	bottom 32 bits of fraction

```
+--+------+----------------------------------+
|CC| FXNM |        FRACTION-LOW              |
+--+------+----------------------------------+
39 37      31                                0
```

In non-generic code double-precision floating-point numbers are often represented with a special immediate representation as a pair of fixnums. Avoiding the normal in-memory object representation saves consing overhead.

[Figure caption: Representation of a double-precision floating-point number.]

*
****

Notes

Similar to the 3600, except that a cons is used instead of a structure to eliminate the overhead of a header word.

Note that the two halves of the number are being stored in arguably the wrong order, since the least-significant bits of the fraction should be first. This is consistent with the 3600. The real basis for deciding should be the order that data are fed into the double-precision floating-point processor chip, if there is one.

****

Complex-Number Representation

The object reference to a complex number is a word of data type **user::dtp-complex**, whose address points to a cons pair. The car of the cons contains the real part of the number, and the cdr contains the imaginary part. See Figure COMPLEX-NUMBER.

[Figure caption: Representation of a complex number.]

The Spare-Number Type

An object reference using **user::dtp-spare-number** can be employed by software to implement additional numeric data types. Functions that require numeric data types as arguments will behave properly (usually trapping out to user-defined handlers) with **user::dtp-spare-number** operands.

*

****

Notes

The representations of numbers has changed significantly from the 3600: there are no longer 16 types of fixnums and 16 types of DTP-FLOAT. The following types are new: DTP-SMALL-RATIO, DTP-BIG-RATIO, DTP-BIGNUM, DTP-BIG-RATIO, DTP-COMPLEX. The types DTP-SINGLE-FLOAT and DTP-DOUBLE-FLOAT have replaced DTP-FLOAT. There is no DTP-EXTENDED-NUMBER.

****

Representations of Lists

The object reference to a list is a word of data type **user::dtp-list**, whose address field contains the address of a word that contains the car of a cons. The storage representation of a list is usually a linked collection of conses. Refer to the *Reference Guide to Symbolics Lisp* for a complete description of conses and lists. In compact form, however, a list can be stored in a sequence of adjacent memory words. See Figure LIST-REPRESENTATION.

[Figure caption: Ordinary and compact list structures.]

The cdr-code tag of a memory word that constitutes an element of a list specifies how to get the cdr of its associated cons according to whether the list is stored in normal linked-list form or in compact form. The cdr-code tag works as follows:

Code	*Symbolic Name*	*Description*
0	**user::cdr-next**	Increment the address to get a reference to the cdr, itself a cons. This is used for compact lists.
1	**user::cdr-nil**	The cdr is **user::nil.** This is used for both kinds of list.
2	**user::cdr-normal**	Fetch the next memory word; it contains a reference to the cdr. This is used for normal lists.
3		*(illegal)*

A typical, that is, not compact, two-word cons has **user::cdr-normal** in the cdr-code tag of its first word and **user::cdr-nil** in that of second. The **car** and **cdr** operations ignore the cdr code in the second word, but it is helpful to the garbage collector.

In general, a list representation consists of a contiguous block of one or more memory words. The cdr code of the last word is always **user::cdr-nil.** The cdr code of the second-to-last word may be **user::cdr-normal** or **user::cdr-next.** The cdr code of every other word is **user::cdr-next.** Note that when a cons consists of exactly two words, the cdr-normal form is always used in its representation, and the cdr code of the second word is always **user::cdr-nil.** If the cons happens to be a list, the cdr code of the first word is **user::cdr-next**; otherwise, the cdr code of the first word of the cons is **user::cdr-normal.**

Note that a **user::dtp-list** pointer can point into the middle of a list representation. This happens any time **user::cdr-next** is used; for instance, if a list of four elements is fully cdr-coded -- that is, it is stored in compact form -- its representation consists of four words. The contents of each word is an element of the list. The cdr codes of the first three words are **user::cdr-next**; the cdr code of the last word is **user::cdr-nil.** An object reference to the cddr of this list has data type **user::dtp-list** and the address of the third word. The garbage collector protects the entire block of storage if any word in it is referenced. See Figure LIST-STRUCTURE.

[Figure caption: An object reference to the **cddr** of a list.]

The **rplacd** operation interacts with cdr coding. An illustration of this was presented in an earlier section. See the section "Forwarding (Invisible) Pointers". **rplacd** of a cons represented with **user::cdr-normal** simply stores into the second word. But **rplacd** of a cons represented with **user::cdr-next** or **user::cdr-nil** must change the representation so that the cdr is represented explicitly before it can be changed. There is one exception; if the cdr is being changed to **nil,** the

user::cdr-nil cdr code is used to represent it. Use of **rplacd** can split an object representation into two independent object representations, one of which might then be garbage-collected.

user::dtp-header-forward is used to implement list forwarding. If the data-type tag (of the car) is **user::dtp-header-forward,** the cdr code is ignored (except by the garbage collector, which expects it to be **user::cdr-nil**). The address in the forwarding pointer points to a pair of words that contain the car and cdr.

Representations of Arrays

The object reference to an array is a word with data type **user::dtp-array** or **user::dtp-string.** The representation of arrays described here does not apply to object references with data type **user::dtp-array-instance** or **user::dtp-string-instance.**

Whether an array is referred to by **user::dtp-array** or **user::dtp-string** has no effect on its stored representation: the data type of the object reference simply serves to make the **stringp** predicate faster.

An array is a structure consisting of a prefix followed by optional data. (Data does not follow the prefix of an array structure if, for example, the array is displaced.) A prefix is defined to be a word whose data type is **user::dtp-header-i** and whose header type is **user::%header-type-array**, followed by zero or more additional words. The prefix defines the type and shape of the array. This is similar to the 3600. The detailed format of the prefix is different from the 3600, and simpler. The data is a sequence of object references or of fixnums containing packed bytes.

The byte fields in a prefix header's 32-bit immediate field are:

Position	*Bits*	*Symbolic Name*	*Description*
<31:26>	6	ARRAY-TYPE-FIELD	Combination of fields below
<31:30>	2	ARRAY-ELEMENT-TYPE	Element type, one of: fixnum, character, boolean, object-reference.
<29:27>	3	ARRAY-BYTE-PACKING	Byte packing. Base 2 logarithm (0 to 5) of the number of elements per word. 6 or 7 in this field is undefined.
<26>	1	ARRAY-LIST-BIT	1 in ART-Q-LIST arrays, 0 otherwise
<25>	1	ARRAY-NAMED-STRUCTURE-BIT	1 in named-structures, 0 otherwise
<24>	1	ARRAY-SPARE-1	(spare for software use)
<23>	1	ARRAY-LONG-PREFIX-BIT	1 if prefix is multiple words
<22:15>	8	ARRAY-LEADER-LENGTH-FIELD	Number of elements in the leader
<14:0>	15	ARRAY-	Use of these bits depends on the prefix type, as described below in the definitions of prefix types

```
  +--+-----+--+---+-+-+-+-+-----+-----------------+
  |AR|HDR-I|TY|BPB|L|S|-|P|L-LEN|                 |
  +--+-----+--+---+-+-+-+-+-----+-----------------+
 39 38    31 30  27       23    15                0
```

Bits <31:27> correspond to the same bits of the control word of an array register. Array registers are discussed in the following section. See the section "I-Machine Array Registers". Bits <26:24> are not used by hardware. Bits <31:27,23> enable various special pieces of hardware (or microcode dispatches). Bits <22:0> are used by hardware under microcode control. Bits <31:26> are sometimes grouped together as ARRAY-TYPE-FIELD.

Some arrays include *packed data* in their stored representation. For example, character strings store each character in a single 8-bit byte. This is more efficient than general arrays, which require an entire word for each element. Accessing the *n*th character of a string fetches the *n*/4th word of the string, extracts the mod(*n*,4)th byte of that word, and constructs an object reference to the character whose code is equal to the contents of the byte. Machine instructions in compiled functions are stored in a similar packed form. For uniformity, the stored representation of an object containing packed data remains a sequence of object references. Each word is an immediate object reference to an integer (that is, the word has data type fixnum), whose 32 bits are broken down into packed fields as required, such as four 8-bit bytes in the case of a character string.

An array can optionally be preceded by a leader, a sequence of object references that implements the array-leader feature. If there is a leader, the leader is preceded by a header of its own, tagged **user::dtp-header-p** and **user::%header-type-leader**; the address field of this header contains the address of the array's main header -- that is, the address of the header of the array prefix. Note that if an array has a leader, the address field of an object reference designating that array contains the address of the main header, the one after the leader, not the address of the header at the beginning of the array's storage, before the leader. Refer to the diagram, Figure SHORT-PREFIX-ARRAYS.

[Figure caption: Short-prefix arrays with and without leaders.]

The address of leader element **user::i** of an array whose address is **user::a**, regardless of whether the prefix is long or short, is given by **(- user::a user::i 1)**.

The two array formats (**user::%array-prefix-short** and **user::%array-prefix-long**) are provided to optimize speed and space for simple, small arrays, which are the most common. Wherever possible fields have been made identical in both formats to simplify the implementation.

Description of the two prefix types:

user::%array-prefix-short:

Position	*Bits*	*Symbolic Name*	*Description*
<14:0>	15	ARRAY-SHORT-LENGTH-FIELD	Length of the array.

```
   +--+-----+--+---+-+-+-+-+-----+-----------+
   |AR|HDR-I|TY|BPB|L|S|-|0|L-LEN| AR-LENGTH |
   +--+-----+--+---+-+-+-+-+-----+-----------+
  39 38    31 30  27       23    14          0
```

The prefix is one word. The array is one-dimensional and not displaced, but may have a leader. Most common arrays including defstructs, editor lines and most arrays with fill-pointers use this type. (You can find out about fill pointers by using the Document Examiner, or refer to the *Reference Guide to Symbolics Lisp.*) See Figure SHORT-PREFIX-ARRAYS.

The address of data element **user::i** of a short-prefix array whose address is **user::a** and whose ARRAY-BYTE-PACKING field is **user::b** is given by (+ **user::a (ash user::i (- user::b)) 1)**. When **user::b** is greater than zero, packed array elements are stored right-to-left within words, thus the right shift to right-justify data element **user::i** is **(ash (logand user::i (1- (ash 1 user::b))) (- 5 user::b))**.

user::%array-prefix-long:

Position	*Bits*	*Symbolic Name*	*Description*
<14:3>	12	ARRAY-LONG-SPARE	Spare.
<2:0>	3	ARRAY-LONG-DIMENSIONS-FIELD	Number of dimensions.

```
 +--+-----+--+---+-+-+-+-+-----+-----+----+
 |AR|HDR-I|TY|BPB|L|S|-|1|L-LEN|SPARE|DIMS|
 +--+-----+--+---+-+-+-+-+-----+-----+----+
39 38    31 30  27      23    14     3    0
```

The long prefix format is used for displaced arrays (including indirect arrays), arrays that are too large to fit in the short-prefix format, and multidimensional (including zero-dimensional) arrays. The first word of the prefix contains the number of dimensions in place of the length of the data. The total length of the prefix is **(+ 4 (* user::d 2))** where **user::d** is the number of dimensions.

The second word of the prefix is the length of the array. For conformally displaced arrays, this is the maximum legal linear subscript, not the number of elements (which may be smaller).

The third word of the prefix is the index offset. This word is always present, even for non-indirect arrays. Zero should be stored here in non-displaced arrays, since the this word is always added to the subscript. Always having an index offset keeps the format uniform and allows the feature that displaced arrays of packed elements can be non-word-aligned (not presently in the 3600, but planned to be added soon in support of some new type of TV).

The fourth word of the prefix is the address of the data. This is a locative to the first word after the prefix for normal arrays, except for normal arrays with no elements, in which case it is a locative to the array itself to avoid pointing to garbage. For displaced arrays, this is a locative or a fixnum. For indirect arrays, this is an array. The cdr code of this word is 0 for a normal array, 1 for a displaced/indirect array.

The remaining words of the prefix consist of two words for each dimension. The first word is the length of that dimension and the second word is the value to multiply that subscript by. Note that this is different from the 3600. See Figure ARRAY-PREFIX-MULTI.

[Figure caption: A two-dimensional array.]

A one-dimensional array with a subscript multiplier not equal to 1 cannot be encached in an array register. Currently the software considers such arrays illegal and will never create one.

The way you tell a displaced/indirect array from a normal array is by checking the cdr code of the fourth word of the prefix (assuming the array has its long prefix bit set). Indirect arrays can be can detected by the data type tag of the fourth word. Figure ARRAY-PREFIX-LONG shows a simple displaced array, while the figure in Figure INDIRECT-ARRAY shows a one-dimensional array indirected to another two-dimensional array. The following code generates two such arrays:

```
(setq a (make-array '(4 7))
      b (make-array 4 :displaced-to a
                      :dispaced-index-offset 10.))
```

[Figure caption: A simple displaced array.]

[Figure caption: A one-dimensional array indirected to a two-dimensional array.]

*
****

Notes:

The precise algorithm to be used when accessing an indirect array will be specified later, as a Lisp program. A prototype of it, using the 3600 array format instead of this array format, can be found in the file V:>Moon>IMach>3600>array.lisp. This was translated from the existing, working 3600 microcode.

Re ARRAY-BYTE-PACKING field, trapping on 6 or 7: [Trapping this is probably a pain - BEE] Deleted: The byte fields in the leader's header word are:

```
<31:23>  9  ARRAY-LEADER-SPARE-1
<22:15>  8  ARRAY-LEADER-LENGTH-FIELD Number of elements in the leader
<14:0>  15  ARRAY-LEADER-SPARE-2
   +--+-----+-------+-----+--------+
   |LD|HDR-I|SPARE-1|L-LEN| SPARE-2|
   +--+-----+-------+-----+--------+
  39 38    31      23    14        0
```

Some static analysis of arrays, in a system 311 world that been used for a week: 99.65% of all arrays are one-dimensional. 2-dimensional and 3-dimensional arrays exist; no higher-dimensional or 0-dim arrays. The average size of an array is 38 words. There is no category of arrays whose average size is larger than will fit in 15 bits; unfortunately I didn't measure the size distribution of arrays directly, so I don't know the percentage of arrays whose size will not fit in 15 bits, but it must be very small. All array types are used at least once. The maximum leader length seen is 38 elements. Unfortunately I didn't measure what fraction of arrays are displaced. --Moon Putting the leader before the header of the array rather than between the prefix and the data, or after the data, increases the size of any array with a leader by one word, because an additional header before the leader is required. This costs 2% for the average array with a leader and 5% for the average non-fonted editor line. This overhead was deemed worthwhile because it simplifies the hardware; otherwise it would be necessary to allow for the size of the leader when addressing the data, or vice versa.

The longest array-leader observed was 38 elements, so a maximum limit of 255 elements should not be restrictive. The maximum on the 3600 is 1023.

The leader header uses **user::dtp-header-p** rather than **user::dtp-header-i** because there were more spare header-type codes available for that type of header.

More information from Rel 6.1 99.54% of the arrays are one-dimensional, of which 99.54 are direct (not displaced). Totals: 453049 arrays, 450961 one-dim, 448900 direct one-dim, 2061 indirect one-dim. The distribution of the LOG2(LENGTH) is as follows: 0: 12493; 1: 16181; 2: 35701; 3: 130120; 4: 93601; 5: 85780; 6: 44053; 7: 26594; 8: 2447; 9: 873; 10: 615; 11: 324; 12: 48; 13: 25; 14: 20; 15: 12; 16: 6; 17: 3; 19: 1; 20: 3.

20.33% of those arrays have a leader The distribution of the LOG2(LEADER-LENGTH) is as follows: 0: 2; 1: 7962; 2: 5870; 3: 3428; 4: 73835; 5: 181; 6: 1.

Note that it may be possible to get rid of the leader header. This has no hardware implications.

The subscript multiplier for the last subscript, at the end of the prefix is always 1 and might be removed.
****

I-Machine Array Registers

An array register is four words on the stack that contain a decoded form of an array, permitting faster access because no reference to the prefix is required. I-machine array registers are essentially the same as those on the L-machine, with the addition of an index-offset feature to allow non-word-aligned array registers with reasonable speed (on the L-machine they are very slow).

The four array-register words on the stack are

Control word a fixnum containing the following packed fields:

Position	*Bits*	*Symbolic Name*	*Description*
<31:30>	2	%%ELEMENT-TYPE	One of: fixnum, character, boolean, or object-reference
<29:27>	3	%%BYTE-PACKING	Base 2 logarithm (0 to 5) of the number of elements per word

`<26:22>`	5	`%%BYTE-OFFSET`	`Offset from word boundary in units of array elements`
`<21:0>`	22	`%%EVENT-COUNT`	`Used for validity checking`

Base address	The address of the first element in the array
Array length	The number of elements in the array
Array	Object reference

The %%EVENT-COUNT field is a copy of the internal processor register array-event-count. This copy is set when the array register is created, and updated by Lisp code whenever an exception is taken because the %%EVENT-COUNT field does not match the array-event-count register. The array-event-count register is incremented by Lisp code whenever the size of an array is changed, invalidating all array registers that have been created. The array-event-count register is by convention always nonzero, forcing the Lisp code to do an extra increment if the new contents would be zero. This convention permits the creation of array registers that always trap, which may be used for encaching objects of type **user::dtp-array-instance** and **user::dtp-string-instance.**

To read an element of an array encached in a array register:

1. If the event count is not equal to the contents of the internal processor register array-event-count, trap and re-decode the array into the array register. This trap need not be handled in hardware/firmware since it will not happen often.

2. Add the low 5 bits of the subscript to %%BYTE-OFFSET; save the 5-bit sum and save the carry out of bit (%%BYTE-PACKING - 1).

3. Compare the subscript against the array length, trap unless **(user::%unsigned-lessp user::subscript length)** is true.

4. Shift the subscript right by <%BYTE-PACKING> bits.

5. Add the shifted subscript, the base address, and the saved carry. Read that memory word.

6. Use the low %%BYTE-PACKING bits of the 5-bit add, %%BYTE-PACKING, and %%ELEMENT-TYPE to extract the array element from the word read from memory.

Much of the above happens in parallel, as it does on the L-machine. The comparison against the array length actually happens after the address is sent to memory, but if the subscript is out of bounds the memory read is cancelled and no page fault occurs.

The following table lists the valid array types for each array element type for all possible values of array byte packing.

array-byte-packing	*fixnum*	*character*	*boolean*	*object*
0	`art-fixnum`	`art-fat-string`	`xxx`	`art-q`
1	`art-16b`	`xxx`	`xxx`	`xxx`
2	`art-8b`	`art-string`	`xxx`	`xxx`
3	`art-4b`	`xxx`	`xxx`	`xxx`

4	art-2b	xxx	xxx	xxx
5	art-1b	xxx	art-boolean	xxx

****

Notes: This (non-word-alligned array registers) can be optimized by an additional 5-bit adder and a special carry input to the main adder.

****

Representations of Functions and Closures

Representation of Compiled Functions

The object reference to a compiled function is a word of data type **user::dtp-compiled-function**, whose address field points to a word inside a compiled-function structure. The compiled-function structure consists of three parts: the prefix, the body, and the suffix. The prefix is two words long and has a fixed format. The body is a sequence of one or more instructions. The suffix is at least one word long and contains debugging information and constant data. The object reference to a compiled function contains the address of the first word in the body, which is usually the first instruction executed when the function is called. The prefix extends to lower addresses. The suffix is at higher addresses than the body. The hardware, however, knows nothing about the format of the prefix or suffix.

I-Machine compiled functions differ from those of the 3600 by not having a constants/external references table, since references to constants and to external value and function cells are stored in-line in the body. In addition, the "args-info" of an I-Machine compiled function is not stored explicitly, since it can easily be reconstructed from the entry instruction by software.

The first word in the prefix is a header word that identifies this object as a compiled function and specifies its size and the sizes of its parts. The bits in this word are:

39-38	%HEADER-TYPE-COMPILED-FUNCTION
37-32	DTP-HEADER-I
31-18	Size of the suffix (14 bits)
17-0	Total size of the object (18 bits)

The second word in the prefix is available for use as the function cell that contains the current definition of the function. Typically the function cell of the symbol that names a function contains a **user::dtp-one-q-forward** invisible pointer with the address of the function cell of the compiled function, which contains a dtp-compiled-function reference to the beginning of its own body. This is the same as on the 3600. If the function is redefined, then the function cell will point someplace else and execution will be slower. If **user::dtp-call-compiled-even/odd** is used, inter-function references bypass the function cell. This is discussed in detail in the chapter on function calling. See the section "Function Entry".

The even half of the first word in the body is the first instruction of the function, known as the entry instruction. This is the point at which execution usually begins. The entry instruction checks the number of arguments. This is discussed in detail in the chapter on function calling. See the section "Function Entry".

The first word in the suffix contains an object reference to a list containing information not needed while executing the function. This information is used mainly by the debugger (also by the compiler and the interpreter). The car of this list is the name of the function and the cdr of the list is an a-list containing information such as names and stack locations of local variables. The cdr code of the first word in the suffix is **user::cdr-nil** (encoded as 1), which is the illegal instruction sequencing code. This word, with this cdr code, serves as a "fence" that prevents instruction fetchahead from running past the end of the body of a function.

If the body contains any full-word function-calling instructions, the suffix contains linkage information beginning at its second word. The linkage information is a sequence of fixnums joined together by cdr-next codes and terminated by a cdr-nil code. There is a 4-bit byte for each full-word function-calling instruction in the body, which contains the number of arguments to that call (0 to 13), or 14 if the number of arguments is larger than 13, in which case the next two 4-bit bytes contain the number of arguments, or 15 if the compiler does not know the number of arguments or does not want the linker to bypass the entry instruction of the called function. If the linkage information terminates with **user::cdr-nil** before all of the full-word function-calling instructions have been accounted for, the missing 4-bit bytes are assumed to contain 15.

Succeeding words of the suffix contain the stored representations of list-type constants used by the function (including double-floats, ratios, and complex numbers). Putting these constants in the suffix of the function that uses them minimizes paging. Structure-type constants are typically stored immediately after the function that uses them, again to minimize paging.

See Figure COMPILED-FUNCTION

Another section in this chapter discusses the data types of the instructions. (See the section "Instruction Representation".) Refer to the chapter on the instruction set for a discussion of instruction sequencing. See the section "Instruction Sequencing".

[Figure caption: The structure of a compiled function.]

*
****

Notes:

Not only does this (using the cdr code 1 as a fence) avoid loading the instruction cache with extraneous words from functions other than the one being executed, but more importantly it avoids a subtle bug involving fetchahead past the free-pointer for allocation of compiled code, after a sequence of timing coincidences has left words there containing valid data types for instructions. The bug is that obsolete data could get into the instruction cache and not get cleared out when a new function was created at the same address.

Note that the hardware does not do the invalidate Icache. Efland

Note that the design is intended to put the function cell and the entry instruction both on the same page and in the same cache line, minimizing the cost of indirecting through a function cell. The loader may want to insert extra words to keep compiled functions aligned on appropriate boundaries so that the function cell and entry instruction always fall into the same cache line, if we have a cache.

****

Generic Functions

An object reference to a generic function has data type **user::dtp-generic-function.** The address field points to a structure whose content is not architecturally defined; it is used internally by the flavor system. See the section "Generic Functions and Message Passing".

Representation of Lexical Closures

The object reference to a lexical closure is a word of data type **user::dtp-lexical-closure**, which points to a cons pair. The car of the cons is the lexical environment, and the cdr is the function.

The lexical environment, in a typical software implementation, is a cdr-coded list of value cells associated with the closure. In such an implementation, this list must be compact, that is, cdr-coded using **user::cdr-next,** since instructions that access the lexical variables compute addresses of the variables simply as an offset past the address of the environment. See Figure LEXICAL-CLOSURE.

When a lexical closure is called as a function, the environment will be made an argument to the function. For more information, refer to the chapter on function calling. See the section "Starting a Function Call".

[Figure caption: The structure of a lexical closure.]

Representation of Dynamic Closures

The object reference to a dynamic closure is a word of data type **user::dtp-dynamic-closure**, which points to a list structure. The format of a dynamic closure is not architecturally defined, but is determined by software. (The hardware traps to Lisp to funcall dynamic closures.)

The list representation allows closures to be stored in the stack (a la **with-stack-list**); certain special forms such as **error-restart** exploit this.

The list is always cdr-coded, but nothing actually depends on this. The first

element of the list is the function. Succeeding elements are taken in pairs. The first element of each pair is a locative pointer to the value cell to be bound when the closure is called. The second element of each pair is a locative pointer to the closure value cell to which that cell is to be linked. See
Figure DYNAMIC-CLOSURE.

[Figure caption: The structure of a dynamic closure.]

Instruction Representation

The instructions in a compiled function are a sequence of words whose data-type field selects among three types of words:

- *Packed instructions* -- data types with type codes 60-77 are used for words that contain two 18-bit instructions. These are the usual stack-machine type instructions, similar to those of the 3600.

- *Full-word instructions* -- data types coded 50 through 57 are used for words that contain a single instruction, with an address field. These are used for starting function calls. In addition, data type **user::dtp-external-value-cell-pointer** (type code 4) is used to fetch the contents of the value cell of a special variable or the function cell of a function and push it on the stack. This is actually an optimization to save space and time (one-half word and one cycle); the value cell address could be pushed as a constant locative and then a **car** instruction could be executed. Besides these, there is one other full-word instruction type, the entry instructions, which do not contain addresses, but instead look like pairs of half-word instructions. These are decoded by their opcode field, not by the data-type field.

- *Constants* -- all other data types encountered among the instructions in a compiled function are constants. The word from the instruction stream is pushed on the stack. The hardware will signal an error if the word is a header or an invisible pointer.

The fields within the various types of instructions are described in the chapter on the instruction set. See the section "Macroinstruction Set".
*
****
Notes:

This scheme, different from the 3600, is designed to eliminate the constants/external-references table in a compiled function and thereby to enable prefetching of such data through the normal instruction pipeline. This saves time and simplifies the hardware by eliminating an addressing mode. H says the average number of references per constant is small enough that this actually saves space, compared to the 3600. In cases where there are many calls to the same function or references to the same constant, the compiler can attemp to encache it

in a local variable.
****

Program-Counter Representations

The program counter (pc) is a register in the I machine that contains the virtual address of the currently executing instruction. Since most instructions are packed two-to-a-word, that address has to include information about which half-word instruction is executing. This information is included in the data-type code of the pc contents; thus there are two pc data types, **user::dtp-even-pc** and **user::dtp-odd-pc**. Words of these data types are not usually found in the stored representations of Lisp objects, but occur within stack frames or inside compiled functions for long branches. See the section "Function Calling, Message Passing, Stack Group Switching".
*
****
Notes

The reference for(Topic={Representation of Stack Groups},Type={Section}) has been removed. The section will probably end up in the function calling chapter.
****

Data-Type Code Assignments

This section summarizes all of the different data types defined by the architecture. The data type of a word is stored in its tag field.

It is important to note that not all data types are necessarily understood completely by a particular implementation. For example, the hardware probably understands that **user::dtp-complex** is a number, but it may not be capable of performing arithmetic operations on complex numbers.

The following tables enumerate all sixty-four data types, along with a brief description of each. Note that the sixty-four types are grouped into several common classes.

Headers, Special Markers, and Forwarding Pointers

Eight types:

Type Code	*Symbolic Name*	*Description*
0	DTP-NULL	Unbound variable/function, uninitialized storage
1	DTP-MONITOR-FORWARD	This cell being monitored
2	DTP-HEADER-P	Structure header, w pointer field
3	DTP-HEADER-I	Structure header, w immediate bits
4	DTP-EXTERNAL-VALUE-CELL-POINTER	Invisible except for binding
5	DTP-ONE-Q-FORWARD	Invisible pointer (forwards 1 cell)
6	DTP-HEADER-FORWARD	Invisible pointer (forwards whole structure)
7	DTP-ELEMENT-FORWARD	Invisible pointer in element of structure

Number Data Types

Eight types:

Type Code	*Symbolic Name*	*Description*
10	DTP-FIXNUM	Small integer
11	DTP-SMALL-RATIO	Ratio with small numerator and denominator
12	DTP-SINGLE-FLOAT	Single-precision floating point
13	DTP-DOUBLE-FLOAT	Double-precision floating point
14	DTP-BIGNUM	Big integer
15	DTP-BIG-RATIO	Ratio with big numerator or denominator
16	DTP-COMPLEX	Complex number
17	DTP-SPARE-NUMBER	A number to the hardware trap mechanism

Instance Data Types

Four types:

Type Code	*Symbolic Name*	*Description*
20	DTP-INSTANCE	Ordinary instance
21	DTP-LIST-INSTANCE	Instance that masquerades as a cons
22	DTP-ARRAY-INSTANCE	Instance that masquerades as an array
23	DTP-STRING-INSTANCE	Instance that masquerades as a string

Primitive Data Types

Eleven types:

Type Code	*Symbolic Name*	*Description*
24	DTP-NIL	The symbol NIL
25	DTP-LIST	A cons
26	DTP-ARRAY	An array that is not a string
27	DTP-STRING	A string
30	DTP-SYMBOL	A symbol other than NIL
31	DTP-LOCATIVE	Locative pointer
32	DTP-LEXICAL-CLOSURE	Lexical closure of a function
33	DTP-DYNAMIC-CLOSURE	Dynamic closure of a function
34	DTP-COMPILED-FUNCTION	Compiled code
35	DTP-GENERIC-FUNCTION	Generic function (see later section)
36	DTP-SPARE-POINTER-1	Spare pointer
37	DTP-SPARE-POINTER-2	Spare pointer
40	DTP-SPARE-IMMEDIATE-1	Spare immediate
41	DTP-SPARE-IMMEDIATE-2	Spare immediate
42	DTP-SPARE-POINTER-3	Spare pointer
43	DTP-CHARACTER	Common Lisp character object
44	DTP-SPARE-POINTER-4	Spare pointer

Note that codes 36, 37, 42, and 44 are spare pointer data types and codes 40 and 41 are spare immediate data types. Object references with these data types can be used perfectly normally, but there are no built-in hardware operations that do anything with them.

Special Marker for Garbage Collector

One type:

Type Code	*Symbolic Name*	*Description*
45	DTP-GC-FORWARD	Object-moved flag for garbage collector

Data Types for Program Counter Values

Two types:

Type Code	*Symbolic Name*	*Description*
46	DTP-EVEN-PC	PC at first packed instruction in word, or of full-word instruction
47	DTP-ODD-PC	PC at second instruction in word

Full-Word Instruction Data Types

Eight types:

Type Code	*Symbolic Name*	*Description*
50	DTP-CALL-COMPILED-EVEN	Start call, address is compiled-function
51	DTP-CALL-COMPILED-ODD	Start call, address is compiled-function
52	DTP-CALL-INDIRECT	Start call, address is function cell
53	DTP-CALL-GENERIC	Start call, address is generic-function
54	DTP-CALL-COMPILED-EVEN-PREFETCH	Same as DTP-CALL-COMPILED-EVEN but prefetch is desirable
55	DTP-CALL-COMPILED-ODD-PREFETCH	Same as DTP-CALL-COMPILED-ODD but prefetch is desirable
56	DTP-CALL-INDIRECT-PREFETCH	Same as DTP-CALL-INDIRECT but prefetch is desirable
57	DTP-CALL-GENERIC-PREFETCH	Same as DTP-CALL-GENERIC but prefetch is desirable

Half-Word Instruction Data Types

Sixteen types:

Type Code	*Symbolic Name*	*Description*
60-77	DTP-PACKED-INSTRUCTION	Used for instructions in compiled code.

Each word of this type contains two 18-bit instructions, which is why sixteen data types are used up. Bits <37-36> contain 3 to select the instruction data type. Bits <39-38>, the cdr code, contain sequencing information described in the chapter on the instruction set. The instruction in bits <17-0> is executed before the instruction in bits <35-18>. See the section "Instruction Sequencing".