ZETA-C$^{(TM)}$ User’s Guide

November 23, 2008

Contents

Introduction and obligatory hype

Lisp Machines – the Symbolics 3600(R) series, LMI LAMBDA(R), and TI Explorer(R) – provide the best software development environments in the world, but to date they have not been very useful for cross-development: it has been difficult to take a program created and debugged on a Lisp Machine and transfer it to some other environment, as this usually requires translating the program from Lisp to some other language. Conversely, programs written in other languages in other environments could not be run on a Lisp Machine.

ZETA-C(TM) helps bridge this portability gap by bringing C – the popular systems language of UNIX(R) – to the Lisp Machine. With ZETA-C, it is possible to develop programs in C and still make use of the power of the Lisp Machine environment.

C, as traditionally implemented, is *unsafe*: it is possible for an erroneous procedure to damage data structures to which it is not intended to have access, typically by storing through an invalid pointer or storing into an array at an invalid index. Indeed, the greater part of debugging a C program often consists of tracking down such errors. Also, errors like these are often especially hard to find, as it will only be sometime later that the damaged data structure will cause the process to crash, and by the time the crash finally comes, a lot of information may have been lost that would be useful in diagnosing the error.

ZETA-C, on the other hand, is *safe*: the automatic array-bounds and pointer-validity checking which are built into the Lisp Machine ensure that no procedure can damage a data sructure to which it doesn't explicitly have access. An attempt to store through an uninitialized pointer, or into an array at an invalid index (whether by way of a pointer or an explicit `array[index]` reference) will be trapped immediately, giving the user the full power of the Lisp Machine's sophisticated debugger. The user can then examine the context in which the damage was about to be done, rather than the unrelated context in which it would, in the traditional scenario, have been discovered. The time required to diagnose the problem is thus reduced from hours or days to minutes.

Another time-saving feature of the Lisp Machine system is its facility for incremental compilation and dynamic linking. Traditionally, when one makes a change to a single procedure, the entire file containing the procedure must be recompiled, and the entire program (which may consist of several files) relinked, before the change can be tested. For even medium-sized programs, this debugging turnaround can easily run to 10 minutes. Under ZETA-C, all of the ”symbol tables” and intermediate data structures (which a traditional system must recreate from scratch for every compilation and linking) are maintained incrementally; so only the procedures changed need be recompiled, and there is no link phase. The turnaround is thus a few tens of *seconds*, independent of the size of the program.

ZETA-C programs live in the Lisp world, and can call Lisp functions and access some kinds of Lisp data structures directly. Thus, it is possible to incrementally convert a program from Lisp to C or vice versa. For instance, one might prototype a program in Lisp, then convert its modules one by one into C, testing the whole system as each module is converted. Conversely, it is easy to interface existing C programs to the Lisp world.

The price of the runtime array-bounds checking is, unavoidably, a certain performance penalty for array- and pointer-intensive C code (and, of course, most C code is array- and pointer-intensive). More subtly, algorithms are written very differently in C than in Lisp, and Lisp Machine architectures are, obviously, optimized for the Lisp way of doing things. As a result, a 3600/3670/3640 runs a typical C program about the speed of a VAX 11/750(R). So we do not expect all CPU-intensive C programs to run acceptably under ZETA-C without some hand-tuning.

Compatibility of ZETA-C with other C implementations is very good. We have used as our reference *C: A Reference Manual* by Samuel P. Harbison and Guy L. Steele Jr. (Prentice-Hall, 1984). Chapter 4 documents the differences between ZETA-C and that standard.

How to use this guide

This User's Guide is organized into five chapters. The first provides an overview of ZETA-C and some examples of its use. The second describes important details of the ZETA-C implementation and its interface to the Lisp Machine environment. The third discusses the extensions that have been made to Zmacs for editing C files. The fourth describes in detail the differences between ZETA-C and the Harbison & Steele standard. (These first four are probably of immediate interest to the first-time user or prospective purchaser.) The fifth is a reference manual for the library of I/O routines and ”system calls”.

This guide assumes throughout that the reader has some familiarity with the Lisp Machine software system and the Common Lisp language. It also assumes a working acquaintance with C.

1 Overview and examples

ZETA-C is a fully *integrated* implementation of C within the Lisp Machine environment. Its design philosophy has been, not to ”glue” a distinct C editing/compilation/execution system onto the side, as it were, of the Lisp Machine software, but rather to *extend* the existing Lisp programming system to incorporate C. Thus, Lisp and C functions can freely call each other; Lisp can access all C data structures directly (no new fundamental datatypes have been created), and C can access many Lisp data structures; C code, like Lisp code, is edited with Zmacs, to which commands specific to C have been added; debugging is done with the same facilities; in fact, ZETA-C compiles a C function by translating it into Lisp and handing the result to the existing Lisp compiler.

To see how all this works in practice, let’s run through a simple example. (This example is designed for you to follow along at your own console, but it will still be comprehensible if you don’t. Following along, of course, assumes that ZETA-C has been installed at your site.) First, see if ZETA-C is loaded into your world; if not or if you don’t know how to tell, type at a Lisp listener `(make-system 'zeta-c :noconfirm)`. Second, get a Zmacs window, and edit a new file named ”`hello.c`” (in your home directory). As the first line of the file, type

```
/* -*- Mode: C; Package: (hello C) -*- */
```

Issue the Zmacs command `meta-X Reparse Attribute List`; you will notice that the Zmacs mode line now says `(C)`, indicating that C mode is active, and the wholine at the bottom of the screen now shows the current package as `HELLO:`. Next, enter the following C function.

```
main()
{
     printf("\nHello, world!\n");
}
```

Now give the Zmacs command `control-shift-C`. You will see in the echo area, first ”`Compiling MAIN`”, then ”`-- compiled.`” Now get to a C listener by typing `[Suspend]`, and type ”`main();`”. You will see

```
Hello, world!
```

Type `[Resume]` or `[Abort]` to return to Zmacs. Now let’s save this program as a file and compile it to produce a `.BIN` file. Save the file with `control-X control-S`, then select a Lisp listener (with `[Select] L`) and type the form

(zeta-c:c-compile-file "*directory* hello.c")

where *directory* is the name of your home directory. The value returned will be the pathname of the `.BIN` file; type ”`(load *)`” to load it, and ”`(hello:|main|)`” to run the program. Note the vertical bars around the symbol `main`. These are necessary to suppress the normal conversion of letters to uppercase.[1] ZETA-C, unlike Lisp, is *case-sensitive*, and will not recognize ”MAIN” or ”Main” as a version of ”main”. Instead these are three different symbols, and in our example, only the last one has been defined. Be careful *not* to extend the upright bars around the package prefix, or Lisp will not recognize it as such. If you wish to avoid using upright bars when calling C from Lisp, simply use all-uppercase names in your C code.

Now let’s try a fancier example. Still in Zmacs, find the file ”`zeta-c:source;turtle.c`”. Give the command `meta-X Compile Buffer` (you will see a message like ”`Warning:  the package TURTLE failed the validation function; the standard value for it will not be changed.`”; ignore it). When the compilation is complete, again hit `[Suspend]`, and type ”`init();`”. The mouse cursor will change to an inverted-L shape, indicating that you are being asked to designate the corners of a window. Make a small (2”) square window which does not overlap with the editor window (this may not be possible, in which case after you make the small window you should use the Edit Screen option on the system menu to reshape the editor window so it does not overlap the new window; in this case type ”`init();`” again after you uncover the new window). You will see a small triangle (the ”turtle”) in the center of your window. Those of you who have ever played with the kids’ programming language Logo will recognize the commands available:

`fd(`*dist*`);`	Move the turtle forward *dist* pixels, drawing a line.
`bk(`*dist*`);`	Move the turtle back *dist* pixels, drawing a line.
`rt(`*angle*`);`	Turn the turtle right by *angle* degrees.
`lt(`*angle*`);`	Turn the turtle left by *angle* degrees.
`pu();`	(”Pen Up”) Raise the turtle’s ”pen”, so that it does not leave a line when it moves.
`pd();`	(”Pen Down”) Lower the turtle’s ”pen”, so that it *does* leave a line when it moves.

These primitives are sufficient for drawing many kinds of pictures, from simple to quite complex. You can define procedures using the C listener, just by typing in the definition as if you were entering it into a C source file. Try, for instance, defining a procedure that draws a square of specified size:

```
square(len) int len; { int i; for (i = 0; i < 4; ++i) { fd(len); rt(90); } }
```

Call it with various arguments. (You may notice that round-off error causes the squares to be slightly distorted. Rewriting the arithmetic in `turtle.c` to prevent this is left as an exercise for the reader.) Write another procedure that displays several squares of different sizes and/or at different orientations or origins. Play! The C listener is a very important tool for interacting with your ZETA-C program, and it is worthwhile to spend a few moments simply getting comfortable with it.

[1] The Lisp reader’s uppercase conversion allows the strings ”`foo`”, ”`Foo`”, and ”`fOo`” (for example) all to be read as the symbol `FOO`; so Lisp code can normally be written in upper, lower, or mixed case. C, on the other hand, requires that such strings be read as distinct identifiers.

2 The ZETA-C Implementation

This chapter discusses details of the ZETA-C implementation and its interface to the Lisp Machine environment, both in terms of the user side (how one goes about compiling and running C programs) and the Lisp side (how one connects Lisp programs to C programs). Because of the open nature of the Lisp Machine software system, these two aspects are often deeply intertwined.

2.1 Packages

It is important to understand how ZETA-C makes use of Lisp **packages**. (The following discussion assumes an understanding of the package system; if you are not familiar with its use, you should read the appropriate section of the Lisp Machine documentation). It is important that names (of variables, functions, etc.) in a C program be kept distinct from those in the Lisp world as well as those in other C programs. For instance, the ZETA-C user must be allowed to define a function `car` without it conflicting with the Lisp function of that name. To accomplish this, first, ZETA-C itself defines a package `C:` which *does not* inherit from the `GLOBAL:` package; then, users define packages which inherit from `C:` in which to intern their programs (we will call these latter ”C program packages”).

The Standard Value system checks, whenever you enter a Lisp Listener (by entering the Debugger because of an error, or by hitting `[Suspend]`), that the current package inherits from `GLOBAL:`; if it doesn't (no C program package does) then it will choose some other package – often `USER:` – to make current. So on entering the debugger you will see a message like ”`Binding PACKAGE to #<Package USER 16600000> (old value was #<Package CPROG 20560347>).`” So when you're in the Debugger or a Break loop, you will have to type an explicit package prefix to access symbols in your C program package.

Also, in ZMACS, whenever you select a buffer in C mode, you will see a message like ”`Warning: the package CPROG failed the validation function; the standard value for it will not be changed.`” These messages can be ignored.

Of course, what usually happens if one omits a needed package prefix is an undefined-function or unbound-symbol error, which can easily be fixed either with one of the fancy Debugger options or by simply hitting `[Abort]` and retyping the form.

See p. 22 to see how to create a C program package.

C identifiers may also contain package prefixes, delimited with the ”`$`” character (which is otherwise unused in C). So, for instance, the C statement ”`TV$BEEP();`”, equivalent to the Lisp ”`(tv:beep)`”, would beep the console beeper (and/or flash the screen).

2.2 Data structures

To interface C and Lisp code, or just to understand some of the performance issues inherent in ZETA-C, it is very handy to know how ZETA-C implements C data structures in terms of those provided by Lisp.

2.2.1 Numbers

ZETA-C does not make any distinction between types `int` and `long`, or between `unsigned int` and `unsigned long`. These are all implemented with the numeric type most natural to the Lisp Machine, namely arbitrary-precision integers. We do not restrict the widths of these types to, say, 32 bits, because doing so would make ZETA-C object code decidedly *slower*.

The treatment of unsigned arithmetic in ZETA-C was problematic. What should be the value, for instance, of `(unsigned)-1`? Since the ”word length” of arbitrary-precision integers is effectively infinite, there is no particular positive integer that it makes sense to use in this case. What we do is this[1]: we use the negative fixnum `-1` as a representation of the unsigned value ∞`-1`, and so on. The only operation that's actually affected by this interpretation is that of comparison: we simply arrange that any unsigned value implemented as a negative integer is treated as *greater* than any value implemented as a positive integer.

The types `short` (equivalent to `signed short`) and `unsigned short` are truncated to 16 bits. This truncation is inefficient on the Lisp Machine, and so for optimum performance we do not recommend the use of the short types for variables or structure components; arrays of them work better (see below). Similarly for `char` (equivalent to `unsigned char`) and `signed char`, which are truncated to 8 bits. The `char` type is unsigned by default not only because of this efficiency difference, but also because C requires that all characters in the host character set be represented by positive values of `char` variables, and the Lisp Machine character set defines several control characters above 0200 octal.

Variables of all integer types are stored so that their actual values are equal to their ”apparent” values (as seen from Lisp). This means that any value greater than 2^{31}-1 will require a bignum to represent it.

The `float` type is implemented with single-precision floating-point numbers, which have a 24-bit mantissa and an 8-bit exponent; `double` uses double-precision (53-bit mantissa, 11-bit exponent). Double-precision arithmetic is rather less efficient than single-precision – not just because the computation takes longer, but also because double-floats are consed in the heap rather than fitting in a pointer, as single-floats do – so ZETA-C provides an option to suppress the automatic conversion of `floats` to `doubles` before performing any arithmetic. Just put the line

```
#define ZETA_C_SUPPRESS_AUTO_FLOAT_TO_DOUBLE
```

at the beginning of your program.

[1] I am indebted to John Rose of Thinking Machines for this solution.

2.2.2 Pointers

The choice of representations for pointers is constrained on the one hand by the ways that pointers can be used in C programs, and on the other by the design goal that ZETA-C be a *safe* implementation. For instance, on the one hand, it is necessary that pointer arithmetic be possible: that one be able to create a pointer to an element of an array, and then add an integer to that to get a pointer to a different element; on the other hand, if in doing this one creates a pointer to a non-existent element, an attempt to assign or reference through that pointer must be trapped.

To satisfy these constraints, it is clearly insufficient that a pointer be, as in C, merely an address. The system must keep track of the array-bounds information that goes with the address. We accomplish this by representing a pointer as a pair <*array*, *index*>, where the *array* is a Lisp array object representing a C array or structure (see the next section), and the *index* specifies an element thereof. How the pointer is actually represented depends on where it is stored. Pointer variables are implemented as pairs of Lisp variables, named *ptr*`.array` and *ptr*`.index`, where *ptr* is the name of the C pointer variable. Pointers as array or structure elements take up two consecutive elements of the Lisp array; the first is *array* and the second is *index*. These implementation matters are of course entirely invisible to C programs, provided it is not assumed that a pointer and an `int` are the same size.

When a pointer is passed to a function, it is passed as two consecutive arguments; just like a pointer variable, a pointer parameter in a function definition turns into a pair of parameters, which appear consecutively in the lambda-list. A pointer is returned from a function as two values, the array and index. Again, this is completely invisible to C code, provided all functions that accept and return pointers are correctly declared.

In order to provide for the creation of pointers by taking the address (unary `&`) of scalar variables (as opposed to elements of aggregates), ZETA-C creates ”address arrays”. The address array of variable *var* is kept in a variable *var*`.address`. (Address arrays are always created for external and static variables, but only if needed for automatic variables.) So, a pointer to *var* is the pair <*var*`.address`, `0`>.

An example should make all of this clear. The C function

```
gubbish(ipp)
     int **ipp;
{
     static int i;
     int *ip;
     ip = &i;
     *ipp = ip;
     frobozz(ip);
     return ip;
}
```

would be implemented in Lisp something like this (much irrelevant detail, notably the initialization of `i` and `i.address`, has been omitted):

```
(defun gubbish (ipp.array ipp.index)
  (let ((ip.array nil) (ip.index 0))
    (setq ip.array i.address)
    (setq ip.index 0)
    (setf (aref ipp.array ipp.index) ip.array)
    (setf (aref ipp.array (1+ ipp.index)) ip.index)
    (frobozz ip.array ip.index)
    (values ip.array ip.index)))
```

Some obvious consequences of these decisions: to increment a pointer we increment its index part; two pointers are equal iff their array parts are `eq` and their index parts are equal; the difference of two pointers is the difference of their index parts (assuming they point into the same array; an error occurs otherwise), scaled down if necessary by the size of the objects they point to.

Special cases: the null pointer (of any type) is represented as having array part `nil` and index part `0`. Any attempt to dereference it will of course cause an error to be signalled, since `nil` is not an array. A pointer-to-function is the exception to *all* of this – it's not a pair, it's just a single value: the symbol that names the function. (A null function pointer is the symbol `zeta-c:null-function-pointer`.)

2.2.3 Arrays, structures, and unions

C arrays are Lisp arrays. Since C and Lisp use the same bounds convention – an array of size *n* contains elements numbered 0 through *n*-1, inclusive – no subscript translation is necessary.

The representation chosen for an array depends on the type of the elements to be stored in it. Arrays of `int`, `long`, `unsigned int`, `unsigned long`, any kind of pointer, or of structures or unions are implemented with Lisp array type `art-q`. Arrays of `short` and `unsigned short` are implemented as `art-16b` arrays; arrays of `char` and `signed char` are implemented as `art-8b` arrays. There is no extra cost associated with using these to hold unsigned values, since that is the interpretation the microcode imposes, but `signed chars` and (`signed`) `shorts` must be sign-extended after being loaded from an array; this renders the use of the signed types slightly slower.

Lisp Machines are fundamentally word-addressed machines. However, in order to accomodate programs that take, for instance, a pointer to `short`, cast it to a pointer to `char`, store two characters at successive locations through that pointer, and then expect both characters to have fit in a single `short` – a nonportable but all too common practice – ZETA-C simulates byte-addressing by means of displaced arrays. To continue this example, consider a pointer, of type "pointer to `short`", which at runtime points to element *i* of an array *a16b* of type `art-16b`. Casting this pointer to a `char`-pointer causes the creation (at runtime) of an `art-8b` array *a8b*, whose length is twice that of *a16b* and whose contents are displaced to overlay those of *a16b*; the result of the cast is a pointer to element 2*i* of *a8b*. If one stores characters through successive values of this pointer (i.e., to elements 2*i*, 2*i*+1, ... of *a8b*) and then looks at the result through the original `short`-pointer, the characters will be seen to have been stored two-per-halfword starting at the low-order end of the halfword.

Types stored in `art-q` arrays can take part in this pointer casting also. However, there is a slight complication, since `art-q` arrays can hold non-numeric objects, and it would not make sense to load or store bytes or halfwords out of or into a Lisp object pointer. Fortunately, the 36xx microcode takes care of this case very nicely. If you try, for instance, to load a byte out of a word that contains the array part of a ZETA-C pointer, you will get an error to the effect that `The AR-1 microcode encountered an ARRAY-WORD that was not a fixnum`. One way to do this is (you can try this in a C listener):

```
char *cp, *cparray[1] = { "Hello there" };
cp = (char *)cparray;
*cp;
```

Structures are represented as `art-q` arrays; the elements of the structure occupy successive elements of the array. ZETA-C, like traditional C implementations, uses a ”flat” representation for nested aggregates, such that the storage occupied by an inner aggregate is part of that occupied by the outer one. So, for instance, the declaration

```
struct foo {
     int foovals[3];
     struct foo *nextfoo;
     } fooarray[20];
```

allocates a single `art-q` array of length 100 (remember, a pointer takes up two cells). Furthermore, arrays of `shorts` or `chars` within a struct will be implemented using the same displaced-array machinery that handles pointer casts; so the declaration

```
struct bar {
     int a;
     struct foo *foop;
     char name[20];
     } abar;
```

allocates an `art-q` of length 8: 1 for `a`, plus 2 for `foop`, plus 5 for `name`. It also creates and caches (see below) an `art-8b` of length 32, displaced onto the `art-q`, so that `name` is accessible as elements 12 through 21 thereof.

Normally, ZETA-C uses an ”unpacked” representation for structures, in the sense that all scalar elements of numeric type are allocated an entire word each. (Arrays within structs are always packed, as we have just discussed.) So, for instance, the declaration

```
struct zot {
     unsigned short z1;
     char z2, z3;
     struct foo *foop;
     } azot;
```

creates an `art-q` of length 5, not 3 as one might imagine (expecting `z1`, `z2`, and `z3` all to fit in the first cell). This is done for performance reasons: we don’t want waste a lot of time switching representations to load structure elements. If for some reason it’s important to your application that structs be ”packed” (so that `z1`, `z2`, and `z3` all *would* fit in the first cell and the total length *would* be 3), use the keyword `packed_struct` in place of `struct` for declaring all structs that need to be packed. (`packed_structs` can be nested inside `structs` and conversely, with complete freedom, though we can’t imagine why this would be useful.)

The implementation of unions is very much like that of structures. A union of aggregates is an `art-q` array whose size is the size of the largest aggregate in the union, and each of whose elements can be referenced as the corresponding element of any of the aggregates. For example, given the declaration

```
union point {
     struct rect_pt {
          float x, y;
          } rp;
     struct polar_pt {
          float r, theta;
          } pp;
     } pnt;
```

`pnt` will be an array of length 2; `pnt.rp.x` and `pnt.pp.r` both refer to element 0 of this array; `pnt.rp.y` and `pnt.pp.theta` both refer to element 1.

When ZETA-C allocates a Lisp array to represent a C aggregate, it sets it up with a named-structure-symbol and other information in the array-leader so the Lisp printer will print it recognizably; e.g., the declaration `char buffer[256];` creates an array that will print as `#{(char [256]) BUFFER}`. Slots are also allocated in the array-leader to cache the displaced-arrays created by pointer-casting. Specifically, here is how ZETA-C uses each element of the array-leader:

Note that one of the last three slots will always contain the array itself, since it has to be in one of the three representations.

2.2.4 Strings

String constants in C code are written, of course, according to the standard C conventions. However, if you look at a string in the C listener, you will see that it doesn't look quite the same when printed out as it does in your code. The special \-sequences will have been converted to the appropriate single characters, and a `NUL` character (which displays as a raised dot, ·) will have been appended. So, for instance:

What you write in your C code:
```
"Hello, world!\n"
```

What is printed:
```
"Hello, world
·"
```

Note that if you pass a Lisp string to a C function, you must be sure to have appended the `NUL` yourself. (If you don't, the C function will get an array-bounds error when it scans off the end of the string, since this is the only way C code has of knowing where a string ends.) Conversely, for Lisp code to make proper use of a C string, the `NUL` must be stripped and the character-pointer converted to a Lisp string. The following functions exist for this purpose.

`zeta-c:string-to-C` *str* *Function*

Given a Lisp string, returns a C character pointer (as two values: array and index) that points to the beginning of a copy of the string. Appends a NUL to the copy.

`zeta-c:string-to-lisp` *str.array str.index &optionalcase* *Function*

Given a C character pointer <*str.array*, *str.index*>, returns a copy of its contents as a Lisp string. *case*, if supplied, may be `:upcase` or `:downcase`, requesting a forced conversion to upper or lower case respectively.

The Lisp Machine's character set is unusual in that the first 128 characters (0 through 177 octal) are all printing characters; the control characters are octal 200 through 237. See your Lisp manual for more details. As long as you use the predefined escape sequences `\r`, `\n`, `\b`, `\t`, `\f`, and `\v`, this will not make any difference to your programs, unless you are using the high-order bits of characters for some special purpose. The sequences `\r` and `\n` both name the `[Return]` character, octal 215; `\b` is `[Overstrike]`, octal 208; `\t` is `[Tab]`, octal 209; and `\f` and `\v` are both `[Page]`, octal 214.

2.3 Debugging hints

This section contains explanations of the error messages issued by ZETA-C, along with some material that will help you interpret them and find the problem efficiently.

2.3.1 Syntax errors

When the ZETA-C parser encounters a syntax error, it displays three lines of context with a marker at the point where the error was detected, like this:

```
<< Error in reading >>
 Error while parsing line 58 of ED-BUFFER: CTEST.C#> GYRO.ZETA-C; ASTARTE::
Expression syntax
Error happened somewhere before the point indicated by> "->HERE<-" in:
{
     *junkp += c
} ->HERE<-
```

In this case, as you see, the error was a missing semicolon. Here are the error messages the parser can emit whose interpretations may not be obvious:

`Expression syntax` *Error*

An error was found at the expression level. Look for a missing semicolon, unbalanced parentheses, or the like.

`External definition syntax` *Error*

An error was found in an external variable declaration or function definition. Look for a mistyped type name, a missing semicolon or comma, or mismatched delimiters.

`Statement syntax` *Error*

An error was found at the statement level. Look for an incorrectly written `if`, `for`, `while`, etc.

When a syntax error occurs, the parser will attempt to recover and continue. In the case of something simple, like a missing semicolon, this will usually work, but many errors will throw the parser off completely. When this happens, there are likely to be more syntax errors; just fix the first error and the rest will probably go away by themselves.

2.3.2 Compilation errors

When an error is encountered in semantic analysis or ”code generation”, a message is issued which displays the offending structures in ZETA-C’s internal representations. So, clearly, in order to be able to fully understand such a message, one should know how to interpret those representations.[2] There are two important representational systems: that used for the parsed input expressions, and that used to describe the types of values.

Representation of expressions

Expressions (and statements) are represented quite straightforwardly as Lisp forms; for instance, the C statement

```
a = b + 3*foo((c == 0) ? d : e) + f[g++];
```

is represented as the Lisp form

```
(= A (+ B (+ (* 3 (FOO (|?:| (== C 0) D E))) ([] F (X++ G)))))
```

From this example, several points should be visible immediately:

- Operators have names which are identical to, or at least strongly suggestive of, their C notations.
- Calls to user functions appear just as they do in Lisp.
- Variable names are converted to upper case (see p. 35).

A complete list of the ZETA-C primitives appears in Figure 2.1. These are all symbols in package `C:`; they name macros which invoke the ZETA-C analysis and translation apparatus. Some of the names end in plus signs to avoid conflict with legal C identifiers (those which are alphabetic, but do not end in ”+”, are ZETA-C reserved words).

The syntax of two of these primitive macros deserves examples. First, let’s look at function definition. Here is a sample C function:

```
char *
foo(x, y)
     int x, y;
{
     int quux;
     bar(x + y);
     }
```

[2] This is not a deficiency on ZETA-C’s part vis-a-vis other compilers, since the latter don’t have such detailed error messages in the first place.

Here is its ZETA-C internal representation:

```
(DEFUNC+ ((CHAR) (* (FCN+ FOO X Y))) (((INT) X Y)) (BLOCK+ (((INT) QUUX)) (BAR (+ X Y))))
```

And here are some sample declarations:

```
char foo(), *bar, *baz[47];
struct thing {
 int who, *why;
 struct thing *this, *that;
 } thingarray[128];
```

And here are their internal representations:

```
(DECL+ (CHAR) (FCN+ FOO) (* BAR) (* ([] BAZ 47)))
(DECL+ ((STRUCT THING ((INT) WHO (* WHY)) (((STRUCT THING)) (* THIS) (* THAT)))) ([] THINGARRAY 12
```

Note some features of this representation:

- The name of the function is embedded inside a type declarator (see below).
- The second subform of the `defunc+` form is a list of parameter declarations.
- The third subform is the body, and is always a `block+` form.
- The first subform of a `block+` form is a list of local variable declarations.

Representation of types

ZETA-C represents types internally as list structure. Figure 2.3 shows the type description language.

Compilation error reference

`Assignments to structures are not allowed.  If you would like to permit them (for UNIX com`

See p. 43.

`Attempt to call` *exp* `of type` *type* `as a function` *Error*

Chances are, either you have a variable with the same name as a function, or you wrote some expression like `(*ftab[ifunc])()` incorrectly, or you declared something incorrectly.

`Attempt to use expression` *exp*`, of type` *type*`, as a predicate` *Error*

Arrays, structures, and unions may not be used as predicates in conditionals, as in, *e.g.*, `if (frob)` ... where `frob` is a structure.

`BREAK not inside WHILE, FOR, DO, or SWITCH` *Error*

A `break` statement may only appear lexically inside one of these constructs.

`CASE not inside SWITCH` *Error*

A `case` statement may only appear in the body of a `switch` statement.

`CONTINUE not inside WHILE, FOR, or DO` *Error*

A `continue` statement may only appear lexically inside one of these constructs.

`Element` *name* `not found in struct/union type` *type* *Error*

You have attempted to reference a structure element which is not present in this structure. Remember that ZETA-C does not use a single namespace for structure elements (see p. 43).

`Excess initializer values:` *vals* *Error*

You have given a list of initializer values which is longer than the aggregate being initialized.

`Expression` *exp* `of type` *type* `cannot be used as an lvalue` *Error*

You have attempted to assign to, increment, or take the address of something which is not a variable or array or structure element.

`Illegal use of storage class` *sclass* `in context` *context* *Error*

The storage class of a declaration does not make sense in the context; for instance, `auto` in an external declaration.

`Initializer expression` *exp* `is not of type` *type* *Error*

The type of the initializer expression supplied does not match the type, *type*, of the variable being initialized.

`Initializer nested too deeply:` *init-exp* *Error*

A brace-delimited list of initializer values was supplied where only a single, undelimited value was expected.

`Internal error:` *message* *Error*

One of ZETA-C's internal consistency checks has been violated. This represents a bug in ZETA-C. If possible, please send a bug report with a full backtrace and a copy of the C code that excited the bug. To get into the error handler in the error context, you may have to set `compiler:warn-on-errors` to `NIL` and recompile.

`Mismatched consequent and alternate types to "?:":` *type1* `and` *type2* *Error*

In a conditional expression, the types of the two expressions on either side of the colon must match (one must be coercible into the other).

`Parameter` *name* `appeared more than once in the parameter list` *list* *Error*

The same name may not be used for two different parameters of a function.

`Returning structures from functions is not allowed.` *Error*

The treatment of structures as firstclass objects has been disabled. If you would like to enable it – to allow structures to be assigned, passed as arguments, and returned from functions – set `zeta-c:*firstclass-structures*` to `T`. (See p. 43).

`The body of a switch statement must be a block, with no declarations` *Error*

This is a ZETA-C restriction. (Blocks *inside* the body may have declarations.)

`Type mismatch between function declaration and RETURN value:` *type1* `declared,` *type2* `returnec`

The value you are attempting to return is not of the same type as the declared type of the function.

`Undeclared struct/union tag:` *tag* *Error*

You have written `struct` *tag* or `union` *tag* without declaring or having declared *tag*'s elements.

`Variable` *name* `appeared in the parameter declarations, but is not one of the parameters` *list*

You have declared *name* in the parameter declarations at the head of a function, but *name* is not in the parameter list of the function.

`Wrong argument type ...` *Error*

You have supplied a value of incorrect or nonsensical type to one of the ZETA-C primitives; for example, perhaps you attempted to add a structure or multiply a pointer.

2.3.3 Runtime debugging

[[This subsection will list some runtime errors whose meaning in the context of a C program may not be clear, and suggest places to look to find the real problem.]]

[[It will also talk about the use of the Lisp debugger for debugging C, or the C object debugger if that gets written.]]

2.4 Program construction

You have several options in building programs using ZETA-C. The simplest approach is to keep files of C code, compile and load them by hand, and call the functions in them by hand from a Lisp listener or from other Lisp or C programs. Or, if you want your C programs to be callable from Lisp in roughly the same way that UNIX programs are callable from the shell, you can use `zeta-c:create-c-program` to create the appropriate top level. You can use the Lisp `defsystem` facility to partially automate the maintenance of these ”stand-alone” programs, or to integrate C code into Lisp programs (ZETA-C itself takes this approach – its parser was built using the UNIX **lex** and **yacc** utilities, which produce files of C code).

You should bear in mind, while preparing C programs for execution on the Lisp Machine, the way ZETA-C depends on the Lisp package system (see p. 11). You will need to select a name for your C program package, and add a file attribute list at the beginning of each file specifying this package and C mode (see p. 29). If your program redefines any of the functions in the ZETA-C standard library (see chapter 5), you should arrange to shadow their names in your C program package; one way to do this is to include at the beginning of the main header (`.h`) file for the program lines like

```
#lisp
  (gl:shadow '(|putc| |getc|))
#endlisp
```

Alternatively, if you are using `defsystem`, you can add the shadowing declaration to the package definition in the system definition file.

Here are the tools available for compiling files and building programs:

`zeta-c:c-compile-file` *infile &optional outfile* *Function*

> C-compiles the specified file, producing a standard `.BIN` file. All declarations and #definitions made by the file compiled or by any of the files it includes remain in effect after the compilation is complete, and are accessible via the C listener.

`zeta-c:create-c-program` *name* *Function*

> Sets up a function called *name* – in package `GLOBAL:` if possible, else in the *name*`:` package – which is to be used as the top-level invocation function for a C program. The created function takes one required argument, which is the current directory in which the program is to be run, followed by an &rest argument of strings, which are the ”command line arguments”. It initializes externals, binds `stdin` etc. to the appropriate streams, packages up the ”command line arguments” into `argc` and `argv` in the standard UNIX way, and calls *name*`:main`. For instance,
>
> ```
> (zeta-c:create-c-program 'cweed)
> ```
>
> sets up a function `cweed` which might then be called thus:
>
> ```
> (cweed "oz:<x.gyro” ”-afd” ”oz:<x.gyro>foo.text”)>
> ```

The following transformations are provided for users of `defsystem`. (See the section on `defsystem` in your Lisp Machine documentation.)

`:c-compile` *input dependencies condition* — *Defsystem transformation*

Calls `cc-file` to compile the indicated files, whose names must have canonical type `:c`. *condition* defaults to `si:file-newer-than-file-p`.

`:c-compile-load` *c-dep l-dep c-cond l-cond* — *Defsystem transformation*

Equivalent to `(:fasload (:c-compile` *c-dep c-cond*`)` *l-dep l-cond*`)`.

2.5 C Listener

ZETA-C provides a ”C listener”, a counterpart to the Lisp Listener, for the manual entry and execution of C expressions and declarations. You can get to an independent C listener (that runs in its own process) by typing `[Select] {`, by selecting the C Listener item in the Programs column of the System Menu, or by the Create, Split Screen, or Edit Screen menus in the Windows column of the Lisp Listener. Or, you can get a C listener that runs in your Zmacs process by typing `[Suspend]` to a Zmacs window whose buffer is in C mode (hit `[Suspend]` again to get a Lisp listener).

(The C listener is on `[Select] {` rather than `[Select] C` because the latter is standardly used for the Converse program, and `{` is a character suggestive of the C language. However, your local system maintainer may have put Converse on some other key, making `C` available for the C listener. Type `[Select] [Help]` to be sure.)

When you create an independent C listener, the first thing it does is ask you what package you would like to work in. If this package already exists, it must be a C program package (see p. 11); if it is not, the C listener will request another package name. If the package does not exist, the C listener offers to create it. Next, the C listener will ask you if you wish the external and static variables in this package to be freshly initialized; you may want this if you are debugging a program and want to start its execution over from the beginning. Then the listener will request a default directory for file operations to take place in. Finally, the listener will issue a prompt (usually `C:`, but see below) and await your typein. If at any time you wish to change the package or directory, or to reinitialize externals and statics, type `meta-[Abort]`, and this series of questions will be repeated.

At this point – whether you’ve created an independent C listener or one within Zmacs – you may type any of the following: a statement, a declaration, a function definition, or even a preprocessor directive. A statement may consist merely of an expression followed by a semicolon; the expression will be evaluated and its value printed. (Thus a C listener makes an excellent infix-notation desk calculator!) Just as with a Lisp listener, parsing happens on the fly, and the statement, declaration, or definition will be processed as soon as it is syntactically complete (as soon as you type the closing semicolon or right brace); a preprocessor directive will require a `[Return]`. Also, the input editor is fully available for editing input, along with the history feature for reentering previous inputs.

Here are some examples; user typein is in `bold`.

```
C package to work in? ctest
Create C package CTEST? (Y or N) Yes.
Initialize externals and statics in package CTEST? (Y or N) No.
Working directory for file I/O? astarte:gyro;
C: int foo; foo is declared
C: foo; foo's value is requested
(int) 0 it's zero, of type int
C: char *bar = "A string"; bar is declared and initialized
C: bar; bar's value is requested
(char *) "->A string" the arrow indicates a pointer to that character of the string
C: for (foo=0; foo<5; ++foo) printf("%s\n", bar + foo);
A string a "for" loop with a function call
 string does just what you'd expect
string
tring
ring
C: #include <stdio.h> we want to do some fancier I/O
C: viewfile (name) we type in a simple function
     char *name; (actually, this is probably more complex
{ a function than you would actually use the
     FILE *fd; C listener for, but it's a good example)
     int c;
     if ((fd = fopen(name, "r")) == NULL) {
          printf("Can't open '%s'.\n", name);
          exit(0);
     }
     while ((c == getc(fd) != EOF) putchar(c);
     fclose(fd);
} we type the closing brace, and the function gets compiled
C: viewfile("hello.c"); and we run it
/* -*- Mode: C; Package: (hello C) -*- */ printing the file we typed in long ago
main() {
     printf("\nHello, world!\n");
}
C: #include "hello.c" we load that file
C: HELLO$main(); and call its function (note the package specification)
Hello, world!
C: etc....
```

Currently (Release 1.1), errors in execution of C expressions will still invoke the standard Lisp Debugger. Also, typing `[Suspend]` will give you a Lisp read-eval-print loop.

`zeta-c:*c-listener-prompt*` *Variable*

A string containing the prompt issued by a C listener when it is ready for an expression. The default value is `"C:"`.

2.6 Function Type Checking

Along with incremental compilation, ZETA-C provides an ”incremental **lint**” facility that checks, at *load* time, that the types of function arguments match the types of the corresponding formal parameters, even when the call and the definition are in different files. ”Load” time, in this case, is either when the `.BIN` file is loaded, or, for code being compiled incrementally, essentially the same as compile time. That is to say, when a function is compiled incrementally or loaded from a file, all known calls to it are checked as well as all the calls it makes to other known functions. The warnings generated in case of mismatches are placed in the compiler warnings database, and can be reviewed via the appropriate Zmacs commands (see the section on compiler warnings in your Zmacs manual).

Similar checking is performed for the type of the return value of a function.

+	Addition (one argument may be a pointer).
-	Subtraction (one or both arguments may be pointers).
*	With one argument, pointer dereferencing; with two, multiplication.
/ (//)[3]	Division.
%	Remainder.
<<	Shift left.
>>	Shift right.
&	Bitwise AND.
| (/|)	Bitwise OR.
^	Bitwise XOR.
~	Bitwise NOT (one argument).
==	Equality comparison.
!=	Inequality comparison.
<	Less-than comparison.
>	Greater-than comparison.
<=	Less-than-or-equal comparison.
>=	Greater-than-or-equal comparison.
!	Logical NOT.
&&	Logical AND.
|| (/|/|)	Logical OR.
=	Assignment.
++x	Preincrement.
x++	Postincrement.
--x	Predecrement.
x--	Postdecrement.
+=	Add and assign (first argument may be a pointer).
-=	Subtract and assign (first argument may be a pointer).
*=	Multiply and assign.
/= (//=)	Divide and assign.
%=	Take remainder and assign.
<<=	Shift left and assign.
>>=	Shift right and assign.
&=	Bitwise AND and assign.
|= (/|=)	Bitwise OR and assign.
^=	Bitwise XOR and assign.
[]	Array reference.
. (/.)	Structure element reference.
->	Indirect structure element reference.

Figure 2.1: ZETA-C primitives — expressions.

`progn+`	The comma operator (which acts like the Lisp progn).
`?: (?/:)`	The expression conditional.
`if`	The statement conditional.
`block+`	Encloses statements in a block (see text).
`goto`	Go to the specified label.
`label+`	The first argument is a label; the second is a statement.
`while`	(while *looptest body*): iterate.
`do`	(do *body looptest*): iterate, doing *body* at least once.
`for`	(for *init looptest increment body*): iterate with syntactic sugar.
`break`	Skip to the end of this while, do, for, or switch.
`continue`	Skip to the next iteration of this while, do, or for.
`return`	Return from this function (argument, if any, is value to return).
`switch`	(switch *exp body*): go to the case in *body* matching *exp*.
`case`	(case *value*): meaningful only in switch bodies.
`cast+`	(cast+ *type expression*): cast the type of *expression* to *type*.
`sizeof`	The size of a type or instance.
`#lisp (/#lisp)`	Encloses a group of Lisp forms introduced with #lisp (see p. 35).
`defunc+`	Function definition (see text for explanation).
`decl+`	External declaration.
`fcn+`	Indicates a function declaration.
`list+`	Encloses a list (written with braces in the source) of initializer values.

Figure 2.2: ZETA-C primitives — statements.

:char	A character.
:signed-char	A signed character.
:short	A short (16-bit signed).
:unsigned-short	An unsigned short.
:int	Used for an int or long: an arbitrary-precision signed integer.
:unsigned	Used for an unsigned int or unsigned long: an arbitrary-precision unsigned integer.
:float	A single-precision floating point number.
:double	A double-precision floating-point number.
:zero	The constant 0.
:void	The void type.
:lispval	A lisp value (can be assigned or passed to and from functions, but nothing else).
:boolean	The type of an expression evaluated for control, not value.
(:pointer *type*)	A pointer to type *type*.
(:pointer *type* :null)	A null pointer to type *type*.
(:array *type length*)	An array of *length* elements, each of type *type*. *length* can be NIL, meaning the length is not known.
(:function *type*)	A function returning type *type*.
(:struct . *tag-or-elts*)	A structure. If *tag-or-elts* is a symbol, it's a tag; if a list, it's an alist associating element names with types and accessing information.
(:union . *tag-or-elts*)	A union; like :struct.
(:enum . *tag-or-elts*)	An enumeration type; like :struct.

Figure 2.3: ZETA-C type descriptions.

3 Editing C with Zmacs

ZETA-C provides some Zmacs extensions to simplify the editing of C programs. These extensions fall into several categories: cursor movement; indentation support and other convenience commands; sectionization; and compilation. In order to activate these, you must specify C mode in the file attribute list ("-*- line") of each C source file; for instance

```
/* -*- Mode: C; Package: (name C) -*- */
```

Note that the attribute list must be enclosed in comment delimiters (`/*` ... `*/`).

3.1 Cursor movement

`control-meta-A` *Beginning of C Function or Declaration* *Key*

Moves to the beginning of the current C function or declaration. If the cursor is already at the beginning of one, moves to the beginning of the previous one.

`control-meta-E` *End of C Function or Declaration* *Key*

Moves to the end of the current C function or declaration. If the cursor is already at the end of one, moves to the end of the next one.

`control-meta-H` *Mark C Function or Declaration* *Key*

Puts **point** at the beginning, and **mark** at the end, of the current C function or declaration. Given a positive or negative numeric argument, marks that many objects forward or backward, respectively.

3.2 Indentation and misc.

C programs are indented in a variety of different styles. ZETA-C's indentation support assumes that braces are written only at the ends of lines; within this constraint, the following variations are supported:

`zwei:*C-block-indentation*` *Variable*

The distance, in spaces, to indent nested C blocks. If `NIL` (the default), the distance defaults to the current tab width.

`zwei:*C-indent-}-as-outside*` *Variable*

If `T` (the default), a right brace on a line by itself is lined up with the statements *outside* the block it closes:

```
        foo();
    }
    bar();
```

If `NIL`, the brace is lined up with the statements *inside* the block:

```
        foo();
        }
    bar();
```

`Tab` *Indent for C* *Key*

Indents the current line in the current C style. With a numeric argument, indents that many lines starting at the current line.

`control-meta-Q` *Indent Region for C* *Key*

Indents all lines in the region in the current C style.

`control-;` *Indent for Comment* *Key*

Moves to or creates a comment. Finds the start of any existing comments, or creates one at the end of the current line. With a numeric argument, re-aligns existing comments for that many lines, but does not create any. `*COMMENT-COLUMN*` is the minimum column (in pixels) for aligning comments. (This is, in fact, the standard Indent for Comment function; C mode just sets up the string variables specifying the comment-start and -end strings.)

`meta-;` *End Comment* *Key*

Closes the comment on this line (if any) and moves to the next line.

`meta-P` *Up Comment Line* *Key*

Moves to or creates a comment on the previous line; but first, if the current line contains a null comment, it is deleted.

`meta-N` *Down Comment Line* *Key*

Moves to or creates a comment on the next line; but first, if the current line contains a null comment, it is deleted.

`control-meta-R` *Reposition Window for C* *Key*

Tries to get all of the current C function or declaration on the screen. Repeated invocations will scroll the comments immediately above the function or declaration on and off the screen, alternately.

3.3 Sectionization

The term ”sectionization” is Zmacs jargon for finding the boundaries between the top-level declarations and function definitions that appear in a source file. The sectionizer is also responsible for finding a suitable name for each section; the name it chooses will be used by `meta-.` and other section-related commands. Since the indentation and layout conventions for C are considerably less standardized than they are for Lisp, and since C syntax is more complex, it is a lot harder to tell where sections begin and end in C programs. In order that the sectionizer should still work reasonably on syntactically incorrect code, it does not even attempt to parse a file completely. Instead, it uses a simple heuristic to guess where things must begin and end.[1] Basically, whenever it sees a word – as opposed to punctuation – starting in column 0, it considers that line to start a section. If the section contains a function definition, the section takes its name from the function, just as you would expect; if the section contains a declaration, the section takes its name from the first identifier declared (in the case of a structure or union declaration, this is the structure or union tag if one is given, otherwise the name of the first element).

To repeat: any word in column 0 is taken to start a section. It is worth mentioning some ramifications of our use of this heuristic.

- Each line of a comment must start with whitespace or some non-alphabetic character.
- Each line of a function but the first must be indented, unless it starts with punctuation such as ”{”; we have seen C indentation styles wherein the declarations of functional parameters were not indented, thus:

  ```
  foo(a)
  int a;
   { ...
  ```

  The parameter declaration must be indented:

  ```
  foo(a)
       int a;
  { ...
  ```
- Likewise, statement labels must be indented.
- If an external variable declaration is to be recognized as starting a section, it must begin in column 0. Alternatively, if you wish to group several consecutive declarations into one section (so all of them can be recompiled if any has changed), indent all but the first.

[1] The Lisp sectionizer, as you may know, takes the same approach – it just looks for left parentheses in column 0. The heuristic for C is more complex but in the same spirit.

- Preprocessor directives never start sections. Furthermore, each section extends to the beginning of the next section, so preprocessor directives are considered to belong with the *preceding* declaration.

EXCEPTION: we specifically provide for the type of a function being defined to go on a line by itself, thus:

```
struct env *
parent(e)
     struct env *e;
{ ...
```

If you are uncertain about exactly how some code was sectionized, you can do `meta-X List Definitions` to see if any of the sections are spurious or missing. If you are wondering why anyone would care, read on to the next section!

3.4 Compilation

ZETA-C provides commands for compilation of single sections (see above), specified regions, or entire buffers.

`hyper-control-C (or control-shift-C)` *Compile Region* *Key*

If there is a region, it is C-compiled; otherwise, the current section (function definition or external declaration) is C-compiled.

`hyper-meta-C (or meta-shift-C)` *Compile Buffer Changed Definitions* *Key*

C-compiles any sections in the current buffer that have been edited since they were last compiled. With a numeric argument, asks whether to compile each changed section.

`meta-X Compile Buffer` *Extended Command*

C-compiles the entire current buffer. With a numeric argument, compiles the rest of the buffer (from **point** to the end).

ZETA-C provides an ”incremental **lint**” facility to keep track of function argument and return value types and verify consistency across files. Incremental compilation is one way to invoke this facility. See p. 25 for more information.

4 The ZETA-C Dialect

Our goal has been and continues to be to make ZETA-C highly compatible with standard C implementations, most notably the Berkeley 4.2bsd and Bell UNIX(R) System V compilers, as well as the specification given in *C: A Reference Manual* by Samuel P. Harbison and Guy L. Steele Jr. (Prentice-Hall, 1984). (We also plan to meet the ANSI C standard, as soon as that standard is released to the public.) This chapter documents in detail remaining incompatibilities, known bugs, and implementation-defined behavior. We start with a couple of sections describing extensions ZETA-C makes to standard C.

4.1 Variable numbers of arguments

In traditional C implementations, there is no checking at either compile or run time that the number of arguments passed to a function is the same as the number of parameters the function declares. In fact, most existing C programs are written assuming that it is possible for functions to take a variable number of arguments; either the last one or more arguments will simply be omitted from a call, or functions (notably `printf`) will accept what amounts to a variable-length list of arguments.

The Lisp Machine, on the other hand, provides run-time checking (and compile-time checking too, for calls to known functions) of the number of arguments passed to a function, and requires explicit declarations (`&optional` and `&rest` keywords in Lisp) when the function is to take a variable number of arguments. ZETA-C likewise requires explicit declarations of optional and "rest" arguments, by means of the storage class specifiers `optarg` and `restarg` (applicable only to parameters). `optarg` indicates that the argument (actual parameter) may be omitted; all subsequent arguments may also be omitted, even if the corresponding parameters were not specifically declared `optargs`. `restarg`, on the other hand, indicates that an arbitrary number of arguments may be passed corresponding to this parameter; it is ignored except on the last parameter in the list. For example:

```
foo(a, b, c)
     int a;
     optarg int b;
     char *c;
{
  [... some stuff ...]
}

/* These calls to foo will all work */
     foo(1);
     foo(2, 476);
     foo(3, 476, "Hello");
/* These will get a wrong-number-of-arguments error */
     foo();
     foo(44, 55, "Hello", "Goodbye");

send_multiple(dest, messages)
     frob *dest;
     restarg char *messages;
{
     /* We send each message to the destination */
     /* The end of the argument list is indicated by NULL */
     for (i = 0; (&messages)[i]; ++i)
           send(dest, (&messages)[i]);
}
/* An example call to send_multiple */
     send_multiple(turtle, "Forward 4", "Right 45",
                   "Forward 96", NULL);
```

The technique shown in `send_multiple` for accessing each of the supplied arguments is not strictly portable, since it assumes that successive arguments are stored at successively higher addresses, but all the compilers we are familiar with work this way, and this trick has become quite common in C. Note that it is not the parameter itself, but rather its *address*, that is the base of the array of arguments. Note also that you must provide your own convention for determining the end of the argument list: under ZETA-C, an array-bounds error will be signalled if you attempt to access a nonexistent element of the rest-argument array, but under traditional compilers, you will just get garbage.

RESTRICTION: *do not* modify any of the rest-arguments if these arguments are structures. ZETA-C does not make copies of these arguments, as the semantics of C requires. (C is defined to be call-by-value, and ZETA-C conforms to this definition in all but this particular case, where its behavior is call-by-reference.)

To compile your program with a traditional compiler, include the lines

```
#define optarg
#define restarg
```

(other changes may also be necessary, as there is no completely portable way to write functions that accept variable numbers of arguments).

4.2 Lisp objects in C programs

ZETA-C provides a type, `lispval`, which is intended for variables holding miscellaneous Lisp objects (e.g., lists, flavor instances). The operations defined on `lispval`s are: assign, pass as function argument, return as function value, compare for equality (the comparison is done with `eql`), or test as conditional predicate with `if`, `?:`, `!`, `&&`, or `||` (for this purpose, `nil` is treated as **false** and non-`nil` as **true**, just as in Lisp).

4.3 Inclusion of Lisp code

ZETA-C supports two forms of inclusion of Lisp code. One permits entry of a Lisp form as an *expression* in C, the other includes one or more Lisp forms as C *statements.* The expression form (useful primarily in the C listener, but also available in C source files) is `@`*form*. For instance, the demo program supplied with ZETA-C, `zeta-c:source;turtle.c`, contains a line like:

```
window = @tv:(make-window 'window :edges-from :mouse);
```

Note that after the closing right parenthesis of the Lisp form is encountered, the syntax reverts to C, so that a semicolon is required to complete the statement. An expression entered with '`@`' has type `lispval` (see above), but can of course be cast to any type. If it is cast to a pointer type, the expression is expected to return two values, which will become the array part and index part, respectively, of the pointer (see the section on Pointers, page 13).

Much as some C compilers provide the `#asm` construct for literal inclusion of assembler code, ZETA-C provides for the inclusion of Lisp forms as C statements via `#lisp` ... `#endlisp`. The lines between these directives are read as Lisp forms and included in the source, as handed to the Lisp compiler, without further processing. If this construct is used inside a function definition, all arguments, locals, and statics of the function can be referenced by name (see the section on Pointers, page 13, on the naming of pointer variables). A `restarg` parameter will be bound to an array of the arguments. Note that the Lisp forms will be read in the same package as the C code; if you want to access another package, use an explicit prefix. Also note that the vertical bar convention must be used to reference variables whose names contain lowercase letters (see the next section).

Comments between `#lisp` and `#endlisp` must be introduced with semicolons!

Here is an example:

```
print_num (n)
     int n;
{
     /* For no good reason, we use FORMAT instead
      * of PRINTF to print the number. */
#lisp
     (gl:format gl:t "The number is: ~D" |n|)         ; Note "gl:"s
#endlisp
}
```

4.4 ZETA-C Identifiers

Unlike Lisp, C distinguishes upper and lower case in identifiers. So C variable and function names are not automatically converted to upper case. This means that, when referencing a C variable or function from Lisp, one must use Lisp's vertical bar notation if the name contains lowercase letters; for instance: `|main|`, or, if the reference is from another package, "`hello:|main|`" (*not* "`|hello:main|`"). Conversely, when referencing Lisp functions from C, one must write in uppercase, *e.g.*, "`TV$BEEP`".

ZETA-C uses the '`$`' character (otherwise unused in C) for two purposes. One is to delimit package prefixes, just as '`:`' does in Lisp. So, for instance, the C statement "`TV$BEEP();`", equivalent to the Lisp "`(tv:beep)`", would beep the console beeper (and/or flash the screen). The other use of '`$`' is to build internal names (called static alternate names) of variables and functions declared `static` (see the next section). Here's how ZETA-C distinguishes these two uses: if what appears before the first '`$`' in a name is a known package name, then it is taken to be a package prefix; otherwise, it is simply considered to be part of the name. Package prefixes need not be written in upper case, though we recommend this as a convention.

ZETA-C, like the Lisp in which it is implemented, supports arbitrarily long identifiers (well, to be honest, the parser, being written in C, imposes a 4096 character limit), and all characters are significant.

4.5 Static variables and functions

One of the uses of the '`$`' character (see the previous section) is to build internal names (called static alternate names) of variables and functions declared `static`. This works in one of two ways:

- If a variable or function is declared `static` at top level within a file – *i.e.*, the declaration does not appear within a function definition – then its static alternate name is the name part of the file's pathname, followed by a '`$`', followed by the name of the variable or function as written. For instance, if a file "`foo.c`" contains a declaration "`static int bar;`", the alternate name for "`bar`" would be "`foo$bar`".
- If a variable is declared `static` within a function definition, then the static alternate name is the name of the function, '`$`', the name of the variable, '`$`', a decimal number. The decimal number is used to distinguish multiple static variables of the same name in the same function; it is almost always "`1`". So, the static alternate name of a variable declared "`static int quux;`" within a function "`zot`", assuming there's only one such variable "`quux`", would be "`zot$quux$1`".

These two uses can compose, so if there's a variable declared "`static int quux;`" inside a function declared "`static int zot() ...`" in a file "`foo.c`", its static alternate name would be "`foo$zot$quux$1`".

A static variable or function can be referenced by its short name in the file or function in which it is declared, *after* it has been declared static. This means that, in the case of file statics, the variable or function must be declared static textually *before* any reference to it appears. (This is an incompatibility with standard C). However, in the case of a function, this does not mean that the entire function *definition* need appear near the beginning of the file; only that a *declaration* of it as static be near the beginning. So, for instance, one can say "`static int zot();`" near the beginning of the file, refer to `zot` throughout, and at the end define it: "`static int zot() { ...`".

A static variable or function can be referenced by its alternate name from anywhere; specifically, from the C listener. This is true even for static variables within functions (which was the purpose of all this, by the way: to make these variables accessible from the C listener).

4.6 Dialect reference

This section is written specifically as a companion to *C: A Reference Manual* by Samuel P. Harbison and Guy L. Steele Jr. (Prentice-Hall, 1984) (abbreviated below ”*C:ARM*”). For every section in that manual from which ZETA-C differs, or where the behavior of C is specified as implementation-defined, we have included a correspondingly-numbered section here. We assume the reader has copy of *C:ARM* to refer to.

4.6.1 1 Introduction to C

4.6.2 1.2 AN OVERVIEW OF C PROGRAMMING

See the Overview and Examples chapter on p. 9 of this manual.

4.6.3 2 Lexical Elements

4.6.4 2.1 THE SOURCE CHARACTER SET

ZETA-C uses the characters ‘`$`’ and ‘`@`’; ‘`$`’ as the package prefix or static context delimiter, and ‘`@`’ to introduce a Lisp form (see the previous sections for descriptions of both of these).

4.6.5 2.1.1 Whitespace and Line Termination

There is no line length limit in ZETA-C.

4.6.6 2.1.2 Character Encodings

The Lisp Machine’s character set is unusual in that the first 128 characters (0 through 177 octal) are all printing characters; the control characters are octal 200 through 237. See your Lisp manual for more details. As long as you use the predefined escape sequences `\r`, `\n`, `\b`, `\t`, `\f`, and `\v`, this will not make any difference to your programs, unless you are using the high-order bits of characters for some special purpose. The sequences `\r` and `\n` both name the `[Return]` character, octal 215; `\b` is `[Overstrike]`, octal 208; `\t` is `[Tab]`, octal 209; and `\f` and `\v` are both `[Page]`, octal 214.

4.6.7 2.2 COMMENTS

Comments are normally treated as not nestable. However, if you put in the file attribute list (”-*- line”) of a C source file the attribute ”`Comments-nest: T`”, then within that file (or any files it includes), comments will be treated as nestable.

4.6.8 2.4 OPERATORS AND SEPARATORS

Compound assignment operators must be written as single tokens in ZETA-C; *e.g.*, ”`+=`”, not ”`+ =`”.

4.6.9 2.5 IDENTIFIERS

ZETA-C places a 4096 character limit (*i.e.*, effectively infinite) on identifier names; all characters are significant. ‘`$`’ is used as the package prefix delimiter (like ‘`:`’ in Lisp); see page 35 above.

4.6.10 2.6 RESERVED WORDS

ZETA-C defines the additional reserved words `optarg` and `restarg` for declaring variable-length function argument lists (see p. 33); `signed` for declaring signed character variables (see p. 12); `lispval` for declaring variables to contain arbitrary Lisp objects (see p. 34); and `packed_struct` for declaring structures to be implemented with a ”packed” representation (see p. 14).

4.6.11 2.7 CONSTANTS

4.6.12 2.7.1 Integer Constants

ZETA-C recognizes both the suffix ’`l`’ or ’`L`’, indicating a `long` constant, and the suffix ’`u`’ or ’`U`’, indicating an `unsigned` constant; these may be used together. Since ZETA-C supports arbitrary-precision integers, the value written will be the value used (no ”surprises” are possible like the ones listed in this section of *C:ARM*).

4.6.13 2.7.2 Floating-point Constants

Unless specially marked, a floating-point constant has type `double`. The suffix ’`f`’ or ’`F`’ indicates that the constant is to be of type `float`. The suffix ’`l`’ or ’`L`’ is ignored (the type `long double` is the same as `double`).

4.6.14 2.7.3 Character Constants

Character constants have type `char`, rather than `int` as specified in this section of *C:ARM*. (Since `char` is an unsigned type, there is only one situation in which this will make a difference. Consider the comparison `('A' -1)`>. One might expect this comparison to yield **false**, but since `char` is unsigned, it will instead yield **true**, as in the example on p. 90 of *C:ARM*. Note that `(c -1)`> where `c` is a `char` variable will also yield **true**, so by treating character *constants* as `char` rather than `int`, ZETA-C brings their behavior into line with that of `char` *variables*.)

Multicharacter constants are accepted, and treated as they are by the 4.2bsd compiler for the VAX: the rightmost character in the constant goes in the low-order byte of the word (so a `char *` pointer to this word would see it *first*; this is ”backwards” with respect to the way strings are stored). 2-character constants have type `unsigned short`; 3- and 4-character constants have type `unsigned`; longer constants have type `unsigned long` (but note that their use is *completely* nonportable).

4.6.15 2.7.4 String Constants

Identically written string constants within a single function will be represented by the same block of storage. (This optimization is performed by the Lisp compiler, and so we cannot disable it.)

4.6.16 2.7.5 Escape Characters

Hexadecimal escape codes are also supported, in the form `\x`*dd*, where *d* is a hexadecimal digit (`0-9`, `a-f`, `A-F`).

4.6.17 3 The C Preprocessor

4.6.18 3.2 PREPROCESSOR LEXICAL CONVENTIONS

The # must be the first character on the line; there may be any amount of whitespace between the # and the preprocessor command. It is not an error if non-whitespace appears on a line after a command that takes no arguments; such text is ignored.

4.6.19 3.3.2 Defining Macros with Parameters

In the *invocation* of a macro with parameters, ZETA-C will allow one space between the name of the macro and the left parenthesis that begins the argument list, but will issue a portability warning whenever such a space is encountered.

ZETA-C does not recognize formal parameter names within string and character constants.

4.6.20 3.3.3 Rescanning of Macro Expressions

Recursion in macro expansions is not detected, and causes ZETA-C to enter an infinite loop, which can be stopped by typing `control-[Abort]`. (But the example on the bottom of p.35 doesn’t loop! It gets a syntax error as soon as two copies of the argument have been generated. This happens because rescanning is interleaved with parsing.)

4.6.21 3.3.4 Predefined Macros

ZETA-C’s version of `stdio.h` contains the line

```
#define ZETA_C
```

4.6.22 3.3.5 Undefining and Redefining Macros

Macros may be redefined freely; the old definition is simply discarded. No warning is issued. ZETA-C does not keep a stack of definitions; `#undef` will ensure that no definition remains.

4.6.23 3.3.6 Some Pitfalls to Avoid

Macro expansions are actually rescanned as character sequences, but since all sequences of whitespace and/or comments turn into single spaces, this does not generally make any difference. The exception is that the null comment `/**/` is explicitly recognized as a ”token concatenation” operator. Some examples (cf. p. 39 of *C:ARM*):

```
#define INC ++
#define TAB internal_table
#define INCTAB table_of_increments
#define CONC(x,y) x/**/y
#define CONC2(x,y) x /**/ y
#define DONTCONC(x,y) x/* */y
CONC(INC,TAB)  =>INCTAB  =>table_of_increments
CONC2(INC,TAB)  =>INCTAB  =>table_of_increments
DONTCONC(INC,TAB)  =>INC TAB  =>++ internal_table
```

Note that the null comment functions as a concatenation operator regardless of whether there is whitespace around it in the macro definition, but if there are any characters in the comment at all, it will not cause concatenation.

Macro text is checked for balanced single and double quotes when it is first encountered, so the example on the top of p. 40 of Harbison & Steele will cause an error.

4.6.24 3.5 CONDITIONAL COMPILATION

4.6.25 3.5.5 The `defined` Operator

The `defined` operator is not yet implemented.

4.6.26 3.6 EXPLICIT LINE NUMBERING

`#line` directives are ignored. ‘`#`’ on a line by itself is ignored. The use of ‘`#`’ as a synonym for ‘`#line`’ is not supported.

4.6.27 4 Declarations

4.6.28 4.2 TERMINOLOGY

4.6.29 4.2.1 Scope

The scope of a statement label is the innermost enclosing block that contains declarations of automatic variables, rather than the entire function as the standard specifies. The Lisp compiler enforces this very reasonable restriction (jumping into a block that contains automatic variables is likely to cause unpredictable results anyway).

4.6.30 4.2.4 Overloading of Names

Labels are not in the same name space as variables; but see section 4.2.1 above. ZETA-C provides a separate component namespace for each structure and union type (the ”modern” interpretation). Structure and union tags are in the same name space, but enumeration tags are in a separate one.

4.6.31 4.2.8 Initial Values

Like the UNIX compilers, ZETA-C initializes objects of static extent to zeros. New code should, however, not depend on this. Since blocks with automatic variables cannot be entered abnormally in ZETA-C (see section 4.2.1 above), there is no danger that the variables will not be properly initialized.

4.6.32 4.2.9 External Names

ZETA-C will issue an ”Undeclared variable” error for this example.

4.6.33 4.3 STORAGE CLASS SPECIFIERS

ZETA-C defines the additional keywords `optarg` and `restarg`, which behave syntactically like storage-class specifiers and which apply only to parameter declarations. (See page 33.)

4.6.34 4.6 INITIALIZERS

Many C compilers (including the UNIX Portable C Compiler, PCC) make some attempt to ”do what you mean” when given an initializer that does not quite conform to the rules. ZETA-C, on the other hand, imposes a stricter interpretation. For instance

```
char magic_header[] = { "\037\235" };
```

is acceptable to PCC, but not to ZETA-C, which insists on either

```
char magic_header[] = {’\037’, ’\235’, ’\0’};
```
or
```
char magic_header[] = "\037\235";
```

As Guy Steele himself says (personal communication):

It is a very curious thing that a construct can be ILLEGAL and nevertheless highly portable! That comes from many implementations being based on the same widely-distributed but incorrect source code.

The code that processes braces in PCC is rather peculiar and *ad hoc*. It accepts what K&R specifies, but also accepts many other cases of missing or extraneous braces in a rather idiosyncratic pattern. In my opinion this particular use of braces is logically extraneous and should be avoided for maximum portability. The braces ought to be interpreted as being a list of items to be used to fill in the array `magic_header`, which ought to be of length 1 because that is the length of the brace-list. That would be the correct interpretation of

```
char *magic_header[] = { "\037\235" };
```

However, without the "*" one gets the situation of trying to initialize a `char` to a pointer value, and PCC goes to some trouble to figure out what you really might have meant.

ZETA-C allows any expression to be used in an initializer, even for variables of static extent.

4.6.35 4.6.8 Other Types

Objects of type `lispval` are also initializable. Of course, an initializer expression for such an object must be either a variable of type `lispval`, a Lisp expression introduced with the `@` operator (page 35), or any expression explicitly cast to type `lispval`.

4.6.36 4.8 EXTERNAL NAMES

The approach ZETA-C takes to the resolution of external names is the "omitted-`extern`" solution described here.

4.6.37 5 Types

4.6.38 5.1 STORAGE UNITS

See the discussion in the section on Data Structures that begins on page 12.

4.6.39 5.2 INTEGER TYPES

`int` and `unsigned` are all potentially infinite precision (bignums). See the discussion on Numbers on page 12.

4.6.40 5.2.3 CHARACTER TYPE

The type `char` is unsigned. The type `signed char` is available if requested explicitly; signed chars are slower, however (see page 12). Note that some of the characters in the LispM's character set are between 128 and 255 decimal, so they require 8 bits to represent, and will appear negative if assigned to a `signed char` variable.

4.6.41 5.4 POINTER TYPES

4.6.42 5.4.2 Some problems with pointers

See section 6.7.1 below, concerning alignment behavior.

4.6.43 5.6 ENUMERATION TYPES

Enumeration tags are actually in a separate overloading class from structure and union tags.

4.6.44 5.6.1 Detailed Semantics

ZETA-C uses the integer model for enumerations (see *C:ARM*, p. 103). That is, all enumeration types are treated as synonyms for the type `int`; enumeration constants and variables may be mixed freely with integers in expressions.

4.6.45 5.7 STRUCTURE TYPES

4.6.46 5.7.1 Operations on Structures

Treating structures (and unions) as firstclass objects – assigning them, passing them as parameters, and returning them as values – are all supported under ZETA-C. However, since some compilers do not support these operations, we have provided an option to ”turn them off”, causing ZETA-C to signal an error if one of these things is attempted. Just put the line

```
#define ZETA_C_NO_FIRSTCLASS_STRUCTURES
```

at the beginning of your program. Firstclass structures are supported by most of the UNIX compilers, but by fewer of the non-UNIX compilers, so for maximum portability they should not be used.

4.6.47 5.7.2 Components

ZETA-C

follows the ”modern” rule described here, that each structure type defines a separate namespace for its components. That is, component names may be reused freely in different structure types (though of course they must be unique within a single structure type).

4.6.48 5.7.4 Bit Fields

Bit fields are allocated from right to left. They may be signed or unsigned.

4.6.49 5.11 TYPEDEF NAMES

4.6.50 5.11.1 Redefining Typedef Names

ZETA-C does not correctly handle the example given in this section. (This is pretty questionable programming practice, anyway.)

4.6.51 6 Type Conversions

4.6.52 6.3 CONVERSIONS TO INTEGER TYPES

Traditional architectures provide different instructions for operating on different kinds of numbers; it is up to the compiler to keep track of what kind of number is involved in any particular operation, and to generate type conversion instructions as necessary. Lisp Machines, on the other hand, support generic arithmetic instructions, and do type conversion automatically at runtime. This can cause unexpected behavior in one case: if a variable is declared `int`, say, but is given a floating-point value by way of an escape to Lisp (see page 35), computations performed using that value will yield floating-point results. These may be stored in other `int` variables, and so forth, propagating floating-point-ness throughout large parts of the program's data structures. So be careful at the interface between Lisp and C that numeric values match their declared types. (As long as one stays entirely within C, the compiler will warn about mismatches.)

4.6.53 6.3.2 From Floating-point Types

When floating-point numbers are converted to integers, negative numbers are truncated downward (away from 0).

4.6.54 6.3.4 From Pointer Types

See section 6.7.2 below.

4.6.55 6.7 CONVERSIONS TO POINTER TYPES

4.6.56 6.7.1 From Pointer Types

A ZETA-C pointer will be in one of three different representations, called ”scales”. Recall (page 13) that a pointer is a pair of an array and an index. The index will be that used by `aref` to access the element the pointer points to; where this is in memory depends, of course, on whether the array is of type `art-q`, `art-16b`, or `art-8b`. So, when a pointer is cast to a pointer type that uses a different array representation, the index has to be ”rescaled”; for instance, conversion from a pointer to `short` to a pointer to `int` requires dividing the index by 2. The effect is that when a pointer of a smaller scale is converted to a larger scale, it is automatically aligned for the larger scale; in the example just given, the information about *which* halfword in the word the `short`-pointer pointed to is lost.

4.6.57 6.7.2 From Integer Types

Conversions between integers and pointers have the following complicated behavior, which does a pretty good job of simulating conventional implementations in certain important cases:

- When an integer is converted to a pointer, a runtime check is made:
  - If the integer-typed object actually is an integer, the resulting pointer has array `NIL` and index whatever the integer was.
  - If the integer-typed object actually contains a Lisp cons (see below), the array part of the resulting pointer is the `car` of that cons, and the index part is its `cdr`.
- Similarly, when a pointer is converted to an integer:
  - If the array part of the pointer is `NIL`, the resulting integer is just the index part of the pointer.
  - Otherwise, the result is `cons` of the array part and the index part. A cons, of course, is not an integer, so an attempt to do arithmetic on it will cause a wrong-argument-type error. This is how the second case mentioned above can arise.

The result of this scheme is 1) casting `0` to a pointer always gives the null pointer; 2) casting a null pointer to an integer always gives `0`; 3) one can cast an integer to a pointer and back without loss of information, though the pointer thus created cannot be dereferenced; and 4) one can cast a pointer to an integer and back without loss of information, though this causes consing at runtime and the ”integer” cannot be used for arithmetic. Note, incidentally, that assignment of an integer to a pointer or conversely will generate a compile-time warning that a cast is being performed.

4.6.58 6.11 THE ASSIGNMENT CONVERSIONS

If the left and right side types of an assignment expression are not one of the combinations specified here, ZETA-C will issue a warning message to the effect that a cast is being attempted. If the cast is legal, compilation proceeds; otherwise (say, if an attempt is made to assign a `struct` to an `int`), an error is issued.

4.6.59 6.12 THE USUAL UNARY CONVERSIONS

An optional compilation mode is provided to suppress the implicit conversion of `floats` to `doubles` (see p. 12). Just put the line

```
#define ZETA_C_SUPPRESS_AUTO_FLOAT_TO_DOUBLE
```

at the beginning of your program.

4.6.60 6.14 THE FUNCTION ARGUMENT CONVERSIONS

The ZETA-C math library routines will accept arguments of type either `float` or `double`, and will return results of the corresponding type.

ZETA-C's representation of the null pointer is not the same as that of the number 0 (see page 13). Therefore, when a constant null pointer is to be passed as an argument to a function, the number 0 must be explicitly cast to the appropriate type; for instance,

```
foo(bar, (char *)0, quux)
```

If the cast is omitted, ZETA-C's function-argument type checker will issue a warning.

It is traditional (and good style) to define `NULL` to the null pointer expression in the preprocessor, and use it everywhere instead of the constant 0:

```
#define NULL ((char *) 0)
 ...foo(bar, NULL, quux)
```

It is less traditional, and even better style, to have several different null pointers, one for each pointer type in one's program:

```
#define CNULL ((char *) 0)
#define FNULL ((struct foo *) 0)
... etc. ...
```

4.6.61 7 Expressions

4.6.62 7.2 EXPRESSIONS AND PRECEDENCE

4.6.63 7.2.3 Overflow and Other Arithmetic Exceptions

Because of the infinite-precision integer ("bignum") facility of the Lisp Machine, integer addition or multiplication cannot overflow. Division by zero and floating-point overflow will cause a Lisp error to be signalled (see your Lisp manual for details); by default, this will invoke the Debugger. Concerning floating-point underflow, see the description of the Lisp variable `zunderflow` in your Lisp manual.

4.6.64 7.5 BINARY OPERATOR EXPRESSIONS

4.6.65 7.5.1 Multiplicative Operators

Lisp Machine integer division truncates toward zero, rather than toward negative infinity; so, for example:

```
5 / 3  =>  1
-5 / 3 =>  -1
5 % 3  =>  2
-5 % 3 =>  -2
```

4.6.66 7.5.2 Additive Operators

When two pointers are being subtracted, ZETA-C checks at *runtime* that they point into the same array, and signals an error if not (the message is "`Can't subtract pointers into different` `arrays,` *array1* `and` *array2*”).

4.6.67 7.5.3 Shift Operators

Right shifts of a signed value shift in copies of the sign bit at the left. What may seem surprising is that the same is true of right shifts of full-length *unsigned* values; but recall that these are effectively infinitely long, and so even if 0-bits were being shifted in at the ”left”, they would never become visible (see p. 12). Right shifts of (`unsigned`) `chars` and of `unsigned shorts`, on the other hand, shift 0-bits in at the left, as the language specifies and one would expect.

4.6.68 7.5.4 Inequality Operators

Note that comparison of pointers into different top-level aggregates (see the discussion of pointers and aggregates starting on p. 13) cannot yield consistent results on the Lisp Machine, since Lisp’s garbage collector can change the order of arrays in memory at runtime. We deal with this as follows. By default, the array parts of pointers are ignored in comparisons. (We have encountered C code that in fact compares pointers to different arrays, but doesn’t care what the result of the comparison is in that case.) However, ZETA-C provides an option to cause comparison of pointers into different arrays to signal an error at runtime. If you wish such errors signalled, put the line

```
#define ZETA_C_COMPARE_POINTERS_CAREFULLY
```

at the beginning of your program.

4.6.69 7.8 ASSIGNMENT EXPRESSIONS

4.6.70 7.8.2 Compound Assignment

ZETA-C requires compound assignment operators to be written as single syntactic tokens; for example, ”`&=`” rather than ”`& =`”.

4.6.71 7.11 ORDER OF EVALUATION

ZETA-C, like Lisp, evaluates the arguments of a function call in left-to-right order; but don’t count on this, since most C compilers evaluate function arguments right-to-left! ZETA-C does not currently rearrange expressions in any of the ways described in this section of *C:ARM*, but we do not guarantee that it will not do so in the future.

4.6.72 8 Statements

4.6.73 8.4 COMPOUND STATEMENT

As mentioned in section 4.2.1 above, ZETA-C does not support jumping to a labeled statement within a compound statement when the compound statement has declarations of `auto` or `register` variables.

This very reasonable restriction is enforced by the Lisp compiler.

4.6.74 8.7 SWITCH STATEMENT; CASE AND DEFAULT LABELS

ZETA-C requires that the body of a `switch` statement be a compound statement, with no declarations. Furthermore, `case` or `default` labels may not appear in any compound statement inside the `switch` body if that compound statement declares `auto` or `register` variables.

4.6.75 9 Functions

4.6.76 9.5 AGREEMENT OF FORMAL AND ACTUAL PARAMETERS

ZETA-C checks the types of actual parameters (”arguments”) against the types of the corresponding formal parameters at both compile time and load time, issuing warnings if they do not match. Since the check is done at load time, mismatches will be noticed even when the caller and callee are in different source files. Also, while normally ZETA-C conflates all the integral types – `int`s, `long`s, `unsigned`s, etc., may be mixed freely in arithmetic and assignment expressions – these types are all considered distinct for the purpose of parameter type checking (except that the `short` and `char` types are automatically widened to `int` or `unsigned`, as appropriate). This is because most C compilers cannot do type-checking in this case, and (depending on the details of the C implementation) the different types cannot be counted on to be the same length. If a traditional implementation passes function arguments on the stack, and say `int` and `long` are different sizes, and an `int` is passed where a `long` is expected, the arguments will not be aligned as the callee expects, causing erroneous behavior. So, to help the user prevent such errors, ZETA-C here provides some of the functionality of the UNIX **lint** utility.

4.6.77 9.8 AGREEMENT OF ACTUAL AND DECLARED RETURN TYPE

A `return` statement with no expression causes the Lisp object `(:|No value returned from|` *function*`)` to be returned, where *function* is the name of the function. Any attempt to do arithmetic, pointer operations, *etc.* on such an object will of course signal an error, whereupon it will be obvious that *function* was expected to return a value but didn’t.

If the type of the expression in an `return` statement is not convertible by assignment to the declared return type of the function, ZETA-C will issue a warning message to the effect that a casting conversion is being attempted. If that fails, an error is issued.

5 Library routines

This chapter lists the I/O routines and ”system calls” provided by ZETA-C. We have attempted to provide a library complete enough and compatible enough that most ”user-level” UNIX programs will run with few or no changes. We have not, however, supported multiprocessing, pipes, raw device I/O, or some of the more arcane operations UNIX provides on file descriptors. Also, note that most versions of Lisp Machine software do not support bidirectional file streams.

All of these functions are interned in package `C:`.

5.1 File and stream I/O

UNIX provides two levels of file handling routines. One consists of system calls, with which one communicates in terms of ”file descriptors”, which are (as it happens) magic numbers whose meaning is known only to the kernel; the system calls have names like `open`, `close`, `read`, `write`. The other is a collection of library routines, known as the ”standard I/O package”, which talk in terms of ”streams”, which are pointers to structures of type `FILE` (defined in `stdio.h`); most of these routines' names begin with ”f”: `fopen`, `fclose`, `fread` (but note `putc`, `getc`). Consult a UNIX manual for the details; we just wish to point out here that ZETA-C makes a distinction between file descriptors and streams, and will give you an error if you call a routine with the wrong kind of thing. (The implementation difference is just that a stream is a cons of a file descriptor and `NIL`.)

Note that a file name is anything acceptable to `fs:parse-pathname`.

5.1.1 Kernel level I/O

`int open` *name mode* *Function*

Opens the file *name* for reading (if *mode* is 0), writing (if *mode* is 1), or both (if *mode* is 2). Returns a file descriptor, or -1 on error.

`int creat` *name mode* *Function*

Creates a file named *name*. *mode* is ignored. Returns a file descriptor, or -1 on error.

`close` *fd* *Function*

Closes the file on file descriptor *fd*.

`int read` *fd buffer nbytes* *Function*

Reads up to *nbytes* bytes from file descriptor *fd*, putting the result in *buffer*. Returns the number of bytes read, which will be less than *nbytes* if end-of-file was encountered; or -1 on error.

`int write` *fd buffer nbytes* *Function*

Writes *nbytes* bytes to file descriptor *fd* from *buffer*. Returns the number of bytes written, or -1 on error.

`long lseek` *fd offset whence* *Function*

Sets the read/write pointer of file descriptor *fd* to *offset*, if *whence* is 0; or to the current location plus *offset*, if *whence* is 1; or to the end of the file plus *offset*, if *whence* is 2. If the resulting position is before the beginning of the file, does nothing and returns -1; else returns the new value of the pointer.

`long tell` *fd* *Function*

Returns the current value of the read/write pointer for file descriptor *fd*.

`int isatty` *fd* *Function*

Returns **true** (nonzero) iff file descriptor *fd* isassociated with the user's console.

UNIX normally provides input line buffering with simple editing on streams to the user's console. ZETA-C defines the flavor `zeta-c:unix-terminal-io-stream` to provide the same functionality (see `zeta-c:source;zcio.lisp` if you're curious about the details). While many of the various modes, line speeds, etc. that Unix provides its terminal streams are meaningless on the Lisp Machine's direct-connect terminal, ZETA-C does implement the following:

CBREAK mode	When active, turns off line editing, so that each character is returned when typed. *Off* by default.
ECHO mode	When active, causes each character to be echoed as typed. *On* by default.

The following functions are available to set and read these modes:

`int gtty` *fd sgttyb* *Function*

Gets the CBREAK and ECHO modes of the terminal stream associated with *fd*, and sets bits in `sgttyb.sg_flags` appropriately (see `zeta-c:include;sgtty.h`). Note that these are the only modes read. A Lisp error is signalled if *fd* is not associated with a terminal stream. Returns 0.

`int stty` *fd sgttyb &optionaldont_flush* *Function*

Sets the CBREAK and ECHO modes of the terminal stream associated with *fd* according to bits in `sgttyb.sg_flags` (see `sgtty.h`). CBREAK mode is substituted for RAW mode. Note that these are the only modes set. First clears all input from the stream and waits for pending output to complete, unless *dont_flush* is **true** (nonzero). A Lisp error is signalled if *fd* is not associated with a terminal stream. Returns 0.

`void ioctl` *fd opcode thing* — *Function*

Does one of several things depending on *opcode*. The operations available are:

`TIOCGETP`	Equivalent to `gtty(`*fd*`,` *thing*`)`.
`TIOCSETP`	Equivalent to `stty(`*fd*`,` *thing*`)`.
`TIOCSETN`	Equivalent to `stty(`*fd*`,` *thing*`, 1)` (doesn't flush the stream before setting modes).
`FIONREAD`	In this case, *thing* is expected to be a pointer to `int`. `*`*thing* is set to 1 if there are characters available to be read from *fd*, otherwise 0.

A Lisp error is signalled if *fd* is not associated with a terminal stream. Returns 0.

`void ttytimeout` *fd timeout* — *Function*

Waits up to *timeout*/60 seconds for input to be available on the terminal stream associated with *fd*. That is, it returns when input is available or when the time runs out, whichever happens first. If a character is available, it can be read with `getc` or possibly `getchar`.

5.1.2 Stdio level I/O

To use any of these functions, add the line

```
#include <stdio.h>
```

at the beginning of any relevant files. Among other things, this file defines the following variables:

`FILE *stdin` — *Variable*

The stream which is bound to the standard input (literally, to the valueof the Lisp variable `standard-input` at the point the program was invoked.)

`FILE *stdout` — *Variable*

The stream which is bound to the standard output (literally, to the valueof the Lisp variable `standard-output` at the point the program was invoked.)

`FILE *stderr` — *Variable*

The stream which is bound to the standard error output (literally, to thevalue of the Lisp variable `error-output` at the point the program was invoked.)

The standard I/O package provides the following functions:

`FILE *fopen` *name mode* *Function*

Opens the file *name* forreading (if *mode* is ”r”), creates it for writing (if *mode* is ”w”), or opens it for appending (if *mode* is ”a”). Returns a stdiostream, or NULL on error.

`fclose` *stream* *Function*

Closes the stream *stream*.

`fflush` *stream* *Function*

Causes any buffered data for the outputstream *stream* to be written out. The stream remains open.

`int feof` *stream* *Function*

Returns **true** (nonzero) iff the inputstream *stream* is at end-of-file.

`int getc` *stream* *Function*

Reads and returns the next character from *stream*; or, if end-of-file is encountered, returns `EOF` (which is an integer value not equal to any character; be sure to assign the result of `getc` to an `int`, not `char`, variable).

`int getchar` *Function*

Equivalent to `getc(stdin)`.

`int fgetc` *stream* *Function*

A synonym of `getc` (see above).

`ungetc` *c stream* *Function*

Un-gets the previously read character *c* from *stream*, so that it will be returned by the next call to `getc`. Only one character may be ungotten at a time, and it must be the same as the last character read.

`int getw` *stream* *Function*

Reads the next two bytes from *stream*, assembles them into a word, and returns the result. Does not assume, nor enforce, any special alignment. Returns `EOF` if end-of-file is encountered.

`char *gets` *s* — *Function*

Reads a line from `stdin` into the character array *s*. Does not include the trailing newline. Returns `NULL` if end-of-file was encountered at the beginning of the line.

`char *fgets` *s n stream* — *Function*

Reads a line from *stream* into the character array *s*. Reads at most *n* - 1 characters before returning. The line includes the trailing newline, if one was read. Returns `NULL` if end-of-file was encountered at the beginning of the line.

`putc` *ch stream* — *Function*

Writes the character *ch* to *stream*.

`putchar` *ch* — *Function*

Equivalent to `putc(`*ch*`,stdout)`.

`fputc` *ch stream* — *Function*

A synonym of `putc` (see above).

`putw` *w stream* — *Function*

Writes the 16-bit ”word” *w* to *stream* as a pair of bytes, in such a way that `getw` will read the same word back.

`puts` *s* — *Function*

Writes the `NUL`-terminated string *s* to `stdout`, appending a newline character.

`fputs` *s stream* — *Function*

Writes the `NUL`-terminated string *s* to *stream*. (Does not append a newline.)

`int fread` *buffer itemsize itemcount stream* — *Function*

Reads *itemcount* items, each *itemsize* bytes long, from *stream* into *buffer*. Returns the number of items actually read; this will be smaller than *itemcount* if end-of-file was encountered.

`fwrite` *buffer itemsize itemcount stream* — *Function*

Writes *itemcount* items, each *itemsize* bytes long, to *stream* from *buffer*.

`int fseek` *stream offset whence* *Function*

Sets the read/write pointer of *stream* to *offset*, if *whence* is 0; or to the current location plus *offset*, if *whence* is 1; or to the end of the file plus *offset*, if *whence* is 2. If the resulting position is before the beginning of the file, does nothing and returns `-1`; else returns the new value of the pointer.

`int ftell` *stream* *Function*

Returns the current value of the read/write pointer for *stream*. Note that on some file systems, this value is a ”magic cookie” which is only meaningful to `fseek`.

`frewind` *stream* *Function*

Sets the read-write pointer for *stream* to the beginning of the file.

5.1.3 Miscellaneous file operations

`unlink` *name* *Function*

Deletes the file named *name*. Does not expunge it, so if the file system supports undeletion, it can be undeleted.

`chdir` *newdir* *Function*

Changes the current directory (and, optionally, host and device) to that specified by *newdir*. Note that *newdir* must be recognizable by `fs:parse-pathname` as containing a directory.

5.1.4 Formatted output

These functions all interpret their *fmt* argument as described below.

`printf` *fmt &rest args* *Function*

Prints each of the *args* on `stdout` according to *fmt* (see below).

`fprintf` *stream fmt &restargs* *Function*

Prints each of the *args* on *stream* according to *fmt* (see below.)

`sprintf` *string fmt &rest args* *Function*

”Prints” each of the *args* according to *fmt* (see below), placing the output in the character array *string*; appends a `NUL` at the end.

Characters in the format string *fmt* are just copied to the output, except for `%`, which introduces a formatting directive. A directive has the following syntax:

`%[-][0]`[*width*][.*precision*][`l`]*conv*

width and *precision* are both decimal integers. If a minus sign appears before *width*, the converted value is left justified in the field; if a zero appears before *width*, padding will be done with zeros instead of blanks. *conv* may be one of the following:

`d o x`	The integer *arg* is printed in decimal, octal, or hex respectively.
`f g`	The float *arg* is printed in the style '[-]`ddd.ddd`' where the number of d's is *precision* (default 6); exponential notation is used in case of underflow or overflow.
`e`	The float *arg* is printed in the style '[-]`d.dddddde`[-]`dd`' where the number of d's is *precision* (default 6).
`s`	The `NUL`-terminated string *arg* is printed; if *precision* is specified, it is the maximum number of characters to print.
`c`	The character *arg* is printed. `NUL` is ignored.

conv may be in either case. An '`l`' before *conv* is ignored.

Note that the floating-point conversions do not behave identically to those of UNIX `printf`: there is no distinction between the `f` and `g` conversions, and the *precision* is the **total** number of digits rather than the number of digits after the decimal point.

5.2 Formatted input

These functions all interpret their *fmt* argument as described below.

`int scanf` *fmt &rest pointers* *Function*

Reads input from `stdin`, interprets it according to *fmt*, and stores the resulting values through the corresponding *pointers*.

`int fscanf` *stream fmt &rest pointers* *Function*

Reads input from *stream*, interprets it according to *fmt*, and stores the resulting values through the corresponding *pointers*.

`int sscanf` *string fmt &rest pointers* *Function*

Reads input from the `NUL`-terminated character array *string*, interprets it according to *fmt*, and stores the resulting values through the corresponding *pointers*.

[Documentation describing the interpretation of the *fmt* argument has not yet been written. However, the full `scanf` functionality, as described in Harbison & Steele, is supported.]

5.3 String manipulation

These functions work with `NUL`-terminated strings. Note that they will get array-bounds errors if the `NUL` is missing, or if there is not enough room to copy into.

`char *strcpy` *dest source* *Function*

Copies the string at *source* to *dest*. (Note the order of the arguments; think of the function as being like assignment.) Returns *dest*.

`char *strcat` *dest source* *Function*

Concatenates the string at *source* onto the end of that at *dest*. (Note the order of the arguments; see `strcpy` above.) Returns *dest*.

`int strlen` *s* *Function*

Finds the length of the string at *s*.

`int strcmp` *s1 s2* *Function*

Compares strings in ASCII order, returning -1 if $s1 < s2$, 0 if *s1* == *s2*, or 1 if $s1 > s2$.

5.4 Arithmetic and Transcendental Functions

To use any of these functions, add the line

```
#include <math.h>
```

at the beginning of the relevant files.

`int abs` *val* *Function*

Returns the absolute value of *val*.

`double acos` *x* *Function*

Returns the trigonometric inverse cosine, in radians, of its argument.

`double asin` *x* *Function*

Returns the trigonometric inverse sine, in radians, of its argument.

`double atan` *x* *Function*

Returns the trigonometric inverse tangent, in radians, of its argument. The result is between -π/2 and π/2.

`double atan2` *y x* *Function*

Returns the angle part of the polar-coordinate representation of the point (*x*, *y*); that is, the angle between the positive x-axis and a ray drawn from the origin through the point (*x*, *y*). The result is in radians and is between -π and π.

`double atof` *str* *Function*

Converts a string of numeric characters to a floating-point number. Recognizes an optional string of tabs and spaces, then an optional sign, then a string of digits optionally containing a decimal point, then an optional 'e' or 'E' followed by an optionally signed integer. The first unrecognized character ends the string.

`int atoi` *str* *Function*

Converts a string of numeric characters to an `int`. Recognizes an optional string of tabs and spaces, then an optional sign, then a string of digits. The first unrecognized character ends the string.

`long atol` *str* *Function*

Since there's no distinction in ZETA-C between an `int` and a `long`, this is effectively a synonym for `atoi`.

`double ceil` *x* *Function*

Returns the smallest integral value not less than *x*.

`double cos` *x* *Function*

Returns the trigonometric cosine of its argument, which is assumed to be in radians.

`double cosh` *x* *Function*

Returns the hyperbolic cosine of its argument.

`double exp` *x* *Function*

Returns the exponential e^x.

`double fabs` *x* *Function*

Returns the absolute value of *x*.

`double floor` *x* *Function*

Returns the largest integral value not greater than *x*.

`double hypot` *x y* *Function*

Returns the square root of the sum of the squares of *x* and *y*.

`double log` *x* *Function*

Returns the natural logarithm of *x*.

`double log10` *x* *Function*

Returns the base-10 logarithm of *x*.

`double pow` *x y* *Function*

Returns x^y.

`double sin` *x* *Function*

Returns the trigonometric sine of its argument, which is assumed to be in radians.

`double sinh` *x* *Function*

Returns the hyperbolic sine of its argument.

`double sqrt` *x* *Function*

Returns the square root of *x*.

`double tan` *x* *Function*

Returns the trigonometric tangent of its argument, which is assumed to be in radians.

`double tanh` *x* *Function*

Returns the hyperbolic tangent of its argument.

5.5 Memory allocation

`char *malloc` *size* — *Function*

Allocates a block of memory *size* bytes long, and returns a pointer to the beginning of it. This pointer can, of course, be cast to some other type. This works by calling the Lisp `make-array`, and thus can only fail if you run entirely out of virtual memory – but since all processes allocate from the same pool, this may well not be the fault of your C program. The allocated memory will contain zeros, but since other implementations' `malloc` doesn't guarantee this, you shouldn't count on it.

`char *calloc` *nelem elsize* — *Function*

Allocates a block ofmemory `to hold` *nelem* objects each *elsize* bytes long, and returns a pointer to the beginning of it. The *first* thing you should do with this pointer is to cast it to the correct type; in fact, we recommend that you call `calloc` only by way of a macro like the following:

```
#define aalloc(nelem, type)   ((type *)calloc(nelem, sizeof(type)))
```
[1]

ZETA-C explicitly recognizes such a construct, and when it allocates the block of memory, initializes it correctly for the type that it is cast to (which is *not* the same as filling it with zeros, since the null pointer is stored as a `NIL` followed by a zero). This works by calling the Lisp `make-array`, and thus can only fail if you run entirely out of virtual memory – but since all processes allocate from the same pool, this may well not be the fault of your C program.

`char *realloc` *object newsize* — *Function*

Allocates a block of memory *newsize* bytes long, and copies the contents of *object* into it (or as much thereof as will fit, if the new object is smaller). Returns a pointer to the beginning of the new object.

`free` *object* — *Function*

Currently, does nothing. (In the future, this will optionally mark the object so that subsequent accesses to it will be trapped).

5.6 Non-local exits

The "functions" `setjmp` and `longjmp` are provided to allow a function to return control to an active function invocation other than its caller's. To use them, include the line

```
#include <setjmp.h>
```

[1] Unfortunately, this only works for relatively simple types like "char *" and "struct foo"; it fails syntactically for more complex types like "int (*)()", since the second asterisk should go *inside* the first pair of parentheses. So for such cases you'll have to write the calloc expression out in full.

at the beginning of the relevant files.

`int setjmp` *env* *Function*

Sets up a return point, storing the information necessary to return to it in *env* for later use by `longjmp`. Returns 0.

`void longjmp` *env val* *Function*

Returns to the return point created by the last invocation of `setjmp` on *env*. Execution continues as if that call to `setjmp` had returned again, returning the value *val*. (If *val* is 0, or if the function that called `setjmp` has already returned, `longjmp` signals an error.)

5.7 Program termination

`exit` *status* *Function*

Exits the program immediately, closing all files. If *status* is nonzero, first offers to enter the debugger (a nonzero status is traditionally used to indicate that the program encountered a fatal error). This works by signalling `sys:abort`.

`abort` *Function*

Enters the debugger. (On UNIX, this gives a core dump.)

Index