Interlisp-D
Implementor’s Reference

Filed as: {Eris}<LispCore>Internal>Doc>ImplManual.Tedit;22

Contents

Updating the InterLisp Reference Manual [Sannella]:

Documentation files are kept on {eris}<LispManual>*.im. LispCore members are encouraged to modify these documentation files to reflect the changes made as the system is modified. Please be careful.

One important note: it is necessary to keep track of how the manual is changed, in order to provide a list of changes with the next revision of the manual. Therefore, I would strongly suggest that whenever anyone makes a significant change to the manual (adds/deletes a function definition, adds new arguments to a function, non-trivial rewording, etc.) that they send a short message to me (Sannella, not LispCore^).

The manual is stored in a large number of separate files, and it is difficult to know which file contains a particular function definition. Therefore, I have created a small package that will take an "IM Name" (a function, variable, property name, etc), create a TEdit window on the appropriate IM file, and position the TEdit cursor at the right place.

To use this, do

```
_(FILESLOAD (FROM LISPUSERS) IMNAME)
_(INSPECT.IM 'FOO)
```

`INSPECT.IM` uses the hash file package to search a hash file containing index information for the name FOO. If it is found, it will put up a pop-up menu listing references in different files. Selecting one of the references will move move the cursor in the appropriate TEdit window (if there is an active TEdit window to the appropriate file), or create a new TEdit window to the appropriate file.

Sometimes, a particular name is defined as more than one "type" (function, variable, etc.). In this case, a pop-up menu will prompt you to declare which type you are interested in.

A somewhat more convenient way of using this facility, if you want to use it repeatedly, is to do

```
_(MAKE.IM.INSPECTOR)
```

This sets up an “IM Inspector Window”, which contains a menu. Initially, this contains the single selection "Type an IM name", which prompts the user to type a name which will be looked up in the database. Below this window will appear type-selection and reference-selection menus, which do not disappear until another selection is made above them. This is hard to describe.... try it out. [It works great! -- LMM]

MAKING A LOADUP:

Loadups can currently only be made on Dorados. Command files, with extension "CM", are read by the Alto exec using the "@" command.

There are various command files on {Eris}<LispCore>CM>:

LoadFull.CM makes a LISP.SYSOUT and a FULL.SYSOUT from scratch.

LoadFullFromLisp.CM makes a FULL.SYSOUT directly from the LISP.SYSOUT.

LoadDemoFromFull.CM makes a DEMO.SYSOUT

[Note that the command files were modified to use standard cache partitions on Dorados (which are?). These are used to load the sysout that the renamed functions run in, and to save the sysouts. The old non-cached behavior can be had by using command files whose names begin with SLOW-. A complication this causes is that changes to the command files must now be made in two places, the SLOW- and caching versions.]

The first part of the command file runs MAKEINIT to create a file INIT.SYSOUT and then DLFIXINIT to make it dandelion bootable by merging in the dandelion microcode to create an INIT.DLINIT.

[FS: THE FOLLOWING INDENTED SECTION HAS BEEN CHANGED. THE CORRECT EVENTS FOLLOW BELOW:

> The next part starts up the INIT.DLINIT (which will run on all machines), and calls LOADUP(HUGE). The function LOADUP (on the file APUTDQ which is merged in at MAKEINIT time) has directions on how to do various kinds of loadups. It determines what other files are in the default loadup.
>
> If you have a special kind of loadup that you think should be supported for some applications, it is possible to add a separate clause to LOADUP and include that in the standard source.]

The next part starts up the INIT.DLINIT (which will run on all machines), and the CM script loads LOADUP.LISP and LOADFULL.LISP, which call the function LOADUP on the rest of the files in the LISP.SYSOUT and the FULL.SYSOUT, respectively. (LOADUP ’HUGE) is obsolete and should be deleted.

How MAKEINIT works:

Basically, all the storage-modifying functions are redefined so they make their changes to a file. After some initialization of the blank memory space in the SYSOUT file the normal LOAD code is run, but the effects take place in the new sysout file instead of in memory.

Modification of the low-level storage functions is done by the code in the file RENAMEFNS, based on information in the file FILESETS. The function DORENAME is called with the argument I (for INIT). DORENAME uses the RENAMETYPES variable to determine which files to get low level definitions from, and how to rename them. RENAMETYPES also indicates how to create an R (for READSYS) type rename, which is used by teleraid to read definitions out of a sysout (or sysout file).

The end result of DORENAME I is a file I-NEW, containing the "remoted" definitions. This file is loaded and the "renamed" loaded code is run to load up the earliest parts of the system.

The basic goal of MAKEINIT is to make the INIT.SYSOUT capable of loading files over the ethernet and writing out (as with LOGOUT or MAKESYS) the resulting system. This minimal set of files to load is defined in FILESETS in the variables 0LISPSET and 1LISPSET.

Once these files are loaded, INIT.SYSOUT is written out. After having the dandelion microcode spliced into the memory image by DLFIXINIT the resulting INIT.DLINIT is run. The first thing done here is to run the "init expressions" of all the files which were loaded renamed. The expressions could not be evaluated remotely earlier in the init (indeed, the evaluator is not fully loaded at this point).

Things to watch out for: The data type STREAM must be the first datatype declared after MAKEINIT time. This means that no file loaded before FILEIO can declare a datatype. Packages are "turned off" in the early part of the init and symbols written into the INIT.SYSOUT are all package qualified. The file PACKAGE-STARTUP makes the switchover (located at the end of 1LISPSET).

Now the rest of the files in FILESETS are loaded (those in 2LISPSET and up). The greatest number of problems encountered after this point, aside from outright bugs, are dependencies of code on parts of the system which have not yet been loaded.

After this point the LOADUP function loads in the standard sets of files to make LISP.SYSOUT and FULL.SYSOUT (more detail?).

Writing Renamable Code:

Renamed code is used to build loadups and read the format of the resulting memory spaces (see sections above and below).

This is intended as a start at describing what you need to knwo to write renamable code.

`\COPY` is called to move something from the local memory space to the remote one.

`\UNCOPY` brings objects back.

`LOCAL` can be used to ensure that a function's effects occur in the local memory image. It inhibits renaming of forms inside of it.

`ALLOCAL` can be used to ensure that a function's effects occur in the local memory image. It inhibits renaming of forms inside of it.

`UNLESSRDSYS` takes two forms, the first to execute normally, and the second to be used when the function is renamed.

Since `DTEST` doesn't run renamed, code that is intended to run renamed should use `ffetch` and `freplace`

How Teleraid works:

Teleraid runs in two parts: a *server*, running in microcode on one machine (usually entered by pressing UNDO to a front panel error), and a *front-end*, running on another machine. The front end machine examines the other over the network using the server.

The teleraid server is actually a simple memory page server written in microcode. It transfers pages of memory (using PUP protocols) to the front-end machine. It is up to the front-end machine to understand the internal format of the other machine's memory. This happens through a reverse version of the init process. Rather than renaming functions to write onto a remote memory space (in a file), teleraid runs functions which are renamed to read a remote memory space (on another machine or in a file; you can teleraid a non-running sysout file).

A variation of the renaming scheme used to build the init is employed in Teleraid. A large number of functions in the system are renamed to call (at their lowest levels) the Teleraid page server on the other machine.

Since the lowest levels of the system can change between releases it is important to have the same sysout running on the two machines.

The actual file that contains teleraid's renamed functions is `RDSYS`. It is created automatically by `DORENAME` on the file `RENAMEFNS` and must be updated whenever low level representation is changed. Details of what to rename for teleraid are contained on the file `FILESETS`. The functions which get renamed are scattered all through the low level system files (but those are pointed to by `FILESETS`).

Internal Storage Reference Functions [Masinter, van Melle]:

There are a number of low-level functions for directly accessing memory as if it were an enormous array of 16-bit words, bytes, 32-bit cells, etc. In general, don't call any of these directly if you can help it. They are generally "unsafe" and can confuse your system in subtle ways if misused. Also, there are often alternatives that, if not completely safe, are at least less prone to error:

> (A) Write functions or macros to do the accesses, and have them perform suitable type checking. See the `GETPUPWORD` and `PUTPUPWORD` macros, for example.
>
> (B) Define a `BLOCKRECORD` to overlay a given data structure. This is much better than using `\GETBASE` et al if you are using fixed offsets—it is safer (less error-prone) and generally produces better code. With creative use of `LOCF` and `ACCESSFNS`, you can often avoid using explicit `\GETBASE`s altogether, and your code is much more readable. Also check out the `MESATYPES` package, written by Tayloe Stansbury, for producing such expressions from Mesa type declarations.

(`\ALLOCBLOCK` *NCELLS GCTYPE INITONPAGE ALIGN*)

> The basic low level storage allocation function. *NCELLS* is the number of 32 bit cells to allocate. *GCTYPE* is an integer, usually stated as the value of either UNBOXEDBLOCK.GCT (NIL is an old style synonym), PTRBLOCK.GCT (T is an old style synonym) or CODEBLOCK.GCT. *INITONPAGE* is the number of cells at the beginning of the block which must be allocated on the same page. *ALIGN* is the alignment of the address of this block in memory space. The base address will be evenly divisable by this number.

The argument *BASE* in the following functions refers to an address, an Interlisp pointer. For example, if the value of X is an instance of a datatype, then X is actually a pointer to the first cell of that instance. There are essentially only two operations that perform "pointer arithmetic": `\ADDBASE` and `\VAG2`; these compile directly into the `ADDBASE` and `VAG2` opcodes.

(`\ADDBASE` *BASE OFFSET*)

> Produces a new address that is *OFFSET* 16-bit words beyond *BASE*.

(`\VAG2` *HI LO*)

> Produces an address whose left 8 bits is *HI* and whose right 16 bits is *LO*.

There are, however, many other ways to produce addresses that ultimately perform `\VAG2` or `\ADDBASE`, and these are usually preferable. The record `POINTER` is useful for decomposing pointers into page# and word-in-page or cell-in-page quantities. `LOCF` is useful in conjunction with `BLOCKRECORD`s and `DATATYPE`s.

(`LOCF` (**fetch** `FIELDNAME` **of** `datastructure`)) [Macro]

> Produces a pointer to the first word containing `FIELDNAME`. E.g., if `BAR` is declared as a `WORD` field in a record, then `(fetch BAR of X)` is equivalent to `(\GETBASE (LOCF (fetch BAR of X)) 0)`.

(INDEXF (**fetch** FIELDNAME **of** T)) [Macro]

> Returns the word offset to the first word containing FIELDNAME. Since this is independent of the actual datum being operated on, the datum is often given as "T". E.g., if BAR is declared as a WORD field in a record, then (fetch BAR of X) is equivalent to (\GETBASE X (INDEXF (fetch BAR of T))). There is rarely any need for INDEXF.

Note that \ADDBASE, LOCF and other pointer-producing operations are *not* in general safe in Interlisp-D. The garbage collector can get very confused if you save away arbitrary pointers anywhere other than in local variables. This is because the reference count of an object is associated only with the pointer to its beginning, i.e., only with the address that the public traffics in (the pointer returned from **create**, for example). If you must store away internal pointers, be very careful that you continue to hold on to the pointer to the start of the object for as long as you maintain the internal pointer. This assures that the object will not get garbage-collected out from under you, the most common source of such confusion.

(\ADDBASE2 *BASE N*)

> Equivalent to (\ADDBASE *BASE* 2***N*).

(\GETBASE *BASE OFFSET*)

(\PUTBASE *BASE OFFSET VALUE*)

> These fetch and store, respectively, the 16-bit word (as a Lisp small positive integer) located at *OFFSET* words beyond *BASE*. \PUTBASE is *really* dangerous. E.g., (\PUTBASE NIL n) for many small values of n will smash your system beyond repair. Not good for a residential environment where a smashed system can lose a lot of work.

(\GETBASEBYTE *BASE OFFSET*)

(\PUTBASEBYTE *BASE OFFSET BYTE*)

> Fetch and store 8-bit quanta. *BASE* is a word address, and *OFFSET* is a byte offset—counting the high byte of the base word as offset zero.

(\GETBASEPTR *BASE OFFSET*)

> Fetches a pointer at *OFFSET* from *BASE*. A pointer is a 24-bit quantity, which is stored right-justified in a 32-bit cell. Note, however, that *BASE* and *OFFSET* are both still in terms of 16-bit words.

(\PUTBASEPTR *BASE OFFSET PTR*)

> Stores pointer *PTR* at *OFFSET* from *BASE*. This is not a direct inverse of \GETBASEPTR, because it stores a full 32 bits, never mind what used to be the high 8 bits originally stored there. \PUTBASEPTR does not do reference counting, so this can be especially dangerous if not used carefully. *BASE* is a word address, and *OFFSET* is in *words*, not cells! \RPLPTR is similar to \PUTBASEPTR, except that it *does* do reference counting.

(\RPLPTR *BASE OFFSET PTR*)

> Stores a 24-bit pointer, similar to \PUTBASEPTR, except that (a) it stores only 24 bits, preserving whatever used to be in the high 8 bits; and (b) it does reference-counting operations; decrements the count of the pointer being smashed and increments the count of

the pointer being put into the location. This is the proper way to smash a pointer field if you must. However, there is almost never any need for you to call this directly; the usual way to smash a pointer field is to use records.

Implementation notes: \GETBASEBYTE and \PUTBASEBYTE compile directly into the corresponding opcodes, and execute entirely in microcode when the *OFFSET* and *VALUE* arguments are small positive integers. \GETBASE, \PUTBASE, \GETBASEPTR, \PUTBASEPTR, and \RPLPTR compile directly into the corresponding opcodes when the *OFFSET* argument is a constant less than 256; for other *OFFSET* arguments (variable quantities, larger integers), they require an ADDBASE in addition.

(\GETBASEFIXP *BASE OFFSET*)

(\PUTBASEFIXP *BASE OFFSET VALUE*)

These fetch and store 32-bit integers.

(\GETBASEFLOATP *BASE OFFSET*)

(\PUTBASEFLOATP *BASE OFFSET VALUE*)

These fetch and store 32-bit floatps.

(\GETBASESTRING *BASE OFFSET NCHARS*)

Creates a string *NCHARS* characters long whose characters consist of the bytes located starting at *OFFSET* (a byte offset) from *BASE*. Thus, the first character of the result is (\GETBASEBYTE *BASE OFFSET*).

(\PUTBASESTRING *BASE OFFSET STRING*)

Stores the characters of *STRING* as consecutive bytes starting at *OFFSET* (a byte offset) from *BASE*.

(\BLT *DBASE SBASE NWORDS*)

Copies a sequence of *NWORDS* words starting at *SBASE* to corresponding words starting at *DBASE*. This compiles directly into the BLT opcode. In the case where the source and destination ranges overlap, the behavior is well-defined: words are copied from the end of the range backwards to the beginning. Thus, this is equivalent to (for I from NWORDS-1 to 0 by -1 do (\PUTBASE DBASE (\GETBASE SBASE I))). Very fine point: this operation is defined to be completely uninterruptable if NWORDS is less than 10; thus, you can use opcode to make small indivisible transfers.

(\MOVEWORDS *SBASE SOFFSET DBASE DOFFSET NWORDS*)

Obsolete predecessor of \BLT.

(\MOVEBYTES *SBASE SBYTEOFFSET DBASE DBYTEOFFSET NBYTES*)

Copies a sequence of *NBYTES* bytes starting at *SBYTEOFFSET* bytes beyond *SBASE* to *DBYTEOFFSET* bytes beyond *DBASE*. If the ranges overlap, the result is formally undefined.

(\ZEROBYTES *BASE FIRST LAST*)

Stores zeroes into the bytes at offsets *FIRST* thru *LAST*, inclusive, from *BASE*. Thus, a total of *LAST*-*FIRST*+1 bytes are cleared.

(\ZEROWORDS *BASE ENDBASE*)

Stores zeroes into the words from *BASE* thru *ENDBASE*, inclusive. There are obscure reasons for the lack of symmetry among `\ZEROWORDS`, `\ZEROBYTES`, and `\MOVEBYTES`.

ABC, EXPORTS.ALL, &c:

Note that for Interlisp-D, LOAD(ABC) loads <LispCore>Library>EXPORTS.ALL.

(LOAD 'MAKE-EXPORTS.ALL) will connect to <LISPCORE>SOURCES> and gather all exports into the EXPORTS.ALL file.

“ABC” stands for “A Byte Compiler”—meaning the augmented environment required to compile Interlisp system code. The augmentation includes any of the definitions found under the EXPORT FILEPKG command. The variable EXPORTFILES, set up by loading the file FILESETS, contains the rootname of all system files which have any EXPORT commands. A file wil generally export those items that other files need (e.g., records or macros) which are DONTCOPY, and thus not part of the user’s system.

SYSRECORDS [van Melle]:

In order for the inspector to be able to inspect an object of some user-declared datatype, it needs a declaration for it. The declarations for system datatypes are omitted from the loadup (by being marked DONTCOPY and being initialized with INITRECORDS). In order for their instances to be inspectable, they should be added to SYSTEMRECLST by the filepkg command SYSRECORDS, which is syntactically identical to RECORDS. The datatype declaration is actually stripped of comments, subrecords and initialization info before being put out.

Putting new DLion Microcode into a Sysout:

Load the file `SPLICE.DCOM` from <Lispcore>Sources>.

(`NCLIP` *MICROCODEFILE SYSOUTFILE*)

Copies the entire contents of *MICROCODEFILE*, a DLion .db file, into *SYSOUTFILE*, which must be a Lisp sysout. Both files must be random access.

DTigerness vs. DLionness

How programs can tell if they're running on a DTiger

Programs can tell if they are running on a Dandetiger (1108 with CPE) by evaluating `(AND (EQ \MACHINETYPE \DANDELION) (ODDP (\DEVICE.INPUT 8)))` - this will return T when run on a DTiger, NIL when run on anything else. `(\DEVICE.INPUT 8)` returns the microcode version number when run on an 1108; if the version is odd, you're running on a DTiger.

WHEREIS, and the WHEREIS database

The WHEREIS package is used to find where (which source package) an atom has come from.

The expanded WHEREIS package comes preloaded in FULL.SYSOUT, or can be loaded from {ERIS}<LISPCORE>LIBRARY>WHEREIS.DCOM. Normally WHEREIS only searches loaded files. If the function WHEREIS is called with a FILES argument of T it searches the list of hashfiles given in the global WHEREIS.HASH. These hashfiles may have names such as WHEREIS.HASH, LIBRARY.WHEREISHASH, SYSTEM.WHEREISHASH, etc., and may live in places like {ERIS}<LISP>INTERMEZZO>LIBRARY. Usage:

```
_(WHEREIS <foo> 'FNS T)
```

This definition also allows you to ask for MACROS, RECORDS, PROPS, and VARS in addition to FNS.

To make a new WHEREIS database use the function WHEREISNOTICE thus:

```
_WHEREISNOTICE(
      (<LISPCORE>SOURCES <LISPCORE>LIBRARY> <LISPUSERS>)
      T
      <LISPCORE>SOURCES>SYSTEM.WHEREISHASH]
```

Note that it takes a long time to examine all the files. Best to leave this task to a lonely Dorado.

Installing New Opcodes [Masinter]:

[abstracted from messages to Jim desRivieres]

The various functions (CALLSCCODE, PRINTCODE, CHANGENAME) know about opcodes via the list \OPCODES. If you want to add some new opcodes, you can edit the list. The format of \OPCODES is documented, I think, in the function PRINTOPCODES, which is on the file ACODE. Note that if you install new opcodes on the fly you should then reset the variable \OPCODEARRAY to NIL.

You can tell if you have installed the opcodes by calling \FINDOP directly. (\FINDOP 'CAR) should return the opcode-description-record for CAR, while (\FINDOP 231Q) should look up opcode 231.

If you add opcodes, you should send a message to LispCore^ outlining what opcodes you want to reserve. The file OPCODES.TEDIT (on <LispCore>INTERNAL>DOC>) I think has a listing of opcodes too, and if you are reserving a range, that reservation should be documented there too.

Subject: adding UFNs

The UFN mechanism hasn't really been extended for simple experimentation but is workable with a little effort. Normally, UFN entries get set up at MAKEINIT time by a renamed version of a function, I think it is called \SETUFNENTRY or some such. (LLCODE, LLBASIC, LLNEW or one of those). The entries in the OPCODES record is used to set up the ufns. Now, it is currently the case that UFN's can't do anything like push N things on the stack -- all they can do is pop N arguments (N>=0) and push 1 result.

Writing ufn's that do something other than that, e.g., that don't follow the normal function call paradigm, are a lot more work. Basically I think you have to get into the level of stack-hacking that is found inside LLSTK. For example, a UFN that wanted to push a bunch of NIL's would have to do something awful, like steal space out of its own basic frame to give it back to the caller. This kind of code is tricky to write and debug, especially because you can't do things like insert BREAKs.

Popping N off of course is easy since that is what function calls do.

In order to do a jump operation, doing something like (add (fetch PC of (\MYALINK)) 10) would do a relative jump to byte +10.

It may actually be necessary to extend the UFN mechanism to allow some of the extensions that you want. Why don't you figure out what you can do with the current mechanism, and come back with the ones that you can't figure out how to implement.

Adding new opcodes to the Interlisp-D system

Written by: Herb Jellinek
Revised: 14 June 1984

The process of adding new opcodes to the Interlisp-D system has long been a mysterious one. This document is an attempt to shed some light on these mysteries. The document covers: Creating new opcodes, Writing UFNs, and The OpcodeTool. Enjoy.

Creating new opcodes

There are a number of global objects and properties that one must know about in order to install new opcodes/UFNs. Here's a list of them:

`\OPCODES` [List]

A list of the current opcodes, each of which is a record of type `OPCODE`.

`\OPCODEARRAY` [Array]

An array-ified version of `\OPCODES`. If set to `NIL` it will be reinitialized from the contents of `\OPCODES`. `\OPCODEARRAY` is recreated when needed by the function `\FINDOP`.

`DOPVAL` [Property]

Information on how to emit code for a given function. There are two formats:

1. (*nargs* . *opcode-sequence*)

If the number of arguments supplied matches *nargs*, compile into the sequence *opcode-sequence.*

2. ((*nargs1* . *opcode-sequence1*)
   (*nargs2* . *opcode-sequence2*) ...
   (*nargsN* . *opcode-sequenceN*) . *other-cases*)

If the number of arguments supplied matches *nargs1*, compile into *opcode-sequence1*, otherwise see if the number of arguments supplied matches *nargs2*, etc. One may also supply a function name as the tail of the list; the code generator will call that function if none of the other cases apply. The function `OPT.COMPILERERROR` is typically used for this purpose; it is equivalent to `HELP`.

`DOPCODE` [Property]

The `OPCODE` record for a given atom.

`OPCODE` [Record]

A record describing the structure of the `DOPCODE` property and the elements of the list `\OPCODES`; it has the following fields:

UFNFN	name of the ufn. Actually read by MAKEINIT
LEVADJ	stack effect (+/-n) or token. used by PRINTCODE. See code in PRINTCODE for details
OPPRINT	used only by PRINTCODE.
OPNARGS	number of extra bytes
OPCODENAME	name of opcode
OP#	number of opcode, or range for opcode sequences

Herb will document OPPRINT.

Writing UFNs

UFNs are Lisp functions that either run in the place of unimplemented instructions or when the microcode detects a situation that is too complex for it to handle. (This is termed *punting*.) There are two cases involved in writing UFNs: those for single-byte opcodes, and those for multi-byte opcodes.

UFNs for single-byte opcodes

For example, assume we have a single-byte opcode called SQRT, which takes a FLOATP as operand and returns its square root. The instruction is designed to punt out to its UFN (named \SQRT) when its operand is of the wrong type, at which time the UFN can either attempt to coerce the operand to a FLOATP or signal an error. \SQRT need be no more than a function of a single argument.

UFNs for multi-byte opcodes

These UFNs are slightly more complicated, but not much. The difference between single-byte UFNs and multi-byte ones is in the handling of the "extra" (alpha, beta, gamma) byte or bytes. To wit: all multi-byte opcodes that begin with the same byte have the same UFN, and the extra bytes get passed to this UFN in the form of extra arguments. We might have a group of three bit-vector operators (we'll call them BITOP), that all begin with a bytecode of 72Q and vary from 0 to 2 in the second byte. The bytecodes each expect one argument on the stack. The UFN (\BITOP), would probably have the following form:

```
(DEFINEQ
    (\BITOP
        (LAMBDA (BITVECTOR OP)
            (SELECTQ OP
                (0 (\BITOP.MASK BITVECTOR))
                (1 (\BITOP.SHIFT BITVECTOR))
                (2 (\BITOP.ROTATE BITVECTOR))
                (HELP "\BITOP - illegal operation" OP)))))
```

The OpcodeTool

[lmm: I changed the opcode format; I don't know if this works]

[hdj: it doesn’t]

[jop: fixed 5-2-86]

The OpcodeTool is a package that makes it easy to set up and test new opcodes in a running Lisp system. Do

```
    (LOAD '{Eris}<LispCore>Misc>OpcodeTool.dcom)
```

This package has one entry point, `MAKEOPCODE`, a function which takes 8 arguments:

OPNAME	a litatom
NUM	the opcode number
OPNARGS	the number of extra bytes (alpha, beta, etc.)
OPPRINT	usually T
LEVADJ	the stack level adjustment for this opcode
UFNFN	the UFN for this opcode
UNIMPL	a list describing which machines have no microcode for this op
DOPVAL	a (optional) DOPVAL prop for the litatom OPNAME

After you’ve run `MAKEOPCODE`, you can compile functions that use the new opcode and test them out.

Creating New Devices:

Date: 15 MAR 84 09:47 PST
From: MASINTER.PA
Subject: AR for Implementors Manual
To: LispSupport
cc: Kaplan, LispCore^

Section on making new file devices, How To.

VMEM.PURE.STATE [van Melle]

When preparing a demo, it is often nice to set things up in such a way that you can push the boot button at any time to instantly restart the sysout, rather than having to go back to some installation utility to reinstall the sysout. `VMEM.PURE.STATE` is a hack that lets you do this. Basically, while it is on, the page fault handler is altered to write dirty pages beyond the original end of the vmem, thus keeping the original vmem "pure".

`(VMEM.PURE.STATE` *`FLG`*`)` [Function]

> When *FLG* is true, enables "pure vmem" as of the next operation that writes out a consistent vmem, e.g., `LOGOUT`, `SAVEVM`, or `SYSOUT`. While in this state, as long as you do not perform another vmem-writing operation (`LOGOUT`, etc.), you can boot the machine (or slightly more cleanly, call `(LOGOUT T)`) and be back in the same state as the `LOGOUT` (or whatever) that initiated the pure image.
>
> When *FLG* is `NIL`, returns to normal page fault operation. This is usually not too interesting, unless you really do want to `LOGOUT`, etc and forgo the "checkpoint" you set up. Note, however, that in either case, your virtual memory file is bloated by whatever pages had been written to the end of the vmem file instead of where they belonged.
>
> There is a mode in which LOGOUT compresses the pages back to where they belong, but I never got it fully debugged.

There is a new MP error in this state: 9316. It means you wrote out so many dirty pages, you ran into the absolute end of the vmem file (8MB) even though you still have plenty of "virtual memory" left.

There are two typical modes of operation:

(1) Call `(VMEM.PURE.STATE T)` before calling `SYSOUT`, thus making a sysout that has the pure feature turned on for anyone running it.

(2) Start up a sysout not so made, and then call

```
(PROGN (CLEARW (TTYDISPLAYSTREAM))
       (VMEM.PURE.STATE T)
       (LOGOUT))
```

to turn the vmem into which this sysout was loaded one with the pure property.

Dorado Mufflers & Manifolds

```
Date: 23 Sep 84 19:29 PDT
From: JonL.pa
Subject: [JONL.PA: Mufflers and Manifolds]
To: Jellinek

A couple weeks ago I promised to send you "all my knowledge" about the alpha
bytes of the MISC opcodes.  Not much knowledge; don't even know for sure why I
picked "9", but possibly Bill will know of an "8" used for Dolphin-specific
purposes.

-- JonL --

     ----- Begin Forwarded Messages -----

Date:  6 MAR 84 18:42 PST
From: JONL.PA
Subject: Mufflers and Manifolds
To:   vanMelle, Charnley
cc:   Purcell, JonL

Sorry I didn't mention this before -- after talking with Bill and Don I
decided to use opcode MISC1 with alphabyte of 9 to do the equivalent of Mesa's
ReadWriteMufflerManifold operation.  No MISC opcodes at all were implemented
on the Dorado (before now), and the set of alphabytes from 0 through 8 seem to
be exhausted by the Dlion/Dolphin needs.

The interface to this operation is:
  Input -- one 16-bit integer;  low-order 12 bits are Muffler/Manifold
address, bit 2↑15 non-zero means to write manifold, zero means to read
muffler.
  Output -- for write operation: NIL.  for read operation: one 16-bit integer
whose low-order 15 bits are garbage and bit 2↑15 is the 1-bit value of the
muffler.

See the file <LispCore>Misc>MAKEDORADONSHOSTNUMBER for the code I'm about to
install modulo the following: (1) SELECTQ on \MACHINETYPE, (2) don't do it if
(MICROCODEVERSION) is less than 12004Q, (3) do "replaces" into (IFPAGE
NSHost[i]) rather than cons up the NSHOSTNUMBER record.


   ----- End Forwarded Messages -----
```