CHAPTER 22

INTERLISP-10 SPECIFICS

This chapter describes a number of features of Interlisp-10 that are machine or implementation-dependent, and are not expected to be implemented in newer implementations of Interlisp.

22.1 INTERLISP-10 INTERRUPT CHARACTERS

The table below gives the interrupt characters currently enabled in Interlisp-10.

Note: It is possible to change the assignments of control characters to interrupts with INTERRUPTCHAR (page 9.17).

control-B — Generates an immediate error, and causes a break, regardless of the depth or time of the computation. Thus if the function FOO is looping internally, typing control-B will cause the computation to be stopped, the stack unwound to the point at which FOO was called, and then cause a break.

This is a stronger interruption than control-H. Note that the internal variables of FOO above are not available in this break, and similarly, FOO may have already produced some changes in the environment before the control-B was typed. It may not be possible to simply continue the computation, depending on the nature of the function interrupted and when it was interrupted. Therefore whenever possible, it is better to use control-H instead of control-B.

control-C — Computation is stopped, and control returns to the operating system (Tenex, etc.) The program can be continued with the CONTINUE command.

control-D — Aborts the computation, and unwinds the stack to the top level. Calls RESET (page 9.14).

control-E — Aborts the computation, and unwinds the stack to the last ERRORSET. Calls ERROR! (page 9.14).

control-H — At the next point a function is about to be entered, the function INTERRUPT is called instead. INTERRUPT types INTERRUPTED BEFORE FN, constructs an appropriate break expression, and then calls BREAK1. The user can then examine the state of the computation, and continue by typing OK, GO or EVAL, and/or RETFROM back to some previous point, exactly as with a user break. Control-H breaks are thus always ''safe''.

Control-H breaks *only* occur when a function is called, since it is only at this time that the system is in a ''clean'' enough state to allow the user to interact. Thus, if a compiled program is looping without calling any functions (or if Interlisp-10 is in a I/O wait), control-H will not affect it. Control-B, however, will.

As soon as control-H is typed, Interlisp clears and saves the input buffer, and then rings the bell, indicating that it is now safe to type ahead to the upcoming break. If the break returns a value, i.e., is not aborted via ^ or control-D, the contents of the input buffer before the control-H was typed will be restored.

Note: Control-H will *not* interrupt at linked function calls (see page 12.18).

control-O Clears the teletype output buffer.

control-P Changes the PRINTLEVEL setting (see page 6.18).

control-S Changes the MINFS setting (see page 22.10).

control-T Prints total execution time for the program, as well as other status information.

22.2 TYPE NUMBER FUNCTIONS

Each data type in Interlisp has an associated ''type name''. In Interlisp-10, each data type also has a ''type number'', which can be accessed and manipulated with the functions below. In general, it is preferable to use the type name functions (see page 2.1).

(NTYP DATUM) [Function]

Returns the type number for the data type of DATUM , e.g., (NTYP '(A . B)) is 8, the type number for lists.

(TYPEP DATUM N) [Function]

Value is T, if the type number of DATUM is equal to N.

(TYPENAMEFROMNUMBER N) [Function]

Value is type name for type number N, or NIL if N is not a valid type number, e.g. (TYPENAMEFROMNUMBER 30)= STRING.CHARS.

(TYPENUMBERFROMNAME NAME) [Function]

Value is corresponding type number for NAME , or NIL if NAME is not a type name, e.g. (TYPENUMBERFROMNAME 'STRING.CHARS)= 30.

TYPENUMBERFROMNAME will accept READTABLEP, TERMTABLEP, CCODEP, and ARRAYP, and return the same value for each, which for Interlisp-10 is 1. Note however that (TYPENAMEFROMNUMBER 1)= ARRAYP.

(GETTYPEDESCRIPTION TYPE) [Function]

Returns the type description string for TYPE , a type name or type number.

(SETTYPEDESCRIPTION TYPE STRING) [Function]

Sets the type description string for TYPE to be STRING . The type description is used in garbage collection messages and by STORAGE.

22.3 VALIDITY OF DEFINITIONS IN INTERLISP-10

Although the function definition cell is intended for function definitions, PUTD and GETD do not make thorough checks on the validity of definitions that ''look like'' exprs, compiled code, or SUBRs. Thus if PUTD is given an array pointer, it treats it as compiled code, and simply stores the array pointer in the definition cell. GETD will then return the array pointer. Similarly, a call to that function will simply transfer to what would normally be the entry point for the function, and produce random results if the array were not compiled function.

Similarly, if PUTD is given a dotted pair of the form (number . address) where number and address fall in the subr range, PUTD assumes it is a subr and stores it away as described earlier. GETD would then return a dotted pair EQUAL (but not EQ) to the expression originally given PUTD. Similarly, a call to this function would transfer to the corresponding address.

Finally, if PUTD is given any other list, it simply stores it away. A call to this function would then go through the interpreter.

Note that PUTD does not actually check to see if the s-expression is valid definition, i.e., begins with LAMBDA or NLAMBDA. Similarly, EXPRP is true if a definition is a list and not of the form (number . address), number = 0, 1, 2, or 3 and address a subr address; SUBRP is true if it is of this form. ARGLIST and NARGS work correspondingly.

Only FNTYP and ARGTYPE check function definitions further than that described above: both ARGTYPE and FNTYP return NIL when EXPRP is true but CAR of the definition is not LAMBDA or NLAMBDA.[1] In other words, if the user uses PUTD to put (A B C) in a function definition cell, GETD will return this value, the editor and prettyprint will both treat it as a definition, EXPRP will return T, CCODEP and SUBRP NIL, ARGLIST B, and NARGS 1.

22.4 REUSING BOXED NUMBERS IN INTERLISP-10 - SETN

RPLACA and RPLACD provide a way of cannibalizing list structure for reuse in order to avoid making new structure and causing garbage collections.[2] This section describes an analogous function in Interlisp-10 for reusing large integers and floating point numbers, SETN. SETN is used like SETQ, i.e., its first argument is considered as quoted, its second is evaluated. If the current value of the variable being set is a large integer or floating point number, the new value is deposited into that word in number storage, i.e., no new storage is used.[3] If the current value is *not* a large integer or floating point number, e.g., it can be

---

[1]These functions have different values on LAMBDAs and NLAMBDAs and hence must check. The compiler and interpreter also take different actions for LAMBDAs and NLAMBDAs, and therefore generate errors if the definition is neither.

[2]The nobox package provides a more aesthetic way of reusing cons cells as well as number boxes. However, it is still the case that techniques involving reusing static storage should be used with extreme caution, and be reserved for those cases where the normal method of storage allocation and garbage collection is not workable or practical. The decl package (page 23.18) takes a different approach to the same problem by avoiding creating number boxes in the first place via type declarations in the body of the function definition.

[3]The second argument to SETN must always be a number or a NON-NUMERIC ARG error is generated.

NIL, SETN operates exactly like SETQ, i.e., the large integer or oating point number is boxed, and the variable is set. This eliminates initialization of the variable.

SETN will work interpretively, i.e., reuse a word in number storage, but will not yield any savings of storage because the boxing of the second argument will still take place, when it is evaluated. The elimination of a box is achieved only when the call to SETN is compiled, since SETN compiles open, and does not perform the box if the old value of the variable can be reused.

22.4.1 Caveats concerning use of SETN

There are three situations to watch out for when using SETN. The rst occurs when the same variable is being used for oating point numbers and large integers. If the current value of the variable is a oating point number, and it is reset to a large integer, via SETN, the large integer is simply deposited into a word in oating point number storage, and hence will be interpreted as a oating point number. Thus,

```
_(SETQ FOO 2.3)
2.3
_(SETN FOO 10000)
2.189529E-43
```

Similarly, if the current value is a large integer, and the new value is a oating point number, equally strange results occur.

The second situation occurs when a SETN variable is reset from a large integer to a small integer. In this case, the small integer is simply deposited into large integer storage. It will then print correctly, and function arithmetically correctly, but it is *not* a small integer, and hence will not be EQ to another integer of the same value, e.g.,

```
_(SETQ FOO 10000)
10000
_(SETN FOO 1)
1
_(IPLUS FOO 5)
6
_(EQ FOO 1)
NIL
_(SMALLP FOO)
NIL
```

In particular, note that ZEROP will return NIL even if the variable is equal to 0. Thus a program which begins with FOO set to a large integer and counts it down by (SETN FOO (SUB1 FOO)) must terminate with (EQP FOO 0), not (ZEROP FOO).

Finally, the third situation to watch out for occurs when you want to save the current value of a SETN variable for later use. For example, if FOO is being used by SETN, and the user wants to save its current value on FIE, (SETQ FOO FIE) is not su cent, since the next SETN on FOO will also change FIE, because its changes the word in number storage pointed to by FOO, and hence pointed to by FIE. The number must be copied, e.g., (SETQ FIE (IPLUS FOO)), which sets FIE to a new word in number

storage.

(SETN VAR X) [NLambda Function]
A nlambda function like SETQ. VAR is quoted, X is evaluated, and its value must be a number. VAR will be set to this number. If the current value of VAR is a large integer or oating point number, that word in number storage is cannibalized. The value of SETN is the (new) value of VAR .

22.5 BOX AND UNBOX IN INTERLISP-10

Some applications may require that a user program explicitly perform the boxing and unboxing operations that are usually implicit (and invisible) to most programs. The functions that perform these operations are LOC and VAG respectively. For example, if a user program executes a TENEX JSYS using the ASSEMBLE directive, the value of the ASSEMBLE expression will have to be boxed to be used arithmetically, e.g., (IPLUS X (LOC (ASSEMBLE --))). It must be emphasized that

Arbitrary unboxed numbers should NOT be passed around as ordinary values because they can cause trouble for the garbage collector.

For example, suppose the value of X were 150000, and you created (VAG X), and this just *happened* to be an address on the free storage list. The next garbage collection could be disastrous. For this reason, the function VAG must be used with extreme caution when its argument's range is not known.

LOC is the inverse of VAG. It takes an address, i.e., a 36 bit quantity, and treats it as a number and boxes it. For example, LOC of an atom, e.g., (LOC (QUOTE FOO)), treats the atom as a 36 bit quantity, and makes a number out of it. If the address of the atom FOO were 125000, (LOC (QUOTE FOO)) would be 125000, i.e., the location of FOO. It is for this reason that the box operation is called LOC, which is short for location.

Note that FOO does not print as #364110 (125000 in octal) because the print routine recognizes that it is an atom, and therefore prints it in a special way, i.e., by printing the individual characters that comprise it. Thus (VAG 125000) would print as FOO, and would in fact *be* FOO.

(LOC X) [Function]
Makes a number out of X, i.e., returns the location of X.

(VAG X) [Function]
The inverse of LOC. X must be a number; the value of VAG is the unbox of X.

The compiler eliminates extra VAG's and LOC's for example (IPLUS X (LOC (ASSEMBLE --))) will not box the value of the ASSEMBLE, and then unbox it for the addition.

22.6 MISCELLANEOUS OPERATING SYSTEM FUNCTIONS

(LOADAV) [Function]
Returns the current load average as a oating point number (this number is the

rst of the three printed by the SYSTAT command).

(ERSTR ERN _) [Function]

ERN is an error number from a JSYS fail return. ERN = NIL means the most recent error. ERSTR returns the operating system error diagnostic as a string.

(JSYS N AC1 AC2 AC3 RESUL TAC) [Function]

Loads the (unboxed) values of AC1, AC2, and AC3 into appropriate accumulaters, and executes JSYS number N. If AC1, AC2, or AC3= NIL, 0 is used. JSYS returns the (boxed) contents of the accumulator speci ed by RESUL TAC, i.e., 1 means AC1, 2 means AC2, and 3 means AC3, with NIL equivalent to 1. Compiles open if N is itself a small integer, and RESUL TAC is a small integer, or NIL.

If the JSYS causes a trap, the message TRAP AT LOCATION NNNNN is printed by the operating system, followed by JSYS ERROR: and the operating system diagnostic. The user is then talking to the operating system exactly as though control-C had been typed. If the user then continues using the CONTINUE command, an Interlisp error is generated, JSYS ERROR, and control then proceeds the same as for any other avor of error, i.e. unwinds to last ERRORSET or goes into a break as described on page 9.10.

The CJSYS package (page 23.53) enables calling JSYSes by their corresponding name, rather than their number.

(USERNUMBER A FLG) [Function]

If A= NIL, returns the login user number; if A= T, returns the connected user number; if A is a literal atom or string, USERNUMBER returns the number of the corresponding user, or NIL if no such user exists.

On TOPS-20, there is a di erence between the user number, which is associated with the job, and the directory number, which is associated with the le system. Therefore, on TOPS-20, if FLG= T, USERNUMBER returns the directory number rather than the user number.

(HOSTNAME HOSTN FLG) [Function]

Returns the hostname as a string for host number HOSTN, e.g. "PARC-MAXC2", "BBN-TENEXD", etc. If HOSTN = NIL, the local host is used. If the local host is not an arpanet host, value is NIL. Also returns NIL if HOSTN is not a valid host number.

FLG is interpreted the same as in USERNAME.

(HOSTNUMBER) [Function]

Returns the host numer of the local host, or NIL, if the local host is not an arpanet host.

(TENEX STR FILEFLG) [Function]

Starts up a lower exec (without a message) using SUBSYS, and then if FILEFLG= NIL unreads STR, followed by "QUIT"[4] (using BKSYSBUF, page 6.47). TENEX returns

[4] "POP" for Interlisp on TOPS-20.

T if all of STR is actually processed/read by the lower exec, NIL if the user control-C's and manually QUIT's back to Interlisp.

If FILEFLG = T, TENEX passes the string as the second argument to SUBSYS, instead of unreading it. This has the advantage that STR can be of any length, and also that typeahead will not interfere with the call to the lower exec. The disadvantage is that TENEX cannot tell whether the commands to the lower exec terminated successfully, or were aborted. Thus, if FILEFLG = T, the value of TENEX is always T.

For example, LISTFILES (page 11.9) is implemented using TENEX, with FILEFLG = NIL, so LISTFILES can tell if listings actually were completed.

22.7 STORAGE ALLOCATION AND GARBAGE COLLECTION

In the following discussion, we will speak of a quantity of memory being assigned to a particular data-type, meaning that the space is reserved for storage of elements of that type. *Allocation* will refer to the process used to obtain from the already assigned storage a particular location for storing one data element.

A small amount of storage is assigned to each data-type when Interlisp-10 is started; additional storage is assigned only during a garbage collection.

The page is the smallest unit of memory that may be assigned for use by a particular data-type. For each page of memory there is a one word entry in a type table. The entry contains the data-type residing on the page as well as other information about the page. The type of a pointer is determined by examining the appropriate entry in the type table.

Storage is allocated as is needed by the functions which create new data elements, such as CONS, PACK, MKSTRING. For example, when a large integer is created by IPLUS, the integer is stored in the next available location in the space assigned to integers. If there is no available location, a garbage collection is initiated, which may result in more storage being assigned.

The storage allocation and garbage collection methods differ for the various data-types. The major distinction is between the types with elements of fixed length and the types with elements of arbitrary length. List cells, atoms, large integers, floating point numbers, and string pointers are fixed length; all occupy 1 word except atoms which use 3 words. Arrays, print names, and strings (string characters) are variable length.

Elements of fixed length types are stored so that they do not overlap page boundaries. Thus the pages assigned to a fixed length type need not be adjacent. If more space is needed, any empty page will be used. The method of *allocating* storage for these types employs a free-list of available locations; that is, each available location contains a pointer to the next available location. A new element is stored at the first location on the free-list, and the free-list pointer is updated. [5]

[5]The allocation routine for list cells is more complicated. Each page containing list cells has a separate free list. First a page is chosen, then the free list for that page is used. Lists are the only data-type which operate this way.

Elements of variable length data-types *are* allowed to overlap page boundaries. Consequently all pages assigned to a particular variable length type must be contiguous. Space for a new element is allocated following the last space used in the assigned block of contiguous storage.

When Interlisp-10 is rst called, a few pages of memory are assigned to each data-type. When the allocation routine for a type determines that no more space is available in the assigned storage for that type, a garbage collection is initiated. The garbage collector determines what data is currently in use and reclaims that which is no longer in use. A garbage collection may also be initiated by the user with the function `RECLAIM`.

Data in use (also called active data) is any data that can be ''reached'' from the currently running program (i.e., variable bindings and functions in execution) or from atoms. To nd the active data the garbage collector ''chases'' all pointers, beginning with the contents of the push-down lists and the components (i.e., `CAR`, `CDR`, and function de nition cell) of all atoms with at least one non-trivial component.

When a previously unmarked datum is encountered, it is marked, and all pointers contained in it are chased. Most data-types are marked using bit tables; that is tables containing one bit for each datum. Arrays, however, are marked using a half-word in the array header.

When the mark and chase process is completed, unmarked (and therefore unused) space is reclaimed. Elements of xed length types that are no longer active are reclaimed by adding their locations to the free-list for that type. This free list allocation method permits reclaiming space without moving any data, thereby avoiding the time consuming process of updating all pointers to moved data. To reclaim unused space in a block of storage assigned to a variable length type, the active elements are compacted toward the beginning of the storage block, and then a scan of all active data that can contain pointers to the moved data is performed to update the pointers. [6]

Whenever a garbage collection of any type is initiated, [7] unused space for all xed length types is reclaimed since the additional cost is slight. However, space for a variable length type is reclaimed only when that type initiated the garbage collection.

If the amount of storage reclaimed for the type that initiated the garbage collection is less than the minimum free storage requirement for that type, the garbage collector will assign enough additional storage to satisfy the minimum free storage requirement. The minimum free storage requirement for each data may be set with the function `MINFS`. The garbage collector assigns additional storage to xed length types by nding empty pages, and adding the appropriate size elements from each page to the free list. Assigning additional storage to a variable length type involves nding empty pages and moving data so that the empty pages are at the end of the block of storage assigned to that type.

In addition to increasing the storage assigned to the type initiating a garbage collection, the garbage collector will attempt to minimize garbage collections by assigning more storage to other xed length types according to the following algorithm. If the amount of active data of a type has increased since the last garbage collection by more than 1/4 of the `MINFS` value for that type, storage is increased (if necessary), to attain the `MINFS` value. If active data has increased by less than 1/4 of the `MINFS` value,

---

[6] If Interlisp-10 types the message `ARRAYS FOULED` during a garbage collection, it means that an array header has been clobbered and no longer makes sense. This can be due to hardware malfunction, or an as yet undiscovered bug in Interlisp. The best thing to do under these circumstances is to give up and start over with a fresh system or sysout.

[7] The ''type of a garbage collection'' or the ''type that initiated a garbage collection'' means either the type that ran out of space and called the garbage collector, or the argument to `RECLAIM`.

available storage is increased to 1/2 MINFS. If there has been no increase, no more storage is added. For example, if the MINFS setting is 2000 words, the number of active words has increased by 700, and after all unused words have been collected there are 1000 words available, 1024 additional words (two pages) will be assigned to bring the total to 2024 words available. If the number of active words had increased by only 300, and there were 500 words available, 512 additional words would be assigned.

(RECLAIM TYPE) [Function]

Initiates a garbage collection of type TYPE , where TYPE is either a type name or type number. Value of RECLAIM is number of words available (for that type) after the collection.

Garbage collections, whether invoked directly by the user or indirectly by need for storage, do not con ne their activity solely to the data type for which they were called, but automatically collect some or all of the other types.

(GCGAG MESSA GE) [Function]

A ects messages printed by the garbage collector. If MESSA GE = T, whenever a garbage collection is begun, "collecting" is printed, followed by the type description of the type that initiated the collection.[8] When the garbage collection is complete, two numbers are printed: the number of words collected for that type, and the total number of words available for that type, i.e., allocated but not necessarily currently in use. Note that other types may also have been collected, and had more storage assigned.

Example:

```
_RECLAIM(18)

collecting large numbers
511, 3071 free cells
3071
_RECLAIM(LITATOM)

collecting atoms
1020, 1020 free cells
1020
```

If MESSA GE = NIL, no garbage collection message is printed, either on entering or leaving the garbage collector.

If MESSA GE is a list, CAR of MESSA GE is printed (using PRIN1) when the garbage collection is begun, and CDR is printed (using PRIN1) when the collection is nished. If MESSA GE is a literal atom or string, MESSA GE is printed when the garbage collection is begun, and nothing is printed when the collection nishes.

If MESSA GE is a number, the message is the same as for (GCGAG T), except if the total number of free pages left after the collection is less than MESSA GE , the number of free pages is printed, e.g.,

[8]Note that this type description can be set via the function SETTYPEDESCRIPTION (page 22.2).

```
_GCGAG(100)
T
_RECLAIM()

collecting lists
10369, 10369 free cells, 87 pages left.
```

The initial setting for GCGAG is 40.

The value of GCGAG is its previous setting.

(GCMESS MESSAGE STRING) [Function]

GCGAG is implemented in terms of the primitive GCMESS which can be used to further refine garbage collection messages for specialized applications. The garbage collection message is actually composed of seven separate messages:

```
collecting large numbers[1][2]
511,[3] 3071 free cells[4], 87[5] pages[6] left[7]
```

message #1 is the ''collecting'' string. If NIL, then neither it, nor the type dependent field (which is settable via SETTYPEDESCRIPTION described below) is printed.

message #2 is the carriage-return after the type-dependent field. Thus to simply print a string at the beginning of a garbage collection, perform (GCMESS 1) and (GCMESS 2 STRING).

message #3 is the '','' which comes after the number of cells actually collected. If NIL, then neither it nor that number are printed.

message #4 is the ''free cells'' which comes after the number of cells that are now allocated. If NIL, neither it nor that number are printed.

message #5 is the number of pages left below which the system prints message 6.

message #6 is the ''pages left'' message. If NIL, neither it nor the number of pages left are printed.

message #7 is the terminating carriage return.

(MINFS N TYPE) [Function]

Sets the minimum amount of free storage which will be maintained by the garbage collector for data types of type number or type name TYPE . If, after any garbage collection for that type, fewer than N free words are present, sufficient storage will be added (in 512 word chunks) to raise the level to N.

If TYPE = NIL, LISTP is used, i.e., the MINFS refers to list words.

If N = NIL, MINFS returns the current MINFS setting for the corresponding type.

A MINFS setting can also be changed dynamically, even during a garbage collection, by typing control-S[9] followed by a number, followed by a period. When the control-S is typed, Interlisp immediately clears and saves the input buffer, rings the bell, and waits for input, which is terminated by any non-number. The input buffer is then restored, and the program continues. If the input was terminated by other than a period, it is ignored. If the control-S was typed during a garbage collection, the number is the new MINFS setting for the type being collected, otherwise for type 8, i.e., list words.

(MINHASH X) [Function]

The atom hash table automatically expands by a specified number of pages each time it fills up. The number of pages is set via the function MINHASH. The initial setting is (MINHASH 2) (room for 1024 new atoms).

(GCTRP N) [Function]

''Garbage Collection Trap''. Causes a (simulated) control-H interrupt when the number of free list words remaining equals N, i.e., when a garbage collection would occur in N more conses. The message GCTRP is printed, the function INTERRUPT is called, and a break occurs. Note that by advising INTERRUPT the user can program the handling of a GCTRP instead of going into a break.[10]

GCTRP returns its last setting.

(GCTRP -1) will ''disable'' a previous GCTRP since there are never -1 free list words. GCTRP is initialized this way.

(GCTRP) returns the number of list cells left, i.e., number of CONSes until next type LISTP garbage collection.

(CLOSER A X) [Function]

Stores X into memory location A. Both X and A must be numbers.

(OPENR A) [Function]

Returns the number in memory location A, i.e., boxed.

22.8 THE ASSEMBLER AND LAP

The Interlisp-10 compiler has two principal passes. The first compiles its input into a macro assembly language called LAP.[11] The second pass expands the LAP code, producing (numerical) machine language instructions. The output of the second pass is written on a file and/or stored in binary program space.

---

[9] control-X for Interlisp-10 on TOPS-20.

[10] For GCTRP interrupts, INTERRUPT is called with INTYPE (its third argument) equal to 3. If the user does not want to go into a break, the advice should still allow INTERRUPT to be entered, but first set INTYPE to -1. This will cause INTERRUPT to ''quietly'' go away by calling the function that was interrupted. The advice should *not* exit INTERRUPT via RETURN, as in this case the function that was about to be called when the interrupt occurred would not be called.

[11] The exact form of the macro assembly language is extremely implementation dependent, as well as being influenced by the architecture and instruction set for the machine that will run the compiled program.

Input to the compiler is usually a standard Interlisp EXPR definition. However, in Interlisp-10, machine language coding can be included within a function by the use of one or more ASSEMBLE forms as described below. In other words, ASSEMBLE allows the user to write portions of a function in LAP. Note that ASSEMBLE is only a compiler directive; it has no independent definition. Therefore, functions which use ASSEMBLE must normally be compiled in order to run.[12]

22.8.1 Assemble

Note: ASSEMBLE *is provided for situations where its use is unavoidable. However, its use is definitely not encouraged. The disadvantages are several.* ASSEMBLE *code is unavoidably dependent on the PDP-10, Tenex, and implementation details of Interlisp-10. Thus,* ASSEMBLE *code is not transportable to Interlisp on another machine or operating system, and implementation changes to Interlisp-10 can (and frequently do) require changes to existing* ASSEMBLE *code.*

The format of ASSEMBLE is similar to that of PROG:

(ASSEMBLE V S_1 S_2 . . . S_N)

V is a list of variables to be bound during the first pass of the compilation, *not* during the running of the object code. The assemble statements S_1 ... S_N are compiled sequentially, each resulting in one or more instructions of object code. When run, the value of the ASSEMBLE ''form'' is the contents of AC1 at the end of the execution of the assemble instructions. Note that ASSEMBLE may appear anywhere in an Interlisp-10 function. For example, one may write:

```
(IGREATERP (IQUOTIENT (LOC (ASSEMBLE NIL
                                     (MOVEI 1 , -5)
                                     (JSYS 13)))
                           1000)
           4)
```

to test if job runtime exceeds 4 seconds.[13]

22.8.1.1 Assemble Statements

If an assemble statement is an atom, it is treated as a label identifying the location of the next statement that will be assembled.[14] Such labels defined in an ASSEMBLE form are like PROG labels in that they may be referenced from the current and lower level nested PROGs or ASSEMBLEs.

---

[12]The MACROTRAN package (page 5.19) does permit the user to run programs interpretively which contain ASSEMBLE directives. Each ASSEMBLE directive is compiled as a separate function. There is some loss in efficiency over compiling the entire function as a unit, and not all ASSEMBLE expressions are tractable to this procedure.

[13]This example is to illustrate use of ASSEMBLE, and is *not* a recommendation to use the above code. The function JSYS (page 22.6) is the appropriate method.

[14]A label can be the last thing in an ASSEMBLE form, in which case it labels the location of the first instruction *after* the ASSEMBLE form.

If an assemble statement is not an atom, CAR of the statement must be an atom and one of: (1) a number; (2) a LAP op-def (i.e., has a property value OPD); (3) an assembler macro (i.e., has a property value AMAC); or (4) one of the special assemble instructions given below, e.g., C, CQ, etc. Anything else will cause the error message OPCODE? - ASSEMBLE.

The types of assemble statements are described here in the order of priority used in the ASSEMBLE processor; that is, if an atom has both properties OPD and AMAC, the OPD will be used. Similarly a special ASSEMBLE instruction may be redefined via an AMAC. The following descriptions are of the first pass processing of ASSEMBLE statements. The second pass processing is described in the section on LAP, page 22.15.

(1) numbers

If CAR of an assemble statement is a number, the statement is not processed in the first pass (see page 22.15).

(2) LAP op-defs

The property OPD is used for two different types of op-defs: PDP-10 machine instructions, and LAP macros. If the OPD definition (i.e., the property value) is a number, the op-def is a machine instruction. When a machine instruction, e.g., HRRZ, appears as CAR of an assemble statement, the statement is not processed during the first pass but is passed to LAP. The forms and processing of machine instructions by LAP are described on page 22.16.

If the OPD definition is not a number, then the op-def is a LAP macro. When a LAP macro is encountered in an assemble statement, its arguments are evaluated and processing of the statement with evaluated arguments is left for the second pass and LAP. For example, LDV is a LAP macro, and (LDV (QUOTE X) SP) in assemble code results in (LDV X N) in the LAP code, where N is the value of SP. The form and processing of LAP macros are described on page 22.17.

(3) assemble macros

If CAR of an assemble statement has a property AMAC, the statement is an assemble macro call. There are two types of assemble macros: lambda and substitution. If CAR of the macro definition is the atom LAMBDA, the definition will be *applied* to the arguments of the call and the resulting list of statements will be assembled. For example, REPEAT could be defined as a LAMBDA macro with two arguments, N and M, which expands into N occurrences of M, e.g., (REPEAT 3 (CAR1)) expands to ((CAR1) (CAR1) (CAR1)). The definition (i.e., value of property AMAC) for REPEAT could be:

```
(LAMBDA (N M)
  (PROG (YY)
     A  (COND
           ((ILESSP N 1)
            (RETURN (CAR YY)))
           (T (SETQ YY (TCONC YY M))
              (SETQ N (SUB1 N))
              (GO A)))))
```

If CAR of the macro definition is not the atom LAMBDA, it must be a list of dummy symbols. The arguments of the macro call will be substituted for corresponding appearances of the dummy symbols in

CDR of the de nition, and the resulting list of statements will be assembled.[15] For example, ABS could be a substitution macro which takes one argument, a number, and expands into instructions to place the absolute value of the number in AC1:

```
((X)
 (CQ (VAG X))
 (CAIGE 1 , 0))
 (MOVN 1 , 1))
```

(4) special assemble statements

(CQ E_1 E_N) CQ (compile quote) takes any number of arguments which are assumed to be regular Interlisp expressions and are compiled in the normal way. E.g.

```
(CQ (COND
       ((NULL Y)
        (SETQ Y 1)))
    (SETQ X (IPLUS Y Z)))
```

Note: to avoid confusion and minimize dependence on the current implementation, it is best to have as much of a function as possible compiled in the normal way, e.g., to load the value of X to AC1, (CQ X) is preferred to (LDV (QUOTE X) SP).

(C E_1 E_N) C (Compile) takes any number of arguments which are rst evaluated, then compiled in the usual way. Both C and CQ permit the inclusion of regular compilation within an assemble form.

(E E_1 E_N) E (Evaluate) takes any number of arguments which are evaluated in sequence. For example, (PSTEP) calls a function which increments the compiler variable SP.

(SETQ VAR) Compiles code to set the variable VAR to the contents of AC1.

(VAR (OP AC , VARNAME))
Permits writing a machine instruction with the value of a variable as the operand. Generates the appropriate address and index elds to reference the value of VARNAME . VARNAME may be a locally bound variable, free variable, GLOBALVAR, etc. Note that VAR may generate more than one instruction.

(* ...) Used to indicate a comment; the statement is ignored.

22.8.1.2 COREVALs

There are several locations in the basic machine code of Interlisp- 10 which may be referenced from compiled code. The current value of each location is stored on the property list under the property

[15]Note that assemble macros produce a list of statements to be assembled, whereas compiler macros produce a single expression. An assemble macro which *computes* a list of statements begins with LAMBDA and may be *either* spread or no-spread. The analogous compiler macro begins with an atom, (i.e., is always no-spread) and the LAMBDA is understood.

COREVAL.[16] Since these locations may change in different reassemblies of Interlisp-10, they are written symbolically on compiled code files, i.e., the name of the corresponding COREVAL is written, not its value. Some of the COREVALs used frequently in ASSEMBLE are:

KT contains (pointer to) atom T

KNIL Contains (a pointer to) the atom NIL.

MKN Routine to box an integer.

MKFN Routine to box floating number.

IUNBOX Routine to unbox an integer.

FUNBOX Routine to unbox floating number.

The index registers used for the push-down stack pointers are also included as COREVALS. These are not expected to change, and are not stored symbolically on compiled code files; however, they should be referenced symbolically in assemble code. They are:

PP Parameter stack.

CP Control stack.

VP Basic frame pointer.

22.8.2 LAP

LAP (for LISP Assembly Processor) expands the output of the first pass of compilation to produce numerical machine instructions.

22.8.2.1 LAP Statements

If a LAP statement is an atom, it is treated as a label identifying the location of the next statement to be processed. If a LAP statement is not an atom, CAR of the statement must be an atom and either: (1) a number; (2) a machine instruction; or (3) a LAP macro.

(1) numbers

If CAR of a LAP statement is a number, a location containing the number is produced in the object code.[17] E.g.,

```
(ADD 1 , A (1))
   .
   .
```

[16]The value of COREVALS is a list of all atoms with COREVAL properties.

[17]Note that if a function is intended to be swappable, it may not contain any relocatable, indexed instructions.

.

```
A    (1)
     (4)
     (9)
```

Statements of this type are processed like machine instructions, with the initial number serving as a 36-bit op-code.

(2) Machine Instructions

If CAR of a LAP statement has a numeric value for the property OPD,[18] the statement is a machine instruction. The general form of a machine instruction is:

```
(OPCODE  AC , @ ADDRESS  (index))
```

OPCODE is any PDP-10 instruction mnemonic or Interlisp UUO.[19]

AC, the accumulator field, is optional. However, if present, it *must* be followed by a comma. AC is either a number or an atom with a COREVAL property. The low order 4 bits of the number or COREVAL are OR'd to the AC field of the instruction.

@ may be used anywhere in the instruction to specify indirect addressing (bit 13 set in the instruction) e.g., (HRRZ 1 , @ 1 (VP)).

ADDRESS is the address field which may be any of the following:

= CONSTANT — Reference to an unboxed constant. A location containing the unboxed constant will be created in a region at the end of the function, and the address of the location containing the constant is placed in the address field of the current instruction. The constant may be a number e.g., (CAME 1 , = 3596); an atom with a property COREVAL (in which case the constant is the value of the property, at LOAD time); any other atom which is treated as a label (the constant is then the address of the labeled location) e.g., (MOVE 1 , = TABLE) is equivalent to (MOVEI 1 , TABLE); or an expression whose value is a number.

' POINTER — The address is a reference to a Interlisp pointer, e.g., a list, number, string, etc. A location containing the pointer is assembled at the end of the function, and the current instruction will have the address of this location, e.g.,

```
(HRRZ 1 , ' "IS NOT DEFINED")

(HRRZ 1 , ' (NOT FOUND))
```

* — Specifies the current location in the compiled function; e.g., (JRST * 2) has the same effect as (SKIPA).

a literal atom — If the atom has a property COREVAL, it is a reference to a system location, e.g., (SKIPA 1 , KNIL), and the address used is the value of the COREVAL.

---

[18]The value is an 18 bit quantity (rather than 9), since some UUO's also use the AC field of the instruction.

[19]The TENEX JSYS's are not defined, that is, one must write (JSYS 107) instead of (KFORK).

Otherwise the atom is a label referencing a location in the LAP code, e.g., `(JRST A)`.

a number — The number is the address; e.g.,

```
(MOVSI 1 , 400000Q)
(HLRZ 2 , 1 (1))
```

a list — The form is evaluated, and its value is the address.

Anything else in the address field causes an error message, e.g., `(SKIPA 1 , KNILL) - LAPERROR`. A number may follow the address field and will be added to it, e.g., `(JRST A 2)`.

`INDEX` is denoted by a *list* following the address field, i.e., the address field *must* be present if an index field is to be used. The index (`CAR` of the list) must be either a number, or an atom with a property `COREVAL`, e.g., `(HRRZ 1 , 0 (1))`.

(3) LAP macros

If `CAR` of a LAP statement is the name of a LAP macro, i.e., has the property `OPD`, the statement is a macro call. The arguments of the call follow the macro name: e.g., `(LQ2 FIE 3)`.

LAP macro calls comprise most of the output of the first pass of the compiler, and may also be used in `ASSEMBLE`. The definitions of these macros are stored on the property list under the property `OPD`, and like assembler macros, may be either lambda or substitution macros. In the first case, the macro definition is applied to the arguments of the call;[20] in the second case, the arguments of the call are substituted for occurrences of the dummy symbols in the definition. In both cases, the resulting list of statements is again processed, with macro expansion continuing till the level of machine instructions is reached.

Some examples of LAP macros are shown below.

```
(DEFLIST
'[(LQ ((X)                              (* LOAD QUOTE TO AC1)
       (HRRZ 1 , ' X)))
  (LQ2 ((X AC)                          (* LOAD QUOTE TO AC)
        (HRRZ AC , ' X)))
  (LDV ((A SP)                          (* LOAD LOCAL VARIABLE TO AC1)
        (HRRZ 1 , (VREF A SP))))
  (STV ((A SP)                          (* SET LOCAL VARIABLE FROM AC1)
        (HRRM 1 , (VREF A SP))))
  (LDV2 ((A SP AC)                      (* LOAD LOCAL VARIABLE TO AC)
         (HRRZ AC , (VREF A SP))))
  (LDF ((A SP)                          (* LOAD FREE VARIABLE TO AC1)
        (HRRZ 1 , (FREF A SP))))
  (STF ((A SP)                          (* SET FREE VARIABLE FROM AC1)
        (HRRM 1 , (FREF A SP))))
  (LDF2 ((A SP)                         (* LOAD FREE VARIABLE TO AC)
         (HRRZ 2 , (FREF A SP))))
  (CAR1 (NIL                            (* CAR OF AC1 TO AC1)
```

[20]The arguments were already evaluated in the first pass, see page 22.13.

```
        (HRRZ 1 , 0 (1))))
  (CDR1 (NIL                          (* CDR OF AC1 TO AC1)
        (HLRZ 1 , 0 (1))))
  (CAR2 ((AC)                         (* CAR OF AC TO AC)
        (HRRZ AC , 0 (AC))))
  (CLL ((NAM N)                       (* CALL FN WITH N ARGS GIVEN)
       (CCALL N , ' NAM)))
  (LCLL ((NAM N)                      (* LINKED CALL WITH N ARGS)
        (LNCALL N , (MKLCL NAM))))
  (RET (NIL                           (* RETURN FROM FN)
       (POPJ CP ,))
  (PUSHP (NIL (PUSH PP , 1)))
  (PUSHQ ((X)                         (* PUSH QUOTE)
          (PUSH PP , ' X)))]
'OPD)
```

22.8.3 Using Assemble

In order to use ASSEMBLE, it is helpful to know the following things about how compiled code is run. All variable bindings and temporary pointers are stored on the parameter pushdown stack (addressed by index register PP). Control information is stored on the control pushdown stack (addressed by index register CP). A function call proceeds as follows:

1. The calling function pushes the argument values on the parameter stack.

2. The calling function invokes a routine that adjusts the number of arguments if too few or too many were supplied, and binds the arguments. Binding usually implies the creation of a basic frame.[21]

3. Then the called function is run.

The arguments in the basic frame are referenced relative to index register VP, e.g., 1(VP) addresses the rst argument. However, it is better to reference variables in less implementation dependent ways, such as (CQ ...) or (VAR (...)). The compiler will then generate the correct code whether the variable is bound locally, is a free reference, is a GLOBALVAR, etc.

The parameter stack may be used for temporary storage of pointers. Both halves of a word on the parameter stack may be pointers. On the control stack the right half of a word must be a pointer, the left a non-pointer. Anything else can cause the garbage collector to fail.

For temporary storage of unboxed numbers, the following ASSEMBLE macros are provided:

(PUSHN ADDR) ''Pushes'' the number referenced by ADDR . ADDR may be any legal ASSEMBLE code address eld, for example: (PUSHN 1), (PUSHN = 0), (PUSHN @ 2)

(POPN ADDR) ''Pops'' the most recent number to ADDR .

[21] Whether a basic frame is created for a PROG or open lambda depends on whether any of the variables are specvars.

`(NREF (OP AC , N))`
References a previously pushed number. `OP` is the opcode, `AC` is the accumulator, `N` is the relative position of the desired number on the pseudo number stack. That is, `N = 0` refers to the most recent number, `N = -1` to the next most recent, etc. For example: `(NREF (MOVN 1, -1))`

`(PUSHNN` N_1 ... N_M `)`
''Pushes'' a sequence of numbers specified by N_i where N_i is a list of any legal address field. For example: `(PUSHNN (1) (2) (= 0))` pushes the contents of `AC1`, the contents of `AC2`, and the constant 0.

`(POPNN N)` ''Pops'' the `N` most recent numbers, discarding the values.

Use of these macros is subject to the following restrictions:

1. `PUSHN`'s and `POPN`'s must be seen by the compiler in the same order and number in which they are executed. The compiler does not analyze the code; it assumes when it encounters a `PUSHN` in the sequential processing of the code that the `PUSHN` will in fact be executed.

2. Every number that is pushed must be popped.

3. In nested `ASSEMBLE` statements, if a `PROG` or open lambda occurs between the inner and outer level `ASSEMBLE`, numbers pushed in the outer `ASSEMBLE` may not be referenced from the inner `ASSEMBLE`.

The value of a function is always returned in `AC1`. Therefore, the pseudo-function, `AC`, is available for obtaining the current contents of `AC1`. For example `(CQ (FOO (AC)))` compiles a call to `FOO` with the current contents of AC1 as argument, and is equivalent to:

```
(PUSHP)
(E (PSTEP))
(CLL (QUOTE FOO) 1)
(E (PSTEPN -1))
```

In using `AC`, be sure that it appears as the first argument to be evaluated in the expression. For example: `(CQ (IPLUS (LOC (AC)) 2))`

There are several ways to reference the values of variables in assemble code. For example:

`(CQ X)` Puts the value of `X` in AC1.

`(LDV2 (QUOTE X) SP 3)`
Puts the value of `X` in AC3.

`(SETQ X)` Sets `X` to the contents of AC1.

`(VAR (HRRM 2 , X))`
Sets `X` to the contents of AC2.

`(CQ (LOC (AC)))`
Boxes the contents of AC1.

`(FASTCALL MKFN)`
Floating boxes the contents of AC1.

(CQ (VAG X)) Puts the unboxed value of X in AC1.

(FASTCALL FUNBOX)
Gets the oating unbox of AC1.

To call a function directly, the arguments must be pushed on the parameter stack, and SP must be updated, and then the function called: e.g.,

```
(CQ (CAR X))
(PUSHP)                  (* stack first argument)
(E (PSTEP))
(PUSHQ 3.14)
(E (PSTEP))              (* stack second argument)
(CLL (QUOTE FUM) 2)      (* call FUM with 2 arguments)
(E (PSTEPN -2))          (* adjust stack count)
```

and is equivalent to:

```
(CQ (FUM (CAR X) 3.14))
```

22.9 INTERFORK COMMUNICATION IN INTERLISP-10

The functions described below permit two forks (one or both of them Interlisp-10) to have a common area of address space for communication by providing a means of assigning a block of storage *guaranteed not to move during garbage collections.*

(GETBLK N) [Function]
Creates a block N *pages* in size (512 words per page). Value is the address of the rst word in the block, which is a multiple of 512 since the block will always begin at a page boundary. If not enough pages are available, generates the error ILLEGAL OR IMPOSSIBLE BLOCK.

Note: the block can be used for storing unboxed numbers ONLY.

To store a number in the block, the following function could be de ned:

```
(SETBLOCK (LAMBDA (START N X) (CLOSER (IPLUS (LOC START) N) X]
```

Some boxing and unboxing can be avoided by making this function compile open via a substitution macro.

Note: GETBLK *should be used sparingly since several unmovable regions of memory can make it di cult or impossible for the garbage collector to nd a contiguous region large enough for expanding array space.*

(RELBLK ADDRESS N) [Function]
releases a block of storage beginning at ADDRESS and extending for N pages. Causes an error ILLEGAL OR IMPOSSIBLE BLOCK if any of the range is not a block. Value is ADDRESS .

22.10 SUBSYS

This section describes a function, SUBSYS, which permits the user to run a Tenex/TOPS-20 subsystem, such as SNDMSG, SRCCOM, TECO, or even another Interlisp, from inside of an Interlisp without destroying the latter. In particular, (SUBSYS 'EXEC) will start up a lower exec, which will print the operating system herald, followed by @. The user can then do anything at this exec level that he can at the top level, without affecting his superior Interlisp. For example, he can start another Interlisp, perform a SYSIN, run for a while, type a control-C returning him to the lower exec, RESET, do a SNDMSG, etc. The user exits from the lower exec via the command QUIT,[22] which will return control to SUBSYS in the higher Interlisp. Thus with SUBSYS, the user need not perform a SYSOUT to save the state of his Interlisp in order to use a Tenex/TOPS-20 capability which would otherwise clobber the core image. Similarly, SUBSYS provides a way of checking out a SYSOUT file in a fresh Interlisp without having to commandeer another terminal or detach a job.

While SUBSYS can be used to run any subsystem directly, without going through an intervening exec, this procedure is not recommended. The problem is that control-C always returns control to the next highest *EXEC*. Thus if the user is running an Interlisp in which he performs (SUBSYS 'LISP), and then types control-C to the lower Interlisp, control will be returned to the exec above the first Interlisp. If the user elects to call a subsystem directly, he must therefore know how it is normally exited and always exit from it that way.[23]

Starting a lower exec does not have this disadvantage, since it can *only* be exited via QUIT or POP, i.e., the lower exec is effectively ''errorset protected'' against control-C.

(SUBSYS FILE/FORK INCOMFILE OUTCOMFILE ENTRYPOINTFLG) [Function]

If FILE/FORK = EXEC, starts up a lower exec, otherwise runs <SUBSYS>system, e.g. (SUBSYS 'SNDMSG), (SUBSYS 'TECO) etc. (SUBSYS) is the same as (SUBSYS 'EXEC). Control-C always returns control to next higher *EXEC*. Note that more than one Interlisp can be stacked, but there is no backtrace to help you figure out where you are.

INCOMFILE and OUTCOMFILE provide a way of specifying files for input and output. INCOMFILE can also be a string, in which case a temporary file is created, and the string printed on it.

ENTRYPOINTFLG may be START, REENTER, or CONTINUE. NIL is equivalent to START, except when FILE/FORK is a handle (see below) in which case NIL is equivalent to CONTINUE.

The value of SUBSYS is a large integer which is a handle to the lower fork. The lower fork is *not* reset unless the user specifically does so using KFORK, described below.[24] If SUBSYS is given as its first

[22] POP on TOPS-20.

[23] Interlisp is exited via the function LOGOUT, TECO via the command ;H, SNDMSG via control-Z, and EXEC via QUIT.

[24] The fork is also reset when the handle is no longer accessible, i.e., when nothing in the Interlisp system points to it. Note that the fork is accessible while the handle remains on the history list.

argument the value of a previous call to SUBSYS,[25] it continues the subsystem run by that call. For example, the user can do (SETQ SOURCES (SUBSYS 'TECO)), load up the TECO with a big source le, massage the le, leave TECO with ;H, run Interlisp for awhile (possibly including other calls to SUBSYS) and then perform (SUBSYS 'SOURCES) to return to TECO, where he will nd his le loaded and even the TECO pointer position preserved.

Note that if the user starts a lower EXEC, in which he runs an Interlisp, control-C's from the Interlisp, then QUIT from the EXEC, if he subsequently continues this EXEC with SUBSYS, he can reenter or continue the Interlisp.

Note also that calls to SUBSYS can be stacked. For example, using SUBSYS, the user can run a lower Interlisp, and within that Interlisp, yet another, etc., and ascend the chain of Interlisps using LOGOUT, and then descend back down again using SUBSYS.

For convenience, (SUBSYS T) continues the last subsystem run.

SNDMSG, LISP, TECO, and EXEC are all LISPXMACROS (page 8.19) which perform the corresponding calls to SUBSYS. CONTIN is a LISPXMACRO which performs (SUBSYS T), thereby continuing the last SUBSYS.[26]

(KFORK FORK) [Function]

Accepts a value from SUBSYS and kills it (RESET in Tenex terminology). If (SUBSYS FORK) is subsequently performed, an error is generated. (KFORK T) kills all outstanding forks (from this Interlisp).

22.11 JFN FUNCTIONS IN INTERLISP-10

JFN stands for Job File Number. It is an integral part of the Tenex le system and is described in [Mur1], and in somewhat more detail in the Tenex JSYS manual. In Interlisp-10, the following functions are available for direct manipulation of JFNs:

(OPNJFN FILE ACCESS) [Function]

Returns the JFN for FILE. If FILE not open, generates a FILE NOT OPEN error. ACCESS = NIL, INPUT, OUTPUT, or BOTH as described in discussion of OPENP. For example, (JSYS 51Q (OPNJFN FILE) BYTE) will write a byte on a le, while (JSYS 50Q (OPNJFN FILE) NIL NIL 2) will read one byte.

(GTJFN FILE EXT V FLAGS) [Function]

Sets up a ''long'' call to GTJFN (see JSYS manual). FILE is a le name possibly containing control-F and/or <esc>. EXT is the default extension, V the default version (overriden if FILE speci es extension/version, e.g., FOO.COM;2). FLAGS is

[25]Must be the exact same large number, i.e., EQ. Note that if the user neglects to set a variable to the value of a call to SUBSYS, (and has performed an intervening call so that (SUBSYS T) will not work), he can still continue this subsystem by obtaining the value of the call to SUBSYS for the history list using the function VALUEOF, described in page 8.16.

[26]The EXEC LISPXMACRO is de ned to save its value on LASTEXEC so that subsequent EXEC commands will restart the same exec.

as described on page 17, section 2 of JSYS manual. FILE and EXT may be strings or atoms; V and FLAGS must be numbers. Value is JFN, or NIL on errors.

(RLJFN JFN) [Function]

Releases JFN. (RLJFN -1) releases all JFN's which do not specify open files. Value of RLJFN is T.

(JFNS JFN AC3 STRPTR) [Function]

Converts JFN (a small number) to a file name. AC3 is either NIL, meaning format the file name as would OPENP or other Interlisp-10 file functions, or else is a number, meaning format according to JSYS manual. The value of JFNS is atomic except where enough options are specified by AC3 to exceed atom size. In this case, the value is returned as a string.

STRPTR is an optional string pointer to be reused. In this case, the string characters are stored in an internal scratch string, MACSCRATCHSTRING, so that a subsequent call to JFNS will overwrite the characters returned by this one. The value of JFNS when STRPTR is supplied is always a string.

The following function is available in Interlisp-10 for specialized file applications:

(OPENF FILE X) [Function]

Opens FILE. X is a number whose bits specify the access and mode for FILE, i.e., X corresponds to the second argument to the Tenex JSYS OPENF (see JSYS Manual). Value is full name of FILE.

The first argument to OPENF can also be a number, which is then interpreted as a JFN. OPENF does not affect the primary input or output file settings, and does not check whether the file is already open - i.e., the same file can be opened more than once, possibly for different purposes.

Note that for almost all applications the function OPENFILE (page 6.1) provides a more convenient (and implementation independent) way of opening files.

22.12 DISPLAY TERMINALS

The value of the variable DISPLAYTERMFLG indicates whether the user is running on a display terminal or not. DISPLAYTERMFLG is used in various places in the system, e.g., PRETTYPRINT, HELPSYS, etc., primarily to decide how much information to present to the user (more on a display terminal than on a hard copy terminal). DISPLAYTERMFLG is initialized to the value of (DISPLAYTERMP), whenever Interlisp is (re)-entered, and after returning from a sysout.

(DISPLAYTERMP) [Function]

Value is T if user is on a display terminal, NIL otherwise. In Interlisp-10, DISPLAYTERMP is defined to invoke the appropriate jsys to check the user's terminal type.

22.13 THE INTERLISP-10 SWAPPER

Interlisp-10 provides a very large auxilary address space exclusively for swappable arrays (primarily compiled function definitions). In addition to the 256K of *resident* address space, this ''shadow space'' can currently accomodate an additonal 256K words, can easily be expanded to 3.5 million words, and with some further modifications, could be expanded to 128 million words. Thus, the overlay system provides essentially unlimited space for compiled code.[27]

Shadow space and the swapper are intended to be more or less transparent to the user. However, this section is included in the manual to give programmers a reasonable feeling for what overlays are like, without getting unnecessarily technical, as well as to document some new functions and system controls which may be of interest for authors of exceptionally large systems.

22.13.1 Overlays

The shadow space is a very large auxiliary address space used exclusively for an Interlisp data-type called a swappable array. The regular address space is called the ''resident'' space to distinguish it from shadow space. Any kind of resident array - compiled code, pointer data, binary data, or a hash array - can be copied into shadow space (''made swappable''), from which it is referred to by a one-word resident entity called a handle. The resident space occupied by the original array can then be garbage collected normally (assuming there are no remaining pointers to it, and it has not been made shared by a `MAKESYS`). Similarly, a swappable array can be made resident again at any time, but of course this requires (re)allocating the necessary resident space.

The main purpose and intent of the swapping system is to permit utilization of swappable arrays directly and interchangeably with resident arrays, thereby saving resident space which is then available for other data-types, such as lists, atoms, strings, etc.

This is accomplished as follows: A section of the resident address space is permanently reserved for a *swapping buffer*.[28] When a particular swappable array is requested, it is brought (swapped) in by mapping or *overlaying* the pages of shadow space in which it lies onto a section of the swapping buffer. This process is the swapping or overlaying from which the system takes its name. The array is now (directly) accessible. However, further requests for swapping could cause the array to be overlaid with something else, so in effect it is liable to go away at any time. Thus all system code that relates to arrays must recognize handles as a special kind of array, fetch them into the buffer (if not already there), when necessary check that they have not disappeared, fetch them back in if they have, and even be prepared for the second fetch to bring the swappable array in at a different place than did the first.

The major emphasis in the design of the overlay system has been placed on running compiled code, because this accounts for the overwhelming majority of arrays in typical systems, and for as much as 60% of the overall data and code. The system supports the running of compiled code directly from the

---

[27]Since compiled code arrays point to atoms for function names, and strings for error messages, not to mention the fact that programs usually have data base, which are typically lists rather than arrays, there is still a very real and finite limit to the total size of programs that Interlisp-10 can accomodate. However, since much of the system and user compiled code can be made swappable, there is that much more resident space available for these other data-types.

[28]Initially 64,512 word pages, but can be changed via the function `SETSBSIZE` described below.

swapping buffer, and the function calling mechanism knows when a swappable definition is being called, finds it in the buffer if it is already there, and brings it in otherwise. Thus, from the user's point of view, there is no need to distinguish between swappable and resident compiled definitions, and in fact CCODEP will be true for either.

22.13.2 Efficiency

Once of the most important design goals for the overlay system was that swappable code should not execute any extra instructions compared to resident code, once it had been swapped in. Thus, the instructions of a swappable piece of code are identical (except for two instructions at the entry point) to those of the resident code from which it was copied,[29] and similarly when a swappable function calls another function (of any kind) it uses the exact same calling sequence as any other code. Thus, all costs associated with running of swappable code are paid at the point of entry (both calling and returning).[30]

The cost of the swapping itself, i.e. the fetch of a new piece of swapped code into the buffer, is even harder to measure meaningfully, since two successive fetches of the same function are not the same, due to the fact that the instance created by the first fetch is almost certain to be resident when the second is done, if no swapping is done in between. Similarly, two successive PMAP's (the Tenex operation to fetch one page) are not the same from one moment to another, even if the virtual state of both forks is exactly the same - a difficult constraint to meet in itself.[31] Thus, all that can be reported is that empirical measurements and observations have shown no consistent slowdown in performance of systems containing swappable functions viz a viz resident functions.

22.13.3 Specifications

Associated with the overlay system is a datatype called a SWPARRAY, (type name SWPARRAYP), which occupies one word of resident space, plus however much of shadow space needed for the body of the array. ARGLIST, FNTYP, NARGS, GETD, PUTD, ARGTYPE, ARRAYSIZE, CHANGENAME, CALLS, BREAK, ADVISE, and EDITA all work equally well with swappable as resident programs. CCODEP is true for all compiled functions/definitions.

(SWPARRAYP x) [Function]
Analogous to ARRAYP. Returns x if x is a swappable array and, NIL otherwise.

---

[29] The relocatable instructions are indexed by a base register, to make them run equally well at any location in the buffer. The net slowdown due to this extra level of indirection is too small to measure accurately in the overall running of a program. On analytical grounds, one would expect it to be around 2%.

[30] If the function in question does nothing, e.g. a compiled (LAMBDA NIL NIL), it costs approximately twice as much to enter its definition if it is swappable as compared to resident. However, very small functions are normally not made swappable (see MKSWAPP, page 22.26), because they don't save much space, and are (typically) entered frequently. Larger programs don't exhibit a measurable slow down since they amortize the entry cost over longer runs.

[31] The cost of fetching is probably not in the mapping operation itself but in the first reference to the page, which has a high probability of faulting. This raises the problem of measuring page fault activity, another morass of uncertainty.

(SCODEP X) [Function]

Analogous to CCODEP. Returns T if X is or has a swapped compiled de nition.

(MKSWAP X) [Function]

If X is a resident array, returns a swappable array which is a copy of X. If X is a literal atom and (CCODEP X) is true, its de nition is copied into a swappable array, and it is (undoably) rede ned with the latter. MKSWAP returns X.

(MKUNSWAP X) [Function]

The inverse of MKSWAP. X is either a swappable array, or an atom with swapped de nition on its CODE property.

(MKSWAPP FNAME CDEF) [Function]

All compiled de nitions begin life as resident arrays, whether they are created by LOAD, or by compiling to core. Before they are stored away into their atom's function cell, MKSWAPP is applied to the atom and the array. If the value of MKSWAPP is T, the de nition is made swappable; otherwise, it is left resident. By rede ning MKSWAPP or advising it, the user can completely control the swappability of all future de nitions as they are created. The initial de nition of MKSWAPP will make a function swappable if (1) NOSWAPFLG is NIL, and (2) the name of the function is not on NOSWAPFNS, and (3) the size of its de nition is greater than MKSWAPSIZE words, initially 128.

(SETSBSIZE N) [Function]

Sets the size of the swapping bu er to N, a number of *pages*. Returns the previous value. (SETSBSIZE) returns the current size without changing it.

Note: Currently, the system lacks error recovery routines for situations such as a call to a swappable function which is too big for the swapping bu er, or when the size is zero. Therefore, SETSBSIZE should be used with care.