INTERNAL REPRESENTATION OF ECL DATA TYPES

5-75

William R. Conrad

Center for Research in Computing Technology

Harvard University
Cambridge, Massachusetts 02138

INTERNAL REPRESENTATION OF ECL DATA TYPES

5-75

William R. Conrad

March 1,1975

Harvard University

Center for Research in Computing Technology

Cambridge, Massachusetts

This work was supported in part by the Advanced Research Projects Agency under contract F19628-74-C-0083

INTRODUCTION

This paper presents the internal representation of ECL data objects, and defines certain of the information necessary to manage these objects. In addition, it presents, for the case of STRUCTs, an example of the method of analysis whereby such information is obtained. It is hoped that this paper will be suggestive to future implementers of higher level languages, as well as be useful to those seeking to understand ECL better.

This paper presents the state of the ECL system as of March 1,1975, and no commitment is made to retain specific details in future ECL systems.

The DDB format and structure analysis algorithm were designed by Benjamin M. Brosgol while he was at Harvard University, and have been extensively revised by the present author. Glenn H. Holloway is responsible for the design of the mode completion code. Acknowlegement is also given to Ben Wegbreit, who designed the ECL programming system, and who supervised its initial implementation.

1. Packing Schemes

1.1 Constraints

The CONTIGUOUS STORAGE AXIOM states that all compound objects directly definable in ECL reside in a contiguous block of storage during the object's lifetime. For example, a STRUCT (a nonhomogeneous compound object) is not implemented as a row of pointers to its components. Rather, each component is laid out one after another as the object is constructed in a single contiguous block of storage.

The SUBOBJECT INDEPENDENCE AXIOM states that each component of a compound object will retain its independent characteristics without reference to the enclosing object. Thus if a row of integers has been selected out of a compound object, a selection routine for a row of integers prepared independently of the definition of the compound object will be able to operate properly. This axiom complements the storage axiom by specifying that not only will an object be found in a contiguous block of storage, but that in a strong sense it will be operationally self contained.

1.2 Objects of Class BASIC

ECL is implemented on the Digital Equipment Corporation's PDP10, a 36-bit two's complement machine with 16 accumulators and index registers. A machine address is 18 bits long, and there are powerful byte manipulating instructions which make practical the packing of bytes within a word. Internally, objects are characterized by two attributes, a mode and an address. The mode of an object is commonly kept in accumulator A and its machine address in accumulator B. The concept of a mode-address pair is often referred to as an AB pair. The mode is the address of a DDB or Data Definition Block. The address may be a simple address or a byte pointer specifying the word address and the size and location of the byte within the word.

For reference, the five objects of class BASIC will be listed before going to more complex objects in the next sections.

BOOL	1 bit described by a byte pointer
CHAR	7 bits described by a byte pointer
INT	1 word described by a word address
REAL	1 word described by a word address

NONE The mode of the nonexistent object NOTHING. Internally represented by the AB pair: NONE, NIL.

1.3 Objects of Class PTR

There are four types of pointers which are listed below:

REF	1 word, LH=mode,RH=address
Full Word United	a restricted form of REF
Half Word United	18 bits pointing to spaced modes only
Simple Pointer	18 bits pointing to an object of one mode

Internally, there will be in A the pointer's mode specifying what the pointer may point to and in B the address of the pointer. At the machine word specified by the pointer's address, will be found the above described 18 or 36 bits of data concerning the object pointed to.

If the VALUE of a pointer is taken, a different object related to the pointer in the following ways is obtained. The mode of the new object is obtained from the left 18 bits of a full word united pointer or REF and its address from the right half. In the case of a simple pointer, its definition gives the single mode of objects to it can point. For half word united pointers which involve only spaced modes, the address of the new object alone will give its mode. The spaced modes are INT, REAL, ATOM, DDB, REF, DTPR, and NONE, and any object of these modes will be placed in a block of storage reserved only for objects of that particular mode as determined by a directory (the QMAP) of all blocks of storage in the system.

1.4 Objects of Class ROW

Rows are homogeneous compound objects of a definite or indefinite number of components (VECTOR and SEQ). Since all objects in ECL have a fixed number of components during their existence, the difference is whether the number of components is fixed when the mode is defined or deferred until an actual object is created. In the case of sequences, the number of components and the total length of the object in words is stored with the object. Sequences are considered to be LENGTH UNRESOLVED, and the concept is extended to include any object with length unresolved components. Otherwise the representation and packing algorithm for row objects is relatively simple. The components are simply laid out one after another in a contiguous block of storage; and, if the components are byte objects, they are laid out left to right within a word without overlaping word boundaries.

The ECL statement
COM<-CONST(SEQ(CHAR) OF %A,%B,%C,%D,%E,%F,%G)
will produce the following object:

```
Word 0   3,,7              /TL,,#components
Word 1   406050,,342212    /ASCII for A,B,C,D,E
Word 2   432160,,0         /and for F,G
```

The first word is a header word which gives the object's total length in words and the number of components. In the second word, packed from left to right, are five 7 bit bytes representing the characters A, B, C, D, and E. In the third word, are the remaining two bytes for F and G. Constructing a vector of three of these objects with the statement:
OBJ<-CONST(VECTOR(3,SEQ(CHAR)) OF COM,COM,COM)
will produce the following internal representation.

```
Word 0   12,,3             /TL,,#components
Word 1   3,,7              /TL,,#components
Word 2   406050,,342212    /ASCII for A,B,C,D,E
Word 3   432160,,0         /and for F,G
Word 4   3,,7              /TL,,#components
Word 5   406050,,342212    /ASCII for A,B,C,D,E
Word 6   432160,,0         /and for F,G
Word 7   3,,7              /TL,,#components
Word10   406050,,342212    /ASCII for A,B,C,D,E
Word11   432160,,0         /and for F,G
```

Observe that there is a header word for the whole object since it is itself length unresolved, and, also, that each component retains its own header word.

1.5 Objects of Class STRUCT

STRUCTS are nonhomogeneous compound objects and may be length resolved or unresolved. Because the components are of varying sizes and modes, selection is handled by a STRUCT Selection Table which contains entries with the mode of each component and its displacement into the struct. Also, the packing scheme is more complicated. If there are any length unresolved components, the header section comes first and consists of the total length of the object and one or more INTERNAL POINTERS to each of the length unresolved components except the first. Then come byte objects laid out from largest to smallest, (with halfword pointers before halfwords) followed by length resolved word objects in the order in which they are encountered in the mode definition. Half word pointers are specially handled in that they are packed right to left in each word so that if there is only one pointer in a word it is in the right half. This restriction is for the benefit of the compactifying garbage collector. Finally, at the end come the length unresolved components if any. Below we will build a typical STRUCT, and give its representation and selection table.

```
MODE1<-STRUCT(A:SEQ(CHAR),B:VECTOR(9,BOOL),
              C:INT,D:SEQ(BOOL))
COM1<-CONST(VECTOR(9,BOOL) OF TRUE,TRUE)
COM2<-CONST(SEQ(BOOL) OF TRUE,FALSE,TRUE)
OBJ<-CONST(MODE1 OF COM,COM1,5,COM2)
```

Below is how OBJ looks internally:

```
Word 0  10,,6           /TL,,Internal Pointer
Word 1  600000,,0       /VECTOR(9,BOOL)
Word 2  5               /INT
word 3  3,,7            /header for SEQ(CHAR)
WORD 4  406050,,342212  /A,B,C,D,E
Word 5  432160,,0       /F,G
Word 6  2,,3            /header for SEQ(BOOL)
Word 7  500000,,0       /TRUE,FALSE,TRUE
```

The header section is one word, with the total length of the object in the left half, and the the internal pointer for the second length unresolved object in the right half. The first length unresolved component, being at the end of the length resolved components, needs no internal pointer. Immediately after the header section comes the only byte component, the VECTOR(9,BOOL). Then comes the length resolved word object, the INT with value of 5. Following that comes the first length unresolved object, the SEQ(CHAR). Then at word 6, as the internal pointer says, comes the second length unresolved object, the SEQ(BOOL).

A STRUCT Selection Table is a sequence of two word STRUCTs called STRTEs. The Ith component of the selection table corresponds to the Ith component of the STRUCT, and its first word contains the mode of the component. If the sign bit of the first word is off, the content of the second word is a relative address or displacement. Otherwise, the second word contains a move instruction which fetches the internal pointer for that component. When the address of the STRUCT is added to the displacement or internal pointer, we obtain the absolute address of the component. Below is the struct table for OBJ.

```
Word 0  11,,4            /header for struct table
Word 1  13000            /DDB for SEQ(CHAR)
Word 2  3                /displacement for A
Word 3  13010            /DDB  for VECTOR(9,BOOL)
Word 4  331100,,1        /byte pointer for B
Word 5  INTP             /DDB for INT
Word 6  2                /displacement for C
Word 7  400001,,13020    /DDB for SEQ(BOOL)
Word 8  HRRZ J1,(B)      /move instruction
```

The left half of the first word of each component of a struct table is used to chain all components containing pointers so as to speed up the garbage collection trace functions for structs.

1.6 Modes of Class GENERIC

There can exist no object in ECL of a mode which is GENERIC. However, function parameters and results may have ascribed to themselves a generic mode which limits the set of allowable modes of objects to which they may be bound. Thus, a function parameter may be specified as "A:ONEOF(INT,REAL)" which will limit it to being either an INT or a REAL during a given activation of the routine.

2. Data Definition Block

2.1 Purpose and Structure

The purpose of a Data Definition Block is to factor out and collect in a convenient format all the constant information about objects of a given mode. A DDB is itself a length resolved STRUCT and, if not already builtin, it would be defined in ECL as follows:

```
DDB::STRUCT(SFLG:BOOL,HFLG:BOOL,PROCFLG:BOOL,
        EPFLG:BOOL,WDFLG:BOOL,LRFLG:BOOL,
        FINFLG:BOOL,SAFLG:BOOL,SUPUGF:BOOL,
        EUGFLG:BOOL,BNDFLG:BOOL,
        CYCFLG:BOOL,PFLG:BOOL,
        BIFLG:BOOL,QMWFLG:BOOL,ANYFLG:BOOL,
        NAME:SYMBOL,PROCD:PTR(PDESC),
        CLASS:SYMBOL,UR:MODE,
        DADV:PTR(SEQ(INT)),
        STRTB:PTR(SEQ(STRTE)),
        UFN:PTR(SEQ(REF)),SFN:HWD,
        AFN:HWD,GENFN:HWD,TRFN:HWD,
        BYPT:HWD,SZ:HWD,D:REF)
```

These fields will be discussed individually below. According to the above definition, a DDB would be layed out as follows.

```
Word 0          PROCD,,NAME
Word 1          UR,,CLASS
Word 2          DADV,,STRTB
Word 3          SFN,,UFN
Word 4          GENFN,,AFN
Word 5          BYPT,,TRFN
Word 6          FLAGS,,SZ
Word 7          D field
```

There is no header section, and the byte objects (half word pointers, half words and bools) come first followed by the only full word object, the D field which is a REF.

2.2 The DDB Fields

PROCD Field [PTR(PDESC)]

The PROC descriptor field points to a table which gives the mode and binding class for the parameters and the result mode of a procedure. This table allows strong typing of routines for the compiler.

NAME Field [SYMBOL]

The NAME field points to an atom whose print name is the name of the mode for this DDB. The relationship

MODE.NAME.SBLOCK.CONSTF=MODE

will always hold (and it establishes the name as a mode constant to the parser and mode constructing routines).

UR Field [MODE]

The UR field points to another DDB which is identical to the current DDB with the exception of a different name and possibly a different set of user defined mode functions. The UR field is used by the LIFT and LOWER subrs to determine the legality of changing an object's mode.

SFN Field [HWD]

This field contains the 18 bit machine address of the system selection function for objects of this mode.

AFN Field [HWD]

This field contains the machine address of the system assignment function for objects of this mode.

CLASS Field [SYMBOL]

This field contains a pointer to the atom for the class of modes this DDB is part of. The classes are BASIC, PTR, ROW, STRUCT, and GENERIC.

GENFN Field [HWD]

This field contains the machine address of the system generation function for objects of this mode.

TRFN Field [HWD]

This field contains the machine address of the system garbage collection trace function for objects of this mode. (This field is also used during mode completion to prevent circular mode definitions.)

BYPT Field [HWD]

This field contains the left hand side (size and position) of a byte pointer for objects of this mode. If the object is a word object, this field is zero.

SZ Field [HWD]

The SZ or size field contains the size of the object in bits if it is a byte object, or, in words, if it is a word object. If the object is length unresolved, then the size field contains the dope vector length (the number of lengths or dimensions to be resolved).

DADV Field [PTR(SEQ(INT))]

The DADV field points to a user defined apparent dope vector which can be used by a user defined generation function.

STRTB Field [PTR(SEQ(STRTE))]

The STRTB field points to the Struct Selection Table for this mode if it is of class STRUCT.

UFN Field [PTR(SEQ(REF))]

The UFN field points to a table of user defined mode functions.

D Field [REF]

The D (for descriptor) field points to one of several different tables which specify the components of compound objects.

FLAGS [BOOLs]

The flags are 1 bit components of a DDB which are selected by the names given below.

HFLG	True if halfword pointer.
SFLG	True if "simple" pointer.
PROCFLG	True if a PROC mode.
EPFLG	True if any component is a pointer.
WDFLG	True if a word object.
LRFLG	True if length resolved.
FINFLG	True if processed by mode compiler.
SAFLG	True if simple assignment object.
SUPUGF	True if user gen function is not to be called when modes are equal.
EUGFLG	True if embedded user gen function.
BNDFLG	True if all symbols bound to modes.
CYCFLG	True if mode is circular (an error).
PFLG	True if mode name to be printed with obj.
BIFLG	True if mode is builtin.
QMWFLG	True if mode is "primitive" (set by user programs).
ANYFLG	True if a generic mode with ANY as UR.

HFLG and SFLG taken together as a two bit octal number specify the four subclasses of pointers as follows.

HFLG	SFLG	Subclass	Type
0	0	0	REF
0	1	1	Full word united
1	0	2	Half word united
1	1	3	Half word simple

FINFLG indicates whether a mode has been completely processed by the mode compiler. BNDFLG indicates whether all symbolic references to modes have been replaced with actual modes everywhere in this and any associated mode.

3. DDB Extensions

3.1 Object Headers

Object headers carry object, not mode, dependent information directly with an object. For length unresolved objects of class ROW, the header is one word at the beginning of the object with its total length in words in the left half and the number of components in the right half. For length unresolved objects of class STRUCT, the header is one or more words at the beginning of the object as follows. The left half of the first word again contains the total length of the object. The right half of the first word and succeeding half words contain internal pointers (e. g. The relative address of the length unresolved components as they are actually calculated during the generation of the object.) to the second, third, fourth and following length unresolved components. There is no internal pointer for the first length unresolved component because it always is at the end of the length resolved part of the struct. The header section may end on a half word boundary, with the other half being used for byte items if any.

3.2 STRUCT Selection Table

The STRUCT Selection Table is a sequence of STRTE's, one for each component of the STRUCT that it describes. The first word of each entry contains the mode of the component in its right half, and a flag bit(the sign bit) in its left half. If the flag bit is off, the second word is a relative address or byte pointer. If it is on, then the second word is a move instruction which when executed will fetch the internal pointer for that component. For consistency, the first length unresolved component's entry will also have its sign bit set even though there is no internal pointer. In this case the move intruction will fetch the relative address directly from the address field of the instruction rather than from an internal pointer. When the object address is added to the relative address or internal pointer, we obtain the component's absolute address. If not builtin, a STRTE would be defined as below in the ECL language.

```
STRTE::STRUCT(JUNK:HWD,SBMD:MODE,RLPT:INT)
```

As mentioned previously (Section 1.5), the rightmost sixteen bits of the JUNK field are used to chain together all pointer containing component entries of a STRUCT selection table. This inexpensive procedure (the space would be unused otherwise, and the setup of the chain is done only once at mode definition time.) greatly speeds up garbage collection since only the components containing pointers may be selected for tracing.

3.3 DADV Table

The DADV table is a SEQ(INT) to be used as a default apparent dope vector by a user defined generation function for that mode. By this feature, it becomes possible, for example, to treat a list as SEQ(FORM) by giving it a default dope vector of length one and writing the appropriate generation function. The default dope vector is used in two ways. If no size specifications are given, it is copied and used as the dope vector for default generation. On the other hand, if size specifications are given, the length of the default dope vector is used to determine the number of unresolved lengths in the mode of the object to be generated. It is thus possible to treat noncompound and length resolved objects as if they were length unresolved.

3.4 STRUCT Definition Table

This table is a sequence of FDSes with name FDSRW. The left half of each entry contains a form which is either a mode or a symbol later to be associated with a mode. This is the component's mode. The right hand side points to the atom for the component's name. There will be one entry for each component of the struct which the table is to describe. If not builtin, a FDS's ECL definition would be

```
FDS::STRUCT(TYPE:FORM,SYM:SYMBOL)
```

3.5 ROW Definition Table

This table is a STRTE (to avoid defining an additional builtin mode). The right half of the first word (the SBMD field) is a form which is either a mode or symbol which will later be associated with a mode. The left half is unused. The second word (the RLPT field) is an integer which gives the number of components. For sequences, the integer always has the value of -1.

3.6 PTR Definition Table

This table is a sequence of forms (half words) which are either modes or symbols later to be associated with modes. This table is used for united pointers or generic modes. Its definition in ECL would be

```
FRMRW::SEQ(FORM)
```

3.7 PROC Definition Table

This table is used to specify the arguments and result type of a procedure in a PROC mode. Its ECL definition would be

```
PDESC::STRUCT(FORMALS:FDSRW,RETYPE:FORM)
```

The FDSRW (see 3.4) will give the mode and binding class of each parameter and the RETYPE will give the result type.

3.8 User Function Table

This table is simply a SEQ(REF). As currently implemented, it is always of size six and each REF points to a user defined mode function in the order conversion, assignment, selection, printing, generation, and dimension. These routines will be used in preference to system routines whenever any of the above tasks is to be performed.

4. ATOMs and SBLOCKs

4.1 ATOM Fields

An ATOM consists of a two word fixed part followed by a variable length ASCII print name terminated by a null byte. The print name is deliberately made inaccesible to users to preserve the integrity of the hashing algorithm used. A string corresponding to the print name may be obtained by the system routine BASIC\STRING, and a SYMBOL (a PTR(ATOM)) may be obtained by the routine HASH. The format and fields of an ATOM are given below.

```
Word 0          LINK,,SBLK
Word 1          TLB
```

The LINK field is a half word (VECTOR(18,BOOL)) which is used internally to link ATOMs together on their hash lists. The SBLK field (POINTER(SBLOCK)) points to the system information block described below. The TLB field (a REF) holds the top level, or global, binding of the symbol.

4.2 SBLOCK Fields

A SBLOCK is a two word structure which contains parsing information related to its symbol. Its format is given below.

```
Word 0          RMTCH,,PLIST
Word 1          SINFO,,CONSTF
```

RMTCH is a symbol which points to the matching bracket for matchfix operators. PLIST is a form which points to an arbitrary user specified property list. CONSTF is a form which points to a DDB and which identifies the symbol to the parser as a mode constant. The SINFO field is a half word the meaning of whose bits will be described below.

Bits	Meaning
0	1 if an operator.
1-7	Grammar terminal code as below:
1-2	Unused for non terminals.
3	1 if a left matchfix bracket.
4	1 if a right matchfix bracket.
5	1 if an identifier.
6	1 if a prefix operator.
7	1 if an infix operator.
8-15	Priority if an infix operator.
16	1 if a nofix operator.
17	1 if a right associative operator.

5. Procedure Formats

5.1 SUBRs

SUBRs are hand coded system subroutines with four components as follows.

```
BODY     SEQ(INT)
RETYPE   MODE
PRMD     MODE
FORMALS  SEQ(FDS)
```

The BODY is the actual machine language code. The RETYPE is the result type. The PRMD is the PROC mode, if any, associated with the SUBR. The FORMALS are the mode-bind class pairs of the parameters. Below is the actual layout of a SUBR.

```
Word 0              TL0,,IP0
Word 1              PRMD,,RETYPE
Word 2              TL1,,# instructions

(instructions go here)

Word 2+TL1-1        last instruction
Word IP0            TL2,,# of  FDS
Word IP0+1          mode,,bind class
Word IP0+2          mode,,bind class

 . . . . . .

Word IP0+TL2-1      mode,,bind class
```

TL0 is the total length of the SUBR in words. IP0 is the internal pointer (relative displacement) of the FORMALS. TL1 is the total length of the BODY. TL2 is the total length of the FDS sequence which makes up the FORMALS. Note that all addresses are taken relative to word 0, the CEXPR's origin.

5.2 CEXPRs

CEXPRs are compiled EXPRs as produced by the ECL compiler. They have four components as follows.

```
BODY     SEQ(INT)
FORMALS  SEQ(FDS)
RETYPE   FORM
DATAB    SEQ(REF)
```

The DATAB component is used to link separate routines together and to hold references to constants and other such data. Since CEXPRs are dumpable and subject to relocation by the compactifying garbage collector, all external references must be done through the DATAB. Additionally, the tail end of the code body contains a relocation bitmap for the machine language code preceding it for the benefit of the garbage collector and program dumper. Below is the layout of a CEXPR.

```
Word 0              TL0,,IP0
Word 1              IP1,,RETYPE
Word 2              TL1,,# instructions

(instructions go here)
(followed by a relocation bitmap)

Word 2+TL1-1        last bitmap word
Word IP0            TL2,,# of FDS
Word IP0+1          mode,,bind class

 . . . . . .

Word IP0+TL1-1      mode,,bind class
Word IP1            TL3,,# of REFs
Word IP1+1          mode,,address

 . . . . . .

Word IP1+TL3-1      mode,,address
```

IP1 is the internal pointer to the DATAB, and TL3 its length. Otherwise, the interpretations for SUBRs apply.

5.3 EXPRs

EXPRs or explicit routines are actually lists composed of dotted pairs. The definition of an EXPR and the list produced are given below. The ECL statement

```
F <- EXPR(A:INT, B:REAL; REAL) [] A+B (];
```

would give F the value

```
(EXPR ((A INT LIKE) (B REAL LIKE)) REAL (BEGIN (+ A B)))
```

By appropriate combinations of CARs and CDRs (as in LISP) the tag EXPR, the parameter specification list, the result type, and the EXPR body may be obtained. When applying an EXPR, the interpreter follows just such a procedure.

6. Stack Useage

6.1 Stack Management

There are three stacks in ECL each with its own stack pointer. They are the control stack, the name stack, and the value stack. Each has its origin and total length stored in a distinguished location, and the entries on each are controlled so that it is possible to identify their types. By virtue of this discipline and the compactifying garbage collector, it is possible to dynamically allocate more space to existing stacks without losing context. The stacks are implemented by the PDP10's hardware and are thus quite efficient. A stack pointer on the PDP10 has the following format: negative number of words left,,current address. A PUSH instruction adds one to each half of the stack pointer and puts an entry on the stack. A POP instruction removes the top entry of the stack and subtracts one from each half of the stack pointer. PUSHJ and POPJ are similar except that they store and fetch subroutine return addresses on the stack.

6.2 Control Stack Entries

The control stack is the only stack used in ECL to call routines (via a PUSHJ). In addition, it contains FORMs which are identifiable by zero left halves, junk words, identifiable, as are return addresses, by non zero left halves, and various blocks of words identifiable by unique addresses in the right halves. The format of three such blocks, or records, will be given below. Note that the stacks grow downward.

MARK records specify a specific point of control to which a program may return to. The record contains a name for identification, a mode for verification of the result being returned, value stack and name stack at the time the mark was made (they are flushed back to that point), a real return address, and the fixup routine address (unique address) which identifies the record type. The fixup routine is reached by a POPJ, flushes the top two words of the record off the stack, restores the old name and value stack, and then does a POPJ which goes to the real return. Below is the format of a MARK record.

```
VPO,,REAL RETURN
MODE,,NPO
NAME,,FIXUP ROUTINE
```

BLOCK records specify a specific control point at which a number of statements are to be executed until either the statement list is empty, or an explicit exit is requested.

In addition to the fixup and real return adddresses, and the initial name and value stack, a BLOCK record contains the name and value stack after locals have been declared. After each statment is executed, the value stack is restored if the result of the statment is not to be retained. The format of a BLOCK record is given below.

```
VPO,,REAL RETURN
STMT. LIST,,NP$
        VP$
NPO,,FIXUP ROUTINE
```

PROC records mark entry into a procedure. They contain the procedure's name, mode, and address, and nominal result mode as well as the usual initial name and value stack, fixup routine, and real return addresses. The format of a PROC record is given below.

```
VPO,,REAL RETURN
PROC. NAME,,NOM. MODE
PROC. MODE,,PROC. ADR
NPO,,FIXUP ROUTINE
```

6.3 Name and Value Stack Entries

There is only one type of name stack entry, and its format is given below.

```
NAME,,MODE
ADDRESS
```

The NAME filed is a symbol, and is used to provide dynamic scoping. When a symbol is to be evaluated, the name stack is searched backwards until a NAME is found which matches the symbol or the origin is reached. If no match is found, the top level binding of the symbol is used. The MODE field characterizes the object to be found at the ADDRESS, which may be either a full word byte pointer or just an 18-bit address with the left half zero. The MODE and ADDRESS constitute an AB pair, and the name stack is used to store objects in such a way that the garbage collector can trace and protect them. In cases where new nomenclature is not desired but the protection is, the NAME may be nil.

The value stack may be thought of as undifferentiated storage using LIFO allocation. Objects on the value stack derive their ECL meaning, if any, by virtue of being pointed to by name stack entries which, of course, give their mode. Local variables reside on the value stack.

7. Mode Compilation

7.1 Preliminary Processing

The ECL parser takes a mode definition such as

```
A <- STRUCT(FIRST: PTR(INT), SEC:REAL)
```

and produces the following list which is passed to the interpreter for evaluation:

```
(<- A (STRUCT (PTR INT)) (SEC REAL)))
```

The list structure is such that when the right hand side of the assignment is evaluated, the evaluater will take the mode constructor, e. g. STRUCT in the above example, as a procedure to be applied to the arguments represented by the remainder of the list. STRUCT, VECTOR, SEQ, PTR, ONEOF, and PROC are the builtin mode constructors in ECL. They are all bound to procedures which actually construct a mode or data type definition, and they return that mode as their result type.

7.2 Intermediate Processing

The mode compiler proper (one routine of which is discussed in section 7.3) operates on a partially complete DDB (one with just D, NAME, and CLASS fields filled in). Routines such as STRUCT take the internal list structure produced by the parser and prepare a partial DDB. This partially complete DDB is examined to see if all its component modes are complete. If so, the mode compiler is called. Otherwise, the mode's generation function is set to a routine which will attempt to complete the mode when an object of that mode is generated. With the exception of the routine ADDR which returns an absolute machine address when given the name of a system mode function, the entire mode definition facility could be written in ECL.

The routines given below handle the intermediate processing for defining modes of class STRUCT.

```
STRUCT <- EXPR(L:FORM; MODE)
      BEGIN
         DECL NAME:SYMBOL; DECL RESULT:MODE;
         DECL LEN:INT BYVAL LISTLENGTH(L);
         DECL DEF:REF BYVAL ALLOC(SEQ(FDS) SIZE LEN);

/* 'LOAD THE STRUCTURE DEFINITION TABLE WITH NAMES AND';
/* 'MODES (POSSIBLY SYMBOLIC) OF THE COMPONENTS';

         FOR I TO LEN
                REPEAT
                  DEF[I].SYM <- L.CAR.CAR;
                  DEF[I].TYPE <- EVFMS(L.CAR.CDR.CAR);
                  L <- L.CDR;
                END;

/* 'CONSTRUCT THE NAME OF THE MODE BY GOING THROUGH THE';
/* 'D FIELD COMPONENT BY COMPONENT.';

         BEGIN
            DECL PSTRING:PTR(STRING) BYVAL
               ALLOC(STRING BYVAL 'STRUCT(');
            FOR I TO LEN
                REPEAT
                     PSTRING <-
                        CONCAT(PSTRING,
                           BASIC\STR(DEF[I].SYM));
                     PSTRING <- CONCAT(PSTRING, %:);
                     PSTRING <-
                        CONCAT(PSTRING, PTYPE(DEF[I].TYPE));
                     PSTRING <- CONCAT(PSTRING, %,);
                  END;
            PSTRING[LENGTH(PSTRING)] <- %);
            NAME <- HASH(VAL(PSTRING));
         END;

/* 'SEE IF AN ALREADY DEFINED MODE ASSOCIATED WITH';
/* 'THE NAME.  IF NOT, THEN OBTAIN A NEW DDB AND';
/* 'SET THE NAME, D, AND CLASS FIELDS';

         RESULT <- GETDDB(NAME, DEF, "STRUCT");

/* 'IF FINFLG ALREADY TRUE, THEN RETURN OLD MODE';
/* 'OTHERWISE ATTEMPT TO COMPLETE NEW PARTIAL DDB';

         RESULT.FINFLG => RESULT;
         DEFDDB(RESULT);
      END;
```

```
EVFMS <- EXPR(F:FORM; FORM)
      BEGIN
         DECL T:ANY BYVAL EVAL(F);
         MD(T).CLASS # "PTR" -> ERROR('TYPE FAULT');

/* 'TRY TO OBTAIN A MODE OR SYMBOL FROM A FORM.';
/* 'IF ALREADY A MODE, THEN RETURN IT.';

         MD(VAL(T)) = DDB => T;
/* 'IF A SYMBOL, THEN TRY TO OBTAIN A MODE';

         MD(VAL(T)) = ATOM => SYTOM(T);
         ERROR('TYPE FAULT');
      END;


SYTOM <- EXPR(T:SYMBOL; FORM)
      BEGIN

/* 'IF NO ASSOCIATED MODE, THEN RETURN THE SYMBOL';

         T.SBLK = NIL OR MD(VAL(T.SBLK.CONSTF)) # DDB => T;

/* 'OTHERWISE RETURN ASSOCIATED MODE';

         T.SBLK.CONSTF;
      END;


GETDDB <- EXPR(N:SYMBOL, D:REF, CLS:SYMBOL; MODE)
      BEGIN
         DECL NEWMD:MODE;

/* 'IF MODE ASSOCIATED WITH NAME, RETURN IT';

         N.SBLK # NIL AND MD(VAL(N.SBLK.CONSTF)) = DDB =>
            N.SBLK.CONSTF;

/* 'OTHERWISE GET A NEW DDB AND SET NAME, CLASS, D FIELDS';

         NEWMD <- ALLOC(DDB); NEWMD.D <- D;
         NEWMD.CLASS <- CLS; NEWMD.NAME <- N;

/* 'MAKE SURE NAME IS NOW ASSOCIATED WITH THIS MODE';

         N.SBLK = NIL -> N.SBLK <- ALLOC(SBLOCK);
         N.SBLK.CONSTF <- NEWMD;
      END;
```

```
DEFDDB <- EXPR(NEWM:MODE; MODE)
      BEGIN
         DECL FIN, BND:BOOL BYVAL TRUE;

/* 'CALL CMPDDB TO ATTEMPT COMPLETION OF THE MODE';
/* 'IF POSSIBLE, CMPDDB WILL CALL THE MODE COMPILER';
/* 'IF SUCCESSFUL, RETURN THE COMPLETED MODE';

                CMPDDB(NEWM) => NEWM;

/* 'OTHERWISE SET THE NEW MODE'S GENERATION FUNCTION';
/* 'TO BE  A COMPLETE-ME FUNCTION WHICH WILL ATTEMPT TO';
/* 'COMPLETE THE MODE WHEN AN OBJECT IS GENERATED';

         NEWM.GENFN <- ADDR(CMPFN);
         NEWM;
      END;

PTYPE <- EXPR(F:FORM; STRING)
      BEGIN
         DECL PS:PTR(STRING) BYVAL ALLOC(STRING BYVAL '"');
         DECL S:SYMBOL;
         DECL MDV:MODE BYVAL MD(VAL(F));

/* 'OBTAIN PRINT NAME FOR A COMPONENT MODE.';
/* 'GET SYMBOL WHOSE PRINT NAME WILL BE USED BELOW';

         MDV = ATOM -> S <- F;
         MDV = DDB -> S <- F.NAME;

/* 'IF NOT EXTENDED MODE, JUST USE PRINTNAME';

         MDV # ATOM AND F.UR = NIL => BASIC\STR(S);

/* 'OTHERWISE USE QUOTED PRINTNAME';

         PS <- CONCAT(PS, BASIC\STR(S));
         VAL(CONCAT(PS, %"));
      END;
```

7.3 Final Processing

The final step in mode processing consists of a detailed analysis of the structure of the mode and its component modes. Various fields of the DDB are filled in with information obtained from the analysis, and the proper system mode functions for assignment, selection, genration, and garbage collection tracing are set. The routine below builds the structure selection table, picks the correct selection function for the mode, and sets the LRFLG, SAFLG, WDFLG, and SZ field. Almost all of the work is devoted to building the structure selection table since there are only two selection functions possible for a structure. For row modes (VECTORs and SEQs) with their homogeneous components, more effort is devoted to the choice of the proper system selection function since the algoritm of the selection function itself exemplifies the results of the analysis. Similar routines exists for the setting of assignment and generation functions. It is hoped that the sample given below will illustrate the type of analysis used.

```
/* 'BUILD STRUCT SELECTION FUNCTIONS';

BLDSSF <- EXPR(; NONE)
      BEGIN
         DECL TMP:MODE;
         DECL WFLG:BOOL;
         DECL J, LINK, LINK1:INT;
         DECL LEN:INT BYVAL LENGTH(AB.D);
         DECL STR:REF BYVAL ALLOC(SEQ(STRTE) SIZE LEN);
         DECL WRK:VECTOR(LEN, INT);
         DECL CHAINS:VECTOR(WRDSIZ+2, INT);
         DECL CLRI:INT BYVAL WRDSIZ+1;
         DECL CLUI:INT BYVAL WRDSIZ+2;

/* 'CALCULATE THE NUMBER OF LENGTH UNRESOLVED ITEMS';
/* 'AND SET LRFLG, SAFLG, AND EPFLG.';

         FOR I TO LEN
                REPEAT
                  WRK[I] <- WRDSIZ;
                  TMP <- AB.D[I].TYPE;
                  NOT TMP.LRFLG ->
                     [) NLUI <- NLUI + 1;
                          AB.SAFLG <- TMP.SAFLG (];
                  TMP.EPFLG -> AB.EPFLG <- TRUE;
               END;
         NLUI = 0 -> AB.LRFLG <- TRUE;
         NLUI GT 1 -> AB.SAFLG <- FALSE;
```

```
/* 'RESERVE SPACE FOR TOTAL LENGTH IF ANY LUIS';

        NLUI # 0 -> WRK[1] <- WRK[1] - IPTRSZ;

/* 'PUT MODES FROM STRUCTURE DEFINITION INTO STRUCT TABLE'
/* 'CHAIN ALL BYTE ITEMS BY SIZE AND ALL LENGTH RESOLVED';
/* 'AND LENGTH UNRESOLVED ITEMS. CHAINS[LINK]';
/* 'CONTAINS THE FIRST ENTRY FOR A CHAIN OF SIZE LINK-1';
/* 'IF LE WRDSIZ. OTHERWISE, IT MARKS THE CHAIN FOR LR';
/* 'ITEMS OR LU ITEMS.';

        FOR I TO LEN
              REPEAT
                TMP <- STR[I].SBMD <- AB.D[I].TYPE;
                LINK <- CLUI;
                TMP.LRFLG ->
                   BEGIN
                      TMP.WDFLG =>
                         [) WFLG <- TRUE; LINK <- CLRI (];
                      LINK <- TMP.SZ+1;
                      BITSZ <- BITSZ + TMP.SZ;
                   END;
                STR[I].RLPT <- CHAINS[LINK];
                CHAINS[LINK] <- I;
             END;
       BITSZ GE WRDSIZ -> WFLG <- TRUE;

/* 'ACCUMULATE DOPE VECTOR LENGTH FOR ALL LENGTH';
/* 'UNRESOLVED ITEMS. MAKE INTERNAL POINTERS FOR ALL BUT';
/* 'FIRST ITEM WHICH DOES NOT NEED ONE.';

        (LINK <- CHAINS[CLUI]) # 0 ->
           BEGIN
              DVL <- STR[LINK].SBMD.SZ;
              LINK <- STR[LINK].RLPT;
                    REPEAT
                       LINK = 0 => NOTHING;
                       DVL <- DVL +
                        (1 <- STR[LINK].SBMD.SZ);
                       FOR J SHARED FROM 0
                          REPEAT
                            WRK[J + 1] GE IPTRSZ
                            =>NOTHING
                          END;
                       WRK[J + 1] <- WRK[J + 1] - IPTRSZ;
                       LINK1 <- STR[LINK].RLPT;
```

```
/* 'MAKE AN INTERNAL POINTER FETCHING INSTRUCTION';

                        STR[LINK].RLPT <- MAKIPT(J);
                        LINK <- LINK1;
                    END;
                END;

/* 'PROCESS BYTE ITEMS FROM LARGEST TO SMALLEST,';
/* 'ALL OF A GIVEN SIZE AT THE SAME TIME';
/* 'WRK[J+1] GIVES NUMBER OF FREE BITS IN J TH WORD';
/* 'OF STRUCT BEING LAID OUT';

        FOR I TO WRDSIZ
            REPEAT
                DECL BYTSIZ:INT BYVAL WRDSIZ - I;
                LINK <- CHAINS[BYTSIZ + 1];
                    REPEAT
                      LINK = 0 =>NOTHING;
                      FOR J SHARED FROM 0
                        REPEAT
                          WRK[J + 1] GE BYTSIZ
                          => NOTHING;
                        END;
                       WRK[J + 1] <- WRK[J + 1] - BYTSIZ;
                       LINK1 <- STR[LINK].RLPT;

/* 'MAKE A BYTE POINTER FOR THIS ENTRY USING COMPONENT';
/* 'SIZE, LEFT TO RIGHT PACKING, AND NUMBER OF BITS LEFT';
/* 'IN J TH WORD TO DETERMINE POSITION';

                        STR[LINK].RLPT <- MAKBYT(J);
                        LINK <- LINK1;
                    END;
            END;

/* 'COUNT WORDS USED FOR INTERNAL PTRS AND BYTE ITEMS';

        FOR J SHARED FROM 0
          REPEAT
            WRK[J + 1] = WRDSIZ => NOTHING;
          END;
```

7.4 Mode Functions

By an analysis of the mode definition which it is processing, the mode compiler selects from a small set of system functions, routines for assignment, selection, generation, and garbage collection tracing of objects of that mode. These routines are tailored to the structural characteristics of the object which they will process, but they will obtain specific information such as the size of an object from the given object's DDB. For example, there are assignment functions for byte objects, one word objects, and multi-word objects. A generation function for an object containing pointers must properly initialize them, while one for non-pointer objects can simply zero the object. Selection routines must know whether components are byte or word objects, and whether they are length resolved or not. Because of the one-time-only analysis which is done at mode definition time much overhead is saved during actual program execution. By way of illustration, the selection functions for structures are described.

The first selection function handles byte structures, and from this fact it knows that the structure and all its components are length resolved and byte objects. Using the integer index given it, the proper displacement in the structure selection table is obtained for the component to be selected. From that entry are obtained the mode and additive refinement of the component. By adding the refinement to the byte pointer for the whole structure, a byte pointer for the component is obtained. The mode and byte pointer are then returned as an AB pair (see section 1.2). In machine language the entire routine takes about 16 instructions including the selection table address setup and index range checking.

The second routine handles arbitrary structures and thus must check the sign bit of the first word of the structure selection table entry for the component. If it is on, the component is length unresolved, and the second word must be executed to fetch an internal pointer. Adding the internal pointer to the address of the structure produces the address of the component. If the sign bit is not on, then the second word contains the relative displacement directly. This routine also takes about 16 words.

7.5 Mode Completion

As mentioned previously, it may not be possible to completely process a mode at the time of its definition because of symbolic references to the component modes used in the mode definitition. Either an explicit call to the COMPLETE subr or an attempt to generate an object of an incomplete mode will result in a call to the routine CMPDDB given below. CMPDDB is also called by DEFDDB in an attempt to complete a new mode at the time of its creation. The DEPTH field used below is identical to the TRFN field which is set later in the process of mode completion.

```
CMPDDB <-
  EXPR(M:MODE; BOOL)
  BEGIN
    DECL DEPTH:INT;
    DECL CRCFLG, UCMFLG, BOUND:BOOL;
    M.FINFLG => TRUE;
    BOUND <- TRUE;
    BIND(M);
    UCMFLG => ERROR("UNRESOLVED MODE IN MODE DEF");
    CRCFLG => ERROR("CIRCULAR MODE DEF");
    BOUND -> FINISH(M);
    BOUND;
  END;

BIND <-
  EXPR(M:MODE; NONE)
  BEGIN
    M.FINFLG => NOTHING;
    M.CYCFLG => DEPTH = M.DEPTH -> CRCFLG <- TRUE;
    M.CYCFLG <- TRUE;
    M.DEPTH <- DEPTH;
    M.CLASS = "PTR" -> DEPTH <- DEPTH + 1;
    ITERATE(M, BIND1);
    M.CYCFLG <- FALSE;
    DEPTH <- M.DEPTH;
  END;
```

```
BIND1 <-
  EXPR(F:FORM SHARED, M:MODE;NONE)
  BEGIN
    MD(VAL(F)) = ATOM AND
      MD(VAL(F <-EVFMS(F))) = ATOM => BOUND <- FALSE;
    F.CLASS = "GENERIC" AND M.PROCD = NIL -> UCMFLG <- TRUE;
    BIND(F);
  END;

FINISH <-
  EXPR(M:MODE; NONE)
  BEGIN
    ITERATE(M, FINISH2);
    M.FINFLG => NOTHING;
    FINISH1(M);
  END;

FINISH1 <-
  EXPR(M:MODE BYVAL; NONE)
  BEGIN
        REPEAT
        (M.FINFLG OR
           BEGIN
             M.FINFLG <- TRUE;

/* 'ANALYZE MODE AND SET MODE FUNCTION FIELDS';

             MODE\COMPILER(M);

             (M <- M.UR) = NIL;
           END;)=>NOTHING;
        END;
  END;

FINISH2 <-
  EXPR(F:FORM SHARED, M:MODE; NONE)
  BEGIN
    F.FINFLG => NOTHING;
    F.CLASS = "PTR" -> FINISH1(F);
    FINISH(F);
  END;
```

The routine ITERATE maps a routine over the components of a compound mode. The mapped routine takes a component form (mode or symbol) and the parent mode as arguments.

```
ITERAT <- EXPR(A:MODE, B:ROUTINE; NONE)
      BEGIN
         DECL MT:INT;

 /* 'SET MT=-1 FOR PROC, 1 FOR PTR(M1, M2...),'
 /* 'ZERO OTHERWISE'

         A.CLASS = "STRUCT" =>
            [] FOR I TO LENGTH(VAL(A.D))
               REPEAT B(A.D[I].TYPE) END; [];
         A.CLASS = "ROW" => B(A.D.SBMD);
         A.CLASS = "GENERIC" =>
            FOR I TO LENGTH(VAL(A.D))
                REPEAT B(A.D[I]) END;

 /* 'MUST BE A POINTER';
 /* 'IF PROC MODE, ITERATE OVER PROCD COMPONENTS';

         A.PROCFLG =>
            BEGIN
               MT <- - 1;
               B(A.PROCD.RETYPE);
               FOR I TO LENGTH(A.PROCD.FORMALS)
                  REPEAT B(A.PROCD.FORMALS[I].TYPE) END;
            END;

/* 'IF A SIMPLE   POINTER, CHECK ONE MODE GIVEN BY D FIELD';

         MT <- 1;
         MD(VAL(A.D)) = DDB => B(A.D);

/* 'OTHERWISE CHECK ALL COMPONENT MODES';

         FOR I TO LENGTH(VAL(A.D))
                REPEAT B(A.D[I]) END;
      END;
```