CYBIL HANDBOOK

60457290 01

REVISION RECORD

REVISION		DESCRIPTION
01	(09-23-83)	Preliminary manual released.

Address comments concerning this manual to:

Control Data Corporation
Software Engineering Services
4201 North Lexington Avenue
St. Paul, Minnesota 55112

60457290 01

Table of Contents

1.0 APPLICABLE DOCUMENTS

The following documents should prove to be helpful in your CYBIL development.

1.1 GENERAL

- CYBIL Language Specification (ARH2298)
- This CYBIL Handbook (ARH3078)
- CYBIL Formatter ERS (ARH2619)

1.2 C170

- CYBIL I/O ERS (ARH2739)
- CYBIL Debugger ERS (ARH3142)
- SES User Handbook (ARH1833)
- SES Miscellaneous Routines Interface (SESD003)

2.0 COMMON CYBIL COMPILER FRONT END

This section details the characteristics of all CYBIL compilers.

2.1 INLINE PROCEDURES IMPLEMENTATION

The CYBIL Language Specification lists language considerations for inline procedures. Listed below are specific features of the inline procedure implementation:

- Local variable declarations in an inline procedure become part of the calling procedure's stack frame.

- Formal parameters are treated as local variable declarations in the inline procedure. At the point of call to an inline procedure the actual parameter is assigned to the corresponding formal parameter local variable. Reference parameters are accessed by assigning a pointer to the actual parameter to the formal parameter local variable.

- When the actual parameter for a value parameter is of an adaptable type or is a substring then the parameter is treated as though it were a read-only reference parameter, i.e. a local copy of the parameter is not created. This is necessary to allow type-fixing at execution time. A restriction is imposed on adaptable array/record value parameters that the actual parameter be aligned to a machine addressable boundary.

- Nested calls to inline procedures are arbitrarily limited to 5 levels of nesting on the assumption that an inappropriate amount of code expansion may be occurring when the nesting level becomes too great. Excessive call nesting levels and recursive calls are considered errors and terminate inline substitution.

- Source statements in an inline procedure body are not listed at the point of call to an inline procedure.

- Inline procedures may be used with the interactive debugger. The debugger considers an inline procedure call expansion to be a series of statements on the same line as the procedure call. Local variables declared in an inline procedure may not be accessible directly by name following

an inline procedure call since the substitution process can result in the creation of non-unique variable names. Variable names in the calling procedure will always take precedence for the debugger.

2.2 SOURCE LAYOUT CONSIDERATIONS

If a source text line contains non-blank characters beyond the column specified for the right source margin then a '| ' character string is inserted in the source listing line after the right margin. This is done to indicate the end of the compiler's scan should a source text line erroneously exceed the designated right margin.

3.0 CYBIL-CC DATA MAPPINGS

3.0 CYBIL-CC DATA MAPPINGS

The actual CYBER 60-bit word formats of each of the CYBER 170 CYBIL data types is described below. This information will provide some insight into the amount of storage required for various CYBIL data structures. This will allow the user to predict the storage efficiency of his program. Unpacked data types provide for more efficient data access at the expense of storage efficiency. Packed data types provide for more efficient storage utilization at the possible expense of access time and extra code. When data (or a field of data) is aligned it will be placed on a CYBER 60-bit word boundary. Unused fields are not necessarily zeros and should not be altered by the (assembly language) programmer.

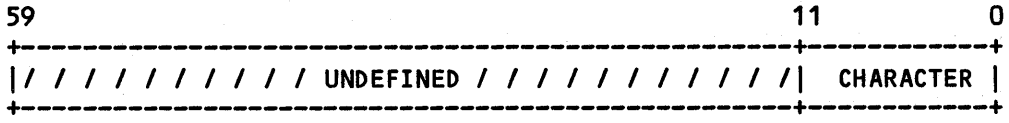
3.1 UNPACKED BASIC TYPES

3.1.1 UNPACKED INTEGER

The unpacked integer format consists of one 60-bit word. The integer value is limited to the rightmost 48 bits of the word. Ones's complement data representation is used. Integer values are therefore restricted to -(2**48 - 1) <= INTEGER <= (2**48 - 1) or -281474976710655 <= INTEGER <= 281474976710655. In the diagram below, SIGN indicates sign extension. This field will be all zero's if the integer is positive and all one's if the integer is negative.

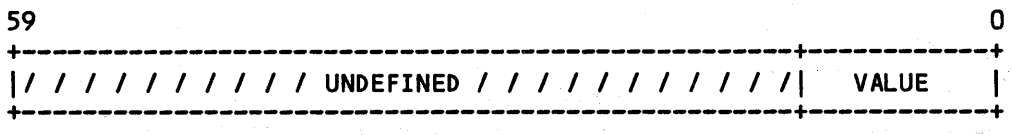
3.1.2 UNPACKED CHARACTER

The unpacked character format consists of one 8-bit ASCII character right justified in the rightmost 12 bits of one 60-bit CYBER word. Bit positions 11 through 8 are always zero. The remaining 48 bits of the word are unused. This format provides for the most efficient data access of characters at the expense of storage efficiency. The ASCII data representation is used. For example, an unpacked character 'A' would be represented as XXXXXXXXXXXXXXXX0101 (octal), 65 (decimal). The X's indicate unused bit positions.



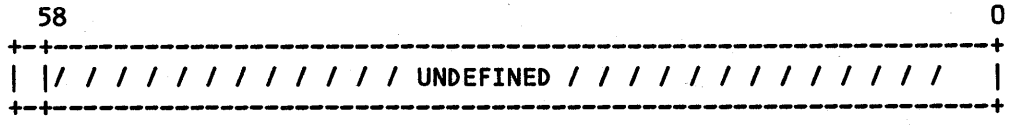
3.1.3 UNPACKED ORDINAL

An unpacked ordinal is represented as a positive integer value in the rightmost bits of a 60-bit word. The integer value designates the current ordinal value. The number of bits required to represent an ordinal of N elements is: ceiling(log2(N)). For example, an ordinal containing 10 decimal elements would require ceiling(log2(10)) or 4 bits.



3.1.4 UNPACKED BOOLEAN

An unpacked boolean type will occupy one 60-bit word. Only one bit (the sign bit) is used. The other 59 bits are unused. A sign bit of 1 indicates the boolean value true. A sign bit of 0 indicates the boolean value false.



3.1.5 UNPACKED SUBRANGE

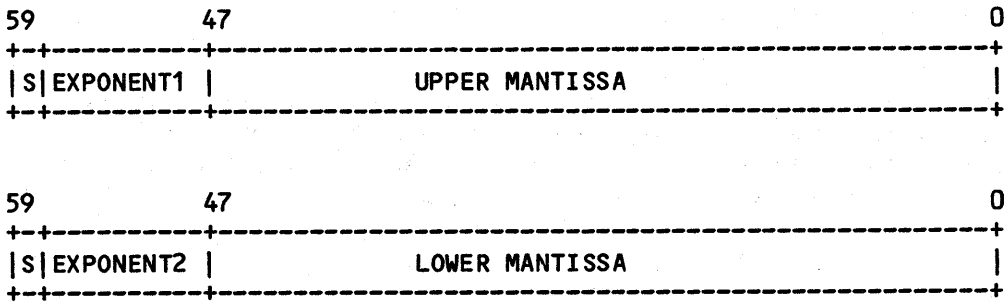
An unpacked subrange of any scalar type is represented in the same manner as the scalar type of which it is a subrange.

3.1.6 UNPACKED REAL

The unpacked real format consists of one 60-bit word. The mantissa is located in the right most 48 bits of the word. The sign is located in bit 59, and the biased exponent occupies the next 11 bits. One's complement data representation is used. Real values are limited in magnitude to the range of 6.2630*10**(-294) to 1.2650*10**322, or zero.

3.1.7 UNPACKED LONGREAL

The unpacked real format consists of two adjacent 60-bit words. The format of each word is the same as the format of a real number. The first word contains the most-significant half of the mantissa, the exponent and the sign of the number. The second word contains the least-significant half of the mantissa, an exponent 48 less than that in the first word, and the same sign as in the first word. Longreal values are limited in magnitude to the range 6.2630*10**(-294) to 1.2650*10**322, or zero.

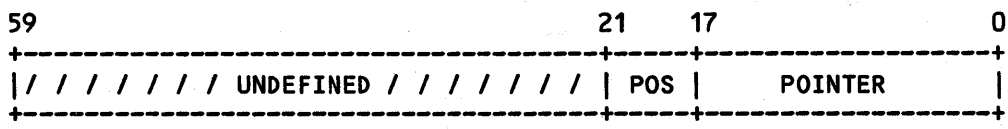


3.1.8 POINTER TO FIXED TYPES

Pointers to fixed types (excluding strings, fixable types, procedure types and sequence types) occupy the rightmost 18 bits of a 60-bit word. For all pointer types, the NIL pointer is represented as an 18 bit field with the rightmost 17 bits all ones. In the specific example of the direct pointer to fixed types a NIL pointer would have the data representation XXXXXXXXXXXXXX377777 octal where the X's indicate unused bit positions.

3.1.9 POINTER TO STRING

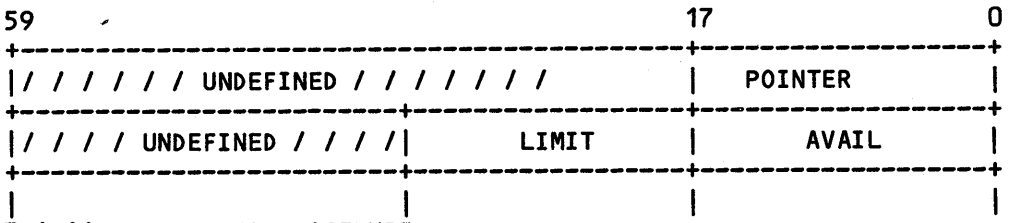
Pointers to strings are 18 bits long but have an additional 4 bit "position" field to indicate which of the ten positions (POS) in a CYBER word contains the first character of the string. A string may begin on any 12 bit boundary (bit positions 59,47,35,23, or 11). The POS field will contain a value (0,2,4,6, or 8) indicating the starting position of the string. For example, a POS value of 0 indicates that the string begins in the leftmost (bit 59) position of the word pointed to.



3.1.10 POINTER TO SEQUENCE

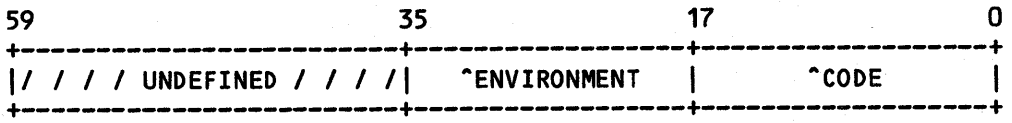
Pointers to sequences contain the pointer plus an additional descriptor word. This descriptor word contains an offset to the next available (AVAIL) location in the sequence and an offset to the top (LIMIT) of the sequence.

3.0 CYBIL-CC DATA MAPPINGS
3.1.10 POINTER TO SEQUENCE



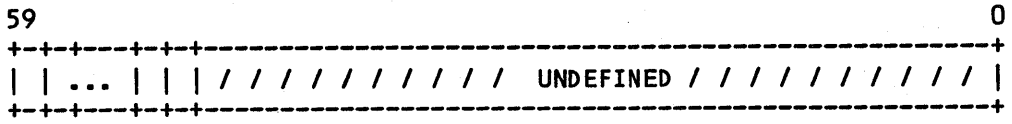
3.1.11 POINTER TO PROCEDURE

Pointers to procedures are 36 bits long. Two 18 bit pointers are contained in the 36 bit field. One of the pointers points to the code and the other pointer points to the environment (stack) of the procedure. For the outermost procedures, the ^Environment is equal to zero.



3.1.12 UNPACKED SET

An unpacked set will be left justified in the word or words it occupies. One bit is required for each member in the set. A bit set to one indicates that the set member is present. A zero bit indicates the set member is absent. If all the bits associated with a set are zero the representation is of an "empty set". For example, a set of 75 members will occupy two 60-bit words (120 bits). The leftmost 75 bits of the 120 bit field will be used to represent the set. The maximum size allowed for a set is 32,768 elements.



3.1.13 UNPACKED STRING

Unpacked strings will be 12 bits per character, five characters per word, left justified in the word or words they occupy. The data representation is the ASCII encoding (8 bits) right-justified within a field of 12 bits.

3.0 CYBIL-CC DATA MAPPINGS
3.1.13 UNPACKED STRING

3.1.14 UNPACKED ARRAY

An unpacked array is a contiguous list of aligned instances of its component types. A two dimensional array is thought of as a one dimensional array of components which are one dimensional arrays. This structure is continued for multi-dimensional arrays. Storage for the array is mapped such that the right-most (inner-most) array is allocated contiguous storage locations. Considering the typical two dimensional array consisting of "rows and columns" the data mapping would be by rows. The maximum number of elements in an array is 262143. In general, there mut be sufficient storage to contain the array.

3.1.15 UNPACKED RECORD

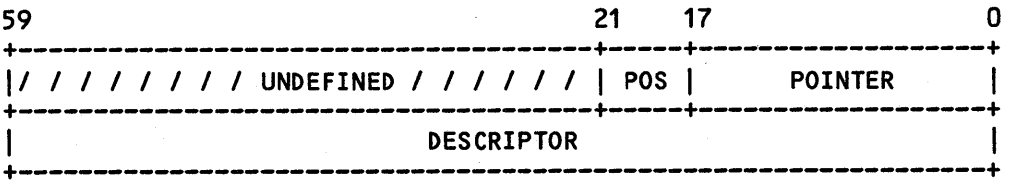
An unpacked record is a contiguous list of aligned fields.

3.2 OTHER TYPES

3.2.1 ADAPTABLE POINTERS

Pointers to adaptables are identical to pointers to the corresponding non-adaptable type with the addition of descriptors giving the length of the structures. In order to determine the size of an adaptable pointer a scan is made of the target type and all its contained types.

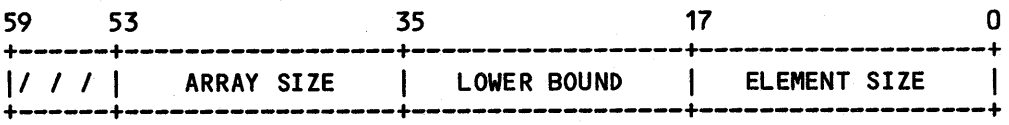
3.0 CYBIL-CC DATA MAPPINGS
3.2.1 ADAPTABLE POINTERS



The POS field is used only for adaptable strings as described above in the discussion on Direct Pointer to String.

3.2.1.1 Adaptable Array Pointer

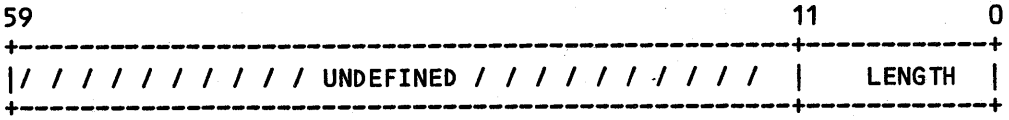
The descriptor for an adaptable array is:



The ARRAY and ELEMENT SIZE fields are either both in bits, or both in words. The value for the sizes are in bits when the array is packed and is in words when the array is unpacked.

3.2.1.2 Adaptable String Pointer

A pointer to an adaptable string will have a descriptor word. The descriptor will contain the length of the adaptable string in 6 bit quantities (i.e., twice the number of characters) as shown below:

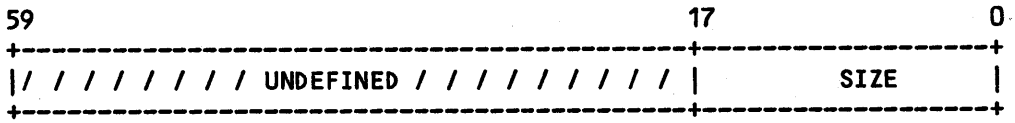


3.2.1.3 Adaptable Sequence Pointer

A pointer to an adaptable sequence will have the same format as the pointer to a fixed size sequence, as described above.

3.2.1.4 Adaptable Heap Pointer

A pointer to an adaptable heap will have one descriptor word. This word will contain the total size of the space allocated (in words) as shown below:

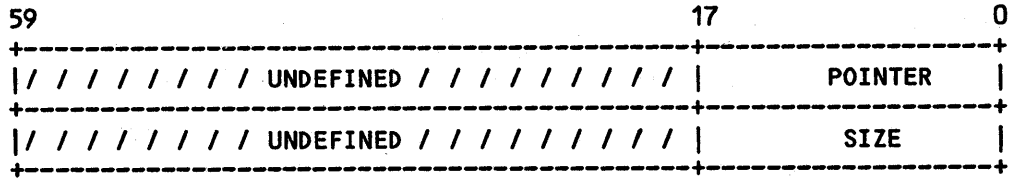


3.2.1.5 Adaptable Record

An adaptable record may have at most one adaptable field. A pointer to an adaptable record requires a descriptor word for the adaptable field. Since the adaptable field must be one of the above types, the descriptor will be as described above.

3.2.2 BOUND VARIANT RECORD POINTERS

A pointer to a bound variant record will consist of a pointer to the record followed by a descriptor word which contains the size of the particular bound variant record in use.



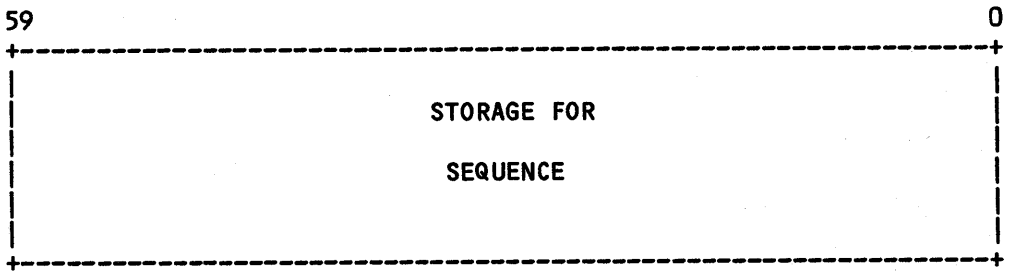
3.2.3 STORAGE TYPES

The amount of storage required for any user declared storage type (sequence or heap) may be determined by summing the #SIZE of each span plus, in the case of user heaps, some conrol information.

3.2.3.1 Sequences

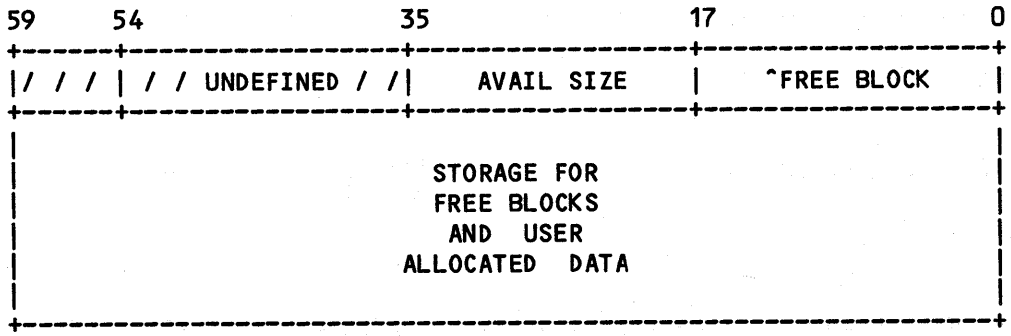
Access to a sequence is through the control information associated

with the pointer to sequence. The layout of the sequence is shown below:



3.2.3.2 Heaps

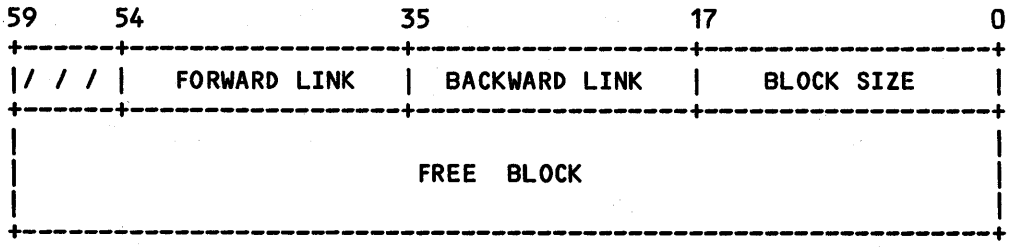
User declared heap storage must be managed differently than the sequence because explicit programmer written ALLOCATE's and FREE's may be executed. The heap, in general, consists of 1) a header word, 2) free areas (blocks) which are linked together (forward and backward) and 3) areas in use as a result of explicit ALLOCATE statement(s). For the heap data type, one additional header word is added for each repetition count for each span specified. The heap with its header word is illustrated below:



3.2.3.2.1 FREE BLOCKS

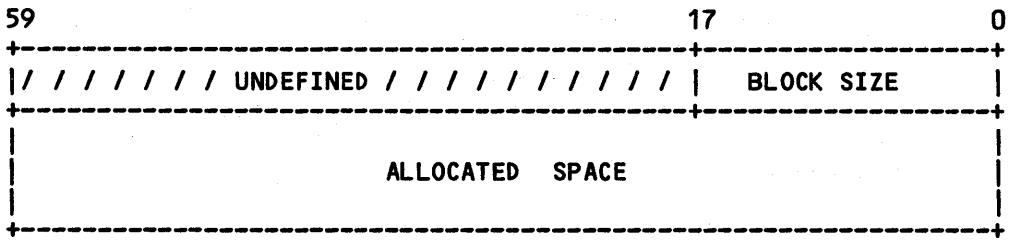
The free blocks are a circular forward and backward linked list. Free blocks are condensed each time the user code executes a FREE statement referencing this heap. The storage map of a typical free block is shown below:

3.0 CYBIL-CC DATA MAPPINGS
3.2.3.2.1 FREE BLOCKS



3.2.3.2.2 ALLOCATED BLOCKS

When the CYBIL program executes an ALLOCATE statement the free block chain is re-arranged to make room for the allocated space in the heap. For each ALLOCATE a one word header is added to the space to maintain the size of the allocated area. This size information is used to verify subsequent FREE statements. The format of an allocated area in the user declared heap is:



3.2.4 CELLS

A cell is allocated a word and is always aligned.

3.3 PACKED DATA TYPES

Packed data types are provided to allow the programmer to conserve storage space at the possible expense of access time. The choice is easily made by the programmer by simply using the 'PACKED' attribute in the declaration of the structured type.

A packed integer occupies a 60 bit word.

A packed character is 8 bits (ASCII encoded).

A packed boolean is 1 bit.

A packed set occupies as many bits as there are elements in the

set.

A packed ordinal of N elements is as long as the packed subrange 0..N-1.

A packed subrange of any type except integer is as long as the packed type of which it is a subrange.

A packed subrange of integers a..b has its length computed as follows: If a is >= 0, then ceiling(log2(b+1)), else 1+ceiling(log2(max(abs(a),b)+1)).

A packed real occupies a 60 bit word.

A packed longreal occupies two consecutive 60 bit words.

A packed string is the same as an unpacked string except that it is aligned on a 12 bit boundary instead of a word boundary.

A packed array is a contiguous list of unaligned instances of its packed component type with the length of the component type increased by the smallest number of bits that will make the new length an even divisor of 60 or a multiple of 60 bits; such that the array will fit in an integral number of 60 bit words.

The length of a packed record is dependent upon the length and alignment of its fields. The representation of a packed record is independent of the context in which the packed record is used. In this way, all instances of the packed record will have the same length and alignment whether they be variables, fields in a larger record, elements of an array, etc. When the ALIGNED clause is used on a field within a packed record, the field will be aligned to the next word boundary.

A packed pointer to fixed type requires 18 bits. A packed pointer to an adaptable type would require 120 bits. A packed pointer to procedure requires 36 bits.

Storage types (heaps and sequences) require as much space as the sum of the space requirements for each span as if it were defined as an unpacked array.

A packed cell is allocated a word and is always aligned.

3.4 SUMMARY

TYPE	SIZE	ALIGNMENT	
		UNPACKED	PACKED
BOOLEAN	bit	LJ word	bit
INTEGER	word	word	word
SUBRANGE	as needed	RJ word	bit
ORDINAL	as needed	RJ word	bit
CHARACTER	12 bits/ 8 bits	RJ word	bit
REAL	word	word	word
LONGREAL	2 words	word	word
STRING	n * 12 bits	LJ word	12 bit
SET	as needed	LJ word	bit
ARRAY/RECORD	component dependent	word	unaligned components
FIXED POINTER	18 bits	RJ word	bit
CELL	word	word	word

Note: The abbreviations LJ and RJ in the above table stand for left and right justification.

4.0 CYBIL-CC RUNTIME ENVIRONMENT

4.1 STORAGE LAYOUT OF A CYBIL-CC PROGRAM

The first 101(8) words are (as always on CYBER) the job communication area, which is described in the appropriate reference manual. The following storage area comprises the static part (code and static data) of the program. Usually it starts with the modules loaded from the load file(s) (in the order of the LOAD requests), followed by the modules from the library. The following storage area, the dynamic area starts immediately after the static area and is controlled by the memory manager. It contains:

- The stack.
- Dynamically allocated memory.

The dynamic area is capable of expanding and, if necessary, the memory manager incrementally extends the field length up to the system permitted maximum.

4.2 REGISTER USAGE

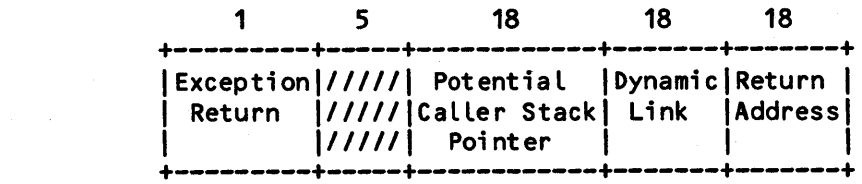
B0 = 0
B1 = 1
B2 = dynamic link - callers stack frame pointer (top of stack)
B3 = stack segment limit
B4 = static link - set before a nested procedure is called
B5 = pointer to extended parameter list

X1
X2
X3 last 5 parameters passed to callee, starting with X1
X4
X5

X1 = on return from callee must contain the linkage word
X7 = linkage word passed to callee

X6 = The function result if the value is one word or less; otherwise it is a pointer to the function value and the actual value is built in the callee's stack frame. The caller must save it before any other stack activity (procedure/function calls, or PUSH statements) takes place.

4.3 LINKAGE WORD



The linkage word is identical to the first word of the stack (the stack header), which if expressed in CYBIL syntax would be:

```
TYPE
  stack_header: PACKED RECORD
    exceptional_return: boolean,
    filler: 0..1F(16),
    potential_caller_stkp: pointer,
    dynamic_link: pointer,
    return_address: address,
    RECEND;
```

The meaning of the fields is as follows:

EXCEPTIONAL_RETURN: This field is set whenever after the procedure received control, a new stack segment was acquired. It is not used by the stack manager, but is meant as an aid for post mortem processors and programmers. Not normally used.

POTENTIAL_CALLER_STKP: This field is set to the dynamic predecessor's stack frame pointer if the dynamic predecessor has multiple stack frames. Otherwise, it is zero. Not normally used.

DYNAMIC_LINK: This field contains whatever the current procedure found in B2 when it received control (pointer to caller's stack frame).

RETURN_ADDRESS: Address to which the epilog will go to.

4.4 STACK FRAME LAYOUT

SF + 0 Will contain the linkage word.

SF + 1 Will normally be the start of the user's data in the stack frame if coding a COMPASS subroutine. Internally a CYBIL procedure starts the user's data at SF + 5.

4.5 CALLING SEQUENCES

The interfaces described in this section are available on common deck ZPXIDEF which is availabe through the CYBCCMN parameter on SES procedure GENCOMP.

4.5.1 PROCEDURE ENTRANCE (PROLOG)

```
MORE    RJ      =XCIL#SPE       increase field length
START   SX0     B2              caller's stack frame pointer to X0
        LX0     18
        BX6     X7+X0           merge into linkage word
        SB7     size of stack frame needed
        SB2     B2-B7           move stack frame pointer
        GE      B3,B2,MORE      check if room
        SA6     B2              store linkage info in stack
          .
          .
          .
```

4.5.2 PROCEDURE EXIT (EPILOG)

```
RETLAB  BSS     0
        SA1     B2              load linkage word
        SB7     X1              return address to B7
        SB2     B2+size of stack frame needed
        JP      B7
```

4.5.3 CALLING A PROCEDURE

1) Set up parameters in X1...X5 plus B5 if necessary.
2) Set up linkage word in X7.
3) Use an EQ instruction to jump to the procedure in mind. Must not use a return jump.

4.6 PARAMETER PASSAGE

4.6.1 REFERENCE PARAMETERS

In the case of reference parameters a pointer to the actual data is generated and the pointer is passed as the parameter.

4.6.2 VALUE PARAMETERS

In the case of "big" value parameters (i.e., larger than 1 word in length) the parameter list contains a pointer to the actual parameter and the callee's prolog copies the parameter to the callee's stack frame.

If the parameter length is less than or equal to a word then it is a candidate for passing via one of the 5 X registers as described above. If all 5 X registers are all ready in use, passing other value parameters, then the parameter is included in the extended parameter list entries. In either case it is a copy of the actual data.

Remember that adaptable pointers are bigger than one word in length and consequently when they are passed as a value parameter they are considered a "big" parameter.

4.7 RUN TIME LIBRARY

4.7.1 MEMORY MANAGEMENT

4.7.1.1 Memory Management Categories

Three categories of memory management occur for CYBIL programs:

1) Run Time Stack;
2) Default Heap; and
3) User Heap.

The run time stack and default heap managers use blocks of memory obtained through run time library calls to the Common Memory Manager (CMM). User heaps occupy memory designated by the CYBIL program and are managed entirely by CYBIL run time routines.

4.7.1.2 Stack Management

Most of the stack management is done in the compiler generated code. Only under exceptional conditions will run time library routines be invoked. Each procedure activation has associated with it a stackframe, which is used to keep local variables, compiler generated temporaries, and procedure linkage information. The stackframe consists of several fragments:

1) The base fragment, which is acquired during the prolog, and

2) The extension fragments, which are acquired during the execution of the procedure body through PUSH statements or through space required to copy adaptable value parameters. At procedure termination, the epilog releases the activation's stack frame, possibly to be reused on later procedure activations.

This dynamic behavior implies that the run time stack must be part of the dynamic memory area; i.e., must coexist with the memory manager.

The model used by CYBIL is a compromise between efficiency and flexibility. It uses stack segments, each of which accommodates at least one, but usually many, fragments. Within a stack segment, the acquisition of a new fragment is done by inline code, unless the current segment is exhausted where upon a stack management routine is called to obtain a new stack segment from the memory manager. Registers B2 and B3 are reserved throughout program execution to maintain the state of the stack.

The default stack segment size is 3000(8) words which according to our experience, is normally enough. In the case where additional memory is required additional stack segments are obtained with an incremental size of 2000(8) until adequate memory is obtained.

4.7.1.3 Default Heap Management

Memory Management for the default heap is done by calls to CMM from a run time routine when an allocate or free request is made. In some cases the run time interface for allocate may be able to release unused stack segments to become available for the default heap. The run time interface allows CMM to increase field length as necessary but does not allow CMM to reduce field length, in order to curb the potential for a job's field length to change up and down many times during execution. Apart from the cases mentioned here, however,

default heap management is under the control of CMM and is essentially transparent to the CYBIL program.

4.7.1.4 User Heap Management

The user heap manager manages contiguous storage areas (heaps) which are organized into memory blocks. Each block is either free or allocated. The free blocks are linked to form a free block chain, whose start is identified by a free chain pointer. Initially, each heap contains one free block.

An allocate request causes the memory manager to search the specified heap's free block chain for a block that is sufficiently big. Depending on the found block's excess size, either the whole block or a sufficiently large part of it is returned to the caller (in the latter case the remainder is removed from the block and inserted (as a new free block) into the free block chain). If it is impossible to allocate a block of the requested size a nil pointer value is returned for the request.

A free request causes a block to be inserted into the free block chain of a heap. In order to reduce memory fragementation, it is merged immediately with adjacent free blocks (if they exist).

4.7.1.5 CMM Error Processing

The CYBIL run time interface to CMM traps any fatal errors detected by CMM. If the error condition is no more memory available then a nil pointer is returned for the allocate call. For all other other error conditions the job step is aborted with the dayfile message '- FATAL CMM ERROR'. When the job is aborted register X1 contains the CMM status word. See the CMM Reference Manual (Pub. No. 60499200) section on own-code error processing for a description of the CMM status word.

4.7.2 I/O

The CYBIL I/O utilities are available as part of the run time system contained on CYBCLIB. The I/O interfaces are described in document ARH2739 and supported via common decks on CYBCCMN in the SES catalog.

4.7.3 SYSTEM DEPENDENT ACCESS

A set of CYBIL callable routines are available and described in the SES document: ERS for Miscellaneous Routines Interface SESD003.

4.8 VARIABLES

4.8.1 VARIABLES IN SECTIONS

Using the section attribute on a variable has no effect on the variable other than to assure its residence with the static variables.

4.8.2 GATED VARIABLES

The #GATE attribute is ignored on both variables and procedures.

4.8.3 VARIABLE ALLOCATION

Space for variables is allocated in the order in which they occur in the input stream. No reordering is done. If a variable is not referenced, no space is reserved.

4.8.4 VARIABLE ALIGNMENT

The <offset> mod <base> alignment feature of the language is ignored. Quoting any combination of alignments will always result in word alignment.

4.9 STATEMENTS

This section describes what may be less than obvious implementations of certain CYBIL statements.

4.9.1 CASE STATEMENTS

Alternate code is generated for case statements depending on the density of selection specs. The "span" of selection values is equal to the highest value found in a sellction spec minus the lowest value found in a selection spec, plus one. This is the number of words

that would be needed in a jump table, with one entry per word. A series of conditional jumps requires two words per selection spec (one test against each bound). The CC code generator picks the method that will result in less code: if the span of selection values is less than twice the number of selection specs then a jump table is generated, otherwise, a series of conditional jumps is generated. If a conditional jump sequence is being generated and there is 9 or more selection specs present a "midpoint label" is generated to bisect the conditional jump sequence.

5.0 CYBIL-CP TYPE AND VARIABLE MAPPING

The Pcode data formats for each of the supported CYBIL data types is described in the following sections. These data mappings are compatible with the UCSD version IV.0 format.

The Pcode interpreter supports three basic data types as follows:

- o Bits
- o Bytes (8 bits)
- o Words (16 bits)

Integers are represented in two's complement form.

Quoting any combination of the CYBIL alignment attribute will result in word alignment.

5.1 POINTERS

A pointer consists of an address field of 2 bytes and, for certain pointer types, a descriptor. The address field contains a 16-bit address of the first byte of the object (data or procedure).

The value of the nil data pointer is constructed via the LDCN pcode instruction whose normal value is:

0001 (16)

The address field for a nil procedure pointer is described in the paragraph on procedure pointers.

With the exception of pointers to string and pointers to sequences, pointers to fixed size data objects consist of the address field only.

A pointer to string consists of an even, 2-byte address field followed by a 2-byte field indicating the starting byte offset of the possible substring. A value of zero indicates the first character position of the string and the bytes are numbered consecutively.

A pointer to a sequence consists of the 2-byte address field followed by 2 2-byte fields indicating the size of the sequence in words, and the word offset to the next available position in the sequence.

5.0 CYBIL-CP TYPE AND VARIABLE MAPPING
5.1.1 ADAPTABLE POINTERS

5.1.1 ADAPTABLE POINTERS

Adaptable pointers are identical to pointers to the corresponding fixed type with the exception that the pointer consists of the address field and a descriptor containing information such as the size of the structure.

An adaptable string pointer consists of the 2-byte address field, followed by a 2-byte position indicator, followed by a 2-byte size field indicating the length of the string in bytes.

An adaptable array pointer consists of the 2-byte address field followed by 3 2-byte fields indicating the array size, the lower bound and the upper bound. The value for the array size is in words independent of packing.

An adaptable sequence pointer consists of the 2-byte address field followed by 2 2-byte fields indicating the size of the sequence in words, and the word offset to the next available position in the sequence.

An adaptable heap pointer consists of the 2-byte address field followed by a 2-byte size field containing the size of the heap in words.

An adaptable record pointer consists of the 2-byte address field followed by one of the above descriptors depending on the adaptable field of the record. Thus, if the adaptable field is a string, the adaptable record pointer consists of a 2-byte address field, followed by a 2-byte position indicator, followed by a 2-byte size field indicating the length of the string in bytes.

5.1.2 PROCEDURE POINTERS

A procedure pointer consists of a 2-byte field containing the procedure number, followed by a 2-byte pointer to E_rec field, followed by a 2-byte static link.

A level 0 procedure does not require a static link. Therefore, the nil data pointer is used.

For a nil procedure pointer, the address field contains the address of a run time library procedure which handles the call as an error, and the static link field contains a nil data pointer.

5.1.3 BOUND VARIANT RECORD POINTERS

A bound variant record pointer consists of the 2-byte address field followed by a 2-byte size field, containing the size of the record in words.

5.1.4 POINTER ALIGNMENT

All pointer types are word aligned.

5.2 INTEGERS

Integer types are allocated 16 bits.

An unpacked integer type is word aligned.

A packed integer type is word aligned.

An integer variable is mapped as an unpacked integer type.

5.3 CHARACTERS

An unpacked character type is allocated 16 bits and is right justified on a word boundary.

A packed character type is allocated 8 bits and is bit aligned.

A character variable is mapped as an unpacked character type.

5.4 ORDINALS

Ordinal types are mapped as the integer subrange 0..n-1, where n is the number of elements in the ordinal type.

5.5 SUBRANGES

5.5.1 WITHIN INTEGER DOMAIN

An unpacked integer subrange type is allocated a word (16 bits) and is word aligned.

A packed subrange type, a..b, with a negative is allocated and

5.0 CYBIL-CP TYPE AND VARIABLE MAPPING
5.5.1 WITHIN INTEGER DOMAIN

aligned as an unpacked integer subrange type. If a is non-negative then it is bit aligned and it has its allocated bit length, L, computed as follows:

L:= CEILING (LOG2(b+1))

A subrange variable is mapped as an unpacked subrange type.

5.5.2 OUTSIDE INTEGER DOMAIN

Subranges of integer type can encompass the range -32768 .. 32767. For these large subranges, the implementation for packed will be the same as that for unpacked. This requires a minimum of 3 words, the first reserved for sign, the remaining to contain four digits per word, four bits per digit.

For the subrange a .. b, let
n := number_of_digits (max (abs (a), abs (b)))
then the number of data words required, would be:

n	#words
5..8	3
9..12	4
13..16	5

The internal representation of long subranges is as binary integers.

5.6 BOOLEANS

An unpacked boolean type is allocated 16 bits right justified on a word boundary.

A packed boolean type is allocated 1 bit and is bit aligned.

A boolean variable is mapped as an unpacked boolean type.

The internal value used for FALSE is zero and for TRUE is one.

5.7 REALS

Real types are allocated 32 bits.

An unpacked real type is word aligned.

A packed real type is word aligned.

A real variable is mapped as an unpacked real type.

5.0 CYBIL-CP TYPE AND VARIABLE MAPPING
5.7 REALS

See the UCSD P-system Internal Architecture Guide, page 14, for the internal representation of real numbers.

5.8 LONGREALS

Treated the same as reals.

5.9 SETS

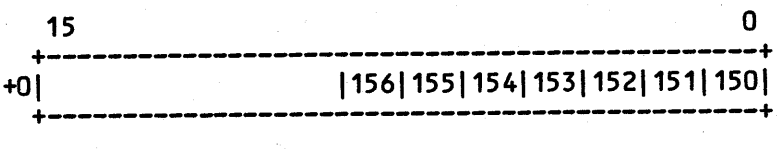
The number of contiguous bits required to represent a set is the number of elements in the base type of the associated set type. The rightmost bit represents the first element, the next bit represents the second element, etc.

An unpacked set type is allocated a field of enough words to contain the set elements. The set field is word aligned.

Example -

```
  TYPE
    S1 = SET OF 150..156;
  VAR
    A: S1;
```

Set A resides as follows:



A packed set type is mapped as an unpacked set type.

A set variable is mapped as an unpacked set type.

The maximum size allowed for a set is 4079 elements.

5.10 STRINGS

A string type is allocated the same number of bytes as there are characters in the string.

An unpacked string type is word aligned and occupies an integral

number of words. Any filler byte is zero.

A packed string type is word aligned and occupies an integral number of words.

A string variable is mapped as an unpacked string type.

The maximum length of a string is limited to 32767 characters.

In many respects a string is represented as a packed array of character. String constants reside in the constant pool with the odd character positions occupying the lower portion of each word. The even character positions occupy the upper portion of each word.

5.11 ARRAYS

An unpacked array type is a contiguous list of aligned instances of its component type. The array is aligned on a word boundary and occupies an integral number of words.

A packed array type is a contiguous list of unaligned instances of its component type with the restriction that the component type can not cross word boundaries. The array is aligned on its first element and occupies as many bits as needed.

An array variable is mapped as an unpacked array type.

In general, array sizes are limited by storage availability.

5.12 RECORDS

An unpacked record type is a contiguous list of aligned fields. It is aligned on a word boundary, and occupies an integral number of words.

A packed record type is a contiguous list of unaligned fields with the restriction that a component field can not cross word boundaries. It is aligned on its first field, and occupies as many bits as needed.

A record variable is mapped as an unpacked record type.

5.13 SEQUENCES

A sequence type consists of the data area required to contain the span(s) requested by the user. A sequence type is always word aligned, and occupies an integral number of words.

5.14 HEAPS

5.14.1 SYSTEM HEAP

The system heap is as described in the UCSD manuals.

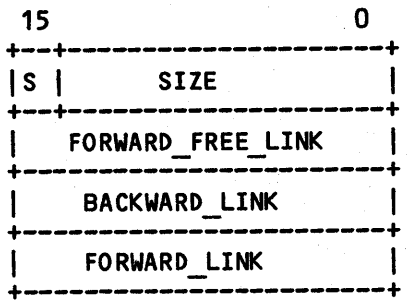
5.14.2 USER HEAPS

A user heap consists of a Free Chain Header and storage for Allocated Blocks and Free Blocks.

An Allocated Block consists of an Allocated Block Header followed by storage for user data.

A Free Block consists of a Free Block Header followed by storage which is available for use.

A common format is used for all 3 headers as follows:



The field, S, indicates the status of the block, AVAILABLE or USED.

The CYBIL description of the common header format is as follows:

5.0 CYBIL-CP TYPE AND VARIABLE MAPPING
5.14.2 USER HEAPS

```
BLOCK_HEADER = PACKED RECORD
  BLOCK_STATUS: (AVAILABLE,USED),
  SIZE: 0..7FFF(16),
  FORWARD_FREE_LINK: 0..0FFFF(16),
  BACKWARD_LINK:     0..0FFFF(16),
  FORWARD_LINK:      0..0FFFF(16),
RECEND;
```

For the Free Chain Header, the fields are as follows:

BLOCK_STATUS: Set to AVAILABLE
SIZE: Size of heap
FORWARD_FREE_LINK: Link to Free Block.
BACKWARD_LINK: 0
FORWARD_LINK: 0

For the Allocated Block Header, the fields are as follows:

BLOCK_STATUS: Set to USED.
SIZE: Size of block
FORWARD_FREE_LINK: Not used
BACKWARD_LINK: Link to preceeding block
FORWARD_LINK: Link to succeeding block

For the Free Block Header, the fields are as follows:

BLOCK_STATUS: Set to AVAILABLE
SIZE: Size of Block
FORWARD_FREE_LINK: Link to succeeding Free Block.
BACKWARD_LINK: Link to preceeding block
FORWARD_LINK: Link to succeeding block

Initially, a user heap consists of the Free Chain Header and a Free Block. Typically, an ALLOCATE request is made causing the Free Block to be divided into a Free Block and an Allocated Block.

Adjacent free blocks are always combined as part of FREE request processing.

The amount of storage allocated for a user heap is the sum of the following:

- 8 bytes for the Free Chain Header
- 8 times the repetition count for each span specified (in order to provide for block headers)
- sum of the spans specified

5.15 CELLS

A cell type is allocated 16 bits and is always word aligned.

5.16 SUMMARY FOR THE PCODE GENERATOR

	UNPACKED		PACKED	
TYPE	ALIGN	SIZE	ALIGN	SIZE
BOOLEAN	word	word	bit	bit
INTEGER	word	word	word	word
SUBRANGE	word	word	bit	bits
	word	long		
ORDINAL	word	word	bit	bits
CHARACTER	word	word	bit	byte
STRING	word	words	word	bytes
SET	word	words	word	words
ARRAY	word	words	word	words
RECORD	word	words	word	words
POINTER	word	words	word	words
CELL	word	word	word	word

6.0 CYBIL-CP RUN TIME ENVIRONMENT

The instructions generated by the CYBIL Pcode generator are per the UCSD version IV.0 P-system.

6.1 MEMORY

With regard to memory, a CYBIL program has the following parts:

- Code and Literals
- Static Storage
- Stack Heap Area

6.1.1 CODE AND LITERALS

Program counter relative addressing is used to refer to code and literals except for the following:

- Pointers to procedures
- Calls to external procedures

For the above, full 16-bit addresses are used.

6.1.2 STATIC STORAGE

The lifetime of static variables is the life of the program execution.

6.1.3 STACK HEAP AREA

The Stack Heap area is a storage area for the stack and the system heap. The stack grows from high numbered locations to low. The system heap grows from low numbered locations to high. If a collision occurs, the program aborts.

6.1.3.1 STACK FRAMES

The stack frame consists of four parts ordered from high addresses to low:

- Function return value (optional)

- Argument list (optional)
- Fixed sized part containing all automatic and implied, local variables and fixed local copies of non-scalar, value parameters (optional)
- Mark Stack Control Word (MSCW) provided and manipulated by the Pcode interpreter during call and RPU Pcode interpretations.

The first two parts are pushed onto the operand stack as the call is being formed. The next part and the MSCW is placed onto the stack by the interpreter as part of the call interpretation. The RPU (return) instruction causes the discarding of all but the optional return value.

6.1.3.1.1 FUNCTION RETURN VALUE

A scalar size operand normally. For functions that provide a pointer value requiring a descriptor (adaptable, bound variant), the Pcode calling/returning sequence may have as many as three words of returning value. For functions returning large integer subranges, the value may require four to six words.

6.1.3.2 ARGUMENT LIST

Each actual parameter is represented in the parameter list as a value or a pointer. The pointer may include descriptor information for adaptable and bound variant formal parameters.

Adaptable parameters may be declared such that not all bounds and size information is known at compile time. In this case the compiler allocates a type descriptor which contains the result of the calculation of all variable bounds, and a variable descriptor which contains information to locate the base address of the variable bound part of the automatic stack. These descriptors are in the argument list of the stack frame.

6.1.3.2.1 FIXED SIZE PART

The Fixed Size Part contains data which the procedure may access directly. The Fixed Size Part contains the following:

- Automatic Variables
- Value Parameters

- Workspace

Automatic variables and value parameters may be declared such that all bounds and size information is known at compile time. In this case, the required storage is allocated from the Fixed Size Part of the stack frame.

6.1.3.2.2 MARK STACK CONTROL WORD

Five full words providing:

- MSSTAT - pointer to the activation record of the lexical parent.
- MSDYN - pointer to the activation record of the caller.
- MSIPC - seg-relative byte pointer to point of call in the caller.
- MSENV - E_Rec pointer of the caller.
- MSPROC - procedure number of caller.

6.1.4 HEAP

Memory management for the system heap and user heaps is done via calls to standard run time routines.

6.1.4.1 System Heap

To allocate space on the system heap a procedure call of the form:

```
SYSALLOC ( pointer_to_type, number_of_words )
```

is generated. To de-allocate space on the system heap a call of the form:

```
VARDISPOSE ( pointer_to_type, number_of_words )
```

is generated. The value of NIL is assigned to the variable pointer_to_type.

6.1.4.2 User Heap

To allocate space on the user heap a call of the form:

```
CYP$ALLOCATE_IN_USER_HEAP(pointer_to_type,number_of_words,
pointer_to_user_heap)
```

is generated. The result of the call is a pointer that has the address of the first location allocated in the user heap.

To de-allocate space on a user heap a call of the form:

```
CYP$FREE_IN_USER_HEAP(pointer_to_type,pointer_to_user_heap)
```

is generated. The value of NIL is assigned to the reference parameter pointer_to_type.

To reset a user heap a call of the form:

```
CYP$RESET_USER_HEAP(pointer_to_user_heap: ^HEAP(*))
```

is generated.

6.2 PARAMETER PASSAGE

6.2.1 REFERENCE PARAMETERS

For a reference parameter, a pointer to the data is passed as the parameter.

6.2.2 VALUE PARAMETERS

There are two styles of passing value parameters. Scalar types and sets are passed by copying the value of the variable onto the stack.

Other structured types are passed by pushing the address of the structure. In the prolog of the called procedure, the structure is copied into the local data area.

In order to preserve the string pointer structure (pointer/offset), string constants, when appearing as the actual parameter will be copied into the caller's local storage as part of the call.

Adaptable value parameters are passed as if they were reference parameters. This is done because there is no mechanism to "PUSH"

stack space.

6.3 VARIABLES

6.3.1 VARIABLE ATTRIBUTES

6.3.1.1 Variables in Sections

Using the section attribute on a variable has no effect on the variable other than to assure its residence with the static variables.

6.3.1.2 Read Attribute

The READ attribute, when associated with a variable, causes compile time checking of access to the variable. No provision for execution time checking is made.

6.3.1.3 #Gate Attributes

The #GATE attribute is ignored.

6.3.2 VARIABLE ALLOCATION

Space for variables is allocated in the order in which they occur in the input stream. No reordering is done other than allocating space in the stack from high numbered locations to low.

If a variable is not referenced, no space is reserved.

6.3.3 VARIABLE ALIGNMENT

A subset of the ALIGNED feature of the language is implemented. The subset provides for guaranteeing addressability only. Any offset or base specification is ignored.

6.4 EXTERNAL REFERENCES

During the compilation process a hash is computed for each XDCL and XREF variable and procedure. The hash is based on an accumulation of data typing. In the case of procedures the parameter list is included in the process. A loader may check these hash values to

assure that the data types for all XDCL and XREF items agree.

6.5 EXTERNAL NAMES

The external/entry point names are limited by the UCSD system to be the first 8 characters.

6.6 PROCEDURE REFERENCE

6.7 FUNCTION REFERENCE

A function is a procedure that returns a value. The function value is returned via the RPU pcode instruction.

6.8 PROCEDURE CALL AND RETURN INSTRUCTION SEQUENCES

6.8.1 PROCEDURE CALL

A procedure/function call can be separated into several subsequences. If the called procedure is a function, then the initial Pcode sequence causes room for the function return value, e.g.,

```
SLDC 0
```

would be appropriate for an integer function call.

Because of the high to low allocation mechanism of UCSD stack frames, the procedure body of the called function will reference the function return value in the last allocated space of its stack frame.

Should the called procedure have parameters, then the parameter values or addresses are pushed onto the stack in the normal left to right order. If the formal parameter is of reference type, then the address of the actual parameter is pushed. Otherwise, if the parameter is of scalar type then its value is pushed, else the address is pushed and the procedure's prolog will make a local copy.

In some cases above where "the address is pushed" is used, if the formal parameter requires a descriptor (adaptables and bound variant records), then the description is pushed along with the address.

Within the called procedure, because of the high to low nature of the stack frame, the first formal parameter will be allocated the highest offset in the frame (just lower than the optional function return value). This repeats with the last parameter having the lowest

offset of all parameters.

Summarizing, a procedures stack frame is allocated beginning at word offset 1 in the following order:

- Automatic variables and local copies for value, non-scalar parameters.
- Parameter value and address/descriptors in a right to left order.
- Function return value.

The procedure call Pcode instruction is selected from a set of several depending upon the lexicgraphical distance between caller and callee. All calls contain the called procedures ordinal. This ordinal is a Pcode Generator assigned value assigned from 2 (except for PROGRAM declarations which will be given ordinal number 1) upwards (p-ord in examples below).

Examples:

CPL p-ord	Used to call local (child) procedures to the calling procedure and its body (i.e., LEX = +1).
SCPI 1 p-ord	Used to call sibling procedures of the calling procedure (LEX = 0).
SCPI 2 p-ord	Used to call parent procedures of the calling procedure (LEX = -1).
CPI n p-ord	Used to call intermediate, but non-global procedures (LEX < -1).
CPG p-ord	Used to call outer level procedures local to this module.
CXG seq p-ord	Used to call XREF procedures that are located in other compilation units.
CPF	Used to call formal procedures that have been introduced in CYBIL text as pointers to procedures.

6.9 PROLOG

All non-scalar, value parameters have an area for a local copy of the actual parameter. The prolog for a procedure will contain Pcodes to move the data into this local area.

Parameters of adaptable type are loaded by the calling mechanism in reverse order (because of the downward growing operand stack). Prolog code appears to reverse this order.

Implicit within the interpretation of the procedure call Pcodes are several functions that classically have been the explicit jobs of prolog in Pcode machines.

Since these will not be present in the PROLOG, but assumed the responsibility of the interpreter, it is worthwhile to list them:

- Stack frame creation - each procedure has a fixed stack frame size; the interpreter must "push" this area onto the dynamic stack; this size is the datasize word at the head of the procedure's code.

- Mark Stack Control Word (MSCW) located at the head of the stack frame.

6.10 EPILOG

The epilogue contains only the following:

RPU size

Size is the number of words to release from the stack. It is based on the two fixed sizes for:

- Automatic variables and local parameter storage.

- Actual parameters.

The value of size for RPU is not necessarily the same as the datasize value used by the interpreter in the prolog. It differs by and includes the additional size of the actual parameters.

6.11 RUN TIME LIBRARY

6.11.1 UNKNOWN AND/OR UNEQUAL LENGTH STRINGS

Support for unknown and/or unequal length strings is provided by calls to standard run time routines.

6.11.1.1 String Assignment

For string assignments a call of the following form is provided:

```
CYP$MOVE_STRING(pointer_to_left_string,left_string_length,
pointer_to_right_string,right_string_length).
```

6.11.1.2 String Comparison

For string comparison a function call of the following form is provided:

```
CYP$COMPARE_STRING ( operation, pointer_to_left_string,
left_string_length, pointer_to_right_string,
right_string_length ) : boolean.
```

The boolean function value indicates the result of applying one of the six relational operators on the specified strings. The relational operators are represented as: equal = 1, not equal = 2, greater than or equal = 3, less than = 4, less than or equal = 5, and greater than = 6.

7.0 EFFICIENCIES

This section lists a group of programming tips to help the user make better utilization of the CYBIL development environment. As such, it is not an exhaustive list and will be added to as additional hints become known. The CYBIL Project would appreciate any other information which may assist the usage of CYBIL.

7.1 SOURCE LEVEL EFFICIENCIES

7.1.1 GENERAL

- There is a significant amount of overhead associated with any procedure call. If a procedure is being called in a looping construct, it may pay to call the procedure once and put the loop tests inside the called procedure.
- References to variables via the static chain in nested procedures cause an overhead associated with that reference. In general, a procedure should only reference static variables, arguments and its own automatic variables.
- A copy is currently being made of all value parameters. This implementation is subject to change.
- Assignment of records is done with one large move, while record comparison is done field by field.
- Move structures rather than lots of elementary items. This may require structuring the elements together especially for this purpose.
- Reference to adaptable structures are slower than references to fixed structures because the adaptable has a descriptor field which must be accessed.
- References to fields within a record require no execution penalty.
- Repeated references to complex data structured (via pointers or indexing operations) can be made more efficient by pointing a local pointer at the structure and use it to replace the complex references.
- Inappropriate use of the null string facility can be an expensive

7.0 EFFICIENCIES
7.1.1 GENERAL

NOOP.

- Initialization of static variables incurs no run time overhead.

- If a record is being initialized with constants at run time it is often more efficient to define a statically initialized variable of the same type and do record assignment.

- A packed structure will generally require less space at the possible cost of greater overhead associated with access to its components. This is because elements of packed structures are not guaranteed to lie on addressable memory units.

- When organizing data within a packed structure it is more space efficient to group bit aligned elements together.

- The STRING data type is a more efficient declaration than a PACKED ARRAY OF CHAR.

- When considering alternative data structures for homogenous data the user should first consider ARRAYs, then SEQuences and finally HEAPs.

- When considering alternatives between the HEAP and SEQuence storage types, the following should be considered. The HEAP is the more inefficient mechanism requiring the greatest overhead in terms of space requirements and the more execution overhead. SEQuences are the more efficient in terms of both storage and execution overhead.

- The NEXT and RESET statements as used on sequences and user heaps are implemented as inline code. Whereas the implementation for ALLOCATE and FREE is a procedure call to run time library routines.

- Space in a heap is consumed only when an ALLOCATE statement is executed. In addition to the space ALLOCATEed by the CYBIL program, a header is added to maintain certain chaining information. For this reason, ALLOCATEing small types incurs a large percentage overhead.

- Code for the PUSH statement is generated inline and, as such, is considerably faster than an ALLOCATE and FREE combination.

- When a definition contains a number of 'flags' or attributes, the following should be considered when chosing between BOOLEANs or a SET type:

7.0 EFFICIENCIES
7.1.1 GENERAL

- If the record is not packed the SET will reduce the size of the definition
- Any sub-set of the attributes of a SET can be tested at once.
- If a single element test is desired an unpacked BOOLEAN is slightly more efficient than a SET.

- Usage of boolean expressions is more efficient than IF statements. For example, use:

```
equality := (a=b);
```

Do not use:

```
IF a=b THEN
   equality := TRUE;
ELSE
   equality := FALSE;
IFEND;
```

- Rather than coding long IF sequences a CASE statement should be considered when using a proper selector.

- Compound boolean expressions should be ordered such that the first condition is the one which has the highest probability of terminating the condition evaluation.

- Compile time evaluation of expressions involving constants produces better object code if all constants (at the same level) in the expression are grouped together. For example, the expression:

```
X := 5 * Y * C * 2 ;
```

will produce object code using two constants (5 and 2) and two variables (Y and C). If the expression is rewritten:

```
X := 5 * 2 * Y * C;
```

with the constants together, the compiler (at compile time) will combine the expression "5 * 2" into the constant "10" and produce object code to evaluate the expression using only one constant (the ten) and two variables (Y and C).

- Range checking code requires additional storage space and is time consuming. One can eliminate all generated range checking code by setting "CHK=0" on the call statement (or ??SET(CHKRNG:=OFF)?? n the source program). Setting CHK=0 on the call statement, while

7.0 EFFICIENCIES
7.1.1 GENERAL

debugging programs, is not recommended since legitimate program errors may not be diagnosed. A better approach is to request range checking on the call statement (or in the source program) and then minimize, using good programming practice, the amount of checking code generated. Consider the following program segment:

```
TYPE
  a = 0..10;
VAR
  index,y: a,
  x: array [a] of integer;
   .
y:=5;.
index:=y:
x[index] :=3;
```

Since variables "index" and "y" are defined to be of type "a" (the subrange 0..10) the assignment "index :=y;" will not (and need not) be checked for proper range even if range checking is requested. Similarly, the statement "x[index] :=3;" will not (and need not) contain range checking code. If variables "y" and "index" were declared to be INTEGER (or some type other than the subrange 0..10) range checking code would be required.

- Any timed executions should be run after the CYBIL code has been built with checking code turned off.

- Certain conversion functions (i.e.,ORD,CHR,etc.) require no execution time overhead.

- The code generated for STRINGREP is a call to a run time library routine.

- A file should not be opened before it is needed. As soon as a file is no longer needed, it should be closed. An overhead is involved in opening & closing files. Therefore, unnecessary opens & closes should be avoided.

7.1.2 CC EFFICIENCIES

- Pointers to strings are inefficient because the string may, in general, begin at any character boundary. These pointers may be created explicitly by assignment statements or implicitly by supplying a string as an actual parameter for a call by reference formal parameter. If possible, align strings so that they begin on a word boundary.

7.0 EFFICIENCIES
7.1.2 CC EFFICIENCIES

- Run time routines are called for the string operations of assignment & comparison when:

  1) Neither string is aligned or,
  2) Lengths are known and unequal or,
  3) Either or both lengths are unknown at compile time.

  Otherwise the faster inline code is generated.

- It is possible to modify the buffer size used by the CYBIL I/O package. For an explanation see the ERS for CYBIL I/O (ARH2739). If there are very few accesses to a file, it may be best to select a small buffer, since overall field length will be reduced, thereby increasing total system throughput by decreasing swap rates, allowing more jobs to run concurrently, etc.

7.2 COMPILATION EFFICIENCIES

If compilation time is a factor the following items could be considered as they do affect the compilation rate.

- The generation of information to interface to the symbolic debuggers slows the compilation process.

- The generation of range checking code slows the compilation process.

- The selection of listings slows the compilation process. This includes the source listing, the cross reference listing and the attribute list.

- Generating a source listing with the generated code included is slower than if just the source listing is being obtained.

- Actually, for the normal CYBIL user very little can be done to improve the compilation rate. However, rest assure that considerable effort has been expended to reduce the number of recompilations necessary to produce a debugged program.

8.0 IMPLEMENTATION LIMITATIONS

8.1 GENERAL

- Maximum number of lines in a single compilation unit is 65535.
- Maximum number of unique identifiers allowed in a single compilation unit is 16383.
- Maximum number of procedures in a single compilation unit is 999.
- Procedures can only be nested 255 levels deep.
- Maximum number of compile time variables used in conditional compilations is limited to 1023.
- Maximum number of error messages printed per module is 2000.
- Maximum number of elements defined in a single ordinal list is limited to 16384.
- Integer constants are restricted to 48 bits.

8.2 CC LIMITATIONS

- Case selector values limited to less than 2**17.
- Pointer fields within initialized packed records must be aligned for use within C170 capsules or overlay capsules.

9.0 COMPILER AND SPECIFICATION DEVIATIONS

9.0 COMPILER AND SPECIFICATION DEVIATIONS

This section is intended to provide sufficient detail to be able to understand those features where the compiler implementation lags the language specification.

**Indicates plans do not include the implementation of that feature in the R1 timeframe.

9.1 GENERAL

CYBIL Implementation - Deviations

- Double Precision Floating Point. **
- Initialization of static pointers to NIL and zeroing the adaptable descriptor fields is not done. **
- #SIZE of adaptable types. **
- Run time checking on accessing fields of variant records not supported. **
- Restricting pointers to not point to data with less scope. **
- Pre-defined identifiers are implemented as reserved words. **

9.2 CC DEVIATIONS

- Relative Pointer Types. **
- Partial condition evaluation on OR operator not supported. **
- Actual value parameters > 1 word must be addressable. **

9.3 CP DEVIATIONS

- Static initialization. **
- PUSH statement is not supported. **
- Relative Pointers. **
- General Intrinsics. **