HANDBOOK

for the

Cyber Implementation Language

(CYBIL)

Submitted: H. A. Wohlwend

Approved: ______________________

____

____

REVISION DEFINITION SHEET

REV	DATE	DESCRIPTION
A	12/15/78	Original.
B	12/19/79	Updated to reflect current product status.
C	09/17/80	Updated to reflect current product status.
D	05/08/81	Updated to reflect current product status.
E	12/11/81	Updated to reflect current product status.
F	08/05/82	Updated to reflect current product status.
G	04/22/83	Updated to reflect current product status.

1.0 COMPILING AND EXECUTING CYBIL PROGRAMS ON THE C170

Two methods to access the CYBIL compilers are described in this section. The compilers provided through the SES are the more stable and more widely used. The compilers available through the project catalog (LP3) are considerably more dynamic and are updated more frequently.

1.1 SES PROCEDURE INTERFACE

An SES procedural interface is available for access to the compiler and is described in the SES User's Handbook (ARH1833).

1.2 THE C170 CYBIL COMMAND

The CYBIL command calls the compiler, specifies the files to be used for input and output, and indicates the type of output to be produced. This call statement may be in any one of the following forms:

```
CYBIL(p1,p2,..,pn) comments

CYBIL. comments

CYBIL,p1,p2,..,pn. comments

CYBIL,p1,p2,..,pn.
```

Example:

```
CYBIL(I=COMPILE,L=LIST,B=BIN1) COMPILE TEST CASES
```

The CYBIL compilers currently reside in the tools catalog SES and in the project catalog LP3. To access the CYBIL compiler which runs on C170 and generates code for the C170 (CC):

```
ATTACH,CYBIL=CYBILC/UN=LP3.
```

To access the CYBIL-CC run time library:

```
ATTACH,CYBCLIB/UN=LP3.
```

1.3 C170 COMMAND PARAMETERS

The optional parameters p1,p2,..,pn must be separated by commas and may be in any order. If no parameters are specified, CYBIL is followed by a period or right parenthesis. If a parameter list is specified, it must conform to the syntax for job control statements as defined in the NOS REFERENCE MANUAL (Publication number: 60435400), with the added restriction that the comma, right parenthesis, and period are the only valid parameter delimiters. If comments are specified they are ignored by the compiler, but printed in the dayfile. Default values are used for omitted parameters.

In the following description of command parameters, <lfn> indicates a file name consisting of one letter followed by 0-6 letters or digits. <chars> indicates one letter followed by 0-6 letters. <digit> indicates a single digit.

PARAMETER	DESCRIPTION
EXIT OPTION	(Default: A=0)
A	System searches the control card record for an EXIT card at the end of compilation if fatal errors have been found. If such an EXIT card is not present, the job terminates.
A=0	System advances to the next control card at the end of compilation if fatal errors have been found. If the EXIT option parameter is omitted, this option is assumed.
OBJECT FILE	(Default: B=LGO)
B	Object code is written on file LGO. If this parameter is omitted, this option is assumed.
B=0	If this parameter is specified, the compiler performs a full syntactic and semantic scan of the program, but object code will not be generated, data will not be mapped, and machine dependent errors are not detected.
B=<lfn>	Object code is written on file <lfn>.
CHECKING MODE	(Default: CHK=RST)
CHK=<chars>	Selects a maximum of three of the following

checking modes. Modes unspecified are de-selected.

N Produce compiler generated code to test for de-reference of NIL pointer.

R Produce compiler generated code to test ranges. Range checking code is generated for assignment to integer subranges, ordinal subranges or character variables. All CASE statements are checked to ensure that the selector corresponds to one of the selection specs specified when no ELSE clause has been provided. All references to substrings are verified. Verify that the offset specified on a RESET..TO statement is legitimate for the specified sequence.

S Produce compiler generated code to test subscripting of arrays.

T Produce compiler generated code to verify that access to a variant record is consistent with the value of its tag field (if the tag field is present). This option is not currently supported.

CHK=0	De-selects the compiler's checking modes.
CHK	Same as CHK=NRST.

<u>DEBUGGING OPTION</u> (Default: D=OFF)

D=<chars> Selects a combination of the following options.

DS Debugging Statements. All debugging statements will be compiled. A debugging statement is a statement in the source which is ignored by the product unless this option is specified. Such statements are enclosed by the NOCOMPILE/COMPILE maintenance control pragmats.

FD Full Debug. Produce the symbolic debug information plus stylize the code generated. This option is currently supported only with the CC compiler.

SD Symbolic Debug. Produce a symbol table and

	line table for interactive debugging.
	When more than one option is desired the format is D=DSSD.
<u>SOURCE INPUT</u>	(Default: I=INPUT)
I	CYBIL source text is to be read from file COMPILE. If the SOURCE INPUT parameter is omitted, the source text is read from file INPUT. Source input ends when an end-of-record, end-of-file, or end-of-information is encountered on the source input file.
I=<lfn>	Source text is read from file <lfn>.
<u>LIST OUTPUT</u>	(Default: L=OUTPUT)
L	Compilation listing is written on file OUTPUT. When the LIST OUTPUT parameter is omitted this option is assumed.
L=0	All compile time output is suppressed. List control toggles are ignored.
L=<lfn>	Compilation listing is to be written on file <lfn>.
<u>LIST OPTIONS</u>	(Default: LO=S)
LO=<chars>	Selects a maximum of six of the following list options.

A Produce an attribute list of source input block structure and relative stack. The attribute listing is produced following the source listing on the file declared by the L option or on the file OUTPUT if L is absent.

F Produce a full listing. This option selects options A, S and R.

O Lists compiler generated object code. When O is selected, the listing includes an assembly like listing of the generated object code. This option has no meaning if the (object file) B option has been set to 0.

R Symbolic cross reference listing showing

		location of program entity definition and use within a program.
	RA	Symbolic cross reference listing of all program entities whether referenced or not.
	S	Lists the source input file.
	W	Lists fatal diagnostics. If this option is omitted, informative as well as fatal diagnostics are listed.
	X	Works in conjunction with the LISTEXT pragmat such that LISTings can be EXTernally controlled on the compiler call statement.
LO=0	No list options.	
<u>OPTIMIZATION</u>	(Default: OPT=0) (Not supported on all processors)	
OPT=<number>	0	Provides for keeping constant values in registers.
	1	Provides for keeping local variables in registers.
	2	Provides for passing parameters to local procedures in registers and for eliminating redundant memory references, common subexpressions, and jumps to jumps.
<u>PADDING</u>	(Default: PAD=0) (Not supported on all processors)	
PAD=<number>	Provides for generation of NOOP type instructions between live instructions.	

1.4 INTERACTIVE CYBIL ON C170

For the programmer using interactive job processing, the following illustrates the typical sequence of commands necessary to compile and execute a CYBIL program. An alternative to this method of operation is detailed in the section "BATCH CYBIL". The example below assumes that you know how to use a terminal and have some minimal knowledge of the NOS operating system. After you have logged in:

NOS COMMAND	DESCRIPTION
BATCH	ENTER BATCH
GET,SOURCE	GET CYBIL SOURCE PROGRAM TEXT
ATTACH,CYBILC/UN=LP3	ATTACH CYBIL COMPILER
CYBILC,I=SOURCE,L=LISTING	COMPILE CYBIL SOURCE TEXT
GET,DATA	GET DATA FILE
ATTACH,CYBCLIB/UN=LP3	GET CYBIL RUN TIME LIBRARY
LGO	EXECUTE PROGRAM. ASSUMES THAT THE CYBIL PROGRAM REFERENCES FILE NAMED "DATA". LGO WAS PRODUCED BY THE COMPILATION PROCESS (CYBILC,I=SOURCE,L=LISTING).

1.5 BATCH CYBIL ON C170

A CYBIL compilation and execution may be run as part of a NOS submitted job. In this mode, the terminal is used to start the compilation and execution. The programmer may then log off the terminal (or do other terminal work) while the job is being completed as a NOS batch job. Facilities exist to check on the progress of a submitted batch job and to examine the output as well as the dayfile from the terminal. A typical file for accomplishing either immediate batch or deferred batch job submission is shown below. The user number, password, and charge card must be changed for successful execution.

```
/JOB
XYZ,CM130000,T100.          PROGRAMMER NAME
USER,USE,PSWRD,FAMILY.      SUBSTITUTE APPROPRIATE INFORMATION
CHARGE,DEPT,PROJECT.
GET,SOURCE.                 GET CYBIL SOURCE FILE
ATTACH,CYBILC/UN=LP3.       ATTACH CYBIL-CC COMPILER
CYBILC,I=SOURCE.            COMPILE CYBIL SOURCE
GET,DATA.                   GET FILE OF DATA
ATTACH,CYBCLIB/UN=LP3.      GET CYBIL RUN TIME LIBRARY
LGO.                        EXECUTE PROGRAM
DAYFILE,TEMP.
REWIND,TEMP.
COPYSBF,TEMP,OUTPUT.
REPLACE,OUTPUT=LISTING.     SAVE LIST
SES.PRINT OUTPUT            PRINT OUTPUT
DAYFILE,LOOKSEE.
REPLACE,LOOKSEE.            SAVE DAYFILE
EXIT.                       EXIT HERE ON ERRORS
DAYFILE,LOOKSEE.
REPLACE,LOOKSEE.            ERROR DAYFILE
```

The control cards should be stored on some file (for example, CYBCRUN). To compile and execute a CYBIL program (on file SOURCE) simply use the NOS submit command: SUBMIT,CYBCRUN. NOS will respond with the time of day (e.g., 10.21.57) and a job name (e.g., ABUSF4Y). These two pieces of information should be written down for future reference. The programmer can determine the status of this submitted job with the NOS command: ENQUIRE,JN=F4Y. NOS will reply with the job status (i.e., EXECUTING, JOB IN ROLLOUT QUEUE, INPUT QUEUE, PRINT QUEUE, or JOB NOT FOUND). The PRINT QUEUE and JOB NOT FOUND message indicates that the job is completed. The file LOOKSEE contains the complete dayfile for the job. So, from a terminal the commands: GET,LOOKSEE then LIST,F=LOOKSEE lists the contents of the dayfile. This will provide an overview of the execution of the job. To obtain a detailed list of the output from the job, the commands: GET,LISTING then EDIT,LISTING are used. Text editor commands are then used to examine the desired portions of the listing.

Using this approach, the programmer has necessary information available to him at the terminal. But, the programmer need not sit at (or tie up) a terminal unnecessarily while the program is actually compiling and executing.

2.0 COMPILING AND EXECUTING CYBIL PROGRAMS ON THE C180

Two methods to access the CYBIL compilers are described in this section. The compiler provided through the standard system command call is the more stable and more widely used. The compilers available through the project catalog (LP3) are considerably more dynamic and are updated more frequently. For a detailed description of the command syntax see the NOS/VE Command Interface ERS.

2.1 THE CYBIL COMMAND

The CYBIL command calls the compiler, specifies the files to be used for input and output, and indicates the type of output to be produced. This call statement has the following positional form:

```
CYBIL [input=<file reference>]
      [list=<file reference>]
      [binary_object=<file reference>]
      [list_op=<options>]
      [debug=<options>]
      [el=<options>]
      [opt=<options>]
      [pad=<integer>]
      [runtime_checks=<options>]
      [status=<status variable>]
```

Example:

```
CYBIL I=COMPILE L=LIST B=BIN1 "COMPILE TEST CASES"
```

A more dynamic version of the CYBIL compiler resides in the project catalog LP3. To access this compiler and its runtime library (both of which run on the C180 (II)):

```
ATTF .LP3.CYBILII CYF$RUN_TIME_LIBRARY
SETCL ADD=$LOCAL.CYF$RUN_TIME_LIBRARY
```

It should be noted that this library contains a PROGRAM DESCRIPTOR (CYBIL) which uses the compiler on this library rather than the system version.

2.2 C180 COMMAND PARAMETERS

The parameter format matches the style indicated by the System Interface Standard (S2196).

PARAMETER NAMES	PARAMETER DESCRIPTION
BINARY_OBJECT B	(Default: B=$LOCAL.LGO) If this parameter is omitted object code is written to file $LOCAL.LGO. If this parameter is specified as B=$null, the compiler performs a full syntactic and semantic scan of the program, but object code will not be generated. If the parameter is specified as B=<file reference>, object code is written on file <file reference>.
DEBUG D	(Default: Generate symbol and line tables.) Selects a combination of the following debug options. If this parameter is omitted the symbol and line tables are generated by default. DS Debugging Statements. All debugging statements will be compiled. A debugging statement is a statement in the source which is ignored by the product unless this option is specified. Such statements are enclosed by the NOCOMPILE/COMPILE maintenance control pragmats. The symbol table and line table for interactive debugging will also be generated. NT No Tables. Do not generate symbol table and line table with the object code. (The default is to always generate these tables.)
ERROR_LEVEL EL	(Default EL=W) F List Fatal Diagnostics. If this option is selected only fatal diagnostics will be listed.

	W List informative as well as fatal diagnostics.
INPUT I	(Default: I=$INPUT) If the parameter is omitted, the source text is read from file $INPUT. Source input ends when an end-of-partition or end-of-information is encountered on the source input file. If the parameter is specified as I=<file reference>, the source text is read from file <file reference>.
LIST L	(Default: L=$LIST) When the LIST parameter is omitted the compilation listing is written on file $LIST. If the parameter is specified as L=$null, all compile time output is suppressed. If the parameter is specified as L=<file reference>, the compilation listing is written on file <file reference>.
LIST_OPTIONS LO	(Default: LO=S) Selects a combination of the following list options. A Produce an attribute list of source input block structure and relative stack. The attribute listing is produced following the source listing on the file declared by the L option or on the file $LIST if L is absent. F Produce a full listing. This option selects options A, S and R. O Lists compiler generated object code. When O is selected, the listing includes an assembly like listing of the generated object code. This option has no meaning if the (object file) B option has been set to $null. R Symbolic cross reference listing showing location of program entity definition and use within a program.

	RA Symbolic cross reference listing of all program entities whether referenced or not.
	S Lists the source input file.
	X Works in conjunction with the LISTEXT pragmat such that LISTings can be EXTernally controlled on the compiler call statement.
	If the parameter is specified as LO=NONE, no list options are selected.
OPTIMIZATION OPT	(Default: OPT=DEBUG)
	DEBUG Object code is stylized to facilitate debugging. Stylized code contains a separate packet of instructions for each executable source statement, carries no variable values across statement boundaries in registers, notifies debug each time a beginning of statement or procedure is reached.
	LOW Provides for keeping constant values in registers.
	HIGH Provides for keeping local variables in registers, passing parameters to local procedures in registers, eliminates redundant memory references, common subexpressions and jumps to jumps.
PAD	(Default: PAD=0)
	Provides for generation of NOOP type instructions between live instructions.
RUNTIME_CHECKS RC	(Default: RC=(R,S))
	Selects a combination of the following options.
	N Produce compiler generated code to test for de-reference of NIL pointer.
	R Produce compiler generated code to test ranges. Range checking code is generated for assignment to integer subranges, ordinal subranges or character variables. All CASE statements are checked to ensure that the

selector corresponds to one of the selection specs specified when no ELSE clause has been provided. All references to substrings are verified. Verify that the offset specified on a RESET..TO statement is legitimate for the specified sequence.

S Produce compiler generated code to test subscripting of arrays.

T Produce compiler generated code to verify that access to a variant record is consistent with the value of its tag field (if the tag field is present). This option is not currently supported.

NONE If this option is specified then no runtime checking code will be generated.

STATUS
ST

(DEFAULT: not specified)

The compiler will always return a status variable indicating whether any FATAL errors were found during the compilation just completed.

If a user status variable is specified, SCL will pass the compilation status to the user and the user can take action if fatal compilations occurred by testing this variable.

If a user status variable is not specified, then SCL will terminate the current command sequence if status returned from the the compiler is abnormal.

2.3 INTERACTIVE CYBIL ON C180

For the programmer using interactive job processing, the following illustrates the typical sequence of commands necessary to compile and execute a CYBIL program. An alternative to this method of operation is detailed in the section "BATCH CYBIL". The example below assumes that you know how to use a terminal and have some minimal knowledge of the NOS/VE operating system. After you have logged in:

NOS/VE COMMAND	DESCRIPTION
colt group_to_get	DEFINE WHAT GROUP TO GET OFF
include_group widgets	THE SCU PL (SCU_PL)
**	TERMINATE THIS FILE
scu ba=$user.scu_pl	CREATE COMPILE FILE
expd cr=group_to_get	MOVE "WIDGETS" TO COMPILE
end wl=false	TERMINATE SCU
cybil i=compile l=list	COMPILE CYBIL TEXT
attf $user.data	GET DATA FOR PROGRAM JUST COMPILED
lgo	EXECUTE PROGRAM. ASSUMES THAT THE CYBIL PROGRAM REFERENCES FILE NAMED "DATA". LGO WAS PRODUCED BY THE COMPILATION PROCESS ... cybil i=compile l=list

2.4 BATCH CYBIL ON C180

A CYBIL compilation and execution may be run as part of a NOS/VE submitted job. In this mode, the terminal is used to start the compilation and execution. The programmer may then log off the terminal (or do other terminal work) while the job is being completed as a NOS/VE batch job. Facilities exist to check on the progress of a submitted batch job and to examine the output as well as the log from the terminal. A typical file for accomplishing a batch job submission is shown below.

```
job job_name=widgets

when any_fault do             TO DO ONLY IF ERRORS OCCUR
disl all o=$user.job_failed  SAVE JOB_LOG FOR REVIEW
whenend
colt group_to_get             DEFINE WHAT GROUP TO GET OFF
include_group widgets         THE SCU PL (SCU_PL)
**                            TERMINATE THIS FILE
scu ba=$user.scu_pl           CREATE COMPILE FILE
expd cr=group_to_get          MOVE "WIDGETS" TO COMPILE
end wl=false                  TERMINATE SCU
cybil i=compile l=list        COMPILE CYBIL TEXT
attf $user.data               GET DATA FOR PROGRAM JUST COMPILED
lgo                           EXECUTE THE PROGRAM "WIDGETS"
disl all o=list.eoi           ADD THE JOB_LOG TO "LIST"
printf list                   PRINT LIST
delf $user.job_failed ..      IF JOB PASSED DELETE "JOB_FAILED"
status=ignore_status          (IN CASE FILE WAS NOT DEFINED)

jobend
```

The commands should be stored on some file (for example, widgets_job). To compile and execute the CYBIL program WIDGETS (on SCU_PL) simply use the NOS/VE INCLUDE command:

```
                              include $user.widgets_job
```

IF the job fails, then the file JOB_FAILED contains the complete log for the job.

So, from a terminal the user can do the following:

```
disjs all                     FIND OUT IF "WIDGETS" HAS FINISHED
edif $user.job_failed         DETERMINE FAILURE (IF FILE THERE)
```

Using this approach, the programmer has necessary information available to him at the terminal. But, the programmer need not sit at (or tie up) a terminal unnecessarily while the program is actually compiling and executing.

3.0 APPLICABLE DOCUMENTS

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

3.1 GENERAL

o CYBIL User's Guide (60456320-01)

o CYBIL Language Specification (ARH2298)

o This CYBIL Handbook (ARH3078)

o CYBIL Formatter ERS (ARH2619)

3.2 C170

o CYBIL I/O ERS (ARH2739)

o CYBIL Debugger ERS (ARH3142)

o SES User Handbook (ARH1833)

o SES Miscellaneous Routines Interface (SESD003)

3.3 ADVANCED SYSTEM

o CYBIL Reference Manual (60457310)

o Debugger ERS (S4024)

o System Interface Standard (S2196)

3.4 MC68000

3.5 PCODE

o UCSD P-system Internal Architecture Guide
(SofTech Microsystems, Inc.)

3.6 C200

o C200 Standards and Conventions (17329020)

4.0 COMMON CYBIL COMPILER FRONT END

This section details the characteristics of all CYBIL compilers.

4.1 INLINE PROCEDURES IMPLEMENTATION

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

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

o 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.

o 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.

o 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.

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

o 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.

4.2 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.

5.0 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.

5.1 UNPACKED BASIC TYPES

5.1.1 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.

```
59          47                                                   0
+------------+----------------------------------------------------+
|    SIGN    |                  INTEGER VALUE                     |
+------------+----------------------------------------------------+
```

5.1.2 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.

```
59                                                   11          0
+----------------------------------------------------+------------+
|/ / / / / / / / / / UNDEFINED / / / / / / / / / / /| CHARACTER |
+----------------------------------------------------+------------+
```

5.1.3 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.

```
59                                                               0
+----------------------------------------------------+------------+
|/ / / / / / / / / / UNDEFINED / / / / / / / / / / /|   VALUE    |
+----------------------------------------------------+------------+
```

5.1.4 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.

```
  58                                                             0
+-+---------------------------------------------------------------+
| |/ / / / / / / / / / / / / UNDEFINED / / / / / / / / / / / /   |
+-+---------------------------------------------------------------+
```

5.1.5 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.

5.1.6 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.

```
59          47                                                  0
+-+----------+--------------------------------------------------+
|S|EXPONENT  |                MANTISSA                          |
+-+----------+--------------------------------------------------+
```

5.1.7 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.

```
59          47                                                  0
+-+----------+--------------------------------------------------+
|S|EXPONENT1 |             UPPER MANTISSA                       |
+-+----------+--------------------------------------------------+


59          47                                                  0
+-+----------+--------------------------------------------------+
|S|EXPONENT2 |             LOWER MANTISSA                       |
+-+----------+--------------------------------------------------+
```

5.1.8 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.

```
59                                            17                 0
+---------------------------------------------+-------------------+
|/ / / / / / / / UNDEFINED / / / / / / / / / |        POINTER    |
+---------------------------------------------+-------------------+
```

5.1.9 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.

```
59                                     21    17                  0
+--------------------------------------+-----+-------------------+
|/ / / / / / / UNDEFINED / / / / / / / | POS |     POINTER       |
+--------------------------------------+-----+-------------------+
```

5.1.10 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.

```
59                                            17                  0
+---------------------------------------------+-------------------+
|/ / / / / / UNDEFINED / / / / / / /          |   POINTER         |
+-------------------------+-------------------+-------------------+
|/ / / / UNDEFINED / / / /|       LIMIT       |       AVAIL       |
+-------------------------+-------------------+-------------------+
|                         |                   |                   |
```

5.1.11 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.

```
59                        35                  17                  0
+-------------------------+-------------------+-------------------+
|/ / / / UNDEFINED / / / /|   ^ENVIRONMENT    |       ^CODE       |
+-------------------------+-------------------+-------------------+
```

5.1.12 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.

```
59                                                                0
+-+-+---+-+-+-----------------------------------------------------+
| | |...| | | / / / / / / / / / / /  UNDEFINED / / / / / / / / / / |
+-+-+---+-+-+-----------------------------------------------------+
```

5.1.13 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.

```
59          47          35          23          11           0
+------------+------------+------------+------------+------------+
|   CHAR     |   CHAR     |   CHAR     |   CHAR     |   CHAR     |
+------------+------------+------------+------------+------------+
```

5.1.14 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.

5.1.15 UNPACKED RECORD

An unpacked record is a contiguous list of aligned fields.

5.2 OTHER TYPES

5.2.1 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.

```
59                                      21    17                   0
+-------------------------------------+-----+--------------------+
|/ / / / / / / / UNDEFINED / / / / / / | POS |      POINTER       |
+-------------------------------------+-----+--------------------+
|                           DESCRIPTOR                            |
+-----------------------------------------------------------------+
```

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

5.2.1.1 Adaptable Array Pointer

The descriptor for an adaptable array is:

```
59     53                 35                 17                   0
+------+------------------+------------------+--------------------+
|/ / / |    ARRAY SIZE    |   LOWER BOUND    |    ELEMENT SIZE    |
+------+------------------+------------------+--------------------+
```

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.

5.2.1.2 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:

```
59                                                   11           0
+---------------------------------------------------+------------+
|/ / / / / / / / / / / UNDEFINED / / / / / / / / / / |   LENGTH   |
+---------------------------------------------------+------------+
```

5.2.1.3 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.

5.2.1.4 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:

```
59                                            17                  0
+---------------------------------------------+-------------------+
|/ / / / / / / / UNDEFINED / / / / / / / / / |        SIZE       |
+---------------------------------------------+-------------------+
```

5.2.1.5 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.

5.2.2 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.

```
59                                            17                  0
+---------------------------------------------+-------------------+
|/ / / / / / / / UNDEFINED / / / / / / / / / |      POINTER      |
+---------------------------------------------+-------------------+
|/ / / / / / / / UNDEFINED / / / / / / / / / |        SIZE       |
+---------------------------------------------+-------------------+
```

5.2.3 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.

5.2.3.1 Sequences

Access to a sequence is through the control information associated with the pointer to sequence. The layout of the sequence is shown below:

```
59                                                                0
+-----------------------------------------------------------------+
|                                                                 |
|                          STORAGE FOR                            |
|                                                                 |
|                           SEQUENCE                              |
|                                                                 |
|                                                                 |
+-----------------------------------------------------------------+
```

5.2.3.2 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:

```
59     54                  35                 17                  0
+------+-------------------+------------------+-------------------+
|/ / / | / / UNDEFINED / /|    AVAIL SIZE     |    ^FREE BLOCK    |
+------+-------------------+------------------+-------------------+
|                                                                 |
|                          STORAGE FOR                            |
|                          FREE BLOCKS                            |
|                           AND  USER                             |
|                        ALLOCATED  DATA                          |
|                                                                 |
+-----------------------------------------------------------------+
```

5.2.3.2.1 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:

```
59     54                  35                 17                  0
+-------+------------------+------------------+-------------------+
|/ / /  |   FORWARD LINK   |  BACKWARD LINK   |    BLOCK SIZE     |
+-------+------------------+------------------+-------------------+
|                                                                 |
|                          FREE  BLOCK                            |
|                                                                 |
+-----------------------------------------------------------------+
```

5.2.3.2.2 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:

```
59                                            17                  0
+---------------------------------------------+-------------------+
|/ / / / / / / UNDEFINED / / / / / / / / / / |   BLOCK SIZE      |
+---------------------------------------------+-------------------+
|                                                                 |
|                       ALLOCATED  SPACE                          |
|                                                                 |
+-----------------------------------------------------------------+
```

5.2.4 CELLS

A cell is allocated a word and is always aligned.

5.3 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.

5.4 SUMMARY FOR THE C170

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.

6.0 CYBIL-CC RUNTIME ENVIRONMENT

6.1 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:

- o The stack.
- o 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.

6.2 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.
```

6.3 LINKAGE WORD

```
        1       5         18         18      18
   +---------+-----+------------+-------+-------+
   |Exception|/////| Potential  |Dynamic|Return |
   | Return  |/////|Caller Stack| Link  |Address|
   |         |/////|  Pointer   |       |       |
   +---------+-----+------------+-------+-------+
```

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.

6.4 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.

6.5 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.

6.5.1 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
          .
          .
          .
```

6.5.2 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
```

6.5.3 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 <u>not</u> use a return jump.

6.6 <u>PARAMETER PASSAGE</u>

6.6.1 REFERENCE PARAMETERS

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

6.6.2 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.

6.7 <u>RUN TIME LIBRARY</u>

6.7.1 MEMORY MANAGEMENT

6.7.1.1 <u>Memory Management Categories</u>

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.

6.7.1.2 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.

6.7.1.3 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.

6.7.1.4 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).

6.7.1.5 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.

6.7.2 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.

6.7.3 SYSTEM DEPENDENT ACCESS

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

6.8 VARIABLES

6.8.1 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.

6.8.2 GATED VARIABLES

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

6.8.3 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.

6.8.4 VARIABLE ALIGNMENT

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

6.9 STATEMENTS

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

6.9.1 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.

6.9.2 INTER-OVERLAY PROCEDURE CALL

Loading of user overlays must not clobber data residing in the calling overlay. This is particularily true of data passed via parameters.

7.0 CYBIL-CI/II TYPE AND VARIABLE MAPPING

The data mappings described in this section describe the mappings as they are implemented today, with an eye toward the future and conformance to the SIS (S2196).

7.1 POINTERS

A pointer to an object of data is composed of the address of the first byte of the object plus any information required to describe the data.

The address field of a pointer is a 6 byte Process Virtual Address (PVA) which is always byte aligned and it has the following format:

```
     PROCESS_VIRTUAL_ADDRESS = PACKED RECORD
       RING_NUMBER: 0 .. 15,          {  4 bits, unsigned}
       SEGMENT_NUMBER: 0 .. 4095,     { 12 bits, unsigned}
       BYTE_NUMBER: HALF_INTEGER,     { 32 bits, signed}
     RECEND.
```

The HALF_INTEGER type is defined as the following subrange:

```
    HALF_INTEGER = -80000000(16) .. 7FFFFFFF(16).
```

The NIL pointer is the following constant:

```
     NIL: PROCESS_VIRTUAL_ADDRESS := [ 0F(16), 0FFF(16), 80000000(16).
```

Pointers to all fixed size objects contain only the PROCESS. VIRTUAL ADDRESS. Pointers to adaptable type objects contain the PROCESS VIRTUAL ADDRESS (6 bytes) and the descriptor for the adaptable type object (the descriptor follows physically the PVA).

7.1.1 ADAPTABLE POINTERS

Descriptors for adaptable types are byte aligned and they have the following formats:

a) STRING - 2 byte size field indicating the length of the string (0..65535) in bytes.

b) ARRAY - 12 byte descriptor:

```
          ARRAY_DESCRIPTOR = RECORD
            ARRAY_SIZE: HALF_INTEGER,  " in bits or bytes "
            LOWER_BOUND: HALF_INTEGER,
            ELEMENT_SIZE: HALF_INTEGER," in bits or bytes "
          RECEND.
```

ARRAY_SIZE and ELEMENT_SIZE are either both in bits, or both in bytes. The value for the sizes are in bits when the array is packed and is in bytes when the array is unpacked. Note: The ELEMENT_SIZE may be dropped in future compiler updates.

c) USER HEAP - 4 byte size field indicating the maximum length of the structure in bytes.

d) SEQUENCE - The format of a pointer to an adaptable sequence will have the same format as the pointer to a fixed size sequence as described below.

e) RECORD - Adaptable records have the descriptor of their adaptable field as described above.

7.1.2 POINTERS TO SEQUENCES

The 14-byte pointer to sequence (fixed or adaptable has the following format:

```
          SEQUENCE_POINTER = RECORD
             POINTER_SEQUENCE: PROCESS_VIRTUAL_ADDRESS,
             LIMIT: HALF_INTEGER,
             AVAIL: HALF_INTEGER,
          RECEND.
```

The LIMIT is an offset to the top of the sequence and the AVAIL is an offset to the next available location in the sequence.

7.1.3 PROCEDURE POINTERS

The 12-byte pointer to procedure has the following format:

```
      PROC_POINTER = RECORD
```

```
        POINTER_TO_PROCEDURE_DESCRIPTOR: PROCESS_VIRTUAL_ADDRESS,
        STATIC_LINK_OR_NIL: PROCESS_VIRTUAL_ADDRESS,
      RECEND.
```

The first entry of the procedure pointer is a pointer to the procedure descriptor in the Binding Section. This procedure descriptor consists of two entries: a Code Base Pointer and a Binding Section Pointer. This implies that the Code Base Pointer will have the External Procedure Flag set for all procedures (including internal procedures) which are called via a pointer to procedure. This is done to ensure that the Binding Section Pointer is always placed in register A3 during a call.

The second entry of the procedure pointer is the static link. A level 0 procedure does not require a static link and, therefore, the nil pointer is used. This is done to ensure that pointer comparison will always work.

The nil procedure pointer is the following constant:

```
       NIL_PROC_POINTER: PROC_POINTER :=
         [ POINTER_TO_NIL_PROCEDURE_DESCRIPTOR, NIL ]
```

where the nil procedure descriptor points to a run time library procedure which handles the call through a nil procedure pointer as an error.

7.1.4 BOUND VARIANT RECORD POINTERS

Pointers to bound variant records consist of a 6 byte PVA followed by a 4 byte size descriptor.

7.1.5 POINTER ALIGNMENT

Pointer types are always byte aligned.

Pointer variables which occupy 8 bytes or more are word aligned on the left; whereas, smaller pointers are right justified in a word. Pointer types are always byte aligned even in packed structures.

7.2 RELATIVE POINTERS

A relative pointer is a 4 byte field which gives the byte offset of the object field from the start of the parent:

```
                    RELATIVE_ADDRESS = 0 ..  0FFFFFFFF(16).
```

Relative pointers are always byte aligned.

7.2.1 ADAPTABLE RELATIVE POINTERS

Relative pointers referencing adaptable type objects consist of the 4 byte relative-address plus a descriptor for the adaptable object type. This descriptor physically follows the relative-address field. Descriptors for adaptable relative pointer types have the alignment and formats described above in the section titled Adaptable Pointers.

7.2.2 RELATIVE POINTERS TO SEQUENCES

The 12-byte relative pointer to sequence (fixed or adaptable) has the following format:

```
                    RELATIVE_POINTER_TO_SEQUENCE = RECORD
                      RELATIVE_POINTER: RELATIVE_ADDRESS,
                      LIMIT: HALF_INTEGER,
                      AVAILABLE: HALF_INTEGER,
                    RECEND.
```

7.2.3 RELATIVE POINTERS TO BOUND VARIANT RECORDS

Relative pointers to bound variant records consist of a 4-byte relative_address followed by a 4-byte size descriptor.

7.3 INTEGERS

Integer type variables are allocated 64 bits and are word aligned.

Unpacked and packed types are byte aligned when within a structure.

7.4 CHARACTERS

Character type variables are allocated 8 bits. Unpacked character types are byte aligned while packed character types are bit aligned.

A character variable is mapped as an unpacked character type and it is right aligned in a word.

7.5 ORDINALS

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

7.6 SUBRANGES

An unpacked subrange type variable is allocated 8 bytes if its lower bound is negative; 1 to 8 bytes otherwise (depending on value of upper bound). An unpacked subrange type is byte aligned.

A packed subrange type, a .. b, is bit aligned and it has its allocated bit length, L, computed as follows:

```
  if a >= 0, then  L =     CEILING (LOG2 (       b+1       ))
  if a <  0, then  L = 1 + CEILING (LOG2 (MAX (ABS(a), b+1)))
```

A subrange variable is mapped as an unpacked subrange type and it is right aligned in a word. A subrange with a negative lower bound occupies the entire word.

7.7 BOOLEANS

An unpacked boolean type is allocated 1 byte and it is byte aligned.

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

A boolean variable is mapped as an unpacked boolean type and it is right justified in a word.

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

7.8 <u>REALS</u>

Real type variables are allocated 64 bits and are word aligned. Unpacked and packed types are byte aligned when within a structure. The magnitude of a real value can range from 4.8*10**(-1234) to 5.2*10**1232, or it can be zero.

7.9 <u>LONGREALS</u>

Longreal type variables are allocated two consecutive 64 bit words and are word aligned. Unpacked and packed types are byte aligned when within a structure. The magnitude of a longreal value has the same range as a type real value, described above.

7.10 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 leftmost bit in the set representation corresponds to the first element of the base type, the next bit corresponds to the second element of the base type, etc.

An unpacked set type is allocated a field of enough bytes to contain the set elements and the set field is byte aligned.

A packed set type which contains more than 57 set elements is mapped as an unpacked set type. A packed set type which contains 57 or less set elements is allocated a field with the number of bits necessary to contain the set elements and the set field is bit aligned.

If the set elements occupy a set field which is larger than the number of elements in the base type of the set, then the set entries are right justified in the field and the filler bits to the left of the set elements are always zero.

A set variable is mapped as an unpacked set type. If the set field containing the set elements will fit into a word then it is right justified in the word; otherwise, the set field is word aligned on the left.

The maximum size allowed for a set is 32,768 elements.

7.11 STRINGS

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

String types are always byte aligned.

A string variable which occupies more than 8 bytes is word aligned on the left; whereas, a smaller string is right aligned in a word.

7.12 ARRAYS

An unpacked array type is a contiguous list of aligned instances of its component type.

A packed array type is a contiguous list of unaligned instances of its component type. The array is aligned on a byte boundary if its element type starts on a byte boundary, or if the array is greater than 57 bits.

If the array component type is byte aligned, then it occupies an integral number of bytes.

Array variables are word aligned on the left.

The size of an array of aligned records will be a multiple of the records alignment base.

In general, the size of arrays are limited by availability of sufficient storage.

7.13 RECORDS

An unpacked record type is a contiguous list of aligned fields. It is aligned on the boundary of the coarsest alignment of any of its fields.

A packed record type is a contiguous list of unaligned fields. It is aligned on the maximum alignment of its component fields subject to the rule that it must be at least byte aligned if the record is greater than 57 bits.

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.

In an unpacked or packed record, the following field types (they must not be the subject of a pointer or a reference parameter) are defined as expandable: character, ordinal, subrange, boolean, and set. If an expandable field is followed by a field of dead bits which extends to the next field of the record (or to the end of the record), then the expandable field is expanded to include as many bits as possible up to the next field. Character, ordinal, subrange, and boolean expansion is restricted to 32 bits. A set which contains less than 57 elements can be expanded up to 57 bits, if it can be expanded to the next field. A set which contains more than 57 elements can be expanded to the next byte boundary or to the next field, whichever comes first.

If a record is byte aligned, then it occupies an integral number of bytes.

The fields are allocated consecutively subject to their alignment restrictions.

Record variables which take more than a word are left aligned in the first word. Record variables which take less than a word are right aligned in the word.

When the ALIGNED feature is used on a field within a record, the algorithm used will attempt to satisfy the offset value first (within the word being allocated).

7.14 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 control information.

7.14.1 HEAPS

Both the Default Heap and the User Heap have the following format:

```
HEAP = PACKED RECORD
  BLOCK_STATUS:    (AVAIL, USED),
  SIZE:              0..7FFFFFFF(16),
  FORWARD_FREE_LINK: 0..0FFFFFFFF(16),
  BACKWARD_LINK:     0..0FFFFFFFF(16),
  FORWARD_LINK:      0..0FFFFFFFF(16),
  DATA_AREA: SPACE,
RECEND.
```

For the heap data type, an additional 16 byte header is added for each repetition count for each span specified.

7.14.2 SEQUENCES

Sequences have the following format:

```
SEQUENCE = RECORD
  DATA_AREA: SPACE,
RECEND.
```

As demonstrated the sequence has the space required to contain the span(s) requested by the user.

7.15 CELLS

A cell type is allocated a byte and is always byte aligned.

7.16 SUMMARY

TYPE	SIZE	ALIGNMENT	
		UNPACKED	PACKED
BOOLEAN	bit	RJ byte	bit
INTEGER	8 bytes	byte	byte
SUBRANGE	as needed	RJ byte	bit
ORDINAL	as needed	RJ byte	bit
CHARACTER	byte	byte	bit
REAL	8 bytes	byte	byte
LONGREAL	16 bytes	byte	byte
STRING	n bytes	byte	byte
SET	as needed	RJ byte	bit
ARRAY/RECORD	component dependent	byte	unaligned components
FIXED POINTER	6 bytes	byte	byte
FIXED REL PTR	4 bytes	byte	byte
CELL	BYTE	BYTE	BYTE

8.0 CYBIL-CI/II RUN TIME ENVIRONMENT

The run time environment described in this section is as implemented today, with an eye toward conformance to the Advanced System SYSTEM INTERFACE STANDARD (S2196).

8.1 REGISTER ASSIGNMENT

A0	- DYNAMIC SPACE POINTER	-	DSP
A1	- CURRENT STACK FRAME POINTER	-	CSF
A2	- PREVIOUS SAVE AREA POINTER	-	PSA
A3	- BINDING SECTION POINTER	-	BSP
A4	- ARGUMENT LIST POINTER	-	ALP
A14	- STATIC LINK	-	SL
X14	- LINE NUMBER FOR RANGE CHECKING	-	LN

The registers A0, A1 and A2 always contain the assigned values. Registers A3, A4, A14 and X14 may be assigned other values during the execution of the procedure.

Dynamic Space Pointer indicates the top of the current stack frame.

Current Stack Frame pointer indicates the start of the current stack frame.

Previous Save Area pointer indicates the location of the save area for the previous procedure. When the previous procedure issued a call for the current procedure, all relevant information for the previous procedure was stored in the save area. This save area contains the contents of all hardware registers that are required for the previous procedure to execute normally when a return is issued by the current procedure.

One of the functions of the hardware call instruction is to save a designated set of registers into a save area. The save area is built on top of stack frame of the procedure that issued the call. The stack frame of the called procedure is built above the save area of the calling procedure (Note that a CYBIL-CI/II program executes in one ring only). The save area contains the following information:

8.0 CYBIL-CI/II RUN TIME ENVIRONMENT
8.1 REGISTER ASSIGNMENT

```
                      <---------------------------------
   REGISTERS: P,A0,A1,A2      Minimum
   FRAME DESCRIPTION          Save
   USER MASK                  Area             Maximum
                      <--------------------   Save
   REGISTERS: A3 .. AF                         Area
   REGISTERS: X0 .. XF
                      <---------------------------------
```

Binding Section Pointer indicates the binding section of the currently executing procedure.

Argument List Pointer points to the parameter list passed by the calling procedure. The number of parameters passed will be contained in register X0.

Static Link Pointer indicates the stack frame of the enclosing procedure if the called procedure is an internal procedure of the calling procedure and is meaningless otherwise.

8.2 STACK FRAME DEFINITION

The stack frame consists of two distinct sections. The first section contains all data whose size is known at compile time. The other section contains all adaptable structures whose size can only be determined at execution time.

```
             |                              |
A2=PSA ->    |______________________________|
             |                              |
             | Previous Save Area           |
             |                              |
A1=CSF ->    |==============================|  <--------------------
             | Reserved Condition Handler   |
             |------------------------------|
             | Function Result Save Area    |
             |------------------------------|
             | Display                      |
             |------------------------------|                C
             |                              |                U
             | Pointers to Adaptable        |                R
             |   Value Parameters & Long    |                R
             |   Fixed Value Parameters     |                E
             | Automatic Variables          |   Fixed        N
             |                              |                T
             | Short Fixed Value Parameters |   Size
             |                              |                S
             | Descriptors and Workspace    |   Part         T
             |                              |                A
             | Parameter List Workspace     |                C
             |                              |                K
             |------------------------------|
             |                              |                F
             | Register Overflow Area       |                R
             |                              |                A
             |==============================|  <------------ M
             |                              |   Variable     E
             | Adaptable Value Parameters & |   Size
             | Long Fixed Value Parameters  |   Part
 A0=DSP ->   |==============================|   <--------------------
```

8.2.1 FIXED SIZE PART

This section contains some data, enough information to provide addressability to all other data accessible by the current procedure plus an initialized 8 byte field which has been provided for condition handling plus a word to be used as a function result save area when the function has a non-local exit.

a) The "display" consists of pointers which enable the procedure to access variables that have been declared in all inclosing procedures. The format of the "display" is as follows:

```
         |                                    |
 CSF ->  |====================================|
         | Reserved for Condition Handling    |
         |------------------------------------|
         | Function Result Save Area          |
         |------------------------------------|  <------------------
         | CSF of Current Level 0 Procedure   |
         |------------------------------------|
         | CSF of Current Level 1 Procedure   |
         |------------------------------------|
         |          :           :             |   Copied from the
         |          :           :             |    Caller's Display
         |------------------------------------|
         | CSF of Current Level (n-2) Proc.   |
         |------------------------------------|  <------------------
         | CSF of Current Level (n-1) Proc.   |  Set up
         |------------------------------------|   by the Prolog
         | Argument List Pointer              |  (only if necessary)
         |------------------------------------|  <------------------
         |                                    |
```

The prolog will save the static link (if it was passed in register A14) into the display if and only if the procedure is nested. The prolog will also save the parameter list pointer (if it was passed in register A4) into the display if and only if the procedure contains at least one locally defined procedure.

The static links, current stack frame pointers for each currently active procedure, enable the current procedure to access variables from containing procedures.

Each display entry is a six byte pointer which is right justified in its display word. The total size of the display for a particular procedure is based on that procedures nesting level.

b) Automatic variables or value parameters may be declared such that all bounds and size information is known at compile time. In this case, this fixed amount of storage required for the variable is allocated out of the fixed bound part of the automatic stack.

c) Adaptable parameters may be declared such that some bounds and size information is not known at compile time. In this case we must allocate a type descriptor for the type 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 all allocated in the fixed bound part of the automatic stacks. In addition, a workspace may be required in the fixed size part to hold temporaries for runtime descriptor calculations.

d) A fixed size area is used to hold the parameter lists for procedure calls. If the current procedure calls other procedures, then the parameter list must be allocated in its own fixed part area. Each actual parameter is represented in the parameter list as either a value or a pointer. If the parameter is passed by value and its formal parameter length is less than or equal to 8 bytes, then the parameter will be represented in the list by its value in the least number of bytes required to hold the value. All other parameters are represented by 6 byte pointers (plus descriptor if required).

e) The overflow workspace is used to hold the contents of hardware registers which are preempted during execution. The size of this can be determined at compile time.

8.2.2 VARIABLE SIZE PART

This area contains storage for all adaptable value parameters whose bounds and size information is not determinable at compile time. The descriptors for these variables are contained in the fixed size part.

8.3 PARAMETER PASSAGE

8.3.1 REFERENCE PARAMETER

In the case of reference parameters a pointer to the actual data is generated and the pointer is passed as the parameter. The parameter would be on a word boundary and be left aligned.

8.3.2 VALUE PARAMETERS

In the case of "big" value parameters (i.e., larger than 1 word in length) the parameter list contains a pointer (left aligned and on a word boundary) to the actual parameter and the callee's prolog copies the parameter to the callee's stack frame. The prolog also generates a pointer to the copied data and stores it onto the callee's stack. The generation of the pointer to the parameter is done because the caller may be executing in a different ring than the callee.

If the parameter length is less than or equal to a word then a copy of the actual parameter is made in the parameter list. The parameter would be right aligned but on a word boundary.

8.4 BINDING SECTION DESCRIPTION

Binding Section Segments are intended to faciliate software linking of both code and data segments from one procedure to another. It is created by the system linker. The Binding Section Segments are readable, but not writeable in the user ring.

The binding section for each separately compiled module must contain any addressing information required by the procedures within the module.

The following information is required in the binding section:

i) Addresses of external (XREF and EXTERNAL) data - 1 word each.
ii) Base addresses of portions of other segments to which code or data is allocated - 1 word each.
iii) Addresses of external (XREF and XDCL) procedures and their binding section addresses within the binding segment - 2 words each.
iv) Addresses of any internal procedures which are assigned to ^PROCEDURE in an assignment statement, or which appear as actual parameters - 1 word each.

Note that all constant offsets within the binding section, that are encoded within the code or initialized data blocks of a module, must be marked as such - this will enable a linker to reorder or combine binding segments.

The Binding Section starts on a word boundary and each entry occupies a full word. There are three types of Binding Section entries:

1) DATA POINTERS. Each data pointer is a PVA which occupies the rightmost 48 bits of the word entry.

2) INTERNAL PROCEDURE POINTERS. Each internal procedure pointer is a 64 bit Code Base Pointer.

3) EXTERNAL PROCEDURE POINTERS. Each external procedure pointer consists of two consecutive entries. The first entry is a 64 bit Code Base Pointer. The second entry is a PVA (occupying the rightmost 48 bits of the word entry) which is the Binding Section Pointer for the external procedure.

8.5 EXECUTION ENVIRONMENT

The following segments are required in the execution environment of a CYBIL-CI/II external procedure:

1) An extensible stack described by the hardware registers: DSP=A0, CSF=A1, PSA=A2.

2) Binding segment portion described by a base address in the binding section of the linked and loaded processes ... address passed as parameter in A3 to the procedure when invoked.

3) Zero or one code segment.

4) Zero or more data segment portions.

Notes:

a) Addressability of all static data and code is provided by addresses contained in binding section.

b) Addressability of all enclosing level automatic references is provided by addresses contained in the "display" which is located in the first few words of the automatic stack frame of the current procedure.

c) Addressability of parameters is provided by the address of the parameter list passed in A4 on any call.

8.5.1 VARIABLES

8.5.1.1 Variable Attributes

8.5.1.1.1 READ ATTRIBUTE

The READ attribute when associated with a variable, will be used to control compiler checking access by the user to the variable. As such, the space for the variable will be reserved in the static working section which has read and write attributes. To include a variable in read only memory, the section declaration facility can be used.

8.5.1.1.2 #GATE ATTRIBUTE

If you have to ask what this feature is used for you probably should not be using the facility as it is hardware and O.S.

dependent. The reader who really wants to know is referred to the NOS/VE documentation.

8.5.1.2 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.

8.5.1.3 Variable Alignment

The ALIGNED feature of the language is implemented in the language such that an attempt is made to honor the <offset> field first. If the data allocation is all ready beyond the <offset> in the word then the <base> is honored first and then the <offset>.

8.5.2 STATEMENTS

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

8.5.2.1 CASE Statement

The jump table always generated for the CASE statement actually resides as a 2 byte entry in a table which resides in the read only working storage section. The code generated does a load from this table and then does a (BRREL) branch relative instruction to the appropriate case selector.

8.5.2.2 Records

Per agreement with NOS/VE, when a record value whose size is less than or equal to 64 bits is loaded, the entire record value must be accessed with a single load instruction. In particular, a single instruction must be generated even if one of the fields of the record is a pointer value.

8.6 EXTERNAL REFERENCES

During the compilation process a hash is computed for each XDCL and XREF'ed variable and procedure. The hash is based on an accumulation of the data typing. In the case of procedures the parameter list is included in the process. The loader then checks these hash values to assure that the data types for all XDCL's and XREF's agree. If they do not agree an appropriate error message is generated by the loader.

8.7 PROCEDURE REFERENCES

Registers A1, A3, A4 and A14 are used to pass information used by a procedure to locate its data:

```
    a) External Procedure:  A1  <---> Current Stack Frame Pointer
                            A3  <---> Binding Section Pointer
                            A4  <---> Argument List Pointer

    b) Internal Procedure:  A1  <---> Current Stack Frame Pointer
                            A3  <---> Binding Section Pointer
                            A4  <---> Argument List Pointer
                            A14 <---> Static Link
```

8.8 FUNCTION REFERENCES

A function is a procedure that returns a value, as such the register conventions are identical to procedure references described above. The function value is in registers or in memory depending on the type of value being returned.

If the function value is a simple pointer, then the value is returned as a PVA in A15.

If the function value is a scalar of known length less than or equal to 64 bits in length, it is returned right aligned in X15. Fill (if any) is zero bits.

If the function value is double precision then the value is returned in registers X14 & X15. X15 holds the least significant 64 bits of the value.

If the function value is not of a type described above then the result is stored left justified as the first element of the parameter list. The second element of the parameter list, in this case, specifies the first actual parameter.

8.9 RUN TIME LIBRARY

The procedures described below are available on:

```
ATTACH,CYBILIB/UN=LP3.
```

8.9.1 HEAP MANAGEMENT

8.9.2 I/O

An elementary I/O capability is provided for execution on the Advanced Systems Simulator. This procedure will display a string expression on OUTPUT.

```
PROCEDURE [XREF] PXIO (str: string (*));
```

Note: This capability is replaced by the Simulated NOS/VE I/O Interface (DAP ARH2735).

9.0 CYBIL-CM/IM TYPE AND VARIABLE MAPPING

The MC68000 data formats for each of the supported CYBIL data types is described in the following sections.

The MC68000 supports five basic data types as follows:

- o Bits
- o BCD digits (4 bits)
- o Bytes (8 bits)
- o Words (16 bits)
- o Long Words (32 bits)

CYBIL does not utilize BCD digits.

Memory addresses are byte addresses. The byte address for a word or a long word must be an even number.

On the MC68000, integers are represented in two's complement form.

Many packed types are bit aligned and are allocated the number of bits necessary to hold the item. However, if the number of bits necessary to hold the item exceeds 32, the item is word aligned and is allocated an integral number of words.

9.1 POINTERS

A pointer consists of an address field of 4 bytes and, for certain pointer types, a descriptor. The address field contains a 24-bit address of the first byte of the object (data or procedure). The 24-bit address appears right adjusted in the 4-byte field with upper bits zero.

All pointers are word aligned.

The address field for a nil data pointer is the following constant:

```
         00000000 (16)
```

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

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

A pointer to a sequence consists of the 4-byte address field followed by 2 4-byte fields indicating the size of the sequence in bytes, and the byte offset to the next available position in the sequence.

9.1.1 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 4-byte address field followed by a 4-byte size field indicating the length of the string in bytes.

An adaptable array pointer consists of the 4-byte address field followed by 2 4-byte fields indicating the array size and the lower bound. The value for the array size is in bytes when the array is unpacked, and in bits when the array is packed.

An adaptable sequence pointer consists of the 4-byte address field followed by 2 4-byte fields indicating the size of the sequence in bytes, and the byte offset to the next available position in the sequence.

An adaptable heap pointer consists of the 4-byte address field followed by a 4-byte size field containing the size of the heap in bytes.

An adaptable record pointer consists of the 4-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 4-byte address field followed by a 4-byte size field indicating the length of the string in bytes.

9.1.2 PROCEDURE POINTERS

A procedure pointer consists of the 4-byte address field followed by a 4-byte field containing the static link.

The address field contains the address of the procedure code.

The static link contains the address of the stack frame of the enclosing procedure if the pointer is to an enclosed procedure.

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 and the static link field contains a nil data pointer. The run time library procedure handles the call as an error.

9.1.3 BOUND VARIANT RECORD POINTERS

A bound variant record pointer consists of the 4-byte address field followed by a 4-byte size field, containing the size of the record in bytes.

9.1.4 POINTER ALIGNMENT

All pointer types are word aligned.

9.2 INTEGERS

Integer types are allocated 32 bits.

An unpacked integer type is word aligned.

A packed integer type is bit aligned.

An integer variable is mapped as an unpacked integer type.

9.3 CHARACTERS

An unpacked character type is allocated a byte and is byte aligned.

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

A character variable is mapped as an unpacked character type.

9.4 ORDINALS

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

9.5 SUBRANGES

An unpacked integer subrange type is allocated a word or a long word depending on the values of the lower and upper bounds. An unpacked integer subrange type is word aligned.

A packed subrange type, a..b, is bit aligned. Its allocated bit length, L, is computed as follows:

```
if a>= 0 then L:=  CEILING (LOG2(b+1))
if a<  0 then L:= 1 + CEILING (LOG2(MAX(ABS(a),b+1)))
```

A subrange variable is mapped as an unpacked subrange type.

The maximum integer subrange is -80000000(16) .. 7fffffff(16).

9.6 BOOLEANS

An unpacked boolean is allocated a byte and is byte aligned.

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

9.7 REALS

Real types are allocated 32 bits.

An unpacked real type is word aligned.

A packed real type is bit aligned.

A real variable is mapped as an unpacked real type.

9.8 LONGREALS

Longreal types are allocated 64 bits.

An unpacked longreal type is word aligned.

A packed longreal type is byte aligned.

A longreal variable is mapped as an unpacked longreal type.

9.9 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 leftmost 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 and is word aligned. An unpacked set type is left justified in its allocated field.

A packed set is allocated as follows:

- o If the number of set elements is 32 or fewer, the set is bit aligned and is allocated a field of enough bits to contain the set elements.
- o If the number of set elements is greater than 32, the set is word aligned and is allocated a field of enough words to contain the set elements. The set is left justified in the field.

A set variable is mapped as an unpacked set.

The maximum size allowed for a set is 32768 elements.

9.10 STRINGS

An unpacked string type is word aligned and occupies an integral number of words. Any filler byte is undefined.

A packed string type is byte aligned and occupies an integral number of bytes if the string length is 4 (32 bits) or less. Otherwise, a packed string type is mapped as an unpacked string type.

A string variable is mapped as an unpacked string type.

9.11 ARRAYS

An unpacked array type is a contiguous list of unpacked 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 packed instances of its component type. The array is allocated as follows:

o If the array size is 32 bits or less, the array is bit aligned and is allocated enough bits to contain the array.

o If the array size is greater than 32 bits, the array is word aligned and is allocated enough words to contain the array.

An array variable is mapped as an unpacked array type.

In general, array sizes are limited by storage availability.

9.12 RECORDS

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

A packed record type is a contiguous list of packed fields. The record is allocated as follows:

o If the record size is 32 bits or less, the record is bit aligned and is allocated enough bits to contain the record.

- o If the record size is greater than 32 bits, the record is word aligned and is allocated enough words to contain the record.

A record variable is mapped as an unpacked record type.

9.13 <u>SEQUENCES</u>

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.

9.14 <u>HEAPS</u>

A 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:

```
 31                    0
+--+----------------------+
|S |      SIZE            |
+--+----------------------+
|   FORWARD_FREE_LINK     |
+-------------------------+
|    BACKWARD_LINK        |
+-------------------------+
|    FORWARD_LINK         |
+-------------------------+
```

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

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

```
BLOCK_HEADER = PACKED RECORD
  BLOCK_STATUS: (AVAIL,USED),
  SIZE: 0..7FFFFFFF(16),
  FORWARD_FREE_LINK: 0..0FFFFFFFF(16),
  BACKWARD_LINK:     0..0FFFFFFFF(16),
  FORWARD_LINK:      0..0FFFFFFFF(16),
RECEND;
```

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

```
  BLOCK_STATUS:  Not used
  SIZE: 0
  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 AVAIL
  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 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 heap is the sum of the following:

- o 16 bytes for the Free Chain Header
- o 16 times the repetition count for each span specified (in order to provide for block headers)
- o sum of the spans specified

9.15 CELLS

A cell type is allocated a byte and is always byte aligned.

9.16 SUMMARY FOR THE MC68000

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

10.0 CYBIL-CM/IM RUN TIME ENVIRONMENT

10.1 MEMORY

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

- o Code
- o Static Storage
- o Stack
- o Heap

10.1.1 CODE

The code section contains the instructions of the program.

10.1.2 STATIC STORAGE

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

Static storage may contain the following kinds of sections:

- o Read Only Sections
- o Read Write Sections

10.1.3 STACK

The storage area for the stack is determined at load time. The stack grows from high numbered locations to low.

10.1.3.1 Stack Frame

The stack frame consists of the following parts:

- o The Fixed Size Part contains all data whose size is known at compile time.

- The Variable Size Part contains all adaptable structures whose size can only be determined at execution time. The Variable Size Part also contains storage allocated using PUSH statements.

- The Argument List Part contains the parameters of call to the procedure.

- The P-register Part contains the return address.

10.1.3.1.1 FIXED SIZE PART

The Fixed Size Part contains some of the data which the procedure may access directly, and addressing information for other data which the procedure may access. The Fixed Size Part contains the following:

- Dynamic Link
- Display
- Automatic Variables
- Value Parameters copied by Prolog
- Pointers to Adaptable Value Parameters
- Workspace
- Register Overflow Area
- Register Save Area

The Dynamic Link is the address of the stack frame for the calling procedure.

The Display consists of Current Stack Frame (CSF) pointers for all enclosing procedures. These pointers enable the procedure to access variables that have been declared in all enclosing procedures. The format of the Display is as follows:

```
        +-----------------------------------+
low     | CSF of Current Level (n-1) Proc   |
        +-----------------------------------+
        | CSF of Current Level (n-2) Proc   |
        +-----------------------------------+
        |                                   | Copied from
        |                                   | caller's Display
        +-----------------------------------+
        | CSF of Current Level 1 Proc       |
        +-----------------------------------+
high    | CSF of Current Level 0 Proc       |
        +-----------------------------------+
```

If a procedure is nested, its prolog copies the caller's Display to its Display. If a nested procedure has enclosed procedures, the nested procedure's prolog also saves the Static Link (SL) in its Display.

A Display entry is a 24-bit address which is right adjusted in a long word with zero bits on the left.

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.

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 Fixed Size Part of the stack frame. In addition, a workspace may be required in the Fixed Size Part to hold temporaries for run time descriptor calculations.

The overflow workspace is used to hold the contents of hardware registers which are preempted during execution. The size of this is determined at compile time.

The Register Save Area is used to hold registers saved as part of procedure or function call processing. The area consists of two parts: one contains caller registers saved on entry to the procedure or function; the other contains registers saved prior to calling a procedure or function. The specific registers preserved by the caller and callee are specified elsewhere in this document.

10.1.3.1.2 VARIABLE SIZE PART

This area contains storage for all adaptable value parameters whose bounds and size information is not determinable at compile time. The Variable Size Part also contains storage allocated using PUSH statements.

10.1.3.1.3 ARGUMENT LIST PART

The argument list contains the actual parameters of call to a procedure.

The caller pushes parameters onto the stack from right to left.

10.1.3.1.4 P-REGISTER PART

The P-register Part contains the return address.

10.1.4 SYSTEM HEAP

The ALLOCATE statement has the following forms:

o ALLOCATE <allocation designator> IN <heap variable>;
o ALLOCATE <allocation designator>;

If the second form is used, allocation takes place out of the default heap. This is done by making an operating system request to obtain the memory dynamically to satisfy the ALLOCATE statement.

The FREE statement has the following forms:

o FREE <pointer variable> IN <heap variable>;
o FREE <pointer variable>;

If the second form is used, an operating system request is made to release the memory dynamically to satisfy the FREE statement.

10.1.5 REGISTERS

```
  A7 - DYNAMIC SPACE POINTER        - DSP
  A6 - CURRENT STACK FRAME POINTER  - CSF
  A4 - STATIC LINK                  - SL
```

Registers DSP and CSF always contain the assigned values. Other registers may be assigned other values during the execution of the procedure.

The Dynamic Space Pointer indicates the top of the current stack frame. Register A7 has special hardware significance as the system stack pointer.

The Current Stack Frame pointer indicates the start of the current stack frame.

The Static Link pointer indicates the stack frame of the enclosing procedure if the called procedure is an internal procedure of the calling procedure. The Static Link pointer is meaningless otherwise.

10.2 PARAMETER PASSAGE

10.2.1 REFERENCE PARAMETERS

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

10.2.2 VALUE PARAMETERS

For value parameters, the parameter list contains either a copy of the actual parameter or a pointer to the parameter depending on the parameter type. If the parameter list contains a pointer to the actual parameter, the callee's prolog copies the parameter to the callee's stack frame.

Value parameters appear in the parameter list as follows:

Type	Copy or Pointer	Parameter List Entry Size
Pointer	copy	2 words for fixed pointer (except sequence). 6 words for fixed ptr to sequence. 4 words for adaptable string pointer. 6 words for adaptable array pointer. 6 words for adaptable sequence pointer. 4 words for adaptable heap pointer. 4-6 words for adaptable record pointer.
Integer	copy	2 words
Character	copy	1 word. The character is in the lower 8 bits of the word with upper bits undefined.
Ordinal	copy	1 word
Integer Subrange	copy	1 or 2 words
Boolean	copy	1 word. The boolean value is in the lower 8

		bits of the word with upper bits undefined.
Real	copy	2 words
Longreal	copy	4 words
Set	copy	1-2048 words
String	Pointer	2 words for fixed string. 4 words for adaptable string.
Array	Pointer	2 words for fixed array. 6 words for adaptable array.
Record	Pointer	2 words for fixed record. 4-6 words for adaptable record.
Cell	copy	1 word. The cell is in the lower 8 bits of the word with the upper 8 bits of the word undefined.
Sequence	Pointer	6 words for fixed sequence. 6 words for adaptable sequence.
Heap	Pointer	4 words

10.3 VARIABLES

10.3.1 VARIABLE ATTRIBUTES

10.3.1.1 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.

10.3.1.2 #Gate Attributes

The #GATE attribute is carried forward into the object text.

10.3.2 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.

10.3.3 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.

10.4 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.

10.5 PROCEDURE REFERENCES

The following registers are used to pass information on a procedure call:

```
  a) External Procedure: DSP - Dynamic Space Pointer
                         CSF - Current Stack Frame Pointer

  b) Internal Procedure: DSP - Dynamic Space Pointer
                         CSF - Current Stack Frame Pointer
                         SL  - Static Link
```

10.6 FUNCTION REFERENCE

A function is a procedure that returns a value. As such, the register conventions are identical to those for procedure references.

The function value is returned in a register or in memory depending on the type of value being returned.

The function value is returned right aligned with sign extended or zero filler bits on the left as appropriate in D-register RV (D7) if the function value is a simple pointer or a scalar.

For a longreal, the function value is returned in registers D6 and D7.

If the function value is not of a type described above, the result is stored left justified as the first element of the parameter list. The second element of the parameter list, in this case, specifies the first actual parameter. For example, this may occur if a function returns a pointer to a non-fixed type such as an adaptable array. The pointer does not fit in a register, and therefore the parameter list is used.

10.7 <u>PROCEDURE CALL AND RETURN INSTRUCTION SEQUENCES</u>

In the code sequences, symbols will be used to designate registers as follows:

```
    A7 - DYNAMIC SPACE POINTER       - DSP
    A6 - CURRENT STACK FRAME POINTER - CSF
    A4 - STATIC LINK                 - SL
    D7 - RETURNED VALUE              - RV
```

Except for a function return value in D7 (and D6 for longreals), the condition codes and the following registers are undefined on return from a function:

```
                D0-D2/D6-D7/A0-A1/A5
```

Thus, the responsibilities for preserving registers are as follows:

```
      caller: D0-D2/D6-D7/A0-A1/A5
      callee: D3-D5/A2-A4/A6
```

10.7.1 PROCEDURE CALL

The following illustrate instruction sequences for procedure calls:

External Procedure without Parameters

```
  MOVEM.L  reglist,own_regs(CSF) Save Caller Resp Regs
  JSR      external_procedure    Call Procedure
  MOVEM.L  own_regs(CSF),reglist Restore Caller Resp Regs
```

Internal Procedure and Static Link (SL)

```
  MOVEA.L  CSF,SL                Static Link
  MOVEM.L  reglist,own_regs(CSF) Save Caller Resp Regs
  BSR      internal_proc         Call Procedure
  MOVEM.L  own_regs(CSF),reglist Restore Caller Resp Regs
```

Pointer to Procedure

```
  MOVEM.L  proc_ptr(base),A0/SL  Proc Addr & Static Link
  MOVEM.L  reglist,own_regs(CSF) Save Caller Resp Regs
  JSR      (A0)                  Call Procedure
  MOVEM.L  own_regs(CSF),reglist Restore Caller Resp Regs
```

Setup Argument List on Procedure Call

```
  internal_proc(A,B,C,D)         CYBIL statement

  LEA      -args_len(DSP),DSP    Adv Top of Stack
  MOVEA.L  DSP,A0                Argument List Base
  CLR.W    (A0)+                 1-byte parameter
  MOVE.B   A(CSF),-2(A0)
  MOVE.W   B(CSF),(A0)+          1-word parameter
  MOVE.L   C(CSF),(A0)+          2-word parameter
  MOVE.L   D(CSF),(A0)+          3-word parameter
  MOVE.W   D+4(CSF),(A0)+
  MOVEM.L  reglist,own_regs(CSF) Save Caller Resp Regs
  BSR      internal_proc         Call Procedure
  MOVEM.L  own_regs(CSF),reglist Restore Caller Resp Regs
  LEA      args_len(DSP),DSP     Pop Parameters
```

10.8 PROLOG

The basic instruction sequence for the prolog is as follows:

```
prolog:
  LINK     CSF,#-frame_size      Form Dyn Link & Update DSP
  MOVEM.L  reglist,callers(CSF)  Save Callee Resp Regs
```

If the frame_size is greater than 15 bits, the following instruction sequence is used instead of the LINK instruction:

```
  LINK     CSF,#-32766           Form Dynamic Link
  SUBA.L   #frame_size-32766,DSP Update DSP
```

If the display must be copied, the prolog is as follows:

```
prolog:
  LINK     CSF,#-frame_size      Dynamic Link & Update DSP
  MOVEM.L  reglist,callers(CSF)  Save Callee Resp Regs

  LEA      display(SL),A0        Address to copy display from
  LEA      display(CSF),A1       Address to copy display to
  MOVE.W   #lex_level-1,D0       Number of entries to copy - 1

copy_display:
  MOVE.L   -(A0),-(A1)
  DBF      D0,copy_display

  MOVE.L   SL,-(A1)              Add Static Link to display
```

If the number of display entries to be copied is small, a loop will not be used.

For a value parameter, if the parameter list contains a pointer to the actual parameter, the prolog must copy the parameter to the Fixed Size Part of the stack frame, e.g.,

```
  MOVE.L   arg_n(CSF),A0         Address to copy from
  LEA      fixed_place(CSF),A1   Address to copy to
  MOVE.W   #arg_size-1,D0

copy_arg:
  MOVE.W   (A0)+,(A1)+
  DBF      D0,copy_arg
```

10.9 EPILOG

The basic instruction sequence for the epilog is as follows:

```
epilog:
  MOVEM.L  callers(CSF),reglist  Restore Callee Resp Regs
  UNLK     CSF                   DSP := CSF
                                 DSP := DSP + 4
                                 CSF := (DSP)
  RTS                            Return
```

10.10 RUN TIME LIBRARY

The run time library consists of a set of modules containing object code which generated code may reference. With the exception of the arithmetic routines, run time library routines use normal calling conventions.

The run time library contains the following modules:

- o CYM$ALLOCATE - Contains procedure CYP$ALLOCATE for allocating a block in the system heap or in a user heap.
- o CYM$FREE - Contains procedure CYP$FREE for freeing a block in the system heap or in a user heap.
- o CYM$NILERR - Contains procedure CYP$NIL to process calls to a NIL pointer to procedure, and contains procedure CYP$ERROR to process CYBIL run time detected errors.
- o CYM$STRINGREP - Contains procedure CYP$STRINGREP for the STRINGREP built-in procedure.
- o CYM$MPY - Contains procedure CYP$MPY_4_BYTES_BY_4_BYTES for integer multiplication.
- o CYM$DIV - Contains procedure CYP$DIV_4_BYTES_BY_4_BYTES for integer division.
- o CYM$MOD - Contains procedure CYP$MOD_4_BYTES_BY_4_BYTES for integer remainder.

In addition to the above, there may be other compiler-related modules. Also, the run time library may contain other miscellaneous utility modules, which are not compiler-related.

10.11 HEAP MANAGEMENT

The system heap is managed by making calls to the operating system to dynamically allocate and free memory.

User heaps are managed using run time routines. These run time routines provide for allocating and freeing blocks of storage within a storage area, along with combining adjacent free blocks.

11.0 CYBIL-CP/IP 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.

11.1 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.

11.1.1 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.

11.1.2 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.

11.1.3 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.

11.1.4 POINTER ALIGNMENT

All pointer types are word aligned.

11.2 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.

11.3 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.

11.4 ORDINALS

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

11.5 SUBRANGES

11.5.1 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 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.

11.5.2 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.

11.6 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.

11.7 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.

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

11.8 <u>LONGREALS</u>

Treated the same as reals.

11.9 <u>SETS</u>

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:

```
       15                                             0
      +-----------------------------------------------+
   n+0|                   |156|155|154|153|152|151|150|
      +-----------------------------------------------+
```

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.

11.10 <u>STRINGS</u>

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.

11.11 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.

11.12 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.

11.13 <u>SEQUENCES</u>

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.

11.14 <u>HEAPS</u>

11.14.1 SYSTEM HEAP

The system heap is as described in the UCSD manuals.

11.14.2 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:

```
 15                     0
+--+---------------------+
|S |      SIZE           |
+--+---------------------+
|   FORWARD_FREE_LINK    |
+------------------------+
|    BACKWARD_LINK       |
+------------------------+
|    FORWARD_LINK        |
+------------------------+
```

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

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

```
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

11.15 CELLS

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

11.16 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

12.0 CYBIL-CP/IP RUN TIME ENVIRONMENT

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

12.1 MEMORY

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

- o Code and Literals
- o Static Storage
- o Stack Heap Area

12.1.1 CODE AND LITERALS

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

- o Pointers to procedures
- o Calls to external procedures

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

12.1.2 STATIC STORAGE

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

12.1.3 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.

12.1.3.1 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.

12.1.3.1.1 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.

12.1.3.2 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.

12.1.3.2.1 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.

12.1.3.2.2 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.

12.1.4 HEAP

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

12.1.4.1 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.

12.1.4.2 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.

12.2 PARAMETER PASSAGE

12.2.1 REFERENCE PARAMETERS

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

12.2.2 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.

12.3 <u>VARIABLES</u>

12.3.1 VARIABLE ATTRIBUTES

12.3.1.1 <u>Variables in Sections</u>

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

12.3.1.2 <u>Read Attribute</u>

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.

12.3.1.3 <u>#Gate Attributes</u>

The #GATE attribute is ignored.

12.3.2 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.

12.3.3 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.

12.4 <u>EXTERNAL REFERENCES</u>

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.

12.5 EXTERNAL NAMES

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

12.6 PROCEDURE REFERENCE

12.7 FUNCTION REFERENCE

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

12.8 PROCEDURE CALL AND RETURN INSTRUCTION SEQUENCES

12.8.1 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 <u>right to left</u> 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.

12.9 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.

12.10 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.

12.11 RUN TIME LIBRARY

12.11.1 UNKNOWN AND/OR UNEQUAL LENGTH STRINGS

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

12.11.1.1 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).
```

12.11.1.2 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.

13.0 CYBIL-CS/SS TYPE AND VARIABLE MAPPING

13.1 POINTERS

For this document the term address means a bit address.

A pointer to an object of data is composed of the address of the first byte of the object plus any information required to describe the data.

The NIL pointer is the following constant:

```
NIL: ADDRESS := 000000000000(16).
```

Pointers to all fixed size objects contain only the ADDRESS. Pointers to adaptable type objects contain the ADDRESS (6 bytes) and the descriptor for the adaptable type object (the descriptor physically follows the Address).

13.1.1 ADAPTABLE POINTERS

Descriptors for adaptable types are word aligned and they have the following formats:

a) STRING - 2 byte size field indicating the length of the string (0..65535) in bytes.

b) ARRAY descriptor:

```
ARRAY_DESCRIPTOR = RECORD
  ARRAY_SIZE: INTEGER,  " in bits or bytes "
  LOWER_BOUND: INTEGER,
  UPPER_BOUND: INTEGER,
RECEND.
```

The value for the ARRAY_SIZE field is in bits when the array is packed and is in bytes when the array is unpacked.

c) USER HEAP - 6 byte size field indicating the maximum length of the structure in bytes.

d) SEQUENCE - The format of a pointer to an adaptable sequence will have the same format as the pointer to a fixed size sequence as described below.

e) RECORD - Adaptable records have the descriptor of their adaptable field as described above.

13.1.2 POINTERS TO SEQUENCES

The 3 word pointer to sequence (fixed or adaptable) has the following format:

```
          SEQUENCE_POINTER = RECORD
             POINTER_SEQUENCE: ADDRESS,
             LIMIT: INTEGER,
             AVAIL: INTEGER,
          RECEND.
```

The LIMIT is an offset to the top of the sequence and the AVAIL is an offset to the next available location in the sequence.

13.1.3 PROCEDURE POINTERS

The 2 word pointer to procedure has the following format:

```
      PROC_POINTER = RECORD
        ADDRESS_OF_THE_ENTRY_POINT: ADDRESS,
        ADDRESS_OF_MODULE_DATA_BASE: ADDRESS,
      RECEND.
```

The second entry of the procedure pointer is the address of the data base for the module which contains the entry point.

The nil procedure pointer is the following constant:

```
       NIL_PROC_POINTER: PROC_POINTER :=
         [ NIL, undefined ].
```

13.1.4 BOUND VARIANT RECORD POINTERS

Pointers to bound variant records consist of a 6 byte Address right justified in the first word followed by a 6 byte size descriptor right justified in the second word.

13.1.5 POINTER ALIGNMENT

Pointer variables occupy a word and are right justified in a word. Pointers with descriptors have each field of the descriptor word aligned and right justified. Pointer types have this same mapping, even in packed structures.

13.2 RELATIVE POINTERS

A relative pointer is a 4 byte field which gives the byte offset of the object field from the start of the parent:

```
                    RELATIVE_ADDRESS = 0 ..  0FFFFFFFF(16).
```

Relative pointers are always byte aligned. The relative pointer is constrained to never cross a word boundary.

13.2.1 ADAPTABLE RELATIVE POINTERS

Relative pointers referencing adaptable type objects consist of the 4 byte relative-address plus a descriptor for the adaptable object type. This descriptor physically follows the relative-address field. Descriptors for adaptable relative pointer types have the alignment and formats described above in the section titled Adaptable Pointers.

13.2.2 RELATIVE POINTERS TO SEQUENCES

The 3 word relative pointer to sequence (fixed or adaptable) has the following format:

```
              RELATIVE_POINTER_TO_SEQUENCE = RECORD
                RELATIVE_POINTER: RELATIVE_ADDRESS,
                LIMIT: INTEGER,
                AVAILABLE: INTEGER,
              RECEND.
```

13.2.3 RELATIVE POINTERS TO BOUND VARIANT RECORDS

Relative pointers to bound variant records consist of a 4-byte relative_address right justified in the first word followed by a 6-byte size descriptor right justified in the second word.

13.3 INTEGERS

Integer type variables are allocated 64 bits and are word aligned. The integer value is limited to the rightmost 48 bits of the word.

Unpacked and packed types are also word aligned even when within a structure and never cross a word boundary.

An integer value is represented by a two's complement binary representation in the range of +(2**47-1) to -(2**47).

13.4 CHARACTERS

Character types are allocated 8 bits. Unpacked character types are right justified in a word. Packed character types are byte aligned.

A character variable is mapped as an unpacked character type and it is right aligned in a word.

13.5 ORDINALS

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

13.6 SUBRANGES

An unpacked subrange type is allocated 8 bytes if its lower bound is negative; 1 to 8 bytes otherwise (depending on value of upper bound). An unpacked subrange type is byte aligned. The subrange is constrained to never cross a word boundary.

A packed subrange type, a .. b, is bit aligned and it has its allocated bit length, L, computed as follows:

```
if a >= 0, then  L =     CEILING (LOG2 (       b+1       ))
if a <  0, then  L = 1 + CEILING (LOG2 (MAX (ABS(a), b+1)))
```

A subrange variable is mapped as an unpacked subrange type and it is right aligned in a word. A subrange with a negative lower bound occupies the entire word.

13.7 BOOLEANS

An unpacked boolean type is allocated 1 word and it is word aligned.

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

A boolean variable is mapped as an unpacked boolean type and it is right justified in a word.

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

13.8 REALS

Real type variables are allocated 64 bits and are word aligned.

Unpacked and packed types are also word aligned when within a structure and never cross a word boundary.

The magnitude of a real value can range from (2**(-28672+47)) to (2**(28671+48)).

13.9 LONGREALS

Longreal type variables are handled identical to real type variables.

13.10 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 leftmost bit in the set representation corresponds to the first element of the base type, the next bit corresponds to the second element of the base type, etc.

An unpacked set type is allocated a field of enough bytes to contain the set elements and the set field is byte aligned.

A packed set type is allocated a field with the number of bits necessary to contain the set elements and the set field is bit aligned.

Packed and unpacked set types are left justified in their allocated field.

A set variable is mapped as an unpacked set type.

The maximum size allowed for a set is 32,768 elements.

13.11 STRINGS

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

String types are always byte aligned.

A string variable is word aligned and left justified.

13.12 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. The array is aligned on a byte boundary if its element type starts on a byte boundary.

If the array component type is byte aligned, then it occupies an integral number of bytes.

Array variables are word aligned on the left.

The size of an array of aligned records will be a multiple of the records alignment base.

In general, the size of arrays are limited by availability of sufficient storage.

13.13 <u>RECORDS</u>

An unpacked record type is a contiguous list of aligned fields. It is aligned on the boundary of the coarsest alignment of any of its fields.

A packed record type is a contiguous list of unaligned fields. It is aligned on the maximum alignment of its component fields.

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.

In an unpacked or packed record, the following field types are defined as <u>expandable:</u> character, ordinal, subrange, boolean, and set. If an expandable field is followed by a field of dead bits which extends to the next field of the record (or to the end of the record), then the expandable field is expanded to include as many <u>bits</u> as possible up to the next field.

If a record is byte aligned, then it occupies an integral number of bytes.

The fields are allocated consecutively subject to their alignment restrictions.

Record variables are left aligned in the first word.

When the ALIGNED feature is used on a field within a record, the algorithm used will attempt to satisfy the offset value first (within the word being allocated).

13.14 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 control information.

13.14.1 HEAPS

Data in both the default Heap and the User Heap have the following format:

```
       ADDRESS = -1 .. 7FFFFFFFFFFF(16)

       BLOCK_HEADER = PACKED RECORD
         BLOCK_STATUS: (FILLER, AVAIL, USED, INTERNAL),
         FILLER: 0 .. 7FFF(16),
         SIZE:              0..7FFFFFFFFFFF(16),
         FORWARD_FREE_LINK: ALIGNED [2 MOD 8] ADDRESS,
         BACKWARD_LINK:     ALIGNED [2 MOD 8] ADDRESS,
         FORWARD_LINK:      ALIGNED [2 MOD 8] ADDRESS,
         DATA_AREA: SPACE,
       RECEND.
```

For the heap data type, an additional 24 byte header is added for each repetition count for each span specified.

13.14.2 SEQUENCES

Sequences have the following format:

```
     SEQUENCE = RECORD
       DATA_AREA: SPACE,
     RECEND.
```

As demonstrated the sequence has the space required to contain the span(s) requested by the user.

13.15 CELLS

A cell type is allocated a byte and is always byte aligned.

13.16 SUMMARY FOR THE CYBER 200

		ALIGNMENT		
TYPE	SIZE	UNPACKED	PACKED	VARIABLE
BOOLEAN	bit	RJ word	bit	RJ word
INTEGER	word	RJ word	RJ word	RJ word
SUBRANGE	as needed	RJ word	bit	RJ word
ORDINAL	as needed	RJ word	bit	RJ word
CHARACTER	byte	RJ word	byte	RJ word
REAL	word	word	word	word
LONGREAL	word	word	word	word
STRING	n bytes	LJ word	byte	LJ word
SET	as needed	LJ word	bit	LJ word
ARRAY/RECORD	component dependent	field alignment	unaligned components	LJ word
FIXED POINTER	6 bytes	RJ word	RJ word	RJ word
FIXED REL PTR	4 bytes	RJ word	byte	RJ word
CELL	byte	LJ word	byte	LJ word

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

14.0 CYBIL-CS/SS RUN TIME ENVIRONMENT

14.1 RUN TIME LIBRARY

The following interfaces are implicitly callable during the execution of any C200 CYBIL program.

To allocate space in the heap:

```
  CYP$ALLOCATE
      (VAR alloc_ptr: ^block_header;
      length: half_word {in bytes};
      heap_ptr: ^ARRAY[index_range] OF cell;
      base: 0 ..  0ffffffff(16));

      heap_ptr = NIL => pointer to system heap
```

To free an allocated block in a specified heap:

```
  CYP$FREE (VAR user_space_to_be_freed: ^CELL;
      heap_ptr: ^ARRAY [index_range] OF cell);

      heap_ptr = NIL => pointer to system heap
```

To reset a user heap:

```
  CYP$RESET
      (heap_ptr: ^array [index_range] OF CELL;
      heap_size: 0 ..  max_heap_size);
```

To determine the string representation of a given type:

```
  CYP$STRINGREP (VAR dest_size: INTEGER;
      VAR dest: STRING(*);
      elem_list: ARRARY [*] OF put_elem_description_type);
```

To process CYBIL runtime detected errors:

```
  CYP$ERROR (error_number: INTEGER;
      line_number: INTEGER;
      module_name_ptr: ^mod_name);
```

To process calls to a NIL pointer to procedure:

```
  CYP$NIL;
```

14.0 CYBIL-CS/SS RUN TIME ENVIRONMENT
14.1 RUN TIME LIBRARY

To terminate execution gracefully call:

```
CYP$TERMINATE;
```

14.1.1 RUNTIME ERROR MESSAGES

```
0...unequal_string_length   1...adaptable_length_error
2...subscript_error         3...range_error
4...undefined_case          5...reset_to_error
6...stack_size_error        7...tag_fixer_error
8...span_fixer_error        9...length_fixer_error
10..subrange_fixer_error    11..division_by_zero
12..mantissa_error          13..exponent_error
14..substring_start_error   15..substring_length_error
16..translate_length_error  17..translate_table_overflow
18..negative_allocation     19..wrong_size_expr_for_REP
20..nil_pointer             21..unselected_CASE
22..free_of_unalloc._block  23..lower_merge_error
24..upper_merge_error       25..err_no outside msg array
```

14.1.2 CYBIL ERROR HANDLER INTERFACE TO VSOS

Any runtime detected error needs to be communicated to the user and then the task terminated. the following VSOS SIL interfaces will be used to do this:

```
Q5SNDMJC (ptr_to_len_msg, ptr_to_len_of_msg{in bytes},
          ptr_to_msg_msg, ptr_to_msg_to_be_sent,
          ptr_to_status_msg, ptr_to_status,
          ptr_to_errmsg_msg, ptr_to_errmsg);
```

After the runtime error message is sent, the task is terminated:

```
Q5TERM (termination_state, system_return_code);

termination_state => ptr to 'ABORT' message

system_return_code => ptr to 'FATAL' message
```

The following comment has been added so that the problem it addresses will not be overlooked but the problem itself does not affect the CYBIL implementation:

If the job monitor is to be rewritten in CYBIL, a means must be found to allow that task to send a runtime message. Under the current mechanism this is not possible.

14.1.3 HEAP MANAGEMENT

The basic approach is that within the heap sufficient information will be maintained that a chain of free and used space is available. On an ALLOCATE, a scan is done from the start of the heap to find space sufficient for what is requested; upon finding such a spot it is marked as used. Once a space is "allocated", it stays in the same place for the life of the data.

On a FREE request, the space is marked as available and combined (if possible) with other adjacent free areas to reduce memory fragmentation and, hence, becomes <u>reusable</u> memory. There is <u>no</u> attempt made at garbage collection.

If the programmer has not specified a user heap on the ALLOCATE and FREE statements, the compiler assumes the system heap is intended to be used.

Alignment specified on the first field of a record to be allocated will be honored by the allocation processor.

14.1.4 ADDITIONAL DESIGN CONSIDERATIONS

The following crieria were used in designing the HEAP MANAGEMENT MODEL:

(a) Space that has been FREE'D must be potentially reuseable.

(b) Space for the SYSTEM HEAP must be obtained thru standard VSOS linkages.

(c) Linkage must be provided so that it is possible to get to STATIC space allocated for the SYSTEM HEAP that is not necessarily contiguous.

14.1.5 ALLOCATE

If the user specifies a non-zero allignment base, ALLOCATE performs alignment processing as described in the following section. Normal allocation starts on a two word boundary and proceeds as follows.

14.1.5.1 The Unaligned Allocate

(1) Starting with the forward_free_link in the first heap word, search for a block whose size is greater than or equal to length requested +24 bytes.

(2) If the block's size is 40 or more bytes larger than the amount needed, split it into two blocks. Allocate the lower block to the user, and return the second to the free chain. (A lagging pointer points to previous free block.) Return to caller.

Otherwise: Remove the block from the free chain and allocate entire block to the user. Return to caller.

IFEND.

(3) If the search failed, set alloc_ptr to NIL and return.

14.1.5.2 The Aligned Allocate

(4) If alignment_base < 8, go to (1).

(5) Starting with the forward_free_link in the first heap word, search for a block whose size is greater than or equal to space requested +24 bytes.

```
Compute L = (block offset+24) MOD alignment_base.
If L=0,
  If block size > (length requested +40),
    Put upper part of block on free chain
    Allocate length requested to user
    Return to caller
  Otherwise:
    Allocate entire block to user
    Return to caller
  ifend
Otherwise:
  Compute loc_difference = alignment_base-L.
  If length_requested+40 <= block size - loc_difference,
    Put upper part of block on free chain
    Allocate length requested to user
  Otherwise:
    Allocate entire block to user
  ifend
```

```
        if loc_difference>= 24
          Put block of loc_difference bytes on free chain
        Otherwise:
          Put loc_difference bytes into previous block
        ifend
      ifend
      Return to caller.

  (6) If search failed, set alloc_ptr to NIL and return to caller.
```

14.1.6 FREE

FREE processing inserts the specified block of memory back into the free chain. In addition, if the previous or next block, or both, are free, they are combined with the current block, as described below. FREE processing proceeds as follows.

```
  (1) If the current block's block_status is not USED, issue an error
        message and abort.
      Otherwise:
        Set combined to false.
      ifend

  (2) If the block below current bock is AVAIL,
        combine current block with previous  block  by  revising  its
        size  and forward_link.  Revise the next blocks backward_link
        and make the lower previous block the current block.   Revise
        the  forward_free_link  in previous free block.  Set combined
        to true.
      ifend

  (3) If the block above current block is AVAIL,
        combine current block with the next  block  by  revising  the
        current  block's  size  and  forward_link.   Set  the current
        block's block_status to AVAIL.  Return to caller.
      Otherwise:
        If combined is true, return to caller.
        Otherwise:
          Put the current block at the head of the free chain in  the
          first  word  of the heap and set its block_status to AVAIL.
          Return to caller
        ifend
      ifend
```

14.1.7 RESET A USER HEAP

RESET initializes the free-chain header by storing a descriptor in the first word of the heap. The word indicates the size of the heap and the first usable byte in the heap. The second word is initialized by the ALLOCATE procedure. The code necessary to accomplish this task is done by a call to a run time routine.

14.1.8 ESTABLISHING THE SYSTEM HEAP

The system heap is initialized at run time for the first ALLOCATE by calling the VSOS system interface (to obtain memory space):

```
     Q5MEMORY(ptr_to_space_req_msg,ptr_to_space_req{in words},
              ptr_to_space_acq_msg,ptr_to_space_acq{bit addr})
```

This space is then initialized by setting up 3 block_headers:

```
     (1) block_header (STATUS = USED, SIZE = 0)
     (2) block_header (STATUS = AVAIL, SIZE = space-3*24)
     (3) block_header (STATUS = USED, SIZE = 0)
```

If more space is required than is available in the current memory space, Q5MEMORY is called again to get more space and then block_header (3) of the old space and block_header (1) of the new space are linked together so there is always a path from one static space to the next.

The default size requested of Q5MEMORY will be 16384 (32*512) words. In the case where the user requests space greater than the default the actaul size requested will be passed along to Q5MEMORY.

14.1.9 POSSIBLE (HEAP) BLOCK_HEADER SETTINGS

BLOCK_HEADER (CYBIL description in STORAGE TYPES sec.):

```
        | 01 |                                48 |
        +----+-----------------------------------+
        | S  |                SIZE               |
        +----------------------------------------+
        |                   FORWARD_FREE_LINK    |
        +----------------------------------------+
        |                   BACKWARD_LINK        |
        +----------------------------------------+
        |                   FORWARD_LINK         |
        +----------------------------------------+
        |                                        |
        |      FREE SPACE (OR DATA AREA)         |  SIZE-24 BYTES
        |                                        |
        +----------------------------------------+
```

Free Block Format:

S=BLOCK_STATUS: Designates whether block is available or used. avail in this case.
SIZE: Size of block in bytes, limited to 2**47-1.
FORWARD_FREE_LINK: Offset in bytes to next free block.
BACKWARD_LINK: Offset to prev. block (alloc. or free).
FORWARD_LINK: Offset to next block (alloc. or free).

Allocated Block Format:

S=BLOCK_STATUS: Set to used in this case.
Remaining fields are as described above.

Free Chain Header Format:

S=BLOCK_STATUS: Set to avail.
SIZE: Size of heap initially
FORWARD_FREE_LINK: Set to 24

14.1.10 RESTRICTIONS

(1) Allocation occurs on a word boundary. If other than a word

boundary is desired, set alignment_base to the desired alignment.

(2) If a Free is done referencing a heap after a RESET of that heap (and before the appropriate ALLOCATE), the results are undefined.

(3) Specification of a very large alignment_base value may result in no block being allocated even in an empty heap and the value NIL returned.

(4) If any block header information is altered by the user, further results are undefined when allocating or freeing in that heap.

15.0 PROCEDURE INTERFACE CONVENTIONS

15.1 INTRODUCTION

The purpose of this section is to describe the conventions that should generally be used by designers of procedural interfaces.

15.2 PURPOSE

The purpose of the following conventions is to achieve a software system which exhibits the beneficial characteristics of being understandable, reliable, efficient, maintainable, etc.

15.3 GENERAL PHILOSPHY

o Select simple straightforward interfaces. Complex interfaces, those whose description contain 'and', 'or', and conditional clauses, impair understanding of the function. If there is not an evident choice between a single complex interface and multiple simple interfaces, choose the simple interfaces.

  - A single interface encompassing multiple intrinsic functions, which cannot be performed in conjunction with one another, unduly increases validation overhead. A simple interface for each intrinsic function is preferred.

  - If the intrinsic functions encompassed by a single interface require different degrees of user privilege, each intrinsic function should be a single simple interface.

  - The combination of multiple intrinsic functions into a single interface is practical when the functions can logically be performed in conjuction with one another.

o Input parameters should be validated early in the processing when the correlation between the potential error and the actual parameter is readily identifiable. This aids in ensuring that diagnostics accurately reflect the cause of the error.

o Wherever feasible, delegate the error prognosis to the requestor (i.e., return control to the requestor with accurate information when an error is detected).

o Refrain from exposing internal structures or concepts via externalized interfaces. Before externalizing internal structures or concepts rate the probability of change and the user consequences (re-code, re-compilation, etc.) if in fact the externalization changes.

15.3.1 INPUT PARAMETER CONVENTIONS

Input parameters in the following conventions are formal parameters in the Xref procedure declaration.

o Declare all input parameters to be <value params>.

- If for any reason input parameters are declared as <reference params>, the actual parameters must be moved to local automatic variables prior to validity check and subsequent usage. Further, all input parameters declared as <reference params> must be moved before any validation or usage occurs.

o All input parameters must be checked for validity with explicit language statements prior to use. In fact all input parameters should be validated before any parameter is used.

o Input parameters which specify subfunction or function option should be discrete parameters (i.e., should not be a field of a record).

15.3.2 PARAMETER TYPING - CYBIL USAGE

Parameter types are declared in terms of the CYBIL pre-defined types or type identifiers which resolve to the pre-defined types.

o The first inclination should be to declare parameter types as type identifiers, declaring their ultimate types with type declarations.

- The language and general ease of use dictates that ordinal, array, and record parameter types be declared as type identifiers.

- For parameter types other than pointer and cell, before selecting the pre-defined types consider the following: 1) if the concept of the parameter is used by more than one external interface, use a type identifier; 2) if the parameter type has any significant probability of change, use a type identifier; and 3) if the parameter identifier cannot accurately convey the purpose and intent, use a supportive type identifier.

o Ordinal or boolean parameter types are preferred over integer or integer subrange when declaring subfunction or option parameters. If the scope of a boolean parameter type has any significant probability of exceeding binary, that parameter type should be declared as an ordinal type.

o Take advantage of the self documenting aspect of ordinals by using descriptive ordinal type identifiers and ordinal constant identifiers on parameters.

o An ordinal type should be consistent within itself, that is, there should be an evident relationship among the ordinal type identifier and the ordinal constant identifiers.

o An ordinal type should support only one concept.

o Before utilizing an ordinal subrange in an interface, consider defining a new ordinal type. If an ordinal subrange is the appropriate choice, declare that subrange as a type identifier.

o Integer subrange is preferred over integer when declaring numeric parameters. Further, the integer subrange should be declared as a type identifier and the bounds of the subrange should be specified with descriptive constant (CONST) declarations. The low bounds, if zero or one, need not be specified with constant declarations.

o Use a constant (CONST) declaration to specify length of string type parameters.

o Set type provides a mechanism by which multiple subfunctions or options may be discretely specified with a single parameter. This use of set type is preferred over the use of codes each of which specifies a combination of subfunctions or options.

o Array type parameters will provide a convenient, useful, efficient interface in the bounds of the convention objectives if the following criteria is achieved

  - The function can be logically performed on multiple arguments of the same type (array components) with one request; or the function can logically generate multiple values of the same type with one request.

  - Each array component can be acted upon (or generated) in absence of all other components.

  - The result of the function relative to one component has no effect on the result of the function for any other component.

  - The order of the components has no bearing on the individual results.

o Record type parameters provide a convenient, useful interface in the bounds of the convention objectives if the record can be thought of (in the user's sense) as a single unified entity (i.e., no field of the record has particular significance in absence of any other field). If a field does not meet this criteria, it should be a discrete parameter.

  - A record parameter type will simplify interfaces and be convenient when the record is also a parameter of other external interface procedures and does not require user intialization or manipulation of contents - the user need only be concerned with the concept of the parameter, its structure and contents are transparent.

  - Each field should have an evident consistent relationship with the other fields of the record. Merely being parameters of a function does not establish the unified relationship.

  - If a field by itself has particular significance, that field should be a discrete parameter. Fields which are subfunction or option parameters to a function have such significance and should be discrete parameters.

  - A record type parameter should not contain fields which are superfluous to the execution of a function. Each field of an input parameter record should be essential to the execution of the function (i.e., each field should be a required argument). Each field of an output parameter record should contain a value returned by the function.

    - Record type parameters may contain superfluous fields if the fields are present for symmetry with other functions supporting the same concept. Use of this direction to justify superfluous fields should be minimized - superfluous fields will impair user understanding and result in excessive re-work at maintenance and extension time.

    - System architecture may dictate that some seemingly superfluous fields appear in a record to reserve space for data used internally by a function in support of other functions relating to the same concept - this is justifiable.

- A record type parameter should be solely an input parameter or solely an output parameter (i.e., a record should not contain some fields which are input parameters and other fields which are output parameters).

o Input parameters should not be pointers (CYBIL pointer type) to internal objects - validation of the pointer object would be virtually impossible.

o Pointers to internal objects (output parameters of CYBIL pointer type) must not be returned to the user - unnecessary exposure of internal data will result if such pointers are returned.

o Pointer type formal parameters should be declared only when the pointer object of the actual parameter can take one of several types (i.e., the pointer object type is <u>not</u> known at compile-time, but is resolved at execution-time). The formal parameter pointer type should ultimately resolve to '^cell'.

o Packed structures, adaptable types, and bound variant records have some applicability in external interfaces, but their use should be the exception rather than the norm.

16.0 PROGRAM LIBRARY CONVENTIONS

16.1 DECK NAMING CONVENTIONS

Deck names have the following format:

```
          PPCZZZZ
     where

          PP = two character product identifier
          C = one character indicating deck class
          ZZZZ = one to four character mnemonic  for  uniqueness
               within product
```

Allowable codes for deck type are as follows:

```
  M = CYBIL code module
  X = CYBIL xref declaration (common deck)
  D = CYBIL type and const declarations (common deck)
  H = Documentation header (common deck)
  I = CYBIL internal in-line procedure (common deck)
```

Note that decks of type M must consist of <u>exactly</u> <u>one</u> module (compilation unit).

When converting to the source code utility (SCU) all XREF declarations, documentation headers and module decks can be renamed. The new deck name will have the same three character prefix but the suffix (ZZZZ) can be the full name (up to 28 characters) of the item contained in the deck.

16.2 COMMON DECK USAGE

Common decks are restricted to four classes of usage:

- XREF declarations to be used by modules accessing procedures or variables defined in another module.

- TYPE and CONST declarations to be shared by modules dealing with the same data types or constants.

- Documentation header text describing an interface. A common deck of this type must be called from the module which contains the XDCL definition of the interface being described.

- Procedure declarations which may be expanded in-line as part of calling modules; as opposed to being called through an XDCL/XREF interface. Internal in-line procedures may occasionally be the most practical way to implement a "module" (in the Structured Design sense) due to performance and/or scope considerations. All common decks of this type are considered internal interfaces and must be documented accordingly. A procedure implemented in this fashion must not be dependent on the static chain, i.e. it must be completely self-contained.

16.3 COMMON DECK CONTENT

16.3.1 DOCUMENTATION HEADER

16.3.1.1 Procedures

The procedure documentation header consists of CYBIL comments which describe the procedure, its calling sequence and parameters. The general format for the procedure documentation header is as follows:

```
   123456789012345...
 1){}
 2){    The purpose of this request is to ...
 3){  whatever this request does.
 4){}
 5){        XXP$REQUEST_NAME (FIRST_PARAM, ...,
 6){          LAST_PARAM)
 7){}
 8){  FIRST_PARAM: (input) This parameter specifies ...
 9){        whatever this parameter specifies.
10){}
11){  LAST_PARAM: (output) This parameter specifies ...
12){        whatever this parameter specifies.
13){}
```

where:
line 1: blank comment line
line 2: indent 4: describe the purpose of the request
line 3: indent 2: for purpose continuation, if necessary
line 4: blank comment line
line 5: indent 8: request calling sequence; use all capital letters; parameter names must be the same and must be in the same order as in the XREFed procedure declaration

```
line 6: indent 10 for parameter continuation if necessary
line 7: blank comment line
line 8: indent 1: describe first parameter; specify whether it
          is input, input-output, or output
line 9: indent 8: for parameter description continuation, if
          necessary
line 10: blank comment line separates each parameter
line 13: blank comment line
```

Also, when listing parameters one should strive to list all input parameters first followed by input-output parameters followed by all output parameters unless there is an obvious symmetry with other requests that would be violated. The status parameter, if present should always be the last parameter on every request.

16.3.1.2 Data Structures

Each data structure will include a documentation header consisting of CYBIL comments which describe what the structure is for and how it is used. The general format is as described for the "purpose" section of the procedure header.

16.3.2 XREF DECLARATION COMMON DECK

The XREF declaration common deck contains a CYBIL XREF declaration followed by a *callc to all of the TYPE or CONST declaration common decks ("D" decks) necessary to compile this declaration in isolation (assume a CYBIL module only calls one XREF declaration common deck).

It is very important that all XREF declaration common decks perform *callc's (instead of *call) to necessary decks. This prevents duplicate definitions of identifiers in the caller's CYBIL module.

Example:

```
AMXREWD
COMMON

     PROCEDURE [XREF] amp$rewind(file_identifier:
               amt$file_identifier;
               wait:ost$wait;
          VAR status:ost$status);

??  PUSH (LIST := OFF, LISTEXT:=ON ) ??
```

```
     *callc amdfid
     *callc osdwnw
     *callc osdstat
     ??  POP ??
```

16.3.3 TYPE / CONST DECLARATION COMMON DECK

The TYPE / CONST declaration common deck contains CYBIL TYPE and/or CONST declarations followed by a *callc to all of the declaration common decks necessary to compile this common deck in isolation.

It is very important that the declaration common decks perform *callc's (instead of *call) to common decks. This prevents duplicate definitions of identifiers in the caller's CYBIL module.

Example:

```
     AMDNAME
     COMMON

     TYPE
       amt$local_file_name = ost$name;

     *callc osdname
```

16.3.4 EXAMPLE DECK

In order to be certain that interfaces provided for the end-user or other functional areas are specified accurately and consistently, each contributor should produce an example compilation unit that includes references to all type and procedure declarations he/she is responsible for and an example of the usage of each interface. By compiling all declarations, the checking logic of the compilers will aid accuracy and consistency; by trying examples of the interface, the contributor will gain a feeling for the efficacy of the interface.

17.0 CYBIL CODING CONVENTIONS

This document specifies the CYBIL coding conventions suggested for the CYBIL users. There are several general aims of coding conventions which underlie all of the specific proposals that follow:

1. There are a variety of routine, mundane aspects associated with writing programs: a set of coding conventions remove from the programmer trivial decisions relating to module format, name generation, etc. thereby leaving more time to concentrate on important matters.

2. The primary purpose of documentation and the readability of source code is to help someone other than the developer understand what is going on.

3. During the lifetime of a large software product like an operating system or a compiler, the average developer will come in contact with a large number of modules written by and maintained by many other programmers. A consistent set of coding conventions helps the programmer "feel at home" with a new module and therefore is able to begin doing useful work sooner.

4. To as great an extent as reasonable, all coding conventions should be generated and reinforced by automated methods.

5. Source code is the ultimate documentation of any program, particularly a program written in a higher level language such as CYBIL. Therefore, in all CYBIL programming, a consistent emphasis should be placed on producing lucid, readable, self-documenting code.

6. All commentary in the source code should be written so that it: a) only provides information not readily apparent from reading the code and b) is of a sufficiently algorithmic nature such that it rarely, if ever, becomes obsolete as changes are made to the code.

17.1 USAGE OF A SOURCE CODE FORMATTER

The major software tool for generating and enforcing CYBIL coding conventions should be the source code formatter (CYBFORM).

17.2 USE OF CYBIL

. Use block structure to articulate program structure: a declaration should always be declared at the "lowest" level possible.

. Do not use the static chain: in general a procedure should only reference arguments, its own automatic variables and static variables.

. In general, interfaces between modules should be procedures or functions, not XDCL/XREF variables.

. Always use label names that describe the process being performed by the structured statement to which the label refers.

. Always repeat the label in the terminating statement of a structured statement (the formatter will do this): e.g.:

```
  /search_symbol_table/
    for i := 1 to 10 do
       ...
    forend /search_symbol_table/;
```

. In general avoid the use of type INTEGER; few variables require subranges that large.

. In declarations of procedure parameter lists, always separate each formal parameter with a semicolon marking each with a VAR or "absence of VAR" as appropriate.

. Always declare all input parameters before all output parameters unless there is an obvious symmetry that would be disturbed.

. Cover all end cases. CASE statements should cover all statements with ELSE being used to cover "unplanned" cases.

. Procedures and functions should be used for two purposes: 1) "subroutines", 2) to "structure" the program thereby making the function of the program obvious at a high level.

. Arguments to procedures should also be used for two purposes: 1) "subroutine parameters", 2) as documentation which allows the reader to see all data referenced by the procedure by looking at the procedure call statement. In the latter case, the formal and actual parameter names should be the same.

. Trailing comment delimiter of '}' should be used whenever reasonable: i.e., use of EOL as a comment delimiter is discouraged.

. In compound arithmetic, conditional or relational expressions, use parenthesis to denote precedence. Do not depend on the language operator precedence rules.

. Avoid the #LOC function like the plague.

17.3 USE OF THE ENGLISH LANGUAGE

The key to making programs readable is the usage of meaningful, non-cryptic English names for all CYBIL constructs; specifically:

. When naming type identifiers and record fields, particularly fields, consider the way the name will look in the code, not the declaration; e.g.:

```
TYPE
  program_descriptor = record
    load_map; load_map_options,
  recend,
  load_map_options = record
    file_name : file_name,
    options : (all,nothing),
  recend;
VAR
  my_program : program_descriptor;
          ...
  my_program.load_map.file_name := "LOADMAP";
```

. Procedure and function names should describe the process the procedure performs.

. Labels should always describe the function being performed by the structured statement to which they refer; e.g.:

```
/search_symbol_table/ {instead of}
/l1/
```

Labels are a powerful documentary aide and their usage is encouraged.

. Booleans should always describe the TRUE condition; e.g.:

```
     if file_is_open then {instead of}

     if file_switch then
```

17.4 CYBIL NAMING CONVENTION

It cannot be emphasized too strongly that names should be chosen for how they will read in the code body of a procedure, not how they look in the data declaration. This is particularly true of variables and field names in type declarations.

The system naming convention for the user interfaces is described in the System Interface Standard (SIS). That is also the convention for linkage (entry-point or external) names. However, local names should use no convention other than English. For convenience, selected portions of the SIS naming conventions are reproduced below:

System global names will be generated according to the following convention:

PPC$XXX...

where:

PP	=	is a two character product identifier for the owner of this name.
C	=	identifies the class of the name.
$	=	is the special character '$'.
XXX	=	a meaningful English expression or abbreviation that describes or denotes the purpose of the item being named.

Class of Names:

C	constant
E	exception condition name
F	file
M	module
P	procedure
S	section
T	type
V	variable

17.5 MODULE AND PROCEDURE DOCUMENTATION

Standard documentation for each module and each XDCLed procedure or function within a module should be provided. The procedure documentation is also encouraged for local procedures and functions as well. Care should be taken to minimize commentary becoming outdated as changes are made to the code.

```
MODULE <module identifier>;

{  PURPOSE:
{    This should contain the purpose of the module and the
{    reasons for grouping these declarations in the module rather
{    than the purpose of each procedure.
{  DESIGN:
{    This should contain an overview of the module design; i.e.,
{    an outline of how it works in general terms.  Usage of
{    specific variables or procedure names is discouraged in this
{    description.

<procedure or function declaration>;

{  PURPOSE:
{    This should describe the process the procedure or
{    function performs rather than the method used.
{  NOTE:
{    This should contain information of interest to the
{    user or maintainer.
```

17.6 TITLE PRAGMATS

Each module should be titled in the following way:

```
     <major product identifier>[:<component identifier>...]
     <sp><sp>[[XDCL]]<procedure identifier>|<section identifier>
```

for example:

```
     NOS : task establisher
       [XDCL] pmp$establish_task
```

17.7 COMMENTING CONVENTIONS AND GUIDELINES

In general, comments should be standalone blocks describing why or what a series of CYBIL statements are doing. Care should be taken not to use comments that will become outdated by detailed changes to the code. The basic concept behind comments should be to provide nonredundant information. Comments should be preceded and followed by a blank line and start in the first available source character on the line. Again, remember that the purpose of comments is to help someone other than the original developer of the module understand what the module is doing.

17.8 PROCEDURE AND DATA ATTRIBUTE COMMENT CONVENTIONS

Comments should also be used to convey software or system attributes which are not discernable from CYBIL declarations. These comments should be concise and abut CYBIL declaration constructs rather than being standalone blocks.

18.0 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.

These ideas are guidelines, they should be followed only when clarity of code is not compromised.

18.1 SOURCE LEVEL EFFICIENCIES

18.1.1 GENERAL

o 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.

o 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.

o A copy is currently being made of all value parameters. This implementation is subject to change.

o Assignment of records is done with one large move, while record comparison is done field by field.

o Move structures rather than lots of elementary items. This may require structuring the elements together especially for this purpose.

o Reference to adaptable structures are slower than references to fixed structures because the adaptable has a descriptor field which must be accessed.

o References to fields within a record require no execution penalty.

o 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 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:

  - 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 for the nominal case.

- 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).

o 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 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.

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

o Certain conversion functions (i.e.,0RD,CHR,etc.) require no execution time overhead.

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

o 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.

18.1.2 CC EFFICIENCIES

o 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.

o 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.

o 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.

18.1.3 CI/II EFFICIENCIES

o The adaptable string bound construct should be quoted whenever possible to give the compiler a clue as to the maximum length. This will often result in more efficient code being generated for adaptable strings.

o References to XDCL variables and variables declared within a SECTION will be made via the binding section and, consequently, an overhead is associated.

o The code generator does not move invariant code out of loops. Consequently, access to variables through the binding section within a loop would be more efficient if the initial access to the variable is outside the loop.

o The reach of the load & store instructions on the Advanced System is limited to 2**16. When using large variables the offset may become greater than this threshold and result in an extra instruction being generated to handle the large offset. This would indicate organizing the more frequently used variables first in very large user stacks.

18.1.4 CM EFFICIENCIES

18.1.5 CP EFFICIENCIES

- o The UCSD p-system does not have an exclusive or instruction. Therefore, set references using the XOR operator generates a lot of code.
- o Using long integer subranges results in less efficient code.
- o The most commonly used variables should be entered first in the list of variables for a procedure. The first n variables in a procedure are accessed by a 1 byte instruction, the others by 2 bytes. Arrays should be the last variables in the list.
- o Using global variables in other modules should be avoided.
- o Avoid FOR statements in favor of WHILE or REPEAT. They are faster and produce less code.
- o Nested IF statements are more efficient than a single IF with AND connectors. e.g. use:

```
IF condition1 THEN
  IF condition2 THEN
    statements...
  IFEND;
IFEND;
```

- o Use base 0 for arrays rather than 1. E.g. use "ARRAY [0 .. n-1]" instead of "ARRAY [1 .. n]".

18.2 COMPILATION EFFICIENCIES

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

- o The generation of information to interface to the symbolic debuggers slows the compilation process.
- o The generation of stylized code slows the compilation process.
- o The generation of range checking code slows the compilation process.

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

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

o 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.

19.0 IMPLEMENTATION LIMITATIONS

19.1 GENERAL

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

19.2 CC LIMITATIONS

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

19.3 CI/II LIMITATIONS

- o Maximum number of lines in a single compilation unit is 32767 when run time error checking is selected.
- o Nesting level of structured statements is limited to 63 levels deep.
- o FOR statements can only be nested 15 levels deep.
- o Procedures may only be nested 50 levels deep.
- o Number of parameters passed to an xrefed procedure is 127, while

an xrefed function is limited to 126.

- The reach of jump instructions is limited to 2**16 so the size of compilation units should be appropriately controlled.

- The stack size of a single procedure is limited to 2**15 bytes.

19.4 CM LIMITATIONS

19.5 CP LIMITATIONS

- In general the size of arrays and strings should be limited to less than 2**15 bytes.

- Maximum number of procedures in a single module is limited to 254.

- The maximum nesting level of procedures is 30.

- The use of long integer subranges is not allowed in the following areas:
    - Array subscripts,
    - As the <first char> or as the <substring length> on any string reference,
    - As the selector on a case statement,
    - As a actual parameter to a formal reference parameter of type integer.

- The result of a Stringrep operation on a floating point number is limited to 6 digits.

19.6 CS LIMITATIONS

20.0 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.

20.1 GENERAL

CYBIL Implementation - Deviations

- o Support for $CHAR. **
- o Support adaptable arrays of zero dimension. **
- o Double Precision Floating Point. **
- o Initialization of static pointers to NIL and zeroing the adaptable descriptor fields is not done. **
- o #SIZE of adaptable types. **
- o Run time checking on accessing fields of variant records not supported. **
- o Restricting pointers to not point to data with less scope. **

- o Pre-defined identifiers are implemented as reserved words. **

20.2 CC DEVIATIONS

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

20.3 CI/II DEVIATIONS

20.4 CM DEVIATIONS

- o Relative Pointers. **
- o General Intrinsics. **

20.5 CP DEVIATIONS

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

20.6 CS DEVIATIONS

Table of Contents