Table of Contents

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.
H	07/31/84	Updated to reflect current status of the various CYBIL products.
I	09/03/86	Updated to reflect current status of the various CYBIL products.

1.0 INTRODUCTION

1.1 CYBIL COMPILER NAMING CONVENTIONS

The convention for naming the various CYBIL compilers is to refer to a particular compiler as CYBIL/xy where x indicates the host and y indicates the destination machine for that particular compiler. At this time, host machines are CYBER 170 (C), the C180 (I), the Apollo (A) and the C200 (S). Target machines besides those four are UCSD P-system (P), the Motorola MC68000 (M) and the Motorola MC68010 running UNIX (U). Thus CYBIL/II is a compiler running on C180 generating code for C180, while CYBIL/CM is a cross compiler running on C170 generating code for the Motorola MC68000.

1.2 STATUS OF AVAILABLE COMPILERS

COMPILER	STATUS
CYBIL-CC	NOS AND NOS/BE PRODUCT
CYBIL-CI	INTERNAL USE
CYBIL-CP	NOS PRODUCT
CYBIL-CM	NOS PRODUCT
CYBIL-CS	INTERNAL USAGE
CYBIL-CE	INTERNAL/ETA USAGE
CYBIL-CA	INTERNAL USE
CYBIL-CN	INTERNAL USE
CYBIL-CU	INTERNAL USE
CYBIL-II	NOS/VE PRODUCT
CYBIL-IM	INTERNAL USE
CYBIL-IU	INTERNAL USE
CYBIL-SS	INTERNAL USE
CYBIL-EE	INTERNAL/ETA USAGE
CYBIL-AA	INTERNAL/ETA USAGE

C - CYBER 170, P - PCODE MACHINE, M - MC68000,
I - CYBER 180, S - CYBER 200, A - APOLLO,
N - INTEL 8086, E - GF-10, U - MC68010/UNIX

2.0 COMPILING AND EXECUTING CYBIL PROGRAMS ON NOS

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.

2.1 SES PROCEDURE INTERFACE

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

2.2 THE NOS 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.
```

2.3 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 four 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. Verifies that all assignments in sets are within the bounds of that set. 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.

DEBUGGING OPTION (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 (SD parameter) plus stylize the code generated. This option is currently not supported on all compilers.

SD Symbolic Debug. Produce a symbol table and line table for interactive debugging. Also generates a NIL value for initialization of static pointers & in the case of adaptable pointers the descriptor field(s) are zeroed.

When more than one option is desired the format is D=DSSD.

SOURCE INPUT (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>.

LIST OUTPUT (Default: L=OUTPUT)

L Compilation listing is written on file LISTING. When the LIST OUTPUT parameter is omitted file OUTPUT 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>.

LIST FORMAT (Default: LF=CS612)

LF or LF=CS612 Compliation listing file, if selected is output in the NOS 6/12 character set.

LF=CS812 Compliation listing file, if selected is output in the NOS 8/12 character set.

LIST OPTIONS (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. Using both the RA and the A options (i.e. LO=ARA) causes both the cross reference listing and the attribute listing to be provided for 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.

OPTIMIZATION (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.

PADDING (Default: PAD=0) (Not supported on all processors)

PAD=<number> Provides for generation of NOOP type instructions between live instructions.

2.4 INTERACTIVE CYBIL ON NOS

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).

2.5 BATCH CYBIL ON NOS

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.

3.0 COMPILING AND EXECUTING CYBIL PROGRAMS ON NOS/BE

3.0 COMPILING AND EXECUTING CYBIL PROGRAMS ON NOS/BE

3.1 THE NOS/BE 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 compiler currently reside in the project catalog LP3. To access the CYBIL compiler:

```
ATTACH,CYBIL,ID=LP3.
```

To access the CYBIL-CC run time library:

```
ATTACH,CYBCLIB,ID=LP3.
```

3.2 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/BE REFERENCE MANUAL (Publication number: 60493800), 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.

3.0 COMPILING AND EXECUTING CYBIL PROGRAMS ON NOS/BE
3.2 COMMAND PARAMETERS

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 four 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. Verifies that all assignments in sets are within the bounds of that set. 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.

DEBUGGING OPTION (Default: D=OFF)

D=<chars> Selects the following option.

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.

SOURCE INPUT (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>.

The compiler expects the input file to be in 8 in 12 format.

LIST OUTPUT (Default: L=OUTPUT)

L Compilation listing is written on file LISTING. When the LIST OUTPUT parameter is omitted file OUTPUT 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>.

The compiler generates the output file in 8 in 12 format.

LIST OPTIONS (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 O.

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. Using both the RA and the A options (i.e. LO=ARA) causes both the cross reference listing and the attribute listing to be provided for 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.

3.3 INTERACTIVE CYBIL ON NOS/BE

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/BE operating system. After you have logged in:

NOS/BE COMMAND	DESCRIPTION
ATTACH,SOURCE,ID=...	GET CYBIL SOURCE PROGRAM TEXT
ATTACH,CYBILC,ID=LP3	ATTACH CYBIL COMPILER
CYBIL,I=SOURCE,L=LISTING	COMPILE CYBIL SOURCE TEXT
ATTACH,DATA,ID=...	GET DATA FILE
ATTACH,CYBCLIB,ID=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 (CYBIL,I=SOURCE,L=LISTING).

3.4 BATCH CYBIL ON NOS/BE

A CYBIL compilation and execution may be run as part of a NOS/BE 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/BE 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 batch job submission is shown below.

```
XYZ,T100.                  PROGRAMMER NAME
ACCOUNT.USE,PW,FMLY,ABC    SUBSTITUTE APPROPRIATE INFORMATION
ATTACH,SOURCE,ID=...       ACQUIRE CYBIL SOURCE FILE
ATTACH,CYBIL,ID=LP3.       ATTACH CYBIL-CC COMPILER
CYBIL,I=SOURCE.            COMPILE CYBIL SOURCE
ATTACH,DATA,ID=...         GET FILE OF DATA
ATTACH,CYBCLIB,ID=LP3.     GET CYBIL RUN TIME LIBRARY
LGO.                       EXECUTE PROGRAM
EXIT.                      EXIT HERE ON ERRORS
```

Using this approach, the programmer need not sit at (or tie up) a terminal unnecessarily while the program is actually compiling and executing.

4.0 COMPILING AND EXECUTING CYBIL PROGRAMS ON THE C180

4.0 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.

4.1 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=<file reference>]
      [list_options=<options>]
      [debug_aids=<options>]
      [error_level=<options>]
      [optimization_level=<options>]
      [instruction_scheduling=<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.

4.2 C180 COMMAND PARAMETERS

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

PARAMETER NAMES	PARAMETER DESCRIPTION
BINARY B BINARY_OBJECT	(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_AIDS DA	(Default: DA=NONE) Selects a combination of the following debug options. If this parameter is omitted the default is DA=NONE. ALL All of the available options are selected for the Debug_Aids parameter. 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. DT Debug Tables. Generate line number and symbol tables as part of the object code. In addition, also generate a NIL value for initialization of static pointers & in the case of adaptable pointers the descriptor field(s) are zeroed. NONE No options are selected for the Debug_Aids parameter.

ERROR_LEVEL (Default EL=W)
EL

F List Fatal Diagnostics. If this option is selected only fatal diagnostics will be listed.

W List warning as well as fatal diagnostics.

INPUT (Default: I=$INPUT)
I

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>.

INSTRUCTION_SCHEDULING (Default: IS=NO)
IS

This parameter specifies whether or not instruction scheduling will be performed. Quoting IS=YES selects this option, and must be accompanied with the selection of OPTIMIZATION_LEVEL = HIGH. This option is not currently supported.

LIST (Default: L=$LIST)
L

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 (Default: LO=S)
LO

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_LEVEL (Default: OPT=LOW)
OL
OPTIMIZATION
OPT

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=NONE)

Selects a combination of the following options.

ALL All of the available options are selected for the Runtime_Checks parameter.

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

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.

This cannot currently be selected in combination with OPTIMIZATION_LEVEL = HIGH.

STATUS (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.

4.3 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"
quit 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"

4.4 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"
quit 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"
prif 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_FILE command:

```
include_file $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.

5.0 COMPILING AND EXECUTING CYBIL PROGRAMS ON THE APOLLO SYSTEM

5.0 COMPILING AND EXECUTING CYBIL PROGRAMS ON THE APOLLO SYSTEM

5.1 THE APOLLO CYB COMMAND

The CYB 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 form:

```
CYB input_file [-switch] [option] [-switch] [option] ...
```

Example:

```
CYB ABC -L -B
```

The compiler will look for the input file as ABC.CYB, create the list file as ABC.LST and create the object file as ABC.BIN.

5.2 APOLLO COMMAND SWITCHES

These switches were chosen using Apollo Pascal compiler as a guide line. If "N" is the first character in any switch, that indicates the switch is turned off.

SWITCH	DESCRIPTION
-[N]A	(Default: -NA) If this switch is omitted, the compiler attempts to compile the entire input file no matter how many FATAL errors are encountered. If this switch is specified, the compiler will terminate after the first FATAL error is found.
-[N]B [pathname]	(Default: -NB) If this switch is omitted no object code is created. If this switch is specified as -B, the compiler will create an object file and the name of the file will be the input file name with .BIN appended to it. If a pathname was specified, that name will be the name used for the object file.

-[N]CHK [options] (Default: -NCHK)

Selects a combination of the following options for runtime_checking.

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.

If the switch is specified as -CHK, then all of the above options will be set.

-[N]COND (Default: -NCOND)

IF -COND is selected, then all conditional statements will be compiled. A conditional 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.

-[N]DB (default: -DB)

If this switch is specified, the compiler includes Line Number information in the object text.

-[N]DBA (Default: -NDBA)

If this switch is selected, the compiler includes Line Number and Symbol Table information in the object text. In addition, also generate a NIL value for initialization of static pointers & in the case of adaptable pointers the descriptor field(s) are zeroed.

-[N]EXP (Default: -NEXP)

When -EXP is selected, the list file includes an assembly like listing of the generated object code.

-IDIR

This switch enables you to give up to 63 alternate pathnames for INCLUDE files. The format of the switch is as follows:

-IDIR pathname [-IDIR pathname] ...

The compiler builds a search list of pathnames specified in each IDIR switch (up to 63 may be defined). If any INCLUDE file cannot be successfully opened by it's given name, the compiler concatenates each pathname in the search list to the input file name given and attempts to open the file. IF successful, that file will be used as the INCLUDE file.

If the INCLUDE name begins with ".", "¬", or "/", searching beyond the given name will not be attempted.

This option is not currently supported.

-[N]L [pathname] (Default: -NL)

If this switch is specified as -L, then the list file is given the name of the input file except .LST replaces .CYB as the suffix.

If the pathname is specified, that name will be used as the name for the list file.

-LO [options] (Default: -LO S)

Selects a combination of the following list options. For example, -LO AX

A Lists compiler generated object code. When A 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.

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

L Works in conjunction with the LISTEXT pragmat such that LISTings can be EXTernally controlled on the compiler call statement.

M 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.

R Symbolic cross reference listing showing location of program entity definition and use within a program.

S Lists the source input file.

X Symbolic cross reference listing of all program entities whether referenced or not.

If the parameter is specified as -LO 0, no list options are selected.

-[N]OPT (Default: -NOPT)

IF -OPT is selected, then the compiler eliminates redundant memory references and common subexpressions.

6.0 COMPILING AND EXECUTING CYBIL PROGRAMS ON THE C200

6.0 COMPILING AND EXECUTING CYBIL PROGRAMS ON THE C200

6.1 THE CYBIL C200 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 form:

```
CYBIL [input=<file reference>]
      [list=<file reference>]
      [binary=<file reference>]
      [lo=<options>]
      [debug=<options>]
      [elev=<options>]
      [optimize=<options>]
      [pad=<number>]
      [rc=<options>]
```

Example:

```
CYBIL I=COMPILE L=LIST B=BIN1.
```

6.2 C200 COMMAND PARAMETERS

The keywords were chosen using the guide lines defined in the CYBER 200 KEYWORD STANDARDIZATION REPORT (ARH3749). If a short form of a keyword is defined below, it is the minimum required, one can type in more than the minimum and still not use the long form.

PARAMETER NAMES	PARAMETER DESCRIPTION
BINARY B	(Default: B=BINARY) If this parameter is omitted object code is written to file BINARY. If this parameter is specified as B=0, 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
DEB

(Default: no option selected)

Selects a combination of the following debug options.

D 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.

S If this option is selected, the compiler includes Symbol Table information ("level 0" symbols only, no symbols within procedures can be displayed) and a line table for each module compiled.

ELEV

(Default: ELEV=W)

Selects the level of diagnostics that will be listed.

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-group or end-of-file 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=OUTPUT)

When the LIST parameter is omitted the compilation listing is written on file OUTPUT.

If the parameter is specified as L=0, all compile time output is suppressed.

If the parameter is specified as L=<file reference>, the compilation listing is written on file <file reference>.

LO (Default: LO=S)

Selects a combination of the following list options. For example, LO=AX.

A Lists compiler generated object code. When A 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.

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

L Works in conjunction with the LISTEXT pragmat such that LISTings can be EXTernally controlled on the compiler call statement.

M 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.

R Symbolic cross reference listing showing location of program entity definition and use within a program.

S Lists the source input file.

X Symbolic cross reference listing of all program entities whether referenced or not.

If the parameter is specified as LO=0, no list options are selected.

OPTIMIZE
OPT (Default: OPT=0)

Selects the code optimization level.

0 Provides for keeping constant values and local variables in registers.

7.0 APPLICABLE DOCUMENTS

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

7.1 GENERAL

- CYBIL Language Specification (External-60457280, Internal-ARH2298)
- This CYBIL Handbook (External-60457290, Internal-ARH3078)
- CYBIL Formatter ERS (External-60461810, Internal-ARH2619)
- Common CYBIL I/O Reference Manual (ARH6794)
- User Guide for Analyze CYBIL Complexity (ARH7399)
- CYBIL User's Guide (Internal-SESD006)
- NOS Graphics User's Manual (60457270 03)*
- NOS Define Data Dictionary Tools User's Handbook (60457260)*
- TXTCODE User Guide (External-60460280, Internal-ARH2893)
- TXTFORM User Guide (External-60460290, Internal-ARH1737)

7.2 C170

- CYBIL Reference Manual (60455280)*
- CYBIL I/O Reference Manual (External-60460300, Internal-ARH2739)
- CYBIL Debugger ERS (External-60460320, Internal-ARH3142)
- CYBIL Miscellaneous Routines (External-60460310, Internal-SESD003)*
- SES User Handbook (External-60457250, Internal-ARH1833)*
- SES Procedure Writer's Guide (External-60460270, Internal-ARH2894)
- NOS Screen Formatting Reference Manual (60460430)

7.3 C180

- CYBIL for NOS/VE Language Definition (60464113)*
- CYBIL for NOS/VE System Interface Usage (60464115)
- CYBIL for NOS/VE File Interface Usage (60464114)
- CYBIL for NOS/VE Keyed-file & Sort/Merge Interface (60464117)
- NOS/VE Analysis Usage (60463915)
- NOS/VE Program Interface ERS (ARH3610)
- AAM 180 ERS (S2978)
- Post R1 AAM ERS (S4257)
- Debugger ERS (S4028)
- System Interface Standard (S2196)

7.4 MC68000

- CDCNET CYBIL Reference Manual (60462400)*
- CDCNET Motorola 68000 Utilities Reference Manual (External-60462500, Internal-ARH5194)
- ERS for the MC68000 Absolute Linker (ARH4895)
- ERS for SES Object Code Utilities (External-60460330, Internal-ARH2922)
- CDCNET M68000 Cross-Assembler Reference Manual (External-60462700, Internal-ARH6363)

7.5 APOLLO

- APOLLO Aegis Domain System Programmer's Reference Manual (0005)

7.6 PCODE

- CYBIL P-Code Generator Reference Manual (60461290)*
- UCSD P-system Internal Architecture Guide (SofTech Microsystems, Inc.)
- ERS for SES Object Code Utilities (External-60460330,

Internal-ARH2922)

7.7 C200

- C200 Standards and Conventions (17329020)

* Indicates the manual is also available via an online manual.

8.0 COMMON CYBIL COMPILERS

This section details the characteristics of all CYBIL compilers.

8.1 CONSIDERATIONS IMPOSED BY THE NATURE OF CYBIL

CYBIL is a block-structured, recursive language, offering a variety of data structures, and requiring storage management features not found in static languages such as FORTRAN, IMPL or even assembler.

Of critical importance is an understanding of lexical level, scope of identifiers and the implication of recursion.

The visual appearance of a CYBIL module is a static map of the scope of its contained declarations. Each declaration at the outermost level is considered STATIC to the module. Within any declared procedure in the module, contained declarations are at a higher level (smaller scope) than those outside the procedure. Since procedures may contain procedure declarations, the process compounds. If we visualize a counter set to zero at the beginning of the module, and incremented as each procedure declaration is discovered (decremented when that procedure declaration is terminated), we have defined the lexical level for any declaration encountered.

The scope of an identifier is the environment containing it, both statically and dynamically. Hence, the set of identifiers available to a procedure at a given lex level is the set from all lower lex levels which contain the procedure (static scope). Dynamic scope is concerned with which values of an identifier set are available. Because of recursion, a given set of variables (static scope) may have several sets of dynamic values; the set available is the most recently created set (dynamic, or flow, scope).

Recursion is a generalization of the reentrancy problem. For code to be reentrant, two constraints must be met:

1) the code must not modify itself, and

2) a separate data space must be provided for each invocation of the code

The "separate data space" implies a dynamic allocation of the

space, since the number of invocations to be encountered is not predictable. Therefore, the code references the data space indirectly, relative to a supplied locator (pointer). Hence, the data space is based on a locator.

In order to permit recursion (a procedure or function may {indirectly} call itself), the reentrant data space must be again subdivided so that each invocation of a procedure or function may obtain data space for a new set of values for its data variables. Since the number of recursive invocations is not predictable, it is limited only by physical constraints. The variables so allocated at each invocation are called automatic, and have a dynamic lifetime equal to that of the activation of the procedure declaring them. Their values are lost when the procedure or function deactivates (EXITs).

8.1.1 STORAGE MANAGEMENT - DYNAMIC VS. STATIC

8.1.1.1 Stack Frame

In CYBIL terms, the stack frame is the set of automatic variables, parameters, control information, register save areas, etc. associated with a given invocation of a procedure. This frame must be dynamically allocated. Procedure formal parameters are treated as members of this stack frame, ie. as local variables. Special attention is paid to function return values; for a function, its return value is something static to it (i.e. like a formal parameter or a local variable); whereas for the invoker of a function, the return value is merely an operand in an expression. This contradiction must be accommodated.

8.1.1.2 Allocated Space

In addition to (automatic) stack frames, CYBIL requires the implementation of space acquisition for dynamic variables, the number of which is not declarable or predictable. These variables fall into two categories:

- space obtained by a PUSH statement,
- space obtained by an ALLOCATE statement.

A mechanism by which a variable amount of space may be obtained and allocated is thus required, and provided via the PUSH and ALLOCATE statements.

8.2 INLINE PROCEDURES AND FUNCTIONS IMPLEMENTATION

The CYBIL Language Specification lists language considerations for INLINE routines. Listed below are specific features of the implementation:

- Local variable declarations in an INLINE routine become part of the calling procedure's stack frame.
- Formal parameters are treated as local variable declarations in the INLINE routine. At the point of call to an INLINE routine the actual parameter is assigned to the corresponding formal parameter local variable. Reference parameters are accessed by assigning a pointer to the actual parameter to the formal parameter local variable.
- When the actual parameter for a value parameter is of an adaptable type or is a substring then the parameter is treated as though it were a read-only reference parameter, i.e. a local copy of the parameter is not created. This is necessary to allow type-fixing at execution time. A restriction is imposed on adaptable array/record value parameters that the actual parameter be aligned to a machine addressable boundary.
- The result of an INLINE function reference is part of the caller's stack frame. When one INLINE function is called more than once within a statement, the corresponding results are separate, although they share the same name.
- Nested calls to INLINE routines are arbitrarily limited to 5 levels of nesting on the assumption that an inappropriate amount of code expansion may be occurring when the nesting level becomes too great. Excessive call nesting levels and recursive calls are considered errors and terminate inline substitution.
- Source statements in an INLINE routine body are not listed at the point of call.
- No procedure or function declaration which is not XREFed may appear within an INLINE routine declaration.
- No STATIC or XDCL variable may be declared within an INLINE routine declaration.
- INLINE routines 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. The debugger considers an INLINE function call expansion to be a series of statements on the same line as the end of the phrase which includes the INLINE function reference. Local variables declared in an INLINE routine may not be accessible directly by name following an inline 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.

Note: Space reserved via a PUSH statement within an INLINE routine will not be de-allocated until the calling routine exits.

8.3 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.

9.0 CYBIL-CC DATA MAPPINGS

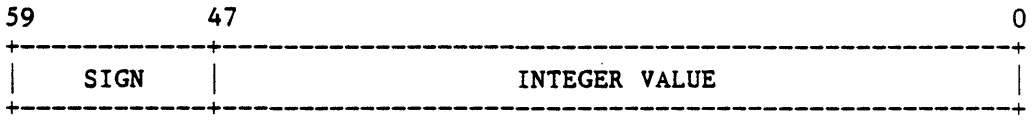
9.0 CYBIL-CC DATA MAPPINGS

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.

9.1 UNPACKED BASIC TYPES

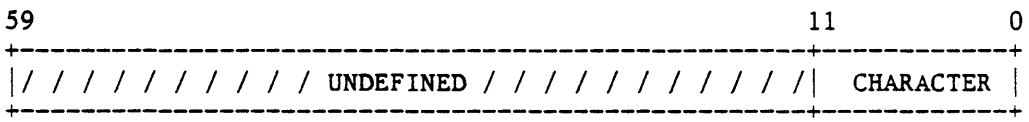
9.1.1 UNPACKED INTEGER

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.



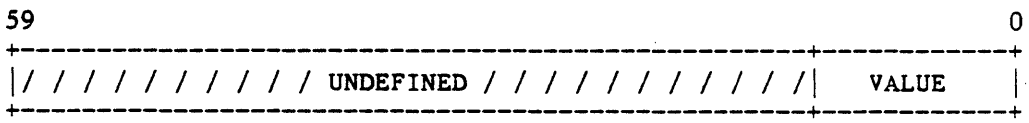
9.1.2 UNPACKED CHARACTER

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.



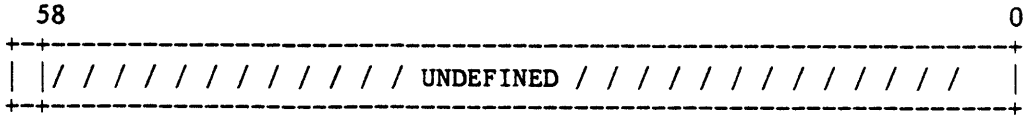
9.1.3 UNPACKED ORDINAL

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.



9.1.4 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.



9.1.5 UNPACKED SUBRANGE

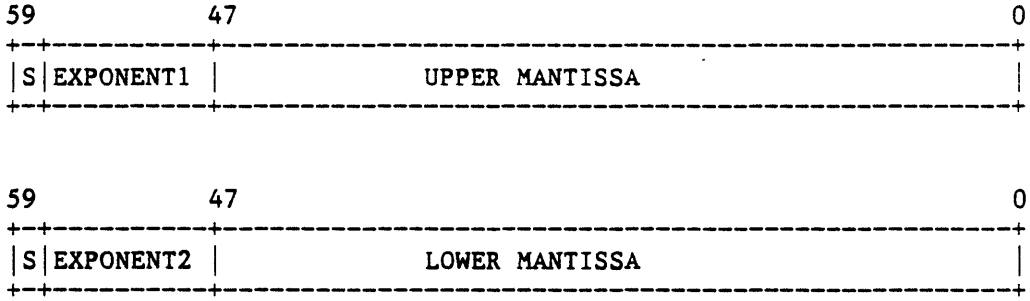
An unpacked subrange of any scalar type is represented in the same manner as the scalar type of which it is a subrange.

9.1.6 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.

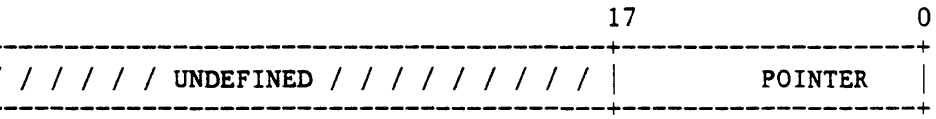
9.1.7 UNPACKED LONGREAL

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.



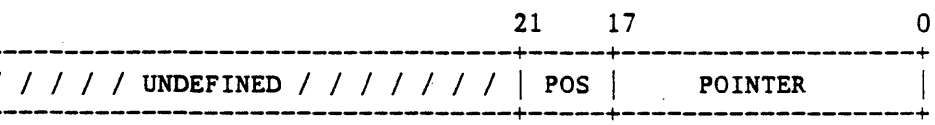
9.1.8 POINTER TO FIXED TYPES

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.



9.1.9 POINTER TO STRING

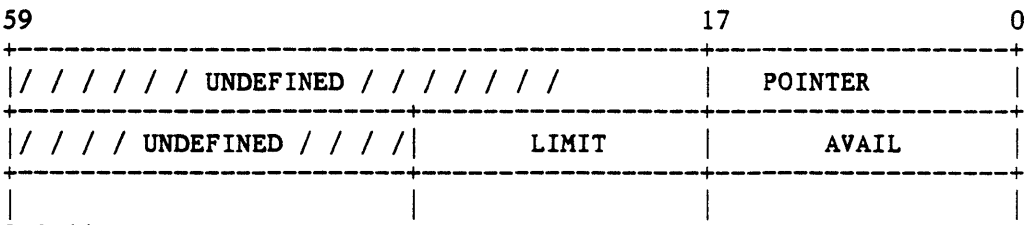
Pointers to strings are 18 bits long but have an additional 4 bit "position" field to indicate which of the ten positions (POS) in a CYBER word contains the first character of the string. A string may begin on any 12 bit boundary (bit positions 59,47,35,23, or 11). The POS field will contain a value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



9.1.10 POINTER TO SEQUENCE

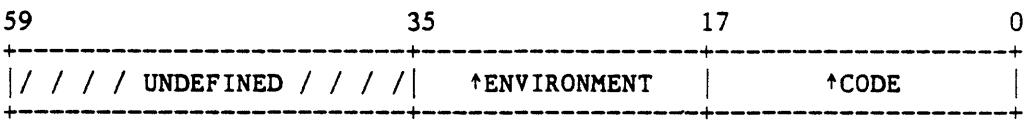
Pointers to sequences contain the pointer plus an additional descriptor word. This descriptor word contains an offset to the next available (AVAIL) location in the sequence and an offset to the top (LIMIT) of the sequence.

9.0 CYBIL-CC DATA MAPPINGS
9.1.10 POINTER TO SEQUENCE



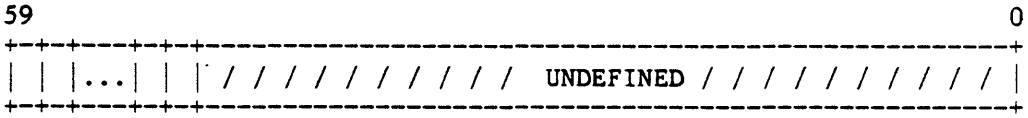
9.1.11 POINTER TO PROCEDURE

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.



9.1.12 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.



9.1.13 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.

9.0 CYBIL-CC DATA MAPPINGS
9.1.13 UNPACKED STRING

9.1.14 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.

9.1.15 UNPACKED RECORD

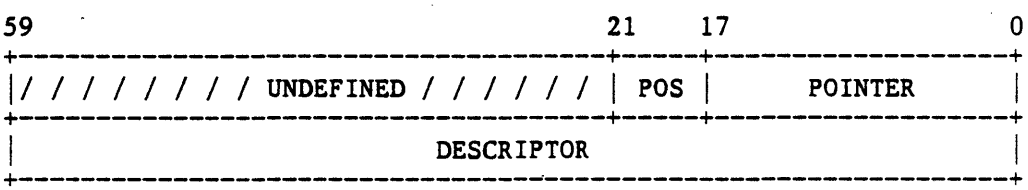
An unpacked record is a contiguous list of aligned fields.

9.2 OTHER TYPES

9.2.1 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.

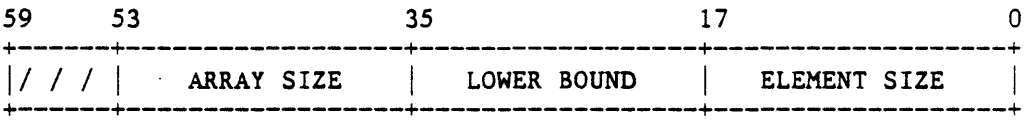
9.0 CYBIL-CC DATA MAPPINGS
9.2.1 ADAPTABLE POINTERS



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

9.2.1.1 Adaptable Array Pointer

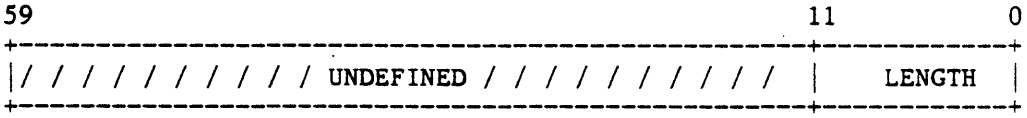
The descriptor for an adaptable array is:



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

9.2.1.2 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:

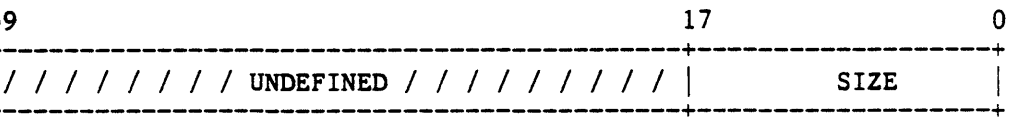


9.2.1.3 Adaptable Sequence Pointer

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

9.2.1.4 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:

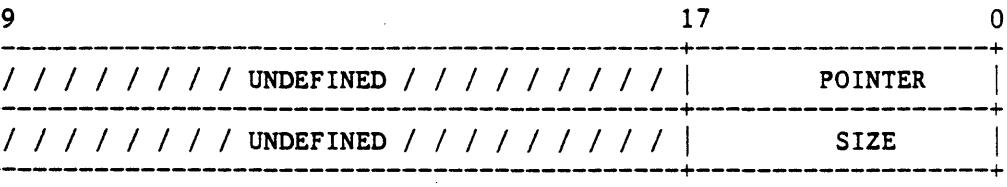


9.2.1.5 Adaptable Record

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.

9.2.2 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.



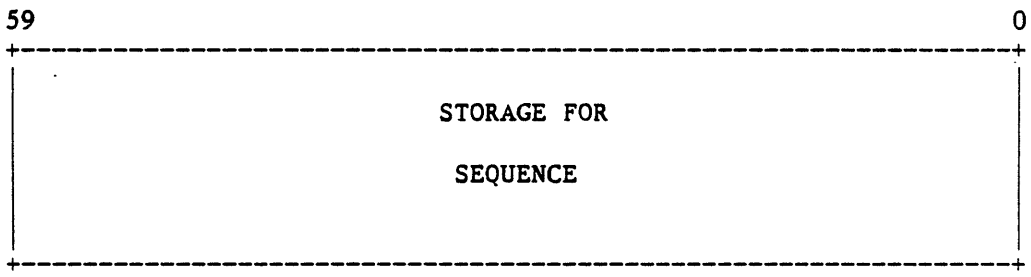
9.2.3 STORAGE TYPES

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 control information.

9.2.3.1 Sequences

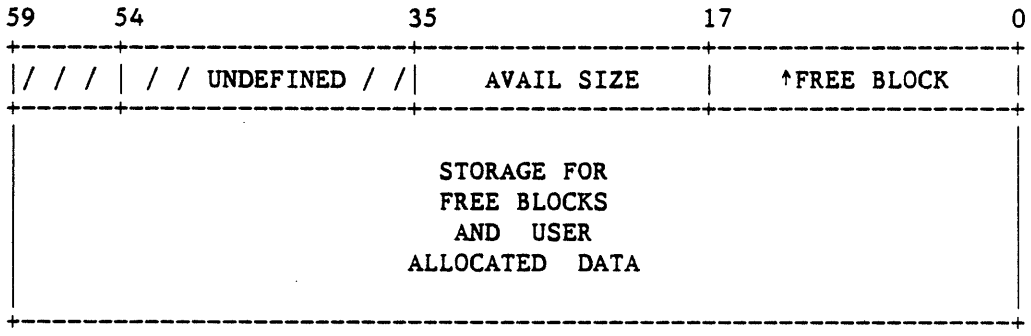
Access to a sequence is through the control information associated

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



9.2.3.2 Heaps

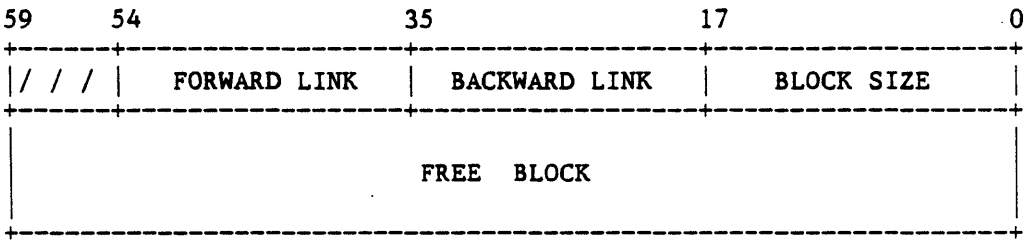
User declared heap storage must be managed differently than the sequence because explicit programmer written ALLOCATE's and FREE's may be executed. The heap, in general, consists of 1) a header word, 2) free areas (blocks) which are linked together (forward and backward) and 3) areas in use as a result 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:



9.2.3.2.1 FREE BLOCKS

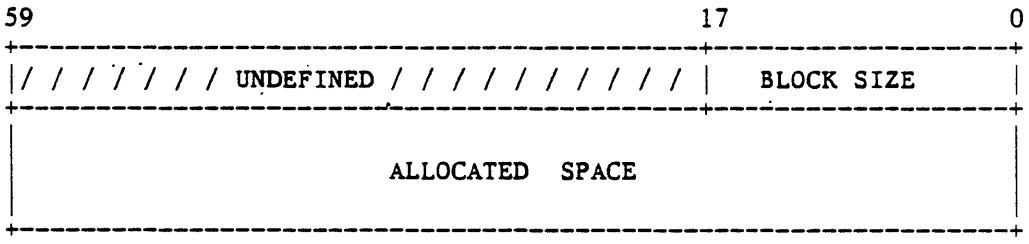
The free blocks are a circular forward and backward linked list. Free blocks are condensed each time the user code executes a FREE statement referencing this heap. The storage map of a typical free block is shown below:

9.0 CYBIL-CC DATA MAPPINGS
9.2.3.2.1 FREE BLOCKS



9.2.3.2.2 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:



9.2.4 CELLS

A cell is allocated a word and is always aligned.

9.3 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.

9.4 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.

10.0 CYBIL-CC RUNTIME ENVIRONMENT

10.1 STORAGE LAYOUT OF A CYBIL-CC PROGRAM

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

- The stack.
- Dynamically allocated memory.

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

10.2 REGISTER USAGE

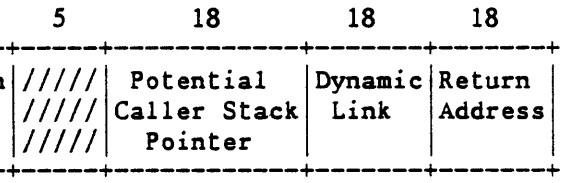
```
B0 = 0
B1 = 1 - the generated code counts on this
B2 = dynamic link - callers stack frame pointer (top of stack)
B3 = stack segment limit
B4 = static link - set before a nested procedure is called
B5 = pointer to extended parameter list

X1
X2   last 5 parameters passed to callee,
X3   that fit into an X register
X4   starting with X1
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.
```

10.3 LINKAGE WORD



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

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

The meaning of the fields is as follows:

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

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

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

RETURN_ADDRESS: Address to which the epilog will go to.

10.4 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.

10.5 CALLING SEQUENCES

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

10.5.1 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
         .
         .
         .
```

10.5.2 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
```

10.5.3 CALLING A PROCEDURE

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

10.6 PARAMETER PASSAGE

10.6.1 REFERENCE PARAMETERS

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

10.6.2 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.

10.7 RUN TIME LIBRARY

10.7.1 MEMORY MANAGEMENT

10.7.1.1 Memory Management Categories

Three categories of memory management occur for CYBIL programs:

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

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

10.7.1.2 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 1760(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 1760(8) until adequate memory is obtained.

10.7.1.3 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.

10.7.1.4 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).

10.7.1.5 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.

10.7.2 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.

10.7.3 SYSTEM DEPENDENT ACCESS

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

10.8 VARIABLES

10.8.1 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.

10.8.2 GATED VARIABLES

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

10.8.3 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.

10.8.4 VARIABLE ALIGNMENT

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

10.9 STATEMENTS

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

10.9.1 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 selection 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.

10.9.2 STRINGREP

10.9.2.1 Pointer Conversions

The default radix for the conversion of a pointer into a string is defined as implementation dependent. For the C170 the resultant string will be the pointer represented in octal notation.

10.9.3 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.

11.0 CYBIL-CI/II TYPE AND VARIABLE MAPPING

11.0 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 conformance with the SIS (S2196).

11.1 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 data objects contain only the PROCESS VIRTUAL ADDRESS, with the exception of pointer to sequence. 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).

11.1.1 ADAPTABLE POINTERS

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

11.0 CYBIL-CI/II TYPE AND VARIABLE MAPPING
11.1.1 ADAPTABLE POINTERS

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.

11.1.2 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.

11.1.3 PROCEDURE POINTERS

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

```
PROC_POINTER = RECORD
```

11.0 CYBIL-CI/II TYPE AND VARIABLE MAPPING
11.1.3 PROCEDURE POINTERS

```
        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.

11.1.4 BOUND VARIANT RECORD POINTERS

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

11.1.5 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.

11.2 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 NIL relative pointer is the following constant:

```
NIL : RELATIVE_ADDRESS := 80000000(16).
```

11.2.1 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.

11.2.2 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.
```

11.2.3 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.

11.0 CYBIL-CI/II TYPE AND VARIABLE MAPPING
11.3 INTEGERS

11.3 INTEGERS

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

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

11.4 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.

11.5 ORDINALS

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

11.6 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.

The maximum integer subrange is -8000000000000000(16) .. 7FFFFFFFFFFFFFFF(16).

11.7 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.

11.8 REALS

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.

11.9 LONGREALS

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.

11.10 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 32,767 elements.

i.e. if the number of bits in the field of bytes for the set is greater than the max set size [illegible] more bits than potential set elements.

11.0 CYBIL-CI/II TYPE AND VARIABLE MAPPING
11.11 STRINGS

11.11 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.

11.12 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. The maximum size an array can ever be is the size of a segment (i.e. 7FFFFFFF(16)).

11.13 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 coarsest 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.

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). If the first field of a record is aligned, the record will take on the alignment base from the aligned attribute quoted on that first field. If the other fields of a record are also aligned, the record takes on the coarsest alignment base from all the fields within the record.

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.

The content of the dead bits is undefined.

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

The fields are allocated consecutively subject to their alignment restrictions.

11.0 CYBIL-CI/II TYPE AND VARIABLE MAPPING
11.13 RECORDS

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.

11.14 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.

11.14.1 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.

11.14.2 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.

11.15 CELLS

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

11.0 CYBIL-CI/II TYPE AND VARIABLE MAPPING
11.16 DETAILED SUMMARY FOR THE C180

11.16 DETAILED SUMMARY FOR THE C180

Data Type	Usage	Number of bits used	Alignment	# of significant bits	Alignment of signif. bits
Pointer to fixed size object	Variable	64	Word	48	Right
	In packed rec	48	Byte	48	Right
	In unpacked rec	48	Byte	48	Right
	In packed array	48	Byte	48	Right
	In unpack array	48	Byte	48	Right
Pointer to adaptable string	Variable	64	Word	len=48	Left
			Byte	len=16	Left
	In packed rec	64	Word	len=48	Left
			Byte	len=16	Left
	In unpacked rec	64	Word	len=48	Left
			Byte	len=16	Left
	In packed array	64	Word	len=48	Left
			Byte	len=16	Left
	In unpack array	64	Word	len=48	Left
			Byte	len=16	Left
Pointer to adaptable array	Variable	144	Word	len=48	Left
			Byte	Desc=96	Left
	In packed rec	144	Word	len=48	Left
			Byte	Desc=96	Left
	In unpacked rec	144	Word	len=48	Left
			Byte	Desc=96	Left
	In packed array	144	Word	len=48	Left
			Byte	Desc=96	Left
	In unpack array	144	Word	len=48	Left
			Byte	Desc=96	Left

11.0 CYBIL-CI/II TYPE AND VARIABLE MAPPING
11.16 DETAILED SUMMARY FOR THE C180

Data Type	Usage	Number of bits used	Alignment	# of significant bits	Alignment of signif. bits
Pointer to user heap	Variable	80	Word	ptr=48	Left
			Byte	len=32	Left
	In packed rec	80	Byte	ptr=48	Left
			Byte	len=32	Left
	In unpacked rec	80	Byte	ptr=48	Left
			Byte	len=32	Left
	In packed array	80	Byte	ptr=48	Left
			Byte	len=32	Left
	In unpack array	80	Byte	ptr=48	Left
			Byte	len=32	Left
Pointer to sequence fixed or adaptable)	Variable	112	Word	ptr=48	Left
			Byte	Desc=64	Left
	In packed rec	112	Byte	ptr=48	Left
			Byte	Desc=64	Left
	In unpacked rec	112	Byte	ptr=48	Left
			Byte	Desc=64	Left
	In packed array	112	Byte	ptr=48	Left
			Byte	Desc=64	Left
	In unpack array	112	Byte	ptr=48	Left
			Byte	Desc=64	Left
Pointer to adaptable record	Variable	48+n	Word	ptr=48	Right
	In packed rec	48+n	Word	ptr=48	Right
	In unpacked rec	48+n	Word	ptr=48	Right
	In packed array	48+n	Word	ptr=48	Right
	In unpack array	48+n	Word	ptr=48	Right
	see type of adaptable for descriptor				
Bound Variant record pointer	Variable	80	Word	ptr=48	Left
			Byte	len=32	Left
	In packed rec	80	Byte	ptr=48	Left
			Byte	len=32	Left
	In unpacked rec	80	Byte	ptr=48	Left
			Byte	len=32	Left
	In packed array	80	Byte	ptr=48	Left
			Byte	len=32	Left
	In unpack array	80	Byte	ptr=48	Left
			Byte	len=32	Left

11.0 CYBIL-CI/II TYPE AND VARIABLE MAPPING
11.16 DETAILED SUMMARY FOR THE C180

Data Type	Usage	Number of bits used	Alignment	# of significant bits	Alignment of signif. bits
Relative Pointer	Variable	32	Word	32	Right
	In packed rec	32	Byte	32	Left
	In unpacked rec	32	Byte	32	Left
	In packed array	32	Byte	32	Left
	In unpack array	32	Byte	32	Left
Adaptable relative pointer	Variable	32+n	Word	ptr=32	Right
		see type of adaptable for descriptor			
	In packed rec	32+n	Byte	ptr=32	Left
		see type of adaptable for descriptor			
	In unpacked rec	32+n	Byte	ptr=32	Left
		see type of adaptable for descriptor			
	In packed array	32+n	Byte	ptr=32	Left
		see type of adaptable for descriptor			
	In unpack array	32+n	Byte	ptr=32	Left
		see type of adaptable for descriptor			
Relative Pointer to Sequence (fixed or adaptable)	Variable	96	Word	ptr=32	Left
			Byte	Desc=64	Left
	In packed rec	96	Byte	ptr=32	Left
			Byte	Desc=64	Left
	In unpacked rec	96	Byte	ptr=32	Left
			Byte	Desc=64	Left
	In packed array	96	Byte	ptr=32	Left
			Byte	Desc=64	Left
	In unpack array	96	Byte	ptr=32	Left
			Byte	Desc=64	Left
Relative Pointer to Bound Variant Record	Variable	64	Word	ptr=32	Left
			Byte	len=32	Left
	In packed rec	64	Byte	ptr=32	Left
			Byte	len=32	Left
	In unpacked rec	64	Byte	ptr=32	Left
			Byte	len=32	Left
	In packed array	64	Byte	ptr=32	Left
			Byte	len=32	Left
	In unpack array	64	Byte	ptr=32	Left
			Byte	len=32	Left

11.0 CYBIL-CI/II TYPE AND VARIABLE MAPPING
11.16 DETAILED SUMMARY FOR THE C180

Data Type	Usage	Number of bits used	Alignment	# of significant bits	Alignment of signif. bits
Pointer to procedure	Variable	96	Word	addr=48	Left
			Byte	addr=48	Left
	In packed rec	96	Byte	addr=48	Left
			Byte	addr=48	Left
	In unpacked rec	96	Byte	addr=48	Left
			Byte	addr=48	Left
	In packed array	96	Byte	addr=48	Left
			Byte	addr=48	Left
	In unpack array	96	Byte	addr=48	Left
			Byte	addr=48	Left
Integer	Variable	64	Word	64	Right
	In packed rec	64	Byte	64	Left
	In unpacked rec	64	Byte	64	Left
	In packed array	64	Byte	64	Left
	In unpack array	64	Byte	64	Left
Characters	Variable	8	Word	8	Right
	In packed rec	8	Bit	8	Left
	In unpacked rec	8	Byte	8	Left
	In packed array	8	Bit	8	Left
	In unpack array	8	Byte	8	Left
Subrange and ordinals	Variable		Word		Right
	In packed rec		Bit		Right
	In unpacked rec	See	Byte	See	Right
	In packed array	Above	Bit	Above	Right
	In unpack array		Byte		Right
Booleans	Variable	64	Word	1	Right
	In packed rec	1	Bit	1	Left
	In unpacked rec	8	Byte	1	Right
	In packed array	1	Bit	1	Left
	In unpack array	8	Byte	1	Right

11.0 CYBIL-CI/II TYPE AND VARIABLE MAPPING
11.16 DETAILED SUMMARY FOR THE C180

Data Type	Usage	Number of bits used	Alignment	# of significant bits	Alignment of signif. bits
Real	Variable	64	Word	64	Right
	In packed rec	64	Byte	64	Right
	In unpacked rec	64	Byte	64	Right
	In packed array	64	Byte	64	Right
	In unpack array	64	Byte	64	Right
Longreal	Variable	128	Word	128	Right
	In packed rec	128	Word	128	Right
	In unpacked rec	128	Word	128	Right
	In packed array	128	Word	128	Right
	In unpack array	128	Word	128	Right
Sets	Variable		Word		Right
	In packed rec		Bit		Left
	In unpacked rec	See	Byte	See	Left
	In packed array	Above	Bit	Above	Left
	In unpack array		Byte		Left
Strings	Variable	n bytes	Word	n bytes	Left
	In packed rec	n bytes	Byte	n bytes	Left
	In unpacked rec	n bytes	Byte	n bytes	Left
	In packed array	n bytes	Byte	n bytes	Left
	In unpack array	n bytes	Byte	n bytes	Left
Cell	Variable	8	Byte	8	Left
	In packed rec	8	Byte	8	Left
	In unpacked rec	8	Byte	8	Left
	In packed array	8	Byte	8	Left
	In unpack array	8	Byte	8	Left

11.0 CYBIL-CI/II TYPE AND VARIABLE MAPPING
11.17 SUMMARY FOR THE C180

11.17 SUMMARY FOR THE C180

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

12.0 CYBIL-CI/II RUN TIME ENVIRONMENT

12.0 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 CYBER 180 SYSTEM INTERFACE STANDARD (S2196).

12.1 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
A5	- STATIC LINK	- SL
A10 .. A14	- PARAMETERS PASSED TO INTERNAL	
X9 .. X13	- PROCEDURES OR FUNCTIONS	
X14	- LINE NUMBER FOR RANGE CHECKING	- LN
A15	- FUNCTION RESULT IF SIMPLE POINTER	
X15	- FUNCTION RESULT IF SCALAR	
X14 & X15	- FUNCTION RESULT IF DOUBLE PRECISION	

The registers A0, A1 and A2 always contain the assigned values. Registers A5, A15, X14 and X15 may be assigned other values during the execution of the procedure.

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

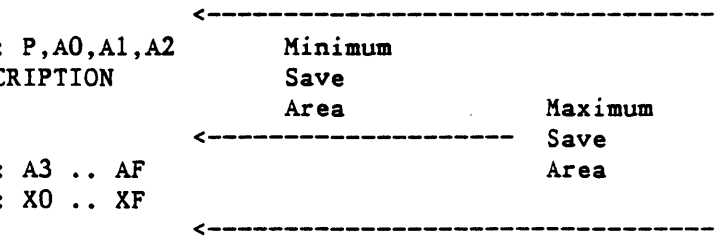
Current Stack Frame pointer indicates the start of the current stack frame.

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

12.0 CYBIL-CI/II RUN TIME ENVIRONMENT
12.1 REGISTER ASSIGNMENT

the calling procedure (Note that a CYBIL-CI/II program executes in one ring only). The save area contains the following information:



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, bits 40 .. 43.

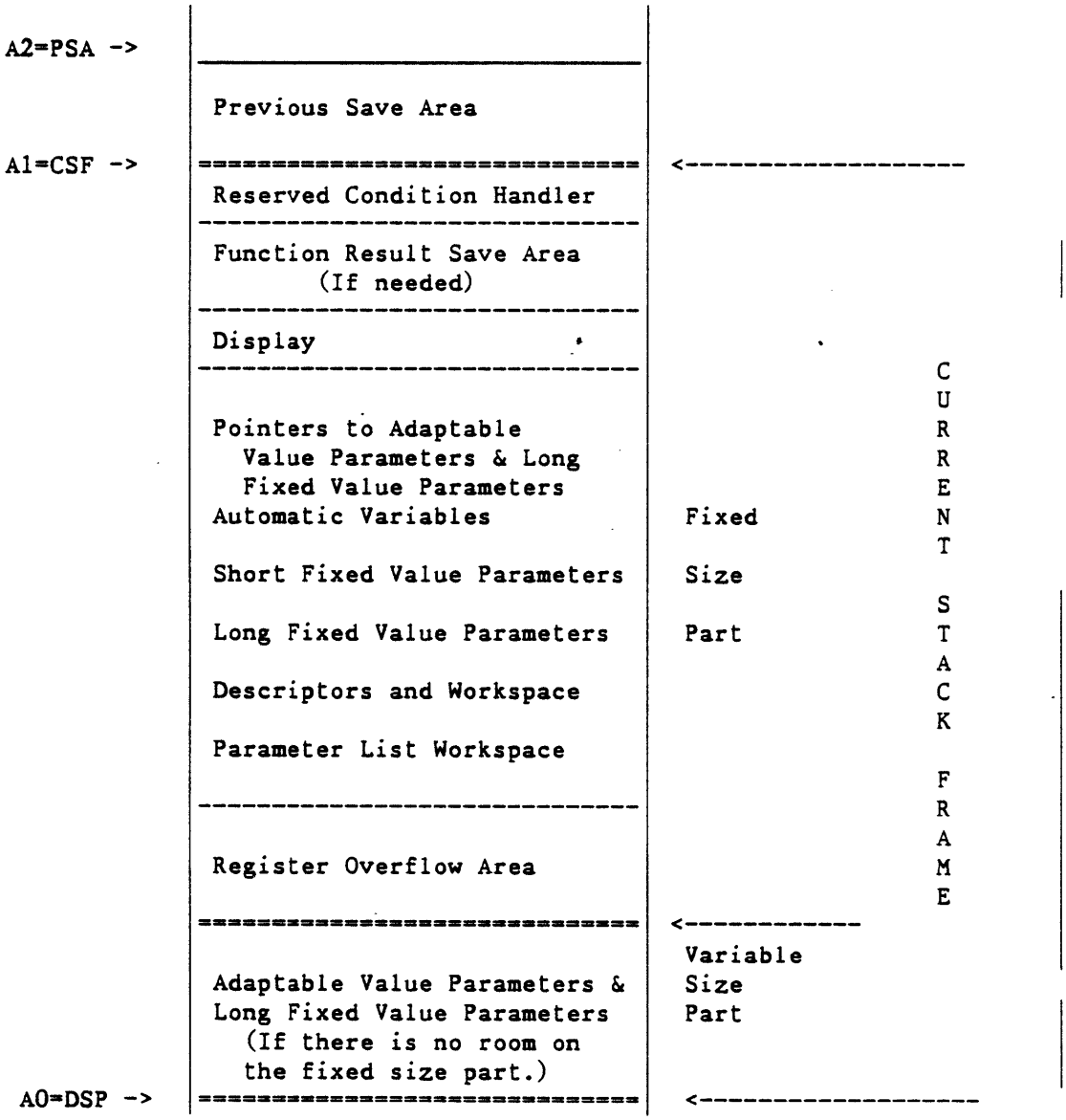
Static Link Pointer indicates the stack frame of the enclosing procedure if the called procedure is an internal procedure of the calling procedure and is meaningless otherwise.

For internal (non-XREFed or XDCLed) procedures and functions references, the parameters that fit into an X register or an A register are passed in registers X9 through X13 and A10 through A14. The A registers are used for passing pointers and the X registers are used for the other basic types which fit into a register. The register selection starts with A10 and X9 respectively and with a left to right parameter selection from the actual parameter list. If all 5 of the particular type of register are all ready in use, passing other parameters, then the parameter, is included with the normal parameter list entries.

12.0 CYBIL-CI/II RUN TIME ENVIRONMENT
12.2 STACK FRAME DEFINITION

12.2 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.

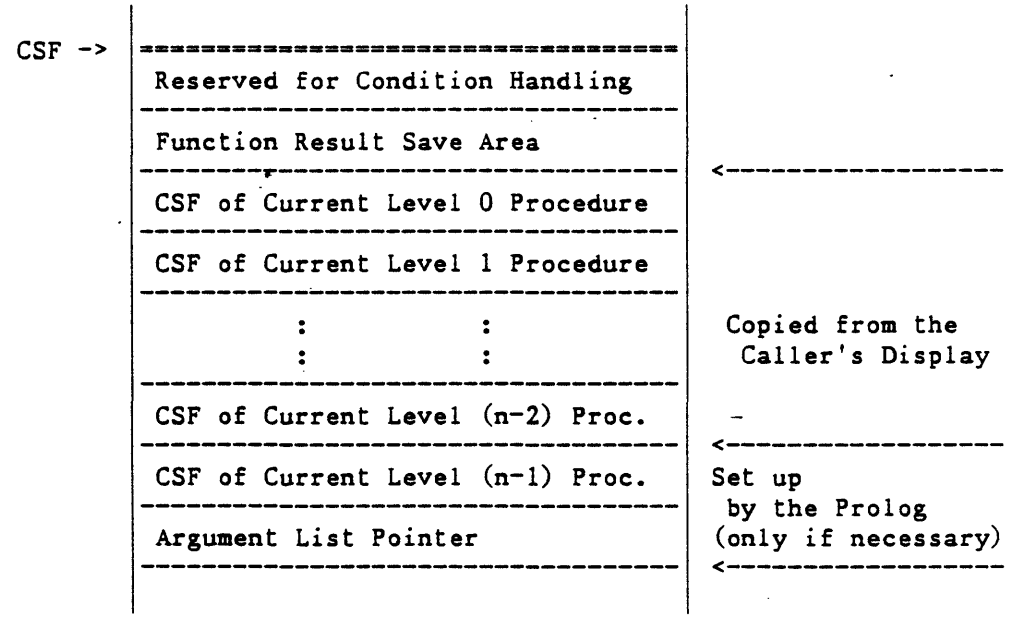


12.0 CYBIL-CI/II RUN TIME ENVIRONMENT
12.2.1 FIXED SIZE PART

12.2.1 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:



The prolog will save the static link (if it was passed in register A5) 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

12.0 CYBIL-CI/II RUN TIME ENVIRONMENT
12.2.1 FIXED SIZE PART

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.

12.2.2 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.

12.3 PARAMETER PASSAGE

12.3.1 REFERENCE PARAMETERS

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

12.3.2 VALUE PARAMETERS

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.

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 either the actual parameter or a copy of the actual parameter.

The nominal circumstance is that a copy is made of the data. However, if one of the following conditions exists, then the value parameter is not copied to the callers stack and a pointer to the data is put into the parameter list.

The following are the rules for Copying Large Value Parameters in the Caller:

1) Large constants that are passed as value parameters are not copied.

2) Parameters that will be copied in the callee are not copied.

3) Copies are not made of value parameters that are passed as value parameters to another procedure.

4) Copies are not made of variables that are defined to be in memory sections that have the READ attribute.

5) Large automatic variables that are passed as value parameters are not copied UNLESS:

   a) The automatic variable is passed by reference in any procedure call within the scope of the defining procedure, OR

   b) A pointer is generated to the automatic variable in an assignment statement using the ↑ or #LOC operators, or the ↑ or #LOC operators are used to pass the address of the automatic

variable in the procedure call statement, OR

c) The automatic variable is being passed to a nested procedure within the scope of the procedure which defined the automatic variable, and the automatic variable is modified in a nested procedure within the scope of the procedure that defined the automatic variable, OR

d) The automatic variable is a pointer and the pointer is dereferenced on the procedure call.

The rules for Prolog Copies of Large Value Parameters in the Callee are:

Copy value parameters in the prolog of the callee if the called procedure or a nested procedure generates a pointer to the value parameter via the unary pointer operator (↑) or the #LOC operator AND:

1) The pointer is passed as a parameter, OR

2) The pointer is not an automatic variable of the procedure, OR

3) The pointer is assigned to another pointer, OR

4) A pointer to the pointer is generated in an assignment statement using the ↑ or #LOC operators, OR

5) A pointer to the pointer is passed in a procedure call using the ↑ or #LOC operators, OR

6) The pointer is dereferenced on the left side of an assignment statement, OR

7) The pointer is dereferenced on a procedure call and the call is by reference, OR

8) The value parameter is a sequence or a structure that contains a sequence AND

   a) The pointer is generated to the sequence AND
   b) A data item pointer is generated into the sequence with a NEXT statement AND
   c) The data item pointer escapes (i.e. one of 1 - 7 above occurs).

If the called procedure or a contained procedure generates a

pointer to the value parameter and the value of that pointer escapes, or what the pointer points at is altered, then 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.

12.3.3 INTERLANGUAGE CALLING

For any potentially interlanguage call in which a System format actual parameter list is passed that contains only simple reference parameters: The parameter list is immediately preceded by a flag word whose value is the 64-bit integer zero. The flag word need not precede any other System format actual parameter lists.

12.4 BINDING SECTION DESCRIPTION

Binding Section Segments are intended to facilitate 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.

12.5 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.

12.5.1 VARIABLES

12.5.1.1 Variable Attributes

12.5.1.1.1 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.

12.5.1.1.2 #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 is referred to the NOS/VE documentation for further information.

To summarize, the #GATE attribute allows a procedure to be accessed by another procedure at a higher ring level.

12.5.1.2 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.

The components of unpacked arrays and records are mapped to computer memory in their "natural order" such that if an array or record was placed in a sequence and the sequence reset to the array or record, the following would be true:

- A NEXT of a pointer to the appropriate component type yields a pointer to the first element of the array or field of the record.
- Subsequent NEXTs of pointers of appropriate component types yield pointers to the second, third, and so on, elements of the array or fields of the record.

If an array or record placed in a sequence is retrieved from the sequence component by component, the types of the data accessed must match the types of the corresponding elements of the array or fields of the record.

12.5.1.3 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>.

The implementation dependent value for the alignment base for the C180 is eight (8).

12.5.2 STATEMENTS

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

12.5.2.1 CASE Statement

There are 2 alternative code sequences generated for the CASE statement, depending on the characteristics of the case selectors.

The normal code sequence generated for a CASE statement is via a series of conditional branches. The branch implementation is chosen if the number of case selectors is less than twenty, or, if the difference between largest and smallest case selectors is greater than 1024 (a NOS/VE page) or, if the difference between largest and smallest case selector is greater than the number of selectors specified to the power of three. If the number of seletors specified is more than 7, the code sequence that is generated is in the form of a binary search.

The other alternative is a code sequence using a jump table for the CASE statement. This jump table actually resides as a contiguous series of 2 byte entries 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.

12.5.2.2 STRINGREP

12.5.2.2.1 POINTER CONVERSIONS

The default radix for the conversion of a pointer into a string is defined as implementation dependent. For the C180 the resultant string will be the pointer represented in hexadecimal notation. Note that a C180 pointer requires a field length of 15 characters.

12.5.2.3 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.

12.6 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.

12.7 PROCEDURE REFERENCES

Registers A1, A3, A4 and A5 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
A5 <---> Static Link

12.8 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.

12.9 RUN TIME LIBRARY

The procedures described below are available on the C180 via:

12.0 CYBIL-CI/II RUN TIME ENVIRONMENT
12.9 RUN TIME LIBRARY

ATTF $LOCAL.CYF$RUN_TIME_LIBRARY.

The procedures described below are available on the C170 via:

ATTACH,CYBILIB/UN=LP3.

12.9.1 HEAP MANAGEMENT

To be supplied.

12.9.2 I/O

12.9.2.1 Common CYBIL I/O

The I/O on NOS/VE can be provided by the Common CYBIL I/O interfaces defined in the Reference Manual ARH6794

12.9.2.2 I/O on the C180 Simulator

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).

13.0 CYBIL-CM/IM TYPE AND VARIABLE MAPPING

13.0 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:

- Bits
- BCD digits (4 bits)
- Bytes (8 bits)
- Words (16 bits)
- 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 scalar types are bit aligned and are allocated the number of bits necessary to hold the scalar. However, if the number of bits necessary to hold the scalar exceeds 15, the scalar is byte aligned and is allocated an integral number of bytes.

Packed aggregates, i.e., arrays, records, sets, and strings, are always byte aligned and are allocated the number of bytes necessary to hold the aggregate.

Many unpacked types are word aligned and are allocated the number of bytes necessary to hold the type. If only one byte is needed, the type is byte aligned.

13.0 CYBIL-CM/IM TYPE AND VARIABLE MAPPING
13.1 POINTERS

13.1 POINTERS

A pointer consists of an address field of 4 bytes and, for certain pointer types, a descriptor. The address field contains a 32-bit address of the first byte of the object (data or procedure).

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.

13.1.1 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 2-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.

13.0 CYBIL-CM/IM TYPE AND VARIABLE MAPPING
13.1.1 ADAPTABLE POINTERS

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 2-byte size field indicating the length of the string in bytes.

13.1.2 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.

13.1.3 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.

13.1.4 POINTER ALIGNMENT

All unpacked pointer types are word aligned. All packed pointer types are byte aligned.

13.2 RELATIVE POINTERS

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

With the exception of relative pointers to sequences, relative pointers to fixed size data objects consist of the relative address field only. A relative pointer to a sequence consists of the 4-byte relative address field followed by 2 4-byte fields indicating the size of the sequence in bytes, and the next available position in the sequence.

An unpacked relative pointer is word aligned.

A packed relative pointer is byte aligned.

The NIL relative pointer is the following constant:

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

13.2.1 ADAPTABLE RELATIVE POINTERS

Adaptable relative pointers are identical to adaptable pointers with the exception that first four bytes holds a relative address rather than an address.

13.2.2 RELATIVE POINTERS TO BOUND VARIANT RECORDS

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

13.3 INTEGERS

Integer types are allocated 32 bits.

An unpacked integer type is word aligned.

A packed integer type is byte aligned.

An integer variable is mapped as an unpacked integer type.

13.4 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.

13.5 ORDINALS

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

13.6 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 and occupies an integral number of bits if the bit length is 15 or less. A packed subrange type is byte aligned and occupies an integral number of bytes if the bit length is greater than 15. The 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).

13.7 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.

13.8 REALS

Real types are allocated 32 bits.

An unpacked real type is word aligned.

A packed real type is byte aligned.

A real variable is mapped as an unpacked real type.

13.9 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.

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 maximum number of elements is 32768. The leftmost bit represents the first element, the next bit represents the second element, etc. The set is left justified in its allocated field.

13.0 CYBIL-CM/IM TYPE AND VARIABLE MAPPING
13.10 SETS

An unpacked set type is word aligned and is allocated the number of bytes necessary to hold the set.

A packed set type is byte aligned and is allocated the number of bytes necessary to hold the set.

A set variable is mapped as an unpacked set type.

13.11 STRINGS

An unpacked string type is word aligned and is allocated the number of bytes necessary to hold the string.

A packed string type is byte aligned and is allocated the number of bytes necessary to hold the string.

A string variable is mapped as an unpacked string type.

13.12 ARRAYS

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

A packed array type is a contiguous list of packed instances of its component type. A packed array is aligned on a byte boundary and occupies an integral number of bytes.

An array variable is aligned on a word boundary unless the array fits in a byte.

In general, array sizes are limited by storage availability.

13.13 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 bytes.

A packed record type is a contiguous list of packed fields. It is aligned on a byte boundary and occupies an integral number of bytes.

A record variable is aligned on a word boundary unless the record fits in a byte.

13.14 SEQUENCES

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

With the exception of strings and sets, data within a sequence is mapped in the same manner as variables. For example, a NEXT statement which specifies a pointer to integer causes allocation of 4 bytes within the sequence starting at the next even byte boundary. A NEXT statement which specifies a pointer to a boolean causes allocation of 1 byte within the sequence starting at the next byte within the sequence.

A NEXT statement which specifies a pointer to a string or set causes allocation of the number of bytes needed to hold the string or set starting at the next byte within the sequence, regardless of whether the address is even or odd.

13.15 HEAPS

13.0 CYBIL-CM/IM TYPE AND VARIABLE MAPPING
13.15.1 SYSTEM HEAP

13.15.1 SYSTEM HEAP

The System Heap consists of blocks of memory dynamically allocated and freed via operating system requests.

13.15.2 USER HEAPS

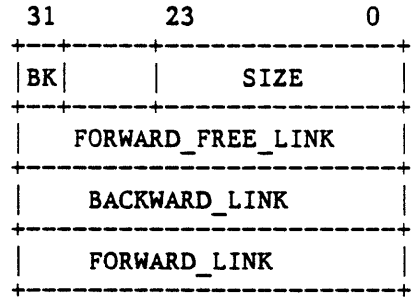
A user heap consists of a block of memory already allocated, either statically or dynamically. Run time routines provide for allocating and freeing blocks within the block already allocated.

A user heap consists of a Heap Header and storage for Data Blocks and Free Blocks.

A Data Block consists of a Data Block Header followed by storage for user data.

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

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



The field, BK, indicates the block kind: free_block, data_block, or heap_block.

13.0 CYBIL-CM/IM TYPE AND VARIABLE MAPPING
13.15.2 USER HEAPS

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

```
BLOCK_KIND_TYPE = (FREE_BLOCK, DATA_BLOCK, HEAP_BLOCK),
OFFSET_TYPE = -80000000(16) .. 7fffffff(16),

BLOCK_HEADER = PACKED RECORD
  BLOCK_KIND: BLOCK_KIND_TYPE,
  SIZE: 0..0FFFFFF(16),
  FORWARD_FREE_LINK,
  BACKWARD_LINK,
  FORWARD_LINK: OFFSET_TYPE,
RECEND;
```

For the Heap Header, the fields are as follows:

```
  BLOCK_KIND: HEAP_BLOCK
  SIZE: 0
  FORWARD_FREE_LINK: Link to Free Block.
  BACKWARD_LINK: 0
  FORWARD_LINK: 0
```

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

```
  BLOCK_KIND: DATA_BLOCK
  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_KIND: FREE_BLOCK
  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 a free block with all three links uninitialized. Thus the header contains only the block kind and the block size. When the first allocate request is made, this free block is transformed into a heap block consisting of a heap header and a free block. Then the allocate request is satisfied from the free block.

13.0 CYBIL-CM/IM TYPE AND VARIABLE MAPPING
13.15.2 USER HEAPS

Subsequent allocate and free requests cause the heap block to be subdivided into several data blocks and free blocks.

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

A RESET request causes the heap block to be changed back to a free block. Again, only the block kind and block size are reset in the header. The links are left unchanged.

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

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

13.16 CELLS

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

13.0 CYBIL-CM/IM TYPE AND VARIABLE MAPPING
13.17 SUMMARY FOR THE MC68000

13.17 SUMMARY FOR THE MC68000

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

14.0 CYBIL-CM/IM RUN TIME ENVIRONMENT

14.0 CYBIL-CM/IM RUN TIME ENVIRONMENT

14.1 MEMORY

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

- Code
- Static Storage
- Stack
- Heap

14.1.1 CODE

The code section contains the instructions of the program.

14.1.2 STATIC STORAGE

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

Static storage may contain the following kinds of sections:

- Read Only Sections
- Read Write Sections

14.1.3 STACK

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

The stack consists of 2 kinds of entries, stack frames and register save areas. The stack consists of alternating stack frames and

register save areas. If there are no registers to save, the size of a register save area is zero.

14.1.3.1 Stack Frame

The stack frame consists of the following parts:

- The Fixed Size Part contains all data whose size is known at compile time.
- The Variable Size Part contains storage allocated using PUSH statements.
- The Argument List Part contains the parameters in the call to the procedure.
- The P-register Part contains the return address.

14.1.3.1.1 FIXED SIZE PART

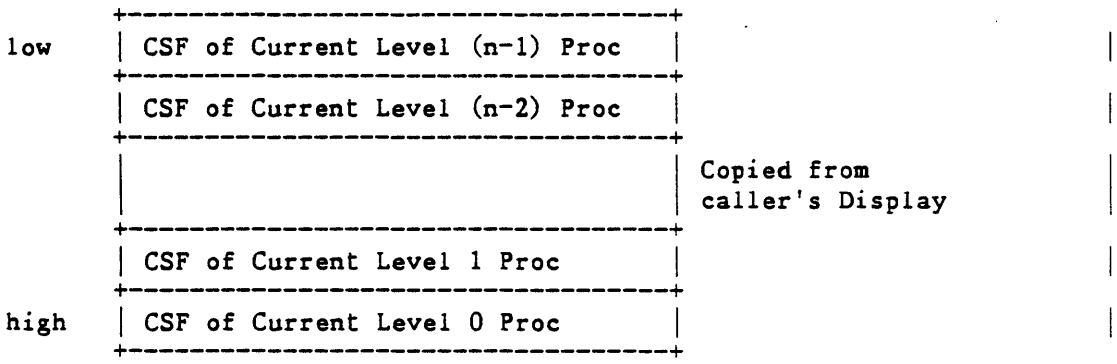
The Fixed Size Part contains some of the data which the procedure may access directly, and addressing information for other data which the procedure may access. The Fixed Size Part contains the following:

- Dynamic Link
- Display
- Automatic Variables
- Register Overflow Area
- Workspace

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:

14.0 CYBIL-CM/IM RUN TIME ENVIRONMENT
14.1.3.1.1 FIXED SIZE PART



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 32-bit address of a stack frame.

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

The Workspace area is used to hold data such as an intermediate result for an expression with sets as operands. This data is PUSH'ed onto the stack.

14.1.3.1.2 VARIABLE SIZE PART

The Variable Size Part contains storage allocated using PUSH statements.

14.1.3.1.3 ARGUMENT LIST PART

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

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

14.1.3.1.4 P-REGISTER PART

The P-register Part contains the return address.

14.1.4 SYSTEM HEAP

The ALLOCATE statement has the following forms:

- ALLOCATE <allocation designator> IN <heap variable>;
- 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:

- FREE <pointer variable> IN <heap variable>;
- 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.

14.1.5 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.

14.0 CYBIL-CM/IM RUN TIME ENVIRONMENT
14.1.5 REGISTERS

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.

14.2 PARAMETER PASSAGE

14.2.1 REFERENCE PARAMETERS

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

14.2.2 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.

A pointer to a packed record or a packed array which begins at an odd address is never placed in the parameter list. Instead, the packed record or packed array is copied onto the stack starting at an even address, and the address of the copy is placed in the parameter list. Hence, the callee is guaranteed that the packed record or packed array begins at an even address.

Value parameters appear in the parameter list as follows:

14.0 CYBIL-CM/IM RUN TIME ENVIRONMENT
14.2.2 VALUE PARAMETERS

Type	Copy or Pointer	Parameter List Entry Size
Pointer	copy	2 words for fixed pointer (except sequence). 6 words for fixed ptr to sequence. 3 words for adaptable string pointer. 6 words for adaptable array pointer. 6 words for adaptable sequence pointer. 4 words for adaptable heap pointer. 3-6 words for adaptable record pointer.
Integer	copy	2 words
Character	copy	1 byte. The character is in the upper 8 bits of the word with lower bits unused.
Ordinal	copy	1 word
Integer Subrange	copy	1 or 2 words
Boolean	copy	1 byte. The boolean value is in the upper 8 bits of the word with lower bits unused.
Real	copy	2 words
Longreal	copy	4 words
Set	pointer	2 words
String	pointer	2 words for fixed string. 3 words for adaptable string.
Array	pointer	2 words for fixed array. 6 words for adaptable array.
Record	pointer	2 words for fixed record. 3-6 words for adaptable record.
Cell	copy	1 word. The cell is in the upper 8 bits of the word with lower bits unused.
Sequence	pointer	6 words for fixed sequence. 6 words for adaptable sequence.
Heap	pointer	4 words

14.3 VARIABLES

14.3.1 VARIABLE ATTRIBUTES

14.3.1.1 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.

14.3.1.2 #Gate Attributes

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

14.3.2 VARIABLE ALLOCATION

With the exception of byte size variables, 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. For byte size variables, however, space may be allocated from gaps in space already allocated.

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

14.3.3 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.

The implementation dependent value for the alignment base for the MC68000 is two (2).

14.4 STATEMENTS

14.4.1 CASE STATEMENT

There are 2 alternative code sequences generated for the CASE statement, depending on the characteristics of the case selectors.

The jump table 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 (JMP) branch instruction to the appropriate case selector.

The other alternative is a code sequence generated via a series of conditional branches. Branch implementation is chosen if the number of case selectors is less than three, or, if the amount of space required for a jump table exceeds the amount of space required for branch instructions. If the number of entries is more than 8, the code sequence that is generated is in the form of a binary search.

14.4.2 STRINGREP

14.4.2.1 Pointer Conversions

The default radix for the conversion of a pointer into a string is defined as implementation dependent. For the MC68000 the resultant string will be the pointer represented in hexadecimal notation.

14.5 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 and functions 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. Agreement between data types must be exact. Even names used to define data types must be identical. For example, the following

seemingly identical definitions produce different hashes:

```
MODULE one;
  TYPE
    p_type = 0 .. 65535;
  PROCEDURE [XREF] two (p: p_type);
MODEND one;

MODULE two;
  PROCEDURE [XDCL] two (p: 0 .. 65535);
  PROCEND two;
MODEND two;
```

14.6 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

14.7 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.

14.8 PROCEDURE CALL AND RETURN INSTRUCTION SEQUENCES

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

14.8.1 PROCEDURE CALL

The following illustrate instruction sequences for procedure calls:

External Procedure without Parameters

```
MOVEM.L  reglist,-(DSP)        Save regs as needed
JSR      external_procedure    Call Procedure
MOVEM.L  (DSP)+,reglist        Restore regs as needed
```

Internal Procedure and Static Link (SL)

```
MOVEM.L  reglist,-(DSP)        Save regs as needed
MOVEA.L  CSF,SL                Static Link
BSR      internal_proc         Call Procedure
MOVEM.L  (DSP)+,reglist        Restore regs as needed
```

14.0 CYBIL-CM/IM RUN TIME ENVIRONMENT
14.8.1 PROCEDURE CALL

Pointer to Procedure

```
MOVEM.L  reglist,-(DSP)         Save regs as needed
MOVEM.L  proc_ptr(base),A0/SL   Proc Addr & Static Link
JSR      (A0)                   Call Procedure
MOVEM.L  (DSP)+,reglist         Restore regs as needed
```

Setup Argument List on Procedure Call

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

MOVEM.L  reglist,-(DSP)         Save regs as needed
MOVE.B   A(CSF),-(DSP)          1-byte parameter
MOVE.W   B(CSF),-(DSP)          1-word parameter
MOVE.L   C(CSF),-(DSP)          2-word parameter
MOVE.W   D+4(CSF),-(DSP)        3-word parameter
MOVE.L   D(CSF),-(DSP)
BSR      internal_proc          Call Procedure
LEA      plist_size(A7),(A7)    Pop parameters
MOVEM.L  (DSP)+,reglist         Restore regs as needed
```

14.9 PROLOG

The basic instruction sequence for the prolog is as follows:

```
prolog:
  LINK     CSF,#-frame_size     Form Dynamic Link & Update DSP
```

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:

14.0 CYBIL-CM/IM RUN TIME ENVIRONMENT
14.9 PROLOG

```
prolog:
  LINK      CSF,#-frame_size      Dynamic Link & Update DSP

  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.

14.10 EPILOG

The basic instruction sequence for the epilog is as follows:

```
epilog:
  UNLK      CSF                   DSP := CSF
                                  DSP := DSP + 4
                                  CSF := (DSP)
  RTS                             Return
```

14.11 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:

- CYM_ABORT - Contains procedure CYP_ABORT for aborting the program.
- CYM_ALLOCATE - Contains procedure CYP_ALLOCATE for allocating a

block in a user heap.

- CYM_FREE - Contains procedure CYP_FREE for freeing a block in a user heap.
- CYM_NIL_ERROR - 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.
- CYM_STRINGREP - Contains procedure CYP_STRINGREP for the STRINGREP built-in procedure.
- CYM_MPY_4_BYTES_BY_4_BYTES - Contains procedure CYP_MPY_4_BY_4 for integer multiplication.
- CYM_DIV_4_BYTES_BY_4_BYTES - Contains procedure CYP_DIV_4_BY_4 for integer division.
- CYM_MOD_4_BYTES_BY_4_BYTES - Contains procedure CYP_MOD_4_BY_4 for integer remainder.

In addition to the above, the compiler generates calls to the following procedures for system heap management:

- CYP_SYS_ALLOC - Procedure to allocate a block in the system heap.
- CYP_SYS_FREE - Procedure to free a block in the system heap.

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.

14.12 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.

15.0 CYBIL-CU/IU TYPE AND VARIABLE MAPPING

15.0 CYBIL-CU/IU TYPE AND VARIABLE MAPPING

The CYBIL-CU/IU type and variable mappings are the same as those for CYBIL-CM/IM.

16.0 CYBIL-CU/IU RUN TIME ENVIRONMENT

16.0 CYBIL-CU/IU RUN TIME ENVIRONMENT

The CYBIL_CU/IU run time environment is the same as that for CYBIL-CM/IM with the following exceptions:

- A function returns its value in register D0 instead of D7.

17.0 CYBIL-CA/AA TYPE AND VARIABLE MAPPING

The MC68010/Apollo data formats for each of the supported CYBIL data types is described in the following sections. The data formats were, where appropriate, selected to conform to the existing definition for Apollo Pascal.

The MC68010/Apollo supports five basic data types as follows:

- Bits
- BCD digits (4 bits)
- Bytes (8 bits)
- Words (16 bits)
- 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 Apollo MC68010, integers are represented in two's complement form.

Certain packed scalar types are bit aligned and are allocated the number of bits necessary to hold the scalar. However, if the scalar would cross two word boundaries if allocated from the current bit position, it will first be word aligned but still bit sized.

Packed sets follow the alignment rules of packed scalars. That is, if the set does not cross more than one word boundary when allocated from the current bit position, this position will be used. Otherwise, it will be word aligned.

Packed aggregates, i.e., arrays, records, and strings, are always word aligned and are allocated the number of bytes necessary to hold the aggregate.

17.0 CYBIL-CA/AA TYPE AND VARIABLE MAPPING

Many unpacked types are word aligned and are allocated the number of words necessary to hold the type. However, types BOOLEAN and CHAR are byte aligned and one byte long.

17.1 POINTERS

A pointer consists of an address field of 4 bytes and, for certain pointer types, a descriptor. The address field contains a 32-bit address of the first byte of the object (data or procedure).

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.

17.1.1 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 2-byte size field indicating the length of the string in bytes.

17.0 CYBIL-CA/AA TYPE AND VARIABLE MAPPING
17.1.1 ADAPTABLE POINTERS

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 2-byte size field indicating the length of the string in bytes.

17.1.2 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.

17.1.3 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.

17.1.4 POINTER ALIGNMENT

All unpacked pointer types are word aligned. All packed pointer types are word aligned.

17.2 RELATIVE POINTERS

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

With the exception of relative pointers to sequences, relative pointers to fixed size data objects consist of the relative address field only. A relative pointer to a sequence consists of the 4-byte relative address field followed by 2 4-byte fields indicating the size of the sequence in bytes, and the next available position in the sequence.

An unpacked relative pointer is word aligned.

A packed relative pointer is word aligned.

The NIL relative pointer is the following constant:

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

17.2.1 ADAPTABLE RELATIVE POINTERS

17.0 CYBIL-CA/AA TYPE AND VARIABLE MAPPING
17.2.1 ADAPTABLE RELATIVE POINTERS

Adaptable relative pointers are identical to adaptable pointers with the exception that first four bytes holds a relative address rather than an address.

17.2.2 RELATIVE POINTERS TO BOUND VARIANT RECORDS

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

17.3 INTEGERS

Integer types are allocated 32 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.

17.4 CHARACTERS

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

A packed character type is allocated a byte and is byte aligned.

A character variable is mapped as an unpacked character type.

17.5 ORDINALS

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

17.6 SUBRANGES

An unpacked integer subrange type will be mapped into a word or a long word depending on the range.

A packed subrange type, a..b, is bit aligned and occupies an integral number of bits if the datum does not cross more than one word boundary.

A packed subrange type is word aligned and occupies an integral number of bits if the datum would cross otherwise two word boundaries.

The 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 -7FFFFFFF(16) .. 7FFFFFFF(16).

17.7 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.

17.0 CYBIL-CA/AA TYPE AND VARIABLE MAPPING
17.7 BOOLEANS

The internal value for FALSE is 0(16). The internal value for TRUE is 0FF(16).

17.8 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.

17.9 LONGREALS

Longreal types are allocated 64 bits.

An unpacked longreal type is word aligned.

A packed longreal type is word aligned.

A longreal variable is mapped as an unpacked longreal type.

17.10 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 maximum number of elements is 32,768 which must fall in the range of 0 .. 32767. The rightmost bit represents the first element, the next bit represents the second element, etc. The set is right justified in its allocated field.

17.0 CYBIL-CA/AA TYPE AND VARIABLE MAPPING
17.10 SETS

An unpacked set type is word aligned. Its allocation size depends on the low ordinal in the set type:

- If the low ordinal is zero, then the set is allocated the number of words to hold it.
- If the low ordinal is greater than zero, then the compiler allocates filler bits for zero thru the low ordinal minus one.

If the packed set type has less than thirty two elements and would not cross two word boundaries, it is bit aligned and allocated the number of bits needed to hold it.

If the packed set type has less than thirty two elements but would cross two word boundaries if allocation started at the current bit, it is word aligned and allocated the minimum number of bits needed to hold it.

If the packed set type has thirty two or more elements, it is aligned and sized as if unpacked.

17.11 STRINGS

An unpacked string type is word aligned and is allocated the number of bytes necessary to hold the string.

A packed string type is byte aligned and is allocated the number of bytes necessary to hold the string.

A string variable is mapped as an unpacked string type.

17.12 ARRAYS

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

An unpacked 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. A packed array is aligned on a word boundary and occupies an integral number of words.

An array variable is always aligned on a word boundary.

Note that APOLLO PASCAL does not support packed arrays (except an array of CHAR which is equivalent to a CYBIL fixed string) and therefore they should not be used when interfacing with the APOLLO system.

17.13 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. It is aligned on a word boundary and occupies an integral number of words.

A record variable is always aligned on a word boundary.

Record fields of types boolean, char, subrange of char, and array of char are byte aligned. The rest are word aligned (unpacked case).

17.14 SEQUENCES

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

With the exception of strings, data within a sequence is mapped in

the same manner as variables. For example, a NEXT statement which specifies a pointer to integer causes allocation of 4 bytes within the sequence starting at the next even byte boundary. A NEXT statement which specifies a pointer to a boolean causes allocation of 1 byte within the sequence starting at the next byte within the sequence.

A NEXT statement which specifies a pointer to a string causes allocation of the number of bytes needed to hold the string starting at the next byte within the sequence, regardless of whether the address is even or odd.

17.15 HEAPS

17.15.1 SYSTEM HEAP

The System Heap consists of blocks of memory dynamically allocated and freed via operating system requests.

17.15.2 USER HEAPS

A user heap consists of a block of memory already allocated, either statically or dynamically. Run time routines provide for allocating and freeing blocks within the block already allocated.

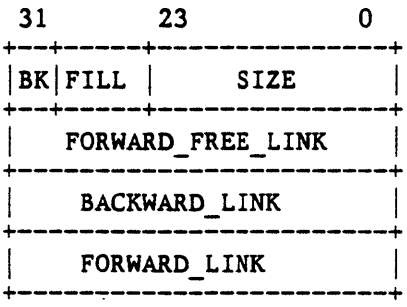
A user heap consists of a Heap Header and storage for Data Blocks and Free Blocks.

A Data Block consists of a Data 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:

17.0 CYBIL-CA/AA TYPE AND VARIABLE MAPPING
17.15.2 USER HEAPS



The field, BK, indicates the block kind: free_block, data_block, or heap_block.

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

```
BLOCK_KIND_TYPE = (FREE_BLOCK, DATA_BLOCK, HEAP_BLOCK),
OFFSET_TYPE = -80000000(16) .. 7fffffff(16),

BLOCK_HEADER = PACKED RECORD
  BLOCK_KIND: BLOCK_KIND_TYPE,
  FILL: 0..3F(16),
  SIZE: 0..0FFFFFF(16),
  FORWARD_FREE_LINK,
  BACKWARD_LINK,
  FORWARD_LINK: OFFSET_TYPE,
RECEND;
```

For the Heap Header, the fields are as follows:

BLOCK_KIND: HEAP_BLOCK
FILL: 0
SIZE: 0
FORWARD_FREE_LINK: Link to Free Block.
BACKWARD_LINK: 0
FORWARD_LINK: 0

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

17.0 CYBIL-CA/AA TYPE AND VARIABLE MAPPING
17.15.2 USER HEAPS

```
BLOCK_KIND: DATA_BLOCK
FILL:  0
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_KIND: FREE_BLOCK
FILL:  0
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 a free block with all three links uninitialized. Thus the header contains only the block kind and the block size. When the first allocate request is made, this free block is transformed into a heap block consisting of a heap header and a free block. Then the allocate request is satisfied from the free block.

Subsequent allocate and free requests cause the heap block to be subdivided into several data blocks and free blocks.

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

A RESET request causes the heap block to be changed back to a free block. Again, only the block kind and block size are reset in the header. The links are left unchanged.

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

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

17.16 CELLS

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

17.17 SUMMARY FOR THE APOLLO

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

18.0 CYBIL-CA/AA RUN TIME ENVIRONMENT

18.0 CYBIL-CA/AA RUN TIME ENVIRONMENT

18.1 MEMORY

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

- Code
- Static Storage
- Stack
- Heap

18.1.1 CODE

The code section contains the instructions of the program.

18.1.2 STATIC STORAGE

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

Static storage may contain the following kinds of sections:

- Read Only Sections
- Read Write Sections

18.1.3 STACK

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

The stack consists of 2 kinds of entries, stack frames and register save areas. The stack consists of alternating stack frames and

register save areas. If there are no registers to save, the size of a register save area is zero.

18.1.3.1 Stack Frame

The stack frame consists of the following parts:

- The Fixed Size Part contains all data whose size is known at compile time.
- The Variable Size Part contains storage allocated using PUSH statements.
- The Argument List Part contains the parameters in the call to the procedure.
- The P-register Part contains the return address.

18.1.3.1.1 FIXED SIZE PART

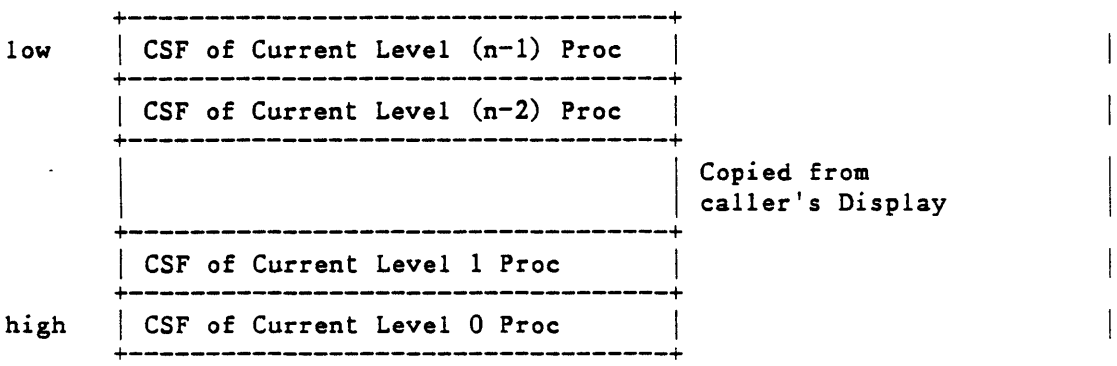
The Fixed Size Part contains some of the data which the procedure may access directly, and addressing information for other data which the procedure may access. The Fixed Size Part contains the following:

- Dynamic Link
- Display
- Automatic Variables
- Register Overflow Area
- Workspace

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:

18.0 CYBIL-CA/AA RUN TIME ENVIRONMENT
18.1.3.1.1 FIXED SIZE PART



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 32-bit address of a stack frame.

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

The Workspace area is used to hold data such as an intermediate result for an expression with sets as operands. This data is PUSH'ed onto the stack.

18.1.3.1.2 VARIABLE SIZE PART

The Variable Size Part contains storage allocated using PUSH statements.

18.1.3.1.3 ARGUMENT LIST PART

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

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

18.1.3.1.4 P-REGISTER PART

The P-register Part contains the return address.

18.1.4 SYSTEM HEAP

The ALLOCATE statement has the following forms:

- ALLOCATE <allocation designator> IN <heap variable>;
- 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:

- FREE <pointer variable> IN <heap variable>;
- 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.

18.1.5 REGISTERS

A0 - POINTS TO THE ENTRY CONTROL BLOCK (ECB) FOR AN XDCL PROCEDURE
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.

18.2 PARAMETER PASSAGE

18.2.1 REFERENCE PARAMETERS

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

18.2.2 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.

18.2.2.1 Value Parameters to Internal Procedures

Value parameters are passed to internal procedures (not XDCLed) as follows:

18.0 CYBIL-CA/AA RUN TIME ENVIRONMENT
18.2.2.1 Value Parameters to Internal Procedures

Type	Copy or Pointer	Parameter List Entry Size
Pointer	copy	2 words for fixed pointer (except sequence). 6 words for fixed ptr to sequence. 3 words for adaptable string pointer. 6 words for adaptable array pointer. 6 words for adaptable sequence pointer. 4 words for adaptable heap pointer. 3-6 words for adaptable record pointer.
Integer	copy	2 words
Character	copy	1 word. The character is in the upper 8 bits of the word with lower bits unused.
Ordinal	copy	1 word
Integer Subrange	copy	1 or 2 words
Boolean	copy	1 word. The boolean value is in the upper 8 bits of the word with lower bits unused.
Real	copy	2 words
Longreal	copy	4 words
Set	copy	1 or 2 words (if the SET size is 4 bytes or less)
	pointer	2 words (if the SET size is greater than 4 bytes)
String	pointer	2 words for fixed string. 3 words for adaptable string.
Array	pointer	2 words for fixed array. 6 words for adaptable array.
Record	copy	1 or 2 words (if the RECORD size is 4 bytes or less)
	pointer	2 words (if the RECORD size is greater than 4 bytes) 3-6 words for adaptable record.
Cell	copy	1 word. The cell is in the upper 8 bits of the word with lower bits unused.

18.0 CYBIL-CA/AA RUN TIME ENVIRONMENT
18.2.2.1 Value Parameters to Internal Procedures

Sequence	pointer	6 words for fixed sequence. 6 words for adaptable sequence.
Heap	pointer	4 words

18.2.2.2 Value Parameters to XDCLed Procedures

If the callee procedure has been XDCLed, the value parameter is copied onto the run time stack. In addition, a pointer to the value is also generated and is passed to the callee.

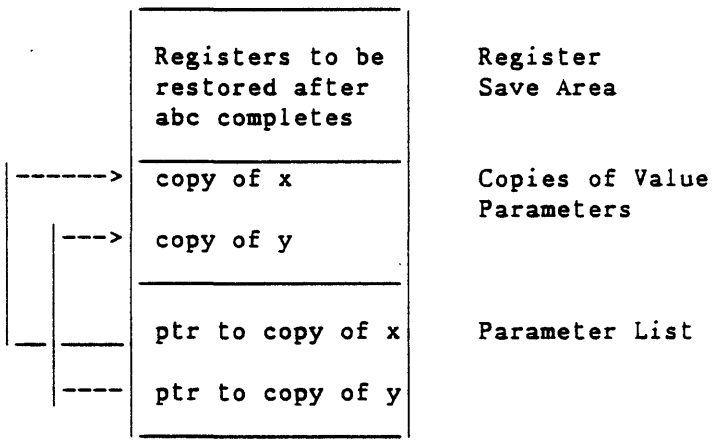
Be aware that all value parameters are copied onto the stack before a pointer to each of them is generated as the parameter to be interpreted by the callee.

The size of the pointer to the copy on the stack is represented by the Entry Size column in the table below. The size of the fixed pointer is always 2 words.

Given a procedure call: abc (x,y);

where x and y are value parameters then

Stack layout at time of call to abc



Refer to the CYBIL-CA/AA Type & Variable Mapping section for a description of the sizes of the copies of the value parameters & the implicitly generated pointers to the value parameters.

18.0 CYBIL-CA/AA RUN TIME ENVIRONMENT
18.2.2.2 Value Parameters to XDCLed Procedures

Value parameters to XDCLed procedures appear in the parameter list as follows:

Type	Copy or Pointer	Parameter List Entry Size
Pointer	pointer	2 words for fixed pointer (except sequence).
	copy	6 words for fixed ptr to sequence. 3 words for adaptable string pointer. 6 words for adaptable array pointer. 6 words for adaptable sequence pointer. 4 words for adaptable heap pointer. 3-6 words for adaptable record pointer.
Integer	pointer	2 words
Character	pointer	2 words. The character value parameter is in the upper 8 bits of the word with lower bits unused.
Ordinal	pointer	2 words
Integer Subrange	pointer	2 words
Boolean	pointer	2 words. The boolean value parameter is in the upper 8 bits of the word with lower bits unused.
Real	pointer	2 words
Longreal	pointer	4 words
Set	pointer	2 words
String	pointer	2 words for fixed string. 3 words for adaptable string.
Array	pointer	2 words for fixed array. 6 words for adaptable array.
Record	pointer	2 words 3-6 words for adaptable record.
Cell	pointer	1 word. The cell is in the upper 8 bits of the word with lower bits unused.

Sequence	pointer	6 words for fixed sequence. 6 words for adaptable sequence.
Heap	pointer	4 words

18.3 VARIABLES

18.3.1 VARIABLE ATTRIBUTES

18.3.1.1 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.

18.3.1.2 #GATE Attributes

Quoting the #GATE attribute causes the internal value parameter code sequence to be generated. This attribute is used within the CYBIL runtime library to interface with calls generated by the compiler. The #GATE attribute should not normally be used.

18.3.2 VARIABLE ALLOCATION

With the exception of byte size variables, 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.

18.3.3 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.

18.4 STATEMENTS

18.4.1 CASE STATEMENT

There are 2 alternative code sequences generated for the CASE statement, depending on the characteristics of the case selectors.

The jump table 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 (JMP) branch instruction to the appropriate case selector.

The other alternative is a code sequence generated via a series of conditional branches. Branch implementation is chosen if the number of case selectors is less than three, or, if the amount of space required for a jump table exceeds the amount of space required for branch instructions. If the number of entries is more than 8, the code sequence that is generated is in the form of a binary search.

18.4.2 STRINGREP

18.4.2.1 Pointer Conversions

The default radix for the conversion of a pointer into a string is defined as implementation dependent. For the Apollo the resultant string will be the pointer represented in hexadecimal notation.

18.5 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 and functions 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. Agreement between data types must be exact. Even names used to define data types must be identical. For example, the following seemingly identical definitions produce different hashes:

```
MODULE one;
  TYPE
    p_type = 0 .. 65535;
  PROCEDURE [XREF] two (p: p_type);
MODEND one;

MODULE two;
  PROCEDURE [XDCL] two (p: 0 .. 65535);
  PROCEND two;
MODEND two;
```

18.6 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

18.7 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 register A0 if the function value is a pointer or in register D0 if it is a scalar.

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.

18.8 PROCEDURE CALL AND RETURN INSTRUCTION SEQUENCES

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
A0,D0 - RETURNED VALUE (which register depends on value type)

18.8.1 PROCEDURE CALL

The following illustrate instruction sequences for procedure calls:

External Procedure (or XDCL Internal Procedures) without Parameters

```
MOVEM.L  reglist,-(DSP)        Save regs as needed
LEA.L    proc_ptr(base),A0
JSR      (a0)                  Call Procedure
MOVEM.L  (DSP)+,reglist        Restore regs as needed
```

Internal Procedure (not XDCLed) and Static Link (SL)

```
MOVEM.L  reglist,-(DSP)        Save regs as needed
MOVEA.L  CSF,SL                Static Link
BSR      internal_proc         Call Procedure
MOVEM.L  (DSP)+,reglist        Restore regs as needed
```

Pointer to Procedure

```
MOVEM.L  reglist,-(DSP)        Save regs as needed
MOVEM.L  proc_ptr(base),A0/SL  Proc Addr & Static Link
JSR      (A0)                  Call Procedure
MOVEM.L  (DSP)+,reglist        Restore regs as needed
```

18.0 CYBIL-CA/AA RUN TIME ENVIRONMENT
18.8.1 PROCEDURE CALL

Setup Argument List on Internal Procedure Call

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

MOVEM.L  reglist,-(DSP)         Save regs as needed
MOVE.B   A(CSF),-(DSP)          1-byte parameter
MOVE.W   B(CSF),-(DSP)          1-word parameter
MOVE.L   C(CSF),-(DSP)          2-word parameter
MOVE.W   D+4(CSF),-(DSP)        3-word parameter
MOVE.L   D(CSF),-(DSP)
BSR      internal_proc          Call Procedure
LEA      plist_size(A7),(A7)    Pop parameters
MOVEM.L  (DSP)+,reglist         Restore regs as needed
```

Setup Argument List on an External Procedure Call

```
external_proc(A,B,C)            Where A and C are value parameters
                                and B is a reference parameter

MOVEM.L  reglist,-(DSP)         Save registers as needed
MOVE.B   A(CSF),-(DSP)          Make copy of A
MOVE.L   C(CSF),-(DSP)          Make copy of C
LEA.L    6(DSP),A0              Address of copy of A
MOVE.L   A0,-(DSP)              Parameter 1 defined
LEA.L    B(CSF),A0              Address of copy of B
MOVE.L   A0,-(DSP)              Parameter 2 defined
LEA.L    0C(DSP),A0             Address of copy of C
MOVE.L   A0,-(DSP)              Parameter 3 defined
LEA.L    proc_ptr(base),A0      Load procedure ECB address
JSR      (A0)                   Call procedure
LEA      plist_size(A7),(A7)    Pop parameters and copies of A & C
MOVEM.L  (DSP)+,reglist         Restore registers
```

18.9 PROLOG

The basic instruction sequence for the prolog is as follows:

18.0 CYBIL-CA/AA RUN TIME ENVIRONMENT
18.9 PROLOG

```
prolog:
  LINK      CSF,#-frame_size       Form Dyn Link & Update DSP
```

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

  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.

18.10 EPILOG

The basic instruction sequence for the epilog is as follows:

18.0 CYBIL-CA/AA RUN TIME ENVIRONMENT
18.10 EPILOG

```
epilog:
  UNLK     CSF                  DSP := CSF
                                DSP := DSP + 4
                                CSF := (DSP)
  ADD.W    #8,A7                POP the zero and the ECB address
  MOVE.L   (A7)+,A5             Restore A5
  RTS                           Return
```

18.11 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_ABORT - Contains procedure CYP_ABORT for aborting the program.
- o CYM_ALLOCATE - Contains procedure CYP_ALLOCATE for allocating a block in a user heap.
- o CYM_FREE - Contains procedure CYP_FREE for freeing a block in a user heap.
- o CYM_NIL_ERROR - 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_4_BYTES_BY_4_BYTES - Contains procedure CYP_MPY_4_BY_4 for integer multiplication.
- o CYM_DIV_4_BYTES_BY_4_BYTES - Contains procedure CYP_DIV_4_BY_4 for integer division.
- o CYM_MOD_4_BYTES_BY_4_BYTES - Contains procedure CYP_MOD_4_BY_4 for integer remainder.

In addition to the above, the compiler generates calls to the following procedures for system heap management:

- CYP_SYS_ALLOC - Procedure to allocate a block in the system heap.

- CYP_SYS_FREE - Procedure to free a block in the system heap.

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.

18.12 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.

19.0 CYBIL-CP TYPE AND VARIABLE MAPPING

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

The Pcode interpreter supports three basic data types as follows:

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

Integers are represented in two's complement form.

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

19.1 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.

19.1.1 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.

19.1.2 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.

19.1.3 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.

19.1.4 POINTER ALIGNMENT

All pointer types are word aligned.

19.2 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.

19.3 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.

19.4 ORDINALS

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

19.5 SUBRANGES

19.5.1 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.

19.5.2 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.

19.6 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.

19.7 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.

19.0 CYBIL-CP TYPE AND VARIABLE MAPPING
19.7 REALS

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

19.8 LONGREALS

Treated the same as reals.

19.9 SETS

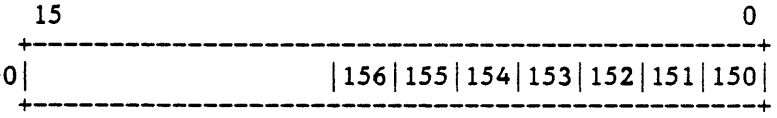
The number of contiguous bits required to represent a set is the number of elements in the base type of the associated set type. The rightmost bit represents the first element, the next bit represents the second element, etc.

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

Example -

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

Set A resides as follows:



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

A set variable is mapped as an unpacked set type.

The maximum size allowed for a set is 4079 elements.

19.10 STRINGS

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

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

19.0 CYBIL-CP TYPE AND VARIABLE MAPPING
19.10 STRINGS

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.

19.11 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.

19.12 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.

19.13 SEQUENCES

A sequence type consists of the data area required to contain the span(s) requested by the user. A sequence type is always word

aligned, and occupies an integral number of words.

19.14 HEAPS

19.14.1 SYSTEM HEAP

The system heap is as described in the UCSD manuals.

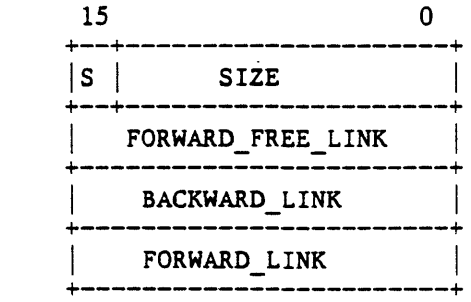
19.14.2 USER HEAPS

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

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

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

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



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

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

```
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

19.15 CELLS

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

19.16 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
REAL	word	2 words	word	2 words
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

20.0 CYBIL-CP RUN TIME ENVIRONMENT

20.0 CYBIL-CP RUN TIME ENVIRONMENT

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

20.1 MEMORY

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

- Code and Literals
- Static Storage
- Stack Heap Area

20.1.1 CODE AND LITERALS

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

- Pointers to procedures
- Calls to external procedures

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

20.1.2 STATIC STORAGE

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

20.1.3 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.

20.1.3.1 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.

20.1.3.1.1 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.

20.1.3.2 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.

20.1.3.2.1 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.

20.1.3.2.2 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.

20.1.4 HEAP

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

20.1.4.1 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.

20.1.4.2 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.

20.2 PARAMETER PASSAGE

20.2.1 REFERENCE PARAMETERS

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

20.2.2 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.

20.3 VARIABLES

20.3.1 VARIABLE ATTRIBUTES

20.3.1.1 Variables in Sections

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

20.3.1.2 Read Attribute

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

20.3.1.3 #GATE Attributes

The #GATE attribute is ignored.

20.3.2 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.

20.3.3 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.

20.4 STATEMENTS

20.4.1 STRINGREP

20.4.1.1 Pointer Conversion

The default radix for the conversion of a pointer into a string is defined as implementation dependent. For Pcode the resultant string will be the pointer represented in hexadecimal notation.

20.5 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.

20.6 EXTERNAL NAMES

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

20.7 PROCEDURE REFERENCE

20.8 FUNCTION REFERENCE

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

20.9 PROCEDURE CALL AND RETURN INSTRUCTION SEQUENCES

20.9.1 PROCEDURE CALL

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

```
SLDC 0
```

would be appropriate for an integer function call.

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

Should the called procedure have parameters, then the parameter

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

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

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

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

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

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

Examples:

CPL p-ord	Used to call local (child) procedures to the calling procedure and its body (i.e., LEX = +1).
SCPI 1 p-ord	Used to call sibling procedures of the calling procedure (LEX = 0).
SCPI 2 p-ord	Used to call parent procedures of the calling procedure (LEX = -1).
CPI n p-ord	Used to call intermediate, but non-global procedures

(LEX < -1).

CPG p-ord — Used to call outer level procedures local to this module.

CXG seq p-ord — Used to call XREF procedures that are located in other compilation units.

CPF — Used to call formal procedures that have been introduced in CYBIL text as pointers to procedures.

20.10 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.

20.11 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.

20.12 RUN TIME LIBRARY

20.12.1 UNKNOWN AND/OR UNEQUAL LENGTH STRINGS

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

20.12.1.1 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).
```

20.12.1.2 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.

21.0 CYBIL-CS/SS TYPE AND VARIABLE MAPPING

21.1 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 := 000000000001(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).

21.1.1 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.

21.0 CYBIL-CS/SS TYPE AND VARIABLE MAPPING
21.1.1 ADAPTABLE POINTERS

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

21.1.2 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.

21.1.3 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 ].
```

21.1.4 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.

21.1.5 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.

21.2 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.

The NIL relative pointer is the following constant:

```
NIL : RELATIVE_ADDRESS := 80000000(16).
```

21.2.1 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.

21.2.2 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.
```

21.2.3 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.

21.3 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, with the leftmost 16 bits being zeroes.

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).

21.4 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.

21.5 ORDINALS

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

21.6 SUBRANGES

An unpacked subrange type is allocated 8 bytes and is word 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)))
```

21.0 CYBIL-CS/SS TYPE AND VARIABLE MAPPING
21.6 SUBRANGES

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.

The maximum integer subrange is -800000000000(16) .. 7FFFFFFFFFFF(16).

21.7 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.

21.8 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**(-28625)) to (2**(28719)). The range of useful coefficients is from 800000000000(16) to 7FFFFFFFFFFF(16) which provides a range of -2 ** 47 through (2 ** 47) -1. Useful exponents range from 9000(16) to 6FFF(16) which provides a range of -28672 to 28671.

21.9 LONGREALS

Longreal type variables are handled identical to real type variables.

21.10 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

21.0 CYBIL-CS/SS TYPE AND VARIABLE MAPPING
21.10 SETS

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 left justified on a word boundary.

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.

21.11 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.

21.12 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. When arrays and records are fields of a packed structure the nested structure begins on a word 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.

21.0 CYBIL-CS/SS TYPE AND VARIABLE MAPPING
21.13 RECORDS

21.13 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. When arrays and records are fields of a packed structure the nested structure begins on a word boundary.

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 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.

The content of the dead bits is undefined.

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

The fields are allocated consecutively subject to their alignment restrictions. To get records to pack up tight, use booleans, ordinals, subranges or sets as they will bit align. However, no subrange or set will cross a word boundary; if it spills into the next word, the whole thing will be mapped into the next word.

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).

21.14 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.

21.14.1 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.

21.14.2 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.

21.15 CELLS

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

21.16 DETAILED SUMMARY FOR THE C200

Data Type	Usage	Number of bits used	Alignment	# of significant bits	Alignment of signif. bits
Pointer to fixed size object	Variable	64	Word	48	Right
	In packed rec	64	Word	48	Right
	In unpacked rec	64	Word	48	Right
	In packed array	64	Word	48	Right
	In unpack array	64	Word	48	Right
Pointer to adaptable string	Variable	128	Word	len=48	Right
			Word	len=16	Right
	In packed rec	128	Word	len=48	Right
			Word	len=16	Right
	In unpacked rec	128	Word	len=48	Right
			Word	len=16	Right
	In packed array	128	Word	len=48	Right
			Word	len=16	Right
	In unpack array	128	Word	len=48	Right
			Word	len=16	Right
Pointer to adaptable array	Variable	256	Word	len=48	Right
			Word	Desc=192	Right
	In packed rec	256	Word	len=48	Right
			Word	Desc=192	Right
	In unpacked rec	256	Word	len=48	Right
			Word	Desc=192	Right
	In packed array	256	Word	len=48	Right
			Word	Desc=192	Right
	In unpack array	256	Word	len=48	Right
			Word	Desc=192	Right

21.0 CYBIL-CS/SS TYPE AND VARIABLE MAPPING
21.16 DETAILED SUMMARY FOR THE C200

Data Type	Usage	Number of bits used	Alignment	# of significant bits	Alignment of signif. bits
Pointer to user heap	Variable	128	Word	ptr=48	Right
			Word	len=48	Right
	In packed rec	128	Word	ptr=48	Right
			Word	len=48	Right
	In unpacked rec	128	Word	ptr=48	Right
			Word	len=48	Right
	In packed array	128	Word	ptr=48	Right
			Word	len=48	Right
	In unpack array	128	Word	ptr=48	Right
			Word	len=48	Right
Pointer to sequence (fixed or adaptable)	Variable	192	Word	ptr=48	Right
			Word	Desc=128	Right
	In packed rec	192	Word	ptr=48	Right
			Word	Desc=128	Right
	In unpacked rec	192	Word	ptr=48	Right
			Word	Desc=128	Right
	In packed array	192	Word	ptr=48	Right
			Word	Desc=128	Right
	In unpack array	192	Word	ptr=48	Right
			Word	Desc=128	Right
Pointer to adaptable record	Variable	64+n	Word	ptr=48	Right
	In packed rec	64+n	Word	ptr=48	Right
	In unpacked rec	64+n	Word	ptr=48	Right
	In packed array	64+n	Word	ptr=48	Right
	In unpack array	64+n	Word	ptr=48	Right
	see type of adaptable for descriptor				
Bound Variant record pointer	Variable	128	Word	ptr=48	Right
			Word	len=48	Right
	In packed rec	128	Word	ptr=48	Right
			Word	len=48	Right
	In unpacked rec	128	Word	ptr=48	Right
			Word	len=48	Right
	In packed array	128	Word	ptr=48	Right
			Word	len=48	Right
	In unpack array	128	Word	ptr=48	Right
			Word	len=48	Right

21.0 CYBIL-CS/SS TYPE AND VARIABLE MAPPING
21.16 DETAILED SUMMARY FOR THE C200

Data Type	Usage	Number of bits used	Align-ment	# of signif-icant bits	Align-ment of signif. bits
Relative Pointer	Variable	32	Word	32	Right
	In packed rec	32	Byte	32	Left
	In unpacked rec	32	Word	32	Right
	In packed array	32	Byte	32	Left
	In unpack array	32	Word	32	Right
Adaptable relative pointer	Variable	64+n	Word	ptr=32	Right
		see type of adaptable for descriptor			
	In packed rec	64+n	Word	ptr=32	Right
		see type of adaptable for descriptor			
	In unpacked rec	64+n	Word	ptr=32	Right
		see type of adaptable for descriptor			
	In packed array	64+n	Word	ptr=32	Right
		see type of adaptable for descriptor			
	In unpack array	64+n	Word	ptr=32	Right
		see type of adaptable for descriptor			
Relative Pointer to Sequence (fixed or adaptable)	Variable	192	Word	ptr=32	Right
			Word	Desc=128	Right
	In packed rec	192	Bit	ptr=32	Left
			Word	Desc=128	Right
	In unpacked rec	192	Word	ptr=32	Right
			Word	Desc=128	Right
	In packed array	192	Bit	ptr=32	Left
			Word	Desc=128	Right
	In unpack array	192	Word	ptr=32	Right
			Word	Desc=128	Right
Relative Pointer to Bound Variant Record	Variable	128	Word	ptr=32	Right
			Word	len=48	Right
	In packed rec	128	Word	ptr=32	Right
			Word	len=48	Right
	In unpacked rec	128	Word	ptr=32	Right
			Word	len=48	Right
	In packed array	128	Word	ptr=32	Right
			Word	len=48	Right
	In unpack array	128	Word	ptr=32	Right
			Word	len=48	Right

21.0 CYBIL-CS/SS TYPE AND VARIABLE MAPPING
21.16 DETAILED SUMMARY FOR THE C200

Data Type	Usage	Number of bits used	Alignment	# of significant bits	Alignment of signif. bits
Pointer to procedure	Variable	128	Word	addr=48	Right
			Word	addr=48	Right
	In packed rec	128	Word	addr=48	Right
			Word	addr=48	Right
	In unpacked rec	128	Word	addr=48	Right
			Word	addr=48	Right
	In packed array	128	Word	addr=48	Right
			Word	addr=48	Right
	In unpack array	128	Word	addr=48	Right
			Word	addr=48	Right
Integer	Variable	64	Word	48	Right
	In packed rec	64	Word	48	Right
	In unpacked rec	64	Word	48	Right
	In packed array	64	Word	48	Right
	In unpack array	64	Word	48	Right
Characters	Variable	8	Word	8	Right
	In packed rec	8	Byte	8	Left
	In unpacked rec	8	Word	8	Right
	In packed array	8	Byte	8	Left
	In unpack array	8	Word	8	Right
Subrange and ordinals	Variable		Word		Right
	In packed rec		Bit		Right
	In unpacked rec	See	Word	See	Right
	In packed array	Above	Bit	Above	Right
	In unpack array		Word		Right
Booleans	Variable	64	Word	1	Right
	In packed rec	1	Bit	1	Left
	In unpacked rec	64	Word	1	Right
	In packed array	1	Bit	1	Left
	In unpack array	64	Word	1	Right

21.0 CYBIL-CS/SS TYPE AND VARIABLE MAPPING
21.16 DETAILED SUMMARY FOR THE C200

Data Type	Usage	Number of bits used	Alignment	# of significant bits	Alignment of signif. bits
Real	Variable	64	Word	64	Right
	In packed rec	64	Word	64	Right
	In unpacked rec	64	Word	64	Right
	In packed array	64	Word	64	Right
	In unpack array	64	Word	64	Right
Longreal	Variable	64	Word	64	Right
	In packed rec	64	Word	64	Right
	In unpacked rec	64	Word	64	Right
	In packed array	64	Word	64	Right
	In unpack array	64	Word	64	Right
Sets	Variable		Word		Left
	In packed rec		Bit		Left
	In unpacked rec	See	Word	See	Left
	In packed array	Above	Bit	Above	Left
	In unpack array		Word		Left
Strings	Variable	n bytes	Word	n bytes	Left
	In packed rec	n bytes	Byte	n bytes	Left
	In unpacked rec	n bytes	Byte	n bytes	Left
	In packed array	n bytes	Byte	n bytes	Left
	In unpack array	n bytes	Byte	n bytes	Left
Cell	Variable	64	Byte	8	Left
	In packed rec	8	Byte	8	Left
	In unpacked rec	64	Byte	8	Left
	In packed array	8	Byte	8	Left
	In unpack array	64	Byte	8	Left

21.0 CYBIL-CS/SS TYPE AND VARIABLE MAPPING
21.17 SUMMARY FOR THE CYBER 200

21.17 SUMMARY FOR THE CYBER 200

TYPE	SIZE	ALIGNMENT		
		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.

22.0 CYBIL-CS/SS RUN TIME ENVIRONMENT

22.1 REGISTER AND STORAGE MODELS

22.1.1 STORAGE MANAGEMENT - DYNAMIC VS. STATIC

22.1.1.1 Display Vector

The CYBIL language has a concept of based storage to permit reentrancy / recursion. As such, three levels of organization of dynamic space are used:

- o reentrant space - allocated for each activation of a process
- o static scope space - the current static scope of a procedure
- o dynamic scope space - the current dynamic set of values to be associated with static scope

The device for implementing and maintaining organization in this space is usually called the "display vector". Conceptually, the display vector is an unbounded, three dimensional array of pointers to data space (each is a "base", as noted above). A pseudo-declaration of the display vector might be:

```
CONST
  maximum_recursion = infinity,
  maximum_lex_level = infinity,
  maximum_number_of_tasks = infinity;

TYPE
  base = ↑data_space_block,
  dynamic_scope_space = ARRAY [0..maximum_recursion] OF base,
  static_scope_space  = ARRAY [0..maximum_lex_level] OF
                          dynamic_scope_space,
  reeentrant_space = ARRAY [0..maximum_number_of_tasks] OF
                       static_scope_space;

VAR
  display_vector: reentrant_space;
```

In general, the display vector is not implemented in this fashion. Indeed the infinity implied is impractical. More importantly code access to data is essentially concerned only with the "middle index",

static scope.

The outer index, reentrant space is operating system defined, provided, and maintained; it is allocated at process creation. Programmatically, a process is only concerned with the space allocated for its execution - it is not concerned with other activations (reentrancies).

The inner index, dynamic space, is generally maintained by procedure linkage mechanisms, and is distributed through stack management logic. This results from the fact that only the most recent activation's data space is available for access; it is only necessary to be able to delink an activation (pop the auto frame) as a procedure terminates.

The middle index is maintained as a singly dimensioned array (vector) of bases to current activations at each lex level, in as much as these bases are used in virtually all data accesses.

22.1.1.2 CYBIL Static Space

Each CYBIL module may contain outer (level 0) variable declarations. Such items are static with respect to the module. (CYBIL also permits STATIC, XREF and XDCL attributes, having the same effect.)

A load module may consist of several CYBIL modules, and thus several static areas. Each of these is compile time (ergo, loader) allocatable, analogous to code space. Each contained variable is located relative to the start of its static block; therefore, it may be accessed by its relative relocatable address. On the other hand, it may also be treated as based on its static block origin.

22.1.2 INTRA-MODULAR BINDING SECTIONS

Because a CYBIL module may contain several procedures, and since both the module and the procedures individually have static (binding section) requirements, and further since the VSOS conventions will be observed, the binding section of a CYBIL module (analogous to "data base" in VSOS) must be partitioned to accommodate module and procedure specific needs.

22.2 REGISTER USAGE CONSIDERATIONS

The CYBER 200 offers a large set of manipulatable registers, together

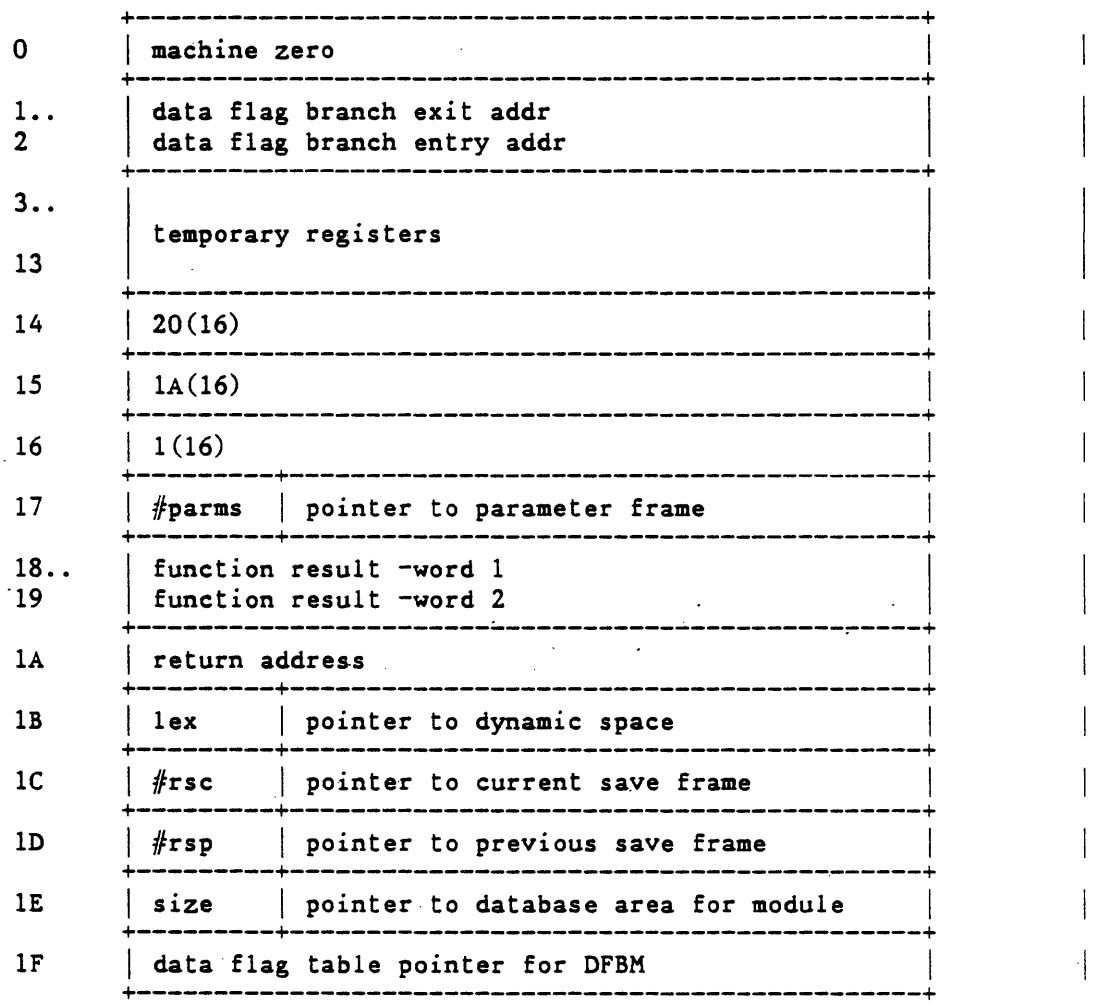
with a means of loading/unloading them (SWAP).

22.2.1 VSOS CONVENTIONS

VSOS conventionalizes the first 20(16) registers. C200 CYBIL will observe these conventions.

It is noted that four of these registers contain the frequently used constants machine zero, one(binary), 1A(16) and 20(16).

22.0 CYBIL-CS/SS RUN TIME ENVIRONMENT
22.2.1 VSOS CONVENTIONS



Register File - VSOS Conventional Usage

22.2.2 DISPLAY VECTOR

Virtually all references to based variables, and optionally static variables, involve adding an offset to the base for computing an address. Register residence is thus indicated for the static-scope index of the display vector. Viewed from a given procedure, the display vector components from static to its lex level should be resident.

Static variables may be addressed by their known location, rather than by their known offset from a base (also known), but this leads to code relocation, which has been avoided in the CYBIL code generator for CYBER 200.

22.2.3 CONSTANTS

Constants derive from many sources in CYBIL program decomposition.

The kinds of constants are:

- scalar computational constants
- real computational constants
- structured computational constants
- pointers to (addresses of) XREF variables
- address offset constants
- addresses of procedures
- addresses of procedure binding sections (data base)

Of the above, only structured constants cannot be register resident; however, their addresses may be. Constants are a major candidate for register residency. C200 CYBIL does, on a procedure basis, determine all used constants, select the subset most frequently used (if register space is limited), and cause them to be preloaded via SWAP in procedure prologue.

22.2.4 OPERAND STACK/PARAMETER FRAME

Use of registers for the operand stack is highly desirable to avoid

superfluous store/load operations. Further, the data mappings selected for C200 CYBIL dictate that scalars will be register sized and manipulatable, obviating the more cumbersome storage to storage operations. Further, in the event of embedded function references, the operand stack must be saved, then reloaded and extended, a natural function in the VSOS linkage convention (if they are register resident).

From the point of view of the calling proc, the parameters to a callee proc are like operands. To the callee, however, they are members of its automatic stack frame. More on this in the next section; at this point is noted the desirability of treating these parameters as operand stack elements.

22.2.5 VARIABLES

It is natural to consider making local automatic variables register resident (RRV). The access speed is obvious. However, several CYBIL realities complicate this, and in some instances lead to difficult to resolve ambiguities, viz:

- Large CYBIL variables will not fit, causing the automatic stack frame to be divergent in character
- Pointers to (or addresses of) register resident variables cannot be easily visualized, and are ambiguous
- What is local to a given procedure, is global to a contained proc, causing complex interpretation of static scope/access.

However, variables meeting the following tests are candidates for register residency. Experience shows many local variables will satisfy these tests.

- Variable is not the object of a pointer, no matter how disguised (either by dereference, actual ref parameter, etc.) and has no component which is
- The variable's size is such that it may be cradled in a register - <= 8 bytes
- The variable is not globally accessed, which would put it in the static scope (storage resident) of code generated elsewhere
- Variable is not a complex pointer (currently, all such pointers exceed 8 byte length)

- Its type is scalar (integer, ordinal, character, boolean, subrange), pointer, cell, or real. String type is not permitted inasmuch as substring references are indirect and could confuse algorithms for string operations. Storage, structured and adaptable types are verboten as they are dynamically allocated and/or indirectly referenced.

Note that nothing prevents the addresses of objectionable objects from being register resident.

22.3 MULTI-FUNCTION SOLUTION FOR REGISTER/STACK/STORAGE

The solution described below accommodates the somewhat unrelated, and sometimes conflicting, requirements of:

- VSOS conventions, particularly regarding linkage and registers
- CYBIL essentials regarding linkage
- Automatic stack frame linkage
- Conversion of parameters to callee stack frame elements
- Function return value conversion
- Register resident variables
- Management of display vector dynamic and static scope
- CYBIL <--> non-CYBIL environment linkage

The dynamic nature of the stack frame and its contents renders static verbal description difficult. The only "time" a procedure's entire stack frame is storage resident is when the procedure has called another procedure (which, following VSOS convention, has saved the register file of the caller). The following descriptions begin with this state, then proceed to an abstraction holding during active execution of the given procedure. In conclusion, the "binding section" is discussed.

22.3.1 STORAGE RESIDENT PICTURE OF STACK FRAME

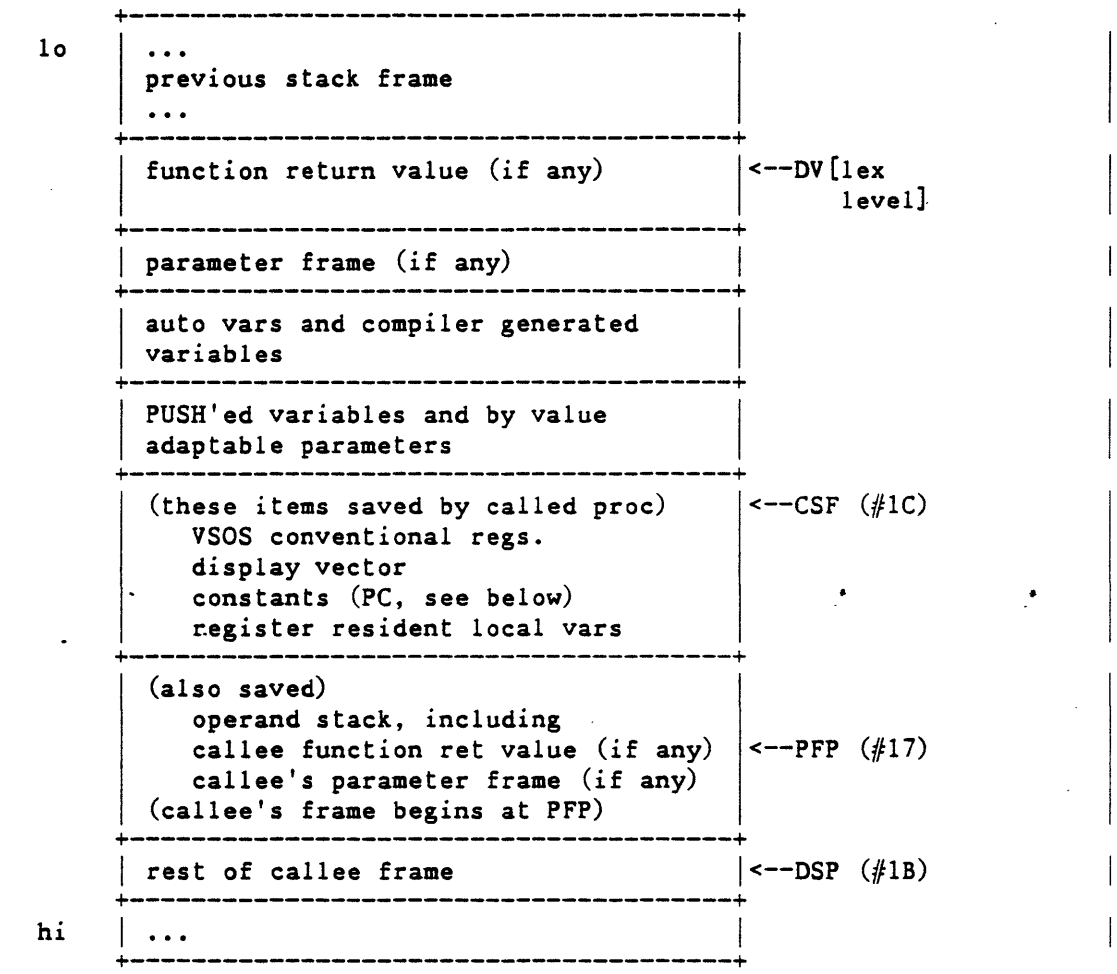
The following figure depicts the stack frame of a given procedure when it is activated but has called another procedure, thus being entirely storage resident. The pointers indicated are the contents

22.0 CYBIL-CS/SS RUN TIME ENVIRONMENT
22.3.1 STORAGE RESIDENT PICTURE OF STACK FRAME

of the noted VSOS conventional registers, as well as the display vector entry for the lexical level of this procedure.

Note also that various segments of the depicted frame may in fact be empty in a given instance.

22.0 CYBIL-CS/SS RUN TIME ENVIRONMENT
22.3.1 STORAGE RESIDENT PICTURE OF STACK FRAME



Stack Frame Layout

The frame begins at DV[lex level] and terminates before PFP.

22.3.2 REGISTER FILE ABSTRACTION OF STACK FRAME

During execution within the subject procedure, a portion of the stack frame is made register resident for efficiency. Initially, the subject procedure's prologue initializes the areas from VSOS convention registers through the PC (Preferred Constants). In

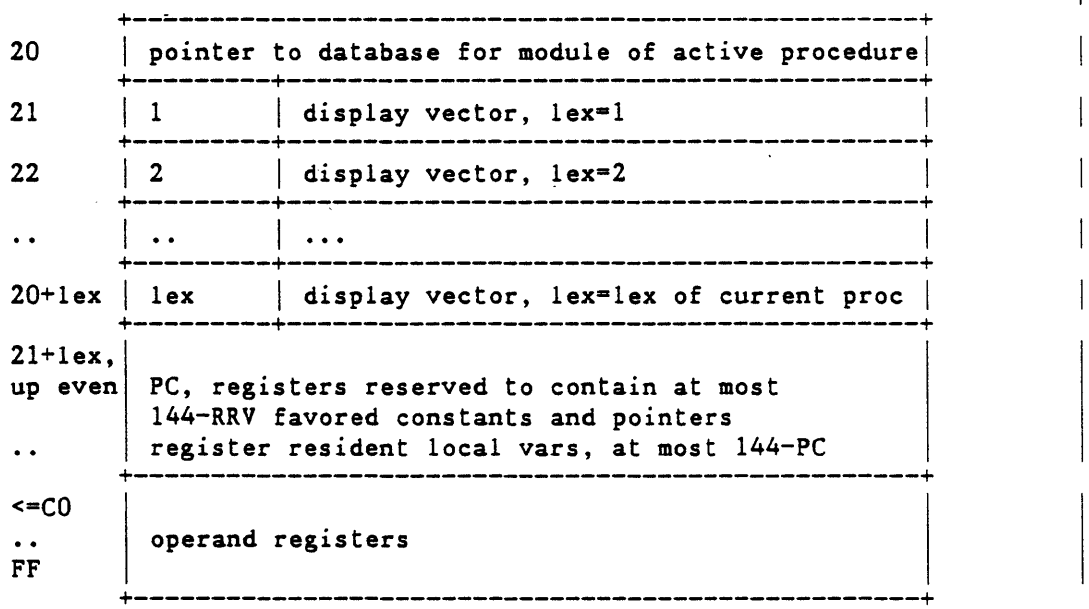
storing its caller's registers, it has defined the function return value / parameter frame (which were operand stack elements in the caller). Ensuing calls to other procedures result in restoration of the register abstract (VSOS convention).

Thus the registers contain dynamically the segment between CSF and DSP.

Upon termination of this given procedure, its stack frame is discarded (popped) with the exception of the function return value.

Upon return to this procedure from a called procedure, the code generator model discards that portion of the operand stack which was the parameter frame for the called procedure (no physical action taken). This is detailed more fully in the succeeding section on linkage.

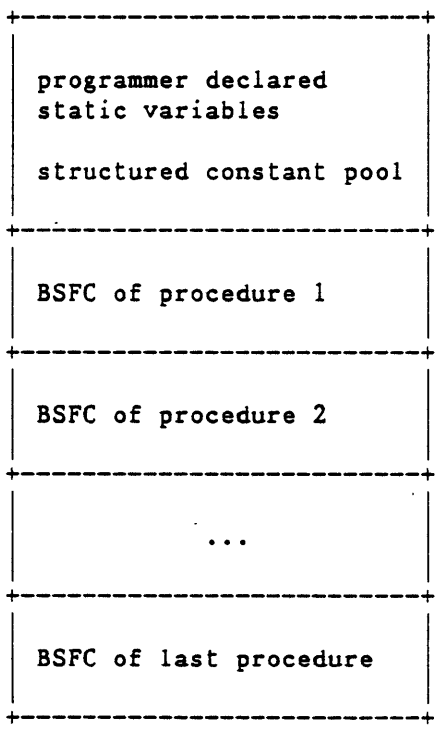
The following figure summarizes the register file abstract.



Register File - CYBIL Convention Extension

22.3.3 BINDING SECTION (VSOS "DATA BASE")

The Binding Section is created per CYBIL module, and is of course static. It contains all global (CYBIL static) variables. In addition, the remainder is subsetted into Binding Section/Favored Constants (BSFC's), one per contained procedure.



Static (Database) Storage Layout

22.0 CYBIL-CS/SS RUN TIME ENVIRONMENT
22.3.3 BINDING SECTION (VSOS "DATA BASE")

Because the size of a BSFC is unlimited (theoretically), but the register resident portion is bounded, each BSFC area is divided into two areas, as shown in Figure 2.5.

preferred const's PC	prologue usable constants pointer to XREF vars pointer to XREF procs pointer to proc binding sect offsets of based (auto) vars computational constants real constants pointer to structured csts
overflow const's	pointer to XREF vars pointer to XREF procedures pointer to XREF proc data base

Expansion of BSFC

22.0 CYBIL-CS/SS RUN TIME ENVIRONMENT
22.3.3 BINDING SECTION (VSOS "DATA BASE")

The next figure illustrates the PC region "prologue usable constants". The full import of this is discussed later: here is noted the construct is devised to simplify procedure prologue and to implement "single-swap" procedures. In the figure, the environment target register is given, followed by the PC offset of the pre-load quantity.

1A: 0	---
1B: 1	lex level of procedure in length field, plus size(auto frame)+size(max save regs) raised even
1C: 2	rounded up even of size of function return value, parameter frame, and automatic variables, plus max save regs (evened) in length field
1D: 3	---
1E: 4	---
1F: 5	---
20: 6	pointer to database (static)
	... --- ...
20+lex: 6+lex	negative size of function return value, if local proc/func - 0 otherwise (XDCL and prog)

Expansion of PC for Prologue Purposes

The Preferred Constant (PC) area is to be loaded by procedure prologue as the PC register constant area, with the balance remaining storage resident.

The prologue usable constants include lexical level, frame sizings, and pointer to module static (data base) useful at prologue.

Note that the PC area is sizewise defined by the register area available for it. It is quite possible that the code generator would

have desired to include more of the items in the categories listed than space permitted. In this event, and with the exception of procedure pointers, the generated code will contain ES/EX instructions to create the desired values, as opposed to loading them from a storage area such as the "overflow" constants.

22.4 PROCEDURE CALL AND LINKAGE

This section describes the object code generated for calling and exiting procedures. It uses the conventions of the VSOS with respect to the register file and dynamic space. It also anticipates that an XDCL CYBIL procedure may have been called from a non-CYBIL environment and that a call to an XREF procedure may leave the CYBIL environment.

The basic method of passing parameters is with a parameter frame. Interfacing with other languages that pass via registers, etc. is not considered here.

The parameter frame is actually created in the operand register area of the caller. The prologue of the callee will save the caller's registers including their mutual parameter frame in a fashion that upon completion, the parameter frame falls into the low area of the callee's stack frame. This is accomplished with the SWAP instruction in the callee's prologue.

22.4.1 PARAMETER FORMATION

The parameter formation will follow contiguously through the area.

Each actual parameter is evaluated and either the actual value or the address of the actual variable is placed into the parameter area. Some actual values (pointers with descriptors) and variable references (adaptable structures) may require several registers.

22.0 CYBIL-CS/SS RUN TIME ENVIRONMENT
22.4.1 PARAMETER FORMATION

The following chart indicates the created register contents per parameter:

Formal Param Type	By Value	By Reference
boolean integer subrange ordinal character real long real cell fixed pointer fixed rel ptr	value	address
pointer to adaptable	value with desc.	address of pointer variable
fixed string	address	address
adapt string	see by ref	address of string and length
set	address	address
fixed array	address	address
adapt array	see by ref	address of array and its descriptor - size, lower and upper bound
fixed record	address	address
adapt record	see by ref	address of record and its descriptor per adaptable field
fixed storage	address	address
adapt storage	see by ref	address with descriptor

Those parameters passed by value that have their addresses placed into a register will have the callee's prologue create a local copy of the parameter value into its own stack frame.

22.4.2 PROCEDURE CALL

After the parameters have been formed, the object procedure is called. (The pseudo code in this document uses the decision construct IF...THEN...ELSE. This construct does not appear in the object code, but rather represents decisions made by the compiler in determining which sequence of code to emit.) The following steps in pseudo code indicate the object instructions:

set the pointer to the callee's database into register #1E

for calls within this module this will be this module's static pointer; for XREF procedure calls it will be to the static area of the module containing the XREF procedure

form parameter frame pointer in #17

enter number of params to length fld of #17; since the call may leave the CYBIL environment, the register must be set up for potential use by the callee

set the address of the target procedure into a register

if the call is to an XREF procedure, store the actual parameters at the location pointed to by the parameter frame pointer (PFP) in register #17. This is necessary in the event the procedure is an IMPL procedure. IMPL accesses the parameters BEFORE its prologue SWAP!

execute a BSAVE instruction

At this point control has been transferred to the called procedure.

22.4.3 PROCEDURE PROLOGUE

The prologue for CYBIL procedures is based upon VSOS conventions with two extensions:

- o an automatic stack frame exists for the procedure which

contains:

- function return value, zero (for pure procedures) or more words at offset zero of the frame
- parameters, zero (for parameterless procedures) or more words
- local declared variables, zero or more words for locally declared variables
- implied variables, the compiler may generate local temporary storage for intermediate results (e.g. set operations, FOR loop control)
- dynamic storage, for objects of PUSHed pointers or storage of adaptable value parameters

o display vector residing in registers 21(16)..20(16)+lex where lex is the lexicographical level of the active (called) procedure

In the next discussion, use is made of the PC area depicted above.

The steps in pseudo code for the prologue consists of:

save registers (1B,) 1C into (10,) 11

these are the immediately useful environment registers that must be kept available for some remaining prologue computation. Note that register 1B (DSP) is only saved in XDCL and program prologues.

compute the address of the callee's register initialization area and set length (raised even), into #1E

add the offset (compiler known constant) of the callee's BSFC area to the address of the module's database (contained in #1E)

protect data flag register #1F

in order to defend 1F against interrupts occurring during the swap, it is stashed in the BSFC location which will be loaded into 1F by the swap.

save the caller's registers at the area designated by #1C and load the PC registers from the BSFC in static

this will not only save the caller's registers into an area just before the start of the callee's stack frame, but also store the parameter values into the front of the callee's stack frame It also loads the PC with 1) lex level, frame sizes, and pointer to static for prologue use, 2) hard constants, 3) pointers to XREF objects, and 4) address offsets to static and automatic variables

set 1D from 11

this has the effect of aging CSF to PSF

set up this levels display vector entry

IF this is XDCL, add preloaded BSFC constant 0 to old DSP (10). IF local, add preloaded constant (-size func ret val) to PFP (17).

update 1B (DSP) packing in lex level

the BSFC will contain lex level for 1B in the address field, and the sum of SIZE(auto frame) (evened) and SIZE(max regs to save during calls) (evened). If XDCL, this is added to old DSP (10), otherwise it is added to DVx computed above. The lex level helps in "long" procedure EXITs and adds information for core and register dumps

compute the current save area address and place into #1C

this is simply the automatic stack frame size (function return value, parameters, local variables) rounded up to even (this constant is preloaded in the BSFC for register 1C) plus either 1) old DSP (10), if XDCL, or 2) DVx, if local

IF this procedure is XDCL THEN

IF there are parameters THEN

copy the frame from #17↑ to (DV[lex]+size(frv))↑

22.0 CYBIL-CS/SS RUN TIME ENVIRONMENT
22.4.3 PROCEDURE PROLOGUE

IFEND

IFEND

this step worries about a call from outside the CYBIL environment where the parameter frame may not have been near the end of the caller's saved registers.

restore other display registers for intermediate lex levels

The values to restore are located at word offsets 6+lex relative to 1D. For each such vector element the instruction sequence would be:

ES #21,1+6

LOD [#1D,#21],#21

where 1 is in the range 1..<current lex>-1

copy in value parameters that were not passed by value

refer to the chart above in the "By Value" column where address is indicated. For adaptable parameters, the storage will be "pushed" into the local stack frame; for fixed parameters, the storage has been pre-allocated.

22.4.4 PROCEDURE EPILOGUE

Both epilogue and post call code conform with the VSOS convention with the extension of needing to interface with non-CYBIL environments relative to function return values. For Epilogue:

IF this procedure is XDCL and size of function return value

is 1 or 2 words THEN

move function return value to register(s) #18 (and #19)

IFEND

this set of instructions anticipates returning to a non-CYBIL

environment such as IMPL which expects the return value(s) in registers #18 and #19

reload (SWAP) the caller's registers using #1D

branch based on register #1A

22.4.5 POST CALL

For post call:

IF call to XREF and size of function return value is 1 or 2 words THEN

move registers #18 (and #19) to operand register area

IFEND

this set of instructions anticipates having returned from a non-CYBIL environment

22.4.6 "LONG" PROCEDURE EXIT

RETURNs and EXITs in the currently active procedure simply generate branches to the epilogue code. EXITs of a globally encompassing procedure by the currently active procedure cause a reloading of the encompassing procedure's environment (SWAPs) prior to generating its epilogue code (see 2.2.5). In high level terms, if the lex level of the proc to exit is dlex, then:

WHILE current lex <> dlex DO

reload registers per #1D

WHILEND

environment has been forced to that for the encompassing procedure

22.0 CYBIL-CS/SS RUN TIME ENVIRONMENT
22.4.6 "LONG" PROCEDURE EXIT

```
do epilogue of procedure
```

The WHILE loop above actually exists in object code. Reducing this to instructions that are more C200-like:

```
ES 10,dlex

SWAP 1D,15,0

LTOR 1B,11

IBXNE,BRB 10,0,2,11,12

<epilogue code>
```

22.4.7 PUSH DYNAMIC SPACE

This function (primitive) is used by both the PUSH storage management statement and the prologue code to find room for adaptable, by value, parameters in the automatic stack frame of the current procedure. Its input is the size of the space needed, its output is the address of the acquired space. The algorithm evens the space required, and adds this increment to both the CSF (1C) and DSP (1B). DSP must of course protect our stack frame from interrupts. The original value of CSF is returned as the pointer to the PUSH'ed space.

22.4.8 EXTERNAL ENVIRONMENT INTERFACE SUMMARY

Problems associated with cross-linkage between CYBIL and non-CYBIL environments derive from linkage mechanics, function return values, and parameter passing (type, method, allocation, and format).

C200 CYBIL will support inter-environment calls when

- The alien environment (AE) observes VSOS conventions for linkage
- The AE observes VSOS convention for functin return value (FRV)
- The AE parameter formats can be successfully described in CYBIL, and implementation dependent mappings, etc. are known to be compatible

22.0 CYBIL-CS/SS RUN TIME ENVIRONMENT
22.4.8 EXTERNAL ENVIRONMENT INTERFACE SUMMARY

Because no CYBIL construct exists to explicitly indicate the environment of a non-local procedure (XREF), or the source of a reference to a local XDCL procedure, C200 CYBIL assumes most such events represent inter-environment linkage. Exceptions are functions whose type is strongly peculiar to CYBIL.

22.4.8.1 CYBIL --> AE Linkage

A CYBIL reference to an XREF procedure is the indicator. The solution of linkage and FRV problems were discussed above. In summary, CYBIL assures:

- o parameter frame pointer #17 is defined to point to params in save area
- o register #1B is rounded even up
- o parameters are stored at PFP↑ (PFP = #17)
- o #18 (and #19) is used as source of FRV

These items may be neglected if within CYBIL environment (CE).

22.4.8.2 AE --> CYBIL Linkage

The indicator is the XDCL proc attribute. In summary, CYBIL assures:

- o display vector entry is set
- o copies params from frame pointer PFP↑ into this proc's auto frame
- o duplicates FRV in #18 and #19

22.4.8.3 Parameter Conformity Considerations

22.4.8.3.1 TYPE

The coder must determine that an intersection exists between the CE types and the AE types. This is commonly implementation dependent.

22.4.8.3.2 METHOD

CYBIL offers passage by value, in which case a copy of the variable is physically passed, and by reference, in which case a pointer to

the actual variable is passed. An additional complication is the by value structure, for which CYBIL passes (or expects) a pointer with the tacit assumption a copy will be made (or is made).

Further, CYBIL interfaces only via a parameter frame in storage. Register resident parameters are not supported.

22.4.8.3.3 ALLOCATION

The user must determine that semantics regarding parameter alignment and mapping, especially size, are compatible. This is very implementation dependent. Watch carefully the semantics regarding objects of pointers.

22.4.8.3.4 FORMAT

Additional (usually implementation dependent) semantics must be determined to be compatible with regard to data format. For example, integer ranges, set member order, value conventions (boolean, NIL), etc.

22.4.8.3.5 EXAMPLE: CYBIL <--> FORTRAN

- All FORTRAN parameters are by reference
- Most intersecting types are conformable in allocation and format

The conformity table is:

CYBIL	FORTRAN	Comments
boolean	No exact equiv	CYBIL logic gives 0..1 acts upon 0,<>0-trouble
integer	integer	
subrange,ordinal	No exact equiv	CYBIL may produce err if given value out range
character	string-length 1	CYBIL passes as ↑s,1=1 receives as ↑s
real	floating point	
longreal	double prec. f/p	Not supported
string	"string"	
set	logical	Careful with semantics
array	array	Depends on elements
cell, pointers, adaptables, records,storage	N/A	

22.4.8.4 Interface to COMMON

CYBIL will likely be used in a hybrid environment with IMPL and FORTRAN. CYBIL has no construct analogous to the COMMON construct of these languages.

C200 CYBIL will cause loader text, compatible with the COMMON construct, to be generated for all CYBIL XDCL variables. This solution results in less efficient code generation for XDCL variable access, and destroys CYBIL semantics regarding static initialization of XDCL variables, and may cause implementation dependent, non-repeating object module/loader performance anomalies.

22.4.9 AN EXAMPLE

In order to demonstrate the pseudo code presented above, consider the following example which has a local CYBIL procedure with parameters and a subsequent call to the procedure:

```
PROCEDURE local(i: integer; VAR s: STRING(*));
  RTOR      #1C,#11          {save CSF
  IS (IX)   #1E,<BSFC offset>
  ELEN      #1E,<PC size>
  ES        #12,0005         {offset in BSFC where 1F will be
                             {swapped in
  STO       [#1E,#12],#1F    {stash DFT so swap will reset
  SWAP      #1E,#15,#1C      {save caller's regs and load PC
  ADDX      #2x,#17,#2x      {set proc's display entry
  ADDX      #1B,#2x,#1B      {set new DSP
  ADDX      #1C,#2x,#1C      {set CSF = new save area
  RTOR      #11,#1D          {restor CSF as PSF

...

 PROCEND local;
  SWAP      #1D,#15,0        {reload caller's registers
  BADF,BR   0,#1A            {return to caller

...

local(5,'abcdefg');
  ES        #C0,5            {load parameter one by value
  RTOR      #sc,#C1          {address of string constant is
                             {in register in PC of caller
  ES        #C2,7            {length for formal adapt
  RTOR      #20,#1E          {move STATIC to DBP
  ES        #10,40(16)*(1st parm reg - 1A(16)) {set up PFP
  ADDX      #1C,#10,#17      {new parm frame pointer
  ELEN      #17,<parm count in words>
  EX        #local,<delta to proc> {address of local in PC
  BSAVE     #1A,#local,#1A         {of caller
```

22.5 VARIABLES

22.5.1 VARIABLE ALIGNMENT

The implementation dependent value for the alignment base for the C200 is eight (8).

22.6 STATEMENTS

22.6.1 CASE STATEMENT

There are 2 alternative code sequences generated for the CASE statement, depending on the characteristics of the case selectors.

The first alternative is a code sequence consisting of a selector range check, followed by an indexed jump into a jump table generated for the CASE statement selector range. This jump table consists of unconditional jumps to the case entry points. The table thus actually resides in the inline executable code section.

The other alternative is a code sequence consisting of a series of conditional branches. Branch implementation is chosen if the number of case selectors is less than three, or, if the difference between smallest and largest case selectors is greater than the number of entries to the third power. If the number of entries is more than 8, the code sequence that is generated is in the form of a binary search.

22.6.2 STRINGREP

22.6.2.1 Pointer Conversions

The default radix for the conversion of a pointer into a string is defined as implementation dependent. For the C200 the resultant string will be the pointer represented in hexadecimal notation.

22.7 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};
```

22.0 CYBIL-CS/SS RUN TIME ENVIRONMENT
22.7 RUN TIME LIBRARY

```
        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 (composite_number: INTEGER;
        module_name_ptr: ↑mod_name);

      where:
        line_count := composite_number DIV 10000(16);
        error_count := composite_number MOD 10000(16);
```

To process calls to a NIL pointer to procedure:

```
    CYP$NIL;
```

To terminate execution gracefully call:

```
    CYP$TERMINATE;
```

22.7.1 RUNTIME ERROR MESSAGES

```
     0...unequal_string_length    1...adaptable_length_error
     2...subscript_error          3...range_error
     4...undefined_case           5...reset_to_error
     14..substring_start_error    15..substring_length_error
```

22.0 CYBIL-CS/SS RUN TIME ENVIRONMENT
22.7.1 RUNTIME ERROR MESSAGES

```
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
```

22.7.2 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.

22.7.3 TRACEBACK CAPABILITY

A traceback capability has been added to the C200 runtime package to facilitate debugging. Each stack frame in the runtime stack that is of CYBIL origin will be analyzed. The data output will include the module and p_address within the module where that procedure will start execution when control is returned to it. The contents of the dynamic space and the active registers at that point will also be output.

Traceback data will be created automatically when any CYBIL runtime error is encountered.

22.0 CYBIL-CS/SS RUN TIME ENVIRONMENT
22.7.3 TRACEBACK CAPABILITY

The name of the traceback file defaults to "TRACE". this can be changed by calling RPVINIT:

```
PROCEDURE [XREF] rpvinit ALIAS 'SW=ERRI' (out: ↑file;
          user_cleanup=proc: ↑procedure);
```

where

OUT has been created by calling PR#OPEN
USER_CLEANUP_PROC is a procedure to be executed after the traceback (normally NIL)

A traceback can be generated at any point by calling RPVTRACE:

```
PROCEDURE [XREF] rpvtrace (str_p: ↑STRING(*) );
```

where

STR_P points to a title line for the traceback.

The tracback analysis does not require that the CYBIL modules be compiled with any DEBUG options set.

There are currently two ways to look at an ASCII file:

Print it:

```
MFQUEUE,TRACE,DD=C8,ST=M10,
      JCS="ROUTE,UJN=zzz,DC=PR,EC=A9,TID=C.".
```

or look at with an editor on the frontend:

```
MFLINK(TRACE,DD=C8,ST=M10,I=SENDTRAC)
(switching to the frontend)
ATTACH(TRACE)
FCOPY(P=TRACE,PC=ASCII8,N=XXX,NC=ASCII)
XEDIT,XXX
```

22.7.4 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 reusable memory. There is no 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.

The following additional criteria were used in designing the HEAP MANAGEMENT MODEL:

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

(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.

22.7.4.1 ALLOCATE

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

22.7.4.1.1 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.

22.7.4.1.2 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.

22.7.4.2 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
```

22.7.4.3 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.

22.7.4.4 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)

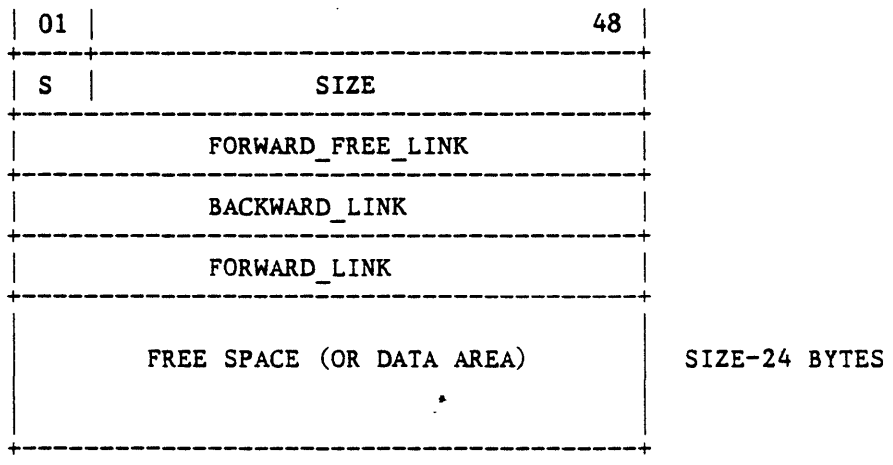
If more space is required than is available in the current memory space, Q5MEMORY is called again to get more space and then block_header (3) of the old space and block_header (1) of the new space are linked together so there is always a path from one static space to the next.

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 actual size requested will be passed along to Q5MEMORY.

22.0 CYBIL-CS/SS RUN TIME ENVIRONMENT
22.7.4.5 HEAP BLOCK_HEADER

22.7.4.5 HEAP BLOCK HEADER

BLOCK_HEADER (CYBIL description in STORAGE TYPES section):



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

22.7.4.6 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.

23.0 PROCEDURE INTERFACE CONVENTIONS

23.0 PROCEDURE INTERFACE CONVENTIONS

23.1 INTRODUCTION

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

23.2 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.

23.3 GENERAL PHILOSPHY

- 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.

- 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.

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

23.0 PROCEDURE INTERFACE CONVENTIONS
23.3 GENERAL PHILOSPHY

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.

23.3.1 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 parameters.

  - If for any reason input parameters are declared as reference parameters, the actual parameters must be moved to local automatic variables prior to validity check and subsequent usage. Further, all input parameters declared as reference parameters 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).

23.3.2 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.

- 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.

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

- 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.

- An ordinal type should support only one concept.

- 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.

- Integer subrange is preferred over INTEGER when declaring numeric parameters. This improves the natural type checking provided by the CYBIL language. 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.

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

- SET type provides a mechanism by which multiple subfunctions or options may be discreetly 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.

- For adaptable STRING formal parameters include the adaptable string bound in the formal parameter definition.

- 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.

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 initialization 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.

  - 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.

- 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.

- 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 not known at compile-time, but is resolved at execution-time). The formal parameter pointer type should ultimately resolve to '↑CELL'.

- 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.

24.0 PROGRAM LIBRARY CONVENTIONS

24.1 DECK NAMING CONVENTIONS

For a detailed discussion of the deck naming conventions the reader is referred to the C180 System Interface Standard. The section titled Systemwide Conventions has considerable discussion of naming conventions, product identifiers and classes of names.

When converting fron the C170 source management facilities to the C180 source code utility (SCU) all XREF declarations, documentation headers and module decks can be renamed. The new deck name can be expanded to a full name (up to 31 characters) of the item contained in the deck.

24.2 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 or FUNCTION declarations which may be expanded INLINE as part of calling modules; as opposed to being called through an XDCL/XREF interface. INLINE procedures OR functions 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 must be documented accordingly. A PROCEDURE or FUNCTION implemented in this fashion must not be dependent on the static link, i.e. it must be completely self-contained.

24.3 COMMON DECK CONTENT

24.3.1 PROGRAM INTERFACE DOCUMENTATION HEADER

24.3.1.1 Procedures and Functions

The PROCEDURE or FUNCTION 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 2: 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.

24.3.1.2 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.

24.3.2 XREF DECLARATION COMMON DECK

The XREF declaration common deck contains a CYBIL XREF declaration followed by a *copyc to all of the TYPE or CONST declaration common 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.

C170 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 ??
```

24.3.3 TYPE / CONST DECLARATION COMMON DECK

The TYPE / CONST declaration common deck contains CYBIL TYPE and/or CONST declarations followed by a *copyc 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 *copyc's (instead of *copy) to common decks. This prevents duplicate

definitions of identifiers in the caller's CYBIL module.

C180 Example:

```
AMT$LOCAL_FILE_NAME

TYPE
  amt$local_file_name = ost$name;

*copyc ost$name
```

24.3.4 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.

25.0 CYBIL CODING CONVENTIONS

25.0 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. Code should have appropriate commentary, i.e. comments that indicate the function being performed by the line(s) of code and not what the statement performs. Don't just echo the code with comments; make every comment count.

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) provides information not readily apparent from reading the code, b) is of a sufficiently algorithmic nature, c) is at a consistent level of detail, and d) when discussing variables, uses the actual variable name.

25.0 CYBIL CODING CONVENTIONS
25.1 USAGE OF A SOURCE CODE FORMATTER

25.1 USAGE OF A SOURCE CODE FORMATTER

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

25.2 USE OF CYBIL

1. Use the straight forward features of the language.

2. Use library routines and built-in functions where ever possible.

3. Keep portability in mind, avoid machine dependent code.

4. Choose a data representation that makes the program simple.

5. Use block structure to articulate program structure: a declaration should always be declared at the "lowest" (or most local) level possible.

6. Avoid the use of static variables and the static link. In general a procedure should only reference its own arguments and its own local variables.

7. In general, interfaces between modules should be procedures or functions, not XDCL/XREF variables. This will lower coupling between separate modules and make it easier to modify one module independently of another.

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

9. 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/;
```

10. In general avoid the use of type INTEGER; few variables require subranges that large. Using proper subranges improves the integrity of your program.

11. In declarations of procedure parameter lists, always separate each formal parameter with a semicolon marking each with a VAR (for

reference parameters) or "absence of VAR" (for value parameters) as appropriate.

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

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

14. It should be clear why the branch points in the code exist. This is best done through proper use of meaningful names, or if that is not possible, through documentation. This includes ELSE & ELSEIF clauses.

15. Avoid unnecessary branch points.

16. 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.

17. Use recursive procedures for recursively-defined data structures.

18. 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.

19. The comments should precede the code which it describes.

20. Don't comment bad code; rewrite it.

21. Trailing comment delimiter of '}' should be used only when required: i.e., usage of end-of-line as a comment delimiter is encouraged.

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

23. If a logical expression is hard to understand, transforming it will often make it simplier to understand.

24. Make exhaustive decisions (i.e. IF statement expressions) and order for machine efficiency.

25. Avoid literal constants, it is better to quote the literal within

the CONSTant declaration and then reference it symbolically.

26. Don't use a variable for more than one purpose.

27. Avoid language tricks. Don't get sucked into the "microefficiency" syndrome. The simplier code is often the more efficient. After the system works, time critical modules can be optimized as necessary.

28. Avoid using the #LOC function as it can have a detrimental effect on code generation efficiency and maintainability.

29. Limit the use of the CYCLE, EXIT and RETURN statements as it makes the code harder to understand for subsequent readers.

25.3 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:

1. 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';
```

2. Meaningful module, procedure, function and variable names should be used.

3. Similar variable names should not be used as they will cause confusion.

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

25.0 CYBIL CODING CONVENTIONS
25.3 USE OF THE ENGLISH LANGUAGE

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

```
/search_symbol_table/ {instead of}
/11/
```

6. Labels are a powerful documentary aide and their usage is encouraged. However, they are not an excuse for writing sloppy, unstructured code.

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

```
IF file_is_open THEN {instead of}

IF file_switch THEN
```

25.4 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 meaningful English. If it is impractical to describe the full meaning of a variable through a meaningful name, then a descriptive comment is required.

25.5 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.
```

25.0 CYBIL CODING CONVENTIONS
25.5 MODULE AND PROCEDURE DOCUMENTATION

```
{  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.
{  NOTES:
{    This should contain information of interest to the
{    user or maintainer. Including items such as: exceptions,
{    special cases, assumptions, dependencies, related documents,
{    etc.

<procedure or function declaration>;

{  PURPOSE:
{    This should describe the process the procedure or
{    function performs rather than the method used.
{  DESIGN:
{    This should contain an overview of the procedures design;
{    i.e., a description of how it works in specific terms.
{    Usage of specific variables or procedure names is discouraged
{    in this description.
{  NOTES:
{    This should contain information of interest to the
{    user or maintainer. Including items such as: exceptions,
{    special cases, assumptions, dependencies, related documents,
{    exit conditions, etc.
{  GLOBAL VARIABLES:
{    This should contain a list of the global variables which are
{    referenced by the procedure.
```

25.6 TITLE PRAGMATS

Using the TITLE pragmat can make the code more readable and speed the finding of certain functions with a compilation unit.

Each module should be titled in the following way:

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

for example:

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

25.7 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.

25.8 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.

25.9 CYBIL CODE INSPECTION CHECKLIST

When doing code inspections on CYBIL source code the following checklist may prove to be helpful. Comments, suggestions and additions to this inspection checklist are welcome and should be sent to Harvey Wohlwend - ARH280.

25.9.1 GENERAL GUIDELINES

	Is the function being performed expressed in plain language in the implemented code?
	Is the product presented in an overall neat appearance?
	Have structured programming techniques been used?
	Does each CASE statement have all variants covered or an ELSE quoted?
	Has the product been formatted with the CYBIL source code formatter?
	Does the product match the CYBIL coding conventions?
	Does the implementation conform to the necessary standards/conventions? (ANSI, ISO, SIS, etc.)
	Are the module and procedure documentation blocks completed?
	Are comments useful or are they simply alibis for poor coding?
	Do the comments and the code agree?
	Are meaningful names used that won't be confused?

25.0 CYBIL CODING CONVENTIONS
25.9.1 GENERAL GUIDELINES

	Have any constraints been listed in the NOTES (as needed)?
	Has the report from the Analyze_CYBIL_Complexity tool been reviewed? If the Discriminant value is HIGH consider reimplementing.
	Has any reusable code been identified & delivered to the Tools Organization?
	Is code from the reusable code library used wherever possible?
	Are there repeated code segments, whether within or between blocks?
	Is there a one-to-one relationship between modules indicated in the design & modules implemented?
	Is there a one-to-one relationship between intermodular connections indicated in the design and intermodular references implemented?
	Is everything in the design implemented?
	Is there anything in the code that is not in the design?
	Has this been built to be testable?

25.9.2 ALGORITHM VERIFICATION

- [ ] Are all variables initialized before use?
- [ ] Any variables or parameters which are not used?
- [ ] Is the branch taken the correct way on equality tests?
- [ ] Any unreachable code segments?
- [ ] Does every data structure result in a control statement?

25.9.3 MODULE DOCUMENTATION

- [ ] Are the module documentation guidelines followed?

25.9.4 PROCEDURE OR FUNCTION DOCUMENTATION

- [ ] Are the procedure documentation guidelines followed?
- [ ] Are the exit conditions listed in the DESIGN description?
- [ ] Does the documentation precede the code?
- [ ] Any obscure code annotated?

26.0 EFFICIENCIES

26.0 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.

26.1 GENERAL CONSIDERATIONS

- Make it right before you make it faster.
- Make it fail-safe before you make it faster.
- Make it clear before you make it faster.
- Keep it simple to make it faster.
- Don't diddle code to make it faster; find a better algorithm.
- It is best to measure your program before making "efficiency" changes.
- Be sure to check the default values for compile options, use the correct values for your situation. For performance runs select optimization when possible. Avoid selecting stylized code for debug since it will generate extra code and slow the program's execution. Also the presence of range checking code can be expensive and the presence of debug & line tables for debugging purposes will slow the loading process.

26.2 SOURCE LEVEL EFFICIENCIES

26.2.1 GENERAL

- Range checking code requires additional storage space and is time consuming. One can eliminate all generated range checking code by setting "RC=NONE" or "CHK=0" on the call statement (or ??SET(CHKRNG:=OFF)?? in 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.

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

- There is a significant amount of overhead associated with any procedure call. If a procedure is being called in a looping construct, it may pay to call the procedure once and put the loop tests inside the called procedure.

- References to variables via the static chain in nested procedures cause an overhead associated with that reference. In general, a procedure should only reference static variables, arguments and its own automatic variables.

- Assignment of records is done with one large move, while record comparison is done field by field. Therefore, all other things being equal, it is best for performance reasons, to organize fields within records with the most likely non-conforming fields first.

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

26.0 EFFICIENCIES
26.2.1 GENERAL

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

- Quoting a constant for the <lower bound spec> on an adaptable array will result in more efficient object code. The most efficient <lower bound spec> is 0 (zero).

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

- Repeated references to complex* data structured (via pointers or indexing operations) may, in certain circumstances, be made more efficient by pointing a local pointer at the structure and use it to replace the complex references.

- For adaptable strings include the adaptable string bound whenever possible.

- 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 homogeneous 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

* don't use 'complex', use 'unstructured'/'simplified' instead to avoid confusion

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.

- For efficiency and maintainability reasons the use of #LOC should be avoided.

- When a definition contains a number of 'flags' or attributes, the following should be considered when choosing 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 conditional 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

26.0 EFFICIENCIES
26.2.1 GENERAL

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:

  important when using named constants to recognize name as constant

  ```
  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).

- When doing divide by a power of two on a positive integer subrange a shift instruction can be generated. Because a shift instruction tends to be considerably faster than a divide instruction it is a benefit to define positive integer subranges.

- Certain conversion functions (i.e. $INTEGER, $CHAR, etc.) require no execution time overhead.

- The code generated for STRINGREP is a call to a run time library routine. Using STRINGREP to concatenate substrings is very inefficient, when substring assignment will fulfill the same purpose.

- 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.

26.2.2 CC EFFICIENCIES

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

- 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.

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

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

  Otherwise the faster inline code is generated.

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

26.2.3 CI/II EFFICIENCIES

- Use ANALYZE_PROGRAM_DYNAMICS and MEASURE_PROGRAM_EXECUTION to do a detailed study of your program's efficiency with respect to execution, page faults and module connectivity. Both are described in the SCL Object Code Management manual.

- Programs should be bound to improve the overall load and execution time. This can be done via ANALYZE_PROGRAM_DYNAMICS, MEASURE_PROGRAM_EXECUTION utility, or either of the Object Library Generator subcommands BIND_MODULE or CREATE_MODULE.

- For large programs with a great deal of static data, prelinking the program may be helpful.

- The NOS/VE command DISPLAY_SYSTEM_DATA can be used to gather overall system information during your program execution during block time.

- There is also a statistic facility that can be used to assist in data collection and reporting. The ACTIVATE_LOCAL_STATISTICS command, the ACTIVATE_STATISTICS command and the DISPLAY_BINARY_LOG utility are the interesting functions here. They are described in the System Performance & Maintenance Manual.

- The adaptable string bound construct should be quoted whenever possible to give the compiler a clue as to the maximum length.

26.0 EFFICIENCIES
26.2.3 CI/II EFFICIENCIES

This will often result in more efficient code being generated for adaptable strings.

- References to XDCL variables and variables declared within a SECTION will be made via the binding section and, consequently, an overhead is associated with the first reference.
- The code generator does not currently 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.
- The reach of the load & store instructions on the C180 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.
- For additional discussion of performance the reader is encouraged to reference the NOS/VE Analysis usage manual.

26.2.4 CM/IM, CU/IU & CA/AA EFFICIENCIES

- Subranges should never be any larger than necessary. Having subranges larger than necessary negates CYBIL's range checking, and generally causes less efficient code. In particular, definitions of symbols such as int16 and uint16, which define 16-bit signed or 16-bit unsigned integer subranges should be avoided. These tend to cause register extend operations when used in arithmetic operations. For example, if 2 16-bit values are added, the values must be extended to 32 bits before adding them in order to guarantee a correct result.

26.2.5 CP EFFICIENCIES

- The UCSD p-system does not have an exclusive or instruction. Therefore, set references using the XOR operator generates a lot of code.
- Using long integer subranges results in less efficient code.
- 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. Consequently large structures (i.e. arrays) should be the last variables in the list.

- Using global variables in other modules should be avoided.

- Avoid FOR statements in favor of WHILE or REPEAT. They are faster and produce less code.

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

26.2.6 CS/SS EFFICIENCIES

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

26.3 COMPILATION EFFICIENCIES

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

- Compiling source code with the sequence numbers provided by the source management system slows the compilation process.

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

- The generation of stylized code slows the compilation process.

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

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

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

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

27.0 IMPLEMENTATION LIMITATIONS

27.1 GENERAL

- Maximum number of lines in a single compilation unit is 65535.
- Maximum number of unique identifiers allowed in a single compilation unit is 16383.
- Maximum number of user defined procedures in a single compilation unit is 999.
- Procedures can only be nested 255 levels deep.
- Maximum number of compile time variables used in conditional compilations is limited to 1023.
- Maximum number of error messages printed per module is 2000.
- Maximum number of elements defined in a single ordinal list is limited to 16384.
- Integer constants are restricted to 48 bits on the C170.
- Nested calls to inline routines (procedures or functions) are limited to 5 levels of nesting.
- Maximum number of stacked(PUSHed), toggle control pragmats is 25.
- Nested calls to INLINE routines are limited to 5 levels of nesting.

27.2 CC LIMITATIONS

- Case selector values limited to less than 2**17.
- Pointer fields within initialized packed records must be aligned for use within C170 capsules or overlay capsules.
- Packed arrays whose element size exceeds 2**17 bits gets a subscript range error.

27.3 CI/II LIMITATIONS

- Maximum number of lines in a single compilation unit is 32767 when run time error checking is selected.
- Nesting level of structured statements is limited to 63 levels deep.
- FOR statements can only be nested 15 levels deep.
- The statement FOR I := j DOWNTO LOWERVALUE (INTEGER) (or its equivalent) will result in an arithmetic overflow if the proper hardware traps are enabled.
- Procedures may only be nested 50 levels deep.
- 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.
- Long constants are not included in the debug symbol tables produced.

27.4 CM/IM, CU/IU & CA/AA LIMITATIONS

- Maximum number of lines in a single compilation unit is 32767 when run time error checking is selected.
- Nesting level of structured statements is limited to 63 levels deep.
- FOR statements can only be nested 15 levels deep.
- Procedures may only be nested 50 levels deep.
- 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 a module should be appropriately controlled.

27.0 IMPLEMENTATION LIMITATIONS
27.4 CM/IM, CU/IU & CA/AA LIMITATIONS

- The stack frame size is limited to 2**15 bytes.

27.5 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.
  - As the control variable, starting value or ending value of a FOR statement.
- The result of a Stringrep operation on a floating point number is limited to 6 digits.

27.6 CS/SS LIMITATIONS

- Array size limited to 2**32.

28.0 COMPILER AND SPECIFICATION DEVIATIONS

28.0 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.

28.1 GENERAL CYBIL IMPLEMENTATION DEVIATIONS

- Support adaptable arrays of zero dimension.
- Double Precision Floating Point (LONGREAL).
- RESET TO with a relative pointer.
- STRLENGTH of constant identifier.

28.2 CC DEVIATIONS

- NIL Relative Pointer value.
- #SPOIL intrinsic.
- #SEQ function.
- #SIZE of adaptable types.
- Run time checking on accessing fields of variant records is not supported.
- Relative Pointer Types.
- General Intrinsics.
- Partial condition evaluation on OR operator not supported.
- Library pragmat.
- Actual value parameters > 1 word must be addressable.

28.3 CI/II DEVIATIONS

- Run time checking on accessing fields of variant records is not supported.
- If a non-local exit from an UNSAFE function is done, the function result value is always undefined.

28.4 CM/IM & CU/IU DEVIATIONS

- NIL Relative Pointer value.
- #SPOIL intrinsic.
- #SEQ function.
- Run time checking on accessing fields of variant records is not supported.
- Library pragmat.
- Single Precision Floating Point (REAL).

28.0 COMPILER AND SPECIFICATION DEVIATIONS
28.5 CA/AA DEVIATIONS

28.5 CA/AA DEVIATIONS

- NIL Relative Pointer value.
- #SPOIL intrinsic.
- #SEQ function.
- Run time checking on accessing fields of variant records is not supported.
- If a non-local exit from an UNSAFE function is done, the function result value is always undefined.
- Library pragmat.

28.6 CP DEVIATIONS

- NIL Relative Pointer value.
- #SPOIL intrinsic.
- Static initialization.
- PUSH statement is not supported.
- #SEQ function.
- #SIZE of adaptable types.
- Run time checking on accessing fields of variant records is not supported.
- Relative Pointers.
- General Intrinsics.
- Library pragmat.
- Copies of adaptable value parameters are never made.

28.7 CS/SS DEVIATIONS

- NIL Relative Pointer value.
- Library pragmat.