CYBER 180

System Interface Standard

by

Sunnyvale Product Design and Advanced Systems Design

Feb 3, 1986 (updated)

1.0 GENERAL

1.1 PREFACE TO CURRENT EDITION (SEE COVER SHEET FOR DATE)

For hardcopy of the SIS, do (SN452/125 in SVL, SN302 in ARH):
acquire,s2196/un=dcs
ses.print s2196
For copy on 8.5" x 11" inch paper (in SVL), add print parameter fc=sp.
(You may have to wait until evening for copy.)

For current status of pending SIS daps, see file SISDAPS/pwo0336 on SN452.

AD&C/BCCB approved SIS daps incorporated in this revision are:

S5036 Change the Key parameter (Wilson)
S5048 Keypoint Ranges (Mages)
S5059 Statistics (Neuhaus)
S5060 Product identifiers for CDCNET: DC & NP (Rundquist)
S5062 Product identifier for Distributed Files: DF (Sprandel)
S5067 Passing BIT data type parameters (Barney)

In addition, the following pending SIS daPs are conditionally included:

S5102 Product identifier for Concurrent Maintenance Utilities: CU (Redig)
S5103 Product identifier for CYBIL Formatter: CF (Wachutka)

The following notes will enable you to look at updated sections of the SIS without printing the entire document:

For S5036, see section 2.2.4.3, change in KEY prameter.
For S5059, see section 3.5.4 Statistics.
For Product identifiers, see section 4.1.1.1.
For S5048, see section 4.7.2.1 Operating System Keypoint ranges.
For S5067, see sections 5.2.1.1 and 5.2.5.2 (including its subsections).

1.2 CHARTER

1.2.1 PURPOSE

The purpose of this standard is to ensure a uniformity across the operating system and product set that will make the total system more easily usable and human engineered.

1.2.2 SCOPE

This standard covers the software system which includes both the operating system and the product set. The standard covers product-to-product, product-to-user, operating system-to-user, and product-to-operating system interfaces. These interfaces may be documented in the NOS/VE and product

ERSs. This System Interface Standard is the controlling document for all such interfaces.

Any external interface which is not defined by an industry standard may be defined in this System Interface Standard. In order to achieve a uniformity across the product set, certain internal interfaces shall be included in this standard, e.g. calling sequences.

With respect to command-level calling sequences, parameters, and options, this standard includes (see section 2.2.4) all options for all parameters of all calling sequences except those of NOS/VE which are documented in the NOS/VE ERS.

(Among options of parameters, there may be inconsistencies from product to product. Such inconsistencies will be removed in the future by daps against the SIS and code changes, or noted as exceptions - see item 5 of section 1.2.4 below. At this time, it is more important to document existing usages in order to.sake the SIS complete than it is to remove inconsistencies and conflicting usages.)

Interfaces in code that do not conform to the SIS, either by commission or omission, should be PSRed. The project-defined priority of such PSRs may not be less than serious, and the such PSRs will not be rejected by projects.

1.2.3 GOALS

The specific goals of the System Interface Standard are:

a. Consistency within and across the system.

b. Human engineered for user.

c. Achievable within CYBER 180 timeframe.

d. Good performance.

e. External interfaces like CY170 where this does not conflict with a, b, c and d above.

There must be more than trivial gain in aspects of human engineering to cause deviation from CY170 external interfaces.

1.2.4 REVIEWING AND UPDATING THIS DOCUMENT

The C180 SIS has been through a number of review cycles and has been formally approved by the C180 Baseline Change Control Board (BCCB). It is thus considered fairly solid.

However, it is recognized that the SIS is a living document with a

continual need for updating. Please follow the following guidelines in reviewing and updating this document:

1. Limit comments or updates to question of inaccuracy, lack of completeness, or necessary technical change. Avoid questions of personal preference.

2. For relatively minor problems or questions resulting from a normal review, a normal DCS comment is appropriate. It is the responsibility of the appropriate author(s) to resolve the comment.

3. For more major updates that may be somewhat controversial, a stand-alone DAP is appropriate. This allows a thorough review of the issues involved. When approved, the DAP will be included in the next SIS update. The SIS referee or editor should be informed of any plans to submit such a DAP and the DAP should be in the form of a proposed SIS update.

4. There will be occassional "minor review cycles" of the SIS to incorporate minor changes and previously approved DAPs. Authors may make minor changes to their sections at this time for review and approval.

5. If conflicts exist between two products that cannot be resolved or change is impractical, the exception will be documented in the SIS. The number of exceptions is expected to be very small.

2.0 INPUT

This section describes the standard and conventions for input to products. Input standard is defined for System Command Language, Control Statement, source file organization and contents.

2.1 SYSTEM_COMMAND_LANGUAGE

The System Command Language is the set of language rules and conventions to be followed by any software product that presents a user interface (which is not defined by an industry standard). It is documented in the NOS/VE ERS (DCS documents ARH3609, ARH3610). For example, commands to call products, and operator commands will conform to this language definition. It is a requirement that all products use the standard command language routines to process system command language statements (such as product call commands or product directives). The intent here is that products do not duplicate code or functions already provided by standard command language routines. See NOS/VE ERS (ARH3610) for a description of these routines.

2.2 PRODUCT_CALL_COMMANDS

This standard specifies the parameters which can be used in commands that call CYBER 180 products. The syntax of the command is documented in the NOS/VE ERS.

2.2.1 APPLICABILITY

This section specifies all parameter names, descriptions and defaults of parameters on a command that calls a product. Requirements for use of the parameters are:

- If a product offers a capability which is the same as one defined in this standard, then the specification in this standard must be used.

- A product is not permitted to use a parameter defined by the standard for a purpose other than that specified by the standard.

- A product need not implement all the parameters or all the parts of a parameter in this standard.

- New parameter names or options must first be approved

as additions to this standard.

- If a product provides a function described by a parameter in this standard, the described parameter name and its standard aliases must be supported by the product as a minimum.

- ALIASES

  A. Standard aliases are made up of the first letters of the parameter name. All products which use the parameter must support the standard aliases.

  B. Aliases which do not conform to the first letter rule, but which have widespread usage, can be standard aliases only if explicitly documented as such in section 2.2.4.3 (Parameter Names and Descriptions) of the SIS.

  C. Non-standard aliases are those aliases which do not conform to the first-letter rule, but which are used for compatibility with older versions of a NOS/VE product. New products should not support non-standard aliases. Older products may want to phase out their non-standard aliases.

  D. 170 compatible aliases are those aliases which do not conform to the first-letter rule, but which are used for compatibility with a 170 product. Products which are not required to be compatible with a 170 product should not use these aliases.

Some guidelines for proposing new parameter names and/or options are:

1. Use a new option of an existing parameter rather than a new parameter name if the capability is an extension of an already defined parameter (example: use D=DS instead of inventing a new parameter DS for debug statements.

2. For related parameters, use aliases that emphasize the relationship (example: LO to relate listing options to the list file, L).

2.2.2 TERMINOLOGY

Default: The value used for a parameter when the parameter does not appear in a command. Section 4.3 on installation parameters indicates which parameter defaults are installation changeable. The defaults specified in section 2.2.4.2 are those expected to be most often used.

2.2.3 SYNTAX

The syntax of the command is defined in the NOS/VE ERS.

If a parameter is omitted, default values are used. Use of <parameter name = OFF> results in turning off a single option parameter or boolean single specified value parameter. Use of <parameter name> = NONE indicates that a specified value is not supplied for a multiple value or multiple option parameter (for example, LO = NONE causes none of the list options to be selected).

When the parameter value is a file name, the file name $NULL should be used to negate that file (for example, B=$NULL causes the product not to produce a binary object code file). $NULL is a reserved file name. A read will respond with an end-of-information. $NULL is an infinite sink for writes.

The following algorithm is applied to parameters:

1. Initially, all value options for this parameter are considered deselected (i.e. there are no initial values).

2. Only the option(s) specified in the value list are then selected.

The <name> used on the command to call a product can be either an alias or a long form as follows:

Alias	Long Form	Description
APL		a programming language
BASIC		beginner's all-purpose symbolic instruction code
CC		The language C
COBOL		common business oriented language

CYBIL		Cyber implementation language
EDIF	EDIT_FILE	Edit Screen (for raw text)
EDIL	EDIT_LIBRARY	Edit Screen (for Source Code Utility libraries)
FMU		file management utility
FTN	FORTRAN	formula translation
LISP		list processor
MAP		Matrix Algorithm Processor
MERGE		merge
PASCAL		Pascal
PROLOG		Programming in Logic
PLI		programming language I
QU		query update
SCU		source code utility
SORT		sort
VX		UNIX system emulator

2.2.4 PARAMETER

Occurrence of any parameter more than once in a control statement is an error.

2.2.4.1 Positional_Ordering_of_Product_Set_Parameters

Product set members providing the I, B, and L parameters must support the following positional ordering on a non-keyword call. There is no guarenteed common ordering of other parameters to a product set member except what might be documented in the reference manual for that product.

1. INPUT

2. BINARY (normally the main desired output of a compiler)

3. LIST

2.2.4.2 Types of Parameters

See the Command Interface (Part I) of the NOS/VE ERS for a description of the file reference, which is the syntax to be used for specifying a file name as a parameter value. If no position is specified, the product will reposition the file before use as follows:

a) for a file named $INPUT, no repositioning will take place if the file is at beginning of information, at end of information, or at a partition boundary. Otherwise, it will be repositioned to end of partition before use.

b) for a file named $OUTPUT, the product will do no repositioning before use.

c) for all other files, the products will reposition to beginning of information before use.

Example: If a call to SCU has been made to write three source decks to COMPILE (the first FTN, the second CYBIL, the third FTN) and they are to be compiled with the object code placed on file LGO, the $ASIS positioning must be specified on the second and third compilations since default positioning is rewind.

```
FTN    I=COMPILE

CYBIL  I=COMPILE.$ASIS,B=LGO.$ASIS,L=$OUTPUT

FTN    I=COMPILE.$ASIS,B=LGO.$ASIS,L=$OUTPUT
```

There are four kinds of parameters:

(1) Single Specified Value

This is a parameter for which the user must specify a value, such as a file reference or a boolean as in the form:

```
Keyword = <boolean>
where:
<boolean> :: = <true> ! <false>
<true> :: = TRUE ! YES ! ON
<false> :: = FALSE ! NO ! OFF
```

For the sake of consistency the values ON and OFF will be used in this document. Products may choose any of the values for <true> and <false> desired and describe the choices as such in the product documentation. The operating system will accept the values for <true> and for <false> equivalently when the standard command language routines for the control statement processing are used. As a result, users will be able to enter any of the values for <true> or for <false> without regard for what values a product has chosen to document.

(2) Multiple Specified Value

This is a parameter for which more than one value (such as file references) may be specified. The form <parameter-name = NONE> will be used to indicate that none of the available options for a parameter are desired.

(3) Single Option

This is a parameter for which the user specifies

<option> = ON

(4) Multiple Option

This is a parameter for which the user may specify the names of more than one option.

For multiple specified value parameters the value list syntax is as described in the NOS 180 ERS, Part I section "Parameter Lists and Types". A value list consists of a series of value sets separated by one or more spaces or by a single comma. When more than one value set is specified, the list must be enclosed in parentheses. A value set consists of a series of values separated by one or more spaces or by a single comma. When more than one value is specified the set must be enclosed in parentheses. The rule is that an outermost pair of parentheses belong to a value list and inner pairs of parentheses belong to value set.

The form <parameter name = NONE> will be used to indicate that none of available options for a parameter are desired.

2.2.4.3 Parameter Names and Descriptions

The parameters are described in alphabetical order.

Parameter Name	Standard Alias	Parameter Description
AUDIT	A	AUD is a non-standard alias for AUDIT. This parameter is used to indicate that the product is being run for audit testing. The parameter causes the selection of any other parameters which may be needed for audit testing as well as selecting the method of processing, which may differ from normal processing. For example, in COBOL the list of items might include the mode where displays of numeric items would not be edited. Single option parameter. Default: the option is not selected. AUDIT = ON selects this option.
BINARY	B	BINARY_OBJECT and BO are non-standard aliases for BINARY. Binary Object code output file. This parameter specifies the file to contain the object code or text produced by a compiler or assembler. B = <file> B=$NULL indicates that no such binary object code output file is to be written. Single specified value parameter, default = $LOCAL.LGO If a list of files is specified for INPUT, then all the binary outputs accumulate on the specified BINARY file.
COLLATING_SEQUENCE_X	CSA CSN CSR CSS	SEQY is a 170 compatibility alias. Collating sequence (X = Name, Step, Remainder, or Alter; and Y=N, S, R, or A). The parameters SEQN, SEQS, SEQR and SEQA control definitions of collating sequences for an applicable product.

CSN. The CSN parameter signals the start of a collating sequence definition. The definition of one collating sequence continues with CSS, CSR, CSA parameters; it is terminated by any parameter not one CSS, CSR, CSA. The form is:

CSN = <name>, where name is the name of the collating sequence.

CSS. Each CSS parameter specifies either a single step or a range of steps. The form is:

CSS = <value-list>, where the expressions in the value list are character expressions.

CSR. This parameter specifies all characters in the character set not specified in a CSR parameter, explicitly or implicitly. The form is:

CSR = ON

CSA. This parameter may be specified to alter all equated characters in output records so they become the first character in the appropriate CSn parameter. The form is:

CSA = ON.

COMPILE | C | Compile file.

This parameter specifies the output file on which compiler source statements are written. Examples are: the output produced by a conversion aid utility; the updated source output by the source maintenance utility for input to an assembler or compiler.

Single specified value parameter, default = COMPILE.

COMPILATION_ | CD | If selected, compilation directives (see

DIRECTIVES		SIS section 2.4) will be recognized. Otherwise compilation directives will not be recognized--if directives are expressed as a special form of comment they will be treated as are all other comments. Single option parameter. Default = ON, directives are recognized.
C170_COMPATIBLE	CC	If selected, all possible CYBER 170 to CYBER 180 product differences will be converted to the CY180 version or diagnosed with messages. For example, in COBOL items specified as COMP-4 will be assumed to be COMP. All products which support this parameter must provide a list of such conversions or assumptions in their manuals. Single option parameter. Default: the option is not selected. C170_COMPATIBLE = ON selects the option.
DEBUG_AIDS	DA	Debugging aids. This parameter specifies the debug options to be selected. All products need not support all options. Multiple options may be specified. The defined options are: 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. Debugging statements usually specify debug actions for the module containing them. See also section 2.4.7 of this standard. DT DEBUG TABLES. Generate line number and symbol tables as part of the object code. OC Object code regardless. Produce object code, regardless of errors in

the source and severity of such errors. For compilers, execution of a line containing a fatal error should result in a call to an object time routine which will terminate the execution with a message. (See section 3.4 for error status returned.) Products with no object time library may generate a zero (program error) instructon for lines in error.

PC Parameter checking. Generate parameter checking information as part of the object code. If PC is specified, any compiler which supports parameter checking will generate actual and formal parameter dscription information in the object code to enable load-time detection of parameter mismatches.

TR Flow tracing. Activate trail pragmats in the source program. Unless TR is specified, trace pragmats have no effect.

Multiple option parameter. The default is DA=NONE

DEFAULT_COLLATION DC This parameter specifies the weight table to be used for the evaluation of character (string) relational expressions and to be used by intrinsic functions which are collated sequence dependent (for example CHAR and ICHAR in FORTRAN). The defined options are:

U or USER
A user specified weight table is used. In FORTRAN a collection of user callable procedures is provided for manipulating the user weight table.

F or FIXED
A fixed (unmodifiable) processor specified weight table is used.

Single specified value parameter, default = FIXED.

DIRECTIVES_FILE DF DIRECTIVES is a non-standard alias.
DIR is a non-standard and a 170 compatible alias.

Additional parameters will be read from this file after all of the control statement parameters have been read.

DF=file-name
Parameters will be read from file, file-name.

DF=(file-name1[,file-name2] . . .)
Parameters will be read from the files in the order that they are named.

Multiple specified value parameter, default = NO ADDITIONAL PARAMETERS ARE READ.

ERROR E Error File.

This parameter specifies the name of the file to receive error listing information. In the event of an error (of EL specified severity or higher) the diagnostic is written to the E file. It is highly recommended (though not required) that a product also output the offending source line or lines to the E file in conjunction with the diagnostic. If there is a listing file (see L parameter) the error line and diagnostic are also written to the L file. If the file name of the E file is the same as the file name of the L file, then the error line and diagnostic are not written twice.

Single specified value parameter, default = $ERRORS

ERROR_LEVEL EL Error Level.

This option indicates the severity level of diagnostics to be printed on the user's listing. The levels are ordered by increasing severity. Specification of

a particular level selects that level and all more severe levels. Products will be allowed some flexibility in specifying the kinds of diagnostics that fall in each of the four categories: informational, warning, fatal, and catastrophic. The following descriptions are provided as a guide. The levels in increasing order of severity are:

I Informational. This is an informational message used to flag a suspicious usage. The syntax is correct but the usage is questionable. For 170 compatibility only, products are free to use 'T' in addition to 'I'. (However, if only one is used, it must be I.) Output must always be 'I', never 'T'.

W Warning. This is a diagnostic where the syntax is incorrect but the product has made an assumption (such as adding a comma) and continued. Messages indicating attempts at error recovery are at this level. Diagnostics of W level should be errors that the user can avoid by program modification.

F Fatal. This is a diagnostic which prevents the product from processing the statement in which it occurs. Unresolvable semantic errors also fall into this class. Such errors may not relate to a specific statement in the program unit. Errors of type 'ERROR' will be treated as equivalent to 'FATAL'.

C Catastrophic. This class of error is fatal to continued processing. The product is unable to continue work on the current program unit. However, it should still advance to the end of the current program unit and attempt to process a subsequent unit (if the product specificaton allows multiple program units in a compilation).

Single specified value parameter, default = W.

EL=NONE causes no errors to be listed.

ESTIMATED_NUMBER_ RECORDS	ENR	Estimated Number of Records. This parameter specifies the estimated number of records to be processed by a product. For example, SORT can use it to cause selection of efficient modes of processing. Single specified value parameter, default = 80000/MRL.
EXCEPTION_ RECORDS_ FILE	ERF	This is a file containing exception information. Products will be allowed flexibility in defining its contents. For example, SORT MERGE will use it for out-of-order merge input records. Single specified value parameter, default is product dependent.
EXPRESSION_ EVALUATION	EE	The options of this parameter control the style of code generated for the evaluation of source expressions. Note that the processing controlled by this parameter is separate from that controlled by the optimization level parameter, but may affect the extent to which optimization is possible. The defined options are: C or canonical The code generated to evaluate an expression will mirror the expression interpretation rules as defined in the product specification. For FORTRAN this would be section 6 of the ANSI standard. This option also serves to inhibit the CCG "regroup" option. ME or maintain_exceptions Inhibit code optimizations which eliminate instructions that might cause hardware exceptions at execution time. This option also serves to inhibit the CCG "unsafe_to_safe" option.

MP or maintain_precision
Inhibit code optimizations which change a floating point operation to a new form that is mathematically equivalent but not computationally equivalent. This option also serves to select the CCG "maintain_precision" option.

R or reference
Intrinsic functions (e.g. those defined in CMML) for which a procedure call is generated will be called by reference rather than by value.

Multiple option parameter.
Default = NONE, none of the options is selected.

EXTERNAL_INPUT	EI	EX_INPUT is a non-standard alias. This file is for use by products which provide the capability of temporarily or alternately obtaining source statements from a file external to the input file. For example, the COBOL COPY statement. Single specified value parameter; default = $NULL.
FASTIO		This parameter can be used only by SORT/MERGE. This parameter is on the predecessor product, SORT 5 on the CYBER 170, and is for compatibility only. This parameter has no effect, will go away at a later release, and will not be described in the reference manual.
FORCED_SAVE	FS	If selected, the definition status, of all entities within a subprocedure of a program will be retained upon exit from that subprocedure. Effectively this disallows placing any variables on the stack. Single option parameter. Default = OFF, definition status need not be retained except where so required by the product specification.

FROM F Old file.

This parameter specifies the data input file for the product. For example: the file from which a copy utility reads.

Multiple specified value parameter;
default = OLD.

INPUT I Input file.

This parameter specifies the source input file name to the product. Where reasonable, a list of file names is allowed.

Multiple specified value parameter;
default = $INPUT.

INSTRUCTION_SCHEDULING IS This parameter specifies whether or not instruction scheduling will be performed.

Single option boolean parameter;

YES This option selects the parameter.
default = NO
omitted is same as NO

INTERACTIVE_INTERFACE II This parameter determines whether the product will initiate interactive processing with the user, instead of operating in its usual batch-oriented fashion. This consists of displays written by the product to file $OUTPUT, and user-supplied answers from the file $INPUT. The interactive interface can be invoked either from an interactive terminal, or a batch job.

Single option boolean parameter:

YES This choice initiates the Interactive Interface. The processing of all other parameters on the command line is product dependent (see the appropriate product manual), except that the STATUS parameter is never ignored. The product may, but is not required to, allow the user to decide interactively whether or not the other parameters on the command line are to be

ignored.

NO Do not invoke the interactive interface.

omitted Same as NO.

KEY K Key Field(s).

This parameter specifies the key fields that determine the manner in which input data might be processed by a product. For example, SORT will use the parameter to determine the order records will be sorted.

KEY=<value-list>

The format of the KEY parameter is product dependent.

LEADING_BLANK_ZERO LBZ If selected, leading blanks in numeric fields are treated as zeros in arithmetic statements and comparisons. If not selected, numeric fields that contain blanks are in error.

Single option parameter. Default: the option is not selected.

LBZ = ON selects the option.

LIST L Listing file.

This parameter specifies the file where the product writes the source listing, diagnostics, statistics, and any additional list information (see LO parameter).

Single specified value parameter, default =$LIST.
If a list of files is specified for input, then all of the list outputs accumulate on the specified LIST file.

LIST_OPTIONS LO Listing options.

The options of this parameter specify what extra information will appear on the

listing file (LIST parameter). Multiple options may be specified. The defined options are:

A Attributes. A listing of the attributes of each entity defined within the program is produced. If R was selected, the references are shown on the same listing. See section 3.3.5 for more information on attributes.

B Prohibit Banner. The banner is not sent to the Listing file.

BO Byte Offset. (Release 2 feature) If source statements are listed, an offset field is included (see section 3.3.3.3). This option is meaningful only for wide format listings.

DE DETAILED EXCEPTIONS. Print out exception file messages as often as a record is sent to the exception file.

M Map. A storage layout map for common blocks and equivalence groups.

MS Merge Statistics. Turn on listing of merge statistics.

O Object code listing. A listing of the generated object code with instruction mnemonics.

P Prohibit prompt. The normal input prompts are not sent to the Listing file.

R Cross reference listing. A cross reference of program entities showing locations of definition and use within the program.

RA Cross reference listing of all program entities whether referenced or not.

		RS Record Statistics. List the statistics for the records sorted/merged.
		S Source. Source listing of the program.
		SA Source listing of all source statements including lines turned off by a source embedded NOLIST directive. (See section 2.4.2)
		Multiple option parameter, default = S.
		LO = NONE causes none of the list options to be selected.
LITERAL_CHARACTER	LC	This parameter can be used to change the character that delimits non-numeric literals. Default literal character is quotation mark.
		LC=OFF is an error.
LOAD_COLLATING_TABLE	LCT	This parameter loads an external weight table and associates it with a collating sequence name. The format of the table is AMT$COLLATE_TABLE.
		LCT=(COLLATING_SEQUENCE_NAME, WEIGHT_TABLE_NAME)
		DEFAULT=no weight table is loaded.
MACHINE_DEPENDENT	MD	This parameter specifies whether use of machine dependent source features is to be diagnosed and if so, how severely. The severity level is one of the following:
		I or informational W or warning F or fatal
		Errors of type 'ERROR' will be treated as equivalent to 'FATAL'.
		Single specified value parameter. Default = NONE, machine dependencies are not to be diagnosed.

MASS_STORAGE_LIMIT	MSL	Mass Storage Limit. This parameter specifies the maximum number of characters that may reside on mass storage during execution of the product (for example, SORT). MSL=expr. The number of characters indicated by expr is the mass storage limit. Expr must be an integer.
OMIT_DUPLICATES	OD	This parameter controls omitting al but one of the records which have equal key values. OD=ON Omit all but one of the records with equal values. OD=OFF Do not omit duplicate records. DEFAULT = OFF
ONE_TRIP_DO	OTD	This parameter selects the minimum trip count for FORTRAN DO-loops to be one rather than zero.
OPTIMIZATION_LEVEL	OL	OPTIMIZATION and OPT are non-standard aliases. This parameter specifies the level of object code optimization. All products need not support all defined levels. However if product supports a defined level, it must be selected by the specified option name. Ideally all products which support this parameter should recognize all defined options and issue informative diagnostics for unsupported options that the user selects. Allowable options are: 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, etc.

LOW — Lowest level of production quality code. Code is not completely stylized.

HIGH — High level of production quality code.

Single specified value parameter; default = LOW

OUTPUT — O — This parameter specifies the file where an interactive product writes its output.

Single specified value parameter, default = $OUTPUT.

OWNCODE_FIXED_LENGTH — OFL — OWNFL is a 170 compatible alias. This parameter specifies the record length in characters of all records that will be input to a product from any owncode procedure. See also OMRL and OPn parameters.

OFL = <integer>. Every record supplied by an owncode procedure will contain exactly <integer> characters. Default: (See OMRL).

OWNCODE_MAXIMUM_RECORD_LENGTH — OMRL — OWNMRL is a 170 compatibility alias. The maximum length in characters of any record supplied by any owncode procedure is specified by this parameter. This parameter may not be specified if the product has input or output files and if any of their associated MRL's are at least as large as this MRL. See also OPn.

OMRL = <integer>. There will be at most <integer> characters in any records supplied by an owncode procedure.

Default: If OFL and OMRL are both omitted, the record length specification will depend on the length specifications of the input and output files. If all input and output files have fixed-length records of the same length that length will serve as the default for OFL. Otherwise the largest MRL or FL from any input or output file will serve as the

		default for OMRL.
OWNCODE_PROCEDURE_n	OPn	OWNn is a 170 compatibility alias. Owncode procedure n (n = 1, 2, 3, 4, 5, ...). The maximum of n is left to the individual product. Owncode procedures are user written routines that may be loaded with the product and executed at specified points during product execution. See other OWNCODE parameters for more information on this capability. The procedure specified by this parameter will be executed at a specified point n during product execution. OPn = proc_name. The procedure proc_name will be executed at a specified point n. Default: No procedure will be executed.
RESULT_ARRAY	RA	RESA is a non-standard alias. This parameter is used to return all or part of the result array. RA=array_name, an SCL array Default: Do not return any information from the result array.
RETAIN_ORGINAL_ ORDER	ROO	RETAIN and RET are non-standard and 170 compatibility aliases. Equivalent records or records with equivalent identifying characteristics will be output in the same order as input by a product. For example, with SORT, the equivalent identifying characteristics would be equal keys. The order in which multiple input files are specified is the order in which records with equivalent characteristics are retained with this parameter. ROO=ON. Records with equivalent characteristics will retain their original order.

		ROO=OFF. Records with equivalent characteristics will not necessarily retain their original order. Default: Same as ROO=OFF.
RUNTIME_CHECKS	RC	This parameter controls which runtime checks are compiled into the object code and/or selected for runtime library routines. Runtime checks are product dependent but if a product supports one of the ones described here, it must be selected by the value specified. Defined values are: ALL All supported values are selected. F Files checking. Selects checking of errors involving file variables and buffer variables. N Pointer checking. Selects checking of misuse of pointer variables. R Range checks. This option selects range checking for one or more of the following - character substring expressions - scalar subrange assignments - case variables S Subscript checks. This option causes subscript and index references to be checked to ensure that they are within program defined limits. T Tag field checks. Selecting this option ensures that accesses to variant records are consistant with the value of their tag field (if one exists). This is a multiple value parameter. Default is RC=NONE.
SCREEN_NAME	SN	The name identifying an object screen definition to be retrieved from the user's object library.

		Single specified value paramter. Optional, no default.
SEQUENCED_LINES	SL	This parameter selects FORTRAN sequenced mode source line format as described in section 3.2 of the FTN180 ERS. Note that this format is incompatible with the standard SIS (section 2.3.2) source line format which allows the length and location of a line number to be specified in the source file attributes. Single option parameter. Default = OFF, source lines conform to the standard SIS format.
SESSION_TYPE	ST	One of the keywords EDIT, HELP, UTILITY. Used to present initial SDF session environment. Single specified valur parameter. Optional, no default.
SOURCE	S	SCU input. Line images of the generated program will be written to this file, in a format acceptable as input to SCU. Each program unit on the S file will be preceded by an SCU directive which indicates the beginning of a new source deck. Single specified parameter value, default = $NULL.
STANDARDS_DIAGNOSTICS	SD	Standards diagnostics. (ANSI or other applicable standard). This parameter specifies whether use of non-standard input source statements are to be diagnosed and if so, how severely. There are two values defined: severity, and name of standard. The severity is one of the following:

I Informational error. Standards errors are treated as errors with this severity.

W Standards errors result in warning messages.

F Fatal error. Non-standard usages result in a fatal error.

Errors of type 'ERROR' will be treated as equivalent to 'FATAL'.

The second value, name of standard, is to be defined by the products as appropriate. If this parameter is not specified, then non-standard extensions to the product are allowed, (not diagnosed as errors).

SD=NONE causes standards errors not to be diagnosed.

Multiple specified value parameter, default = NONE.

STATUS STATUS Status Variable.
No alias is permitted for STATUS.

All products are required to support this parameter.

This parameter specifies the name of the SCL status variable to be set by the product to indicate the occurrence of error conditions. See sections 3.4 and 4.4 for an account of the status variable. See also NOS/VE ERS.

Single specified value parameter.

Errors of type 'ERROR' will be treated as equivalent to 'FATAL'.

Default: None.
See Error Processing section 3.4 for a description of error processing that results from use of the default status variable.

SUM   S   Sum Field(s).

This parameter specifies that units of input data having key fields equal (see KEY parameter) may be combined into items or units in a product dependent manner.

(For example, SORT will use the parameter to combine all records, having key fields equal, into a single record. Each sum field in the new record is formed by summing the values in the corresponding fields of all equal records.)

SUM=<value-list>
The value list will contain one or more value sets. Units of input data with equal key values will be combined into new units or items and fields specified by the value sets will be summed, according to product specifications and needs.

Multiple specified value parameter, default = NO SUM FIELDS.

SUBPROGRAM   SP is a non-standard alias.
If this option is selected, the program is compiled as a subprogram instead of as a main program.

SUBPROGRAM = ON selects the option.

Default: the option is not selected.

TAPE_SCRATCH_FILES   TSF   Tape Scratch Files.
The tapes with the names specified by this parameter will be used by the product to reduce the disk space used. The tapes must have already been requested prior to execution. The form is:

TSF=(file_name ,file_name ...)

Default: Tape scratch files will not be used.

TARGET_MAINFRAME TM

This parameter specifies the machine for which code is to be generated. The default, if no option is selected, is the machine on which compilation is performed.

C180MI or C180_MODEL_INDEPENDENT
The code generated will run on any Cyber 180 model.

C180V or C180_VECTOR
The code generated will run on any Cyber 180 model that has vector instructions.

default = omitted

TERMINATION_ERROR_LEVEL TEL

This parameter specifies the minimum diagnostic severity level which will cause a product to return an abnormal STATUS upon completion of processing. A normal status is returned otherwise. The severity level is one of the following:

I or informational
W or warning
F or fatal
C or catastrophic

For 170 compatibility only, products are free to use 'T' in addition to 'I' (however, if only one is used, it must be 'I'). Output will always use 'I', not 'T'. Errors of type 'ERROR' will be treated as equivalent to 'FATAL'.

Single specified value parameter, default = F.

TEXT_NAME TN

Names of texts to be read from the files or libraries specified by the TEXT_RESIDENCE parameter. The total number of values allowed is product dependent. Products that have a text name directive may choose to support the TEXT_RESIDENCE but not the TEXT_NAME parameter. A fatal error occurs if any of the texts specified is

		not found. Multiple specified value parameter, default is no text.
TO	T	New file. This parameter specifies the data output file for the product. For example: file to which a copy utility writes. Single specified value parameter; default = NEW.
TEXT_RESIDENCE	TR	Names of residences (i.e. files or libraries) to be searched to find texts specified by the TEXT_NAME parameter or by product directives. The total number of values allowed is product dependent. If no text names are provided the first text of the first TEXT_RESIDENCE name is the only one used. If text names are provided and TEXT_RESIDENCE is omitted, the default for TEXT_RESIDENCE will be the TEXT_NAME parameter list. In case texts of duplicate names exist, the first one found (in the order in which TEXT_RESIDENCE names are listed) is used. For each name in the TEXT_RESIDENCE parameter list, the product will look for a local file with that name; if not found, the global library set will be searched for a library with that name. If the name is not found, as a file or library, a fatal error will occur. Multiple specified value parameter. Default value list is text name value list.
example 1:		If file F1 contains texts A, C, and D and library L2 contains texts B and C and file F3 contains texts E and A then TN=(A,B,C,D,E) and TR=(F1,L2,F3) will result in selecting texts as follows:

A, C, and D from file F1
B from library L2
E from file F3

example 2: In the above example, if in addition to a library L2, the user has a local file named L2 containing texts B and E, then
TN=(A,B,C,D,E) and TR=(F1,L2,F3)
will result in selecting texts as follows
A, C, and D from file F1
B and E from file L2
nothing from library L2
nothing from file F3

TERMINAL_TYPE TT Terminal Type.

TT=COR
Correspondence Selectric APL terminal.

TT=APL
This type is appropriate when the communications system translates APL terminal codes into a standard intermediate code.

TT=ASCII
For full ASCII terminals not equipped to print the APL character set. Also used for non-APL correspondence terminals.

TT=UCA
For full ASCII terminals. This avoids frequent use of the shift key for letters.

TT=BATCH
For devices that support the ASCII 64-character set. Usually used for batch or remote batch ASCII printers.

Single specified value parameters.

Default is APL for a time-sharing job; and BATCH for a batch or remote batch job.

VECTORIZATION_ LEVEL	VL	VECTORIZATION and VEC are non-standard aliases. This parameter specifies the vectorization level. The allowable options are: HIGH Production-quality code with a high level of vectorization is generated. IL or INNER_LOOPS Only inner loops are candidates for vectorization. NONE VEC=NONE causes no vectorization to be performed. Default = NONE
VERIFY_MERGE_ INPUT_ORDER	VMIO	VERIFY and VER are non-standard and 170 compatibility aliases. Verify merge input order. Selection of this option causes verification that input records to be merged are in correct order. The form is: VMIO=ON. Verify for correct order. VMIO=OFF. Do not verify for correct order. Default: VMIO=OFF.
WORKSPACE	WS	Initial Workspace specification. This parameter specifies the workspace to be activated when the product is called. The parameter is specified with values consisting of the following parameters defined in the NOS/VE ERS: file

2.3 SOURCE INPUT

This section deals with the standard for the processing of source input files by product set members. In this context, a file can refer to data originating from an interactive terminal as well as conventional storage devices. This standard addresses the areas of source file organization, statement format, blank compression, and

response to an empty input file situation.

2.3.1 SOURCE INPUT FILE ORGANIZATION

Source input to CYBER 180 product set members may be designated by the I directive on the control statement. If the I directive is omitted, the source input defaults to the standard input file (batch mode) or terminal (interactive mode). The source input has a sequential structure, and is accessed by means of standard Record Manager interfaces.

Positioning of the source input at open time is constrained by the requirement to allow different product set members within the same job (e.g. different compilers) to access the same file for their input. Therefore, the source input is opened with no-rewind unless the rewind parameter is specified on the control statement (see Keywords and Parameter Descriptions in section 2.2).

2.3.2 SOURCE STATEMENT FORMAT

Each record in the source input contains one to three parcels of data:

- statement identifier (optional);
- line number (optional);
- statement body.

Products should be able to handle the optional statement identifier and line number.

The source input statement may take the following forms, where

b represents the statement body,
l represents the line number,
s represents the statement identifier,

and brackets specify the optional portions of the form:

```
    b l  s
 s  l b
   s b l
   l b s
```

2.3.2.1 Statement_Identifier

Input source records may contain optional statement identifiers such as SCU identifiers. If present, they occupy either the first or last 'n' characters, where 'n' has a maximum value of 18. If the statement identifier occupies the last character positions of a record, all records must be the same length. The location and length of the identifier are file attributes; they are made available via an operating system request.

This feature is to allow files created by source code utilities to be used as source input.

2.3.2.2 Line_Numbers

Line numbers are numeric entities used by compilers and editors. In general, editors will affix line numbers to lines and compilers will use these line numbers for diagnostics, cross reference maps, run time error messages, etc. Line numbers should not be confused with statement identifiers that are produced by SCU and are alphanumeric.

The location of the line number in a text line may be immediately to the left or the right of the text of the line. The position of the line number field is conveyed via the file attributes. The line number field may be from one to six characters in size. The only valid characters in the field are blanks and the decimal digits 0 to 9. Leading blanks are ignored. A line number is terminated by end of field or one or more blank characters.

Additional semantics for the line number field will vary from processor to processor. In particular, many compilers may not accept more than six digits. Another example is the cross reference map produced by CCM which only has space for a six digit line number. Most processors will also insist that the line numbers be unique, ascending, and that every line be numbered.

2.3.2.3 Statement_Body

The body of each source input record is that part which represents the data to be scanned or processed by a product set member. It begins in position 1 if there are no statement identifiers, or if the identifiers appear at the end of the record. Otherwise, it begins in position (n+1) where 'n' is the length of the statement identifier.

The maximum size of the statement body is product set member dependent and conforms to the size specified for the associated language. Source records shorter than the maximum are scanned to the end of the record. Records exceeding the maximum size are truncated (i.e. data is transferred up to the maximum); a diagnostic is returned by the Record Manager.

2.3.2.4 Blank Compression

The CYBER 180 Record Manager is responsible for compression/expansion of blanks. The capability to read the source input in compressed form is not provided. If the requirement for this capability emerges (for performance optimization), it will be addressed in a revision to the standard.

2.3.2.5 Empty Input File

Diagnosis of an empty input file results in the issuance of a standard log message: EMPTY SOURCE INPUT FILE (formatted in accordance with conventions stated in section 3.2). If the job involved is interactive in origin, the message is also sent to the terminal (see section 3.2.1.2.). In addition, generation of the primary output of the product set member involved (e.g. file specified by B parameter for compilers) is suppressed and the SCL STATUS variable (refer to section 2.2.4.2), is set to reflect the error.

2.3.2.6 Null Source Line Convention

The number of records in the source file should be the same as the number of source lines in the source list. Even though a null record has no data, the record should not be ignored. Since, in the source list, the absence of all characters in a record looks the same as a record containing all blanks, null records should be mapped to all blanks.

2.3.3 DISPOSITION OF INPUT FILE

The final action to be taken with respect to the source input file is dependent on the manner of termination of the product set member. For normal termination, the input file is closed with the no-rewind option; this includes the case where an empty file is detected. For abnormal termination, the product set member is responsible for

positioning the input file as if normal processing had occurred, using appropriate facilities of the Record Manager.

2.4 COMPILATION DIRECTIVES

The user of a Compiler may control various activities of the compiler by specifying one or more compile time directives. The directives are expressed either in a special form of a comment within a particular language (e.g. FORTRAN, COBOL) or in special source statements if the language provides such statements (e.g. CYBIL). Compilers that already have special source statements for directives do not have to process directives embedded inside comments. Compilers which now have compilation directives in comments should honor both old and new directives. When a compilation directive conflicts with a control statement parameter option, the compilation directive overrides For example, the options for the parameter LO will be overridden by a conflicting directive unless specifically stated otherwise, such as LO=SA. However control statement parameters denoting file status or destination would take precendence over directives. For example LIST=$null would take precedence over any directives requesting that something be listed.

Since the major programming languages are subject to standardization by bodies such as ANSI, FIPS, and ISO, initial compliance with the form of compilation directives in this section may have to be altered in the event of standards covering this area. Because of the long term possibility that the major languages will be different, full uniformity across 180 products is unlikely. Therefore, products with CYBER 170 directives that do not conform to the syntax contained here should retain compatibility with the CYBER 170 form to minimize migration problems rather than cause a conversion in going to 180 and possibly have to cause a second conversion to comply with external standards. New directives in areas which will never be subject to standardization should follow the form of this section.

The Compilers support two general classes of directives:

- Compiler Call directives

- Source Embedded directives

As discussed in section 2.2, the directives specified on the command calling the compiler are established for the

entire compilation process. They apply to all subsequent compilation units (program modules or subroutines) within the compile process.

Source embedded directives are established only for the compilation unit in which they appear. They are expressed either in a special form of a comment within a particular language (e.g. FORTRAN, COBOL) or in special source statements if the language provides such statements (e.g. CYBIL). Compilers that already have special source statements for directives do not have to process directives embedded inside comments. The syntax of a compiler directive within a comment is as follows:

```
$ directive  [ ,directive ]  . . .
```

Example - FORTRAN source embedded directives

```
C$ directive  - C in column 1
```

Example - COBOL source embedded directives

```
*$ directive  - * in column 7
```

Multiple directives may be contained on the same input statement.

Where directives have parameters, they follow SCL rules.

Source embedded directive format errors are diagnosed with warning or fatal class error messages, as appropriate.

The following standard applies to compilers that process directives embedded inside comments. A compiler is not required to implement all the features listed below, nor is the list restrictive.

2.4.1 PAGE EJECT

EJECT

This directive causes the page line counter to be reset to 1. When the line counter is reset to 1, a standard listing header will be written on the source listing prior to the next source line. This directive will be listed before the page line counter is reset to 1. If the page is at top-of-form when this directive is processed, it is processed as a "no-op". If a continuous page is being written, this directive will simply result in a triple space without a new listing header.

2.4.2 SOURCE LISTING

LIST and NOLIST

The NOLIST directive causes the listing of the source module, incuding compiler directives, to be suppressed at this point. The LIST directive causes the listing of the source module to resume at this point. The directives LIST or NOLIST are listed at the point they occur.

2.4.3 LINE SKIP

SPACE = number

This directive causes the specified number of print lines to be skipped at the point in the source module listing that it is processed. This directive will be listed before the skip action starts. If the page line counter is exhausted before the specified number of lines have been skipped, the line counter is reset to 1 and skipping terminates.

number : integer value 1 thru n; if omitted (including the "="), the default is 1.

2.4.3.1 LINE_SPACING

SPACING = number

This directive specifies the number of lines to be advanced before each source line is listed. The new value for spacing will take effect with the next line following the spacing directive. When listing a source line if the page line counter is exhausted before the specified number of lines have been skipped, the line counter is reset to 1 and skipping terminates.

number: integer value 1 thru 3 indicating single, double or triple spacing; if omitted (including the "x"), the default is "1".

2.4.4 TITLE LINES

TITLE = character string
SUBTITLE = character string
These directives define strings of alphanumeric characters in SCL format which will be printed following the standard page headers on the source module listing (see TITLE Lines in section 3). TITLE causes a page eject to occur, unless

the page is already at top-of-form. TITLE is listed at the top of the new page.

SUBTITLE also causes a page eject to occur, unless the page is already at top-of-form or TITLE has just been processed. SUBTITLE is listed at the at the top of the page following TITLE if there is one.

Compilation Directives

2.4.5 RANGE CHECK

RANGE and NORANGE

The RANGE directive directs the compiler to generate additional object code which performs range checking for subscripts, indexes, scalar assignments, case variables, etc.

The NORANGE directive directs the compiler to not generate additional range checking object code.

The default for the compilation unit is NORANGE.

2.4.6 EXECUTION TRACE

TRACE and NOTRACE

The TRACE directive directs the compiler to generate additional object code which facilitates tracing the flow of the program during execution. The TRACE directive is ignored unless the DEBUG_AIDS=TR parameter is given in the product call command.

The NOTRACE directive directs the compiler to not generate additional flow tracing object code.

Minimum trace information will always be provided. See section 5.4.1.2.

The default for the compilation unit is NOTRACE.

2.4.7 DEBUG STATEMENTS

DEBUG and NODEBUG
Source input statements that are to be compiled only for debugging purposes are enclosed between DEBUG and NODEBUG directives. Enclosed source statements are compiled only if the DEBUG_AIDS=DS is given in the product call command.

2.4.8 SEQUENCE CHECK

SEQUENCE and NOSEQUENCE

The SEQUENCE directive directs the compiler to check the input source statement sequence numbers for ascending order.

If a sequence error occurs, it will be flagged with a warning diagnostic. (See section 2.2.4.2)

The NOSEQUENCE directive directs the compiler to ignore input source statement sequence numbers.

The default for the compilation unit is NOSEQUENCE.

The SEQUENCE and NOSEQUENCE lines themselves are not checked for sequence.

2.4.9 OBJECT CODE LISTING

OBJLIST and NOOBJLIST

The OBJLIST directive directs the compiler to print the object code listing at this point. The NOOBJLIST directs the compiler to stop printing the object listing at this point. The object code will appear in the object code part of the listing (see section 3.3.4).

OBJLIST and NOOBJLIST act independently of LIST and NOLIST. The default for the compilation unit is NOOBJLIST.

2.4.10 STACKING COMPILATION DIRECTIVES

PUSH (compilation directive) and POP
The PUSH command will place the specified compilation directive on the top of the "directive stack". The POP directive will remove the top directive from that stack. This procedure will allow temporary alteration of the local environment without permanently affecting the global environment. For exemple, if it is desired that a called common deck lists its name on the print file, regardless of whether the entire common deck is being listed, then the following would be placed in the common deck:

```
PUSH (LIST)
comment including common deck name.
POP
```

2.5 PRODUCT DIRECTIVES

The format of product directives (commands) must follow the set of language rules and conventions of the System Command Language. These directives frequently occur in products (often various types of utilities) that are not compilers and are thus listed separately. The standard parameter names as described in sections 2.2.4.2 and 2.5.1 must be used as applicable.

2.5.1 STANDARD PARAMETERS

These parameters occur frequently enough to warrant making sure that all commands using them do so in the same way.

Parameter Name	Alias	Parameter Description
BRIEF	BR	This parameter specifies that a brief form of information is requested for display at a terminal or print file. It is a boolean used in conjunction with the full parameter. The brief selection is used as the default.
FULL	FU	This parameter specifies that a full form of information is requested for display at a terminal or print file. It is a boolean used in conjunction with the brief parameter.
COUNT	COU	This parameter specifies a count of units (e.g. files records) associated with the command function. The default value is one.
FILE	F	This parameter specifies the local file name of a file used as the object of a command function. It is used when the file is not one of the specific files identified in section 2.2.4.2 (e.g. COMPILE, INPUT).
WAIT	WAI	This parameter specifies the requestor should be placed in a wait state if a request can't be immediately processed (e.g. a file is busy). It is a boolean used in conjunction with the nowait parameter.
NOWAIT	NOW	This parameter specifies the requestor should not be placed in a wait state if a request cannot be immediately processed. It is a boolean used in conjunction with the wait

		parameter. The nowait selection is used as the default.
USER	US	This parameter specifies a user identification. It is always the 31-character user and family names as specified to gain access to the system.
PASSWORD	PA	This parameter specifies a 31-character password needed to gain access to an entity or to execute a function.
UPON		This parameter specifies the local file name of an output file. It is used when the file is not one of the specific files identified in section 2.2.4.2 (e.g. LIST, BINARY-OBJECT).
LIBRARY	LI	This parameter specifies the local file name of a library file (e.g. source library, object library).

2.5.2 STANDARD COMMANDS

These commands are required, as a minimum, if the functions described by the commands are included in the utility. Utilities may optionally include aliases to the required command.

Command	Description
QUIT	This directive enables the user to exit, or get out of, a utility.

3.0 OUTPUT

All products will follow a uniform set of conventions for generated output, as specified herein. All CYBER 180 products will use the facilities of the CYBER 180 Record Manager for such output.

3.1 RECOMMENDED NUMBER BASES

The use of hexadecimal numbers on output produced by CY180 software must be controlled to promote readability. All products will follow the set of guidelines set herein.

3.1.1 SITUATIONS AND RECOMMENDED NUMBER BASES

Address, Address Offset:	Hexidecimal. When a length is specified in conjunction with an address or address offset, the length is represented in hexidecimal.
Dayfile information:	Decimal statistics, decimal resource limits.
Object Code Listings:	
Instructions:	Hexadecimal (4 or 8 hex digits)
Operand fields:	Decimal
Branch Destination:	Hexidecimal. The value is the instruction offset of the destination instruction rather than the relative offset from the branch instruction.
Instruction Offset:	Hexidecimal.
Core and File Dump:	Various formats should be available, including hexadecimal, ascii, integer, floating point.
Page numbers:	Decimal.

3.2 LOGS

The logs treated in this section are those maintained by the operating system. The OS provides interfaces to put items into the logs and the SIS provides conventions on how to use these interfaces and on the contents of data put into the logs.

The set of logs is divided into two major classes, ASCII and binary. The ASCII logs contain only ASCII-encoded data. The binary logs may contain any type of data.

The logs include:

- system log (ASCII)
- job log (ASCII)
- account log (binary)
- engineering log (binary)
- statistic log (binary)
- job statistic log (binary)

3.2.1 ASCII LOGS

Each ASCII log contains a set of records ordered by time of entry into the log. Each record contains several fields, some automatically provided by the logging mechanism, and some provided by the caller of the mechanism. The following fields are provided by the logging mechanism:

- time of day of the entry of the record into the log

- origin of the message (command, program-issued, command from procedure, etc. -- that is: callers in OS rings may specify the message origin in the call, callers in users rings may not and their messages are always "program-issued").

The system log has an additional system-provided field to identify the message issuing job. Also, each log record contains a field for data provided by the program making the record entry.

Except for certain operating system programs, the interface to be used by the OS and product set for putting messages into ASCII logs is that provided by the "message generator", a facility of the OS (see NOS/VE ERS). The message generator is given a status record that describes the type of message and any variable data to be "edited"

into the message. The message generator:

- finds the appropriate message skeleton in a library which is in the current job library list

- edits the variable data into the message

- logs the final message in whichever log(s) are specified by a combination of:

  * destination specified within the message skeleton record

  * job option selection (e.g., "log only errors", "log all fatals", etc.) -- things such as message importance level are defined in the message generator call.

- displays the message at a terminal depending upon job option

The use of a message generator eases:

- consistency of messages within and across products

- physical compression of message text

- extraction of message types for documentation

- routing/suppression of messages based upon message levels (e.g., trivial, fatal, etc.) and upon user desire for only certain levels ("level" or "importance" is specified in the message generator call, not in the message skeleton)

- installation control over what kind of messages should appear in the system log

Arbitrary user-initiated logging need not use the message generator.

3.2.1.1 System Log

In addition to the basic system-provided fields, each system log entry contains a field identifying the particular job from which the message came or to which it applies.

3.2.1.1.1 PURPOSE

The primary purpose of the system log is to serve as a repository for information regarding external system workload. That is, the work the system was asked to do via commands and the high level responses of the system in regard to the commands.

3.2.1.1.2 CONVENTIONS

The system log contains predominantly a subset of job log messages that are of interest to the installation. This normally includes at least:

- all system level commands (supplied by OS)
- all command completion messages
- start of each job execution (supplied by OS)
- end of each job execution (supplied by OS)
- rerun of each job execution (supplied by OS)
- system identification (supplied by OS)
- other information supplied by the OS like date, hardware and software configurations and changes, deadstarts, recoveries, etc.

The system log should contain only indications of the major changes in state of the system and of individual jobs.

The specific messages that should be routed to the system log in the default "as-shipped" system will be determined on a case-by-case basis using these general conventions as guidelines.

Note that since message destination (which log(s)) instructions are separate from the message-issuing code, this determination does not involve code modification.

See Job Log, Conventions for further guidelines.

3.2.1.2 Job Log

3.2.1.2.1 PURPOSE

The purpose of the job log is to hold a trace of job execution. Information concerning the work requested and accomplished is recorded here. It provides a summary of the flow of the job, problems encountered and charges accrued by the job.

3.2.1.2.2 CONVENTIONS

Keep log messages simple and short. Use the logs for summary information. Use list files or binary logs for detailed or repetitive data.

Messages longer than the listable output "narrow" format are discouraged.

Simple completion messages that convey no more information than "it's done" are not to be put into logs. In a batch case, completion is obvious from the appearance of the next command. In an interactive case, the OS will see to it that the terminal user is notified of completion.

Completion messages that convey a small amount of useful or interesting information are encouraged in order to enhance the "live" appearance of the system. For example, "23 records sorted." or "Cycle 25 used.". Information not specific to the operation performed should not be included, however (like CPU time for a compilation).

Messages (at least the non-brief mode ones) should have the general appearance of normal sentences. That is, they start with a capital letter, are comprised of both upper and lower case letters, and end with a period. When an "extended message" of more than one line must be issued, each line should not, however, end with a period, but each sentence should. This familiar sentence-type structure adds to the "comfortable" feeling that we'd like our users to have for our system.

Accounting and low-level statistical and hardware error information is not to be placed into ASCII logs except by the OS.

Message-at-a-time "current status" messages (like "compiling alpha... compiling beta...") are not to be placed in logs. The OS will provide a means for programs to post these kinds of messages. The current message will be displayed at an interactive terminal when requested by

the terminal users. Posting of these messages is encouraged.

The message generator will supply product and message type identification based upon the status record passed to it in a call. Products should not include this information in messages.

When more than one datum is to be logged, try to minimize the number of messages lines produced by putting more than one datum on a line. For example, issue:

    23 records sorted; Merge order 12 used; 14 insertions.

rather than:

    23 records sorted.
    Merge order 12 used.
    14 insertions.

3.2.2 BINARY LOGS

Binary logs are provided in order to allow the recording of log information in a compact form that is readable primarily by programs. Each binary log contains CYBIL records ordered by time of entry into the log. Each record contains several fields, some automatically provided by the logging mechanism, and some provided by the caller of the mechanism. The following fields are provided by the logging mechanism:

- time of day of the entry of the record into the log
- the identification of the job from which the record came or to which it applies (this field is not recorded in the job statistic log)
- the origin of the record (system or non-system -- indicates only whether the caller is inside or outside system rings, not which product or which system agency --this latter information is given by the "indicator of the type of record" field.)

Fields provided by the caller include:

- indicator of the type of record (e.g., number of FTN source statements, SRJs at end of job, etc. --the indicator codes will be assigned and managed in a manner similar to that used for status condition

codes as described in section 3.4)

- variable data depending upon the record type

Except for certain operating system programs, the interface to be used by the OS and product set for putting records into binary logs is that provided by the "statistics facility" of the OS. The statistics facility is given a data record that describes the type of record and any variable information associated with the record. The statistics facility finds information about the given record type in a "table". This "table" directs the statistics facility to do some combination of the following things:

- add the variable item(s) to counter(s)

- log accumulated counter values to a specified binary log or set of binary logs when a threshold counter value is reached or when a certain time has elapsed since the last "put" to the log(s) of the appropriate counter(s). The set of logs is specified in the "table".

- simply log this record in the "table-specified" log(s)

The use of the statistics facility for binary logging eases:

- installation tailoring of what is considered to be accounting, performance, etc. data. For example, site A may consider CPU time to be accounting data, while site B considers it a workload statistic and considers "number of statements compiled" to be accounting data

- optional routing of statistics to the job statistic log (based upon user desire, but constrained by installation willingness -- via "table" information -- to divulge certain information)

Since the statistics facility determines the log into which a given statistic (for example, PIDFR data) is to be placed (based upon an installation and CDC defined table), system and product implementors should not be concerned with which log(s) are used for "their" statistics. This mapping will be determined later.

3.2.2.1 Account Log

3.2.2.1.1 PURPOSE

The purpose of the account log is to hold accounting and billing information. This consists of resources and/or services used, "who" used them and "who" to charge. The account log should be the only log needed for an installation to do billing.

3.2.2.2 Engineering Log

3.2.2.2.1 PURPOSE

The purpose of the engineering log is to hold information regarding system hardware usage and errors. The engineering log should be the only log needed to perform hardware usage and error anlaysis.

3.2.2.3 Statistic Log

3.2.2.3.1 PURPOSE

The purpose of the statistic log is to hold:

- detailed system workload information

- detailed system performance information (i.e., the way the system responded to the workload)

Although some of this information is recorded in other logs, a separate log is maintained in order to:

- keep other logs relatively "clean" or oriented to their own purposes

- allow possibly large amounts of data to be recorded in a compact binary form

3.2.2.4 Job Statistic Log

3.2.2.4.1 PURPOSE

The purpose of the job statistic log is similar to that of the (global) statistic log. The global statistic log contains information regarding all jobs in the system, but may be read only by privileged programs / users. Individual users, however, may wish to see information that is available about their own jobs. The job statistic log may be read by normal programs / users and contains

information regarding a single job, similar to the "scope" of the ASCII job log.

3.2.2.5 Binary_Log_Conventions

Avoid the use of character data. Since each record type is pre-defined by a CYBIL record type definition, there is seldom a need to describe the various data fields with keywords or the like.

Message type naming follows the naming conventions described in SIS section 3.4.

Use the binary logging facilities for PIDFR data.

See the OS ERS and the SIS Usage Statistics section for minimum list of items to be logged.

Additional conventions will be added as design proceeds.

3.3 LISTABLE_OUTPUT

When a significant amount of information is to be returned to the user, it is written to a "listable output file". The standard format of such a file is described here.

CYBER 180 Output Standard is defined in terms of:

- Output File Organization
- Listing Page Layouts
- Page Header Format
- Format of Each Listing Type
- Object Code and Debug Code

3.3.1 LISTING PAGE FORMATS

In the sections that follow, the contents and format of the standard listings produced by CYBER 180 Products are defined in terms of a vertical and horizontal layout.

3.3.1.1 Vertical Layout

Vertical layout is defined in terms of the first printable line of a form following top-of-form positioning by the printing device. This position is defined as line 1 of a form and is reserved for the first print line of the standard listing header. The product is not responsible for the physical alignment of line 1 relative to the perforated fold on fan-fold printer forms. This is the responsibility of the user on printers with vertical positioning carriage tape mechanisms or the responsibility of the CYBER 180 OS Device Software on printers without vertical carriage mechanisms.

When the last printable line of a form has been written, the product will reset the page line counter to 1. When the page line counter is equal to 1, the next print line written is preceded by a standard listing header with a top-of-form code in the first character position of the header print record. The product is not responsible for the physical alignment of the last printable line relative to the perforated fold on fan-fold printer forms.

3.3.1.2 Format Attributes

Each product must obtain the output file attributes from the operating system at the time the file is opened. These attributes include print mode, page width, connect status, page format, and page length. Vertical and horizontal print density have operating system defined defaults which may be changed by the user.

Output files may be either continuous, which has a line 1 position but does not have a last line position, or paginated (non-continuous), which has both a line 1 positions and a last line position. Continuous form specification files are intended for users using interactive terminals (display or hard-copy) for listable output. Paginated (or "fan-fold") listings are intended for users using line printers for listable output.
For paginated files, page length minus the number of lines of header determines the available lines per page. The operating system provides a (default) standard page length of 66 lines per page at 6 lines per inch vertical print density. This provides an 11 inch form length. Print mode specifies whether or not the paginated file is "burstable" or "non-burstable", with "non-burstable" being the default.

A continuous form file is detected by checking the file's attribute page format. Connected files will default to continuous form mode, but users may override this by specifying a page length for the connected file.

3.3.1.2.1 CONTINUOUS OUTPUT FILE

When formatting listable output for a continuous form, the product uses a standard triple-space code in the first character position of the line 1 output record (usually the first line of the header) to achieve top-of-form positioning. Products will reformat listings for terminal users when required by this standard.

Each type of listing (source listing, attribute listing, etc.) is preceded by a triple-space and the usual header line(s), but there is not pagination as such.

3.3.1.2.2 PAGINATED OUTPUT FILES

When formatting output for paginated listings, the product uses a standard top-of-form code in the first character position of the line 1 print record (usually the first line of the header) to achieve top-of-form positioning. In burstable listing mode, each type of listing produced by the product (source listing, attribute listing, etc.)) begins at a top-of-form position. In non-burstable mode (sometimes referred to as "paper saving" mode), each type of listing is preceded by a triple-space and the usual header line(s) if "proper space" remains on the current page. "Proper space" is defined as 6 plus the number of header lines (insuring that at least 3 lines of output can be placed at the bottom of the page); if "proper space" does not remain, the triple space is replaced by a top-of-form. The source listing always begins at top-of-form, and user-specified page ejects (via compilation directives) always result in a top-of-form position unless the listing is already positioned there.

3.3.1.3 Standard_Carriage_Control_Codes

Carriage control characters that are used should be restricted to the following set:

Character	Action
blank	Space vertically one line then print.
0	Space vertically two lines then print.

- \- Space vertically three lines then print.
- 1 Eject to the first line of the next page before printing.
- \+ No advance before printing; allows overprinting.

This set represents the full extent of compatibility between current CDC usage and the proposed ANSI standard.

Under NOS 180, horizontal print density and vertical print density are file attributes that the user may modify. The NOS, NOS/BE carriage control codes S and T will not be used to set or clear the 8 lines per inch mode.

It will be necessary to make some provision for selection of print density when NOS 180 print files are to be printed by NOS or NOS/BE. The first release of NOS 180 will depend entirely on 170 state for print files.

3.3.1.4 Horizontal Layout

Horizontal layouts are defined for the standard wide format, that is, 132 columns. Assuming a default density of 10 inches per second, 132 columns uses 13.2 inches of the standard 14-inch line printer paper width.

Products are also required to support two terminal formats: the standard "wide format", 132 columns, and the standard "narrow format", 80 columns. Formatting for other line widths in addition to the standard terminal line is permitted. All formats other than the two standard formats are referred to as "other formats".

The first character position of an output record is interpreted by the output device software as the vertical positioning control character and is never printed or displayed. The 132 character positions following the first character position of an output record contain the characters to be printed or displayed. Therefore, the character in the second positon of the output record is printed or displayed in the first position of the output line.

3.3.1.5 Standard_Listing_Header

All CYBER 180 Products will use a standard 2 line page header format on all listings produced by the products.

Through this section, date and time fields conform to standards defined in section 4.1.

3.3.1.6 OTHER_FORMATS

If the line width specified is other than 80 or 132 the heading will be mapped to one of the two standard listing headers. Other output will honor the actual line width, unless specifically column oriented throughout the line {as opposed to column oriented for the first portion and open ended for the last portion, such as source}.

Line 1 of the common page header contains the following fields (field definitions are in COBOL format):

System Name	x(8)	Operating System name.
Product Name	x(8)	The longhand form of the product name, i.e., FORTRAN, FMU, BASIC, etc.
Product Version	9.99	Product version number. The number after the decimal point is shown left justified, i.e. 5.1, not 5.01. This number is updated at the product source code level by the responsible development organization for each new version release.
Product Level	99999	Product modification level contained within the product itself. It is in the form YYDDD representing the julian date when the product was compiled. This number is updated by the build procedures for each new update release.
Listing Name	x(14)	Name of the particular listing being produced. The acceptable listing names are defined in the following sections which define the

		format of each listing type.
Module Name	x(31)	Name of the source module being compiled or the name of the input file being processed. The module name is obtained from the module identification statement provided within the language, or, the default name provided the product when an identification statement is not used. This name need not appear in the first page header if unobtainable. The name will appear left justified within the field if shorter than 31 characters.
Date:	x(18)	Date at the time the first header was written (listing Page Number reset to 1). The date is obtained from the CYBER 180 OS using a standard Program Management request. The date format will conform to the standard given in section 4.1.
Time:	x(12)	Time of day at the time the first header was written (listing Page Number reset to 1). The time is obtained from the CYBER 180 OS using a standard Program Management request. The time format will conform to the standard given in section 4.1.
Page Number	'PAGE' zzz9	Integer number generated by the product starting at 1 and incremented by 1 for each page header written for a compilation unit. The page number is reset for the first page header written for a compilation unit. This field is omitted from the standard

header when a continuous form is specified. The two parts are always separated by one blank.

3.3.2 FORMATS

Two logical line width listing formats are generated by products:

- Page Formatted lines of 132 characters

- 80 Column Formatted Lines of 80 characters

3.3.2.1 Wide Format (132 columns)

A standard header will be written at the top-of-form position of a listing whenever the page line counter is reset to 1 except when a continuous form is being written. A standard header will be written only at the beginning of a listing when a continuous form is specified. A specified page width of 132 or greater will result in the following heading line.

FILE CONTENTS LIST - WIDE FORMAT

Columns	1-14	Listing Name or Columns 1-46 program name
Columns	16-46	Module Name
Columns	48-53	System Name
Columns	55-(54+n)	Product Name (length=n, n<24)
Columns	(56+n)-(59+n)	Product Version (length=4)
Columns	(61+n)-89	Product Level (length=5, blank filled)
Columns	91-108	Date {right justified}
Columns	110-121	Time {right justified}
Columns	123-132	'PAGE' and Page Number {right justified}

All unspecified columns contain blanks.

FILE CONTENTS LEGIBLE - WIDE FORMAT

Columns	1-14	Listing Name or Columns 1-46 program name
Columns	16-46	Module Name
Columns	48-53	System Name
Columns	55-78	Product Name
Columns	80-83	Product Version
Columns	85-89	Product Level
Columns	91-108	Date {left justified}
Columns	110-121	Time {left justified}
Columns	123-132	PAGE and Page Number {left justified}

3.3.2.2 Narrow Format (80 Columns)

The product will reformat the standard page format for a 80 character line. A physical output line format greater than the specified line size may be right-end truncated by the product to the required specification. The excess characters will appear on the next line. A product may choose to reformat narrow listings within the provisions of this document.

The header format {on terminal formatted listings} consists of two consecutive lines containing the fields defined above in the following positions on lines 1 and lines 2. The PAGE and Page Number fields are optional for continuous files.

This information will appear within the following column positions of the first print line (Product Name, Product Version, and Product Level are left justified, separated by one blank column):

FILE CONTENTS LIST

Line 1

Columns	1-14	Listing Name

Columns 16-46	Module Name
Columns 48-65	Date {right justified}
Columns 70-80	Page {right justified}

Line 2

Columns 1-6	System Name
Columns 8-{7-n}	Product Name {length=n,n = 24}
Columns {9+n} - {12+n}	Product Version {length = 4}
Columns {14+n} - 42	Product Level {length=5, blank fill}
Columns 48-65	Time {right justified, left blank padded]

FILE CONTENTS LEGIBLE - 80 Column Format

Line 1

Columns 1-14	Listing Name
Columns 16-46	Module Name
Columns 48-65	Date {left justified}
Columns 70-80	Page {left justified}

Line 2

Columns 1-6	System Name
Columns 8-{7-n}	Product Name {length=n, n=24}
Columns {9+n}-{12+n}	Product Version {length=4}
Columns {14+n}-42	Product Level {length=5 blank fill}
Columns 48-65	Time [left justified, right blank padded]

3.3.3 SOURCE LISTING FORMATS

The following standard applies to compilers, assemblers and interpreters. Assemblers may optionally insert binary information at the left of the source statement. Page ejects may be suppressed for subsequent listings of each module (e.g. Map, Cross reference) if the source listing is short (e.g. 1/2 a page or less).

The number of records in the source file should be the same as the number of source lines in the source list. Therefore, null records should be mapped to all blanks. (See section 2.3.2.6.)

3.3.3.1 Standard_Header_Contents

Every printable source listing contains the following text in the Listing Name field of the standard listing header:

```
SOURCE LIST OF
  --14 characters--
```

A standard source listing header will be written at the next top-of-form position whenever the page line counter is reset to 1. Only the first source listing header will be written on a continuous form.

3.3.3.2 TITLE_Lines

When source embedded TITLE or SUBTITLE directives are processed, the page line counter is reset to 1 and a standard header is written. The title text is printed beginning in column 25 and ending in column 132 of the line immediately following the first line of the standard header. The title lines are followed by a blank line.

```
standard header -- line 1
title text      -- line 2
subtitle text   -- line 3 - 11 (if any)
blank           -- line n
```

n may take the value 3 to 12, depending upon the presence of a subtitle lines.

When a source listing is being formatted for a continuous form, the title line is simply preceded and followed by a single blank line.

If a SUBTITLE occurs without a TITLE, a blank line is

placed in the position which would have been occupied by the TITLE.

When the source input module does not contain a TITLE directive, two blank lines immediately follow the second line of the standard listing header.

3.3.3.3 Wide_Format

The actual source statement listing begins on the line following the blank line following the header, or titles if present. Each source listing print line contains the following optional fields:

Offset	Z(8)	A zero suppressed hexadecimal number (see section 3.1) giving the byte offset in code section of the first instruction generated for the listed source statement. If this field is supported, it is selected by the list option 80. If selected, the field must be supplied for all source listing lines.
Input Line Number	Z(10)	A numeric, zero suppressed number, up to 10 characters in length, allocated to the source line. See section 2.3.
Left Statement Attributes	x(4)	Language dependent attributes.
Right Statement Attributes	x(4)	Compiler dependent attributes.

The Source Record is a required field

Source Record	x(132)	Up to the first 132 characters of the input source record. If the source line is less than 132 characters, this field is left justified. Source Code Utility (SCU) identifiers are contained within this field, if they exist.

If all fields were present in a source listing, column positions would be:

```
Columns     1-8     Offset
Columns     10-13   Left statement attributes
Columns     15-24   Line number
Columns     26-125  Source (including SCU
                    identification when present)
Columns     127-130 Right statement attributes
```

If an optional field is not used the remaining fields will be adjusted to the left.

When the source record (26-125) includes SCU identification information the following column positions will be adhered to for the source record

```
Columns     26-105  Source
            107-125 SCU identifier.
```

The fields should not be changed (mixed) between successive uses. Once the fields desired are established they must remain unchanged.
Existing fields before and after the source record may be blank. If the source record overflows an additional line is written within the source record field. In this case the right attributes field of the first line contains '...' as the first three characters and the rest of the field and offset field are blank. The overflow line contains blanks in the line number field and the remainder of the source record left justified in the source record field. The right attributes field contains the information which would otherwise have appeared in the first line.

3.3.3.4 Narrow Format

The source listing format written on a terminal formatted listing consists of one or more output lines for each input source record.

The first line consists of the following fields:

Line Identifier	Numeric right justified leading zeros suppressed. Optional variable width field up to 10 characters.
Source Record	The source record field size is

dependent on the file attribute maximum record length and the size of the line number field.

A single blank separates these two fields.
The source record field size is dependent on the file attribute maximum record length.

If the source record is longer than the Source Record field then an additional Line is written. The lines are printed with the same format containing blanks in the Line Number field and the remainder of the source line left-justified in the Source Record field.

3.3.4 OBJECT CODE LISTING FORMAT

This is the format for listing lines of object code produced by the compilers at the users request. Assemblers list their source lines formatted as submitted from the input file.
The object code listing shall take one of two forms. The first consists of lines describing each CY180 instruction embedded in the source listing and, as far as possible, following the same line from which the code is generated. The object code line shall conform to the standard defined below. A group of object code listing lines shall be preceded and followed by a blank line.

In the second form, the lines describing the object code, also conforming to the standard defined below, are collected into a separate listing, the "object code listing" which shall conform to a page format common to the listings produced by all compilers. This is defined as follows.

Object code listings consist of instruction descriptions and comment lines.

Instruction Description

With the exception of BDP instrucions, each instruction emitted is described by a single print line optionally preceded and/or followed by comment lines. The instruction description will contain the following fields in the following order, beginning in column 2 of the listable output.

Offset Z(8) A zero suppressed hexadecimal number (see section 3.1) giving the byte

		offset of the instruction relative to an implementation defined base. This base shall be the same base used in the offset field in the source line (if provided.)
	X(1)	
Input Line Number	ZZZZZ9	The number of the input line for which the code is being generated (as far as is practicable).
	x(2)	
Binary	X(20)	An 4, 8 or 16-digit hexadecimal number (adjusted to the left) representing the binary bit pattern corresponding to the generated instruction or data. For readability the suggested form is to arrange the numbers in groups of 4, separated by blanks. The 4 and 8 digit numbers are followed by blanks to complete the field. For narrow format, this field will not be present.
	X(2)	
Label	X(31)	A 1 to 31 alphanumeric character string identifying the instruction label as defined for the CYBER 180 assembler. The label field can be up to 31 characters in length. It can be used in an implementation manner in conjunction with the comment field.
	X(2)	
Instruction	X(10) B(3) X(21)	A character string identifying the instruction and its operands. The mnemonics to be used are those defined for the CYBER 180 assembler. The mnemonic identifier only may be offset by 2 or 4 spaces to distinguish particular instrucions or instruction sequences. (e.g. to identify code generated out of sequence with the source.) Operands are specified in the order defined in assembler specification which appears in an

Appendix (to be supplied). As shown in the format description, the breakdown of the instruction is as follows:

MNEMONIC	X(10)
	X(3)
OPERANDS	X(21)

	X(1)	
Comment	X(25)	An implementation dependent field typically containing user variable or label identifier, register use cross-references.

Narrow Format

The narrow format and 80 Column Format consist of the offset, line number, label, mnemonic, operands and concatenated fields. The binary field will not be present. If the listed line exceeds 80 columns the line will be continued on the next line (called "folding"). For PN other than 80, the actual width specified will be honored; excess information will be folded.

Bdp Instructions

These are described by a line formatted as above, followed by one or two descriptor descriptions. These are similar to the instruction lines except that the mnemonic field is blank and the operand field contains a descriptor in the form defined by the assembler specification.

Comment Lines

These are used to convey more information than can be accommodated in the comment field of an instruction description. They consist of a comment field as defined for the instruction description.

3.3.4.1 Standard_Header_Contents

Every printable object code listing contains the following text in the Listing Name field of the standard listing header:

OBJECT LISTING OF

--20 characters--

A standard object listing header will be written at the next top-of-form position whenever the page line counter is reset to 1. Only the first object listing header will be written on a continuous form.

3.3.4.2 Standard_Instruction_Mnemonics

The instruction menmonics used by the compilers will be those of the CYBER 180 assembler.

3.3.5 ATTRIBUTES LISTING FORMAT

A common format for the Attribute/Cross Reference listing is defined here. It is useable by all currently planned languages for the Cyber 180 and provides enough flexibility to tailor portions of the listing to individual language needs.

The content of the Attributes Listing will vary slightly depending upon whether Cross References were selected or not, but the essential format will be the same. If the user selects both attribues and references, the normal format will be used. When references are not selected, the heading will reflect the difference, but the format will not vary. If references are selected, but not attributes, then some of the attribute information provided will not be listed, providing some additional space for references on the line.

3.3.5.1 Standard_Header_Contents

Every printable attribute listing or attribute/cross reference listing contains the following text in the Listing Name field of the standard listing header:

```
ATTRIBUTES OF
---14 characters----
```

If no attribute list is selected (cross reference selected only) the following text is placed in the Listing Name field instead:

```
REFERENCES OF
---14 characters----
```

A standard attribute list header will be written at the next top-of-form position or following a triple space, as

specified by sections 3.3.1.2.1 and 3.3.1.2.2, and whenever following page breaks occur. Only the first attribute list header will be written on a continuous form.

The standard header is followed by a blank line and one or more lines containing the attribute/cross reference listing heading. This consists of the field descriptions as defined in the next sections, separated by one or more blanks. Numeric fields in the listing are right-aligned with the right-hand side of the description; character string fields are aligned on the left, where appropriate. Some of the field descriptions may be split between two or more lines if required, or omitted, if necessary, as indicated below.

3.3.5.2 Wide_Format

The listing is made up of entries describing the objects defined in the source program. Each entry consists of a definition line, followed by one or more extension lines if required. The definition line gives the line in which the object was declared (or first referenced if implicitly declared), the identifier, and attributes. Extension lines are used if there are more attributes than can be accommodated on one line, and to hold references if selected. If both attributes and references are selected, the references always begin on an extension line by themselves.

The lines contain the fields described in the table below, in the order specified. The table also contains the field description to be placed in the table heading. The final section of the line (for host supplied free form attributes and the references) is continued on extension lines as necessary.

Entries occur in alphabetical order with a blank line inserted between groups of identifiers starting with the same character. Multiply defined identifiers are consecutive in order of increasing level of nesting or in order of occurrence of block.
Variable format fields are optional. They are in the indicated order if used, otherwise the field is not present. The sizes for the given fields are maximum width.

ATTRIBUTE/CROSS REFERENCE LISTING FIELDS

Fixed Format:

Field	Heading	Size	Meaning
identifier	IDENTIFIER	X(31)	The identifier of the entity. The name appears left justified, blank filled.
blank		X(1)	
definition	DEFINED ON LINE	Z(5)	The source line number in which the entity was defined, or (for languages with implicit definitions) first used. It may extend into the identifier field if larger than five (5) digits. The second line of the heading -ON LINE- appears only in the wide format.
blank		X(1)	
size	SIZE unit	X(3)	Size of the entity, in units, defined by the host (either bits, bytes, or words). The units of the size of the entity will appear as "BIT", "BYTE", or "WORD". Abbreviations are BIT, BYT, WRD. Normally the fields for size unit combination will be size Z(8) blank X(1) unit X(4) If the size field exceeds 8 digits, then the fields will be size Z(10) unit X(3) Special case: If size units is no-size, then the size field is allowed to be a signed integer (64-bit). This will be right justified under the SIZE title if possible. If it is too large, it "grows" to the right. If it is so large as to grow into the TYPE field, the

			TYPE field is pushed to the right. This is possible because the LOCATION field is undefined if the SIZE units are no-size.
blank		X(2)	
type of entity	TYPE	X(10)	The type of the entity being listed. Chosen from the list in section 3.3.5.4.1; if the host wishes further qualifications listed they appear in the attributes list.
blank		X(1)	
location	LOCATION SEC+OFF	Minimum X(6)	The location of the entity, where "SEC" is the section name of the section containing the data for the identified reference, and "off" is the offset to the beginning of the section. The section names are:

$LITERAL	The section containing literal constant data.
$STACK	The section containing variables that are allocated on the stack when the containing procedure is called.
$PARAMETER	A subset of the $STACK section containing parameter list variables allocated on the stack by the calling procedure.
$STATIC	The section containing variables that are statically allocated, are not in common, and are not in an explicitly named section.
$REGISTER	Variables not belonging to any

	memory section but existing only in a hardware register.
$BINDING	The binding section.
$BLANK	Blank (unnamed) common.
CYB$DEFAULT HEAP	The system heap.

Code section names will be set to the name of the procedure the section represents. User defined names of section and user declared common blocks will also be specified in full (up to 31 characters).

When a "sec" name is too large to fit into the default field size allocated, the entire name is printed, expanding to the right. A line feed and re-alignment "back" to the next listing field allows continuation of cross reference data generation. For narrow format (section 3.3.5.3), if the "sec" name does not fit on the line, it will be put on the next line by itself, then the rest of the map will continue following a line feed and re-alignment to the next field.

Variable Format

For narrow format listing the variable format fields, continue on a new line beginning in column 15 and extending to column 80. For wide format listing the variable format fields continue on the same line beginning in column 75.

Field	Heading	Size	Meaning
block number	BLOCK	Z999	Specifies the block or subroutine in which the object was defined.
blank		X(2)	
nesting	NEST	ZZZ999	The nesting level of the

level	LEVEL		declaration of the entity if a block-structured language. If the host is not a block structured language, the nesting level is omitted. The second line of the heading - LEVEL - appears only in the wide format.
blank		X(2)	
containing entity	CONTAIN OR DECLARED IN	X(31)	The name of the containing or qualifying entity. Blank if the entity is not contained or qualified. The "contained within" form is for arrays and structures. The "declared in" form is for local variables. The entire heading is on one line.
blank		X(2)	
basic attributes	ATTRIBUTES	X(12)	The "basic attribute" of the entity (entry, external, etc.) chosen from the list in section 3.3.5.4.2. Blank if non-applicable to the entity. If there are no optional fields and the basic attribute is not present, the whole line is omitted in narrow format.
user attributes	(no heading)	free field start-ing on a sepa-rate line	Other host defined attributes separated by commas. These attributes begin on a separate line beginning with column 54 for wide format and column 15 for narrow format listings. Each definition specified by the host is placed on one line if possible, otherwise each that overflows starts a new line. If the definition doesn't fit alone, it is broken at a blank.
references	REFERENCES For combined map, subheading will be	Z(6)X(2)	References on the identifier line begin at column 54 on. the listing In narrow format the first line has two

"REFS", starting on a separate line.	references. Subsequent lines start in column 15 and have six references per line. In wide format all lines start in column 54 and have 8 references per line. For mixed mode listings, see discussion below.

The format for references is a six digit, right justified, blank filled integer, followed by an optional slash (/), followed by qualifying letter, chosen from the list in section 3.3.5.4.3. This combination is followed by a blank.

In mixed mode (combined attributes and references), both the attributes and references are handled as described above, except that the first reference line has a subtitle -REFS- placed at its left. The subtitle -REFS- is placed in columns 9-13 on the narrow listing and in columns 48-52 in the wide listing.

Since the user may select the attributes listing separate from the references listing, the following changes occur when both are not selected together. If attributes only are selected, the references are not listed. If references only are selected, the identifier, line number and references fields are used and the references begin at the end of the first line, not on an extension line.

3.3.5.3 Narrow_Format

The narrow format listing will have the same format as the wide listing with the exceptions noted in the describitions in section 3.3.5.2. and with the exception that the attributes and refernces fields will continue on an extension line beginning in column 15 and extending to column 80.

3.3.5.4 Standard_Field_Values

3.3.5.4.1 ENTITY TYPES

Each entity is assigned a basic, cross-language type. These appear in the "Type" field as one of the following:

TYPE	ABBREVIATED FORM
Simple var,	VAR

Array,	ARR
Structure,	STRU
Member,	MEM
Condition,	COND
Constant,	CONS
Type,	TYPE
DEF,	DEF
Program,	PROG
Module	MOD
Procedure,	PROC
Function,	FUNC
Label,	LAB
Switch,	SWCH
File,	FILE
Format,	FMT
Paragraph,	PARA
Section,	SEC
Impl name,	(for Implementor name) IMPL
Group,	GRP
Alias	ALIA
Error	ERR
Attr name,	ATTR

Each host need not support all types of entities on this list, but should define a consistent mapping into a subset of the above. The final entry ("Error") should be used for entities whose definitions contain syntax errors sufficient to prevent the compiler from determining the user's intentions.

3.3.5.4.2 BASIC ATTRIBUTES

This field contains attributes basic to the entity definition which are exclusive of one another. If the entity does not fall into one of the following catagories of attributes, then the field is left blank. These are:

Attribute	Abbreviated Form
undefined	UNDEF
unreferenced	UNREF
EntryPoint	ENTRY
External	EXTRN
None (field is blank)	

3.3.5.4.3 REFERENCE TYPES

The standard reference type abbreviations will be:

M	the entity was set (modified),
(blank),	the entity was used (slash is also omitted)
A	the statement defined an entity attribute,
S	the entity was a subscript or index,
I	the entity (usually a file) was referenced in an I/O statement
R	the entity was read into (or, if a file, was read)
W	the entity as written from (or, if a file, was written)
P	the entity was used as an actual parameter

For all listings containing references there is a legend of the possible reference types and their one character abbreviations at the bottom of each page. This legend is right justified and takes the form abbrev = full name, ...

For example: M=modify, A=attribute, S=subscript, I=I/O ref, R=read, W=write, P=param.

Each host may choose to use the entire set or a subset thereof, but it is hoped that most hosts will use the entire set.

3.3.6 DIAGNOSTIC LISTING

The diagnostic listing for compilers, assemblers, interpreters, etc., consists of diagnostic messages. Diagnostics are listed in either of two modes, at the host's choice. The first method lists all diagnostics and a diagnostic summary at the end of the listing, following the Attribute/Reference list (if selected). The second method lists syntax diagnostics in the source listing as they are detected, with later (non-syntax) diagnostics and the diagnostic summary being listed at the end of the Attribute/Reference list. If the first method is selected, the host may also choose to have the location of the diagnostic occurrence flagged in the source listing (by means of a caret symbol under the offending column).

When compilation occurs with zero diagnostics a diagnostic summary will be produced consisting of the single line 'NO ERRORS'.

3.3.6.1 Standard_Header_Contents

Every printable error listing/summary contains the following text in the listing name field of the standard listing header:

```
ERROR LIST OF
---14 characters----
```

A standard error listing header will be written at the next top-of-form position or following a triple space, as specified by sections 3.3.1.2.1 and 3.3.1.2.2, and whenever a subsequent page break occurs. Only the first error listing header is written on a continuous form.

3.3.6.2 Standard_Diagnostic_Listing_Format

All diagnostic listings, whether grouped together at the end of the other listings or printed intermixed with the source listing will have the same basic format. When grouped, they will be listed in source line/statement column/diagnostic number order. When grouped and the diagnostic number is not being printed, they will be listed in source line/statement column/order of issuance order. When printed intermixed with the source listing, they will be printed in the order the host detects them.

Column positions are specified for the case where all fields are used, and remain the same if an optional field is not used.

Column Position	Contents	Format	Meaning
1-9	level	X(9)	error severity level of the diagnostic
11-17	line nr.	Z(6)9	source statement number on which the error occurred. For diagnostics intermixed with the source listing, this field contains *ERROR*.
22-24	diag. no.	ZZ99	diagnostic number of the error (assigned by the host). This field is optional.
26-28	COL	X(3)	The abbreviation for the word column in intermixed mode. If

			the column number field contains zero, 'COL' is suppressed. In grouped mode this field contains the column number described below, and that field is blank.
30-32	col. no.	ZZ9	column number in which the error was detected. Blank if not applicable. In grouped mode the column number is present in the col (26-28) field and the column number field is blank.
34-eol	text		the diagnostic text (defined by the host). Each word within the text is separated by one space and the line is filled as required. Extension lines begin with the text position through the end of the line, single-spaced. In intermixed mode, the *ERROR* indicator is re-issued on extension lines.

Diagnostic summary for products that use diagnostics intermixed with source should include a page list of pages with diagnostics.

3.3.6.3 Standard_Diagnostic_Summary_Format

The diagnostic summary will follow the diagnostic listing for grouped diagnostics, or stand-alone for intermixed diagnostics. In either case it provides the user with a summary of diagnostics detected and listed, as directed by the EL parameter.

There will be an diagnostic summary line for each level of diagnostic detected during the compilation. If no compilation errors (at any level) were detected, then that is noted. The following format will be used for all of the summary lines:

columns 3-6	****	Summary line flag
columns 9-14		Number of diagnostics of a given category, in the format Z(5)9.

columns 16-eol	Text, in the format "aaaa diagnostics", where aaaa is the category being summarized. If the diagnostics were not listed (due to EL setting) then "(unlisted)" is appended to the message.

If only one diagnostic at a given level was issued, the word "diagnostics" will be "diagnostic" in the messages.

3.3.7 COMPILATION OPTIONS

The compilers will produce one or more lines of output to indicate which control statement options were selected for this compile (either by default or explicitly). The format of this line will reflect section 2.2 of this standard. This line will appear after all other listings for each module. It is produced whenever any list option is selected and not produced for LO=NONE.

3.4 ERROR_MESSAGES

This section describes conventions for all ASCII error messages. This includes log messages (to system and job logs), interactive messages, and error messages written to the OUTPUT or other files (reference logs, section 3.2). The conventions include the use of the Message Generator, message identification, and message wording.

3.4.1 CONDITION CODES, EMBEDDED PRODUCTS, AND MESSAGE GENERATION

3.4.1.1 Condition_Codes

A summary of the NOS/VE status record fields is noted below. The NOS/VE ERS should be referenced for a complete description of the status record and the Message Generator interfaces. All products, including products embeded in a host product, shall adhere to these conventions.

Normal - A boolean which has a value of FALSE if a request could not be processed correctly and TRUE if it has been processed correctly.

Condition Code - A unique condition code is defined by a two-character product identifier (see section 4.1.1.1) plus a five-digit error code indicating the specific error (e.g., IDnnnnn). All CDC error numbers must

be in the range 1 to 9999. Error numbers in the range 10000 to 19,999 are reserved to identify errors from user developed products.

In cases where the condition code is communicated in a form other than a status record (e.g., FORTRAN IOSTAT, AAM's ES field) the field "condition" from the status record must be used.

Text - A string used to substitute text into the error message template associated with the condition. The first character of the text signifies the character used as the text delimiter. All text items are terminated by the delimiter or the end of text.

3.4.1.2 Embedded_Products

The embedded product should return abnormal conditions to the host via the standard status variable, and with condition code and product identifier of the embedded product. Conditions anticipated by the host should be translated by the host into the appropriate condition with the host product identifier and condition code where user action is required. Conditions not anticipated by the host can be passed on to the user. Conditions issued by the embedded product should be clear enough for the user to determine required corrective action, including the need for PSR submittal where appropriate.

3.4.1.3 Message_generation

The NOS/VE Message Generator is used to format and output all error messages output to logs or to an interactive users terminal (note this does not include diagnostics generated during compilation). It produces a standardized message using the NOS/VE status record and message templates from a message dictionary.

3.4.2 MESSAGE TEXT

The message templates are determined by each product and included in a message dictionary. The NOS/VE ERS should be referenced to determine the formats of message templates.

3.4.2.1 Message_Formats

The message generator formats and outputs messages according to conventions based on the message's destination: terminal, log, file, or returned to the caller.

Terminal:

```
Format:     text .....  or  IDnnnnnn    text .....
Example:    Permanent file (pfu) not found.
```

Log, OUTPUT, or other file:

```
Format:     IDnnnnnn    text .....
Example:    CB0326      Incorrect delimiter, comma
                        assumed.
```

Returned to caller:

```
Format:     IDnnnnnn    SID  mmm   text .....
Example:    AM1234      SOP  012    File (lfn) already
                                    opened.
```

Where:

text	=	Text of message
ID	=	Product identifier
nnnnnn	=	The error condition code (unique error number for a given product)
SID	=	Product subidentifier
mmm	=	Subcondition code.

The combination IDnnnnnn will be known externally as the "C180 error number". It is a unique system-wide code by which any error message can be identified to the user. It is always printed before the message text on all batch listings. It can optionally be included with messages output to an interactive terminal and is available to the terminal user requesting additional error analysis assistance via the NOS/VE HELP facility.

3.4.2.2 Error_Summaries_in_Logs

When error summaries are listed on a file, log messages should be issued to both the system and user log according to the following rules and formats:

System and User Log

n fatal errors [in x]

User Log Only

n warning or trivial errors [in x]

n number of errors
x is the name of the module, program, subroutine that contains the errors.

Error summaries should only be used when it is inconvenient to provide a description of an actual error.

Catastrophic errors are not included because they should always result in a log message indicating the catastrophic error. The error counts should be issued to the log even if the EL (error level) parameter excludes them from the listing.

3.4.2.3 Message_Wording

Error messages represent a very important, though often neglected, interface between software and user. Proper attention to producing polite, correct, and clear error messages can do a lot toward improving the overall usability of the system. The following conventions should be used in defining error message text:

- Messages should be polite and courteous. Words such as "illegal" should be avoided in favor of words like "incorrect" or "unknown". Error messages should, where appropriate, suggest what the user ought to do to correct the error. For example, use:
  The line number parameter must be an integer.
  not:
  Illegal line number.

- Messages must be formatted for 80 character displays. Telegraph style is much better than long-winded prose. However, the message must be descriptive of the error. Messages like "Bad Argument" don't say enough.
- Consistent terminology is extremely important. For system-wide terms consult Section 6.0 of the SIS. For terminology specific to a product, again consistency is the important factor.

- Identification must be provided with variable information. For example:
  use:

  File (lfn) not found.
  Variable (var) must be scalar.

  not:

  (lfn) not found.
  Variable (var) must be scalar.

- Use ending punctuation. It indicates to the user that the message is not continued on the next line and adds to the readability of the message.

- Messages should be oriented toward an inexperienced or casual user such that the message can be understood and appropriately responded to without reference to a manual.

- Abbreviations should be avoided. Whenever possible limit the characters used to alphanumerics plus english punctuation. Avoid use of characters that appear differently on different devices. CDC's 64-character ASCII subset and lowercase alphanumerics can be used.

- Words beginning with "multi" and "non" are not hyphenated. Don't use "(s)" to indicate an optional plural usage; either singular or plural is acceptable.

- Error messages should use upper and lower case as they are normally used in the English language. Upper case should be used to distinguish "computer" words from normal English words. For example:

  File FRED not found. Specify keyword NEW.

3.5 USAGE STATISTICS

All products are required to collect and log statistical information.

This section describes what these statistics are used for, the NOS/VE Statistics Facility, which statistics will be collected by products and which will be collected by the O/S and when statistics should be logged.

Because the Statistics Facility is under control of NOS/VE

product designers are requested to convey statistics requirements and plans to the NOS/VE design team.

3.5.1 PURPOSE OF STATISTICS

Statistics logged by products may be used for billing, measuring reliability, measuring performance, debugging, product planning or some other purpose. The ultimate use of the data cannot be determined when the product is being designed. For example, a statistic such as "number of source statements compiled", which is normally considered a performance statistic, could just as easily be used as the basis for charging or billing a user. It's not inconceivable that a student could be billed based upon (number of source statements) - (number of comment lines) + n * (number of errors) if this data were available for each compile.

There are three physically different logs for recording statistics. They are the accounting, job, and system statistics logs. See section 3.2. A particular statistic may apply to one or all three of these logs. To prevent products from having to issue the same statistic several times, to prevent product designers from having to decide which statistics will be used for which purpose, and to provide installations and users maximum control over statistics gathering, NOS/VE provides a centralized Statistics Facility.

3.5.2 STATISTICS FACILITY

NOTE: This is preliminary information. The NOS/VE ERS should be referenced for a more complete and up to date specification. The ERS is the controlling document for this product to O/S interface.

The NOS/VE Statistics Facility is used by products and the O/S to accumulate statistics and write records into binary logs.

The Statistics Facility

- associates a statistic code from a status record with a particular table entry

- adds job and task identification to the variable data if appropriate. Task identification specifies which

of the possible several asynchronous instances of execution within a job the current statistic belongs to.

- routes the statistic to the appropriate log or logs and/or adds it to a specific counter as determined by the table entry. Counters can be dumped to binary logs at specific times or events.

Data passed to the Statistics Facility include:

- statistical code - ordinal of this particular statistic.

- optional byte string - for products this string contains product ID, module identifiers if appropriate, and any other product or statistic unique descriptive data. Product ID is the two character identifier defined in section 4.1.1.

- optional count fields - 0 to n numbers, the numeric part of the statistic.

Data returned include:

- Status - boolean indicating whether or not the previous Statistic Facility request was processed correctly.

The method for assigning statistics ordinals will be specified in the ERS. A separate range of numbers will probably be reserved for users.

3.5.3 PRODUCT STATISTICS COLLECTED BY NOS/VE

In general, the O/S is responsible for collecting job step statistics that can be determined external to the product, that is statistics that the O/S is capable of determining.

For each product identified in SIS section 4.1 that is directly invoked by the user, e.g., via command or as a program initiated task, NOS/VE will record resources used per invocation. Resources accounted for include:

- total CP-time

- maximum virtual memory used

- maximum real memory used

- average working set size
- CP-time per memory size used
- number of I/O requests
- amount of data read/written to files

Additional data to be collected for each invocation of a product include:

- origin of job step - batch command, terminal command, procedure file, executing job.
- type of termination - normal, product error, time limit, invalid memory request, operator drop, etc. A recovered condition does not cause product termination.
- average interactive response time for interactive products - the average elapsed time between input data available and output data issued to terminal.
- the fact that the product was invoked (added to count of the number of separate invocations).
- number of modules loaded (input units for the loader)
- source languages of modules loaded (added to language usage count).

- disk accesses per CP second.

These same statistics, resource usage and additional data, may be collected for any user initiated job step whether it is a user supplied program or a CDC supplied product. Statistics for products will be identified by product ID, correction level information, and task number acquired during loading.

Task number identifies which invocation of product x issued the statistic. Several asynchronous tasks may be executing the same product. Statistics for user written tasks may be identified by primary module name, task number, and other data gleaned from the file ID.

Number of invocations will be collected for all products both user called and product called service products such as Access Methods, and all user tasks. It could be

collected for all modules on system libraries. For products, it represents the number of times the product was invoked over a given time span; for user programs it represents the number of times a program module written in language x was used over a given time span. The time span is installation definable.

3.5.4 STATISTICS COLLECTED BY PRODUCTS

In general, products are responsible for collecting internal statistics that only they can know. These statistics provide a means of characterizing the work performed by a product; they are used for evaluating product performance. Statistics to be emitted are specified in the product's DR. There are two classes of product generated statistics - input units and internal usage statistics.

3.5.4.1 Input_Unit_Statistics

This class of statistics is concerned with the number and nature of user controlled data processed by the product. All products are required to log number of input units processed per invocation.

Input units for various product types are:

Product Type	Input Unit
Language translators, such as FTN, COBOL, CYBIL,	Source lines
Utilities such as SORT/MERGE, FMU	Data records
Services such as AAM	Functional requests

Other meaningful input unit related statistics must be emitted by products, where applicable. AD/R specified measurement of performance requires generation of such statistics for certain products. Examples of these other input statistics are:

Language Translators

- number of modules processed
- number of source statements
- number of lines consisting solely of blanks

- number of lines consisting solely of comments

- number of source statement errors for each error level

Utilities

- number of records of each recognizable record type supported by the product

- number of records in error

Services

- number of functions of each defined type

- number of illegal/ill-formed requests

Many other potentially useful input related statistics are possible. Products developers are encouraged to collect additional input statistics they feel are worthwhile. An example is source statement frequency, i.e., number of each type of source statement encountered.

3.5.4.2 Internal Statistics

This class of statistics is concerned with internal measures of the product as opposed to measures of the input data.

An example of such a statistic is:

product options in effect for this execution e.g., what control statement parameters were selected.

Products are required to log major options selected (such as optimization level used by a compiler). Each product is required to specify which options are major.

Many additional statistics (such as internal errors encountered) may be applicable to specific products. Developers are encouraged to collect other statistics they feel are worthwhile.

3.5.5 WHEN TO LOG STATISTICS

The two issues of concern are:

- when should detailed optional, statistics be accumulated and logged?

- when should subordinate service products such as AA

log statistics?

All statistics will be controlled by installation or user controlled switches. The statistics Facility will provide the mechanism for setting and clearing these switches. Each procedure that issues diagnostics must check the appropriate switch before calling the statistics Facility. The switches will probably exist as an array of bits that can be referenced but not changed by user tasks. The NOS/VE ERS will specify the exact form. Subordinate products and routines may either issue continuous statistics at product determined intervals or events or they may accumulate and report them under control of the host product.

For products such as AM and AA whose statistics could be meaningful regardless of the host, the first approach is acceptable. For example, statistics could be gathered from file open to file close for each file. Anyone interested in AA statistics for a job step would have to sum up the individual statistics on the log file.

For subordinate products and routines such as the common compiler modules whose statistics are not meaningful out of context, a mechanism should be provided to enable the host to force out statistics on demand. That is, the host must be able to inform the subordinate that its work is complete. If the subordinate actually issues the statistics, the host must provide its product ID to the subordinates so that ID can be included in the statistics. If the host actually issues the statistic, the subordinate must return all data and identifying information. The first method is preferred since the host does not need to know which or how many statistics the subordinate is collecting.

Note that all methods of statistic reporting require products to recover from catastrophic external and internal errors. Products must regain control so that they can output the accumulated statistics. Furthermore, since O/S logs the reason for termination, products that recover from abnormal external conditions must be able to let the abort happen after they issue their statistics so that the correct reason for the termination is recorded. Products that detect internal errors must be able to indicate that such an error happened when they abort, so that "internal error" is recorded as the reason for the abort. A product may choose to terminate via an abort when no product error has occurred.

4.0 SYSTEMWIDE CONVENTIONS

This section describes the operating system and product set convention which must be followed by all standard software.

The term "global" as used in this section refers to constant and type definitions that are global to several products. It does not mean "global" within a particular product.

4.1 NAMES, DATES AND TIMES

Standard system naming conventions are needed for the following reasons:

1. Permit recognition of the origin and maybe the purpose of the named entity just by its name.

2. Prevent duplication of names between different products.

3. Designate categories of names that are reserved for CDC usage so that they will not be duplicated by application programmers.

These names may be declared as entry point names, file names, SCU deck names, or as names for common system entities such as installation options. The common system entity names must be declared in a form that provides a simple source of availability for use by any system implementation language, (CYBIL or assembly).

4.1.1 NAMING CONVENTIONS

The system defined global names will be generated according to the following convention:

PPC$XXX

where:

PP -- Is a 2 character alphanumeric product identifier or other global identifier for the owner of this symbol.

C -- Identifies the class of the name.

$ -- is the special character $
XXX -- 2 or more alphanumeric characters which establish uniqueness within the levels of identification described above. The maximum length of this field is determined by the other users of these names. Example: The loader determines the maximum length of an entry point, the record manager the maximum length of a file name.

4.1.1.1 Product_Identifiers

AA	Advanced Access method
AD	Ada
AL	Assembly Language
AM	Access Method
AP	APL
AV	Accounting Validation
BA	Basic Access methods
BC	BASIC
CC	Common Compiler modules (CCM)
CB	COBOL
CF	CYBIL Formatter
CG	Common code Generator (CCG)
CL	Command Language
CM	Configuration Management
CS	Character virtual terminal Screen management
CU	Concurrent maintenance Utilities
CV	CYBER Vectorizing Code Generator
CY	CYBIL
DA	DCN dump Analyzer
DB	Interactive Debug
DC	Distributed Communications
DD	Data Dictionary
DF	Distributed Files
DM	Device Management
DP	Display
DS	Deadstart/recovery
DU	NOS Dump Analyser Utility
ES	Edit Screen
FA	File migration Aids
FC	FORTRAN Compiler
FD	Format Display
FL	FORTRAN run time Library
FM	File Management (in BAM)
FM	File Management utility

(Double use of FM to be resolved by DAP, if there is a problem)

FS	File System
FT	FORTRAN. Global to FC and FL

FV CDC FORTRAN (Vectorizing)
GS Graphics virtual terminal Screen management
HP Hardware Performance analyzer (HPA)
HU Help Utilities
IC Interstate Communication
IF Interactive Facility
II Interactive Interface
IM Information Management
IO Input/Output
JF Job File manager
JM Job Management
JS Job Swapper
KR Keypoint Reporting Utility
LG Logs
LI LISP
LL Loader/Library generator
LN Logical Name
LO Loader
LU Link User
MA Maintenance Application Language for Equipment Testing (MALET)
MC Marketing Configurator
ML Math Library
MM Memory Management
MP Matrix Algorithm processor
MS Maintenance Services
MT Monitor
MV MAIL/VE Electronic mail system
NA Network Access method
NC Network Configurator
NF Network File Transfer |
NP Network Performance |
OC Object Code utilities
OF Operator Facility
OS Operating System
PA Pascal
PE Programming Environment
PF Permanent File management
(to be phased out, per SIS Dap S4730)
PM Program Management
PP Peripheral Processor
PR PROLOG
PS Product Set
PT Performance Tools
PU PF Utilities
P1 PL/I
QF Queued Files
QU Query Update
RF Remote Host Facility Access Methods

RH Remote Host facility
RM Resource Management
SC Source Code utility
SD Screen Design Facility
SE Set management
SF Statistics Facility
SM Sort Merge
SR Conversion services
ST Software Tools
(Current use of ST for Sets will be phased out in favor of SE)
SV Shared Variables processor
SY System
TD Test Data base
TM Task Manager
TU Terminal Utility
TW Translater Writing System
US User (e.g., for "user" statistics)
UT Tests
VC C Compiler
VX VX/VE - UNIX Emulator
ZS Zeus

4.1.1.2 Other_Global_Identifiers

RA Release Administrator

This product identifier is used to identify Installation parameters and procedures associated with a NOS/VE product.

4.1.1.3 Classes_of_Names

The following list of identifiers naming classes is used for code and deck naming purposes:

A -- architectural and design documentation
B -- design documentation (internal to CDC).
Also the three following special blocks:
CYB$DEFAULT_HEAP
DBB$NE_LINE_ENCOUNTERED
DBB$NEW_PROC_ENCOUNTERED
C -- constant
D -- declaration (decks containing types and/or constants)
E -- exception condition name
F -- file
H -- documentation (headers, inline text)
I -- inline code or installation/integration
K -- keypoint or keyword

M -- module
P -- procedure
S -- section (static data section and/or common block)
T -- type
V -- variable
X -- XDCL'd (decks containing procedures or variables)

XREF

frequently ...

4.1.1.4 Special_Characters

The use of the $ sign in a name identifies the name as one belonging to CDC. CDC users will avoid any duplication with CDC names by not using the $ in any of their names.

Some programming languages such as FORTRAN do not allow imbedded dollar sign characters in their names. CDC supplied procedures callable from these languages will not conform to the $ sign rule.

4.1.1.5 User_Global_Names

User global names follow the rules defined in sections 4.1.1-4.1.1.4, except the form of a user global name is:

PPC#XXX

4.1.1.6 Deck_Naming_Guidelines

Relationship of Code and Deck names

The deck name must be the same as the code name whenever possible. In instances where it is absolutely necessary to group types, constants, etc. in the same deck, then it is allowable to use a conglomerate deck name which is different than the component code names.

"Design Documentation" Deck Names (A and B)

Class A decks are for architectural and design documentation releasable with the code.

Class B decks are for requirement/design documentation not releasable with the code (e.g., DR-type specifications, such as performance) but relevent to code maintenance.

A "design documentation" deck has the EXPAND attribute value of TRUE or FALSE, depending upon the needs of the product. The content

of this deck and all decks *COPYed by this deck are input to the processor named in the PROCESSOR field of the SCU deck. The PROCESSOR is in the form of a string which represents the command by which the processor is invoked. Documentation decks may not be processed by a compiler but rather by a text formatter processor. For instance, documentation decks might be processed on the C170; then the PROCESSOR might be TXTCODE. In the future, documentation decks may be processed on the C180 by a text processor.

Documentation decks not to be released to customers must be identified (by group) by the development project to Integration, which will remove such decks during preparation of SMD release materials.

"Compilable" Deck Names (M and F)

A "compilable" deck has an EXPAND attribute value of TRUE. The content of this deck and all decks *COPYed by this deck are input to the processor named in the PROCESSOR field of the SCU deck. The PROCESSOR field is in the form of a string which represents the command by which the processor is invoked. Parameters which are to be passed to the processor, and which are meant to be invariant (such as optimization level, or debug level), may be included in this string. The order in which invariant parameters are specified is precisely the order in which they are defined for the command, even though the parameters are specified as equivalenced parameters. File references are disallowed in the processor string.

M class decks contain a processor defined "compilation unit". Examples of such compilation units are: MODULE to MODEND for CYBIL, IDENT to END for ASSEMBLE, PROGRAM to END for FORTRAN, etc. Module decks represent the units which are maintained in a Binary Module Replacement environment. A parent/child relationship exists between M and P (or V) decks which contain XREFs. To denote this association, the name of the parent M deck is assigned as a GROUP attribute of the child P or V deck. Thus, any modifications made to the child deck results in the ability to generate the parent deck by interrogating the GROUP attributes of the child deck. Likewise, all decks which *COPY the child deck can be generated through use of the INCLUDE_COPYING_DECKS Criteria File directive. The name associated with a M class deck is the same as that specified on the MODULE, IDENT, PROGRAM, etc. statements. If a M deck contains code which is later Bound into a Module of a different name via the BIND_MODULE subcommand of CREOL, then the name of this Bound Module is assigned

as a GROUP attribute of the M deck. The name of a corresponding F deck which contains specific CREOL directives associated with the binding of this module is specified as a GROUP attribute of this M deck.

F class decks contain source data which is retained as a file, or contains processor directives for the processor named by the processor field. These decks contain, or *COPY decks containing, information necessary for establishing program descriptions, omitting entry points from Bound Modules, or establishing SCL procedure libraries. A typical F deck might contain COLLECT_TEXT and ADD_MODULE commands, and *COPY's to procedure decks (P decks) which contain the source of procedures to be added to a procedure library. Another use of F decks is as a container for directives to the Real Memory Builder or Virtual Memory Linker in which segment attributes are defined. If a SCL procedure is to be executed from a file rather than a procedure library, then the processor type of the F deck is SCL rather than CREOL. The name associated with F decks is the name of this file as it is accessed when the processor is invoked, or the name of the resultant file which is to be created.

"Non-compilable" Deck Names (C, E, I, K, P, S, T, V)

A "non-compilable" deck is one with the SCU deck EXPAND attribute value of FALSE. This type of a deck is *COPYed by "compilable" decks and assumes any attributes associated with the *COPYing deck.

K class decks contain KEYPOINT, KEYWORD, or statistic codes. These codes are defined in terms of a constant plus relative offset, and define a set of related data. K decks are given a conglomerate name which indicates the type of data being described (KEYPOINT, statistic, or KEYWORD).

C class decks contain Constants. Constants are used to impose an upper limit on ranges, and provide a starting point from which relative offsets are computed. A constant is global in nature by virtue of its appearance in a C deck. Those constants which define product restrictions due to their design (eg. OSC$MAX_NAME_SIZE), and those constants which represent Installation options are the two categories of constants with packaging affects. The former category of constants are named so as to describe the scope of effect upon other products or subproducts. Product specific constants should be named

using product specific two-character identifiers. The latter category of constants are named with the RA product identifier to indicate that the "Release Administrator" assumes ownership for the value assigned to the constant. Since source code will be unavailable at many sites, the use of constant values must be avoided. Global constants should exist as one constant per deck. The name of the deck should be the same as that of the constant being defined. Ownership of a constant is assumed by the decks which *COPY a constant deck. Automated generation of all decks affected by a change to a constant deck is accomplished through the INCLUDE_COPYING_DECKS Criteria File Directive.

T and E class decks contain Types and Exception conditions respectively. Since Exception conditions are typically described in terms of a constant plus a relative offset, it is acceptable for a constant declaration to appear within the E deck. E decks are given a conglomerate name for the condition range. Types may be either fixed or adaptable. In such cases where a type is defined in terms of constant (such as an equivalenced ordinal type) then the constant value may be contained in the T deck. T decks are named the same as the primary type defined in the deck. If the type is a record, then the name of the deck is the name of the record defined in the deck.

P class decks contain code procedures. The content of such decks is the source of non-XDCL'd procedures, SCL procedure definitions, or XREF declarations for XDCL'd procedures. A SCL procedure definition will contain a PROC to PROCEND sequence if the P deck is used to form a procedure library, otherwise the procedure will be defined in a F deck. Code sequences which are not bracketed by PROC to PROCEND, or a corresponding sequence such as SUBROUTINE and END, should be contained in I (inline code) class decks. Each external procedure should have an accompanying H deck that documents the procedure.

V class decks contain variable declarations, or the XREF to XDCL'd variables. A child/parent relationship exists between a V deck containing an XREF and the corresponding M or F deck in which the variable is XDCL'd. The name of the V deck is the same as name of the variable which is defined in the deck. The name of the parent M or F deck is assigned as a GROUP attribute of the V deck.

H class decks contain documentation, such as headers or text that may be called into a generated document such as an ERS.

(A and B class decks may be used for high-level architectural and design documentation. H decks may be used for detailed design documentation, particularly to support external procedures.)

I class decks contain inline code or integration/installation parameters. In the case of code, the justification for such decks involves performance, where repeated code cannot be formed into a PROCEDURE due to the expense incurred in the procedure call. Otherwise, FUNCTIONS or INTRINSICS may be contained in I decks. Inline text is text used for code documentation purposes which may also be called into a generated document such as an ERS.

S class decks contain blocks of related data such as static data of Common Blocks. An aggregate name is associated with this collection of data unless the text data describes a specific entity. In such cases, the text data assumes the same descriptive string as that associated with the entity it is describing (eg. OSS$MAINFRAME_PAGEABLE_HEAP).

"Non-compilable" Deck Names (D, X)

Decks belonging to this category represent packaging anomalies, and should be avoided whenever possible.

D class decks contain conglomerates of Types and/or Constants. Since it is difficult to ascribe meaningful identity to such combinations, the use of the D class should be avoided when possible. It is advantageous to define parameters for procedures in a D class deck. This anomaly exists due to the nature of the constructs necessary to define procedure parameters.

X class decks contain the XDCL definition of procedures or variables. The recommended location for the source of XDCL'd procedures or variables is within a compilable deck (M or F class). Combining XDCL'd procedures into a single module is a function of the CREATE_OBJECT_LIBRARY utility command BIND_MODULE. If the XDCL'd procedure is GATED to other products and/or users, then the XDCL'd name is preserved as a result of Binding, otherwise the name is discarded provided there is a corresponding XREF at binding time. Therefore, it is a product's responsibility to CHANGE_MODULE_ATTRIBUTES of the Bound Module to OMIT names within Bound (or Unbound) modules which are not to be externalized by the product. It is recognized that being able to combine several XDCL'd procedures and/or variables into a single compilation unit can provide

additional debug capabilities provided by a compiler. It is for debug purposes that X class decks exist.

4.1.1.7 SCU_GROUP_NAMING_GUIDELINES

Critical to the structure of the product's source libraries, and to the efficiency of the source maintenance procedures, is the association of SCU "group names" with each deck on the library. These groups may be used to manipulate "blocks" or "groups" of decks, such as all decks in a job template or system core library, fairly easily.

The different types of groups are identified by the conventions used to name them. Except for the "processor" and "generic" groups (described below), the format of all group names is:

xxy$aaaaaa

where 'xx' is the product identifier, "aaaaaa" is a descriptive name for the particular group, and "y" is one of the following:

S Specifies a "Source" group, i.e. a subdivision of the source library into component libraries. Several examples of groups of this type are:

oss$program_interface (osf$program_interface)
fcs$front_end
cbs$cobol_source

F Specifies a "destination File" group (i.e. a file onto which a deck is to be placed after processing). Several examples of groups of this type are:

aaf$44D_library
osf$monitor
osf$object_code_utilities

G Specifies a "Group", i.e. a collection of decks that have been decided would be useful to be able to refer to en masse. Several examples of groups of this type are:

cbg$procedure_common_decks
cbg$run_time_procedures
fcg$bridge_modules

NOTE: All decks on the source library which belong in the library psf$external_interface_source should be associated with the group name "psf$external_interface_source". Keypoint decks and error codes should also belong to this group.

Most other group names will follow the conventions described in the previous text for the product source libraries. However, there are two classes of groups for which this is inappropriate: "processor" groups and "generic" groups.

A "processor" group will be given the same name as the corresponding processor, e.g. the group name for decks to be compiled by CYBIL will be "CYBIL". Other processor group names are:

- assemble
- fortran
- cobol
- pp_compass
- cp_compass
- cybil_cc

Note that some of these must obviously be processed on the NOS side of the machine.

A "generic" group is used in those cases where knowing the processor for a deck is insufficient for some purpose. The generic groups that have been identified and are required, if applicable, are:

- common
- program_descriptions
- message_templates
- scl_procedures
- ccl_procedures
- scu_selection_criteria
- build_procs
- deleted_decks

All decks which are called by other decks will belong to the group "common". When a deck needs to be deleted, it should belong to a group "deleted decks". At this point in time, the deck is not really deleted, so the build procedure must be specifically excluding this deck until someone in integration can physically delete the deck. This will only be done between releases to insure our ability to back up to a previous level.

Some of these generic groups overlap to some extent with the processor groups.

A deck may have up to 255 group names associated with it. At this point in time, a group cannot be associated to another group. A DAP has been written to allow this capability. For the time being, issue the change_deck SCU command to establish all deck to

group associations.

4.1.2 RESERVED FILE NAMES

The following files will have special uses:

INPUT is that portion of the primary input file that follows the System command statements.

OUTPUT is the primary output file and contains a copy of the job dayfile at the end when printed.

For interactive jobs, the terminal is assumed to be both INPUT and OUTPUT.

4.1.3 DATE AND TIME

While NOS/VE provides date and time data in several formats, products are restricted to using one format unless language standards dictate otherwise. The format to be used is the installation defined default format.

For fixed position listing and file formats, date and time fields must be large enough to accommodate the longest forms returned by the O/S.

4.2 INTERACTIVE_PROCESSING

This section identifies capabilities products must provide to support users interfacing the system from interactive terminals.

Products support different levels of interactive usage. Therefore a product does not necessarily support all of the capabilities described below. For example, products that typically perform batch functions (e.g. compile FORTRAN source) do not provide the same level of interactive capability as one that typically performs an interactive function (e.g. query a file).
Many of the capabilities are provided by the operating system and therefore are available to all terminal users independent of the program/application being used.

Specific interactive capabilities to be provided by C180 products are described below. A key is used to indicate which products must include design and implementation of the capabilities. The keys are:

A - It is the responsibility of all products to support

the capabilities marked with the A key.

O - This key notes the terminal capabilities supported in the implementation of the operating system. These are available with all interactive usage and are provided by:

- Job Management
- Message Generator
- File Routing
- Basic Access Method
- Transaction Executive
- Network/Communications Access Method

I - This key notes the terminal capabilities supported by "interactive products". These programs normally carry on a dialogue with a terminal user to obtain feedback and dynamically direct processing. They include:

- Job Management
- Message Generator
- File Routing
- HELP Utility
- Transaction Executive
- BASIC
- APL
- OS Utilities
- Query/Update
- Report Writer
- FMU
- Interactive Debuggers
- SORT/MERGE
- SCU
- Editors
- Conversion Utilities

4.2.1 INTERACTIVE OUTPUT

4.2.1.1 General

a) The page width and length at an output device varies not only by device type, but also by the size of paper being used in the device. The user must be able to indicate the operational page width and page length of the output device. Defaults that correspond to the terminal characteristics are supported.
-O-

b) Lines of data that exceed the output device page width

must be delivered without loss of data. Data that would be output beyond the right side of the page must be folded onto a second or successive line (reference section 3.3.1.5).
-0-

c) The user must be able to have every output line formatted so as not to exceed the output device page width provided the output device page width is not less than 80 characters. As a minimum, the user must be able to specify that output be formatted for page widths of 80 or 132 print positions (reference section 3.3.1.4).
-0-

d) Any output that may go to an ASCII sequential file may instead go to a terminal.
-0-

e) Any output may contain a carriage control character (reference section 3.3.1.3).
-0-

f) The carriage control character will direct printing of an output file and will not appear in the print output.
-0-

4.2.1.2 Messages

a) Messages must be courteous. Words such as "illegal" should be avoided in favor of words like "incorrect" or "unknown". Error messages must, where appropriate, suggest what the user ought to do to correct the error.
-A-

b) Messages must be formatted for narrow listings.
-A-

c) Messages must be meaningful such that an inexperienced or casual user is able to understand the message and respond appropriately without reference to a manual.
-A-

d) Any message longer than 20 characters must have an alternate brief counterpart.
-A-

e) A user must be able to select either a brief or long form of a message. When using the brief form of message, the user should be able to request that the

last message be repeated in its long form.
-O-

f) Messages soliciting input (prompts) should always be used to indicate that the user is expected to supply input.
-I-

g) Prompts should appear on the same line as the input whenever physically possible.
-I-

4.2.1.3 Listings

a) Pages of output that are longer than the output device page length must be delivered without loss of data. Data that exceeds the page length must be continued onto a second or successive page.
-O-

b) Pages of output must not be delivered to a non-hardcopy output device so fast as to overwrite any previous output before the user can read it if a wait option has been selected by the terminal user.
-O-

c) The user should be able to have heading information repeated on the second and successive terminal pages of a listing. When display space is limited and the information band width is low, the user might choose to not use space to display repetitive headings and be able to see more data. Where the listing consists of many columns that are hard to differentiate, the user might choose to have headings repeated on every page. This capability requires that: 1) Page Header text be identified so it can be discarded, and 2) Title text be identified so it can be replicated.
-I-

d) When initiating a function the user must be able to select alternate amounts of detail to be included in the listing. By selecting less detail, the user ought to be able to have more items displayed on each page, and not just get less information per page.
-A-

4.2.2 INTERACTIVE INPUT

These standards supplement section 2.3.

4.2.2.1 General

a) User discovered typing errors must be correctable by backspacing and retyping.
-O-

b) The user must be able to cancel the input line being typed at any point before input completion is indicated.
-O-

c) No extraneous blanks will be appended to the end of the user defined input data for padding. Application of this rule is only required if allowed within a product's standard.
-A-

d) No user typed trailing blanks will be deleted from the input data. The application of this rule is only required if allowed within a product's standards.
-A-

e) Any input that may come from an ASCII sequential file may instead be supplied by a terminal connected as that file.
-A-

f) A single input may consist of more than one line. A prompt may allow multiple lines of input in response. An input collection mode may be implemented in this manner.
-O-

g) Operations requiring only a few parameters should not require more than a single input. The user may enter all parameters for a directive or all directives for a single system level command as a single input in order to reduce the number of interactions and the time to complete the directive or command.
-O-

h) The user must be able to use the standard abbreviations for command names, directives and parameter identifiers in order to reduce typing.
-A-

l) After input of a command or directive has been completed, incomplete input should not be treated as an error, but should cause further prompting for the missing parameters.
-I-

4.2.2.2 Input_Diagnoses

a) Errors in input will be diagnosed immediately following the offending input line.
-I-

b) Diagnosed input errors must be correctable without "exiting" the dialogue with the program.
-I-

c) Where possible allow diagnosed input errors to be corrected without re-entering the entire line.
-I-

d) Any input diagnosed to the terminal must be correctable by terminal input immediately following the diagnostic whether or not the original input was from the terminal (see 4.2.3.1). After receiving the corrected input from the terminal input will revert to the primary source.
-I-

4.2.3 CONTROL

4.2.3.1 Connectivity

a) The user must be able to have his terminal connected as an ASCII sequential input file and an ASCII sequential output file for any program.
-A-

b) The user must be able to suppress the verification listing of input when the input source and the output destination are both the terminal.
-A-

c) Products that allow input directives from a file other than INPUT must allow the user to have input directives from a source other than the terminal listed for verification at the terminal.
-A-

d) Products that allow input directives from a file other

than INPUT must allow the user to have input directives from a source other than the terminal diagnosed to the terminal.
-A-

e) The user should be able to logically disconnect the terminal from an executing program without causing the program to be suspended. The program should continue execution and the user should be able to simultaneously enter other commands (including execution of other programs).
-O-

4.2.3.2 Interrupts and Connection Breaks

a) The user must have a method for gaining control over a program in execution. This is known as an interrupt.
-O-

b) An interrupted program will not be aborted as a result of the interrupt.
-O-

c) For a program written to execute in an interactive environment, an interrupt must cause the program to enter a known state. This state will normally be one that solicits directives or commands from the terminal.
-I-

d) For a program written to execute in a batch environment, an interrupt must cause the program to be suspended in such a manner as to be restartable during the same terminal session. Control is returned to the command language interpreter.
-O-

e) A connection break is often caused by a communication line failure. A connection break must not cause the terminal session to be aborted, but must cause it to be suspended in such a manner as to be restartable when the terminal user can again get connected.
-O-

f) A user must be able to restart a suspended program.
-O-

g) A user must be able to terminate a suspended program without first restarting it.
-O-

h) A program written to execute in an interactive environment must accept a termination directive in the state entered as a result of an interrupt. This directive must be the same as the corresponding system command to terminate a suspended program.
-I-

i) Any incomplete terminal input request from a program that is suspended should be reissued (with the proper prompt) when the program is restarted.
-I-

j) The terminal user must be able to interrupt the output being delivered to the terminal and cause the remainder of the output to not be delivered to the terminal until the next prompt.
-O-

4.2.3.3 Status

a) The terminal user must be able to solicit a report to determine the process of a program, without causing a change in the state of the program.
-O-

b) Progress reports must indicate the functional progress of the program. For example:

"compiling program SAM..."

"compiling subroutine TOM..."

"preparing global cross-reference..."
-I-

c) The terminal user must be able to solicit a report to determine the system environment within which a program is running without causing a change in the state of the program. An installation option to disable this must be provided.
-O-

d) The system environment report must indicate (possibly indirectly) the response time the terminal user can expect to experience. This might be by indicating the length of swap-out queues, the number of interactive users, etc. An installation option to disable this must be provided.

-0-

e) The terminal user must be able to solicit a report of the state of its program without causing a change in the program's state. An installation option to disable this must be provided.
-0-

f) The program state report must indicate the rate at which the user's program is progressing relative to real time, and the impediment to progress. For example:

".14:23:13 - 2.54 CP seconds Swapped Out"

".14:24:40 - 5.72 CP seconds Running"

".14:27:10 - 6.21 CP seconds Finished"

Possible states should recognize the points of delay in the system; these might be Paging, Swap-out, Waiting for terminal input, etc.
-0-

g) The terminal user must be able to define terminal attributes to be associated with an interactive session (e.g., backspace character, echo mode, screen size). The terminal user must be able to display the terminal attributes currently in effect for a terminal.
-0-

4.2.3.4 Help

a) The terminal user should always be able to get a reasonable response to the input HELP. The response should identify the user's alternatives and possible correct input. As a directive, HELP should indicate what directives are able to be used at that point. The user should be able to proceed after the response to a HELP input as if the interaction had never taken place.
-0-

4.2.4 PRODUCT SET RUN TIME COMMANDS

4.2.4.1 PAUSE and STOP Literal

PAUSE n (in FORTRAN) and STOP literal (in COBOL) are very similar. They should be processed in the same way.

a. The message PAUSE text will be displayed on the operator's terminal or console. Text is n or literal, and is a maximum size of 58 characters. For batch jobs, the operator is the primary system operator. An OFP$SEND TO OPERATOR with an OPERATOR ID of 'SYSTEM OPERATOR' is executed to send the message. For interactive jobs, an AMP$PUT NEXT request referencing the file OUTPUT is executed to send the message. This will result in a message on the terminal.

b. In batch, an OFP$RECEIVE FROM OPERATOR with the WAIT parameter and the same id as above is executed to suspend the job and wait for the typein from the system operator. The operator will respond with a REPLY ACTION command. In interactive mode, an AMP$GET NEXT request on the file INPUT is executed (this may not be legal, another connected file may have to be used). In either case, the data is thrown away and the job is continued.

4.2.4.2 ACCEPT FROM CONSOLE

The ACCEPT FROM CONSOLE (in COBOL) should be processed in exactly the same way as STOP literal (4.2.4.1). Text would be the data from a previously executed DISPLAY UPON CONSOLE WITH NO ADVANCING or the message 'ENTER COBOL INPUT VIA REPLY ACTION' if there was no DISPLAY. Interaction is with the system operator only. (If messages sent via OFP$SEND TO OPERATOR also appeared on the terminal, it could cause confusion for the terminal operator.)

4.3 INSTALLATION PARAMETERS

NOS/VE will permit modification of all system parameters dynamically during system execution. The term "installation parameter", as used in the classical CDC sense, is not valid for NOS/VE.

System parameters fall into the following general categories:

- Hardware characteristics (e.g., # of CPU's, type of CPU)

- System and product defaults (e.g., default tape density)
- Accounting parameters
- Limits parameters (e.g., maximum FL)
- Timing parameters

System parameter defaults can be set at the following times:

- Compile time (compilation at CDC)
- Build time (deadstart tape build at user site)
- Deadstart time (via operator type-in)

These parameters may be tested dynamically and action taken accordingly. The product set will require no parameter specifiation, and will dynamically test system parameters during execution via requests to NOS/VE.

The following table indicates the permitted range of system parameter control for the product set and operating system. An X indicates that the option is allowed, and a blank entry indicates that the option is not allowed. Any exception must have the explicit approval of AD&C.

Time of Set and Use / Type of Parameter	Set Times				Use Times	
	Comp. Time	Build Time	D/S Time	Exec. Time	D/S Time	Exec. Time
Product Set						
Hardware						X
Defaults						
Accounting						
Limits						X
Tuning						
OS						
Hardware	X	X	X	X	X	X
Defaults	X	X	X	X	X	X
Accounting	X	X	X	X	X	X
Limits	X	X	X	X	X	X
Tuning	X	X	X	X	X	X

+-------------------+-------+-------+------+-------++------+-------+

4.3.1 GENERAL GUIDELINES

As a general rule, the number of system parameters should be kept to an absolute minimum. This will minimize the additional testing imposed by these options and will reduce the number of "different" versions in the field.

A firm requirement on both the operating system and the product set is that no recompilation at a user site will ever be required to install the software. This is a requirement of binary release.

4.3.2 LIST OF PRODUCT SET PARAMETERS

The following system parameters may be tested dynamically by the product set via requests to NOS/VE (including networking):

- type of CPU
- OS name and version
- line width or screen width
- terminal type
- screen length or page length
- print lines limit

4.4 ERROR PROCESSING

The purpose of this section is to describe the conventions and responsibilities of processing different error conditions.

4.4.1 STATUS VARIABLE

All command and procedure interfaces to the system that are visible to the end user must have a status variable as a parameter. The status variable is used to convey the result of the command or procedure and, in case of error, provide information explaining what went wrong.

For commands, the status parameter should always be optional. When it is quoted by a user, the assumption is that the variable will be tested subsequently in the command stream and some appropriate action taken. Therefore, the conditions returned to the user should only convey information the user is likely to understand.

For procedures, the status parameter is required. Again

the conditions returned should be as understandable to the user as possible. This is particularly important when there are multiple procedure calls made within our software as the result of a single call by a user procedure. Emphasis should be placed on improving the status returned to the user rather than blindly passing back obscure status from the depths of the system.

Detailed formats of the status variable are available in the NOS/VE ERS.

4.4.2 ERROR TERMINATION

There are a number of errors that can occur in a product, some of which can be detected and some of which can't. This section deals with the processing to be performed when detectable errors occur.

First of all, the product should try to detect as many errors as gracefully as possible. This means that internal software tests should be used to detect errors as well as using the condition handling facilities of the operating system to receive control in the event of a system or hardware detected error. The product cannot simply rely on the standard operating system abort processing.
When an error is detected, the product should provide as much of the following error localization information as possible. Some of the information will not be applicable to all products.

- Type of error termination (standard system messages should be used for this message).

- Full traceback of the call sequence to the procedure containing the error. This will be by procedure name and line number or relative address depending upon the amount of traceback/debug information released with the product.

- Information regarding the user data being processed. For a compiler, this might be the procedure name and line number currently being processed. For a utility or data management product, it might be the current record.

- Optional dumps of useful internal tables.

The above information should only be logged for error

terminations that are probably caused by product failure. It should not be logged for conditions such as time limit or operator drop which are clearly not product errors.

4.4.3 INTERACTIVE ERROR PROCESSING

This section supplements section 4.2, "Interactive Processing".

In considering this topic it is necessary to distinguish between error messages and diagnostics. These terms are difficult to define precisely but are intuitively distinct nonetheless. An error message is generally a summary of a command; in an interactive environment it wants to be displayed at the terminal so the user can find out what happened. Diagnostics are generally a part of a larger whole (e.g., listable output) which due to their volume only want to be selectively displayed.
An example is a compiler which provides a single error message telling how many errors occurred during compilation and produces a diagnostic for each compilation error.

4.4.3.1 Error_Messages

a. All error messages should be issued via the standard message generator. The message generator will determine whether the message should go to the terminal or the log, etc.

b. Messages must be courteous. People tend to react in a more emotional fashion when using a computer interactively than when using it in a batch mode. Words such as "illegal" should be avoided in favor of words like "incorrect" or "unknown". Error messages should explain to the users what they did wrong and, if possible, how to correct it.

c. Messages must be meaningful such that an inexperienced or casual user is able to understand the messages and respond appropriately without reference to a manual.

d. Any message longer than twenty characters must have an alternate brief counterpart. The user must be able to select either the brief or the long form of the message.

4.4.3.2 Diagnostics

a) Points b and c, above also apply to diagnostics. Diagnostics should explain the problem from the user's perspective rather than the program's. For example:

"Comma missing after third parameter"

instead of

"QVPPARSEPR detected illegal syntax".

b) While diagnostics are not typically displayed at a terminal by default, they are looked at by interactive users. This must be considered when defining the location of the diagnostics in the listing, identifying the diagnostics with a mark that is uniquely detectable with a text editor, etc.

4.4.3.3 Input Diagnosis

This section applies to all input that can reasonably be expected to come from a terminal (e.g., command utility subcommands).

a. Errors in input will be diagnosed immediately following the incorrect input.

b. Diagnosed input errors must be correctable without exiting the dialogue with the program.

c. Diagnosed input errors may be corrected without reentering the entire line.

d. Any input diagnosed to the terminal must be correctable by terminal input immediately following the diagnostic whether or not the original input was from the terminal.

4.4.4 BATCH ERROR PROCESSING

4.4.4.1 Error Messages

Batch error messages should follow exactly the same guidelines as interactive particularly the usage of the message generator.

4.4.4.2 Input_Diagnosis

The kind of user interaction that is desirable in interactive mode is of course inappropriate in batch mode. Emphasis should be placed on detecting as many real errors as possible even after a fatal error has occurred. The key word here is "real"; producing a large number of extraneous error messages or diagnostics will ultimately lead to people only correcting one problem at a time.

4.4.5 TRANSACTION ERROR PROCESSING

This section will be added when more design on the transaction facility has occurred.

4.4.6 RESTART

This section will be added when more design on the system restart capabilities has occurred.

4.5 EFFECTIVE_USE_OF_C180_HARDWARE

4.5.1 HARDWARE OPERATION

This section describes software conventions which must be followed for the hardware to function in a predictable manner.

4.5.1.1 Interlock_Words

Convention: Locate all interlock words in cache bypass segments.

Special system instructions are provided in the CPU and the IOU to interlock multiple processors/IOU. In general, these function by exchanging the contents of a register and a word in memory. Following this exchange the register may be investigated to determine whether the lock has been set. For example, a zero word in memory can be selected to mean "no lock", then by exchanging a non-zero register the lock will have been set if a zero value is returned. It is imperative that such interlock words be unique. To guarantee this they are placed in cache bypass segments. Notice that the instructions which are designed to test and set locks automatically bypass cache. Problems arise when the interlock words are accessed by other instructions such as loads.

4.5.1.2 Pre-serialization of Clear Lock

Convention: Before clearing a single bit lock (via a Store Bit Instruction) first set the lock by a Test and Set Bit Instruction.

Care must be taken whenever an interlock word is set or cleared to pre-serialize the operation. This is done to ensure that, in the event that memory references are being satisfied out of sequence, all outstanding memory references are completed before changing the lock. In practice, CYBER 180 systems designed to date always satisfy memory references in sequence. However, this may not always be the case. The instruction which sets a single bit lock (Test and Set Bit) performs the necessary pre-serialization. However, to clear the lock a Store Bit (with a zero operand) must be used. Since this instruction has a general utility it does not pre-serialize. To compensate, the Test and Set Bit instruction post-serializes. Hence, to ensure a pre-serialization of the clear lock, the lock should first be set (with a Test and Set Bit instruction), then cleared by the next instruction.

4.5.1.3 Register Reservations

Convention: Registers A0-A4 and X0-X1 shall be reserved for special functions.

The CYBER 180 instructions make use of certain registers to hold given values. The assignments are as follows:

- A0 - Dynamic Space Pointer (DSP)
- A1 - Current Stack Frame Pointer (CSF)
- A2 - Previous Save Area Pointer (PFA)
- A3 - Binding Section Pointer (BSP)
- A4 - Argument List Pointer (ALP)

These registers hold those values by software convention, but a convention which is supported by the hardware. Hence, it is very important that they be supported by all software procedures. In particular, A1 and A2 must never be altered by instructions other than Call, Return and Pop.

In addition to the reservations above, registers X0 and X1 have a special meaning in the hardware. For many instructions, the X0 designator is used to indicate no register. Hence, register X0 cannot be used by these instructions. Both X0 and X1 are used as fixed utility registers for several instructions. Examples are:

1) Load/Store multiple and CALL instructions use X0 for a save area descriptor.
2) All compare instructions return a value to X1-Right, as does the Mark to Boolean instruction.
3) The BDP instructions optionally use X0-Right and X1-Right to hold operand lengths.

Since these registers are used for special purposes, care must be exercised if they are used in a general manner.

4.5.1.4 Alignment_of_Tables_and_Words

Convention: Align certain tables and words on specified boundaries.

Although CYBER 180 is nominally a byte addressable machine, real memory is organized into 64-bit words. Consequently, the performance of certain operations has been optimized by placing the operands on word boundaries. The complete set of data alignments necessary is given below, along with a brief description of why the alignment is required and what will happen when the data is not aligned correctly.

4.5.1.4.1 64-BIT WORD BOUNDARIES

The following data either must be, or should be aligned on word boundaries:

1) Process Segment Table - For performance reasons the hardware indexes into the segment table at a word boundary. The virtual memory address translation mechanism will fail if the segment table is incorrectly aligned.

2) Binding Sections - To maximize the reach into the Binding Section by the Call Indirect instruction, access is made to a word boundary. If the Binding Section is incorrectly aligned, then an Address Specification Error results when a Call Indirect is issued.

3) Procedure Entry Points - To maximize the reach of the Call Relative instruction, a branch is made to a word boundary. Since the instruction forces the address to a word address, results will be unpredictable if the procedure target was not correctly aligned. Note that even though it is not strictly necessary for procedures called via a Binding Section to be word aligned, difficulties could still result if they are not. This is because the CYBER 180 Library Generator, in the process of "binding" may convert Call Indirect instructions to Call Relative instructions.

4) Debug List Entries - To simplify the hardware, and to optimize performance when in debug mode, the hardware accesses debug list entries on word boundaries. Incorrect alignment will cause unpredictable results.

5) Interlock Words - Interlock words used in conjunction with the Compare/Swap operation must be aligned on a word boundary. This is necessary for the processor to satisfy the non-preemptive requirements of the instruction. Processors utilize the 64-bit memory exchange function in this operation. That function operates on a real memory word. Incorrect alignment will yield an Address Specification Error.

6) Stack Frames - By software convention only,

	stack frames should be aligned on word boundaries. This enables the hardware to load and store the registers held in the save area from data on word boundaries. Incorrect alignment will not cause any problems since the hardware always adjusts (forces) the Dynamic Space Pointer to a word boundary before accessing a stack frame.
7) Central Memory Data Accessed by the IOU	- The IOU can only reference central memory words. Hence, it would require some special code in PP's to decode data not stored on word boundaries. This is really a pragmatic software convention since a PP has no way to specify a central memory address other than on a word boundary.

4.5.1.4.2 OTHER BOUNDARIES

The following data must be aligned on boundaries other than 64-bit word or 8-bit byte.

(1) Exchange Packages - 128-bit (2 word) Boundaries

To optimize the performance of the exchange jump on some processors, the hardware addresses two words at one time. Results will be unpredictable if the exchange package is incorrectly aligned.

(2) Instructions - Parcel (2-byte) Boundaries

Instructions, which are either 16-bit or 32-bit quantities, must be aligned on parcel boundaries. Failures to do this will either result in unpredictable behavior, or an Address Specification will be detected.

(3) Page Table - Page Table Length Boundary
To minimize the time needed to translate addresses from virtual to real, the hardware catenates (rather than adds)

the Page Table Addresses (PTA) to the page table index. For the catenation to yield the correct address, the low-order bits of the PTA, as determined by the page table length, must be zero. Failure to structure the PTA in this manner will cause the address translate mechanism to fail.

4.5.2 HARDWARE PERFORMANCE

Whereas the previous section dealt with conventions necessary to make the hardware work correctly, this section deals with conventions necessary to make the hardware work efficiently. As such they are not mandatory, and in some cases represent merely suggestions as to how to optimize certain functions.

4.5.2.1 Locality_of_Reference

Convention: Place all code and all data to be used at one time in one place, and keep to a minimum the number of segments required to execute a given task.

The CYBER 180 virtual memory organization provides the basis for the system security and simplifies the explicit organization of a program into overlays. However, all programmers have responsibilities if system throughput is to be optimized. A prime responsibility is to maintain a strict locality of reference. That is collect all code and all data that is to be used at one time into contiguous pages in one segment (each for code and data). This has two advantages, it minimizes the working set (the number of pages allocated in real memory at any given point of time), and it also minimizes the number of entries which must be made in the buffer memories. By minimizing the working set the number of concurrent tasks which can be held in real memory is maximized. This, in turn, maximizes system throughput.

Optimizing around the buffer memories represent a slightly different problem. These have a finite size and contain the most recently used Segment Descriptor Entries and Page Table Entries. If a large number of segments are in use at one time, or if a large number of pages are in use at one time, then the buffer memories will be unable to hold all the necessary entries and they will be constantly loading new values. The affect will be similar to not having them at all and performance will degrade considerably.

Consequently, not only should programmers maintain a locality of reference, but they should also try to localize the number of segments used by a given task.

4.5.2.2 Register Allocation and Usage

Convention: Allocate A-Registers and X-Registers from the small numbers on up.

As a result of the special functions for which A0-A4 and X0-X1 are used, and the method of saving/restoring contiguous registers by the CALL/RETURN instructions, register usage should always start with the smallest possible number (typically A5 and X2). This will help to minimize the number of registers which must be saved across procedure calls. This, in turn, will optimize performance in this area.

4.5.3 SECURITY

This section lists software conventions needed to provide a secure environment at all times. Since a major objective of the CYBER 180 program is to provide a highly secure system, these conventions become mandatory. These conventions are closely related to those in Section 2. Just as they are required to make the hardware operate in a correct, predictable manner, so are these required to guarantee that the security and protection algorithms function correctly.

4.5.3.1 Procedure Parameters

Convention:

1) Always use caller's argument list pointers for accessing caller's data.

2) Always load pointer parameters directly into A-Registers - via Load A instructions.

3) Whenever possible avoid moving record structures that contain pointers.

4) Avoid passing pointers between rings either way.

5) Avoid data structures containing direct pointers that cross rings either way.

These conventions are mandatory for those procedure calls from one procedure to a second one with more privilege.

When a procedure is called by another procedure, it executes on behalf of the caller. It is the responsibility of the callee to ensure that it does not execute with more privilege than caller. The hardware provides the basic security mechanisms. In this case, it ensures that callee is called from within its call ring bracket, and that it is called via a Binding Section. It may then access code and data belonging to or accessible by caller. This code and data is referenced via pointers held in A-Registers, and the hardware performs a ring number vote whenever an A-Register is loaded. This mechanism ensures the least privilege (highest ring number) is always accorded the user. However, there are many ways this mechanism can be be-passed. The simplest method is for callee to load a pointer into an X-Register, then copy it to an A-Register. If caller places a low ring number (zero would do) in the pointer, then it will end up with callee's ring number in the A-Register. That is it will end up with more privilege than that to which caller is entitled. It is callee's responsibility to ensure this does not happen. The onus for maintaining security always falls on the more privileged procedure. Hence, the convention.

4.6 SUPPORT OF EBCDIC DATA

EBCDIC data can be divided into two distinct classes:

1. all 8-bit character data (also known as coded data, including unpacked numeric data types); and

2. intermixed character and non-character data.

Support for the former (all character) is provided by the operating system. If EBCDIC is specified on the request card, the tape driver automatically translates to ASCII when reading the tape and translates back to EBCDIC when writing the tape.

Support for the latter (intermixed character and non-character), and for the EBCDIC collating sequence, varies by product:

	COBOL	PL/I	FORTRA	S/M	FMU	CRM	DMS180

EBCDIC SUPPORT		N		
Intermixed EBCDIC input file	e			X
Intermixed EBCDIC output file	e			X
EBCDIC collating sequence	X		X	X

X = support required at R1 of product
e = eventual support desirable

Support of intermixed input and output files means use of the special C180 instructions to process the following "translated" non-character EBCDIC data types:

- Binary (signed and unsigned)
- Packed Decimal (signed and unsigned)

4.7 KEYPOINT_USAGE

The CY180 keypoint facility provides a mechanism to enable collection of statistics for performance monitoring. A data reduction software package is available to summarize these statistics based on descriptors contained in a keypoint descriptor file (KDF). This section documents the conventions to be followed by the operating system and product set in the usage of this facility.

4.7.1 KEYPOINT CLASSES

Five keypoint classes named ENTRY, EXIT, UNUSUAL, DEBUG, and DATA are defined for the operating system and product set.

ENTRY
: Every gated procedure plus all major internal procedures (those shared across functional areas) should contain a keypoint of this class. These keypoints should be placed as close as possible to the entry to the procedure.

EXIT
: Every gated procedure plus all major internal procedures (those shared across functional areas) should contain a keypoint of this class. These keypoints should be placed as close as possible to the exit from the procedure.

UNUSUAL — Every situation which is unexpected or quite unusual should contain a keypoint of this class. It is intended that these keypoints would be enabled at all times. The frequency of encountering these keypoints should be very low. The DATA keypoint class is not allowed in conjunction with a keypoint of class unusual.

DEBUG — These keypoints would be for providing additional trace information as an assist in debugging of hardware or software problems. DEBUG class keypoints would be most useful in the more complex areas of the system. The primary use of keypoints in HCS and NOS/VE up to this point has been for debugging purposes.

DATA — This keypoint class can be used with the ENTRY, EXIT, and DEBUG keypoints for the gathering of extra data. All DATA keypoints encountered are supplying additional data which will be associated with the last ENTRY, EXIT or DEBUG keypoint. Hence, they should follow as closely as possible after the ENTRY, EXIT, or DEBUG keypoint; in particular, there should be no intervening CALL instruction. DATA keypoints should be used with care since the PMF hardware can only buffer up 16 keypoints; keypoint clustering can cause lost keypoints.

Keypoint Data and Identification:

Upon successful execution each keypoint instruction will provide a total of 32 bits of information. The convention uses 12 bits of this for keypoint identification and the remaining 20 bits as user supplied data. Try to use this 20 bits to provide meaningful information (taskid, segment number, fileid, queue length, page number, time, etc.). On DATA class keypoints the data belongs to the previous keypoint and the full 32 bits is available for additional user data.

4.7.1.1 Operating_System

The keypoint classes for NOS/VE are as follows:

OSC$DATA=0
OSC$UNUSUAL=1
OSC$ENTRY=2
OSC$EXIT=3
OSC$DEBUG=4

Keypoint class 5 is reserved for NOS/VE.

The operating system keypoint multiplier is OSK$M.

4.7.1.2 Product_Set

The keypoint classes for the product set are as follows:

PSC$DATA=6
PSC$UNUSUAL=7
PSC$ENTRY=8
PSC$EXIT=9
PSC$DEBUG=10

The product set keypoint multiplier is PSK$M.

4.7.1.3 Other_Classes

The keypoint classes 11-14 are reserved for users.
Keypoint class 15 is reserved for PMF hardware control.

The keypoint multiplier for user defined keypoints is OSK$M.

4.7.2 KEYPOINT IDENTIFIERS

A maximum of 65535 keypoint identifiers are available for (each) NOS/VE and the product set. The combination of keypoint class and identifier is unique within the system.

4.7.2.1 Operating_System

The set of 4096 available identifiers is assigned to operating system areas in blocks of 50. Some areas (if needed) will receive two consecutive blocks of 50. The NOS/VE performance project has responsibility for assigning the ranges to areas of the operating system.

The currently assigned values for the operating system are:

Area Identifier		Range
(not used)		0 - 49
AM	Access Method	50 - 149
BA	Basic Access Method	150 - 249
CL	Command Language	250 - 299
CM	Configuration Management	300 - 349
(future expansion)		
DM	Device Management	400 - 549
(future expansion)		
IC	Interstate Communications	600 - 649
IF	Interactive Facility	650 - 699
II	Interactive Interface	700 - 749
(future expansion)		
JM	Job Management	800 - 849
LG	Logs	850 - 899
(future expansion)		
LO	Loader	950 - 999
(future expansion)		
ML	Memory Link	1050 - 1099
MM	Memory Management Monitor Mode	1100 - 1149
MM	Memory Management Job Mode	1150 - 1199
MS	Maintenance Services	1200 - 1249
MT	Monitor	1250 - 1299
OC	Object Code Utility	1300 - 1349
OF	Operator Facility	1350 - 1399
OS	Operating System	1400 - 1449
(future expansion)		
PF	Permanent Files	1500 - 1599
(future expansion)		
PM	Program Management	1600 - 1699
RH	Remote Host	1750 - 1799
SR	Conversion Services	1800 - 1849
ST	Software Tools/Set Management	1850 - 1899
TM	Task Management Monitor Mode	1900 - 1949
TM	Task Management Job Mode	1950 - 1999
JS	Job Swapper Monitor Mode	2000 - 2049
JS	Job Swapper Job Mode	2050 - 2099
AV	Accounting Validation	2100 - 2149
SF	Statistic Facility	2150 - 2199
IO	Input / Output	2200 - 2249
(future expansion)		
DM	Device Management/Tape	2300 - 2349
ST	System	2350 - 2399
NA	Network Access Method	2400 - 2499
NL	Network Access Method	2500 - 2549
NL	Network Access Method	2550 - 2599

	(future expansion)	
FM	File Management	2700 - 2799
FS	File System	2800 - 2899
RM	Resource Management	2900 - 2999
NA	Network Access / Monitor Mode	3000 - 3049
	(future expansion)	
MT	Monitor	4000 - 4049

The keypoint reduction utility and the continuous monitoring facility depend upon the following keypoint values. These routines should be modified to use the keypoint names and not the keypoint values listed below. This modification should be completed by NOS/VE release 1.2.1 (second quarter 1986).

4001 Enter / Exit Monitor Mode	mtk$job_entry_exit
4002 Enter / Exit NOS 170	mtk$170_entry_exit
4003 Monitor Mode Trap Handler	mtk$monitor_mode_trap
4004 Job Mode Trap Handler	mtk$job_mode_trap
2200 Series Physical I/O	
1106 Page Fault Processor	mmk$page_fault
1149 Convert PVA to SVA	mmk$system_virtual_address
1906 Queue Task	tmk$queue_task
1918 Switch Task	tmk$switch_task

4.7.2.2 Product_Set

The set of 65535 available identifiers is assigned to products in blocks of 50. Those product set members which require more than 50 will be assigned one or more additional blocks.

Assigned ranges are:

Product Identifier		Range
(invalid)		0
AA	Advanced Access Method	1 - 49
AP	APL	50 - 99
BC	BASIC	100 - 149
CB	COBOL	150 - 199
DB	Interactive Debug	200 - 249
FC	FORTRAN Compiler	250 - 299
FL	Fortran Run-Time	300 - 349
FM	File Management Utility	350 - 399
FT	FORTRAN Global	400 - 449
IM	Information Management	450 - 499
PA	Pascal	500 - 549
P1	PL/I	550 - 559
QU	Query Update	600 - 649
SM	Sort/Merge	650 - 699
SV	Shared Variables Processor	700 - 749
CC	Common Compiler Modules	750 - 799
CG	Common Code Generator	800 - 849
(future product)		850 - 899
ML	Math Library	900 - 949
CY	CYBIL	950 - 999
SC	Source Code Utility	1000 - 1049
AL	Assembler	1050 - 1099
FA	File Migration Aids	1100 - 1149
LI	LISP	1150 - 1199
AD	Ada	1200 - 1249
FV	CDC Fortran	1250 - 1299
VX	VX/VE	1300 - 1349
VC	C compiler	1350 - 1399
PT	Performance Tools	1400 - 1449
KR	Keypoint Reporting Utility	1450 - 1499
NF	Network File Transfer	1500 - 1549
(Reserved for future products)		1550 - 1999
AA	Advanced Access (2nd block)	2000 - 2049
(Reserved for future products)		2050 - 65535

4.7.3 KEYPOINT USE

From a software point of view, keypoints are special commands that are inserted in a module according to the guidelines specified in section 4.7.1. For a module written in CYBIL, the #KEYPOINT intrinsic can be used to

generate the keypoint instruction (refer to CYBIL Language Specification, ARH2298, and MIGDS, ARH1700, for details).

The main entry keypoint identifying a product set member should include data which indicates the actual version of the product. This is useful for tracking simultaneous execution of the same or different versions of a product.

5.0 COMPILER AND ASSEMBLY CODE CONVENTIONS

This standard is to be followed by the object code generated by the compilers and by any assembler code written as part of standard software.

In addition to these standards, assembler code (handwritten or compiler generated) will conform to the coding standards described in CYBER 180 MAINTENANCE SOFTWARE CODING CONVENTIONS (DAP ARH2160).

5.1 USE OF LOADER FEATURES

1. The loader specification is limited to that written in its formal documentation. Programmers shall not depend on additional characteristics determined by empirical observation, as such behavior may be subject to change. Examples which have caused trouble on CY170 are the presetting of undefined variables, the order of loading from a library, and the address at which the first code is loaded.

2. Runtime routines shall not limit the program structures of their users. On CY170 all CRM 1 routines must be in the root segment of a segmented load, and CMM must have at least one routine in the main overlay of an overlaid program. Such restrictions must be avoided on CY180.

3. The following table shows in which sections particular types of data should be allocated, and the attributes the section should have.

   Attributes R = read, W = write, B = Binding and E = execute.

Data Type	Section Type	Att	Comment and Examples
"Static"	Working	R,W	All variables not allocated on the stack, in common or explicitly allocated to a section. Includes FORTRAN local variables, CYBIL [STATIC] and [XDCL] variables.
Constants(1)	Working	R	All literal constants

			which for reason of indirect addressing or length cannot be expressed directly in the code.
Constants(2)	Code	E	Optionally, constants as in (1) which are less than 8 bytes long and conveniently accessed through the LBYTP instruction. Note that the "constant" may not be a PVA.
"XREF"	Binding	B	Data declared in another unit of compilation are usually referenced through pointers placed in the binding section by the loader (rather than in user sections indirectly referenced through the binding section, where they would be inaccessible to the binder).
Heaps	common extensible	R,W	For the system heap see section 5.4.3. Other heaps are declared in CYBIL.

4. The following action should be taken if a compiler detects a fatal error in the source code it is compiling, unless the compiler was called with "DEBUG=OC" (see section 2.2):

   An IDR record shall be issued containing the string

   "errors in compilation"

   in the comment field. The non-executable attribute shall be set.

   If DEBUG=OC was selected, the compiler shall continue normal processing as far as possible.

5. All compilers should emit loader names (common block

names, XREF names, module names, etc.) using upper case alphabetic letters when letters occur in the names. An exception to this rule is made for any language which requires the distinction between upper and lower case names.

5.2 INTERLANGUAGE CALLING SEQUENCES

Purpose

The purpose of the interlanguage calling sequence is to facilitate inter-language procedure calls. This is particularly desirable on CYBER 180 because of the system level support for sharing of code between executing tasks. For example, it would be desirable to have only one set of mathematical routines to be used by all languages.

Restrictions

All CYBER 180 Compilers must be capable of generating the CYBER 180 Interlanguage Calling Sequence for an externally referenceable code module. It is a goal in the definition of this calling sequence that it be useable by the majority of the compilers as a subset of their standard calling sequence. It obviously cannot meet all of the needs of languages as diverse as BASIC and PL/I. It would be acceptable (but certainly not preferable) if a particular language were to require special declarations or attributes on a procedure call to cause the generation of this calling sequence.

It is expected that users in the various programming languages may have to take additional steps with respect to data declarations to guarantee that the alignment and packing correspond to that specified by this interchange standard. The user is also responsible for the values passed via this calling sequence. For example, a Boolean variable might contain values 0-7 (since it occupies a byte) but the common calling sequence only assures interlanguage capability for the values 0 and 1.
In general, a compiler may employ any calling sequence it chooses between itself and its library or non-external procedures. Exceptions to this will be for routines which can be of general use to many languages (e.g., math library routines). Such routines may have a fast calling sequence but must also provide an entry point conforming to the interlanguage calling sequence.

5.2.1 CALLING SEQUENCE FORMATS

The interlanguage calling sequence is defined to include not only the layout of the parameter list, but also the layout of any descriptors associated with parameters in the list. Two formats for the interlanguage calling sequence are available. The term "interlanguage calling sequence" is used to refer to these two formats collectively. Two different formats are required in order to provide flexibility of usage from language to language while not unreasonably degrading performance and usability. These two formats will be referred to as the "System" and "General" formats. Extensions to either of these formats may be made via a DAP against the SIS.

The calling sequence provided by a compiler for use between internal procedures and functions known to be written in the same language need not conform to either format of the interlanguage calling sequence. Additionally there is no requirement to use the interlanguage calling sequence between compiler generated procedures and functions and any assembler procedures and functions provided in a runtime library specific to that language. In general, assembler procedures and functions are responsible for accepting a parameter list format of the kind generated by their potential callers. However calls to the scalar CMML call-by-reference procedures and functions must conform to the System format, while calls to the vector/array CMML call-by-reference procedures and functions must conform to the General format.

5.2.1.1 Kinds of Parameters

For purposes of exposition, six kinds of parameters will be distinguished: simple value parameters, extended value parameters, simple reference parameters, extended reference parameters, simple bit reference parameters, and extended bit reference parameters.

Value parameters are those parameters for which a value is intended to be passed. The calling program can assume that the actual argument it passes will not be changed by the called program. Note that this does not imply a specific implementation technique (several are possible). Some value parameters also require that certain descriptor information must be passed along with the value.

Simple value parameters are those value parameters which require only a value to be passed to the called routine.

Extended value parameters are those value parameters which are composed of a value plus a descriptor. Included in this category are pointers-to-procedure when they are accompanied by a static link.

Reference parameters are those parameters for which an object is intended to be passed. The calling program must assume that the actual argument it passes may be changed by the called program. Note that this does not imply a specific implementation technique, although at least an address must normally be passed. Some reference parameters also require that certain descriptor information must be passed along with the address.

Simple reference parameters are those reference parameters which require only an address, or only an address plus a string descriptor, to be passed to the calling routine.

Extended reference parameters are those reference parameters which are composed of an address plus a string descriptor plus a non-string descriptor, or of an address plus a non-string descriptor.

Simple bit reference parameters are those reference parameters which require only an address plus a bit string descriptor plus a bit string offset to be passed to the calling routine.

Extended bit reference parameters are those reference parameters which are composed of an address plus a bit string descriptor plus a bit string offset plus a non-string descriptor.

5.2.1.2 System_Format_of_the_Interlanguage_Calling_Sequence

This format is the one used by the system implementation language (CYBIL), and all operating system interfaces. This format is documented in detail in section 5.2.5.1 of the SIS.

5.2.1.3 General_Format_of_the_Interlanguage_Calling_Sequence

This format is more general than the system format. It will be used by Ada and CDC FORTRAN. This format is documented in detail in section 5.2.5.2 of the SIS.

5.2.1.4 Summary_of_Format_Differences

The primary difference between the System and General formats is in the placement and content of descriptors. System format and General format actual parameter lists are identical if only simple reference parameters are passed. All System format descriptors are placed directly in the parameter list following the PVA of the object being described, while General format non-string descriptors are placed outside the parameter list. The General format parameter list contains the PVA of the descriptor as well as the PVA of the object being described.

The System format does not support extended value parameters except for pointers-to-procedure. For simple value parameters, the System Format and the General Format are identical except when the value parameter is less than one word in size. The General format requires that the value parameter be right aligned with sign fill on the left for integers and subranges of integers and zero fill otherwise, while the System format requires right alignment but does not define the fill bits on the left.

Use of the General format of the Interlanguage calling sequence requires that a "big" (i.e. longer than a word) value parameter which is passed via a pointer will have been copied by the caller. The passed pointer is a pointer to the copy, and the called program is free to write into the memory pointed to. The System format does not specify whether or not a "big" value parameter will have been copied by the caller, so in this case the called program should not write into the memory pointed to.

5.2.1.5 Calls Potentially from Another Language

Any procedure or function which is intended to be callable from an external module potentially written in another language should accept for that call one (or a subset of one) of the two formats of the Interlanguage calling sequence. Each compiler must document which of the two sequence formats it accepts, or state that none of its procedures and functions are externally callable from another language.

Language	Interlanguage Format Accepted
ADA	General Format
BASIC	-not interlanguage callable-
C	-to be determined-
COBOL	System format
CYBIL	System format
FORTRAN	General format
Pascal	-not interlanguage callable-

5.2.1.6 Calls Potentially to Another Language

A compiler may assume that no call it generates is an interlanguage call unless the author of the source program has explicitly indicated that a particular call is interlanguage. This means that each language which supports calls to modules written in another language must provide a mechanism within the source language with which the author of the source program can explicitly indicate that a particular call is interlanguage. This mechanism must be

formulated such that the author is further required to state explicitly (by name) which other language is being called. It is then up to the compiler to generate the correct interlanguage calling sequence for the call. Thus the compiler must know which languages accept which calling sequences. It remains the responsibility of the author, not the compiler, to ensure that the actual and formal parameters of the call are compatible. The compiler has the responsibility to generate the correct layout for the parameter list and parameter descriptors, as expected by the called language.

These provisions do not require a compiler or language to provide interlanguage calls, but they do define restrictions on how interlanguage calling is to be supported. A language may support interlanguage calls to only a limited number of other languages, if it so chooses. Note that even if a language supports direct interlanguage calls, it is not required to also support indirect interlanguage calls via dereferenced pointers-to-procedure.

5.2.1.6.1 SUPPORT FOR CALLS TO ANOTHER LANGUAGE

If a language supports calls to modules written in another language, and that other language accepts calls with simple reference parameters, then the calling language must, at the minimum, support calls with simple reference parameters. A string descriptor must be supplied for any object which takes one, unless the author of the calling program has explicitly indicated that no string descriptor need be passed. An explicit indication is possible in languages, such as CYBIL, which allow the reference parameter in an external procedure declaration to be specified as either fixed type (descriptor need not be passed) or adaptable type (descriptor must be passed).

The calling language is strongly encouraged to also provide support for calls with value parameters and extended reference parameters if the called language accepts such calls. This support would consist of a mechanism within the source language to explicitly indicate, for each actual parameter of the interlanguage call, whether the parameter is to be passed by value, by simple reference, or by extended reference. The compiler then has the responsibility to generate the appropriate calling sequence.

5.2.2 CALL

The procedure call instruction CALLSEG, Reference #115 as defined in the CYBER 180 MIGDS will be used to perform the procedure call.

5.2.3 REGISTER SAVING CONVENTIONS

For generalized external calls and calls to formal procedures, the compiler may not assume that the called procedure will save and restore registers. Any registers to be saved must be saved on the stack using the save mechanism of the CALL instruction.

Internal calls need not use the CALLSEG, Reference #115 instruction. They may use CALLREL Reference #116 or any other code sequence which meets their needs. For internal calls the compilers have the option whether to save registers or not. Internal calls include calls to:

a) the compiler's own library routines,

b) nested procedures within the same compilation unit,

5.2.3.1 Information Required Across Call

The following information may be required in making a call. Some of the information is not always required - See footnotes.

Dynamic to Caller and Callee

- basic stack control registers (A0, A1, A2)***
- parameter list pointer (A4)***
- static chain/display*
- binding section pointer (A3)***
- product defined information

Dynamic to Callee, Static to Caller

- line number of call (see traceback section)***
- number of parameters(X0, bits 32-47)***
- descriptor area indicator
- descriptor area pointer (if any)

Static to Caller and Callee

- name of callee (see traceback section)

- size of display/nesting depth*,**
- frame size/language**
- type of frame; e.g. proc, func, co-proc**

* Block structured languages only.
** Traceback mode only.
*** Required on calls made with the interlanguage calling sequence.

5.2.4 FUNCTIONS

A function is a procedure that returns a value. The function value is in the registers or in memory depending on the type of value being returned. Since function references are usually part of another expression that is being evaluated, it is generally desirable to have the value returned in a register.

If the function value is a pointer, then the value is returned as a PVA in AF. A procedure calling a pointer-valued function must not save register AF on the call. A pointer-valued function may have the ring number field of AF altered by the RETURN instruction if it is called across a ring boundary.

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 XF. A procedure calling such a function must not save register XF on the call.

If the function value is double precision or complex then the value is returned in registers XE and XF. XF holds the least significant 64 bits of the value. A procedure calling such a function must not save XE or XF on the call.

If the function value is non-scalar then it is stored at the address defined by the first element of the parameter list. The second element of the parameter list specifies the first actual parameter.

A scalar function result is defined as follows:

- CYBIL - character, boolean, integer, ordinals, subranges, cell, pointer.
- FORTRAN - logical, integer, real, double precision, complex, FORTRAN boolean.*

- COBOL - comp, comp-1, comp-2, boolean.
- PL/I - integer(FIXED REAL), real(FLOAT REAL), complex(COMPLEX)
- BASIC - real.
- Pascal - integer, (enumerated type, sub-range), real

Scalar function values are returned right aligned in the result register. Fill (if any) is zero bits. Note that 8 byte numeric items require no fill.

* FORTRAN boolean corresponds to a full CYBER 180 word without type. It is not the same as the boolean type mentioned elsewhere in this section.

5.2.5 PARAMETER LIST

The parameter list is allocated on a word boundary in memory. Each entry in the parameter list must also begin on a word boundary. On entry to the callee, register A4 will point to the parameter list. Bits 32-47 of register X0 will contain the number of parameters (including the pseudo parameter for non-scalar valued functions). If the procedure being called is a function whose value is to be returned in memory, the first element of the parameter list defines the location at which the value is to be stored. If no parameters (nor pseudo parameters) are to be passed, then the contents of A4 are undefined and X0 must specify zero parameters. Under certain circumstances detailed below, a flag word must immediately precede the first word of the parameter list.

5.2.5.1 System_Format_Parameter_List

[This is currently documented in the CYBIL Handbook, DCS# ARH3078, sections "CYBIL CI/II TYPE AND VARIABLE MAPPING" (old section 7.1) and "RUN TIME ENVIRONMENT" subsection "PARAMETER PASSAGE" (old 8.3). The following addition must be made to that documentation in order to conform to the SIS.]

For any potentially interlanguage call in which a System format actual parameter list is passed that contains only simple reference parameters: The parameter list must be immediately preceded by a flag word whose value is the 64-bit integer zero. The string descriptor must be included for any object which takes one, unless the author of the source program has explicitly indicated that it need not be passed. These restrictions are made to insure compatibility between the release 1.1.2 product set calling

conventions and those for all future releases. A flag word need not precede any other System format actual parameter lists.

5.2.5.2 General Format Parameter List

The General format parameter list must always be preceded by a flag word. The parameter list itself is composed of two parts. The first part has exactly one word for each parameter (including the pseudo parameter for non-scalar valued functions). If the flag word preceding the parameter list is zero then only this first part is present, otherwise the second (extension) part must also be present. This parameter list extension follows immediately after the first part of the parameter list, and has exactly the same length in words. There is a one-to-one correspondence between word j of the first part and word j of the extension.

The parameter list extension is required if and only if one or more of the actual parameters is an extended value parameter or an extended reference parameter, or a (simple or extended) bit reference parameter. !

5.2.5.2.1 FLAG WORD PRECEDING PARAMETER LIST

The flag word immediately preceding a General format actual parameter list must be present for any potentially interlanguage call. This flag word has the following internal structure:

```
record
  f1: 0..0ffffffffff(16),
  f2: 0..0ff(16),
  f3: 0..0ff(16),
recend
```

Field f1 must always be set to integer zero. It is reserved for future uses. Field f2 has a language dependent value, but may be nonzero only if field f3 is nonzero. Field f3 must be set to integer zero if the parameter list extension is absent, and must be set to integer one otherwise. Any language accepting calls according to the General format must accept interlanguage calls for which field f2 is zero. An interlanguage caller will never be required to set field f2 to a non-zero value. If field f2 is set to a non-zero value for an interlanguage call, it is the responsibility of the caller to set the field according to the expectations of the callee.

5.2.5.2.2 GENERAL FORMAT SIMPLE VALUE PARAMETERS

If a simple value parameter is greater than one word in length and is not a pointer-to-procedure, then it is passed using an identical format to that for a reference parameter.

If a simple value parameter is a pointer-to-procedure then the first part of that parameter list entry must contain the left justified PVA of the Code Base Pointer of the procedure in the binding section. The second part of the entry (when an extension is required) must contain the NIL pointer. The 16 bits to the right of each of these PVAs is unused and undefined. This can be diagrammed as:

```
+-------------------------+        +-------------------------+
! PVA (Code Base) ! undef !        ! NIL              ! undef  !
+-------------------------+ _ _ _  +-------------------------+
```

If a simple value parameter is less than or equal to a word in length, then a copy of the value parameter is placed directly in the first part of the parameter list right aligned in a word, with sign fill on the left for integers and subranges of integers and zero fill otherwise. The associated word in the second part (when an extension is required) is unused and undefined. Note that if a PVA having no associated descriptor is passed by value, then by this rule the PVA is placed directly in the parameter list, right aligned in a word, with the word zero-filled on the left. This can be diagrammed as:

```
+-------------------------+        +-------------------------+
! value (right justified) !        !        undefined        !
+-------------------------+ _ _ _  +-------------------------+
```

5.2.5.2.3 GENERAL FORMAT EXTENDED VALUE PARAMETERS

If an extended value parameter is greater than one word in length (excluding the descriptor) and is not a pointer-to-procedure, then it is passed using an identical format to that for a reference parameter. Use of extended value parameters requires that field number three of the flag word preceding the parameter list must have been set to one.

If an extended value parameter is a pointer-to-procedure, then the first part of that parameter list entry must contain the left justified PVA of the Code Base Pointer of the procedure in the binding section. The second part of the entry must contain the left justified PVA of the static link. The 16 bits to the right of each of these PVAs is unused and undefined. This can be diagrammed as:

```
+-------------------------+        +-------------------------+
!  PVA (Code Base) ! undef!        !   Static Link    ! undef!
```

```
+-------------------------+ _ _ _ +-------------------------+
```

If an extended value parameter is less than or equal to a word in length, then a copy of the value parameter is placed directly in the first part of the parameter list right aligned in a word, with sign fill on the left for integers and subranges of integers, and zero fill otherwise. The associated word in the parameter list extension for this entry will contain the PVA of a location (which must be on a word boundary) in memory where the descriptor is located. The PVA in the parameter list extension is left aligned in a word with the rightmost 16 bits being unused and undefined. This can be diagrammed as:

```
+-------------------------+       +-------------------------+
! Value (right justified) !       ! PVA (descriptor)! undef !
+-------------------------+ _ _ _ +-------------------------+
```

5.2.5.2.4 GENERAL FORMAT SIMPLE REFERENCE PARAMETERS

Simple reference parameters are passed either as a PVA or as a PVA plus string descriptor. Parameters consisting solely of a PVA are placed directly in the first part of the parameter list entry left aligned in a word; with the rightmost 16 bits of the word unused and undefined. The value of the word in the associated second part (if an extension is required) must be the 64-bit integer zero. This can be diagrammed as:

```
+-----------------+------+       +-------------------------+
! PVA (object)    ! undef!       !            0            !
+-----------------+------+ _ _ _ +-------------------------+
```

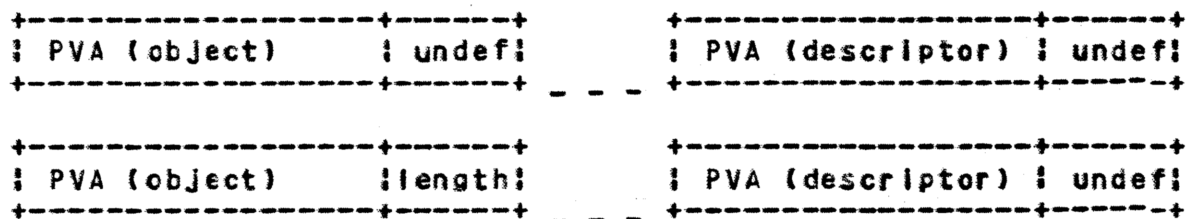
Simple reference parameters consisting solely of a PVA plus a string descriptor are placed directly in the first part of the parameter list entry with the PVA left aligned in a word, followed immediately by the two byte long string descriptor. The value of the word in the associated second part (if an extension is required) must be the 64-bit integer zero. This can be diagrammed as:

```
+-----------------+------+       +-------------------------+
! PVA (object)    !length!       !            0            !
+-----------------+------+ _ _ _ +-------------------------+
```

5.2.5.2.5 GENERAL FORMAT EXTENDED REFERENCE PARAMETERS

Extended reference parameters require that the non-string descriptor be passed indirectly using the parameter list extension, regardless

of the size of that descriptor. Field f3 of the flag word preceding the parameter list must have been set to one. The first part of the parameter list entry will contain the PVA of the object referenced, left aligned. If the reference includes a string descriptor then that descriptor is placed in the 16 bits immediately following the PVA, otherwise those 16 bits are unused and undefined. The parameter list extension for this entry will contain the PVA of a location (which must be on a word boundary) in memory where the descriptor is located. The PVA in the parameter list extension is left aligned in a word with the rightmost 16 bits being unused and undefined. This can be diagrammed as one of:



5.2.5.2.6 GENERAL FORMAT SIMPLE BIT REFERENCE PARAMETERS

Simple bit reference parameters are passed as a PVA plus a bit string descriptor plus a bit string offset. The PVA and bit string descriptor are placed directly in the first part of the parameter list entry with the PVA left aligned in a word, followed immediately by the two-byte long bit string descriptor. The value of the word in the associated second part (an extension is always required) consists of a left aligned 48-bit integer zero, followed by the two-byte long bit string offset. This can be diagrammed as:

5.2.5.2.7 GENERAL FORMAT EXTENDED BIT REFERENCE PARAMETERS

Extended bit reference parameters require that the non-string descriptor be passed indirectly using the parameter list extension, regardless of the size of that descriptor. Field f3 of the flag word preceding the parameter list must have been set to one. The first part of the paramater list entry will contain the PVA of the object referenced, left aligned, with a bit string descriptor placed in the 16 bits immediately following the PVA. The parameter list extension for this entry will contain the PVA of a location (which must be on a word boundary) in memory where the descrriptor is located. The PVA in this parameter list extension is left aligned in a word, followed by the

two-byte long bit string offset. This can be diagrammed as:

```
+-------------------+------+       +------------------+------+
| PVA (object)      |length|       | PVA (descriptor) |offset|
+-------------------+------+ - - - +------------------+------+
```

5.2.5.2.8 GENERAL FORMAT STRING DESCRIPTORS

A string descriptor is a 16-bit unsigned integer (0..65535) indicating the length of a string in bytes. When present, it is placed in the primary portion of the parameter list immediately following (and in the same word as) the PVA of the object being described. A string descriptor is required for all reference parameters to objects of type character, subrange of character, string, substring, or array over a component type of character, subrange of character, string, or substring. The string descriptor for an array indicates the length in bytes of a single element.

5.2.5.2.9 GENERAL FORMAT BIT STRING DESCRIPTORS

A bit string descriptor is a 16-bit unsigned integer (00..65535) indicating the length of a bit string in bytes. When present, it is placed in the primary portion of the parameter list immediately following (and in the same word as) the PVA of the object being described. A bit string descriptor is required for all reference parameters to the objects of type bit, bit string, bit substring, or array over a component type of bit, bit string, bit substring. The bit string descriptor for an array indicates the length in bits of a single element.

5.2.5.2.10 GENERAL FORMAT BIT STRING OFFSET

A bit string offset is a 16-bit unsigned integer with a value in the subrange 0..7. indicating the offset of a bit string, in bits, from a byte address. When present, it is placed right aligned in the extended portion of the parameter list. A bit string offset is required for all parameters to objects of type bit, bit string, bit substring, or array over a component type of bit, bit string, bit substring. The bit string offset for anarray indicates the offset of the first element of the array from a byte address.

5.2.5.2.11 GENERAL FORMAT ARRAY DESCRIPTORS

The layout of an array descriptor must adhere to the pseudo-CYBIL description given below. Note that "extent" refers to the number of

elements in a particular dimension, "stride" refers to the distance (measured in terms of array elements) between two consecutive elements of the same dimension, and "rank" refers to the number of dimensions in the array. Array descriptors must be aligned on a word boundary.

```
array_descriptor =  array [1 .. rank] of record
    extent: integer,
    stride: integer,
    lower_bound: integer,
recend;
```

5.2.5.2.11.1 Stride

For languages such as CYBIL and FORTRAN 77, arrays are represented and stored as contiguous objects; stride is a function solely of the extents. However the introduction of array sections in CDC FORTRAN necessitates that an explicit stride be passed in the parameter list since sections need not be contiguous in memory; they may have a non-unity increment in each dimension of the array, which must be included in the calculation of the stride. The stride value for multi-dimensional arrays is calculated differently depending upon whether arrays are stored columnwise or rowwise. For one dimensional arrays the formulas are equivalent. Note that one dimensional contiguous arrays have a stride of one.

For arrays which are stored columnwise in memory (i.e. with the leftmost subscript varying fastest) the following formula is used:

$$\mathrm{stride}(i) = \mathrm{incr}(i) * \prod_{j=0}^{i-1} E(j)$$

where stride(i) is the stride in the i-th dimension, incr(i) is the increment of the i-th dimension, and E(0) is defined to be one. For contiguous arrays, E(j) is the extent of the j-th dimension. For array sections, E(j) is the extent of the j-th dimension of the contiguous array of which this is a section. For example if we have the FORTRAN declaration:

```
      DIMENSION C(15,30)
```

then for C we have: incr(1)=1, incr(2)=1, extent(1)=15, extent(2)=30, E(1)=15, E(2)=30, stride(1)=1, and stride(2)=15. For the section:

```
      C(1:10:2, 12:22:3)
```

we have: incr(1)=2, incr(2)=3, extent(1)=5, extent(2)=4, E(1)=15,

E(2)=30, stride(1)=2*1=2, and stride(2)=3*1*15=45.

For arrays which are stored rowwise in memory (i.e. with the rightmost subscript varying fastest) the following formula is used:

$$\text{stride}(i) = \text{incr}(i) * \prod_{j=i+1}^{r+1} E(j)$$

where stride(i) is the stride in the i-th dimension, incr(i) is the increment of the i-th dimension, r is the rank of the array, and E(r+1) is defined to be one. For contiguous arrays, E(j) is the extent of the j-th dimension. For array sections, E(j) is the extent of the j-th dimension of the contiguous array of which this is a section. For example if we have the FORTRAN declaration:

```
      ROWWISE R(15,30)
```

then for R we have: r=2, incr(1)=1, incr(2)=1, extent(1)=15, extent(2)=30, E(1)=15, E(2)=30, stride(1)=30, and stride(2)=1. For the section:

```
      R(1:10:2, 12:22:3)
```

we have: r=2, incr(1)=2, incr(2)=3, extent(1)=5, extent(2)=4, E(1)=15, E(2)=30, stride(1)=2*1*30=60, and stride(2)=3*1=3.

5.2.6 DATA REPRESENTATION

The following subsections define the representations of data which must be used if an item of a particular type is to be passed between languages. Languages may have types beyond these but data of those types cannot be passed to other languages. A language is not forced to provide for all of the following data types.

5.2.6.1 Integer

An integer may occupy 1 to 8 bytes of storage. For languages with size allocations dependent on the subrange of integers specified, the amount of storage allocated must be the minimum number of bits needed to hold the specified range rounded up to the next full byte. Subranges that include negative numbers must use the leftmost bit of the field as the sign bit. Negative values are represented as negative two's complement quantities. Subranges of only positive numbers will not provide a sign bit. The range of signed integers is -2**63 < i < 2**63-1. The range of unsigned integers is 0

< i < 2**63-1.

Several languages have an enumerated type called ordinals. These are mapped onto the non-negative integers. Allocation rules are the same as for unsigned integers. If ordinals are passed to a language without ordinals they must be treated as integer values and vice-versa.

Two sizes of integers correspond to easily manipulated hardware formats and are identified as separate subtypes of integer to provide for languages with only options for half or full word signed integer values.

5.2.6.1.1 4 BYTE INTEGER

A half integer will be represented by a 4 byte (32 bit) quantity in the CYBER 180 integer format i.e., a signed two's complement 32-bit quantity, in which the leftmost bit is the sign bit. The range of 4 byte integers is -2**31 < i < 2**31-1.

5.2.6.1.2 8 BYTE INTEGER

A full integer will be represented by an 8 byte (64 bit) quantity in the CYBER 180 integer format i.e., a signed two's complement 64-bit quantity, in which the leftmost bit is the sign bit. The range of 8 byte integers is -2**63 < i < 2**63-1.

5.2.6.2 Fixed_Length_Character_(String)

Fixed length character data will be stored as a sequence of consecutive 8 bit bytes. The character set will be ASCII.

5.2.6.3 Real

Real data will be represented by an 8 byte (64 bit) quantity in the CYBER single precision floating point format. All real data will be normalized.

5.2.6.4 Double_Precision

Double precision data will be represented by a 16 byte (128 bit) quantity in the CYBER 180 double precision floating point format. It must be normalized. The PVA in the parameter list points to the first byte of the double precision datum. The second (lower precision half) is

located at PVA+8 bytes. The sign and exponent fields of the lower part are considered to be correct at any given time. Input and constant assignment routines are responsible for insuring corrct signs and exponents upon initial construction of the number. Double precision operations will maintain this format.

5.2.6.5 Complex

Complex data will occupy 16 bytes (128 bits) in memory and will consist of two reals, where the first real represents the "real" part and the second real represents the "imaginary" part of the complex quantity. The PVA in the parameter list points to the first byte of the complex datum (the real part). The imaginery part is located at PVA+8 bytes.

5.2.6.6 Boolean

Boolean data occupies a single byte. A value of one indicates true and a value of zero indicates false.

5.2.6.7 Pointer

A pointer is a PVA. It occupies six bytes. Pointers may identify data of any of the other data types. The nil pointer is defined as a PVA with a ring field value of "F" hexadecimal, segment field value "FFF" hexadecimal, and address field value "80000000" hexadecimal.

5.2.7 DATA ALIGNMENT AND PACKING

The purpose of the common calling sequence is to provide the ability to pass data between diverse languages. The interlanguage call is assumed to represent a small percentage of all calls and generally be used by knowledgeable users. Therefore, for performance in the word oriented languages (FORTRAN, in particular) a least-common-denominator alignment of word is used.

Data types which require 8 bytes to store are required to be word aligned to improve performance. This permits the use of the load/store word instructions which are faster than load/store of 8 bytes. The space penalty for word aligning simple variables is felt to be small especially since it costs a maximum of 7 bytes per procedure if all the word aligned items are stored contiguously.

5.2.7.1 Variables

Variables may be of any of the above data types. The alignment of a particular type must be as follows:

Data Type	Alignment
1-7 Byte Integer	Byte
8 Byte Integer	Word
Character(String)	Byte
Real	Word
Double Precision	Word
Complex	Word
Boolean	Byte
Pointer	Byte

5.2.7.2 Structures

Structures must begin word aligned.

Alignment of data to be passed between languages in structures must be as follows:

Data Type	Alignment
1-7 Byte Integer	Byte
8 Byte Integer	Word
Character(String)	Byte
Real	Word
Double Precision	Word
Complex	Word
Boolean	Byte
Pointer	Byte

If a byte aligned item is followed by a word aligned item, up to seven bytes may be skipped (and left unused) to regain word alignment. If a byte item follows a byte item, they may be in consecutive bytes.

5.2.7.3 Arrays

5.2.7.3.1 ARRAYS OF VARIABLES

The arrays represent a collection of data items of one uniform type. Arrays must be word aligned if the data type they contain is word aligned. Unless required by an external standard all languages should store arrays with the rightmost subscript varying fastest. FORTRAN, for example, is constrained by ANSI standards to store arrays with the leftmost subscript varying fastest. If a user

passes a multidimensional array between languages with different storage orders, it is the user's responsibility to handle this. Arrays must be byte aligned if all of the constituent elements are byte aligned. The parameter list PVA identifies the first element of the array. Subsequent elements must be contiguous and in ascending storage address sequence.

5.2.7.3.2 ARRAYS OF STRUCTURES

If any element of the structure is required to be word aligned, each array element must start on a word boundary.

5.2.7.3.3 COMMON BLOCKS

Items within common blocks must be aligned consistently to achieve interlanguage communication. Common blocks will begin word aligned. Alignment of data within the common block will be the same as for structures.

5.2.8 LANGUAGE INTERCHANGE TABLE

The following table shows the possible parameter types that may be used between languages. If a letter appears at an intersection in the table, that type may be passed.

Types are encoded as follows:

- J = 1-3, 5-7 Byte Integer
- O = ordinal
- H = 4 Byte Integer
- I = 8 Byte Integer
- C = Character (string)
- R = Real
- D = Double Precision
- Z = Complex
- B = Boolean
- P = Pointer
- A = Array
- S = Structure
- All = all types of the language

caller \ callee	CYBIL	PASCAL	FORTRAN	COBOL	PL/I	BASIC
CYBIL	All	HIJCBPSAOR	ICARD	HICBSARD	HICBPSAR	CR
PASCAL	HIJCBPSAOR	ALL	ICA	HICBSA	HICBPSA	C
FORTRAN	ICARD	ICA	All	ICRDA	ICRDZA	CRA

COBOL	HICBSARD	HICBSA	ICRDA	All	HICBRDSA	CRA
PL/I	HICBPSAR	HICBPSA	ICRDZA	HICBRDSA	All	CRA
BASIC	CR	C	CRA	CRA	CRA	All

Notes:

1) PL/I may not have a double precision data type due to possible high overhead in supporting the maximum precision rules. This will be determined later.
2) If arrays are permitted between two languages, the type of the array is restricted to the types of variables that are permitted between the two languages.
3) Arrays of characters in BASIC cannot be passed to other languages, and vice versa.

5.2.8.1 Extended_Interchange

The language interchange table defines the parameter types that can be used between pairs of languages. In many cases restrictions exist because a particular language lacks a data type. For example, BASIC lacks integer type since it stores them as reals. In many instances the type mismatches could be mapped by interface code between the procedure calls. The following mechanism is proposed to support such mapping when and if it becomes a requirement.

In order to map parameters, an intercept routine must gain control from the caller, map things and pass control to the callee. The reverse may be necessary upon return. The user should not have to be aware of the activities of the interface routine or invoke it directly. To achieve this, the loader must have a mechanism for detecting the need for an interface routine and inserting same in the call/return path. The insertion mechanism can be similar to the one used for Analyze Program Dynamics (APD). Detection of the need for inserting the interface routine can be done with load time argument checking mechanisms.

For each pair of languages (X and Y) where interface mapping is desired, loader tables defining relevant information about actual and formal parameters must be defined. A routine (activated during loading by the loader if a call from X to Y is found) will compare the actual and formal parameter lists to determine if mapping is required. If not, the loader simply links as usual.

Otherwise, a X to Y mapping routine from a library is inserted into the linkage by the loader.

The X to Y mapping routine receives the actual and formal parameter list information from the loader.

The caller information is obtained by giving the P address of caller to a loader service routine which returns a PVA if the actual parameter list information for the current call. The callee information is obtained by giving the code base pointer of callee to a loader service routine.

The mapping routine uses this information to transform the parameter list and/or data representations before calling the callee. When the callee returns, the mapper will receive control to do any mapping on return parameters.

5.2.9 REGISTER CALL FUNCTIONS

In many languages there exist commonly used sets of functions (for example, mathematical functions) for which it is more efficient (though less general) to pass a limited set of parameter values via registers. Up to eight (64 bit values) can be passed in registers X2 - X9. The first parameter value would be in X2, the second in X3, etc. If a double word value (say, double precision) is required, it uses two consecutive registers. The specific register used for a routine may be inferred from the type of the parameter. For example, SQRT(X) will use X2 while DSQRT(D) will use X2 and X3. These rules apply to the following data types as parameters:

1-7 byte integers
8 byte integers
Real
Double Precision
Complex

Return registers for register call functions (see 5.2.3) must not be saved in calling them.

No rules are specified for character, boolean or pointer data pending identification of functions using these argument types that are of general utility.

The register call entry point is not bound by the conventions of the common calling sequence.

All register call functions intended for general use must

also offer an entry point that accepts the common calling sequence (5.2 above) and referenceable by a CALLSEG instruction.

5.3 INTERPRODUCT_FILE_USAGE

Interproduct file sharing between executing subsystems will be addressed. It will specify under what conditions a product will be able to perform I/O on a file declared by another product. It will also address closing and flushing of files at job step termination when interlanguage files are being used.

5.4 STORAGE_MANAGEMENT

Purpose

In order that user object code from different compilers can co-exist in one job step while using a limited number of segments, certain conventions must be observed.

Each user will have a limited number of segments. This means that object code from different compilers must be able to share certain data segments.

5.4.1 STANDARD STACK FRAME

This section describes the standard stack frame which will be set up in conjunction with the CALL instruction. The purpose of standardizing the stack frame layout is to provide common traceback and debugging interfaces. At the same time, allowance is made for a minimum frame for languages such as batch mode FORTRAN, with extensions for the complexity of languages such as PL/I.

A stack frame consists of two areas:

1. The save area.
2. The "environmental" area.

The save area belongs to the caller, the "environmental" area belongs to the callee and both exist in the appropriate rings.

5.4.1.1 Traceback

Traceback is considered to be the lowest level of debugging and as such requires the support of both the loader and the compilers/assembler. Minimum traceback information will always be produced to facilitate some tracing from within the system.

The compilers/assembler will produce traceback tables in the object module which correlate object-code address of entry points and calls with source-code procedure names and line numbers. The loader will maintain the relation of these object code addresses. When traceback is required, these traceback tables, plus the stack, will be interpreted to give the source-code names and line numbers associated with the PVAs obtained during traceback. In full traceback mode entries will exist for each line or source statement; in minimum traceback mode only entry points and calls are monitored.

5.4.1.2 Static Chain vs. Display

(See Glossary for definitions.)

It is not the intention of this standard to dictate whether compiled code will reference globals via the static chain or a display. Either is permitted and must be maintained by the software. Note: this only applies to calls to a nested procedure and hence is intralanguage.

5.4.2 CHAINS OF ON-CONDITION PROCESSORS

Software conventions for a standard on-condition processor chain format are required to ensure that on-conditions can be processed correctly.

The on-condition flag (OCF) in the save area is used to indicate that the stack frame has associated on-condition processors. The first eight bytes of the stack frame (pointed to by the current stack frame (CSF) of the save area) are reserved for the head of the on-condition processor chain. All object code generators must accommodate the head of chain reservation. If the OCF is set in the save area, the eight bytes pointed to by CSF is the head of the on-condition processor chain. If the OCF is not set, the contents of the eight bytes is undefined.

5.4.3 DYNAMIC NON-STACK STORAGE

5.4.3.1 Dynamic Segments

NOS/VE provides the capability of creating new segments during product execution. Since this increases the number of segments in active use and potentially causes a performance degradation, its use should be limited to situations where the alternatives are less satisfactory.

5.4.3.2 Fixed-Position Dynamic Storage

The fundamental support for fixed-position dynamic storage allocation is provided by the CYBIL ALLOCATE statement with no IN option.

Products coded in CYBIL and needing fixed-position dynamic storage should use the ALLOCATE statement directly. Products not coded in CYBIL and needing fixed-position dynamic storage may either:

1) include CYBIL subroutines containing the appropriate ALLOCATE statements, or

2) use a set of common routines which will provide a CMM compatible interface to the ALLOCATE statement.

5.4.3.3 Variable-Position Dynamic Storage

Variable-position dynamic storage is not currently planned for support.

5.5 COMMON SUPPORT MODULES

This section will define modules which will be available for general use.

Math Routines

For a detailed account of the math routines to be provided see C180 Common Modules Math Library (CMML) ERS with DCS log ID S2929. The routines will offer both a register calling sequence and the common calling sequence. Entry point names will meet the specifications of section 4.1.1.

Numeric Conversion Routines

5.0 COMPILER AND ASSEMBLY CODE CONVENTIONS
5.5 COMMON SUPPORT MODULES

Routines will be provided for all products (compiler or runtime systems) to perform numeric input and output conversion. This will ensure that the same numeric representation matches the same internal bit value by all processors. See also C180 common modules math library (CMML) ERS with DCS log ID S2929, and CMML Assembly-language Support System ERS with DCS log ID S3410.

FROM \ TO	Integer	Real	Longreal	ASCII	ASCII (nondec.)	BDP*	Unpacked trailing sign combination hollerith	Number-170
Integer				X	X			
Real				X(1)				
Longreal				X(1)				
ASCII	X	X(1)	X(1)	X(2)			X	
ASCII (nondec.)	X							
BDP*						X		
Unpacked decimal trailing sign combined hollerith				X				
Number 170	X	X						

*includes all BDP types except: alphanumeric

(1) there are additional routines for handling real and longreal conversions to and from ascii in piecemeal fashion

(2) translation, move, etc.

Utilities

A set of common utilities will be provided to carry out the following functions:

- Diagnostic Handling - the formatting of diagnostic lines of output and the construction of the diagnostic listings.

- Source listing formatting - the formatting of the source listing including output of the source lines to a print file.

- Storage map/Attribute/Cross Reference listings - the formatting of this listing and output of its contents to a print file.

- Compiler Usages Statistics - the generation of usage statistics messages.

6.0 GLOSSARY OF TERMS

In writing the System Interface Standard it became necessary to clarify the meaning of certain words. This glossary contains those words which required clarification. The list will be extended.

-a- adjective
-n- noun
-v- verb

Binary	-a-	Of base 2. Not to be used without qualification to mean the object code output from a compiler. Note object code files are one of many different forms of binary files.
Boolean	-n-	Data type which can hold the values "true" or "false".
FORTRAN Boolean	-n-	Boolean data but required to occupy a full computer word.
Diagnostic	-n-	Generally a part of a larger entity, such as listable output, as opposed to an error message, which is generally a summary of a command. Diagnostics are generally issued by a number of the product set, such as a compiler. See also - error message. Example: A compiler may provide a single error message telling how many errors occurred during compilation and produce a diagnostic for each compilation error.
Display	-n-	A mechanism for accessing global variables of a program using a table of stack frame pointers; one pointer for each accessible scope and one table for each active scope.
Error Message	-n-	Generally a summary of a command, as opposed to a diagnostic, which is generally a part of a larger entity, such as listable output. The error message is generally issued by the operating system or by a product via

		the operating system. See also - diagnostic for an example.
Invoke	-v-	Applies only to spirits, witches, etc. Procedures are called.
Job Step	-n-	A job step is the work done as a result of a single command in the job deck/file. Job steps execute sequentially within a job.
Load Module	-n-	Object information produced by object library generator and input to the loader or back into object library generator. Load modules are designed to facilitate processing by the loader.
Object Module	-n-	An object module is a unit containing code and/or data definition that is produced by compilers.
Object Program	-n-	An object program is a set of object modules organized to perform some specific function (e.g., compile COBOL statements). An object program is prepared for execution by the loader.
process(ing)	-v-	Comput(ing). Unrestricted to mean either hardware or software.
Processor	-n-	Restricted to hardware CPU or PPU. May be used for software if sufficiently qualified, e.g. language processor.
Product	-n-	Any part of the standard software which is covered by the System Inerface Standard.
Product Set	-n-	That part of the System which is not part of the Operating System.
record	-n-	A unit of data on a file. e.g. a card image, line image. Not to be used without qualification if meaning a "CYBIL" record or "SCL" record.
Standard	-n-	Plural-Standard not Standards when

6.0 GLOSSARY OF TERMS

		used in the sense of the System Interface standard.
Static chain	-n-	A mechanism for accessing global variables of a program using links through the stack frames.
System	-n-	All products (q.v.) operating as a whole - to be distinguished from Operating System.
Task	-n-	A task is an instance of execution of an object program. Multiple tasks can execute within a single job step. Each task has its own address space (set of memory segments). Tasks may be initiated either synchronously or asynchronously to the initiating task.

Table of Contents

1.0 GENERAL 1-1
1.1 PREFACE TO CURRENT EDITION (SEE COVER SHEET FOR DATE) 1-1
1.2 CHARTER 1-1
1.2.1 PURPOSE 1-1
1.2.2 SCOPE 1-1
1.2.3 GOALS 1-2
1.2.4 REVIEWING AND UPDATING THIS DOCUMENT 1-2

2.0 INPUT 2-1
2.1 SYSTEM COMMAND LANGUAGE 2-1
2.2 PRODUCT CALL COMMANDS 2-1
2.2.1 APPLICABILITY 2-1
2.2.2 TERMINOLOGY 2-3
2.2.3 SYNTAX 2-3
2.2.4 PARAMETER 2-4
2.2.4.1 Positional Ordering of Product Set Parameters 2-4
2.2.4.2 Types of Parameters 2-5
2.2.4.3 Parameter Names and Descriptions 2-7
2.3 SOURCE INPUT 2-29
2.3.1 SOURCE INPUT FILE ORGANIZATION 2-30
2.3.2 SOURCE STATEMENT FORMAT 2-30
2.3.2.1 Statement Identifier 2-31
2.3.2.2 Line Numbers 2-31
2.3.2.3 Statement Body 2-31
2.3.2.4 Blank Compression 2-32
2.3.2.5 Empty Input File 2-32
2.3.2.6 Null Source Line Convention 2-32
2.3.3 DISPOSITION OF INPUT FILE 2-32
2.4 COMPILATION DIRECTIVES 2-33
2.4.1 PAGE EJECT 2-34
2.4.2 SOURCE LISTING 2-35
2.4.3 LINE SKIP 2-35
2.4.3.1 LINE SPACING 2-35
2.4.4 TITLE LINES 2-35
2.4.5 RANGE CHECK 2-36
2.4.6 EXECUTION TRACE 2-36
2.4.7 DEBUG STATEMENTS 2-36
2.4.8 SEQUENCE CHECK 2-37
2.4.9 OBJECT CODE LISTING 2-37
2.4.10 STACKING COMPILATION DIRECTIVES 2-37
2.5 PRODUCT DIRECTIVES 2-38
2.5.1 STANDARD PARAMETERS 2-38
2.5.2 STANDARD COMMANDS 2-39

3.0 OUTPUT 3-1
3.1 RECOMMENDED NUMBER BASES 3-1
3.1.1 SITUATIONS AND RECOMMENDED NUMBER BASES 3-1
3.2 LOGS 3-2
3.2.1 ASCII LOGS 3-2
3.2.1.1 System Log 3-3

3.2.1.1.1 PURPOSE 3-4
3.2.1.1.2 CONVENTIONS 3-4
3.2.1.2 Job Log 3-5
3.2.1.2.1 PURPOSE 3-5
3.2.1.2.2 CONVENTIONS 3-5
3.2.2 BINARY LOGS 3-6
3.2.2.1 Account Log 3-8
3.2.2.1.1 PURPOSE 3-8
3.2.2.2 Engineering Log 3-8
3.2.2.2.1 PURPOSE 3-8
3.2.2.3 Statistic Log 3-8
3.2.2.3.1 PURPOSE 3-8
3.2.2.4 Job Statistic Log 3-8
3.2.2.4.1 PURPOSE 3-8
3.2.2.5 Binary Log Conventions 3-9
3.3 LISTABLE OUTPUT 3-9
3.3.1 LISTING PAGE FORMATS 3-9
3.3.1.1 Vertical Layout 3-10
3.3.1.2 Format Attributes 3-10
3.3.1.2.1 CONTINUOUS OUTPUT FILE 3-11
3.3.1.2.2 PAGINATED OUTPUT FILES 3-11
3.3.1.3 Standard Carriage Control Codes 3-11
3.3.1.4 Horizontal Layout 3-12
3.3.1.5 Standard Listing Header 3-13
3.3.1.6 OTHER FORMATS 3-13
3.3.2 FORMATS 3-15
3.3.2.1 Wide Format (132 columns) 3-15
3.3.2.2 Narrow Format (80 Columns) 3-16
3.3.3 SOURCE LISTING FORMATS 3-18
3.3.3.1 Standard Header Contents 3-18
3.3.3.2 TITLE Lines 3-18
3.3.3.3 Wide Format 3-19
3.3.3.4 Narrow Format 3-20
3.3.4 OBJECT CODE LISTING FORMAT 3-21
3.3.4.1 Standard Header Contents 3-23
3.3.4.2 Standard Instruction Mnemonics 3-24
3.3.5 ATTRIBUTES LISTING FORMAT 3-24
3.3.5.1 Standard Header Contents 3-24
3.3.5.2 Wide Format 3-25
3.3.5.3 Narrow Format 3-30
3.3.5.4 Standard Field Values 3-30
3.3.5.4.1 ENTITY TYPES 3-30
3.3.5.4.2 BASIC ATTRIBUTES 3-31
3.3.5.4.3 REFERENCE TYPES 3-32
3.3.6 DIAGNOSTIC LISTING 3-32
3.3.6.1 Standard Header Contents 3-33
3.3.6.2 Standard Diagnostic Listing Format 3-33
3.3.6.3 Standard Diagnostic Summary Format 3-34
3.3.7 COMPILATION OPTIONS 3-35
3.4 ERROR MESSAGES 3-35
3.4.1 CONDITION CODES, EMBEDDED PRODUCTS, AND MESSAGE GENERATION . . 3-35
3.4.1.1 Condition Codes 3-35
3.4.1.2 Embedded Products 3-36

3.4.1.3 Message generation 3-36
3.4.2 MESSAGE TEXT 3-36
3.4.2.1 Message Formats 3-37
3.4.2.2 Error Summaries in Logs 3-37
3.4.2.3 Message Wording 3-38
3.5 USAGE STATISTICS 3-39
3.5.1 PURPOSE OF STATISTICS 3-40
3.5.2 STATISTICS FACILITY 3-40
3.5.3 PRODUCT STATISTICS COLLECTED BY NOS/VE 3-41
3.5.4 STATISTICS COLLECTED BY PRODUCTS 3-43
3.5.4.1 Input Unit Statistics 3-43
3.5.4.2 Internal Statistics 3-44
3.5.5 WHEN TO LOG STATISTICS 3-44

4.0 SYSTEMWIDE CONVENTIONS 4-1
4.1 NAMES, DATES AND TIMES 4-1
4.1.1 NAMING CONVENTIONS 4-1
4.1.1.1 Product Identifiers 4-2
4.1.1.2 Other Global Identifiers 4-4
4.1.1.3 Classes of Names 4-4
4.1.1.4 Special Characters 4-5
4.1.1.5 User Global Names 4-5
4.1.1.6 Deck Naming Guidelines 4-5
4.1.1.7 SCU GROUP NAMING GUIDELINES 4-10
4.1.2 RESERVED FILE NAMES 4-12
4.1.3 DATE AND TIME 4-12
4.2 INTERACTIVE PROCESSING 4-12
4.2.1 INTERACTIVE OUTPUT 4-13
4.2.1.1 General 4-13
4.2.1.2 Messages 4-14
4.2.1.3 Listings 4-15
4.2.2 INTERACTIVE INPUT 4-16
4.2.2.1 General 4-16
4.2.2.2 Input Diagnoses 4-17
4.2.3 CONTROL 4-17
4.2.3.1 Connectivity 4-17
4.2.3.2 Interrupts and Connection Breaks 4-18
4.2.3.3 Status 4-19
4.2.3.4 Help 4-20
4.2.4 PRODUCT SET RUN TIME COMMANDS 4-20
4.2.4.1 PAUSE and STOP Literal 4-21
4.2.4.2 ACCEPT FROM CONSOLE 4-21
4.3 INSTALLATION PARAMETERS 4-21
4.3.1 GENERAL GUIDELINES 4-23
4.3.2 LIST OF PRODUCT SET PARAMETERS 4-23
4.4 ERROR PROCESSING 4-23
4.4.1 STATUS VARIABLE 4-23
4.4.2 ERROR TERMINATION 4-24
4.4.3 INTERACTIVE ERROR PROCESSING 4-25
4.4.3.1 Error Messages 4-25
4.4.3.2 Diagnostics 4-26
4.4.3.3 Input Diagnosis 4-26
4.4.4 BATCH ERROR PROCESSING 4-26

4.4.4.1 Error Messages 4-26
4.4.4.2 Input Diagnosis 4-27
4.4.5 TRANSACTION ERROR PROCESSING 4-27
4.4.6 RESTART 4-27
4.5 EFFECTIVE USE OF C180 HARDWARE 4-27
4.5.1 HARDWARE OPERATION 4-27
4.5.1.1 Interlock Words 4-27
4.5.1.2 Pre-serialization of Clear Lock 4-28
4.5.1.3 Register Reservations 4-28
4.5.1.4 Alignment of Tables and Words 4-29
4.5.1.4.1 64-BIT WORD BOUNDARIES 4-29
4.5.1.4.2 OTHER BOUNDARIES 4-31
4.5.2 HARDWARE PERFORMANCE 4-32
4.5.2.1 Locality of Reference 4-32
4.5.2.2 Register Allocation and Usage 4-33
4.5.3 SECURITY 4-33
4.5.3.1 Procedure Parameters 4-33
4.6 SUPPORT OF EBCDIC DATA 4-34
4.7 KEYPOINT USAGE 4-35
4.7.1 KEYPOINT CLASSES 4-35
4.7.1.1 Operating System 4-37
4.7.1.2 Product Set 4-37
4.7.1.3 Other Classes 4-37
4.7.2 KEYPOINT IDENTIFIERS 4-37
4.7.2.1 Operating System 4-37
4.7.2.2 Product Set 4-40
4.7.3 KEYPOINT USE 4-41

5.0 COMPILER AND ASSEMBLY CODE CONVENTIONS 5-1
5.1 USE OF LOADER FEATURES 5-1
5.2 INTERLANGUAGE CALLING SEQUENCES 5-3
5.2.1 CALLING SEQUENCE FORMATS 5-4
5.2.1.1 Kinds of Parameters 5-4
5.2.1.2 System Format of the Interlanguage Calling Sequence . . . 5-5
5.2.1.3 General Format of the Interlanguage Calling Sequence . . . 5-5
5.2.1.4 Summary of Format Differences 5-5
5.2.1.5 Calls Potentially from Another Language 5-6
5.2.1.6 Calls Potentially to Another Language 5-6
5.2.1.6.1 SUPPORT FOR CALLS TO ANOTHER LANGUAGE 5-7
5.2.2 CALL 5-7
5.2.3 REGISTER SAVING CONVENTIONS 5-8
5.2.3.1 Information Required Across Call 5-8
5.2.4 FUNCTIONS 5-9
5.2.5 PARAMETER LIST 5-10
5.2.5.1 System Format Parameter List 5-10
5.2.5.2 General Format Parameter List 5-11
5.2.5.2.1 FLAG WORD PRECEDING PARAMETER LIST 5-11
5.2.5.2.2 GENERAL FORMAT SIMPLE VALUE PARAMETERS 5-12
5.2.5.2.3 GENERAL FORMAT EXTENDED VALUE PARAMETERS 5-12
5.2.5.2.4 GENERAL FORMAT SIMPLE REFERENCE PARAMETERS 5-13
5.2.5.2.5 GENERAL FORMAT EXTENDED REFERENCE PARAMETERS 5-13
5.2.5.2.6 GENERAL FORMAT SIMPLE BIT REFERENCE PARAMETERS 5-14
5.2.5.2.7 GENERAL FORMAT EXTENDED BIT REFERENCE PARAMETERS 5-14

5.2.5.2.8 GENERAL FORMAT STRING DESCRIPTORS 5-15
5.2.5.2.9 GENERAL FORMAT BIT STRING DESCRIPTORS 5-15
5.2.5.2.10 GENERAL FORMAT BIT STRING OFFSET 5-15
5.2.5.2.11 GENERAL FORMAT ARRAY DESCRIPTORS 5-15
5.2.5.2.11.1 Stride 5-16
5.2.6 DATA REPRESENTATION 5-17
5.2.6.1 Integer 5-17
5.2.6.1.1 4 BYTE INTEGER 5-18
5.2.6.1.2 8 BYTE INTEGER 5-18
5.2.6.2 Fixed Length Character (String) 5-18
5.2.6.3 Real 5-18
5.2.6.4 Double Precision 5-18
5.2.6.5 Complex 5-19
5.2.6.6 Boolean 5-19
5.2.6.7 Pointer 5-19
5.2.7 DATA ALIGNMENT AND PACKING 5-19
5.2.7.1 Variables 5-20
5.2.7.2 Structures 5-20
5.2.7.3 Arrays 5-20
5.2.7.3.1 ARRAYS OF VARIABLES 5-20
5.2.7.3.2 ARRAYS OF STRUCTURES 5-21
5.2.7.3.3 COMMON BLOCKS 5-21
5.2.8 LANGUAGE INTERCHANGE TABLE 5-21
5.2.8.1 Extended Interchange 5-22
5.2.9 REGISTER CALL FUNCTIONS 5-23
5.3 INTERPRODUCT FILE USAGE 5-24
5.4 STORAGE MANAGEMENT 5-24
5.4.1 STANDARD STACK FRAME 5-24
5.4.1.1 Traceback 5-25
5.4.1.2 Static Chain vs. Display 5-25
5.4.2 CHAINS OF ON-CONDITION PROCESSORS 5-25
5.4.3 DYNAMIC NON-STACK STORAGE 5-26
5.4.3.1 Dynamic Segments 5-26
5.4.3.2 Fixed-Position Dynamic Storage 5-26
5.4.3.3 Variable-Position Dynamic Storage 5-26
5.5 COMMON SUPPORT MODULES 5-26

6.0 GLOSSARY OF TERMS 6-1