CYBIL COMMON INPUT/OUTPUT

REFERENCE MANUAL

REVISION DEFINITION SHEET

REV	DATE	DESCRIPTION
1	02/08/85	Preliminary manual released.
2	04/22/85	Major rewrite of the complete manual.
3	10/01/85	Complete revision of manual.
4	03/15/85	Added additional interfaces and enhancements. Minor text revisions.

Table of Contents

1.0 INTRODUCTION

The CYBIL Common Input/Output package (CYBILIO) is a collection of procedures and data types which provide an Input/Output system that interfaces a CYBIL program to the NOS/VE, NOS, NOS/BE, VSOS, EOS, and APOLLO Aegis I/O systems.

The objectives of CYBILIO are to:

- Provide an input/output capability that is standardized across implementations of CYBIL.
- Ease transportability of programs by reducing operating system dependencies within a program to a minimum.
- Provide a simple, easy to use input/output interface.

Display screen interfaces and the more sophisticated input/output capabilities of the various operating systems are beyond the scope of CYBILIO.

1.1 APPLICABLE DOCUMENTS

60455280	CYBIL Reference Manual
60457280	Language Specification for CDC CYBER IMPLEMENTATION LANGUAGE
60460300	CYBIL I/O ERS (Obsolete)
60457250	SES User's Handbook
60459660	NOS Version 2 Reference Manual (Volume 1)
60459670	NOS Version 2 Reference Manual (Volume 2)
60459680	NOS Version 2 Reference Manual (Volume 3)
60459690	NOS Version 2 Reference Manual (Volume 4)
60450100	NOS Version 1 Modify Reference Manual
60493800	NOS/BE Version 1 Reference Manual
60494100	NOS/BE Version 1 System Programmer's Reference Manual
60499900	Update Version 1 Reference Manual
60464114	CYBIL for NOS/VE File Interface Usage Reference Manual
60459410	VSOS Version 2 Reference Manual (Volume 1)
000529	APOLLO Aegis Domain System Programmer's Reference Manual
60460310	CYBIL Miscellaneous Routines Interface Reference Manual

2.0 CYBILIO FILES

2.0 CYBILIO FILES

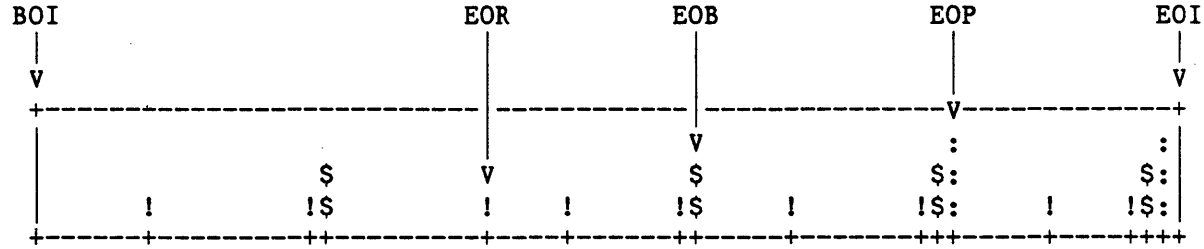
2.1 FILE STRUCTURE

All CYBILIO files have a beginning of information and an end of information. Files can be further subdivided into a maximum of three levels of logical structure. The number of possible levels varies among different operating systems. Within CYBILIO, the possible levels of logical file structure are defined as follows:

partition — A file may be subdivided into partitions. A partition begins either at the beginning of information (BOI) or after the end of partition (EOP) of the previous partition.

block — Partitions may be subdivided into blocks. A block begins at the beginning of information (BOI), after an end of partition (EOP), or after the end of block (EOB) of a preceeding block.

record — The lowest level of subdivision within a file is a record. A record begins at the beginning of information (BOI), after an end of partition (EOP) or end of block (EOB), or after the end of record (EOR) of a preceeding record.



Logical Structure of CYBILIO Files

All or none of the levels of logical file structure may exist within a cybilio file.

2.2 FILE TYPES

CYBILIO defines four distinct types of files:

record files
binary files
text files
display files

Each file type has certain characteristics and limitations. The following subsections describe these characteristics and limitations.

2.2.1 RECORD FILES

Record files are files in which data exists as a sequence of logical records each of which is terminated with an end of record (EOR).

CYBILIO provides facilities to read and write both full and partial records. That is, a record may be transferred as the result of a single read or write operation or, a record may be transferred as the result of several partial read or write operations. Record file reads and writes map the data to a CYBIL data structure. For example, a CYBIL array may be written as a record or partial record. The address and size of the data structure are passed to CYBILIO as a CYBIL sequence pointer. CYBILIO uses this information to write a record that exactly corresponds byte for byte with the way the data is stored in the CYBIL data structure.

CYBILIO supports only sequential access of record files. Data appears on such files in the order in which it was written, and can only be read in that same order.

Record files may be positioned to the beginning or end of information. In addition, record files may be positioned forward or backward a user-specified number of records, blocks or partitions. Note that positioning a record file backwards and then writing to the file implies that any data following that just written to the file is lost. The end of information always immediately follows the last data written to the file.

2.2.2 BINARY FILES

Binary files are files in which the data exists as a "stream" of cells. Binary files may be subdivided into partitions and blocks. CYBILIO does NOT support the subdivision of binary files into records.

CYBILIO imposes no structure on the data in a binary type file. The task that writes data on a binary type file is responsible for determining how the data can later be read. It should write data organization indicators as needed. A program that reads the binary file data must use the data conventions imposed by the program that wrote the data.

CYBILIO supports both random and sequential access of binary files. Random access procedure interfaces transfer data to or from "random addresses" known as file keys. The file keys identify the number of the cell within the file at which the transfer is to begin. Sequential access procedure interfaces transfer data to or from the "address" or file key at which the file is currently positioned. As with record files, the data read or written is transferred as a "block of cells" that are mapped to the CYBIL data structure being read or written.

Binary files may be positioned to the beginning of information, end of information, or to any file key within the file. Because binary files can be accessed randomly, positioning a binary file at the beginning of information and writing to the file does not necessarily imply that existing data (which follows the data being written) will be lost (c.f., record files).

2.2.3 TEXT FILES

Text type files are a variation of record type files. Text files are assumed to contain character data. Since we generally think of character data in terms of "lines", CYBILIO will refer to text file records as lines and the end of record (EOR) for text files as end of line (EOL).

Data is passed to and from the text file procedures as CYBIL strings rather than as CYBIL sequence pointers. Like record type files, text files can only be accessed sequentially.

The basic entity on a text file is a line which can be transferred to/from the file in whole or in part. In addition, there are facilities to tab to a specified column in an output line and skip a specified number of lines. Text files may be positioned to the beginning of information or

to the end of information.

2.2.4 DISPLAY FILES

Display files are a special form of write-only text files. Display type files should be used when the file is to be printed or routed to any device which uses format control characters. CYBILIO automatically prefixes format control characters to each line written to display type files.

Display type files have additional facilities for (vertical) format control. It is possible to limit the number of printed lines on a page, insert a given number of empty lines, overprint lines, position the next line at a specified line number or at the top of the next display page. Several functions are provided to interrogate certain items of display page information for display files.

Display files may only be written. If it is necessary to read a file which was written as a display file, the file should be accessed as a text type file.

The user may associate with each display file, a procedure to be called when a "page overflow condition" occurs for that file. The procedure may be a user-specified procedure or a special internal CYBILIO procedure that produces a "standard" title line.

2.3 CYBILIO DATA TYPES

This section defines the CYBIL "types" required to interface to CYBILIO.

2.3.1 OST$STATUS

```
  TYPE
    ost$status = record
      case normal: boolean of
      = FALSE =
        condition: ost$status_condition_code,
        text: ost$string,
      = TRUE =
        ,
      casend,
    recend;

*copyc ost$status_condition_code
*copyc ost$string

*copyc osc$max_condition
*copyc ost$status_condition
*copyc osc$status_parameter_delimiter
```

All CYBILIO procedures include a status parameter of type ost$status. The status conditions returned by CYBILIO are listed in the STATUS MESSAGES section of this document.

2.3.2 CYT$CURRENT_FILE_POSITION

```
{* ZCYTCFP  cyt$current_file_position *}

  TYPE
    cyt$current_file_position = (cyc$beginning_of_information,
      cyc$middle_of_record, cyc$end_of_record, cyc$end_of_block,
      cyc$end_of_partition, cyc$end_of_information);
```

A variable of this type returns the current position of a file.

2.3.3 CYT$FILE

```
{* ZCYTFIL  cyt$file *}

  TYPE
    cyt$file = ↑SEQ ( * );
```

Every CYBILIO procedure and function has a parameter of this type. CYBILIO defines the value of the variable when the file is opened. The variable remains defined until it is passed to the file close procedure. The consequences of using an undefined or user-altered cyt$file variable to call any CYBILIO procedure, except the file open procedure, is unpredictable.

2.3.4 CYT$FILE_NAME

```
{* ZCYTFN  cyt$file_name *}

  TYPE
    cyt$file_name = string ( * <= cyc$max_file_name_size);

  CONST
    cyc$max_file_name_size = 512;
```

This type is used to identify a file to the file open procedure. File

name length and lower-to-upper case conversion are operating system dependent. See chapter on operating system dependencies.

2.3.5 CYT$FILE_SPECIFICATIONS

```
{* ZCYTFS  cyt$file_specifications *}

  TYPE
    cyt$file_specifications = ↑array [1 .. * ] of cyt$file_specification,

    cyt$file_specification_selector = (cyc$file_kind, cyc$file_access,
      cyc$file_existence, cyc$open_position, cyc$close_file_disposition,
      cyc$file_contents, cyc$file_processor, cyc$file_character_set,
      cyc$new_page_procedure, cyc$page_length, cyc$page_width,
      cyc$page_format, cyc$future_spec1, cyc$future_spec2, cyc$future_spec3,
      cyc$future_spec4, cyc$future_spec5),


    cyt$file_specification = record
      case selector: cyt$file_specification_selector of
      = cyc$file_kind =
        file_kind: cyt$file_kind,
      = cyc$file_access =
        file_access: cyt$file_access,
      = cyc$file_existence =
        file_existence: cyt$file_existence,
      = cyc$open_position =
        open_position: cyt$open_close_position,
      = cyc$close_file_disposition =
        close_disposition: cyt$close_file_disposition,
      = cyc$file_contents =
        file_contents: cyt$file_contents,
      = cyc$file_processor =
        file_processor: cyt$file_processor,
      = cyc$file_character_set =
        file_character_set: cyt$file_character_set,
      = cyc$new_page_procedure =
        new_page_procedure: cyt$new_page_procedure,
      = cyc$page_length =
        page_length: cyt$page_length,
      = cyc$page_width =
        page_width: cyt$page_width,
      = cyc$page_format =
        page_format: cyt$page_format,
      = cyc$future_spec1 =
        ,
      = cyc$future_spec2 =
        ,
```

```
      = cyc$future_spec3 =
        ,
      = cyc$future_spec4 =
        ,
      = cyc$future_spec5 =
        ,
      casend,
    recend;


*copyc cyt$close_file_disposition
*copyc cyt$file_access
*copyc cyt$file_character_set
*copyc cyt$file_existence
*copyc cyt$file_kind
*copyc cyt$file_contents
*copyc cyt$file_processor
*copyc cyt$new_page_procedure
*copyc cyt$open_close_position
*copyc cyt$page_length
*copyc cyt$page_width
*copyc cyt$page_format
```

A variable of this type is passed as a parameter on the cyp$open_file procedure call. CYBILIO uses the file specification records to determine how the file is to be opened, how the file is to be operated upon, and what to do with the file after it is closed.

File specifications are defined by specifying a file specification key to select the desired file specification record. Then, a file specification value is specified that corresponds to the CYBIL type permitted for the record.

The example programs in the appendices show how file specifications may be established. Additional information about file specifications may be found in the following CYBIL type descriptions and in the description of the cyp$open_file interface.

The following subsections describe the various CYBIL types referenced by cyt$file_specifications.

2.3.5.1 cyt$close file disposition

```
{* ZCYTCFD cyt$close_file_disposition *}

  TYPE
    cyt$close_file_disposition = (cyc$delete_file, cyc$retain_file,
      cyc$return_file, cyc$unload_file, cyc$default_file_disposition);

  CONST
    cyc$detach_file = cyc$return_file;
```

This file specification tells CYBILIO what to do with a file after the file is closed. See the chapter on operating system dependencies for the meaning of the various ordinal values.

If the close_file_disposition record is not specified in the file_specifications, CYBILIO assumes a default value of cyc$default_close_disposition.

2.3.5.2 cyt$file access

```
{* ZCYTFA  cyt$file_access *}

  TYPE
    cyt$file_access = (cyc$read, cyc$write, cyc$read_write);
```

This file specification specifies the permitted "direction" of data transfers. CYBILIO retains the file_access specified when the file is opened and validates all read/write requests to the file against the file_access. Attempts to write to a file opened for cyc$read file_access or to read from a file opened for cyc$write file_access will be blocked and abnormal status will be returned in the status variable for the request.

cyc$read:	the file is to be opened for read-only access.
cyc$write:	the file is to be opened for write-only access.
cyc$read_write:	the file is to be opened for read-write access.

If the file_access record is not specified in the file_specifications, CYBILIO assumes a default value of cyc$read_write.

NOTE: If a file is opened for cyc$read file_access and cyc$new_file file_existence, the open will fail and abnormal status will be returned. If a file is opened for cyc$read file_access and cyc$new_or_old_file file_existence and the file does not exist, the open will fail and abnormal status will be returned.

2.3.5.3 cyt$file_existence

```
{* ZCYTFE  cyt$file_existence *}

  TYPE
    cyt$file_existence = (cyc$new_file, cyc$old_file, cyc$new_or_old_file);
```

This file specification specifies whether the file must exist when it is opened, must not exist when it is opened, or may or may not exist when it is opened.

cyc$old_file:	the file must exist or the file open procedure returns abnormal status
cyc$new_file:	the file must NOT exist, or the file open procedure returns abnormal status
cyc$new_or_old_file:	if the file does not exist it will be created,

If the file_existence record is not specified in the file_specifications, CYBILIO assumes a default value of cyc$new_or_old_file.

2.3.5.4 cyt$file kind

```
{* ZCYTFK  cyt$file_kind *}

  TYPE
    cyt$file_kind = (cyc$binary_file, cyc$display_file, cyc$record_file,
          cyc$text_file);
```

This file specification specifies the kinds of CYBILIO calls that may be addressed to a file. CYBILIO retains the file_kind value and validates CYBILIO procedure calls against the file_kind value. For example, if a file is opened as cyc$text_file, any attempt to use any record, binary, or display file procedure calls is prohibited by CYBILIO and the status variable returned will indicate cye$incorrect_operation.

If the file_kind record is not specified in the file_specifications, CYBILIO assumes a default value of cyc$record_file.

2.3.5.5 cyt$file_character_set

```
{* ZCYTFCS  cyt$file_character_set *}

  TYPE
    cyt$file_character_set = (cyc$ascii, cyc$ascii612, cyc$ascii812,
          cyc$display_64, cyc$reserved_code1, cyc$reserved_code2);
```

This file specification specifies the character set for text and display type files.

cyc$ascii	8-bit ASCII code
cyc$ascii612	CYBER 170 6/12 ASCII code
cyc$ascii812	CYBER 170 8/12 ASCII code
cyc$display64	CYBER 170 64-character display code
cyc$reserved_code1	reserved for future use
cyc$reserved_code2	reserved for future use

If the file_character_set record is not specified in the file_specifications, CYBILIO assumes a default value of cyc$ascii612 for NOS implementations, a default value of cyc$ascii812 for NOS/BE implementations, and a value of cyc$ascii for all others.

NOTE: The file_character_set is used only by text and display type files. If this record is defined for binary or record type files, CYBILIO will ignore the file_character_set specification.

2.3.5.6 cyt$file_contents

```
{* ZCYTFC cyt$file_contents *}

{}
{ The following are predefined string constants for file_contents:}
{}
?? FMT (FORMAT := OFF) ??
  CONST
    cyc$ascii_log                = 'ASCII_LOG                ',
    cyc$binary                   = 'BINARY                   ',
    cyc$binary_log               = 'BINARY_LOG               ',
    cyc$data                     = 'DATA                     ',
    cyc$file_backup              = 'FILE_BACKUP              ',
    cyc$legible                  = 'LEGIBLE                  ',
    cyc$legible_data             = 'LEGIBLE_DATA             ',
    cyc$legible_library          = 'LEGIBLE_LIBRARY          ',
    cyc$legible_unknown          = 'LEGIBLE_UNKNOWN          ',
    cyc$list                     = 'LIST                     ',
    cyc$list_unknown             = 'LIST_UNKNOWN             ',
    cyc$object                   = 'OBJECT                   ',
    cyc$object_data              = 'OBJECT_DATA              ',
    cyc$object_library           = 'OBJECT_LIBRARY           ',
    cyc$screen                   = 'SCREEN                   ',
    cyc$screen_form              = 'SCREEN_FORM              ',
    cyc$unknown_contents         = 'UNKNOWN                  ';
?? FMT (FORMAT := ON) ??

  TYPE
    cyt$file_contents = string (31);
```

This file specification specifies a description of the contents of a file. The use of this value is system dependent.

If the file_contents record is not specified in the file_specifications, CYBILIO assumes a default value of cyc$unknown_contents.

2.3.5.7 cyt$file_processor

```
{* ZCYTFP cyt$file_processor *}

{ The following are predefined strings for referring to file processor.}

?? FMT (FORMAT := OFF) ??
  CONST
    cyc$ada                          = 'ADA                          ',
    cyc$apl                          = 'APL                          ',
    cyc$assembler                    = 'ASSEMBLER                    ',
    cyc$basic                        = 'BASIC                        ',
    cyc$c                            = 'C                            ',
    cyc$cobol                        = 'COBOL                        ',
    cyc$cybil                        = 'CYBIL                        ',
    cyc$debugger                     = 'DEBUGGER                     ',
    cyc$fortran                      = 'FORTRAN                      ',
    cyc$lisp                         = 'LISP                         ',
    cyc$pascal                       = 'PASCAL                       ',
    cyc$pli                          = 'PLI                          ',
    cyc$ppu_assembler                = 'PPU_ASSEMBLER                ',
    cyc$prolog                       = 'PROLOG                       ',
    cyc$scl                          = 'SCL                          ',
    cyc$scu                          = 'SCU                          ',
    cyc$unknown_processor            = 'UNKNOWN                      ',
    cyc$vx                           = 'VX                           ';
?? FMT (FORMAT := ON) ??

  TYPE
    cyt$file_processor = string (31);
```

This file specification specifies a description of the processor of a file. The use of this value is system dependent.

If the file_processor record is not specified in the file_specifications, CYBILIO assumes a default value of cyc$unknown_processor.

2.3.5.8 cyt$new_page_procedure

```
{* ZCYTNPP  cyt$new_page_procedure *}

  TYPE
    cyt$new_page_procedure = record
      case kind: cyt$page_procedure_kind of
      = cyc$user_specified_procedure =
        user_procedure: cyt$user_page_procedure,
      = cyc$standard_procedure =
        title: string (cyc$title_size),
      = cyc$omit_page_procedure =
        ,
      casend,
    recend,

    cyt$page_procedure_kind = (cyc$user_specified_procedure,
      cyc$standard_procedure, cyc$omit_page_procedure),

    cyt$user_page_procedure = ↑procedure (display_file: cyt$file;
      next_page_number: integer;
      VAR status: ost$status);

  CONST
    cyc$title_size = 45;

*copyc ost$status
*copyc cyt$file
```

This file specification specifies how CYBILIO will handle "page overflow" conditions for display type files.

If the tag field of the new_page_procedure record specifies cyc$user_specified_procedure, CYBILIO will automatically call the procedure specified by the USER_PROCEDURE field whenever a "page overflow" condition occurs.

If the tag field of the new_page_procedure record specifies cyc$standard_procedure, CYBILIO will automatically initiate a display page eject and produce a "standard" title line followed by one blank line whenever a "page overflow' condition occurs. The title field of the new_page_procedure record specifies a string of characters that CYBILIO will include in the "standard" title line. (See the section on display type files for a description of the "standard" title line.)

2.0 CYBILIO FILES
2.3.5.8 cyt$new_page_procedure

If the tag field of the new_page_procedure record specifies cyc$omit_page_procedure, CYBILIO will simply initiate a display page eject.

If the new_page_procedure record is not specified in the file_specifications, CYBILIO will assume a default value of cyc$omit_page_procedure.

NOTE: A new_page_procedure is used only by display files. If this record is defined for any other type of file, CYBILIO will ignore the new_page_procedure specification.

2.3.5.9 cyt$page_format

```
{* ZCYTPF  cyt$page_format *}

  TYPE
    cyt$page_format = (cyc$continuous_form, cyc$burstable_form,
          cyc$non_burstable_form, cyc$untitled_form);
```

This file specification specifies the presence and frequency of titling in a display file whose file contents is cyc$list or cyc$list_unknown.

cyc$burstable_form	specifies that titling and display page eject should occur at the frequency defined by the page length of the file. This is the recommended value for files that are to be listed on a forms printer with a page eject required for each page.
cyc$non_burstable_form	specifies that titling should be separated from other data by a triple space rather than by forcing a display page eject as in cyc$burstable_form. A display page eject and titling also occur at the frequency defined by the page length of the file.
cyc$continuous_form	specifies that titling should appear once at the beginning of the file followed by triple spacing.
cyc$untitled_form	specifies that no titling and no display page eject should occur anywhere in the file.

Titling, as used in the preceeding explanations, is the processing of the new_page_procedure.

NOTE: A page_format specification is used only by display files. If this record is defined for any other type of file, CYBILIO will ignore the page_format specification.

2.3.5.10 cyt$open_close_position

```
{* ZCYTOCP  cyt$open_close_position *}

  TYPE
    cyt$open_close_position = (cyc$beginning, cyc$end, cyc$asis,
          cyc$default_open_position);
```

This type is used when opening a file to designate where the file should be initially positioned (at its beginning, where ever it happens to be, at its end, or at its default position).

This type is also used when closing a file to designate whether the file is to be rewound, positioned at its end, or left as is. See the chapter on operating system dependencies.

If the open_position record is not specified in the file_specifications, CYBILIO assumes a default value of cyc$default_open_position.

2.3.5.11 cyt$page_length

```
{* ZCYTPL  cyt$page_length *}

  TYPE
    cyt$page_length = 1 .. cyc$page_limit;

  CONST
    cyc$page_limit = 439804651103;
```

This file specification specifies the number of lines on a page for display type files.

If the page_length record is not specified in the file_specifications, CYBILIO assumes a system dependent default value.

NOTE: A page_length is used only by display files. If this record is defined for any other type of file, CYBILIO will ignore the page_length specification.

2.3.5.12 cyt$page_width

```
{* ZCYTPW  cyt$page_width *}

  TYPE
    cyt$page_width = 1 .. cyc$max_page_width;

  CONST
   cyc$wide_page_width = 132,
   cyc$narrow_page_width = 80,
   cyc$max_page_width = 65535;
```

This file specification specifies the maximum length of a text line for display or text type files.

If the page_width record is not specified in the file_specifications, CYBILIO assumes a system dependent default value.

NOTE: A page_width is used only by display and text files. If this record is defined for any other type of file, CYBILIO will ignore the page_width specification.

2.3.6 CYT$SKIP_DIRECTION

```
{* ZCYTSD  cyt$skip_direction *}

  TYPE
    cyt$skip_direction = (cyc$forward, cyc$backward);
```

A value of this type is passed to the cyp$position_record_file procedure to specify the direction in which the file is to be positioned.

2.3.7 CYT$SKIP_UNIT

```
{* ZCYTSU  cyt$skip_unit *}

  TYPE
    cyt$skip_unit = (cyc$record, cyc$block, cyc$partition);
```

A value of this type is passed to the cyp$position_record_file to specify the unit of file structure to be used for file positioning.

2.3.8 CYT$SYSTEM_TYPE

```
{* ZCYTST  cyt$system_type *}

  TYPE
    cyt$system_type = (cyc$nosve, cyc$nos, cyc$nosbe, cyc$vsos, cyc$eos,
          cyc$aegis);
```

A value of this type is returned by the cyp$operating_system function call. The ordinal identifies the operating system on which the program is running.

2.4 USING CYBILIO

2.4.1 NOS/VE

2.4.1.1 Source Code Interface to CYBILIO

To interface to CYBILIO, a CYBIL program module must include the relevant type and procedure declarations. These can be *COPYed from an SCU source library. The name of this source library is :$SYSTEM.$SYSTEM.CYBIL.OSF$PROGRAM_INTERFACE. Refer to the operating system dependent section for a list of CYBILIO deck names.

2.4.1.2 Object Code Interface to CYBILIO

Before a program which uses CYBILIO can be executed, it must be linked with the CYBILIO object modules which are located on the CYBIL run-time library. The name of the CYBIL run-time library is CYF$RUN_TIME_LIBRARY. Linking to the object modules is done by including CYF$RUN_TIME_LIBRARY in the module descriptor.

2.4.2 NOS

2.4.2.1 Source Code Interface to CYBILIO

To interface to CYBILIO, a CYBIL program module must include the relevant type and procedure declarations. These can be *CALLed from a MODIFY/MADIFY source library or *COPYed from an SCU source_library.

The name of the MODIFY/MADIFY source library is CYBCCMN, which is accessible by including the CYBCCMN parameter in the SES.GENCOMP call. The SCU source library is accessible via SES.GETCOMN which makes the library available as local file CYBCCMN. The CYBILIO procedure and type declarations can then be *COPYed by including CYBCCMN on the BASE parameter of the SCU.EXPAND_DECK call. Refer to the operating system dependent section for a list of CYBILIO deck names.

2.4.2.2 Object Code Interface to CYBILIO

Before a program which uses CYBILIO can be executed, it must be linked with the CYBILIO object modules which are located on the CYBIL run-time library. The name of the CYBIL run-time library is CYBCLIB. Linking to the CYBILIO object modules may be done by including the CYBCLIB parameter on the SES.LINK170 call or by having CYBCLIB as a local file and including its name in the loader directives. CYBCLIB can be acquired as a local file via the SES.GETLIB call.

2.4.3 NOS/BE

2.4.3.1 Source Code Interface to CYBILIO

To interface to CYBILIO, a CYBIL program module must include the relevant type and procedure declarations. These can be *CALLed from an UPDATE source library. The name of this program library is CYBCCMN. Refer to the operating system dependent section for a list of CYBILIO deck names.

2.4.3.2 Object Code Interface to CYBILIO

Before a program which uses CYBILIO can be executed, it must be linked with the CYBILIO object modules which are located on the CYBIL run-time library. The name of the CYBIL run-time library is CYBCLIB. Linking to the CYBILIO object modules may be done by having CYBCLIB as a local file and including its name in the loader directives.

2.4.4 VSOS

2.4.4.1 SOURCE CODE INTERFACE TO CYBILIO

>>>>>> to be supplied <<<<<<

2.4.4.2 Object Code Interface to CYBILIO

Before a program which uses CYBILIO can be executed, it must be linked with the CYBILIO object modules which are located on the CYBIL run-time library.

>>>>> to be supplied <<<<<<<<

2.4.5 EOS

2.4.5.1 SOURCE CODE INTERFACE TO CYBILIO

>>>>>> to be supplied <<<<<<

2.4.5.2 Object Code Interface to CYBILIO

Before a program which uses CYBILIO can be executed, it must be linked with the CYBILIO object modules which are located on the CYBIL run-time library.

>>>>> to be supplied <<<<<<<<

2.0 CYBILIO FILES
2.4.6 APOLLO AEGIS

2.4.6 APOLLO AEGIS

2.4.6.1 Source Code Interface to CYBILIO

To interface to CYBILIO, a CYBIL program module must include the relevant type and procedure declarations. These can be INCLUDEd from the /CYBIL/INS directory. The form that the INCLUDE directive should take is as follows:

```
INCLUDE '/cybil/ins/cybilio'
```

All CYBILIO type and procedure declarations are in the above.

2.4.6.2 Object Code Interface to CYBILIO

Before a program which uses CYBILIO can be executed, it must be linked with the CYBILIO object modules which are located on the CYBIL run-time library.

The name of the CYBIL run-time library is CYBALIB.BIN. Linking to the CYBILIO object modules may be done by including /CYBIL/BIN/CYBALIB.BIN as a directive to the BIND command.

3.0 I/O PROCEDURES

3.1 GENERAL PROCEDURES AND FUNCTIONS

The following are general procedures and functions. That is, they may be used with binary files, record files, text files, or display files.

3.1.1 OPENING AND CLOSING FILES

3.1.1.1 cyp$open file

```
{* ZCYPOF  cyp$open_file *}

  PROCEDURE [XREF] cyp$open_file ALIAS 'ZCYPOF'
   (    file_name: cyt$file_name;
        file_specifications: cyt$file_specifications;
    VAR file: cyt$file;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc ost$status
*copyc cyt$file_name
*copyc cyt$file_specifications
*copyc cyt$file
*copyc cye$exception_conditions
?? POP ??
```

This procedure opens the file specified by the FILE_NAME parameter. The length of the FILE_NAME and the characters included in the FILE_NAME must conform to the operating system dependent requirements or the open will be aborted and abnormal status will be returned in the status variable.

The FILE_SPECIFICATION parameter specifies how the file is to be used. If a NIL value is specified for this parameter, the following defaults are selected:

close_file_disposition	cyc$default_file_disposition
file_access	cyc$read_write
file_character_set	cyc$ascii612 for NOS implementations, cyc$ascii812 for NOS/BE implementations, and cyc$ascii for all other implementations
file_existence	cyc$new_or_old_file
file_kind	cyc$record_file
new_page_procedure	cyc$omit_page_procedure
page_format	cyc$burstable_form
open_position	cyc$default_open_position
page_length	system dependent
page_width	system dependent
file_contents	cyc$unknown_contents
file_processor	cyc$unknown_processor

If one or more of the file specifications are not specified, CYBILIO will use the default value for that specification. See the section on CYBIL types for additional information about file_specifications, the values that may be specified, and defaults for unspecified file specifications.

The FILE parameter returns a pointer that must be used on all other calls to CYBILIO. Attempting to call a CYBILIO procedure with an undefined or user-altered pointer will have unpredictable results.

3.1.1.2 cyp$close_file

```
{* ZCYPCF  cyp$close_file *}

  PROCEDURE [XREF] cyp$close_file ALIAS 'ZCYPCF' (file: cyt$file;
        file_position: cyt$open_close_position;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc ost$status
*copyc cyt$file
*copyc cyt$open_close_position
*copyc cye$exception_conditions
?? POP ??
```

This procedure closes the specified FILE.

The FILE_POSITION parameter specifies the position of the file at close.

REMARK:

The close_file_disposition record of the file specifications specified when the file was opened will determine the disposition of the file. That is, the file will be retained, returned, unloaded, or deleted.

3.1.2 POSITIONING FILES

3.1.2.1 cyp$position file at beginning

```
{* ZCYPPFB  cyp$position_file_at_beginning *}

  PROCEDURE [XREF] cyp$position_file_at_beginning ALIAS 'ZCYPPFB'
    (    file: cyt$file;
     VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc ost$status
*copyc cyt$file
*copyc cye$exception_conditions
?? POP ??
```

This procedure positions the specified FILE at its beginning of information.

3.1.2.2 cyp$position file at end

```
{* ZCYPPFE  cyp$position_file_at_end *}

  PROCEDURE [XREF] cyp$position_file_at_end ALIAS 'ZCYPPFE'
    (    file: cyt$file;
     VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc ost$status
*copyc cyt$file
*copyc cye$exception_conditions
?? POP ??
```

This procedure positions the specified FILE at its end of information.

3.1.3 FILE LENGTH INTERROGATION

3.1.3.1 cyp$length_of_file

```
{* ZCYPLOF  cyp$length_of_file *}

  FUNCTION [XREF] cyp$length_of_file ALIAS 'ZCYPLOF'
    (file: cyt$file): integer;

?? PUSH (LISTEXT := ON) ??
*copyc cyt$file
?? POP ??
```

This function returns the length of the specified FILE. The length is the number of cells in the file.

3.1.4 FILE STRUCTURE CREATION/DETECTION

CYBILIO supports the subdivision of files into levels of logical structure: records, blocks, and partitions. The End-Of-Information can only be implicitly created (i.e., the End-Of-Information follows the physically last item written on a file); but it can be explicitly detected.

3.1.4.1 File Structure Creation

Record subdivisions are created through the cyp$put_next_record and cyp$put_next_line procedure calls. In addition, an end of record is created through the cyp$put_partial_record procedure call when the last_part_of_record parameter is true and through the cyp$put_partial_line procedure call when the last_part_of_line parameter is true. The cyp$write_end_of_record and cyp$write_end_of_line procedures also create an end of record in a file.

Blocks and partitions are created through special procedure calls.

3.1.4.1.1 CYP$WRITE END OF BLOCK

```
{* ZCYPWEB  cyp$write_end_of_block *}

  PROCEDURE [XREF] cyp$write_end_of_block ALIAS 'ZCYPWEB'
   (    file: cyt$file;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc cyt$file
*copyc ost$status
*copyc cye$exception_conditions
?? POP ??
```

This procedure writes an End Of Block on the specified FILE.

REMARKS:

If the last write to the specified file was a partial write, that write is completed then the end of block is written.

Attempting a cyp$write_end_of_block to a file NOT opened for file_access = cyc$write or file_access = cyc$read_write will return cyc$incorrect_output_request in the status variable.

3.0 I/O PROCEDURES
3.1.4.1.2 CYP$WRITE_END_PARTITION

3.1.4.1.2 CYP$WRITE END PARTITION

```
{* ZCYPWEP  cyp$write_end_of_partition *}

  PROCEDURE [XREF] cyp$write_end_of_partition ALIAS 'ZCYPWEP'
   (    file: cyt$file;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc cyt$file
*copyc ost$status
*copyc cye$exception_conditions
?? POP ??
```

This procedure writes an End of Partition on the specified file.

REMARKS:

If the last write to the specified file was a partial write, that write is completed then the end of partition is written.

Attempting a cyp$write_end_of_partition to a file NOT opened for file_access = cyc$write or file_access = cyc$read_write will return cyc$incorrect_output_request in the status variable.

3.1.4.2 File Structure Detection

3.1.4.2.1 CYP$CURRENT FILE POSITION

```
{* ZCYPCFP  cyp$current_file_position *}

  FUNCTION [XREF] cyp$current_file_position ALIAS 'ZCYPCFP'
    (file: cyt$file): cyt$current_file_position;

?? PUSH (LISTEXT := ON) ??
*copyc cyt$file
*copyc cyt$current_file_position
?? POP ??
```

This function returns the current position of the specified FILE. The current position reflects the position of the file with respect to the logical subdivisions of the file. That is, the file is positioned at: beginning of information, end of information, in the middle of a record, at the end of a record, at the end of a block, or at the end of a partition.

Following any type of read or positioning operation, the function returns the current file position. Following most types of write operations, this function will return cyc$end_of_information. If the previous operation was a write to a binary type file, this function returns cyc$middle_of_record unless the write extended the length of the file in which case the function returns cyc$end_of_information.

3.1.5 OPERATING SYSTEM TYPE INTERROGATION

3.1.5.1 cyp$operating_system

```
{* ZCYPOS  cyp$operating_system *}

  FUNCTION [XREF] cyp$operating_system ALIAS 'ZCYPOS': cyt$system_type;

?? PUSH (LISTEXT := ON) ??
*copyc cyt$system_type
?? POP ??
```

This function returns a value that identifies the OPERATING_SYSTEM on which a program is running.

This function allows a user to write a program that can run on more than one operating system by including conditional code that handles operating system dependencies. See the copy_binary_file example in Appendix A for an example of how this function might be used.

3.2 RECORD FILE PROCEDURES

3.2.1 READING AND WRITING RECORD FILES

The data transfer procedures for record type files (like any programmer defined procedures in CYBIL) must have parameters of a specific CYBIL type. To transfer data to/from a record type file, the CYBIL type of the parameter that specifies the data to be read or written must match the CYBIL type of the program variable that contains the data to be read or written. The CYBILIO procedures that perform reads and writes on record type files require that the data be specified as a pointer to a CYBIL sequence. Programs that wish to use the record type file procedure interfaces must therefore specify the data as a variable of type pointer to CYBIL sequence. This pointer is usually defined by using the CYBIL #SEQ function.

For example, given the following CYBIL variable declarations:

```
VAR
  data_item_1: my_data_type,
  data_item_2: ↑my_data_type,
  data_item_3: ↑array [1 .. 50] of my_data_type;
```

pointers to CYBIL sequences may be defined as follows:

```
#SEQ (data_item_1)
#SEQ (data_item_2↑)
#SEQ (data_item_3↑)
#SEQ (data_item_3↑ [5])
```

The sample programs in the appendices provide examples of the use of the #SEQ cybil function to pass data to/from the record type file read/write procedures.

Data is read from or written to record type files as full or partial records. These records are NOT to be confused with the CYBIL record type.

3.2.1.1 cyp$put_next_record

```
{* ZCYPPNR  cyp$put_next_record *}

  PROCEDURE [XREF] cyp$put_next_record ALIAS 'ZCYPPNR'
   (    record_file: cyt$file;
        pointer_to_source: ↑SEQ ( * );
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc cyt$file
*copyc ost$status
*copyc cye$exception_conditions
?? POP ??
```

This procedure writes a record on the specified file.

The POINTER_TO_SOURCE parameter specifies the data to be written. The data is written as a complete record. If the last write to the file was cyp$put_partial_record, that record is first completed and then the data in POINTER_TO_SOURCE is written as a new complete record.

REMARKS:

The end of information on a record type file immediately follows the data last written. Thus, writing to a record type file, positioning the file to its beginning or performing a backward record skip, and again writing to the file will result in "lost" data.

Attempting a cyp$put_next_record to a file NOT opened as file_kind = cyc$record_file will return cye$incorrect_operation in the status variable.

Attempting a cyp$put_next_record to a file NOT opened for file_access = cyc$write or file_access = cyc$read_write will return cye$incorrect_output_request in the status variable.

3.2.1.2 cyp$put_partial_record

```
{* ZCYPPPR  cyp$put_partial_record *}

  PROCEDURE [XREF] cyp$put_partial_record ALIAS 'ZCYPPPR'
   (    record_file: cyt$file;
        pointer_to_source: ↑SEQ ( * );
        last_part_of_record: boolean;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc cyt$file
*copyc ost$status
*copyc cye$exception_conditions
?? POP ??
```

This procedure writes a partial record on the specified file.

The POINTER_TO_SOURCE parameter specifies the data to be written.

The LAST_PART_OF_RECORD specifies whether or not more data can be appended to the current record. If LAST_PART_OF_RECORD is TRUE, the data specified by POINTER_TO_SOURCE is written to the file and the record is terminated. The next full or partial write to the file will begin a new record. If LAST_PART_OF_RECORD is FALSE, the data specified by POINTER_TO_SOURCE is written to the file but the record is not terminated. Additional data can be appended to the record if the next write to the file is a cyp$put_partial_record.

REMARKS:

The end of information on a record type file immediately follows the data last written. Thus, writing to a record type file, positioning the file to its beginning or performing a backward record skip, and again writing to the file will result in "lost" data.

Attempting a cyp$put_partial_record to a file NOT opened as file_kind = cyc$record_file will return cye$incorrect_operation in the status variable.

Attempting a cyp$put_partial_record to a file NOT opened for file_access = cyc$write or file_access = cyc$read_write will return cye$incorrect_output_request in the status variable.

3.2.1.3 cyp$write end of record

```
{* ZCYPWER  cyp$write_end_of_record *}

  PROCEDURE [XREF] cyp$write_end_of_record ALIAS 'ZCYPWER'
   (    record_file: cyt$file;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc cyt$file
*copyc ost$status
*copyc cye$exception_conditions
?? POP ??
```

This procedure writes an end-of-record to the specified FILE. If the last write to the FILE was partial, that record is completed; otherwise an empty record results.

REMARKS:

Attempting a cyp$write_end_of_record to a file NOT opened as file_kind = cyc$record_file will return cye$incorrect_operation in the status variable.

Attempting a cyp$write_end_of_record to a file NOT opened for file_access = cyc$write or file_access = cyc$read_write will return cye$incorrect_output_request in the status variable.

3.2.1.4 cyp$get next record

```
{* ZCYPGNR  cyp$get_next_record *}

  PROCEDURE [XREF] cyp$get_next_record ALIAS 'ZCYPGNR'
   (    record_file: cyt$file;
        pointer_to_target: ↑SEQ ( * );
    VAR number_of_cells_read: integer;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc cyt$file
*copyc ost$status
*copyc cye$exception_conditions
?? POP ??
```

This procedure reads the next record from a record type file.

The POINTER_TO_TARGET parameter specifies the data structure into which data is to be read.

The NUMBER_OF_CELLS_READ parameter returns the number of cells actually read.

REMARKS:

If the current file position is not at the beginning of a record, the file is positioned forward to the beginning of the next record, block, or partition before the read begins.

CYBILIO reads data from the file until it encounters the end of the record or the end of the data structure specified by POINTER_TO_TARGET. NUMBER_OF_CELLS_READ will return the number of data cells actually read into the data structure specified by POINTER_TO_TARGET

If the read terminates because the end of the record was encountered, the cyp$current_file_position function will return cyc$end_of_record. If the read terminates because CYBILIO encountered the end of the POINTER_TO_TARGET data structure, the cyp$current_file_position function will return cyc$middle_of_record. To read the remainder of the record, the program must issue cyp$get_partial_record calls until the cyp$current_file_position function returns a value of cyc$end_of_record.

If CYBILIO encounters the end of a block, no data is read, the

NUMBER_OF_CELLS_READ returns a value of 0 (zero), and the cyp$current_file_position function will return a value of cyc$end_of_block.

If CYBILIO encounters the end of a partition, no data is read, the NUMBER_OF_CELLS_READ returns a value of 0 (zero), and the cyp$current_file_position function will return a value of cyc$end_of_partition.

If CYBILIO encounters the end of information, no data is read, the NUMBER_OF_CELLS_READ returns a value of 0 (zero), and the cyp$current_file_position function will return a value of cyc$end_of_information.

Attempting a cyp$get_next_record to a file NOT opened as file_kind = cyc$record_file will return cye$incorrect_operation in the status variable.

Attempting a cyp$get_next_record to a file opened for file_access = cyc$write will return cye$incorrect_input_request in the status variable.

3.2.1.5 cyp$get_partial_record

```
{* ZCYPGPR  cyp$get_partial_record *}

  PROCEDURE [XREF] cyp$get_partial_record ALIAS 'ZCYPGPR'
   (    record_file: cyt$file;
        pointer_to_target: ↑SEQ ( * );
    VAR number_of_cells_read: integer;
    VAR last_part_of_record: boolean;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc cyt$file
*copyc ost$status
*copyc cye$exception_conditions
?? POP ??
```

This procedure reads a portion of a record from a record type file.

The POINTER_TO_TARGET parameter specifies the data structure into which data is to be read.

The NUMBER_OF_CELLS_READ parameter returns the number of cells actually read.

CYBILIO begins reading at the current position of the file.

REMARKS:

CYBILIO reads data from the file until it encounters the end of the record or the end of the data structure specified by POINTER_TO_TARGET. NUMBER_OF_CELLS_READ will return the number of data cells actually read into the data structure specified by POINTER_TO_TARGET

If the read terminates because the end of the record was encountered, the cyp$current_file_position function will return cyc$end_of_record. If the read terminates because CYBILIO encountered the end of the POINTER_TO_TARGET data structure, the cyp$current_file_position function will return cyc$middle_of_record. To read the remainder of the record, the program must issue cyp$get_partial_record calls until the cyp$current_file_position function returns a value of cyc$end_of_record.

If CYBILIO encounters the end of a block, no data is read, the NUMBER_OF_CELLS_READ returns a value of 0 (zero), and the

cyp$current_file_position function will return a value of cyc$end_of_block.

If CYBILIO encounters the end of a partition, no data is read, the NUMBER_OF_CELLS_READ returns a value of 0 (zero), and the cyp$current_file_position function will return a value of cyc$end_of_partition.

If CYBILIO encounters the end of information, no data is read, the NUMBER_OF_CELLS_READ returns a value of 0 (zero), and the cyp$current_file_position function will return a value of cyc$end_of_information.

Attempting a cyp$get_partial_record to a file NOT opened as file_kind = cyc$record_file will return cye$incorrect_operation in the status variable.

Attempting a cyp$get_partial_record to a file opened for file_access = cyc$write will return cye$incorrect_input_request in the status variable.

3.2.2 RECORD FILE POSITIONING

In addition to the cyp$position_file_at_beginning and cyp$position_file_at_end procedure interfaces, record type files may be positioned forward or backward one or more records, blocks or partitions. Positioning may only be performed on record files that are opened for file_access of cyc$read or cyc$read_write.

3.2.2.1 cyp$position_record_file

```
{* ZCYPPRF  cyp$position_record_file *}

  PROCEDURE [XREF] cyp$position_record_file ALIAS 'ZCYPPRF'
    (     record_file: cyt$file;
          direction: cyt$skip_direction;
          count: integer;
          unit: cyt$skip_unit;
     VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc cyt$file
*copyc ost$status
*copyc cyt$skip_direction
*copyc cyt$skip_unit
*copyc cye$exception_conditions
?? POP ??
```

This procedure allows a record type file to be repositioned.

The DIRECTION parameter specifies forward or backward positioning, COUNT specifies the number of units the file is to be positioned, and UNIT specifies positioning by records, blocks, or partitions.

REMARKS:

The position of the file after a positioning operation depends on the positioning unit (records, blocks, or partitions), the initial file position, the number of units positioned, and the positioning direction. The following table lists positioning results assuming that no boundary condition is detected before the positioning count is exhausted.

CYP$POSITION_RECORD_FILE Results

file position before the positioning operation	positioning operation	result
Positioning by records:		
cyc$beginning_of_information, cycend_of_record, cycend_of_block, cyc$end_of_partition, cyc$end_of_information	Position forward or backward zero records.	No movement; the file remains the same as before the positioning operation.
cyc$middle_of_record	Position forward zero records.	The file is positioned to the end of the current record.
cyc$middle_of_record	Position backward zero records.	The file is positioned to the end of the preceeding record.
End of record N	Position forward one or more (M) records.	The file is positioned to the end of record N + M.
End of record N	Position backward one or more (M). records.	The file is positioned to the end of record N - M.
Positioning by blocks:		
cyc$beginning_of_information, cycend_of_block, cycend_of_partition, cyc$end_of_information	Position forward or backward zero blocks.	No movement; the file remains positioned the same as before the positioning operation.
cyc$middle_of_record, cyc$end_of_record	Position forward zero blocks.	The file is positioned to the end of the current block.
cyc$middle_of_record	Position backward	The file is positioned

cyc$end_of_record	zero blocks.	to the beginning of the current block.
End of block N	Position forward one or more (M) blocks.	The file is positioned to the end of block N + M.
End of block N	Position backward one or more (M) blocks.	The file is positioned to the end of block N - M.
Positioning by partitions:		
cyc$beginning_of_information, cyc$end_of_information	Position forward or backward zero partitions.	No movement; the file remains positioned the same as before the positioning operation.
cyc$middle_of_record, cyc$end_of_record, cycend_of_block, cycend_of_partition	Position forward zero partitions.	The file is positioned to the beginning of the next partition.
cyc$middle_of_record, cyc$end_of_record, cycend_of_block, cycend_of_partition	Position backward zero partitions.	The file is positioned to the beginning of the current partition.
cyc$middle_of_record, cyc$end_of_record, cycend_of_block, cycend_of_partition	Position forward one or more (M) partitions.	The file is positioned to the beginning of partition (current + M +1).
cyc$middle_of_record, cyc$end_of_record, cycend_of_block, cycend_of_partition	Position backward one or more (M) partitions.	The file is positioned to the beginning of partition (current - M).

The information in the preceeding table assumes that no boundary conditions are encountered during the positioning operation. If cyp$position_record_file encounters a boundary condition before the COUNT is exhausted, the positioning operation stops at the boundary and

cye$premature_end_of_operation will be returned in the status variable.

The following are the boundary conditions:

- o A position forward by records encounters an end of block, end of partition or end of information.
- o A position forward by blocks encounters an end of partition or end of information.
- o A position forward by partitions encounters end of information.
- o A position backwards by records encounters an end of block, end of partition or beginning of information.
- o A position backwards by blocks encounters an end of partition or beginning of information.
- o A position backwards by partitions encounters beginning of information.

Attempting a cyp$position_record_file to a file NOT opened as file_kind = cyc$record_file will return cye$incorrect_operation in the status variable.

Attempting a cyp$position_record_file to a file opened for file_access = cyc$write will return cye$incorrect_input_request in the status variable.

3.3 BINARY FILE PROCEDURES

3.3.1 READING AND WRITING BINARY FILES

The data transfer procedures for binary type files (like any programmer defined procedures in CYBIL) must have parameters of a specific CYBIL type. To transfer data to/from a binary type file, the CYBIL type of the parameter that specifies the data to be read or written must match the CYBIL type of the program variable that contains the data to be read or written. The CYBILIO procedures that perform reads and writes on binary type files require that the data be specified as a pointer to a CYBIL sequence. Programs that wish to use the binary type file procedure interfaces must therefore specify the data as a variable of type pointer to CYBIL sequence. This pointer is usually defined by using the CYBIL #SEQ function.

For example, given the following CYBIL variable declarations:

```
VAR
  data_item_1: my_data_type,
  data_item_2: ↑my_data_type,
  data_item_3: ↑array [1 .. 50] of my_data_type;
```

pointers to CYBIL sequences may be defined as follows:

```
#SEQ (data_item_1)
#SEQ (data_item_2↑)
#SEQ (data_item_3↑)
#SEQ (data_item_3↑ [5])
```

The sample programs in the appendices provide examples of the use of the #SEQ cybil function to pass data to/from the binary type file read/write procedures.

Any structure to be found in a binary type file must be provided for and interpreted by the user program. CYBILIO simply treats binary files as a sequence of cells. Calls to the binary type file read and write procedure interfaces simply result in a mapping of cells between the file and the CYBIL program variable.

Binary files may be read and written in a random, sequential, or combination random/sequential manner. Random access of binary files is possible via the FILE_KEY parameter on the binary file procedure calls. The FILE_KEY may be viewed as an offset pointer that marks cell addresses within a binary file. It is important to note that CYBILIO does NOT

maintain a directory of FILE_KEYs for binary files. It is the user's responsibility to create and maintain any directories that may be required. Refer to Appendix B for an example of a binary file directory.

When a binary file is opened, the FILE_KEY is undefined. If the file is to be accessed via the random access procedures cyp$put_keyed_binary and cyp$get_keyed_binary, the FILE_KEY must first be equated to the current (open) position of the file. This may be done by making a call to the cyp$get_next_binary procedure or the cyp$binary_file_key function.

3.3.1.1 cyp$put_next_binary

```
{* ZCYPPNB  cyp$put_next_binary *}

  PROCEDURE [XREF] cyp$put_next_binary ALIAS 'ZCYPPNB'
   (    binary_file: cyt$file;
        pointer_to_source: ↑SEQ ( * );
    VAR file_key: integer;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc cyt$file
*copyc ost$status
*copyc cye$exception_conditions
?? POP ??
```

This procedure writes data to a file opened as cyc$binary_file. The data is written to the current position of the file.

The POINTER_TO_SOURCE parameter specifies the data to be written.

The FILE_KEY parameter returns the "file cell address" at which the write started.

REMARKS:

The end of information for a binary type file follows the last physical cell written to the file. Thus, writes can be performed to the file, the file repositioned backwards, and another write performed without affecting the end of information.

The size of the data block written to a binary file is determined by the POINTER_TO_SOURCE parameter. CYBILIO does NOT perform any blocking of data. Thus, writing varying length blocks of data at "random" file addresses can cause previously written data blocks to be partially or fully overwritten.

Attempting a cyp$put_next_binary to a file NOT opened as file_kind = cyc$binary_file will return cye$incorrect_operation in the status variable.

Attempting a cyp$put_next_binary to a file NOT opened for file_access = cyc$write or file_access = cyc$read_write will return cye$incorrect_output_request in the status variable.

3.3.1.2 cyp$put_keyed_binary

```
{* ZCYPPKB  cyp$put_keyed_binary *}

  PROCEDURE [XREF] cyp$put_keyed_binary ALIAS 'ZCYPPKB'
    (   binary_file: cyt$file;
        pointer_to_source: ↑SEQ ( * );
        file_key: integer;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc cyt$file
*copyc ost$status
*copyc cye$exception_conditions
?? POP ??
```

This procedure writes data to a binary type file.

The POINTER_TO_SOURCE parameter specifies the data to be written.

The FILE_KEY parameter specifies the "file cell address" at which the write is to begin.

REMARKS:

The size of the data block written to a binary file is determined by the POINTER_TO_SOURCE parameter. CYBILIO does NOT perform any blocking of data. Thus, writing varying length blocks of data at "random" file addresses can cause previously written data blocks to be partially or fully overwritten.

Attempting a cyp$put_keyed_binary to a file NOT opened as file_kind = cyc$binary_file will return cye$incorrect_operation in the status variable.

Attempting a cyp$put_keyed_binary to a file NOT opened for file_access = cyc$write or file_access = cyc$read_write will return cye$incorrect_output_request in the status variable.

3.3.1.3 cyp$get next binary

```
{* ZCYPGNB  cyp$get_next_binary *}

  PROCEDURE [XREF] cyp$get_next_binary ALIAS 'ZCYPGNB'
   (    binary_file: cyt$file;
        pointer_to_target: ↑SEQ ( * );
    VAR file_key: integer;
    VAR number_of_cells_read: integer;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc cyt$file
*copyc ost$status
*copyc cye$exception_conditions
?? POP ??
```

This procedure reads data from a binary type file. The data is read from the current position of the file.

The POINTER_TO_TARGET parameter specifies the data structure into which data is to be read.

The FILE_KEY parameter returns the "file cell address" from which the read began.

The NUMBER_OF_CELLS_READ parameter returns the number of cells actually read. The value returned is normally the size of the data structure referenced by POINTER_TO_TARGET. However, if end of block, end of partition, or end of information is detected during a read, NUMBER_OF_CELLS_READ returns the only of cells read before the end of block, end of partition or end of information was detected.

REMARKS:

The cyp$current_file_position function returns cyc$middle_of_record following a read from a binary type file unless NUMBER_OF_CELLS_READ returned a value of 0 (zero). In this case, cyp$current_file_position would return cycend_of_block, cycend_of_partition, or cyc$end_of_information to indicate which file boundary condition was encountered.

Attempting a cyp$get_next_binary to a file NOT opened as file_kind = cyc$binary_file will return cye$incorrect_operation in the status

variable.

Attempting a cyp$get_next_binary to a file opened for file_access = cyc$write will return cye$incorrect_input_request in the status variable.

3.3.1.4 cyp$get_keyed_binary

```
{* ZCYPGKB  cyp$get_keyed_binary *}

  PROCEDURE [XREF] cyp$get_keyed_binary ALIAS 'ZCYPGKB'
    (    binary_file: cyt$file;
         pointer_to_target: ↑SEQ ( * );
         file_key: integer;
     VAR number_of_cells_read: integer;
     VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc cyt$file
*copyc ost$status
*copyc cye$exception_conditions
?? POP ??
```

This procedure reads data from a binary type file.

The POINTER_TO_TARGET parameter specifies the data structure into which data is to be read.

The FILE_KEY parameter specifies the "file cell address" at which the read is to begin.

The NUMBER_OF_CELLS_READ parameter returns the number of cells actually read. The value returned is normally the size of the data structure referenced by POINTER_TO_TARGET. However, if end of block, end of partition, or end of information is detected during a read, NUMBER_OF_CELLS_READ returns the only of cells read before the end of block, end of partition or end of information was detected.

REMARKS:

The cyp$current_file_position function returns cyc$middle_of_record following a read from a binary type file unless NUMBER_OF_CELLS_READ returned a value of 0 (zero). In this case, cyp$current_file_position would return cycend_of_block, cycend_of_partition, or cyc$end_of_information to indicate which file boundary condition was encountered.

If FILE_KEY specifies a cell beyond the end of information, no data is read, cyp$get_keyed_binary will return cye$key_past_eoi in the status variable and the position of the file remains unchanged.

3.0 I/O PROCEDURES
3.3.1.4 cyp$get_keyed_binary

Attempting a cyp$get_keyed_binary to a file NOT opened as file_kind = cyc$binary_file will return cye$incorrect_operation in the status variable.

Attempting a cyp$get_keyed_binary to a file opened for file_access = cyc$write will return cye$incorrect_input_request in the status variable.

3.3.2 BINARY FILE POSITIONING

In addition to the cyp$position_file_at_beginning and cyp$position_file_at_end procedure interfaces, binary type files may be positioned to any random "file address" within the bounds of the file.

3.3.2.1 cyp$position binary at key

```
{* ZCYPPBK  cyp$position_binary_at_key *}

  PROCEDURE [XREF] cyp$position_binary_at_key ALIAS 'ZCYPPBK'
   (    binary_file: cyt$file;
        file_key: integer;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc cyt$file
*copyc ost$status
*copyc cye$exception_conditions
?? POP ??
```

This procedure positions a binary type file to a specified "file cell address".

The FILE_KEY parameter specifies the "file cell address" to which the file is to be positioned.

REMARKS:

If FILE_KEY specifies a cell beyond the end of information, cyp$get_keyed_binary will return cye$key_past_eoi in the status variable and the position of the file remains unchanged.

Attempting a cyp$position_binary_at_key to a file NOT opened as file_kind = cyc$binary_file will return cye$incorrect_operation in the status variable.

3.3.3 BINARY FILE POSITION INTERROGATION

3.3.3.1 cyp$binary_file_key

```
{* ZCYPBFK  cyp$binary_file_key *}

  FUNCTION [XREF] cyp$binary_file_key ALIAS 'ZCYPBFK'
    (binary_file: cyt$file): integer;

?? PUSH (LISTEXT := ON) ??
*copyc cyt$file
?? POP ??
```

This function returns the "file cell address" at which a binary type file is currently positioned. If this call is immediately preceeded by a "get" or "put" procedure call, the value returned points to the last cell transferred + 1. If this call is immediately preceeded by a cyp$position_binary_at_key call, the value returned is the "file cell address" to which the file was positioned.

REMARKS:

Attempting a cyp$binary_file_key to a file NOT opened as file_kind = cyc$binary_file will return a meaningless result.

3.4 TEXT AND DISPLAY FILES

3.4.1 READING AND WRITING TEXT FILES AND DISPLAY FILES

Data is transferred to and from text files and display files in terms of lines or partial lines. Internally these (partial) lines are represented by CYBIL strings of characters. Externally (on the file) lines may be represented in 8-bit ASCII, 6-bit display code, NOS 6/12-bit ASCII, or "8 out of 12 bit" ASCII,. This external representation is operating system dependent and may be specified through the file_specifications when the file is opened. Thus, data transfers involving text files or display files may imply a translation between these character sets (unlike binary and record file transfers in which the data are not modified).

The maximum length of lines written to text files or display files and the page size for display files may be specified via the file_specifications parameter on the call to cyp$open_file.

The procedures in this section apply to both text files and display files.

3.4.1.1 cyp$put_next_line

```
{* ZCYPPNL  cyp$put_next_line *}

  PROCEDURE [XREF] cyp$put_next_line ALIAS 'ZCYPPNL'
   (    file: cyt$file;
        line: string ( * <= cyc$max_page_width);
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc cyt$file
*copyc ost$status
*copyc cyt$page_width
*copyc cye$exception_conditions
?? POP ??
```

This procedure writes a string of characters to the specified FILE.

The LINE parameter specifies the string of characters to be written. The characters in LINE are written as a complete line. If the last write to the FILE was a partial line, that line is first completed, and then the characters in LINE are written.

REMARKS:

If the length of the character string exceeds the page width, the line will be truncated.

In the case of a file opened as kind = cyc$display_file, format control characters are automatically prefixed to the LINE by CYBILIO. In addition, if displaying the line causes the display page length to be exceeded, CYBILIO will invoke the page overflow mechanism.

Attempting a cyp$put_next_line to a file NOT opened as file_kind = cyc$text_file or file_kind = cyc$display_file will return cye$incorrect_operation in the status variable.

Attempting a cyp$put_next_line to a file NOT opened for file_access = cyc$write or file_access = cyc$read_write will return cye$incorrect_output_request in the status variable.

3.4.1.2 cyp$put_partial_line

```
{* ZCYPPPL  cyp$put_partial_line *}

  PROCEDURE [XREF] cyp$put_partial_line ALIAS 'ZCYPPPL'
   (    file: cyt$file;
        partial_line: string ( * <= cyc$max_page_width);
        last_part_of_line: boolean;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc cyt$file
*copyc ost$status
*copyc cyt$page_width
*copyc cye$exception_conditions
?? POP ??
```

This procedure writes a string of characters to the specified FILE.

The PARTIAL_LINE parameter specifies the string of characters to be written.

The LAST_PART_OF_LINE parameter specifies whether or not more characters can be written to the current line. If LAST_PART_OF_LINE is TRUE, an end-of-line is appended to the current line after the character string is written. If LAST_PART_OF_LINE is FALSE, subsequent cyp$put_partial_line calls may append data to the current line.

REMARKS:

In the case of a file opened as kind = cyc$display_file, CYBILIO automatically prefixes format control characters to the beginning of each new line. In addition, if LAST_PART_OF_LINE is TRUE and displaying the current line causes the display page length to be exceeded, CYBILIO will invoke the page overflow mechanism.

If the length of the current line exceeds the page width, the line will be truncated.

Attempting a cyp$put_partial_line to a file NOT opened as file_kind = cyc$text_file or file_kind = cyc$display_file will return cye$incorrect_operation in the status variable.

Attempting a cyp$put_partial_line to a file NOT opened for file_access

= cyc$write or file_access = cyc$read_write will return cye$incorrect_output_request in the status variable.

3.4.1.3 cyp$write end of line

```
{* ZCYPWEL  cyp$write_end_of_line *}

  PROCEDURE [XREF] cyp$write_end_of_line ALIAS 'ZCYPWEL'
   (    file: cyt$file;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc cyt$file
*copyc ost$status
*copyc cye$exception_conditions
?? POP ??
```

This procedure writes an end-of-line to the specified FILE. If the last write to the FILE was partial, that line is completed; otherwise an empty line results.

REMARKS:

Attempting a cyp$write_end_of_line to a file NOT opened as file_kind = cyc$text_file or file_kind = cyc$display_file will return cye$incorrect_operation in the status variable.

Attempting a cyp$write_end_of_line to a file NOT opened for file_access = cyc$write or file_access = cyc$read_write will return cye$incorrect_output_request in the status variable.

3.4.1.4 cyp$flush_line

```
{* ZCYPFL  cyp$flush_line *}

  PROCEDURE [XREF] cyp$flush_line ALIAS 'ZCYPFL'
   (    file: cyt$file;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc cyt$file
*copyc ost$status
*copyc cye$exception_conditions
?? POP ??
```

This procedure flushes the line buffer for the specified file. If the line buffer contains data, the line is terminated and then written to the specified file. If the line buffer contains no data, this procedure results in no operation on the file.

REMARKS:

Attempting a cyp$flush_line to a file NOT opened as file_kind = cyc$text_file or file_kind = cyc$display_file will return cye$incorrect_operation in the status variable.

Attempting a cyp$flush_line to a file NOT opened for file_access = cyc$write or file_access = cyc$read_write will return cye$incorrect_output_request in the status variable.

3.4.1.5 cyp$tab_file

```
{* ZCYPTF  cyp$tab_file *}

  PROCEDURE [XREF] cyp$tab_file ALIAS 'ZCYPTF'
   (    file: cyt$file;
        tab_column: cyt$page_width;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc cyt$file
*copyc ost$status
*copyc cyt$page_width
*copyc cye$exception_conditions
?? POP ??
```

This procedure positions a FILE to a specified column or position within a line. This procedure performs a WRITE to the FILE.

The TAB_COLUMN parameter specifies the column to which the file should be positioned.

REMARKS:

If TAB_COLUMN is less than or equal to the file's current column this procedure does nothing. Otherwise, sufficient space characters are written to FILE so that the next partial write to FILE will begin at the specified TAB_COLUMN.

If TAB_COLUMN is larger than the page width of the device associated with FILE, line truncation will occur when the line is written.

Attempting a cyp$tab_file to a file NOT opened as file_kind = cyc$text_file or file_kind = cyc$display_file will return cye$incorrect_operation in the status variable.

Attempting a cyp$tab_file to a file NOT opened for file_access = cyc$write or file_access = cyc$read_write will return cye$incorrect_output_request in the status variable.

3.4.1.6 cyp$skip_lines

```
{* ZCYPSL  cyp$skip_lines *}

  PROCEDURE [XREF] cyp$skip_lines ALIAS 'ZCYPSL'
   (    file: cyt$file;
        number_of_lines: integer;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc cyt$file
*copyc ost$status
*copyc cye$exception_conditions
?? POP ??
```

This procedure writes one or more blank lines to the specified FILE.

The NUMBER_OF_LINES parameter specifies the number of blank lines to be written. If the last write to FILE was partial, that line is first completed and then NUMBER_OF_LINES blank lines are written to the file.

REMARKS:

If the specified file was opened as kind = cyc$display_file and NUMBER_OF_LINES = -1, the next line written to the file will overwrite the current line. In addition, if NUMBER_OF_LINES + current line number exceeds the display page size, the page overflow mechanism will be invoked.

Attempting a cyp$skip_lines to a file NOT opened as file_kind = cyc$text_file or file_kind = cyc$display_file will return cye$incorrect_operation in the status variable.

Attempting a cyp$skip_lines to a file NOT opened for file_access = cyc$write or file_access = cyc$read_write will return cye$incorrect_output_request in the status variable.

3.4.1.7 cyp$get_next_line

```
{* ZCYPGNL  cyp$get_next_line *}

  PROCEDURE [XREF] cyp$get_next_line ALIAS 'ZCYPGNL'
   (    file: cyt$file;
    VAR line: string ( * <= cyc$max_page_width);
    VAR number_of_characters_read: integer;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc cyt$file
*copyc ost$status
*copyc cyt$page_width
*copyc cye$exception_conditions
?? POP ??
```

This procedure reads the next complete line from the specified FILE. If the previous transfer was partial, a skip to the end of that line is performed prior to this read.

The LINE parameter specifies the CYBIL string into which the line read. If the line from FILE is too long to fit into LINE, the line is truncated by skipping to the end of the line after the transfer is complete.

The NUMBER_OF_CHARACTERS_READ parameter returns the number of characters transferred into LINE.

REMARKS:

A line containing zero characters (i.e., the carriage return key was "hit" in the first position of the line or any empty line was written via a call to cyp$write_end_of_line) is returned to the CYBILIO user as an empty string.

Attempting a cyp$get_next_line to a file NOT opened as file_kind = cyc$text_file or file_kind = cyc$display_file will return cye$incorrect_operation in the status variable.

Attempting a cyp$get_next_line to a file opened for file_access = cyc$write will return cye$incorrect_input_request in the status variable.

3.4.1.8 cyp$get_partial_line

```
{* ZCYPGPL  cyp$get_partial_line *}

  PROCEDURE [XREF] cyp$get_partial_line ALIAS 'ZCYPGPL'
   (    file: cyt$file;
    VAR partial_line: string ( * <= cyc$max_page_width);
    VAR number_of_characters_read: integer;
    VAR last_part_of_line: boolean;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc cyt$file
*copyc ost$status
*copyc cyt$page_width
*copyc cye$exception_conditions
?? POP ??
```

This procedure reads the a character string from the specified FILE.

The PARTIAL_LINE parameter specifies the CYBIL string into which the character string is read.

The NUMBER_OF_CHARACTERS_READ parameter returns the number of characters transferred into PARTIAL_LINE.

The LAST_PART_OF_LINE parameter returns a TRUE value if the end of the line was encountered, and a value of FALSE otherwise.

REMARKS:

A line containing zero characters (i.e., the carriage return key was "hit" in the first position of the line or any empty line was written via a call to cyp$write_end_of_line) is returned to the CYBILIO user as an empty string.

Attempting a cyp$get_partial_line to a file NOT opened as file_kind = cyc$text_file or file_kind = cyc$display_file will return cye$incorrect_operation in the status variable.

Attempting a cyp$get_partial_line to a file opened for file_access = cyc$write will return cye$incorrect_input_request in the status variable.

3.4.2 TEXT AND DISPLAY FILE STATUS INTERROGATION

3.4.2.1 cyp$file_connected_to_terminal

```
{* ZCYPFCT  cyp$file_connected_to_terminal *}

  FUNCTION [XREF] cyp$file_connected_to_terminal ALIAS 'ZCYPFCT'
    (file: cyt$file): boolean;

?? PUSH (LISTEXT := ON) ??
*copyc cyt$file
?? POP ??
```

This function returns a value of TRUE if the file is connected to a terminal. Otherwise, a value of FALSE is returned.

This function call may be used to determine if the calling program needs to limit line size or perform any special data formatting for terminal FILEs.

REMARKS:

Attempting a cyp$file_connected_to_terminal to a file NOT opened as file_kind = cyc$text_file or file_kind = cyc$display_file will return cye$incorrect_operation in the status variable.

3.4.2.2 cyp$current_column

```
{* ZCYPCC  cyp$current_column *}

  FUNCTION [XREF] cyp$current_column ALIAS 'ZCYPCC'
    (file: cyt$file): cyt$page_width;

?? PUSH (LISTEXT := ON) ??
*copyc cyt$file
*copyc cyt$page_width
?? POP ??
```

This function returns the current column within the current line of the specified FILE; that is, the column at which the next read or write will begin.

REMARKS:

Attempting a cyp$current_column to a file NOT opened as file_kind = cyc$text_file or file_kind = cyc$display_file will return an undefined result.

3.4.2.3 cyp$page_width

```
{* ZCYPPW  cyp$page_width *}

  FUNCTION [XREF] cyp$page_width ALIAS 'ZCYPPW'
    (file: cyt$file): cyt$page_width;

?? PUSH (LISTEXT := ON) ??
*copyc cyt$file
*copyc cyt$page_width
?? POP ??
```

This function returns the page width associated with FILE.

REMARKS:

Attempting a cyp$page_width to a file NOT opened as file_kind = cyc$text_file or file_kind = cyc$display_file will return an undefined result.

3.5 DISPLAY FILES

3.5.1 PAGE OVERFLOW PROCESSING

CYBILIO counts the lines written to a display type file. When the number of lines written exceeds the page_length for the file, CYBILIO resets its line count to zero, invokes the page overflow mechanism, and again begins to count lines written to the display file. The page overflow mechanism is simply the sequence of events performed by CYBILIO whenever display page length is exceeded.

CYBILIO first checks for a user-supplied page overflow procedure. If one has been established through the file_specifications when the file was opened, the user specified procedure is called. In the absence of a user-specified page overflow procedure, CYBILIO checks to see if the file_specifications specified use of "standard" page headers. If standard headers have been selected, CYBILIO will format and display the header. If the user has neither specified a page overflow procedure nor the use of the standard header, CYBILIO simply performs a page eject. The sequence of events may be approximated as follows:

```
get next display line

IF (line_count + 1) > display page length THEN
  line_count := 0
  IF user-specified new_page_procedure THEN
    call user-specified procedure
  ELSEIF standard procedure selected THEN
    perform display page eject
    format and display standard header
    skip 1 line
  ELSE
    perform display page eject
  IFEND
IFEND

display the display line
```

3.0 I/O PROCEDURES
3.5.1 PAGE OVERFLOW PROCESSING

Standard page headers are either narrow format or wide format. The format is automatically selected by CYBILIO. If the page_width established when the file is opened is greater than or equal to 132, the wide format is selected; otherwise, the narrow format is selected. (The page_width is specified via the file_specifications parameter on the call to cyp$open_file.

The standard page headers are formatted as follows:

NARROW FORMAT

Line 1

Columns	1-46	string contained in the title field of the new_page_procedure record of the file_specifications specified when the file was opened.
Columns	48-55	date in mm/dd/yy format
Columns	62-72	'PAGE ' and page number

Line 2

Columns	1-22	Operating system version
Columns	48-59	Time in system default format or, if no default is available, in hh:mm:ss format

WIDE FORMAT

Columns	1-46	string contained in the title field of the new_page_procedure record of the file_specifications specified when the file was opened.
Columns	48-69	operating system version
Columns	91-98	date in mm/dd/yy format
Columns	110-121	Time in system default format or, if no default is available, in hh:mm:ss format
Columns	123-132	'PAGE ' and page number

All fields in the standard headers are displayed left-justified with blank fill to the right.

Standard title lines can be produced from within user-specified new page procedures through the use of the cyp$display_standard_title procedure.

3.5.2 DISPLAY FILE PROCEDURES AND FUNCTIONS

Files which are opened as file_kind = cyc$display_file may make use of special procedures for handling page overflow conditions and form layout.

The procedures and functions in this section apply only to display files.

3.5.2.1 cyp$start new display page

```
{* ZCYPSNP  cyp$start_new_display_page *}

  PROCEDURE [XREF] cyp$start_new_display_page ALIAS 'ZCYPSNP'
   (    display_file: cyt$file;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc cyt$file
*copyc ost$status
*copyc cye$exception_conditions
?? POP ??
```

This procedure invokes the CYBILIO page overflow mechanism.

REMARKS:

If the last write to the display file was a partial write, that line is terminated and then a new display page is started.

Attempting a cyp$start_new_display_page to a file NOT opened as file_kind = cyc$display_file will return cye$incorrect_operation in the status variable.

Attempting a cyp$start_new_display_page to a file opened NOT opened for file_access = cyc$write or file_access = cyc$read_write will return cye$incorrect_output_request in the status variable.

3.5.2.2 cyp$display_standard_title

```
{* ZCYPDST  cyp$display_standard_title *}

  PROCEDURE [XREF] cyp$display_standard_title ALIAS 'ZCYPDST'
       (file: cyt$file;
        title: string (* <= cyc$title_size);
        lines_after_title: cyt$page_length;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc cyt$file
*copyc ost$status
*copyc cyt$new_page_procedure
*copyc cyt$page_length
*copyc cye$exception_conditions
?? POP ??
```

This procedure formats and writes a standard title line to the specified file.

The TITLE parameter specifies the text that is to appear in columns 1 thru 46 in the standard title.

The SKIP_LINES parameter specifies the number of blank lines that are to appear between the standard title and the next display line.

REMARKS:

If the last write to the display file was a partial write, that display line is terminated and then standard title is written.

Attempting a cyp$display_standard_title to a file NOT opened as file_kind = cyc$display_file will return cye$incorrect_operation in the status variable.

Attempting a cyp$display_standard_title to a file NOT opened for file_access = cyc$write or file_access = cyc$read_write will return cye$incorrect_operation in the status variable.

3.5.2.3 cyp$position display page

```
{* ZCYPPDP  cyp$position_display_page *}

  PROCEDURE [XREF] cyp$position_display_page ALIAS 'ZCYPPDP'
   (    display_file: cyt$file;
        line_number: cyt$page_length;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc cyt$file
*copyc cyt$page_length
*copyc ost$status
*copyc cye$exception_conditions
?? POP ??
```

This procedure positions a DISPLAY_FILE at a specified line.

The LINE_NUMBER parameter specifies the display line at which the file is to be positioned.

REMARKS:

If LINE_NUMBER is greater than the current line number and less than or equal to page size, the file is positioned to that line on the current page. If LINE_NUMBER is less than or equal to the current line number, the page overflow mechanism is invoked and the file is positioned at LINE_NUMBER on the next page. If LINE_NUMBER is greater than the page size, the page overflow mechanism is invoked and the file will be positioned at the top of the next page.

If the last write to the display file wa a partial write, that line is terminated and then the display page is positioned.

Attempting a cyp$position_display_page to a file NOT opened as file_kind = cyc$display_file will return cye$incorrect_operation in the status variable.

Attempting a cyp$position_display_page to a file NOT opened for file_access = cyc$write or file_access = cyc$read_write will return cye$incorrect_operation in the status variable.

3.5.2.4 cyp$display_page_eject

```
{* ZCYPDPE  cyp$display_page_eject *}

  PROCEDURE [XREF] cyp$display_page_eject ALIAS 'ZCYPDPE'
   (    display_file: cyt$file;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc cyt$file
*copyc ost$status
*copyc cye$exception_conditions
?? POP ??
```

This procedure positions DISPLAY_FILE at the first line (top) of the next page. This procedure should only be called from a user-specified page overflow procedure.

REMARKS:

If the last write to the display file was a partial write, that line is terminated and then a display page eject is performed.

Attempting a cyp$display_page_eject to a file NOT opened as file_kind = cyc$display_file will return cye$incorrect_operation in the status variable.

Attempting a cyp$display_page_eject to a file NOT opened for file_access = cyc$write or file_access = cyc$read_write will return cye$incorrect_operation in the status variable.

3.5.2.5 cyp$current_display_line

```
{* ZCYPCDL  cyp$current_display_line *}

  FUNCTION [XREF] cyp$current_display_line ALIAS 'ZCYPCDL'
    (display_file: cyt$file): cyt$page_length;

?? PUSH (LISTEXT := ON) ??
*copyc cyt$file
*copyc cyt$page_length
?? POP ??
```

This function returns the number of the current line within the current page of DISPLAY_FILE.

REMARKS:

After any vertical spacing command (cyp$skip_lines, cyp$display_page_eject, cyp$position_display_page, etc.), the value returned is the next line to be displayed. After a write command (cypput_next_line, cypput_partial_line, cyp$write_end_of_line, etc.), the value returned is the line just displayed.

Attempting a cyp$current_display_line to a file NOT opened as file_kind = cyc$display_file will return an undefined result.

3.5.2.6 cyp$current_page_number

```
{* ZCYPCPN  cyp$current_page_number *}

  FUNCTION [XREF] cyp$current_page_number ALIAS 'ZCYPCPN'
    (display_file: cyt$file): integer;

?? PUSH (LISTEXT := ON) ??
*copyc cyt$file
?? POP ??
```

This function returns the DISPLAY_FILE's current page number.

REMARKS:

Attempting a cyp$current_page_number to a file NOT opened as file_kind = cyc$display_file will return an undefined result.

3.5.2.7 cyp$display_page_length

```
{* ZCYPDPL  cyp$display_page_length *}

  FUNCTION [XREF] cyp$display_page_length ALIAS 'ZCYPDPL'
    (display_file: cyt$file): cyt$page_length;

?? PUSH (LISTEXT := ON) ??
*copyc cyt$file
*copyc cyt$page_length
?? POP ??
```

This function returns the page length associated with DISPLAY_FILE.

REMARKS:

Attempting a cyp$display_page_length to a file NOT opened as file_kind = cyc$display_file will return an undefined result.

4.0 CYBILIO STATUS

4.1 CYBILIO STATUS MESSAGES

This section describes the status messages that may be returned either as a result of improper use of CYBILIO or as a result of detecting an error. If one of these conditions arises, the status condition will be returned in the status variable parameter.

In the message descriptions that follow, filename will be replaced by the name of the file in question when the message appears in the message template.

FILE NAME TOO LONG, filename

This message indicates the file name given has more characters than the target operating system will allow.

FILE NOT OPEN

This message indicates that an undefined variable of type cyt$file was passed to a CYBILIO procedure other than one of the open procedures. The file name is not known.

INCORRECT FILE NAME, filename

This message means that an attempt was made to open a file with a name that does not conform to the file naming conventions of the target operating system.

INCORRECT INPUT REQUEST FOR FILE filename

This message means that an attempt was made to read from a file that was opened only for output.

4.0 CYBILIO STATUS
4.1 CYBILIO STATUS MESSAGES

INCORRECT DISPLAY LINE POSITION FOR FILE filename

This message means that the cyp$position_display_page procedure was passed a line number less than 1.

INCORRECT TAB COLUMN FOR FILE filename

This message means that the cyp$tab_file procedure was passed a tab_column less than 1.

INCORRECT OPEN REQUEST FOR FILE filename

This message means that an invalid combination of parameters was given to an open procedure (e.g., "cycnew_file, cycread" is incorrect).

INVALID OPERATION ATTEMPTED ON FILE filename

This message means that an operation was attempted that does not match the file_kind specified for the file on the call to the open file procedure. For example, a cyp$get_next_binary attempted on a file opened with file_kind = cyc$text_file.

INCORRECT OUTPUT REQUEST FOR FILE filename

This message means that an attempt was made to write to a file that was opened only for input.

INCORRECT SKIP COUNT filename

This message indicates that the cyp$skip_lines procedure was passed a skip count less than -1.

KEY BEYOND E-O-I ON FILE filename

This message indicates that an attempt was made to perform a binary

4.0 CYBILIO STATUS
4.1 CYBILIO STATUS MESSAGES

file operation with a key that was outside the bounds of the file (i.e., the key did not specify a "random address" that is in the file).

PREMATURE END OF OPERATION ON FILE filename

This message means that a boundary condition was encountered during a cyp$position_record_file before the count was exhausted.

NO MEMORY TO OPEN FILE filename

This message means that there was insufficient space to allocate the descriptor and/or buffer for the file.

COULD NOT FIND FILE filename

This message means that an attempt was made to open an old file that CYBILIO cannot find.

FILE filename ALREADY EXISTS

This message means that an attempt was made to open a new file and a file with that name already exists.

4.2 CYBILIO STATUS CONDITIONS

The following is a list of the status conditions that may be returned by CYBILIO. The presence of these conditions may be tested by examining the condition field of the status variable used in the CYBILIO procedure call.

```
{* ZCYECIO  cye$exception_conditions *}

?? NEWTITLE := '¬¬¬¬¬¬ cybil i/o errors : CY 6200 .. 6299' ??
?? FMT (FORMAT := OFF) ??

  CONST
    cyc$min_ecc = (($INTEGER('C')*100(16))+$INTEGER('Y'))*1000000(16),
    cyc$max_ecc = cyc$min_ecc + 9999;

  CONST

    cyc$min_ecc_cybil_input_output  = cyc$min_ecc + 6200,

    cye$file_name_too_long          = cyc$min_ecc_cybil_input_output + 10,
    {E File name too long, +P.}

    cye$file_not_open               = cyc$min_ecc_cybil_input_output + 15,
    {E File NOT open.}

    cye$illegal_file_name           = cyc$min_ecc_cybil_input_output + 20,
    {E Incorrect file name, +P.}

    cye$illegal_input_request       = cyc$min_ecc_cybil_input_output + 25,
    {E Incorrect input request for +P.}

    cye$illegal_line_number         = cyc$min_ecc_cybil_input_output + 30,
    {E Incorrect display line position for +P.}

    cye$illegal_tab_column          = cyc$min_ecc_cybil_input_output + 31,
    {E Incorrect tab column for +P.}

    cye$illegal_open_request        = cyc$min_ecc_cybil_input_output + 35,
    {E Incorrect open request for +P.}

    cye$illegal_operation           = cyc$min_ecc_cybil_input_output + 37,
    {E Incorrect command issued to +P.}

    cye$illegal_output_request      = cyc$min_ecc_cybil_input_output + 40,
```

```
    {E Incorrect output request for +P.}

    cye$illegal_skip_count          = cyc$min_ecc_cybil_input_output + 45,
    {E Incorrect skip count +P.}

    cye$key_past_eoi                = cyc$min_ecc_cybil_input_output + 50,
    {E Key beyond E-O-I on +P.}

    cye$premature_end_of_operation  = cyc$min_ecc_cybil_input_output + 51,
    {E Premature end of operation on file +P.}

    cye$no_memory_to_open_file      = cyc$min_ecc_cybil_input_output + 55,
    {E No memory to open file +P.}

    cye$file_not_found              = cyc$min_ecc_cybil_input_output + 56,
    {E Could NOT find file +P.}

    cye$file_already_exists         = cyc$min_ecc_cybil_input_output + 57,
    {E File (+P) already exists.}

    cyc$max_ecc_cybil_input_output  = cyc$min_ecc_cybil_input_output + 99;

?? FMT (FORMAT := ON) ??
?? OLDTITLE ??
```

5.0 OPERATING SYSTEM DEPENDENT FEATURES

5.0 OPERATING SYSTEM DEPENDENT FEATURES

5.1 NOS/VE

5.1.1 DECK NAMES

Decks names are the same as the procedure or function declared within the deck. For example, the declaration for cyp$open_file is contained within deck cyp$open_file. The following table lists deck names which do not conform to this general guideline.

DECLARATION	DECK NAME
CYBILIO types	cyt$cybil_input_output
CYBILIO status_conditions	cye$exception_conditions
all declarations applicable to binary files	cyd$binary_file
all declarations applicable to record files	cyd$record_file
all declarations applicable to display files	cyd$display_file
all declarations applicable to text files	cyd$text_file

5.0 OPERATING SYSTEM DEPENDENT FEATURES
5.1.2 FILE NAMES

5.1.2 FILE NAMES

File names specified on the open command are interpreted as file references. That is, a file path, cycle designator and file position may be specified in the file name. File names must conform to the naming conventions for NOS/VE.

5.1.3 FILE POSITION

File position is specified both when a file is opened and when a file is closed. File position is a parameter on the cyp$open_file and cyp$close_file procedure calls. In addition, file position may be included in the file name passed to cyp$open_file.

If a file position is specified within the file name, the file_position parameter passed on the call to cyp$open_file should specify cyc$default.

When a file is opened, CYBILIO establishes the open position as follows:

5.0 OPERATING SYSTEM DEPENDENT FEATURES
5.1.3 FILE POSITION

FILE_POSITION	OPEN POSITION
cyc$beginning	beginning of information
cyc$end	end of information
cyc$asis	If the file was previously opened within the job and was closed with a file_disposition of cyc$retain_file, the open position is whatever was specified as the file position on the close. If the file was not previously opened within the job or was closed with a file_disposition other than cyc$retain_file, the open position is beginning of information.
cyc$default_open_position	If a file position was specified as part of the file name or, open position was specified on an amp$file call or set_file_attributes command, CYBILIO uses that position as the open position. If a file position was NOT previously specified, the open position is beginning of information.

On a call to cyp$close_file, the caller can specify file positioning to be done before the file is closed. A close file position of cyc$default is the equivalent of cyc$asis. It should be noted that the file position specified when closing a file is meaningful only if the close_disposition parameter specifies a value of cyc$retain_file and subsequent opens within the job specify file_position = cyc$asis.

5.1.4 FILE DISPOSITION

File disposition is specified whenever a file is closed. One of the following dispositions may be selected:

cyc$unload_file or cyc$return_file or cyc$detach_file	An explicit detach is performed when the file is closed. If the file has no other instances of open outstanding in the job, the file is detached.
cyc$retain_file:	If the file was explicitly attached prior to open, the file remains attached.
cyc$delete_file:	This disposition causes the file to be detached/deleted.
cyc$default_file_disposition:	If the file was implicitly attached by cyp$open_file and the file has no other instances of open outstanding in the job, the file is detached when the file is closed.

5.1.5 FILE ATTRIBUTES

Because CYBILIO is intended to provide a set of I/O interfaces that is standard across a number of different operating systems, no provision is made within CYBILIO to directly set or interrogate NOS/VE file attributes except as described in the following paragraphs.

CYBILIO follows a simple set of rules for file attributes. If the file has never been opened, the file is a new file and CYBILIO defines file attributes as specified in the following table unless the file specifications provided on the open contain a value from which the attribute may be set. If the file has been previously opened, CYBILIO considers the file to be an old file and does NOT modify or define any file attributes.

FILE ATTRIBUTE DEFINITIONS FOR NEW FILES

FILE_ ATTRIBUTE	BINARY FILES	RECORD FILES	TEXT FILES	DISPLAY FILES
file contents	cyc$unknown_ contents	cyc$unknown_ contents	cyc$legible	cyc$list
file structure	cyc$unknown_ structure	cyc$unknown_ structure	cyc$unknown_ structure	cyc$unknown_ structure
file processor	cyc$unknown_ processor	cyc$unknown_ processor	cyc$unknown_ processor	cyc$unknown_ processor
page format	system default	system default	cyc$contin- uous_form	cyc$burst- able_form
page length	system default	system default	system default	system default
page width	system default	system default	system default	system default

The page_length, page_width, page_format, file_contents, and file_processor file attributes for new files may be defined by the user on

the call to cyp$open_file. This is accomplished through the file_specifications parameter.

File attributes may be defined via system commands or CYBIL procedures prior to calling cyp$open_file. In this case, CYBILIO considers the file to be an old file and does not define or modify any of the permanent attributes.

5.1.6 FILE STRUCTURE CREATION/DETECTION

CYBILIO supports the subdivision of files into levels of logical structure. For NOS/VE, CYBILIO supports subdividing files into records and partitions. The End-Of-Information can only be implicitly created (i.e., the End-Of-Information follows the physically last item written on a file); but it can be explicitly detected.

On NOS/VE, the subdivision of files into partitions should only be done when there is some very good reason to do so. Partitions in a file usually do more harm than good.

NOS/VE does not support the subdivision of files into blocks. Calling the cyp$write_end_of_block procedure interface is essentially a no-op.

5.1.7 NOS/VE SPECIFIC PROCEDURES

The following procedures are available only in NOS/VE implementations of CYBILIO. These procedures provide access to facilities that are operating system dependent. Applications that are intended to be portable between operating systems should minimize use of these interfaces.

5.1.7.1 cyp$get file identifier

```
{* cyp$get_file_identifier *}

  PROCEDURE [XREF] cyp$get_file_identifier (file: cyt$file;
    VAR file_identifier: amt$file_identifier;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc cyt$file
*copyc cyt$file_control_block
*copyc ost$status
*copyc cye$exception_conditions
?? POP ??
```

This procedure returns the file identifier identifying the instance of open for the specified file. The FILE_IDENTIFIER returned by this procedure may be used on calls to access method procedures such as amp$fetch which are specific to an instance of open.

5.1.7.2 cyp$get_binary_file_pointer

```
{* cyp$get_binary_file_pointer *}

  PROCEDURE [XREF] cyp$get_binary_file_pointer (file: cyt$file;
    VAR binary_file_pointer: ↑amt$segment_pointer;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc cyt$file
*copyc amt$segment_pointer
*copyc ost$status
?? POP ??
```

CYBILIO reads and writes binary files using segment access. This procedure returns a pointer to the segment pointer that CYBILIO uses. CYBILIO gets the segment pointer as a sequence pointer (that is, the kind field of the segment pointer record is amc$sequence_pointer). The sequence pointer may be accessed by referencing the sequence_pointer field of the segment pointer. For example:

```
NEXT variable_pointer IN segment_pointer↑.sequence_pointer;
```

REMARKS:

This procedure provides a means for the user to directly manage the reading and/ or writing of binary files in those special cases where cyp$put_next_binary and cyp$get_next_binary are not sufficient (for example, pointer information is to be stored as part of the data to be written).

5.1.7.3 cyp$open_binary_file

```
{* cyp$open_binary_file *}

  PROCEDURE [XREF] cyp$open_binary_file (file_name: cyt$file_name;
        file_access: cyt$file_access;
        file_attachment: ↑fst$attachment_options;
        default_creation_attribute: ↑fst$file_cycle_attributes;
        mandated_creation_attribute: ↑fst$file_cycle_attributes;
        attribute_validation: ↑fst$file_cycle_attributes;
        attribute_override: ↑fst$file_cycle_attributes;
    VAR file_control: cyt$file;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc ost$status
*copyc cyt$file_name
*copyc cyt$file_access
*copyc cyt$file
*copyc cye$exception_conditions
*copyc fst$attachment_options
*copyc fst$file_cycle_attributes
?? POP ??
```

This procedure provides a means of using the flexibility of the fsp$open_file file interface with CYBILIO binary files. The file_attachment, default_creation_attribute, mandated_creation_attribute, attribute_validation, and attribute_override parameter values are passed directly to fsp$open_file. CYBILIO does NOT attempt to validate or in any way evaluate the values passed in these parameters. CYBILIO simply creates the environment that allows a user program to call the cybilio procedures and functions associated with binary files.

5.1.7.4 cyp$open record file

```
{* cyp$open_record_file *}

  PROCEDURE [XREF] cyp$open_record_file (file_name: cyt$file_name;
        file_access: cyt$file_access;
        file_attachment: ↑fst$attachment_options;
        default_creation_attribute: ↑fst$file_cycle_attributes;
        mandated_creation_attribute: ↑fst$file_cycle_attributes;
        attribute_validation: ↑fst$file_cycle_attributes;
        attribute_override: ↑fst$file_cycle_attributes;
    VAR file_control: cyt$file;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc ost$status
*copyc cyt$file_name
*copyc cyt$file_access
*copyc cyt$file
*copyc cye$exception_conditions
*copyc fst$attachment_options
*copyc fst$file_cycle_attributes
?? POP ??
```

This procedure provides a means of using the flexibility of the fsp$open_file file interface with CYBILIO record files. The file_attachment, default_creation_attribute, mandated_creation_attribute, attribute_validation, and attribute_override parameter values are passed directly to fsp$open_file. CYBILIO does NOT attempt to validate or in any way evaluate the values passed in these parameters. CYBILIO simply creates the environment that allows a user program to call the cybilio procedures and functions associated with record files.

5.1.7.5 cyp$open_text_file

```
{* cyp$open_text_file *}

  PROCEDURE [XREF] cyp$open_text_file (file_name: cyt$file_name;
        file_access: cyt$file_access;
        file_attachment: ↑fst$attachment_options;
        default_creation_attribute: ↑fst$file_cycle_attributes;
        mandated_creation_attribute: ↑fst$file_cycle_attributes;
        attribute_validation: ↑fst$file_cycle_attributes;
        attribute_override: ↑fst$file_cycle_attributes;
    VAR file_control: cyt$file;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc ost$status
*copyc cyt$file_name
*copyc cyt$file_access
*copyc cyt$file
*copyc cye$exception_conditions
*copyc fst$attachment_options
*copyc fst$file_cycle_attributes
?? POP ??
```

This procedure provides a means of using the flexibility of the fsp$open_file file interface with CYBILIO text files. The file_attachment, default_creation_attribute, mandated_creation_attribute, attribute_validation, and attribute_override parameter values are passed directly to fsp$open_file. CYBILIO does NOT attempt to validate or in any way evaluate the values passed in these parameters. CYBILIO simply creates the environment that allows a user program to call the cybilio procedures and functions associated with text files.

5.1.7.6 cyp$open_display_file

```
{* cyp$open_display_file *}

  PROCEDURE [XREF] cyp$open_display_file (file_name: cyt$file_name;
        file_access: cyt$file_access;
        file_attachment: ↑fst$attachment_options;
        default_creation_attribute: ↑fst$file_cycle_attributes;
        mandated_creation_attribute: ↑fst$file_cycle_attributes;
        attribute_validation: ↑fst$file_cycle_attributes;
        attribute_override: ↑fst$file_cycle_attributes;
    VAR file_control: cyt$file;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc ost$status
*copyc cyt$file_name
*copyc cyt$file_access
*copyc cyt$file
*copyc cye$exception_conditions
*copyc fst$attachment_options
*copyc fst$file_cycle_attributes
?? POP ??
```

This procedure provides a means of using the flexibility of the fsp$open_file file interface with CYBILIO display files. The file_attachment, default_creation_attribute, mandated_creation_attribute, attribute_validation, and attribute_override parameter values are passed directly to fsp$open_file. CYBILIO does NOT attempt to validate or in any way evaluate the values passed in these parameters. CYBILIO simply creates the environment that allows a user program to call the cybilio procedures and functions associated with display files.

5.2 NOS AND NOS/BE

5.2.1 DECK NAMES

DECLARATION	DECK NAME
CYBILIO types	ZCYTCIO
CYBILIO status conditions	ZCYECIO
cyp$open_file	ZCYPOF
cyp$close_file	ZCYPCF
cyp$position_file_at_beginning	ZCYPPFB
cyp$position_file_at_end	ZCYPPFE
cyp$length_of_file	ZCYPLOF
cyp$write_end_of_block	ZCYPWEB
cyp$write_end_of_partition	ZCYPWEP
cyp$current_file_position	ZCYPCFP
cyp$operating_system	ZCYPOS
cyp$put_next_record	ZCYPPNR
cyp$put_partial_record	ZCYPPPR
cyp$write_end_of_record	ZCYPWER
cyp$get_next_record	ZCYPGNR
cyp$get_partial_record	ZCYPGPR
cyp$position_record_file	ZCYPPRF

cyp$put_next_binary	ZCYPPNB
cyp$put_keyed_binary	ZCYPPKB
cyp$get_next_binary	ZCYPGNB
cyp$get_keyed_binary	ZCYPGKB
cyp$position_binary_at_key	ZCYPPBK
cyp$binary_file_key	ZCYPBFK
cyp$put_next_line	ZCYPPNL
cyp$put_partial_line	ZCYPPPL
cyp$write_end_of_line	ZCYPWEL
cyp$flush_line	ZCYPFL
cyp$tab_file	ZCYPTF
cyp$skip_lines	ZCYPSL
cyp$get_next_line	ZCYPGNL
cyp$get_partial_line	ZCYPGPL
cyp$file_connected_to_terminal	ZCYPFCT
cyp$current_column	ZCYPCC
cyp$page_width	ZCYPPW
cyp$start_new_display_page	ZCYPSNP
cyp$display_standard_title	ZCYPDST
cyp$position_display_page	ZCYPPDP
cyp$display_page_eject	ZCYPDPE
cyp$current_display_line	ZCYPCDL
cyp$current_page_number	ZCYPCPN

cyp$display_page_length	ZCYPDPL
all declarations applicable to binary files	ZCYDBF
all declarations applicable to record files	ZCYDRF
all declarations applicable to display files	ZCYDDF
all declarations applicable to text files	ZCYDTF

5.2.2 FILE NAMES

File names are limited to a maximum of 7 alphanumeric characters. The cyp$open_file procedure will convert any lower case letters in the file name to the corresponding upper case letters.

5.2.3 FILE POSITION

File position is a parameter on both the cyp$open_file and cyp$close_file procedure calls. The file position is of type cyt$open_close_position.

When a file is opened, CYBILIO establishes the open position as follows:

FILE_POSITION	OPEN POSITION
cyc$beginning	beginning of information
cyc$end	end of information
cyc$asis	If the file was previously opened within the job and was closed with a file_disposition of cyc$retain_file, the open position whatever was specified as the file position on the close. If the file was not previously opened within the job or was closed with a file_disposition other than cyc$retain_file, the open position is beginning of information.
cyc$default_open_position	beginning of information.

On a call to cyp$close_file, the caller can specify file positioning to be done before the file is closed. A close file position of cyc$default is the equivalent of cyc$asis. It should be noted that the file position specified when closing a file is meaningful only if the close_disposition parameter specifies a value of cyc$retain_file and subsequent opens within the job specify file_position = cyc$asis.

5.2.4 FILE DISPOSITION

File disposition is specified whenever a file is closed. One of the following dispositions may be selected:

cyc$return_file or cyc$detach_file	This disposition returns the file to the operating system and may release file space.
cyc$retain_file:	This disposition keeps the file attached to the job.
cyc$delete_file:	This disposition returns the file to the operating system and releases file space.
cyc$unload_file:	This disposition releases the file and may release file space. It differs from the cyc$return_file disposition in the handling of files on removable devices.
cyc$default_file_disposition:	This disposition keeps the file attached to the job.

5.2.5 FILE STRUCTURE CREATION/DETECTION

CYBILIO supports the subdivision of files into levels of logical structure. For NOS/VE and NOS/BE CYBILIO supports subdividing files into records, blocks, partitions. The End-Of-Information can only be implicitly created (i.e., the End-Of-Information follows the physically last item written on a file); but it can be explicitly detected.

When reading from a text file assigned to an interactive terminal, any cyc$end_of_block or cyc$end_of_partition positions returned by cyp$current_file_position after a read from a "terminal file" are discarded by CYBILIO (cyc$end_of_information is never possible from a terminal).

5.3 VSOS AND EOS

5.3.1 DECK NAMES

>>>>> to be supplied <<<<<<<<

5.3.2 FILE NAMES

File names are limited to a maximum of 8 alphanumeric characters.

5.3.3 FILE POSITION

>>>>>> to be supplied <<<<<<

5.3.4 FILE DISPOSITION

>>>>>> to be supplied <<<<<<

5.3.5 FILE STRUCTURE CREATION/DETECTION

CYBIL IO supports both the creation and detection of file structuring "marks".

5.4 AEGIS

5.4.1 DECK NAMES

>>>>>> to be supplied <<<<<<

5.4.2 FILE NAMES

file names can be up to 32 characters.
additional information >>>>>> to be supplied <<<<<<

5.4.3 FILE POSITION

>>>>>> to be supplied <<<<<<

5.4.4 FILE DISPOSITION

>>>>>> to be supplied <<<<<<

5.4.5 FILE STRUCTURE CREATION/DETECTION

CYBILIO supports both the creation and detection of file structuring "marks".

a1.0 BINARY FILE EXAMPLES

a1.0 BINARY FILE EXAMPLES

a1.1 COPY BINARY FILE

The following example illustrates the use of the binary file procedures to make a copy of a file (without knowing beforehand the structure or length of the file).

```
MODULE copy_example ALIAS 'zexmcop';

*copy cyd$binary_file


  PROGRAM copy ALIAS 'zexpcop';

    CONST
      in_name = 'OLD',
      out_name = 'NEW',
      buffer_length = 64;

    VAR
      in_file: cyt$file,
      out_file: cyt$file,
      buffer: ↑array [1 ..  * ] of cell,
      short_buffer: ↑array [1 ..  * ] of cell,
      sequence_ptr: ↑SEQ ( * ),
      transfer_length: integer,
      in_file_specifications: cyt$file_specifications,
      out_file_specifications: cyt$file_specifications,
      dummy_key: integer,
      copy_error: boolean,
      status: ost$status;

{*}
{  Set up file specifications }
{*}
    PUSH in_file_specifications: [1 .. 4];
    in_file_specifications↑ [1].selector := cyc$file_access;
    in_file_specifications↑ [1].file_access := cyc$read;
    in_file_specifications↑ [2].selector := cyc$file_kind;
```

a1.0 BINARY FILE EXAMPLES
a1.1 COPY BINARY FILE

```
    in_file_specifications↑ [2].file_kind := cyc$binary_file;
    in_file_specifications↑ [3].selector := cyc$file_existence;
    in_file_specifications↑ [3].file_existence := cyc$old_file;
    in_file_specifications↑ [4].selector := cyc$open_position;
    in_file_specifications↑ [4].open_position := cyc$beginning;

    PUSH out_file_specifications: [1 .. 4];
    out_file_specifications↑ [1].selector := cyc$file_access;
    out_file_specifications↑ [1].file_access := cyc$write;
    out_file_specifications↑ [2].selector := cyc$file_kind;
    out_file_specifications↑ [2].file_kind := cyc$binary_file;
    out_file_specifications↑ [3].selector := cyc$file_existence;
    out_file_specifications↑ [3].file_existence := cyc$new_or_old_file;
    out_file_specifications↑ [4].selector := cyc$open_position;
    out_file_specifications↑ [4].open_position := cyc$beginning;

    IF (cyp$operating_system () = cyc$nos) OR (cyp$operating_system () =
          cyc$nosbe) THEN
      PUSH buffer: [1 .. buffer_length];
    ELSE
      PUSH buffer: [1 .. buffer_length * 8];
    IFEND;

    copy_error := FALSE;

    cyp$open_file (in_name, in_file_specifications, in_file, status);
    IF NOT status.normal THEN
      RETURN; {----->
    IFEND;
    cyp$open_file (out_name, out_file_specifications, out_file, status);
    IF NOT status.normal THEN
      RETURN; {----->
    IFEND;

  /main_loop/
    REPEAT
      cyp$get_next_binary (in_file, #SEQ (buffer↑), dummy_key, transfer_length,
            status);
      IF status.normal THEN
        CASE cyp$current_file_position (in_file) OF
        = cyc$end_of_information =
          ; { all done copying }
        = cyc$end_of_partition =
          cyp$write_end_of_partition (out_file, status);
        = cyc$end_of_block =
          cyp$write_end_of_block (out_file, status);
        ELSE
```

a1.0 BINARY FILE EXAMPLES
a1.1 COPY BINARY FILE

```
          IF transfer_length = #SIZE (buffer↑) THEN
            cyp$put_next_binary (out_file, #SEQ (buffer↑), dummy_key, status);
          ELSE
{*}
{   only part of the buffer was filled }
{*}
            sequence_ptr := #SEQ (buffer↑);
            RESET sequence_ptr;
            NEXT short_buffer: [1 .. transfer_length] IN sequence_ptr;
            IF short_buffer = NIL THEN
              copy_error := TRUE;
            ELSE
              cyp$put_next_binary (out_file, #SEQ (short_buffer↑), dummy_key,
                    status);
            IFEND
          IFEND;
        CASEND;
        copy_error := (status.normal = FALSE);
      IFEND;
    UNTIL ((cyp$current_file_position (in_file) = cyc$end_of_information) OR
          (copy_error));
    IF NOT status.normal THEN
      RETURN; {----->
    IFEND;

    cyp$close_file (in_file, cyc$beginning, status);
    IF NOT status.normal THEN
      RETURN; {----->
    IFEND;
    cyp$close_file (out_file, cyc$beginning, status);

  PROCEND copy;

MODEND copy_example;
```

a1.0 BINARY FILE EXAMPLES
a1.1 COPY BINARY FILE

The next two examples illustrate the use of random access with binary files. The first example creates a "library" of "text modules" from a text file. The modules on the source (text) file are represented as blocks whose first line contains the module name (and nothing else). The second example extracts from the library one of the modules and copies it to a file whose name is that of the module.

a1.2 CREATE TEXT LIBRARY

```
MODULE create_text_library ALIAS 'zexmcre';

*copyc cyp$open_file
*copyc cyp$get_next_binary
*copyc cyp$get_next_line
*copyc cyp$write_end_of_block
*copyc cyp$write_end_of_partition
*copyc cyp$current_file_position
*copyc cyp$put_next_binary
*copyc cyp$close_file
*copyc cyp$position_file_at_beginning
*copyc cyp$put_keyed_binary


  TYPE
    directory_descriptor = record
      key: integer,
      length: integer,
    recend,

    directory_entry = record
      name: string (7),
      length: integer,
      key: integer,
    recend;

  CONST
    source_name = 'SOURCE',
    lib_name = 'LIBRARY',
    directory_name = 'SCRATCH';


  PROGRAM create ALIAS 'zexpcre';
```

```
VAR
  source_file_specs: [STATIC] array [1 .. 4] of cyt$file_specification
    := [[cyc$file_access, cyc$read], [cyc$file_kind, cyc$text_file],
    [cyc$file_existence, cyc$old_file], [cyc$open_position,
    cyc$beginning]],
  directory_file_specs: [STATIC] array [1 .. 3] of
    cyt$file_specification := [[cyc$file_kind, cyc$binary_file],
    [cyc$open_position, cyc$beginning], [cyc$close_file_disposition,
    cyc$return_file]],
  library_file_specs: [STATIC] array [1 .. 3] of cyt$file_specification
    := [[cyc$file_access, cyc$write], [cyc$file_kind, cyc$binary_file],
    [cyc$open_position, cyc$beginning]],
  source_file: cyt$file,
  library_file: cyt$file,
  directory_file: cyt$file,
  directory: directory_descriptor,
  current_module: directory_entry,
  line: string (256),
  line_length: integer,
  module_index: integer,
  first_key: integer,
  dummy_key: integer,
  cells_read: integer,
  read_status: ost$status,
  write_status: ost$status,
  status: ost$status;

PROCEDURE copy_a_module (VAR module_status: ost$status);

  VAR
    copy_status: ost$status,
    get_status: ost$status,
    put_status: ost$status;


  PROCEDURE copy_the_module_text (VAR local_status: ost$status);

    VAR
      get_status: ost$status,
      put_status: ost$status;

    local_status.normal := TRUE;

  /copy_text_loop/
    WHILE TRUE DO
      cyp$get_next_line (source_file, line, line_length, get_status);
      IF NOT get_status.normal THEN
```

```
          EXIT /copy_text_loop/;
        IFEND;

        CASE cyp$current_file_position (source_file) OF
        = cyc$end_of_information, cyc$end_of_partition, cyc$end_of_block
            =
        ELSE
          current_module.length := current_module.length + 1;
          cyp$put_next_binary (library_file, #SEQ (line_length),
                dummy_key, put_status);
          IF put_status.normal THEN
            cyp$put_next_binary (library_file, #SEQ (line (1,
                  line_length)), dummy_key, put_status);
          IFEND;
          IF NOT put_status.normal THEN
            EXIT /copy_text_loop/;
          IFEND;
        CASEND;
      WHILEND /copy_text_loop/;

      local_status.normal := get_status.normal AND put_status.normal;

    PROCEND copy_the_module_text;

  /copy_module_loop/
    WHILE TRUE DO
      cyp$get_next_line (source_file, line, line_length, get_status);
      IF NOT get_status.normal THEN
        EXIT /copy_module_loop/;
      IFEND;

      CASE cyp$current_file_position (source_file) OF
      = cyc$end_of_information, cyc$end_of_partition, cyc$end_of_block =
        EXIT /copy_module_loop/;
      ELSE
        directory.length := directory.length + 1;
        current_module.name := line (1, line_length);
        current_module.length := 1;
        cyp$put_next_binary (library_file, #SEQ (current_module.name),
              current_module.key, put_status);
        IF NOT put_status.normal THEN
          EXIT /copy_module_loop/;
        IFEND;

        copy_the_module_text (copy_status);
        IF NOT copy_status.normal THEN
          EXIT /copy_module_loop/;
```

```
        IFEND;

        cyp$put_next_binary (directory_file, #SEQ (current_module),
              dummy_key, put_status);
        IF NOT put_status.normal THEN
          EXIT /copy_module_loop/;
        IFEND;
      CASEND;
    WHILEND /copy_module_loop/;
    module_status.normal := copy_status.normal AND put_status.normal AND
          get_status.normal;

  PROCEND copy_a_module;

  PROCEDURE copy_directory_to_library (VAR local_status: ost$status);

    VAR
      module_index: integer,
      read_status: ost$status,
      write_status: ost$status;

    cyp$get_next_binary (directory_file, #SEQ (current_module),
          dummy_key, cells_read, read_status);
    IF read_status.normal THEN
      cyp$put_next_binary (library_file, #SEQ (current_module),
            directory.key, write_status);
      IF write_status.normal THEN

      /read_loop/
        FOR module_index := 2 TO directory.length DO
          cyp$get_next_binary (directory_file, #SEQ (current_module),
                dummy_key, cells_read, read_status);
          IF NOT read_status.normal THEN
            EXIT /read_loop/;
          IFEND;
          cyp$put_next_binary (library_file, #SEQ (current_module),
                dummy_key, write_status);
          IF NOT write_status.normal THEN
            EXIT /read_loop/;
          IFEND;
        FOREND /read_loop/;
        IF read_status.normal AND write_status.normal THEN
          cyp$put_keyed_binary (library_file, #SEQ (directory),
                first_key, write_status);
        IFEND;
      IFEND;
    IFEND;
```

```
      local_status.normal := read_status.normal AND write_status.normal;

    PROCEND copy_directory_to_library;


    cyp$open_file (source_name, ↑source_file_specs, source_file, status);
    IF status.normal THEN
      cyp$open_file (directory_name, ↑directory_file_specs, directory_file,
            status);
      IF status.normal THEN
        cyp$open_file (lib_name, ↑library_file_specs, library_file,
              status);
      IFEND;
    IFEND;

    IF status.normal THEN

    /main_program/
      BEGIN
{*}
{  reserve space for a directory
{*}
        directory.length := 0;
        cyp$put_next_binary (library_file, #SEQ (directory), first_key,
              write_status);
        IF write_status.normal THEN
          copy_a_module (read_status);
          cyp$close_file (source_file, cyc$end, status);
          IF ((read_status.normal) AND (directory.length > 0)) THEN
            cyp$position_file_at_beginning (directory_file, status);
            IF NOT status.normal THEN
              EXIT /main_program/;
            IFEND;

            copy_directory_to_library (status);

          IFEND;
        IFEND;

      END /main_program/;

    IFEND;
    cyp$close_file (directory_file, cyc$asis, status);
    cyp$close_file (library_file, cyc$beginning, status);

  PROCEND create;
```

a1.0 BINARY FILE EXAMPLES
a1.2 CREATE TEXT LIBRARY

```
MODEND create_text_library;
```

a1.3 EXTRACT FROM TEXT LIBRARY

```
MODULE extract_from_text_library ALIAS 'zexmef1';

*copyc cyp$open_file
*copyc cyp$close_file
*copyc cyp$get_next_binary
*copyc cyp$get_keyed_binary
*copyc cyp$position_binary_at_key
*copyc cyp$put_keyed_binary
*copyc cyp$put_next_line
*copyc cyp$current_file_position

  TYPE
    directory_descriptor = record
      key: integer,
      length: integer,
    recend,

    directory_entry = record
      name: string (7),
      length: integer,
      key: integer,
    recend;

  CONST
    lib_name = 'LIBRARY';

  CONST
    name_of_module = 'TEXTMOD';


  PROGRAM extract ALIAS 'zexpef1';

    VAR
      library_file_specs: [STATIC] array [1 .. 4] of cyt$file_specification
        := [[cyc$file_kind, cyc$binary_file], [cyc$open_position,
        cyc$beginning], [cyc$file_existence, cyc$old_file],
        [cyc$file_access, cyc$read]],
      output_file_specs: [STATIC] array [1 .. 3] of cyt$file_specification
        := [[cyc$file_access, cyc$write], [cyc$file_kind, cyc$text_file],
        [cyc$open_position, cyc$beginning]],
      library_file: cyt$file,
      out_file: cyt$file,
      directory: directory_descriptor,
```

```
    current_module: directory_entry,
    line: string (256),
    line_length: integer,
    module_found: boolean,
    dummy_key: integer,
    cells_read: integer,
    status: ost$status;

  PROCEDURE search_for_module (library_directory: directory_descriptor;
    VAR module_is_in_directory: boolean;
    VAR search_status: ost$status);

    VAR
      module_index: integer;


    module_is_in_directory := FALSE;
    search_status.normal := TRUE;

    cyp$position_binary_at_key (library_file, library_directory.key,
          search_status);
    IF NOT search_status.normal THEN
      RETURN; {----->
    IFEND;

  /search_directory/
    FOR module_index := 1 TO library_directory.length DO
      cyp$get_next_binary (library_file, #SEQ (current_module),
            dummy_key, cells_read, search_status);
      IF NOT search_status.normal THEN
        RETURN; {----->
      IFEND;
      IF current_module.name = name_of_module THEN
        module_is_in_directory := TRUE;
        EXIT /search_directory/;
      IFEND;
    FOREND /search_directory/;

  PROCEND search_for_module;

  PROCEDURE copy_the_module_text (VAR copy_status: ost$status);

  /module_loop/
    WHILE current_module.length > 1 DO
      cyp$get_next_binary (library_file, #SEQ (line_length), dummy_key,
            cells_read, copy_status);
      IF NOT copy_status.normal THEN
```

```
        EXIT /module_loop/;
      IFEND;
      cyp$get_next_binary (library_file, #SEQ (line (1, line_length)),
            dummy_key, cells_read, copy_status);
      IF NOT copy_status.normal THEN
        EXIT /module_loop/;
      IFEND;
      cyp$put_next_line (out_file, line (1, line_length), copy_status);
      IF NOT copy_status.normal THEN
        EXIT /module_loop/;
      IFEND;
      current_module.length := current_module.length - 1;
    WHILEND /module_loop/;
  PROCEND copy_the_module_text;

  cyp$open_file (lib_name, ↑library_file_specs, library_file, status);
  IF NOT status.normal THEN
    RETURN; {----->
  IFEND;
  cyp$get_next_binary (library_file, #SEQ (directory), dummy_key,
        cells_read, status);
  IF NOT status.normal THEN
    RETURN; {----->
  IFEND;
  IF directory.length = 0 THEN
    RETURN; {----->
  IFEND;
  search_for_module (directory, module_found, status);

  IF status.normal AND module_found THEN
    cyp$open_file (name_of_module, ↑output_file_specs, out_file, status);
    IF NOT status.normal THEN
      RETURN; {----->
    IFEND;
    cyp$get_keyed_binary (library_file, #SEQ (current_module.name),
          current_module.key, cells_read, status);
    IF NOT status.normal THEN
      RETURN; {----->
    IFEND;
    cyp$put_next_line (out_file, current_module.name, status);
    IF NOT status.normal THEN
      RETURN; {----->
    IFEND;

    copy_the_module_text (status);
  IFEND;
```

```
    cyp$close_file (library_file, cyc$beginning, status);
    IF NOT status.normal THEN
      RETURN; {----->
    IFEND;

    cyp$close_file (out_file, cyc$beginning, status);
    IF NOT status.normal THEN
      RETURN; {----->
    IFEND;

  PROCEND extract;

MODEND extract_from_text_library;
```

b1.0 RECORD FILE EXAMPLES

b1.1 EXAMPLE - EXTRACT INFORMATION FROM RECORDS

The following example illustrates the use of record file procedures. The input file is assumed to contain several kinds of logical records. An id record identifies the following record as an employee record or a vendor record. A vendor record is followed by one or more product records. The program produces a list of vendor names and the names of the products supplied by each vendor.

```
MODULE list_vendor_and_products ALIAS 'zexmvap';

*copyc cyp$open_file
*copyc cyp$close_file
*copyc cyp$get_next_record
*copyc cyp$put_next_line
*copyc cyp$put_partial_line
*copyc cyp$position_record_file
*copyc cyp$tab_file
*copyc cyp$current_file_position


 PROGRAM list_vendor_and_products ALIAS 'zexpvap';

   CONST
     in_name = 'EMPDB',
     out_name = 'EMPLIST';

   TYPE
     full_name = record
       first: string (10),
       initial: char,
       last: string (15),
     recend,

     employee_entry = record
       number: 0 .. 999999,
       name: full_name,
       department_number: 0 .. 9999,
```

b1.0 RECORD FILE EXAMPLES
b1.1 EXAMPLE - EXTRACT INFORMATION FROM RECORDS

```
      department_name: string (20),
    recend,

    vendor_entry = record
      number: 0 .. 99999999,
      name: string (30),
      street_address: string (30),
      city_state: string (30),
      zip_code: 0 .. 99999,
      number_of_products: integer,
    recend,

    product_entry = record
      name: string (20),
      product_number: string (10),
    recend,

    entry_id = (employee_id, vendor_id);

  VAR
    in_file: cyt$file,
    out_file: cyt$file,
    in_file_specs: cyt$file_specifications,
    out_file_specs: cyt$file_specifications,
    cells_read: integer,
    vendor: vendor_entry,
    product: product_entry,
    record_id: entry_id,
    i: integer,
    status: ost$status;

  PUSH in_file_specs: [1 .. 4];
  in_file_specs↑ [1].selector := cyc$file_kind;
  in_file_specs↑ [1].file_kind := cyc$record_file;
  in_file_specs↑ [2].selector := cyc$file_access;
  in_file_specs↑ [2].file_access := cyc$read;
  in_file_specs↑ [3].selector := cyc$file_existence;
  in_file_specs↑ [3].file_existence := cyc$old_file;
  in_file_specs↑ [4].selector := cyc$open_position;
  in_file_specs↑ [4].open_position := cyc$beginning;

  PUSH out_file_specs: [1 .. 3];
  out_file_specs↑ [1].selector := cyc$file_kind;
  out_file_specs↑ [1].file_kind := cyc$text_file;
  out_file_specs↑ [2].selector := cyc$file_access;
  out_file_specs↑ [2].file_access := cyc$write;
  out_file_specs↑ [3].selector := cyc$open_position;
```

b1.0 RECORD FILE EXAMPLES
b1.1 EXAMPLE - EXTRACT INFORMATION FROM RECORDS

```
    out_file_specs↑ [3].open_position := cyc$beginning;


    cyp$open_file (in_name, in_file_specs, in_file, status);
    IF NOT status.normal THEN
      RETURN; {----->
    IFEND;
    cyp$open_file (out_name, out_file_specs, out_file, status);
    IF NOT status.normal THEN
      RETURN; {----->
    IFEND;

  /main_loop/
    WHILE status.normal DO
      cyp$get_next_record (in_file, #SEQ (record_id), cells_read, status);
      IF NOT status.normal THEN
        EXIT /main_loop/;
      IFEND;
      CASE cyp$current_file_position (in_file) OF
      = cyc$end_of_partition, cyc$end_of_block =
        CYCLE /main_loop/;
      = cyc$end_of_information =
        EXIT /main_loop/;
      = cyc$middle_of_record =
        cyp$put_next_line (out_file, 'ERROR reading input file', status);
        EXIT /main_loop/;
      = cyc$end_of_record =
        CASE record_id OF
        = employee_id =
          cyp$position_record_file (in_file, cyc$forward, 1, cyc$record,
                status);
          IF NOT status.normal THEN
            EXIT /main_loop/;
          IFEND;
        = vendor_id =
          cyp$get_next_record (in_file, #SEQ (vendor), cells_read, status);
          IF NOT status.normal THEN
            EXIT /main_loop/;
          IFEND;
          IF cyp$current_file_position (in_file) = cyc$end_of_record THEN
            cyp$put_next_line (out_file, vendor.name, status);
            IF NOT status.normal THEN
              EXIT /main_loop/;
            IFEND;

            FOR i := 1 TO vendor.number_of_products DO
              cyp$get_next_record (in_file, #SEQ (product), cells_read,
```

b1.0 RECORD FILE EXAMPLES
b1.1 EXAMPLE - EXTRACT INFORMATION FROM RECORDS

```
                status);
            IF cyp$current_file_position (in_file) <> cyc$end_of_record
                THEN
              cyp$put_next_line (out_file, 'ERROR reading input file',
                  status);
              EXIT /main_loop/;
            ELSEIF (NOT status.normal) OR (cells_read <> #SIZE (product))
                THEN
              EXIT /main_loop/;
            IFEND;
            cyp$tab_file (out_file, 10, status);
            IF NOT status.normal THEN
              EXIT /main_loop/;
            IFEND;
            cyp$put_partial_line (out_file, TRUE, product.name, status);
            IF NOT status.normal THEN
              EXIT /main_loop/;
            IFEND;
          FOREND;
        ELSE
          cyp$put_next_line (out_file, 'ERROR reading input file',
              status);
          EXIT /main_loop/;
        IFEND;
      CASEND;
    CASEND;
  WHILEND /main_loop/;

  cyp$close_file (in_file, cyc$beginning, status);
  IF NOT status.normal THEN
    RETURN; {----->
  IFEND;
  cyp$close_file (out_file, cyc$beginning, status);
  IF NOT status.normal THEN
    RETURN; {----->
  IFEND;

PROCEND list_vendor_and_products;

MODEND list_vendor_and_products;
```

c1.0 TEXT FILE EXAMPLES

c1.1 EXAMPLE - COPY COLUMN RANGE OF TEXT FILE

The following example illustrates the use of text file procedures to copy one text file to another. Only data between selected columns on the old file is written to the new file, and within those columns, trailing space characters are deleted.

```
MODULE truncate ALIAS 'zexmtru';

*copyc cyp$open_file
*copyc cyp$close_file
*copyc cyp$get_next_line
*copyc cyp$put_partial_line
*copyc cyp$write_end_of_line
*copyc cyp$write_end_of_block
*copyc cyp$write_end_of_partition
*copyc cyp$current_file_position


  PROGRAM truncate ALIAS 'zexptru';

    CONST
      in_name = 'OLD',
      out_name = 'NEW',
      leftmost_column_# = 11,
      rightmost_column_# = 72;

    VAR
      in_file: cyt$file,
      out_file: cyt$file,
      in_file_specs: cyt$file_specifications,
      out_file_specs: cyt$file_specifications,
      line_ptr: ↑string ( * <= cyc$max_page_width),
      line_length: integer,
      status: ost$status;

    PUSH in_file_specs: [1 .. 4];
    in_file_specs↑ [1].selector := cyc$file_kind;
```

c1.0 TEXT FILE EXAMPLES
c1.1 EXAMPLE - COPY COLUMN RANGE OF TEXT FILE

```
    in_file_specs↑ [1].file_kind := cyc$text_file;
    in_file_specs↑ [2].selector := cyc$file_access;
    in_file_specs↑ [2].file_access := cyc$read;
    in_file_specs↑ [3].selector := cyc$file_existence;
    in_file_specs↑ [3].file_existence := cyc$old_file;
    in_file_specs↑ [4].selector := cyc$open_position;
    in_file_specs↑ [4].open_position := cyc$beginning;

    PUSH out_file_specs: [1 .. 3];
    out_file_specs↑ [1].selector := cyc$file_kind;
    out_file_specs↑ [1].file_kind := cyc$text_file;
    out_file_specs↑ [2].selector := cyc$file_access;
    out_file_specs↑ [2].file_access := cyc$write;
    out_file_specs↑ [3].selector := cyc$open_position;
    out_file_specs↑ [3].open_position := cyc$beginning;

    ALLOCATE line_ptr: [rightmost_column_#];

    cyp$open_file (in_name, in_file_specs, in_file, status);
    IF NOT status.normal THEN
      RETURN; {----->
    IFEND;
    cyp$open_file (out_name, out_file_specs, out_file, status);
    IF NOT status.normal THEN
      RETURN; {----->
    IFEND;

  /main_loop/
    WHILE status.normal DO
      cyp$get_next_line (in_file, line_ptr↑, line_length, status);
      IF NOT status.normal THEN
        EXIT /main_loop/;
      IFEND;
      CASE cyp$current_file_position (in_file) OF
      = cyc$end_of_partition =
        cyp$write_end_of_partition (out_file, status);
        IF NOT status.normal THEN
          EXIT /main_loop/;
        IFEND;
      = cyc$end_of_block =
        cyp$write_end_of_block (out_file, status);
        IF NOT status.normal THEN
          EXIT /main_loop/;
        IFEND;
      = cyc$end_of_information =
        EXIT /main_loop/;
      ELSE
```

c1.0 TEXT FILE EXAMPLES
c1.1 EXAMPLE - COPY COLUMN RANGE OF TEXT FILE

```
        WHILE (line_length > leftmost_column_#) AND (line_ptr↑
              (line_length) = ' ') DO
          line_length := line_length - 1;
        WHILEND;
        line_length := line_length - leftmost_column_# + 1;
        IF line_length > 0 THEN
          cyp$put_next_line (out_file, line_ptr↑ (leftmost_column_#,
                line_length), status);
        ELSE
          cyp$write_end_of_line (out_file, status);
        IFEND;
        IF NOT status.normal THEN
          EXIT /main_loop/;
        IFEND;
      CASEND;
    WHILEND /main_loop/;

    cyp$close_file (in_file, cyc$beginning, status);
    IF NOT status.normal THEN
      RETURN; {----->
    IFEND;
    cyp$close_file (out_file, cyc$beginning, status);
    IF NOT status.normal THEN
      RETURN; {----->
    IFEND;

    FREE line_ptr;

  PROCEND truncate;

MODEND truncate;
```

d1.0 DISPLAY FILE EXAMPLES

d1.1 EXAMPLE - DISPLAY A TEXT FILE

The following example illustrates the use of display file procedures (and text file procedures). Note particularly the page overflow processing procedure.

```
MODULE zexmlis ALIAS 'zexmlis';

*copyc cyp$open_file
*copyc cyp$close_file
*copyc cyp$current_file_position
*copyc cyp$current_display_line
*copyc cyp$get_partial_line
*copyc cyp$display_page_length
*copyc cyp$tab_file
*copyc cyp$skip_lines
*copyc cyp$position_display_page
*copyc cyp$put_partial_line
*copyc cyp$write_end_of_line
*copyc cyp$display_page_eject
*copyc cyp$current_page_number


  CONST
    in_name = 'TEXFILE';

  VAR
    file_numb: integer := 1,
    record_numb: integer := 1;


  PROGRAM list ALIAS 'zexplis';

    CONST
      out_page_width = 80,
      out_page_length = 50,
      footing_line_number = out_page_length - 2,
      out_name = 'OUTPUT';
```

d1.0 DISPLAY FILE EXAMPLES
d1.1 EXAMPLE - DISPLAY A TEXT FILE

```
VAR
  in_file_specs: cyt$file_specifications,
  out_file_specs: cyt$file_specifications,
  in_file: cyt$file,
  out_file: cyt$file,
  line: string (80),
  line_length: integer,
  eol: boolean,
  status: ost$status;

PROCEDURE my_new_page_proc (print_file: cyt$file;
      next_page_number: integer;
  VAR status: ost$status);

  VAR
    str_holder: string (10),
    str_length: integer;

  cyp$display_page_eject (print_file, status);
  IF NOT status.normal THEN
    RETURN; {----->
  IFEND;
  cyp$put_partial_line (print_file, FALSE, 'LISTING OF ', status);
  IF NOT status.normal THEN
    RETURN; {----->
  IFEND;
  cyp$put_partial_line (print_file, FALSE, in_name, status);
  IF NOT status.normal THEN
    RETURN; {----->
  IFEND;
  cyp$tab_file (print_file, 50, status);
  IF NOT status.normal THEN
    RETURN; {----->
  IFEND;
  cyp$put_partial_line (print_file, FALSE, 'FILE ', status);
  IF NOT status.normal THEN
    RETURN; {----->
  IFEND;
  STRINGREP (str_holder, str_length, file_numb);
  cyp$put_partial_line (print_file, FALSE, str_holder (1, str_length),
      status);
  IF NOT status.normal THEN
    RETURN; {----->
  IFEND;
  cyp$put_partial_line (print_file, FALSE, ', RECORD ', status);
  IF NOT status.normal THEN
```

d1.0 DISPLAY FILE EXAMPLES
d1.1 EXAMPLE - DISPLAY A TEXT FILE

```
      RETURN; {----->
    IFEND;
    STRINGREP (str_holder, str_length, record_numb);
    cyp$put_partial_line (print_file, TRUE, str_holder (1, str_length),
        status);
    IF NOT status.normal THEN
      RETURN; {----->
    IFEND;
    cyp$skip_lines (print_file, 2, status);
    IF NOT status.normal THEN
      RETURN; {----->
    IFEND;

  PROCEND my_new_page_proc;


  PROCEDURE print_page_footer (print_file: cyt$file;
    VAR status: ost$status);

    VAR
      str_holder: string (3),
      str_length: integer,
      page_number: integer;

    cyp$put_partial_line (print_file, TRUE, ' ', status);
    IF NOT status.normal THEN
      RETURN; {----->
    IFEND;
    page_number := cyp$current_page_number (print_file);
    cyp$tab_file (print_file, 70, status);
    IF NOT status.normal THEN
      RETURN; {----->
    IFEND;
    cyp$put_partial_line (print_file, FALSE, 'PAGE ', status);
    IF NOT status.normal THEN
      RETURN; {----->
    IFEND;
    STRINGREP (str_holder, str_length, page_number);
    cyp$put_partial_line (print_file, TRUE, str_holder (1, str_length),
        status);
    IF NOT status.normal THEN
      RETURN; {----->
    IFEND;

  PROCEND print_page_footer;
```

d1.0 DISPLAY FILE EXAMPLES
d1.1 EXAMPLE - DISPLAY A TEXT FILE

```
    PUSH in_file_specs: [1 .. 4];
    in_file_specs↑ [1].selector := cyc$file_kind;
    in_file_specs↑ [1].file_kind := cyc$text_file;
    in_file_specs↑ [2].selector := cyc$file_access;
    in_file_specs↑ [2].file_access := cyc$read;
    in_file_specs↑ [3].selector := cyc$file_existence;
    in_file_specs↑ [3].file_existence := cyc$old_file;
    in_file_specs↑ [4].selector := cyc$open_position;
    in_file_specs↑ [4].open_position := cyc$beginning;

    PUSH out_file_specs: [1 .. 6];
    out_file_specs↑ [1].selector := cyc$file_kind;
    out_file_specs↑ [1].file_kind := cyc$display_file;
    out_file_specs↑ [2].selector := cyc$file_access;
    out_file_specs↑ [2].file_access := cyc$write;
    out_file_specs↑ [3].selector := cyc$file_existence;
    out_file_specs↑ [3].file_existence := cyc$new_or_old_file;
    out_file_specs↑ [4].selector := cyc$page_width;
    out_file_specs↑ [4].page_width := out_page_width;
    out_file_specs↑ [5].selector := cyc$page_length;
    out_file_specs↑ [5].page_length := out_page_length;
    out_file_specs↑ [6].selector := cyc$new_page_procedure;
    out_file_specs↑ [6].new_page_procedure.kind :=
          cyc$user_specified_procedure;
    out_file_specs↑ [6].new_page_procedure.user_procedure :=
          ↑my_new_page_proc;

    cyp$open_file (in_name, in_file_specs, in_file, status);
    IF NOT status.normal THEN
      RETURN; {----->
    IFEND;
    cyp$open_file (out_name, out_file_specs, out_file, status);
    IF NOT status.normal THEN
      RETURN; {----->
    IFEND;

  /main_loop/
    WHILE TRUE DO
      cyp$get_partial_line (in_file, line, line_length, eol, status);
      IF NOT status.normal THEN
        EXIT /main_loop/;
      IFEND;
      CASE cyp$current_file_position (in_file) OF
      = cyc$end_of_information =

        cyp$position_display_page (out_file, footing_line_number, status);
        IF NOT status.normal THEN
```

d1.0 DISPLAY FILE EXAMPLES
d1.1 EXAMPLE - DISPLAY A TEXT FILE

```
      EXIT /main_loop/;
    IFEND;
    print_page_footer (out_file, status);
    IF NOT status.normal THEN
      EXIT /main_loop/;
    IFEND;
    EXIT /main_loop/;

  = cyc$end_of_partition =
    file_numb := file_numb + 1;
    record_numb := 1;
    cyp$position_display_page (out_file, footing_line_number, status);
    IF NOT status.normal THEN
      EXIT /main_loop/;
    IFEND;
    print_page_footer (out_file, status);
    IF NOT status.normal THEN
      EXIT /main_loop/;
    IFEND;

  = cyc$end_of_block =
    record_numb := record_numb + 1;
    cyp$position_display_page (out_file, footing_line_number, status);
    IF NOT status.normal THEN
      EXIT /main_loop/;
    IFEND;
    print_page_footer (out_file, status);
    IF NOT status.normal THEN
      EXIT /main_loop/;
    IFEND;

  ELSE
    IF cyp$current_display_line (out_file) = footing_line_number THEN
      print_page_footer (out_file, status);
      IF NOT status.normal THEN
        EXIT /main_loop/;
      IFEND;
    IFEND;
    IF line_length > 0 THEN
      cyp$put_partial_line (out_file, eol, line (1, line_length),
            status);
    ELSE
      cyp$write_end_of_line (out_file, status);
    IFEND;
    IF NOT status.normal THEN
      EXIT /main_loop/;
    IFEND;
```

d1.0 DISPLAY FILE EXAMPLES
d1.1 EXAMPLE - DISPLAY A TEXT FILE

```
      CASEND;
    WHILEND /main_loop/;

    cyp$close_file (in_file, cyc$beginning, status);
    IF NOT status.normal THEN
      RETURN; {----->
    IFEND;
    cyp$close_file (out_file, cyc$asis, status);
    IF NOT status.normal THEN
      RETURN; {----->
    IFEND;

  PROCEND list;

MODEND zexmlis;
```