EXTERNAL REFERENCE SPECIFICATION

for

CYBILIO

Submitted: S. L. Eagles

Approved:

W. R. Bayer

P. W. Haynes

J. R. Ruble

REVISION DEFINITION SHEET

REV	DATE	DESCRIPTION
A	06/23/78	Original Release.
B	11/15/78	Complete reprint. Incorporates changes for CYBIL Version 2 (e.g., uses pointer to procedure for page overflow procedure for print files); eliminates the "directory" at the beginning of direct files; and adds two "status interrogation" procedures for direct files. Also, certain clarifications have been added, and typographical errors corrected.
C	02/18/80	Modifications for SES Release 13.
D	07/18/80	Document format changes. Obsoletes all previous revisions.

1.0 INTRODUCTION

The CYBILIO package (CYBILIO) is a collection of procedures and data types which provide an Input/Output system that interfaces a CYBIL program to the NOS I/O system.

Note that this package is intended solely for use as a building block for SES tools and other programs that run on a CYBER 170 under NOS. No compatibility with NOSVE I/O interfaces is implied or intended.

1.1 APPLICABLE DOCUMENTS

ARH2298	Language Specification for CDC CYBER IMPLEMENTATION LANGUAGE
ARH1833	SES User's Handbook
60435400	NOS Version 1 Reference Manual (Volume 1)
60445300	NOS Version 1 Reference Manual (Volume 2)
60450100	NOS Version 1 Modify Reference Manual

1.2 FILE TYPES

CYBILIO deals with a (small) number of distinct types of files. The properties of the various file types are described in the subsections which follow.

1.2.1 BINARY FILES

Binary files have only sequential access. Data appears on such files in the order in which it was written, and can only be read in that same order. These files may be positioned at the beginning or end of information. Note that positioning at the beginning and then writing a binary file implies that all data which was previously on the file is lost.

Binary files may be structured using the NOS record/file marks, and detection of the structure is possible.

1.2.2 DIRECT FILES

Direct files are like binary files except that data may be transferred to/from them at "random addresses" known as keys. Note that writing (from the beginning of) a direct file does not necessarily imply that existing data (which follows the data being written) will be lost (c.f., binary files).

In addition to the positioning facilities provided for binary files, direct files may be positioned via a key to any location.

1.2.3 LEGIBLE FILES

Legible files are sequentially accessed and are assumed to contain character data in NOS 6/12 representation. Legible I/O procedures provide for the conversion between the external (on the file) data format and the internal format (CYBIL strings). The basic entity on a legible file is a line which can be transferred to/from the file in whole or in part. In addition, there is a facility to tab to a specified column in an output line.

The same structuring and positioning facilities provided for binary files are also available for legible files.

1.2.4 PRINT FILES

Print files are legible files which have additional facilities for (vertical) format control. It is possible to limit the number of 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 page. Several procedures are provided to change and interrogate certain items of control information for print files.

The user may associate with each print file, a procedure to be called when a "page overflow condition" occurs for that file. Such a procedure can perform page heading (titling) and footing operations.

1.3 CYBILIO DATA TYPES

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

```
TYPE file = ^CELL;
```

This type is used when calling any of the CYBILIO procedures. A variable of this type is defined when passed to one of the file open procedures, and remains defined until the corresponding close procedure is called.

```
TYPE file_status = (new#, old#);
```

This type is used when opening a file to designate whether the file already exists or needs to be "created".

```
TYPE file_mode = (input#, output#, concurrent#);
```

This type is used when opening a file to designate the "direction" of data transfers.

```
TYPE file_position = (first#, asis#, last#);
```

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, or at its end).

```
CONST return# = last#;
TYPE file_disposition = first# .. return#;
{ i.e., (first#, asis#, return#) }
```

This type is used when closing a file to designate at what "position" (or with which "disposition") the file should be left (at its beginning, where ever it happens to be, or "return" it to NOS).

TYPE file_encoding = (ascii64#, ascii612#, ascii#);

This type is used to define the (external) character set for a legible or print file. The default, when the file is opened, is ascii612# which designates the NOS 6/12 character set. The user can select the 6-bit display code character set designated by ascii64#, or the "8 out of 12 bit" ASCII character set designated by ascii#.

TYPE file_mark = (data#, eor#, eof#, eoi#);

This type is used to designate the NOS file structure marks. A value of this type can be obtained for a file subsequent to a performing an input (read, get) request on the file. Thus, data# means "no mark encountered"; eor# means "a (logical) End Of Record was encountered"; eof# means "a (logical) End Of File was encountered"; and eoi# means "the End of Information was encountered".

1.4 <u>USING CYBILIO</u>

1.4.1 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 program library (PL). The name of this program library is CYBCCMN, which is accessible by including the CYBCCMN parameter in the SES.GENCOMP call or can be made local by SES.GETCCMN. The CYBILIO type declarations are on common deck PXIOTYP and each procedure declaration is on its own common deck (see the section on naming conventions and the individual procedure descriptions to determine the common deck names).

1.4.2 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-CC run-time library, which is accessible by including the CYBCLIB parameter in the SES.LINK170 call.

1.4.3 NAMING CONVENTIONS

The identifiers for all CYBILIO procedures adhere to the following naming convention:

- all BINARY file procedure identifiers begin with <u>bi#</u>
- all DIRECT file procedure identifiers begin with <u>di#</u>
- all LEGIBLE file procedures identifiers begin with <u>lg#</u>
- all PRINT file procedure identifiers begin with <u>pr#</u>
- identifiers for procedures which are applicable to all file types begin with <u>f#</u>

The names of the common decks which contain the CYBILIO procedure declarations are derived by taking (up to) the first seven characters of the procedure name and changing the <u>#</u> character in the procedure name to a <u>Z</u> for the common deck name.

Note: that all common decks required for a particular file type can be included with MODIFY's *CALLALL directive. For example, a program that uses legible and print files could bring in all the relevant declarations as follows:

```
*CALL pxiotyp
*CALLALL lgz
*CALLALL prz
*CALLALL fz
```

1.4.4 FILE VARIABLE USAGE

CYBILIO considers a variable of type file to be undefined until one of the open procedures has been called; and to become undefined once one of the close procedures has been called. The consequences of using an undefined file variable to call any CYBILIO procedure (except one of the open procedures) is unpredictable.

1.4.5 FILE NAMES

File names (which are passed as adaptable CYBIL strings to the open procedures) must be from one to seven alphanumeric characters (i.e., letters or digits). The open procedures will convert any lower case letters in a file name to the corresponding upper case letters.

1.4.6 NOS FILE STRUCTURE CREATION / DETECTION

CYBILIO supports both the creation and detection of NOS file structuring "marks". There are two such marks : End Of (logical) Record; and End Of (logical) File. 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.

Note that detection of a file structure mark can only be meaningfully attempted after an input request on the file.

When performing input operations on binary and direct files, it is possible to have an incomplete transfer. This can result from reading a file not created by CYBILIO, or not reading a file in a manner which mimics the way in which the file was written. In order to handle these incomplete transfers, a procedure is provided to return the <u>last transfer length</u> for a file. The value returned by this procedure is only meaningfull if the immediately preceding operation on the file was one of : bi#get, bi#put, di#get, di#getdir, lg#get, lg#getpart, lg#put, lg#putpart, lg#tab, or lg#weol.

1.4.7 CIO BUFFER SIZE CONTROL

The size of the CIO circular buffer can be selected for files used with CYBILIO by setting the INTEGER variable <u>px#iobs</u> to the desired size. The value of this variable is used by the file open procedures in order to create a CIO circular buffer with the designated size. The declaration for this variable is contained on common deck PXZIOBS.

1.4.8 LONG STRING OF BLANK (SPACE) CHARACTERS

CYBILIO needs a long (256) string of blank characters in order to efficiently perform the lg#tab and pr#tab operations. This string is made available to the user in the variable <u>px#blnk</u>. The declaration for this variable is located on common deck PXZBLNK.

2.0 I/O PROCEDURES

2.1 BINARY FILES

2.1.1 OPENING AND CLOSING BINARY FILES

2.1.1.1 BI#OPEN - Open Binary File

Opens binary_file as local file_name.

```
*callc pxiotyp

{ BIZOPEN    Opens binary file as local file. }

  PROCEDURE [XREF] bi#open (VAR binary_file: file;
    file_name: string ( * );
    status: file_status;
    mode: file_mode;
    position: file_position);
```

2.1.1.2 BI#CLOSE - Close Binary File

Closes binary_file.

```
*callc pxiotyp

{ BIZCLOS    Closes binary file. }
```

```
    PROCEDURE [XREF] bi#close (binary_file: file;
      disposition: file_disposition);
```

2.1.2 POSITIONING BINARY FILES

2.1.2.1 BI#FIRST - Position Binary File at BOI

Positions binary_file at its beginning of information.

```
*callc pxiotyp

{ BIZFIRS    Positions binary file at its beginning of information. }

  PROCEDURE [XREF] bi#first (binary_file: file);
```

2.1.2.2 BI#LAST - Position Binary File at EOI

Positions binary_file at its end of information.

```
*callc pxiotyp

{ BIZLAST    Positions binary file at its end of information. }

  PROCEDURE [XREF] bi#last (binary_file: file);
```

2.1.3 BINARY FILE STRUCTURE CREATION / DETECTION

2.1.3.1 BI#WEOR - Write End Of Record on Binary File

Writes an End Of Record mark on **binary_file.**

```
*callc pxiotyp

{ BIZWEOR    Writes and End of Record mark on binary file. }

  PROCEDURE [XREF] bi#weor (binary_file: file);
```

2.1.3.2 BI#WEOF - Write End Of File on Binary File

Writes an End Of File mark on **binary_file.**

```
*callc pxiotyp

{ BIZWEOF    Writes an End of File mark on binary file. }

  PROCEDURE [XREF] bi#weof (binary_file: file);
```

2.1.3.3 F#MARK - Check Structure Mark on File

Returns the "file structure mark" last encountered on any_file.

```
*callc pxiotyp

{ FZMARK    Returns the file structure mark last encountered on file. }

  PROCEDURE [XREF] f#mark (any_file: file;
    VAR mark: file_mark);
```

2.1.3.4 F#WORDS - Last Transfer Length on File

Returns the length of the last transfer to/from any_file.

```
*callc pxiotyp

{ FZWORDS    Returns length of last transfer to/from file. }

  PROCEDURE [XREF] f#words (any_file: file;
    VAR last_transfer_length: integer);
```

2.1.4 READING AND WRITING BINARY FILES

Transfer of data to and from binary files is performed in groups of words (cells). Since the data transfer procedures for binary files (like all other programmer defined procedures in CYBIL) must have parameters of a specific type, and since we want to be able to transfer items of any data type to/from a binary file, the objects of transfer are passed to the procedures in two parts: address (usually via the #loc function); and length (usually via the #size function). CYBILIO has no way to insure that the address and length parameters refer to the same object, therefore be warned: "you're on your own" as far as checking parameter correctness is concerned.

2.1.4.1 BI#PUT - Write to Binary File

Writes **length_of_source** words (cells) beginning at the address specified by **pointer_to_source** to **binary_file**.

```
*callc pxiotyp

{ BIZPUT    Writes specified information to binary file. }

  PROCEDURE [XREF] bi#put (binary_file: file;
    pointer_to_source: ^cell;
    length_of_source: integer);
```

2.1.4.2 BI#GET - Read from Binary File

Reads up to **length_of_target** words (cells) from **binary_file** to the address specified by **pointer_to_target**. Note that an "incomplete transfer" can result from this request (see the section on "NOS File Structure Creation / Detection" for more information on this subject).

```
*callc pxiotyp

{ BIZGET    Reads info from binary file to address specified. }

  PROCEDURE [XREF] bi#get (binary_file: file;
    pointer_to_target: ^cell;
```

```
     length_of_target: integer);
```

2.1.5 EXAMPLE - 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 ALIAS "zpxmcop";

*CALL pxiotyp
*CALL bizopen
*CALL bizclos
*CALL bizget
*CALL bizput
*CALL bizweor
*CALL bizweof
*CALL fzmark
*CALL fzwords


  PROGRAM copy ALIAS "zpxpcop";

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

    VAR
      in_file : file,
      out_file : file,
      buffer : ARRAY [1 .. buffer_length] of CELL,
      transfer_length : INTEGER,
      mark : file_mark;

    bi#open (in_file, in_name, old#, input#, first#);
    bi#open (out_file, out_name, new#, output#, first#);
  /main_loop/
    WHILE TRUE DO
      bi#get (in_file, #LOC(buffer), #SIZE(buffer));
      f#mark (in_file, mark);
      CASE mark OF
      =eoi#=
        EXIT /main_loop/;
      =eof#=
        bi#weof (out_file);
      =eor#=
        f#words (in_file, transfer_length);
        IF transfer_length > 0 THEN
          bi#put (out_file, #LOC(buffer), transfer_length);
        IFEND;
        bi#weor (out_file);
      =data#=
        bi#put (out_file, #LOC(buffer), #SIZE(buffer));
      CASEND;
    WHILEND /main_loop/;
    bi#close (in_file, first#);
    bi#close (out_file, first#);

  PROCEND copy;

MODEND copy;
```

2.2 DIRECT FILES

2.2.1 OPENING AND CLOSING DIRECT FILES

2.2.1.1 DI#OPEN - Open Direct File

Opens **direct_file** as local **file_name**. Note: that a direct file cannot be opened at position asis#.

```
*callc pxiotyp

{ DIZOPEN    Opens direct file as local file. }

  PROCEDURE [XREF] di#open (VAR direct_file: file;
    file_name: string ( * );
    status: file_status;
    mode: file_mode;
    position: file_position);
```

2.2.1.2 DI#CLOSE - Close Direct File

Closes **direct_file**.

```
*callc pxiotyp

{ DIZCLOS    Closes direct file. }

  PROCEDURE [XREF] di#close (direct_file: file;
    disposition: file_disposition);
```

2.2.2 POSITIONING DIRECT FILES

2.2.2.1 DI#FIRST - Position Direct File at BOI

Positions **direct_file** at its beginning of information.

```
*callc pxiotyp

{ DIZFIRS    Positions direct file at its beginning of information. }

  PROCEDURE [XREF] di#first (direct_file: file);
```

2.2.2.2 DI#LAST - Position Direct File at EOI

Positions **direct_file** at its end of information.

```
*callc pxiotyp

{ DIZLAST    Positions direct file at its End of Information. }

  PROCEDURE [XREF] di#last (direct_file: file);
```

2.2.2.3 DI#LOCATE - Position Direct File via Key

Positions **direct_file** at the location specified by key. If key specifies a position outside the bounds of the file, then the program is in error.

```
*callc pxiotyp

{ DIZLOCA    Positions direct file at location specified. }

  PROCEDURE [XREF] di#locate (direct_file: file;
    key: integer);
```

2.2.3 DIRECT FILE STRUCTURE CREATION / DETECTION

2.2.3.1 DI#WEOR - Write End Of Record on Direct File

Writes an End Of Record mark on **direct_file**.

```
*callc pxiotyp

{ DIZWEOR    Writes an End of Record mark on direct file. }

  PROCEDURE [XREF] di#weor (direct_file: file);
```

2.2.3.2 DI#WEOF - Write End Of File on Direct File

Writes an End Of File mark on **direct_file**.

```
*callc pxiotyp

{ DIZWEOF    Writes an End of File mark on direct file. }

  PROCEDURE [XREF] di#weof (direct_file: file);
```

2.2.3.3 F#MARK - Check Structure Mark on File

Returns the "file structure mark" last encountered on **any_file**.

```
*callc pxiotyp

{ FZMARK    Returns the file structure mark last encountered on file. }

  PROCEDURE [XREF] f#mark (any_file: file;
    VAR mark: file_mark);
```

2.2.3.4 F#WORDS - Last Transfer Length on File

Returns the length of the last transfer from any_file.

```
*callc pxiotyp

{ FZWORDS    Returns length of last transfer to/from file. }

  PROCEDURE [XREF] f#words (any_file: file;
    VAR last_transfer_length: integer);
```

2.2.4 READING AND WRITING DIRECT FILES

Transfer of data to and from direct files is performed in groups of words (cells). Since the data transfer procedures for direct files (like all other programmer defined procedures in CYBIL) must have parameters of a specific type, and since we want to be able to transfer items of any data type to/from a direct file, the objects of transfer are passed to the procedures in two parts : address (usually via the #loc function); and length (usually via the #size function). CYBILIO has no way to insure that the address and length parameters refer to the same object, therefore be warned: "you're on your own" as far as checking parameter correctness is concerned.

2.2.4.1 DI#PUT - Sequential Write to Direct File

Writes length_of_source words (cells) from the address specified by pointer_to_source to direct_file at its current position. The "random file address" of the data written is returned in key.

```
*callc pxiotyp

{ DIZPUT     Writes info from address spec. to direct file's current pos. }

  PROCEDURE [XREF] di#put (direct_file: file;
    VAR key: integer;
    pointer_to_source: ^cell;
    length_of_source: integer);
```

2.2.4.2 DI#PUTDIR - Random Write to Direct FIle

Writes length_of_source words (cells) from the address specified by pointer_to_source to direct_file at the "random file address" specified by key.

```
*callc pxiotyp

{ DIZPUTD     Writes info to direct file at random address specified. }

  PROCEDURE [XREF] di#putdir (direct_file: file;
    key: integer;
    pointer_to_source: ^cell;
    length_of_source: integer);
```

2.2.4.3 DI#GET - Sequential Read from Direct File

Reads up to **length_of_target** words (cells) from **direct_file** at its current position to the address specified by **pointer_to_target**. The "random file address" of the data read is returned in **key**. Note that an "incomplete transfer" can result from this request (see the section on "NOS File Structure Creation / Detection" for more information on this subject).

```
*callc pxiotyp

{ DIZGET    Reads info from direct file's current position to add. spec. }

  PROCEDURE [XREF] di#get (direct_file: file;
    VAR key: integer;
    pointer_to_target: ^cell;
    length_of_target: integer);
```

2.2.4.4 DI#GETDIR - Random Read from Direct File

Reads up to **length_of_target** words (cells) from **direct_file** at the "random file address" specified by **key** to the address specified by **pointer_to_target**. Note that an "incomplete transfer" can result from this request (see the section on "NOS File Structure Creation / Detection" for more information on this subject).

```
*callc pxiotyp

{ DIZGETD    Reads info from direct file's random file address. }

  PROCEDURE [XREF] di#getdir (direct_file: file;
    key: integer;
    pointer_to_target: ^cell;
    length_of_target: integer);
```

2.2.5 DIRECT FILE STATUS INTERROGATION

2.2.5.1 DI#LENGTH - Direct File Length?

Returns the **file_length_in_words** of **direct_file.**

```
*callc pxiotyp

{ DIZLENG    Returns length in words of direct file. }

  PROCEDURE [XREF] di#length (direct_file: file;
    VAR file_length_in_words: integer);
```

2.2.5.2 DI#KEY - Direct File Current Position?

Returns the **current_position_key** designating the current position of **direct_file.**

```
*callc pxiotyp

{ DIZKEY    Returns the KEY designating direct file"s current position. }

  PROCEDURE [XREF] di#key (direct_file: file;
    VAR current_position_key: integer);
```

2.2.6 EXAMPLES OF DIRECT FILE USAGE

The examples which follow illustrate the use of direct file procedures. The first example creates a "library" of "text modules" from a legible file. The modules on the source (legible) file are represented as NOS records 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.

2.2.6.1 Create_Text_Library

```
MODULE create_text_library ALIAS 'zpxmcre';

*CALL pxiotyp
*CALL lgzopen
*CALL lgzclos
*CALL lgzget
*CALL fzmark
*CALL dizopen
*CALL dizclos
*CALL dizput
*CALL dizputd
*CALL bizopen
*CALL bizclos
*CALL bizput
*CALL bizget


  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',
     dir_name = 'SCRATCH';


   PROGRAM create ALIAS 'zpxpcre';

     VAR
       source : file,
       library : file,
       dir_file : file,
       directory : directory_descriptor,
       current_module : directory_entry,
       line : STRING (256),
       line_length : INTEGER,
       module_index : INTEGER,
       first_key : INTEGER,
       dummy_key : INTEGER,
       mark : file_mark;

     lg#open (source, source_name, old#, input#, first#);
     bi#open (dir_file, dir_name, new#, output#, first#);
     di#open (library, lib_name, new#, output#, first#);
     directory.length := 0;
     di#put (library, first_key,
             #LOC(directory), #SIZE(directory));

   /copy_module_loop/
     WHILE TRUE DO
       lg#get (source, line_length, line);
       f#mark (source, mark);
       CASE mark OF
       =eol#=
         EXIT /copy_module_loop/;
       =eof#, eor#=
         CYCLE /copy_module_loop/;
       =data#=
         directory.length := directory.length + 1;
         current_module.name := line(1, line_length);
         current_module.length := 1;
         di#put (library, current_module.key,
                 #LOC(current_module.name),
                 #SIZE(current_module.name));
       /copy_text_loop/
         WHILE TRUE DO
           lg#get (source, line_length, line);
           f#mark (source, mark);
           IF mark <> data# THEN
             EXIT /copy_text_loop/;
           IFEND;
```

```
          current_module.length := current_module.length + 1;
          di#put (library, dummy_key,
                  #LOC(line_length), #SIZE(line_length));
          di#put (library, dummy_key,
                  #LOC(line), #SIZE(line(1, line_length)));
        WHILEND /copy_text_loop/;
        bi#put (dir_file, #LOC(current_module),
                          #SIZE(current_module));
      CASEND;
    WHILEND /copy_module_loop/;
    lg#close (source, first#);

    IF directory.length > 0 THEN
      bi#close (dir_file, asis#);
      bi#open (dir_file, dir_name, old#, input#, first#);
      bi#get (dir_file, #LOC(current_module),
                        #SIZE(current_module));
      di#put (library, directory.key,
              #LOC(current_module), #SIZE(current_module));
      FOR module_index := 2 TO directory.length DO
        bi#get (dir_file, #LOC(current_module),
                          #SIZE(current_module));
        di#put (library, dummy_key,
                #LOC(current_module), #SIZE(current_module));
      FOREND;
      di#putdir (library, first_key,
                 #LOC(directory), #SIZE(directory));
    IFEND;

    bi#close (dir_file, return#);
    di#close (library, first#);

  PROCEND create;

 MODEND create_text_library;
```

2.2.6.2 <u>Extract from Text Library</u>

```
 MODULE extract_from_text_library ALIAS 'zpxmefl';

 *CALL pxiotyp
 *CALL dizopen
 *CALL dizclos
 *CALL dizloca
 *CALL dizgetd
 *CALL dizget
 *CALL lgzopen
 *CALL lgzclos
 *CALL lgzput


   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 'zpxpefl';

     VAR
       library : file,
       out_file : file,
       directory : directory_descriptor,
       current_module : directory_entry,
       line : STRING (256),
       line_length : INTEGER,
       module_index : INTEGER,
       dummy_key : INTEGER;

     di#open (library, lib_name, old#, input#, first#);
     di#get (library, dummy_key,
             #LOC(directory), #SIZE(directory));
     IF directory.length = 0 THEN
       { ERROR - module not found }
       RETURN;
     IFEND;
     di#locate (library, directory.key);
   /search_directory/
     BEGIN
```

```
      FOR module_index := 1 TO directory.length DO
        di#get (library, dummy_key,
                #LOC(current_module), #SIZE(current_module));
        IF current_module.name = name_of_module THEN
          EXIT /search_directory/;
        IFEND;
      FOREND;
      { ERROR - module not found }
      RETURN;
    END /search_directory/;
    lg#open (out_file, name_of_module, new#, output#, first#);
    di#getdir (library, current_module.key,
              #LOC(current_module.name),
              #SIZE(current_module.name));
    lg#put (out_file, current_module.name);
    WHILE current_module.length > 1 DO
      di#get (library, dummy_key,
              #LOC(line_length), #SIZE(line_length));
      di#get (library, dummy_key,
              #LOC(line), #SIZE(line(1, line_length)));
      lg#put (out_file, line(1, line_length));
      current_module.length := current_module.length - 1;
    WHILEND;
    di#close (library, first#);
    lg#close (out_file, first#);

  PROCEND extract;

 MODEND extract_from_text_library;
```

2.3 LEGIBLE FILES

2.3.1 OPENING AND CLOSING LEGIBLE FILES

2.3.1.1 LG#OPEN - Open Legible File

Opens legible_file as local file_name.

```
*callc pxiotyp

{ LGZOPEN    Opens legible file as local file. }

  PROCEDURE [XREF] lg#open (VAR legible_file: file;
    file_name: string ( * );
    status: file_status;
    mode: file_mode;
    position: file_position);
```

2.3.1.2 F#SABF - Setup File for Automatic Buffer Flushing

Sets up any_file so that its CIO buffer will automatically be flushed (if necessary) whenever the program is rolled out. This facility is useful when a program issues prompts to a terminal user and then requests input, since normally to insure the prompt reaches the user before the input request, the program would write an End Of Record (causing the buffer to be flushed). The mechanism used is described in the NOS Reference Manual in the section on "Program Writing Techniques".

```
*callc pxiotyp

{ FZSABF    Sets up file for automatic buffer flushing. }

  PROCEDURE [XREF] f#sabf (any_file: file);
```

2.3.1.3 LG#CODESET - Set Legible File Character Set

Sets the external character set for legible_file to codeset (default, on open, is asc1i612#).

```
*callc pxiotyp

{ LGZCODE    Sets external character set for legible file. }

  PROCEDURE [XREF] lg#codeset (legible_file: file;
    codeset: file_encoding);
```

2.3.1.4 LG#CLOSE - Close Legible File

Closes legible_file.

```
*callc pxiotyp

{ LGZCLOS    Closes legible file. }

  PROCEDURE [XREF] lg#close (legible_file: file;
    disposition: file_disposition);
```

2.3.2 POSITIONING LEGIBLE FILES

2.3.2.1 LG#FIRST - Position Legible File at BOI

Positions legible_file at its beginning of information.

```
*callc pxiotyp

{ LGZFIRS    Positions legible file at its Beginning Of Information. }

  PROCEDURE [XREF] lg#first (legible_file: file);
```

2.3.2.2 LG#LAST - Position Legible File at EOI

Positions legible_file at its end of information.

```
*callc pxiotyp

{ LGZLAST    Positions legible file at its End Of Information. }

  PROCEDURE [XREF] lg#last (legible_file: file);
```

2.3.2.3 LG#TAB - Position Legible File at Column

If column_number is less than or equal to legible_file's current column or if it is greater than 256, this procedure does nothing. Otherwise, sufficient space characters are written to legible_file so that the next (partial) write to legible_file will begin at the specified column_number.

```
*callc pxiotyp

{ LGZTAB    Positions column of next partial write to legible file. }
```

2.0 I/O PROCEDURES
2.3.2.3 LG#TAB - Position Legible File at Column

```
   PROCEDURE [XREF] lg#tab (legible_file: file;
     column_number: integer);
```

2.3.3 LEGIBLE FILE STRUCTURE CREATION / DETECTION

2.3.3.1 LG#WEOR - Write End Of Record on Legible File

Writes an End Of Record mark on legible_file.

```
*callc pxiotyp

{ LGZWEOR    Writes an End Of Record mark on legible file. }

  PROCEDURE [XREF] lg#weor (legible_file: file);
```

2.3.3.2 LG#WEOF - Write End Of File on Legible File

Writes an End Of File mark on legible_file.

```
*callc pxiotyp

{ LGZWEOF    Writes an End Of File mark on legible file. }

  PROCEDURE [XREF] lg#weof (legible_file: file);
```

2.3.3.3 F#MARK - Check Structure Mark on File

Returns the "file structure mark" last encountered on any_file.

```
*callc pxiotyp

{ FZMARK    Returns the file structure mark last encountered on file. }

  PROCEDURE [XREF] f#mark (any_file: file;
    VAR mark: file_mark);
```

2.3.3.4 F#WORDS - Last Transfer Length on File

Returns the last_transfer_length of the last transfer to/from any_file.

```
*callc pxiotyp

{ FZWORDS    Returns length of last transfer to/from file. }

  PROCEDURE [XREF] f#words (any_file: file;
    VAR last_transfer_length: integer);
```

2.3.4 READING AND WRITING LEGIBLE FILES

Data is transferred to and from legible files in terms of lines or partial lines. Internally these (partial) lines are represented by CYBIL strings of characters. Externally (on the file) lines are represented in 6-bit display code, NOS 6/12-bit ASCII, or "8 out of 12 bit" ASCII. Thus, data transfers involving legible files imply a translation between these character sets (unlike binary and direct file transfers in which the data are not modified).

Note: that when reading from a legible file assigned to an interactive terminal, the only file mark possible is data#. Any eor# or eof# marks returned to CYBILIO by NOS after a read from a "terminal file" are discarded by CYBILIO (eoi# is never possible from a terminal). A line (entered at a terminal) containing zero characters (i.e., the carriage return key was "hit" in the first position of the line) is returned to the CYBILIO user as an empty line.

2.3.4.1 LG#PUT - Write Line to Legible File

Writes the line string as a complete line to legible_file. If the last write to legible_file was a partial line, that line is first completed, and then the line is written.

```
*callc pxiotyp

{ LGZPUT    Writes source string as complete line to legible file. }

  PROCEDURE [XREF] lg#put (legible_file: file;
    line: string ( * ));
```

2.3.4.2 LG#PUTPART - Write Partial Line to Legible File

Writes the partial_line string to legible_file. If last_part_of_line is TRUE, then the line is completed after partial_line is written by writing an End Of Line to legible_file.

```
*callc pxiotyp
```

```
{ LGZPUTP    Writes source string as partial line to legible file. }

  PROCEDURE [XREF] lg#putpart (legible_file: file;
    last_part_of_line: boolean;
    partial_line: string ( * ));
```

2.3.4.3 LG#WEOL - Write End Of Line on Legible File

Writes an End Of Line to **legible_file**. If the last write to **legible_file** was partial, that line is completed; otherwise an empty line results.

```
*callc pxiotyp

{ LGZWEOL    Writes an End Of Line to legible file. }

  PROCEDURE [XREF] lg#weol (legible_file: file);
```

2.3.4.4 LG#GET - Read Line from Legible File

Reads the next complete line from **legible_file** into **line**. The actual number of characters transferred to **line** is returned in **number_of_characters_read**. If the previous transfer was partial, a skip to the end of that line is performed prior to the transfer to **line** being done. If the line from **legible_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.

```
*callc pxiotyp

{ LGZGET    Reads next complete line from legible file. }

  PROCEDURE [XREF] lg#get (legible_file: file;
    VAR number_of_characters_read: integer;
    VAR line: string ( * ));
```

2.3.4.5 LG#GETPART - Read Partial Line from Legible File

Reads the next partial line from legible_file into partial_line. last_part_of_line will be set to TRUE if the end of the line was encountered, and set to FALSE otherwise. The actual number of characters transferred will be returnec in number_of_characters_read.

```
*callc pxiotyp

{ LGZGETP    Reads next partial line from legible file. }

  PROCEDURE [XREF] lg#getpart (legible_file: file;
    VAR last_part_of_line: boolean;
    VAR number_of_characters_read: integer;
    VAR partial_line: string ( * ));
```

2.3.5 LEGIBLE FILE STATUS INTERROGATION

2.3.5.1 F#TERMINAL - File is a Terminal?

Returns in **file_is_a_terminal** TRUE if **any_file** is connected to a terminal, and FALSE otherwise.

```
*callc pxiotyp

{ FZTERMI    Returns boolean if file is connected to a terminal. }

  PROCEDURE [XREF] f#terminal (any_file: file;
    VAR file_is_a_terminal: boolean);
```

2.3.5.2 LG#OLDCODESET - Legible File Character Set?

Returns the designator for the external character set associated with **legible_file**.

```
*callc pxiotyp

{ LGZOLDC    Returns designator for ext. char. set of legible file. }

  PROCEDURE [XREF] lg#oldcodeset (legible_file: file;
    VAR codeset: file_encoding);
```

2.3.5.3 LG#COLNO - Legible File Column Number?

Returns the number of the column within the current line of **legible_file** that was last transferred to/from **legible_file**. Put another way, **column_number** is set to the number of characters so far transferred to/from the current line of **legible_file**.

```
*callc pxiotyp
```

2.3.5.3 LG#COLNO - Legible File Column Number?

```
 { LGZCOLN    Returns col. no. in line of legible file last transferred. }

   PROCEDURE [XREF] lg#colno (legible_file: file;
     VAR column_number: integer);
```

2.3.6 EXAMPLE - COPY COLUMN RANGE OF LEGIBLE FILE

The following example illustrates the use of legible file procedures to copy one legible 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 'zpxmtru';

*CALL pxiotyp
*CALL lgzopen
*CALL lgzclos
*CALL lgzget
*CALL lgzput
*CALL lgzweol
*CALL lgzweor
*CALL lgzweof
*CALL fzmark


  PROGRAM truncate ALIAS 'zpxptru';

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

    VAR
      in_file : file,
      out_file : file,
      line_ptr : ^STRING (*),
      line_length : INTEGER,
      mark : file_mark;

    ALLOCATE line_ptr : [rightmost_column_#];
    lg#open (in_file, in_name, old#, input#, first#);
    lg#open (out_file, out_name, new#, output#, first#);
  /main_loop/
    WHILE TRUE DO
      lg#get (in_file, line_length, line_ptr^);
      f#mark (in_file, mark);
      CASE mark OF
      =eol#=
        EXIT /main_loop/;
      =eof#=
        lg#weof (out_file);
      =eor#=
        lg#weor (out_file);
      =data#=
        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
          lg#put (out_file, line_ptr^(leftmost_column_#,
                                      line_length));
        ELSE
          lg#weol (out_file);
        IFEND;
      CASEND;
    WHILEND /main_loop/;
    lg#close (in_file, first#);
    lg#close (out_file, first#);
    FREE line_ptr;

  PROCEND truncate;

MODEND truncate;
```

2.4 PRINT FILES

2.4.1 OPENING AND CLOSING PRINT FILES

2.4.1.1 PR#OPEN - Open Print File

Opens **print_file** as local **file_name** (note the lack of a **file_mode** parameter for this procedure, since print files can only be written).

```
*callc pxiotyp

{ PRZOPEN    Opens print file as local file. }

  PROCEDURE [XREF] pr#open (VAR print_file: file;
    file_name: string ( * );
    status: file_status;
    position: file_position);
```

2.4.1.2 PR#PGOV - Define Page Overflow Procedure

Associates with **print_file**, the procedure designated by **page_overflow_proc** which will be called whenever the page size of **print_file** is exceeded. Page size is set by **PR#LIMIT**, default is 60 lines. The procedure designated by **page_overflow_proc** should not be called directly by the user. If the user wishes to explicitly advance to the next page, a call to the pr#page procedure should be used.

If there is no user supplied page overflow procedure for a print file, then CYBILIO simply performs a page eject for the file when the page overflow condition occurs. If NIL is specified for **page_overflow_proc**, any user supplied page overflow procedure currently associated with the file is disassociated from the file.

```
*callc pxiotyp
```

```
{ PRZPGOV    Calls procedure needed to advance file to next page. }

  PROCEDURE [XREF] pr#pgov (print_file: file;
    page_overflow_proc: ^procedure (print_file: file;
    next_page_#: integer));
```

2.4.1.3 PR#CODESET - Set Print File Character Set

Sets the external character set for print_file to codeset (default, on open, is ascii612#).

```
*callc pxiotyp

{ PRZCODE    Sets ext. char. set for print file. }

  PROCEDURE [XREF] pr#codeset (print_file: file;
    codeset: file_encoding);
```

2.4.1.4 PR#LIMIT - Set Print File Page Size

Sets the page size (line limit) for print_file to lines_per_page (default, on open, is 60).

```
*callc pxiotyp

{ PRZLIMI    Sets page size (line limit) for print file. }

  PROCEDURE [XREF] pr#limit (print_file: file;
    lines_per_page: integer);
```

2.4.1.5 PR#SETPGNO - Set Print File Page Number

Sets the current page number for print_file to current_page_number (default, on open, is 0).

```
*callc pxiotyp

{ PRZSETP    Sets current page number for print file. }

  PROCEDURE [XREF] pr#setpgno (print_file: file;
    current_page_number: integer);
```

2.4.1.6 PR#CLOSE - Close Print File

Closes print_file.

```
*callc pxiotyp

{ PRZCLOS    Closes print file. }

  PROCEDURE [XREF] pr#close (print_file: file;
    disposition: file_disposition);
```

2.4.2 POSITIONING PRINT FILES

2.4.2.1 PR#FIRST - Position Print File at BOI

Positions **print_file** at its beginning of information.

```
*callc pxiotyp

{ PRZFIRS    Positions print file at its Beginning Of Information. }

  PROCEDURE [XREF] pr#first (print_file: file);
```

2.4.2.2 PR#LAST - Position Print File at EOI

Positions **print_file** at its end of information.

```
*callc pxiotyp

{ PRZLAST    Positions print file at its End Of Information. }

  PROCEDURE [XREF] pr#last (print_file: file);
```

2.4.2.3 PR#TAB - Position Print File at Column

If **column_number** is less than or equal to **print_file**'s current column or if it is greater than 136, this procedure does nothing. Otherwise, sufficient space characters are written to **print_file** so that the next (partial) write to **print_file** will begin at the specified **column_number**.

```
*callc pxiotyp

{ PRZTAB    Positions print file at column for next partial write. }

  PROCEDURE [XREF] pr#tab (print_file: file;
    column_number: integer);
```

2.4.2.4 PR#LINE - Position Print File at Line

Positions print_file at the specified line_number. This will be on the current page if line_number is greater than the current line number and less than or equal to page size; or on the next page (after invoking the page overflow mechanism) if line_number is less than or equal to the current line number. If line_number is greater than the page size, the file will be positioned at the top of the next page.

```
*callc pxiotyp

{ PRZLINE    Positions print file at specified line. }

  PROCEDURE [XREF] pr#line (print_file: file;
    line_number: integer);
```

2.4.2.5 PR#SKIP - Skip Lines on Print File

If number_of_lines = -1, the next line written to print_file will overprint the current line. If number_of_lines + print_file's current line number is greater than the page size, the page overflow mechanism is invoked. Otherwise, number_of_lines empty lines will be written to print_file.

```
*callc pxiotyp

{ PRZSKIP    Skips lines on print file from current position. }

  PROCEDURE [XREF] pr#skip (print_file: file;
    number_of_lines: integer);
```

2.4.2.6 PR#EJECT - Position Print File at Top of Page

Positions **print_file** at the first line (top) of the next page. This procedure should only be called by the routine that processes page overflow conditions: pr#pgov (see the section on "Print Files" under "File Types").

```
*callc pxiotyp

{ PRZEJEC    Positions print file to first line (top) of next page. }

  PROCEDURE [XREF] pr#eject (print_file: file);
```

2.4.2.7 PR#PAGE - Start New Page on Print File

Increments **print_file**'s page number and calls the routine that processes page overflow conditions: pr#pgov (see the section on "Print Files" under "File Types").

```
*callc pxiotyp

{ PRZPAGE    Increments print file's page number. }

  PROCEDURE [XREF] pr#page (print_file: file);
```

2.4.3 PRINT FILE STRUCTURE CREATION

2.4.3.1 PR#WEOR - Write End Of Record on Print File

Writes an End Of Record mark on **print_file.**

```
*callc pxiotyp

{ PRZWEOR    Writes an End Of Record mark on print file. }

  PROCEDURE [XREF] pr#weor (print_file: file);
```

2.4.3.2 PR#WEOF - Write End Of File on Print File

Writes an End Of File mark on **print_file.**

```
*callc pxiotyp

{ PRZWEOF    Writes an End Of File mark on print file. }

  PROCEDURE [XREF] pr#weof (print_file: file);
```

2.4.4 WRITING PRINT FILES

Print files are a special form of legible files used only for output. In addition to the (partial) line writes similar to those for legible files, print files also have "format" control procedures to handle page overflow processing and vertical spacing and tabbing (see the section on "Positioning Print Files" for more information).

2.4.4.1 PR#PUT - Write Line to Print File

Writes the line string as a complete line to print_file. If the last write to print_file was a partial line, that line is first completed, and then the line for line is written.

```
*callc pxiotyp

{ PRZPUT    Writes source string as a complete line to print file. }

  PROCEDURE [XREF] pr#put (print_file: file;
    line: string ( * ));
```

2.4.4.2 PR#PUTPART - Write Partial Line to Print File

Writes the partial_line string to print_file. If last_part_of_line is TRUE, then the line is completed after partial_line is written by writing an End Of Line to print_file.

```
*callc pxiotyp

{ PRZPUTP    Writes source string as a partial line to print file. }

  PROCEDURE [XREF] pr#putpart (print_file: file;
    last_part_of_line: boolean;
    partial_line: string ( * ));
```

2.4.4.3 PR#WEOL - Write End Of Line on Print File

Writes an End Of Line to print_file. If the last write to print_file was partial, that line is completed; otherwise an empty line results.

```
*callc pxiotyp

{ PRZWEOL    Writes an End Of Line to print file. }

  PROCEDURE [XREF] pr#weol (print_file: file);
```

2.4.5 PRINT FILE STATUS INTERROGATION

2.4.5.1 PR#OLDCODESET - Print File Character Set?

Returns the designator for the external character set associated with print_file.

```
*callc pxiotyp

{ PRZOLDC     Returns designator for ext. char. set of print file. }

  PROCEDURE [XREF] pr#oldcodeset (print_file: file;
    VAR codeset: file_encoding);
```

2.4.5.2 PR#COLNO - Print File Column Number?

Returns the number of the column within the current line of print_file that was last transferred to/from print_file. Put another way, column_number is set to the number of characters so far transferred to/from the current line of print_file.

```
*callc pxiotyp

{ PRZCOLN     Returns current line col. no. of print file last transferred.

  PROCEDURE [XREF] pr#colno (print_file: file;
    VAR column_number: integer);
```

2.4.5.3 PR#LINO - Print File Line Number?

Returns the number of the current line within the current page of print_file. After any repositioning command (skip, eject, set_line_number) the line_number returned is the next line to be printed. After a print command (put, putpart, weol), the line_number is the line just printed.

Before doing any I/O, the line number is 500.

```
*callc pxiotyp

{ PRZLINO    Returns no. of lines within current page of print file. }

  PROCEDURE [XREF] pr#lino (print_file: file;
    VAR line_number: integer);
```

2.4.5.4 PR#PGNO - Print File Page Number?

Returns the number of the current page for print_file.

```
*callc pxiotyp

{ PRZPGNO    Returns number of current page for print file. }

  PROCEDURE [XREF] pr#pgno (print_file: file;
    VAR page_number: integer);
```

2.4.5.5 PR#OLDLIMIT - Print File Page Size?

Returns print_file's page size (line limit).

```
*callc pxiotyp

{ PRZOLDL    Returns print file's page size (line limit). }

  PROCEDURE [XREF] pr#oldlimit (print_file: file;
    VAR lines_per_page: integer);
```

2.4.6 EXAMPLE - LIST LEGIBLE FILE

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

```
MODULE list ALIAS 'zpxmlis';

*CALL pxiotyp
*CALL lgzopen
*CALL lgzclos
*CALL lgzgetp
*CALL fzmark
*CALL przopen
*CALL przpgov
*CALL przclos
*CALL przejec
*CALL przskip
*CALL przline
*CALL przlimi
*CALL przoldl
*CALL przpgno
*CALL prztab
*CALL przputp
*CALL przweol

  CONST
    in_name = 'LEGFILE';

  VAR
    file_# : INTEGER := 1,
    record_# : INTEGER := 1;


  PROCEDURE page_overflow_handler
      (    f : file;
           next_page_# : INTEGER);

    VAR
      conv_holder : STRING (10),
      conv_length : INTEGER,
      old_page_size : INTEGER;
      IF next_page_# > 1 THEN
        pr#oldlimit (f, old_page_size);
        pr#limit (f, old_page_size + 2);
        pr#line (f, old_page_size + 2);
        pr#tab (f, 70);
        pr#putpart (f, FALSE, 'PAGE ');
        STRINGREP (conv_holder, conv_length, next_page_# - 1);
        pr#putpart (f, TRUE, conv_holder(1, conv_length));
        pr#limit (f, old_page_size);
      IFEND;
      pr#eject (f);
      pr#putpart (f, FALSE, 'LISTING OF ');
      pr#putpart (f, FALSE, in_name);
      pr#tab (f, 50);
      pr#putpart (f, FALSE, 'FILE ');
      STRINGREP (conv_holder, conv_length, file_#);
      pr#putpart (f, FALSE, conv_holder(1, conv_length));
      pr#putpart (f, FALSE, ', RECORD ');
      STRINGREP (conv_holder, conv_length, record_#);
      pr#putpart (f, TRUE, conv_holder(1, conv_length));
      pr#skip (f, 2);

    PROCEND page_overflow_handler;


    PROGRAM list ALIAS 'zpxplis';

      CONST
        out_name = 'OUTPUT';

      VAR
        in_file : file,
        out_file : file,
        original_page_size : INTEGER,
        page_# : INTEGER,
        line : STRING (80),
        line_length : INTEGER,
        eol : BOOLEAN,
        mark : file_mark;

      lg#open (in_file, in_name, old#, input#, first#);
      pr#open (out_file, out_name, new#, asis#);
      pr#pgov (out_file, ^page_overflow_handler);
      pr#oldlimit (out_file, original_page_size);
      pr#limit (out_file, original_page_size - 2);
    /main_loop/
      WHILE TRUE DO
        lg#getpart (in_file, eol, line_length, line);
        f#mark (in_file, mark);
        CASE mark OF
        =eoi#=
          pr#limit (out_file, original_page_size);
          pr#line (out_file, original_page_size);
          pr#tab (out_file, 70);
          pr#putpart (out_file, FALSE, 'PAGE ');
          pr#pgno (out_file, page_#);
          STRINGREP (line, line_length, page_#);
          pr#putpart (out_file, TRUE, line(1, line_length));
          EXIT /main_loop/;
        =eof#=
          file_# := file_# + 1;
          record_# := 1;
          pr#line (out_file, original_page_size - 2);
          pr#weol (out_file);
        =eor#=
          record_# := record_# + 1;
          pr#line (out_file, original_page_size - 2);
          pr#weol (out_file);
        =data#=
          IF line_length > 0 THEN
            pr#putpart (out_file, eol, line(1, line_length));
          ELSE
            pr#weol (out_file);
          IFEND;
        CASEND;
      WHILEND /main_loop/;
      lg#close (in_file, first#);
      pr#close (out_file, asis#);

    PROCEND list;

  MODEND list;
```

3.0 CYBILIO ERROR MESSAGES

This section describes the error messages that may be received as a result of improper use of CYBILIO. If a condition described by one of these messages arises:

- the I/O error message will be sent to the dayfile
- the message - INTERNAL ERROR IN prognam will be sent to the dayfile (where prognam is the name of the program as extracted from the job communication area)
- the program is aborted.

In the message prototypes that follow filenam will be replaced by the name of the file in question when the message appears in the dayfile. The reason that some of the messages do not have the file name in them is that, in those conditions, the file name is not known.

-IO ERR- NO MEM TO OPEN FILE filenam

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

-IO ERR- ILLEGAL FILE NAME

This message means that an attempt was made to open a file with a name that did not consist of from 1 to 7 letters and/or digits.

-IO ERR- ILLEGAL OPEN REQ filenam

This message means that an invalid combination of parameters was given to an open procedure (e.g., "new#, input#" is illegal).

-IO ERR- FILE NOT OPEN

This message indicates that an undefined variable of type file was passed to a CYBILIO procedure other than one of the open procedures.

3.0 CYBILIO ERROR MESSAGES

-IO ERR- ILLEGAL INPUT REQ filenam

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

-IO ERR- ILLEGAL OUTPUT REQ filenam

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

-IO ERR- KEY BEYOND E-O-I filenam

This message indicates that an attempt was made to perform a direct 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).

-IO ERR- ILLEGAL LINE NUM filenam

This message means that the pr#line procedure was passed a line number less than 1.

-IO ERR- ILLEGAL SKIP COUNT filenam

This message indicates that the pr#skip procedure was passed a skip count less than -1.

Table of Contents