1.0 INTERFACE_AND_FORMAT_SPECIFICATIONS

1.1 INTERFACE_SPECIFICATION

1.1.1 INTRODUCTION

This document will describe the interface to those routines known collectively as the run time library. Those routines are generally grouped as data editing, user callable, and intrinsic functions. Although the library's conception and design was primarily for the FORTRAN language, there is nothing inherently present to prevent other language processors from interfacing to it. The implementation will be in CYBIL unless a deviation is approved (see PJP, DCS Log ID S3138 and IPP, DCS Log ID S3243).

1.1.2 DOCUMENT FORMAT

This document expresses the interface Predominantly in the form of CYBIL declarations. Each area of the interface is contained in a separate MADIFY common deck on a PL available from the FRTL project. The name of the common deck precedes each set of declarations in the text of the document. The name is enclosed in braces (e.g., {FTTGLOB}). Symbols in the common decks that are global to the compiler and the library start with the letters "ft". Symbols specific to the library start with the letters "fl". The third and fourth characters are defined according to SIS conventions.

1.1.3 APPLICABLE DOCUMENTS

1.1.3.1 FORTRAN_Language_Processor

1) American National Standards Programming Language FORTRAN,
ANSI X3.9 - 1978 FORTRAN.
2) Cyber 180 FORTRAN External Reference Specification,
DCS Log Id S2650.
3) Cyber 180 System Interface Specification,
DCS Log Id S2196.
4) Cyber 180 CCM Interface Specification,
DCS Log Id S2987.
5) Cyber 180 Products Error Termination,
DCS Log Id S2771.
6) Cyber 180 Project Plan[PJP],
DCS Log Id S3138.
7) Cyber 180 Implementation Plan[IPP],
DCS Log Id S3243.

1.1.3.2 Other Language Processors

Not applicable at this time of design.

2.0 GLOBAL CONSTANT AND TYPE DEFINITIONS

String fields supplied to the library from the compiler (e.g., symbolic names) shoulld be blank filled to facilitate compares.

2.1 FTN/FRTL GLOBAL

```
  ?? SKIP := 2 ??
  ?? PUSH (LIST := ON) ??
  { FTTGLOB - global types for FTN and FRTL }
  ?? POP ??

  CONST
    ftc$max_arguments = 500,
    ftc$max_dims = 7,
    ftc$max_char_len = 0ffff(16),
    ftc$max_cybil_string_len = 0ffff(16),
    ftc$max_array_elements = 80000000(16),
    ftc$max_char_ord = 255,
    ftc$max_symbolic_name_len = 7,
    ftc$max_char_per_seg = osc$max_segment_length,
    ftc$bits_per_char = 8,
    ftc$bits_per_word = 64,
    ftc$chars_per_word = 8,
    ftc$chars_per_double_word = 2 * ftc$chars_per_word,
    ftc$max_single_word_per_seg = ftc$max_char_per_seg DIV
      ftc$chars_per_word,
    ftc$max_boolean_per_seg = ftc$max_single_word_per_seg,
    ftc$max_logical_per_seg = ftc$max_single_word_per_seg,
    ftc$max_real_per_seg = ftc$max_single_word_per_seg,
    ftc$max_complex_per_seg = ftc$max_single_word_per_seg DIV 2,
    ftc$max_double_word_per_seg = ftc$max_single_word_per_seg DIV 2,
    ftc$max_fmt_nesting_level = 9,
    ftc$max_num_namelist_group_item = 20000,
    ftc$default_print_limit = 5000;

  CONST
    { Values of LOGICAL constants.

    ftc$true = -1,
    ftc$false = 0;

  CONST
    {Values to be used to return string comparison results to the
    {compiler.}
    {These values correspond to those generated by the hardware
    {compare instructions.}

    ftc$left_lt_right = 0c0000000(16),
    ftc$left_eq_right = 0,
  ftc$left_gt_right = 04000000(16);

TYPE
  ftt$dimension = 1 .. ftc$max_dims,
  ftt$char_size = 0 .. ftc$max_char_ord, {for characters returned as
    {integer}
  {value}
  ftt$compare_result = 0 .. 0ffffffff(16),
  ftt$char_element_size = 1 .. ftc$max_char_len,
  ftt$arg_ordinal = 1 .. ftc$max_arguments,
  ftt$symbolic_name = string (ftc$max_symbolic_name_len);

{ The following type defines the format of a FORTRAN actual argument
{list}
{item for a character data type entity.}

TYPE
  ftt$string_vector_ptr = record
    case 1 .. 3 of
    = 1 =
      str: ^string ( * ),
    = 2 =
      vec: ^ftt$max_char_vector,
      len: 0 .. 0ffff(16),
    = 3 =
      ch: ^char,
      unused: 0 .. 0ffff(16)
    casend
  recend;

{ ftt$string_desc type is used for passing strings to the support
{and/or}
{checking routine for manipulating character data.}

TYPE
  ftt$string_desc = record
    char_string: ^^string ( * ),
    case is_substring: boolean of
    = TRUE =
      substring_begin: ftt$char_element_size,
      substring_end: ftt$char_element_size,
    = FALSE =
      ,
    casend,
  recend;

TYPE
  ftt$data_type = (ftc$boolean, ftc$logical, ftc$integer, ftc$real,
    ftc$double, ftc$complex, ftc$character, ftc$typeless);

{ The long_real declaration should be changed to longreal when CYBIL
{supports this type.}
```

```
TYPE
  long_real = record
    up: real,
    low: real,
  recend,
  ftt$logical = integer,
  ftt$boolean = integer;

TYPE
  ftt$single_word_type = ftc$boolean .. ftc$real,

  ftt$single_word = record
    case 1 .. 5 of
    = 1 =
      l: ftt$logical,
    = 2 =
      b: ftt$boolean,
    = 3 =
      i: integer,
    = 4 =
      r: real,
    = 5 =
      s: string (ftc$chars_per_word)
    casend,
  recend,


  ftt$double_word_type = ftc$double .. ftc$complex,

  ftt$double_word = record
    case 1 .. 3 of
    = 1 =
      d: long_real,
    = 2 =
      c: ftt$complex,
    = 3 =
      s: string (ftc$chars_per_double_word)
    casend
  recend,

  ftt$complex = record
    real_: real,
    imag: real,
  recend;

TYPE

  ftt$max_single_word_vector = array [1 ..
    ftc$max_single_word_per_seg] of ftt$single_word,
  ftt$max_real_vector = array [1 .. ftc$max_real_per_seg] of real,
  ftt$max_double_word_vector = array [1 ..
    ftc$max_double_word_per_seg] of ftt$double_word,
  ftt$max_char_vector = array [1 .. ftc$max_char_per_seg] of char,
  ftt$max_int_vector = array [1 .. ftc$max_single_word_per_seg] of
    integer,
  ftt$max_boolean_vector = array [1 .. ftc$max_boolean_per_seg] of
    ftt$boolean,
  ftt$max_logical_vector = array [1 .. ftc$max_logical_per_seg] of
    ftt$logical,
  ftt$max_long_real_vector = array [1 ..
    ftc$max_double_word_per_seg] of long_real,
  ftt$max_complex_vector = array [1 .. ftc$max_complex_per_seg] of
    ftt$complex;

{ This type defines the components of a CYBER 180 real number. It is
{used in}
{examining the parts of a real number as in LEGVAR.}

TYPE

  ftt$parsed_real = packed record
    sign: 0 .. 1,
    expon: 0 .. 07fff(16),
    norm: 0 .. 1,
    coeff: 0 .. 07fffffffffff(16),
  recend;



TYPE
  ftt$arg_list = array [ftt$arg_ordinal] of ftt$arg_list_item,

  ftt$arg_list_item = record
    case 1 .. 9 of
    = 1 =
      logical_addr: ^ftt$logical,
      l_fill: 0 .. 0ffff(16),
    = 2 =
      boolean_addr: ^ftt$boolean,
      b_fill: 0 .. 0ffff(16),
    = 3 =
      integer_addr: ^integer,
      i_fill: 0 .. 0ffff(16),
    = 4 =
      real_addr: ^real,
      r_fill: 0 .. 0ffff(16),
    = 5 =
      long_real_addr: ^long_real,
      lr_fill: 0 .. 0ffff(16),
    = 6 =
      complex_addr: ^ftt$complex,
      c_fill: 0 .. 0ffff(16),
    = 7 =
      char_addr: ^string ( * ),
    = 8 =
      string_Vec_addr: ftt$string_vector_ptr,
    = 9 =
      cell_addr: ^cell,
      cell_fill: 0 .. 0ffff(16)
    casend,
  recend;

TYPE

  ftt$version = record
    version: 1 .. 255, {FTN 1.0, 2.0,...}
    level: 0 .. 255, {FTN x.0, x.1,...}
    date: ost$date, {date the compiler was built in ISO format}
  recend;



{Pointers created by the ASSIGN statement will have a type field
{carried in}
{the upper 16 bits to allow for compiler and library validation of
{the}
{contents. The field value is chosen so that most integer values
{will not}
{appear to be valid label addresses.}

TYPE
  ftt$pointer_type = (ftc$data, ftc$assigned_executable_label,
    ftc$assigned_format_label);

TYPE
  ftt$pointer = packed record
    case ptr_type: 0 .. 0ffff(16) of
    = ORD (ftc$assigned_format_label) =
      fmt: ^ftt$encoded_fmt_spec,
    = ORD (ftc$data), ORD (ftc$assigned_executable_label) =
      cell_ptr: ^cell,
    casend,
  recend;
```

2.2 FRTL_GLOBAL

```
?? SKIP := 2 ??
?? PUSH (LIST := ON) ??
{ FTTGLIO - global IO types }
?? POP ??

TYPE
  ftt$record_length = amt$max_record_length,
  ftt$unit_name = string (7),
  ftt$return_address = ^cell,
  ftt$segment_offset = 0 .. 0fffffff(16),
  ftt$print_limit = 0 .. 9999999999;
```

3.0 GENERAL_I/O_TYPE_DEFINITIONS

Interfaces in this section are generated by the source language compiler. The interface to the input/output library will have the form of a CYBIL call.

3.1 CONTROL_INFORMATION_LIST_DEFINITION

All control information list specifiers must appear in the first (initialization) call to the library as each input/output statement is processed. The order is completely arbitrery. On a restart I/O call, the cilist consists of a single entry containing a end_cilist.

3.1.1 CILIST (EXTERNAL FILES AND INTERNAL FILES)

3.1.1.1 External_Unit_Entry

An external unit identifier is indicated in the cilist by either an external unit entry (ftc$ext_unit) or a verified unit name entry (ftc$verified_unit_name). The second form is described in the next paragraph. The first form, described here, is used when the unit is expressed as a variable expression (e.g., 'UNIT = N'). The entry points to an integer, which can be viewed as STRING (8) using the tagless variant. The library will convert the integer using flp$convert_unit_spec, which is described in the next paragraph. A unit cilist entry is required on sequential formatted and unformatted I/O statements, direct access formatted and unformatted I/O statements, list directed, namelist, buffer I/O, OPEN, CLOSE, REWIND, ENDFILE, and BACKSPACE I/O statements. For INQUIRE it is present if it is an INQUIRE by UNIT.

3.1.1.2 Verified_Unit_Name_Entry

This form of expressing an external unit identifier [see the above paragraph for context] is used when the unit is given as a constant expression or default (e.g., 'UNIT = 1 + 4', 'UNIT = L"TTY"', 'UNIT = *'). The entry points to a string of length 7, which contains a valid unit name (left-justified, upper-case, blank-filled). In order to convert the constant expression to a unit name, the compiler will call library routine flp$convert_unit_spec, passing it the integer (actually ftt$unit_name_or_number) value of the constant expression. flp$convert_unit_spec will return an indicator of whether the integer is a valid unit identifier. If it is valid, the routine also returns a string of length 7 containing the left-justified, upper-case, blank-filled name of the unit. If the first byte of the integer is null (00(16)), then the integer must be less than 1000 (to be valid) and the string returned will be 'TAPE' concatenated with the (minimal-length) string representation of the integer with trailing blanks if necessary. When the first byte of the integer is not null, then the integer regarded as a string of 8 characters must contain a valid FTN symbol, left-justified and zero-filled in order to be valid. If it is valid, the string returned will be the first seven characters of the FTN symbol converted to upper-case with nulls changed to blanks. The previous paragraph specifies when this cilist entry is required.

```
  ?? SKIP := 2 ??
  ?? PUSH (LIST := ON) ??
  { FLPCUS - flp$convert_unit_spec }
  ?? POP ??

  PROCEDURE [XREF] flp$convert_unit_spec (unit:
    ftt$unit_name_or_number;
    VAR unit_is_invalid: boolean,
    converted_unit_name: ftt$unit_name);
```

3.1.1.3 Internal_File_Entry

This type of entry is used to specify the user character entity that is the object of internal I/O. Its form is similar to an iolist character array entry. This cilist entry is required on internal I/O calls.

3.1.1.4 Extended_Internal_File_Entry

This entry is used for ENCODE/DECODE statements only. It contains a pointer to a record describing the extended internal file. The record contains a pointer to the integer value of the record length specified in the statement, a pointer to the integer value of the number of words in the file, and a pointer to the first byte of the file. If the size of the file is unknown at run-time, then ftc$max_single_word_per_seg should be provided as the size. (Note: Although the file specified in the statement cannot have CHARACTER type, the pointer to the first byte is of type ftt$max_char_vector for FRTL efficiency.)

3.1.1.5 Assigned_Format_Entry

Assigned formats consist of a pointer to ftt$pointer of type assigned-format. The type field in assigned format label will be used to detect cases where the user does not supply the address of an encoded format. This cilist entry must appear on sequential formatted, direct formatted, internal, and extended internal I/O calls if the format was given by an ASSIGNed variable.

3.1.1.6 Boolean_Format_Entry

This type of entry is used for formats stored as Hollerith in non-character arrays. The maximum length of the format is unknown and is located only by finding the terminating right parenthesis. This cilist entry will appear on sequential formatted, direct formatted, internal, and extended internal I/O calls if the format was given by a Boolean item.

3.1.1.7 Character_Format_Entry

This type of entry is used for character formats that were in some sense variable at compile time. The compiler will fully encode any compile time constant format used in a format context (e.g. WRITE(5, '(F10.5)') X). The standard requires the library to accept a format descriptor which is an array name and to allow the format to span array elements. If an array element name is used, the format must not span elements. For character variables, character array elements, and a character array name, the compiler should use an entry like the iolist entry for a character array. The array size will be one for character variables or array element names. It will be the array size for character arrays. The length will be the length of the char item. This cilist entry must appear in sequential formatted, direct formatted, internal, and extended internal I/O calls where the format is given by a non-compile-time encodable character expression.

3.1.1.8 Encoded_Format_Entry

This is a pointer to an encoded description of a format. It is described in a separate section on encoded format specifications.

3.1.1.9 IOSTAT_Entry

This is a pointer to the integer variable or integer array element that is to receive the return status. This cilist entry may appear on all I/O calls accepting a cilist except extended internal and buffer I/O.

3.1.1.10 ERR,_END_Entry

These are addresses of labels in the calling routine where control is to go on an error or end of file condition, respectively. These cilist entries may appear on all I/O calls accepting a cilist except extended internal and buffer I/O.

3.1.1.11 SKIP_Entry

This is an address supplied by the compiler to allow the library to bypass any remaining restart calls for this I/O transmission. It is used in the event of an error or end of file that precludes further data transfer. This cilist entry must appear for all I/O statements.

3.1.1.12 RECL_Entry

This is a pointer to a user integer used to supply the record length on an OPEN or interrogate it on an INQUIRE. This cilist entry must appear in a OPEN or INQUIRE containing a RECL spec.

3.1.1.13 FILE_entry

This is a character data item supplied by the user giving the file name on an OPEN. On INQUIRE, it gives the file name on an INQUIRE by file. This cilist entry must appear in a OPEN or INQUIRE containing a FILE spec.

There are two ways of passing the file name. The FILE name entry (ftc$file) is used when the file name is a variable expression. The verified FILE name entry (ftc$verified_file_name), described in the following paragraph, is used when the file name is a constant expression.

3.1.1.14 Verified_FILE_Name_Entry

This entry is used when the file name is a constant expression. It points to an amt$local_file_name. In order to get the verified file name, the compiler will call library routine flp$convert_file_name with the value of the expression pointed to by a ^string(*). If the expression satisfies the syntax for a local file name (see NOS/VE ERS), parameter name_is_invalid is returned as false, and the value of the expression is returned in parameter coverted_file_name with letters converted to upper-case, and blank-fill on the right. (Note: In order for the name to be valid, any blank characters in the expression must all be on the right). converted_file_name is of type amt$local_file_name, a fixed-length 31 character string. See the previous paragraph for further information on when this entry is required.

```
  ?? PUSH (LIST := ON) ??
  { FLPCFN - flp$convert_file_name }
  ?? POP ??

  PROCEDURE [XREF] flp$convert_file_name (VAR {IN} file_name: string ( * );
    VAR name_is_invalid: boolean;
    VAR converted_file_name: amt$local_file_name);
```

3.1.1.15 NAME_entry

For an INQUIRE by unit, this is a pointer to a user character entity that receives the file name. This cilist entry must appear for an INQUIRE containing a NAME spec.

3.1.1.16 EXIST_entry

This is a pointer to a user logical variable or array element that receives a return value giving the existence status of the file. This cilist entry must appear on an INQUIRE call if the EXIST spec was used.

3.1.1.17 OPENED_entry

This is a pointer to a user logical variable or array element that receives a return value giving the opened status. This cilist entry must appear on an INQUIRE call if the OPENED spec was used.

3.1.1.18 NAMED_entry

This is a pointer to a user logical variable or array element that receives the named status of the file on an INQUIRE. This cilist entry must appear on an INQUIRE call if the NAMED spec was used.

3.1.1.19 NUMBER_entry

This is a pointer to a user integer variable or array element that receives a return value giving the unit number of a file on an INQUIRE. This cilist entry must appear on an INQUIRE call if the NUMBER spec was used.

3.1.1.20 NEXTREC_entry

This is a pointer to a user integer variable or array element that receives a return value giving the next record number on a direct file. This cilist entry must appear on an INQUIRE call if the NEXTREC spec was used.

3.1.1.21 FORM_entry

This is a pointer to a user character entity that receives a return value stating the formatting mode of the file. This entry must appear on a OPEN or INQUIRE call if the FORM spec was used.

3.1.1.22 ACCESS_entry

This is a pointer to a user character entity that receives a return value stating the access mode of the file (sequential or direct). This entry must appear on a OPEN or INQUIRE call if the ACCESS spec was used.

3.1.1.23 SEQUENTIAL_entry

This is a pointer to a user character entity that receives a value stating whether sequential is an allowed access mode. This entry must appear on an INQUIRE call if the SEQUENTIAL spec was used.

3.1.1.24 DIRECT_entry

This is a pointer to a user character entity that receives a value stating whether direct is an allowed access mode. This entry must appear on an INQUIRE call if the DIRECT spec was used.

3.1.1.25 FORMATTED_entry

This is a pointer to a user character entity that receives a value stating whether the file is formatted or not. This entry must appear on an INQUIRE call if the FORMATTED spec was used.

3.1.1.26 UNFORMATTED_entry

This is a pointer to a user character entity that receives a value stating whether the file is unformatted or not. This entry must appear on an INQUIRE call if the UNFORMATTED spec was used.

3.1.1.27 BLANK_entry

This is a pointer to a user character entity that receives a value giving the BLANK= status of the file (ZERO or NULL). This entry must appear on an OPEN or INQUIRE call if the BLANK spec is used.

3.1.1.28 BUFL_entry

This is a pointer to a user integer entity that gives the buffer size to be used for the file. It is ignored by FRTL. It need not appear at all. It may appear only on an OPEN call.

3.1.1.29 BUFMODE_entry

This is a pointer to a user integer entity that gives the mode of the BUFFER I/O operation. This entry must appear with Buffer I/O calls.

3.1.1.30 NAMELIST_entry

This is a pointer to the namelist group table. This entry must appear on NAMELIST I/O calls.

3.1.1.31 BUFIO_FWA_entry

This is a pointer to the first word of the BUFFER I/O transfer region. This entry must appear on BUFFER I/O calls.

3.1.1.32 BUFIO_LWA_entry

This is a pointer to the last word area of the buffer I/O transfer region. It must be incremented by one by the compiler for double word entities. This entry must appear on BUFFER I/O calls.

3.1.1.33 REC_entry

This is a pointer to a user integer entity that supplies the direct access record number to be read or written. This entry must appear on all direct access I/O calls.

3.1.1.34 STATUS_entry

This is a pointer to a user character entity that supplies the type of the file on OPEN or the disposition on CLOSE. This entry must appear on a OPEN or INQUIRE call uses the STATUS spec.

3.1.1.35 ONE_TRIP_DO_entry

This cilist entry specifies that all array iolist elements with negative or zero lengths will transfer one item. Its effect persists for the duration of this I/O statement. By default, the library assumes zero trip DO's. This entry may appear in the initial cilist of any I/O statement which allows implied DOs.

3.1.1.36 Cilist_Term_Entry

This cilist entry is used to terminate the cilist and will always end all cilist's.

3.1.2 CILIST TYPE DESCRIPTION

The cilist description is as follows: (Note that the type declaration ftt$cilist does not imply an array of exactly ftt$cilist_desc_type elements in the order given by the ordinal type ftt$cilist_desc_type. The array may be smaller than the maximum size and the elements can be in any order since they are distinguished by their tag type. This strange notation is needed because CYBIL will not allow use of LOWERBOUND and UPPERBOUND in compile time expressions.)

```
  ?? SKIP := 2 ??
  ?? PUSH (LIST := ON) ??
  { FTTCILS - cilist types }
  ?? POP ??

  TYPE
    ftt$assigned_format = ftt$pointer;

  TYPE
    ftt$unit_name_or_number = record
      case 1 .. 2 of
      = 1 =
        unit_name: string (8),
      = 2 =
        unit_number: integer,
      casend,
    recend;

  TYPE
    ftt$cilist_desc_type = (ftc$ext_unit, ftc$verified_unit_name, ftc$int_file,
      ftc$extd_int_file, ftc$assigned_fmt, ftc$boolean_fmt, ftc$character_fmt,
      ftc$encoded_fmt, ftc$iostat, ftc$err, ftc$end, ftc$skip, ftc$file,
    ftc$verified_file_name, ftc$exist, ftc$access_inquire, ftc$access_op ,
    ftc$blank_inquire, ftc$blank_open, ftc$recl_inquire, ftc$recl_open,
    ftc$form_inquire, ftc$form_open, ftc$bufl, ftc$bufmode, ftc$direct,
    ftc$sequential, ftc$formatted, ftc$unformatted, ftc$name, ftc$named,
    ftc$nextrec, ftc$number, ftc$opened, ftc$namelist, ftc$rec,
    ftc$status_open, ftc$status_close, ftc$bufio_fwa, ftc$bufio_lwa,
    ftc$one_trip_do, ftc$end_cilist);

CONST
  ftc$min_cilist_index = ORD (ftc$ext_unit),
  ftc$max_cilist_index = ORD (ftc$end_cilist),
  ftc$cilist_len = ftc$max_cilist_index - ftc$min_cilist_index + 1;

TYPE
  ftt$cilist = array [ftc$min_cilist_index .. ftc$max_cilist_index] of
    ftt$cilist_desc;

TYPE
  ftt$cilist_desc = record
    case type_: ftt$cilist_desc_type of
    = ftc$ext_unit =
      ext_unit_desc: ^ftt$unit_name_or_number,
    = ftc$verified_unit_name =
      verified_unit_name: ^ftt$unit_name,
    = ftc$int_file =
      int_file_desc: ^ftt$iolist_array_desc,
    = ftc$extd_int_file =
      extd_int_file: ^record
        recl_ptr: ^integer,
        size_ptr: ^integer,
        file_ptr: ^ftt$max_char_vector
      recend,
    = ftc$assigned_fmt =
      assigned_fmt: ^ftt$assigned_format,
    = ftc$boolean_fmt =
      boolean_fmt: ^ftt$max_char_vector,
    = ftc$character_fmt =
      character_fmt: ^ftt$iolist_array_desc,
    = ftc$encoded_fmt =
      encoded_fmt: ^ftt$encoded_fmt_spec,
    = ftc$iostat =
      iostat: ^integer,
    = ftc$err =
      err: ftt$return_address,
    = ftc$end =
      end_: ftt$return_address,
    = ftc$skip =
      skip: ftt$return_address,
    = ftc$recl_open =
      recl_open: ^integer,
    = ftc$recl_inquire =
      recl_inquire: ^integer,
    = ftc$file =
       file: ^^string ( * ),
     = ftc$verified_file_name =
       verified_file_name: ^amt$local_file_name,
     = ftc$name =
       name: ^^string ( * ),
     = ftc$exist =
       exist: ^ftt$logical,
     = ftc$opened =
       opened: ^ftt$logical,
     = ftc$named =
       named: ^ftt$logical,
     = ftc$number =
       number: ^integer,
     = ftc$nextrec =
       nextrec: ^integer,
     = ftc$form_open =
       form_open: ^^string ( * ),
     = ftc$form_inquire =
       form_inquire: ^^string ( * ),
     = ftc$access_open =
       access_open: ^^string ( * ),
     = ftc$access_inquire =
       access_inquire: ^^string ( * ),
     = ftc$sequential =
       sequential: ^^string ( * ),
     = ftc$direct =
       direct: ^^string ( * ),
     = ftc$formatted =
       formatted: ^^string ( * ),
     = ftc$unformatted =
       unformatted: ^^string ( * ),
     = ftc$blank_open =
       blank_open: ^^string ( * ),
     = ftc$blank_inquire =
       blank_inquire: ^^string ( * ),
     = ftc$bufl =
       bufl_desc: ^integer,
     = ftc$bufmode =
       bufmode: ^integer,
     = ftc$namelist =
       namelist: ^ftt$namelist_group_desc,
     = ftc$buflo_fwa =
       buflo_fwa: ^cell,
     = ftc$buflo_lwa =
       buflo_lwa: ^cell,
     = ftc$rec =
       rec: ^integer,
     = ftc$status_open =
       status_open: ^^string ( * ),
     = ftc$status_close =
       status_close: ^^string ( * ),
     = ftc$one_trip_do =
       ,
    = ftc$end_cilist =
      ,
    casend,
  recend;
```

3.1.3 I/O LIST DEFINITION

An iolist is an array of ftt$iolist_desc which is terminated by one of an ftc$end_iolist_restart, ftc$end_iolist_final, or ftc$end_iolist_trailing_comma entry. If the I/O operation is incomplete the iolist should be ended by a ftc$end_iolist_restart entry. If it is complete on this call, the iolist should be ended by a ftc$end_iolist_final entry except when the trailing comma form output list occurs in which case the ftc$end_iolist_trailing_comma entry serves to end the list.

Iolist items are broken into two major types, variables and arrays. The format of a variable entry is in most cases simply a 6 byte PVA pointing to the user variable (first word if double word). In the case of a character variable, the 6 byte PVA points to a standard character string entry which is a PVA pointing to the data followed by two bytes giving the length of the character string.

Array iolist items (short list or collapsed) consist of a PVA which points to an array descriptor entry. This entry contains a PVA pointing to an integer which gives the size of the array in elements. A tagless variant then selects among PVA's that point to the appropriate kind of data (single word, double word, or char). In the case of character, the data PVA actually points to a character string format entry (PVA, length) as in the character variable case.

The arg_variable and arg_array types are used only by NAMELIST I/O, and are defined in the section on NAMELIST I/O statements.

```
  ?? SKIP := 2 ??
  ?? PUSH (LIST := ON) ??
  { FTTIOL - iolist types }
  ?? POP ??

  CONST
    ftc$iolist_len = 800;

  TYPE
    ftt$iolist = array [1 .. ftc$iolist_len] of ftt$iolist_desc;

  TYPE
    ftt$iolist_data_type = (ftc$boolean_variable,
      ftc$logical_variable, ftc$integer_variable, ftc$real_variable,
      ftc$double_variable, ftc$complex_variable, ftc$char_variable,
      ftc$boolean_arg_variable, ftc$logical_arg_variable,
      ftc$integer_arg_variable, ftc$real_arg_variable,
      ftc$double_arg_variable, ftc$complex_arg_variable,
    ftc$char_arg_variable, ftc$boolean_array, ftc$logical_array,
    ftc$integer_array, ftc$real_array, ftc$double_array,
    ftc$complex_array, ftc$char_array, ftc$boolean_arg_array,
    ftc$logical_arg_array, ftc$integer_arg_array,
    ftc$real_arg_array, ftc$double_arg_array, ftc$complex_arg_array,
    ftc$char_arg_array, ftc$end_iolist_restart,
    ftc$end_iolist_trailing_comma, ftc$end_iolist_final);

TYPE
  ftt$single_word_variable_type = ftc$boolean_variable ..
    ftc$real_variable,
  ftt$double_variable_type = ftc$double_variable ..
    ftc$complex_variable,
  ftt$variable_type = ftc$boolean_variable .. ftc$char_variable,
  ftt$array_type = ftc$boolean_array .. ftc$char_array,
  ftt$single_word_array_type = ftc$boolean_array .. ftc$real_array,
  ftt$double_array_type = ftc$double_array .. ftc$complex_array,
  ftt$namelist_variable_type = ftc$boolean_variable ..
    ftc$char_arg_variable,
  ftt$namelist_array_type = ftc$boolean_array .. ftc$char_arg_array;

TYPE
  ftt$iolist_desc = record
    case data_type: ftt$iolist_data_type of
      {=LOWERVALUE(ftt$variable_type) ..
      {UPPERVALUE(ftt$variable_type)=}
    = ftc$boolean_variable .. ftc$char_variable =
      variable_ptr: ^ftt$iolist_variable,
      {= LOWERVALUE(ftt$array_type) .. UPPERVALUE(ftt$array_type) =}
    = ftc$boolean_array .. ftc$char_array =
      array_desc_ptr: ^ftt$iolist_array_desc,
    = ftc$end_iolist_trailing_comma =
      ,
    = ftc$end_iolist_restart =
      ,
    = ftc$end_iolist_final =
      ,
    casend,
  recend;

TYPE

  ftt$iolist_variable = record
    case ftt$variable_type of
      {=LOWERVALUE(ftt$single_word_variable_type) ..
      {UPPERVALUE(ftt$single_word_variable_type)=
    = ftc$boolean_variable .. ftc$real_variable =
      sv: ftt$single_word,
      {=LOWERVALUE(ftt$double_word_variable_type) ..
      {UPPERVALUE(ftt$double_word_variable_type)=
    = ftc$double_variable .. ftc$complex_variable =
      dv: ftt$double_word,
    = ftc$char_variable =
        char_variable_desc: ^string ( * ),
      casend,
    recend,

    ftt$iolist_array_desc = record
      size_ptr: ^integer,
      case ftt$array_type of
        {= LOWERVALUE(ftt$single_word_array_type) ..
        {UPPERVALUE(ftt$single_word_array_type)
      = ftc$boolean_array .. ftc$real_array =
        s: ^ftt$max_single_word_vector,
        {= LOWERVALUE(ftt$double_word_array_type) ..
        {UPPERVALUE(ftt$double_word_array_type)
      = ftc$double_array .. ftc$complex_array =
        d: ^ftt$max_double_word_vector,
      = ftc$char_array =
        c: ^ftt$string_vector_ptr,
      casend,
    recend;
```

3.2 ENCODED_FORMAT_SPECIFICATIONS

The purpose of this section is to describe the encoding of FORTRAN formats. The input/output data editing routines for the FORTRAN CYBER 180 will use an internal representation of the format specification.

3.2.1 METHOD

When the format specification is known to the compiler, it will be encoded from its character form to an internal representation as part of the compilation process. If the format is not available at compilation time (e.g., a run time format) then it will be encoded as part of the cilist processing.

The compiler participation will be to encode the formats to the appropriate internal representation which is to be stored in a compiler-declared area. The address of the compiler-declared area is to be passed to the input/output data editing routines during cilist processing.

3.2.2 ENCODED FORMAT SPECIFICATION TABLE DESCRIPTION

The compiler-declared area will consist of the current FORTRAN format label being processed with the encoded format to follow. If the format label is missing then the format_label must contain blanks.

3.2.2.1 A_edit_descriptor

This edit descriptor has no associated width field. The length derives from the associated character item in the iolist.

3.2.2.2 Aw_edit_descriptor

This edit descriptor allows a width of 65K where the width is explicit in the edit descriptor and overrides the length in the iolist. This form is also used for boolean items, which have no implicit length.

3.2.2.3 BN_or_BZ_edit_descriptor

The BN edit descriptor set blanks to be treated as nulls in numeric fields. The BZ edit descriptor sets blanks to be treated as zeroes in numeric fields.

3.2.2.4 Dw.d_edit_descriptor

This edit descriptor gives the field width (w) and the decimal position (d) for double precision data transfer.

3.2.2.5 Ew.d_edit_descriptor

This edit descriptor gives the field width (w) and the decimal position (d) for transfer of the floating types (real, double, complex).

3.2.2.6 Ew.dEe_edit_descriptor

This edit descriptor is the same as Ew.d except that it is always immediately followed by an extended e edit descriptor.

3.2.2.7 Extended_e_edit_descriptor

This is an artifical edit descriptor introduced to reduce the maximum number of bytes needed to encode all edit descriptors. It carries the exponent size for Ew.dEe and Gw.dEe forms. The format encoder will take the exponent width part and generate an extended e code that follows a normal Ew.dEe or Gw.dEe code in the encoded stream.

3.2.2.8 Ew.d_edit_descriptor

This edit descriptor gives the field width (w) and the decimal position (d) for transfer of the floating types (real, double, complex).

3.2.2.9 Gw.d_edit_descriptor

This edit descriptor gives the field width (w) and the decimal position (d) for transfer of the floating types (real, double, complex).

3.2.2.10 Gw.dEe_edit_descriptor

This edit descriptor is the same as Gw.d except that it is always immediately followed by an extended e edit descriptor.

3.2.2.11 Iw_edit_descriptor

This edit descriptor allows a w field of 255 and converts integer forms. The d part should be set to zero by the format encoder.

3.2.2.12 Iw.m_edit_descriptor

This edit descriptor allows a w field of 255 and converts integer forms with m specifying the number of leading zeroes desired (also limited to 255).

3.2.2.13 Lw_edit_descriptor

This edit descriptor handles up to 255 wide logical data transfer. The d field should be set to zero by the format encoder.

3.2.2.14 Ow_edit_descriptor

This edit descriptor handles up to 255 wide octal data transfer. The d field should be set to zero by the format encoder.

3.2.2.15 Ow.m_edit_descriptor

This edit descriptor allows a w field of 255 and converts octal forms with m specifying the number of leading zeroes desired (also limited to 255).

3.2.2.16 Rw_edit_descriptor

This edit descriptor handles up to 255 wide right adjusted boolean data. The d field should be set to zero by the format encoder.

3.2.2.17 Tc_edit_descriptor

This edit descriptor specifies the column to be tabbed to (1..65K).

3.2.2.18 TLc_and_TRc_edit_descriptor

These edit descriptors specify the relative distance to be tabbed (1..65K) from the current position in a left or right direction, respectively.

3.2.2.19 Zw_edit_descriptor

This edit descriptor handles up to 255 wide hexadecimal conversion. The d field should be set to zero by the format encoder.

3.2.2.20 Zw.m_edit_descriptor

This edit descriptor allows a w field of 255 and converts hex forms with m specifying the number of leading zeroes desired (also limited to 255).

3.2.2.21 nH_edit_descriptor

This edit descriptor supplies the number of characters in the nH string (n). Successive groups of three bytes after this encoded entry up to enough to accommodate the n specified contain the remainder of the byte string. Note that this depends on the use of tagless variants. This form also represents quote delimited strings which must be converted to the counted form by the encoder. n is restricted to 65K.

3.2.2.22 +nP,_-nP_edit_descriptor

These edit descriptors supply the + or - scale factor. Two separate edit descriptors are used for efficiency in the run time format package.

3.2.2.23 nX_edit_descriptor

This edit descriptor gives a count of the number of positions to skip over (to the right). This presumes the output area is preblanked.

3.2.2.24 Repeat_Count_edit_descriptor

This is an artificial edit descriptor created to carry the repeat count preceding a repeatable edit descriptor. It is carried separately for space efficiency reasons and to allow a repeat count up to 65K.

3.2.2.25 Beginning_of_Group_edit_descriptor

This edit descriptor corresponds to a left parenthesis in the format spec. The first token in the encoded format must be one of these. Any nested left parens are encoded as a beginning of group and the preceding repeat count (if any) is supplied via this edit descriptor. The encoder must supply a repeat count of one if there is no explicit repeat count.

3.2.2.26 End_of_Group_edit_descriptor

This edit descriptor corresponds to a right parenthesis in the format spec. The last token in the encoded format must be one of these. Any nested right parens are encoded as an end of group and contain the subscript index of the matching beginning of group. The last token in the encoded format will contain the backward index of the beginning of group of the last preceding nested paren group or the first beginning of group if there are no nested groups.

3.2.2.27 /_edit_descriptor

This edit descriptor specifies the start of a new record. It is not a repeatable edit descriptor. The repeat count is provided for reduction of cases such as (I10////I10) by the format encoder.

3.2.2.28 Colon_edit_descriptor

This edit descriptor specifies termination of end of format scan when the iolist is exhausted.

3.2.2.29 S_edit_descriptor

This edit descriptor selects default processor sign control.

3.2.2.30 SP_edit_descriptor

This edit descriptor selects forced plus sign output.

3.2.2.31 SS_edit_descriptor

This edit descriptor selects suppressed plus sign output.

```
  ?? SKIP := 2 ??
  ?? PUSH (LIST := ON) ??
  { FTTFMAT - format types }
  ?? POP ??

  CONST
    ftc$max_count = 0ffff(16),
    ftc$max_scale = 0ffff(16),
    ftc$max_group_repeat = ftc$max_count,
    ftc$max_encoded_fmt_len = 0ffff(16), {65K tokens is max encoded
    {format size
    ftc$max_tab = ftc$max_count,

    ftc$max_repeat_count = 0ffff(16),
    ftc$max_w = 0ff(16),
    ftc$max_d = 0ff(16),
    ftc$max_e = 6,
    ftc$max_m = ftc$max_d,
    ftc$max_nh = ftc$max_count,
    ftc$max_long_w = 0ffff(16);

  TYPE
    ftt$backward_index = 0 .. ftc$max_encoded_fmt_len,
    ftt$count = 1 .. ftc$max_count,
    ftt$group_repeat = ftt$count,
    ftt$repeat_count = ftt$count,
    ftt$scale_factor = 0 .. ftc$max_scale,
    ftt$w = 1 .. ftc$max_w, { field width }
    ftt$d = 0 .. ftc$max_d, { decimal width }
    ftt$e = 1 .. ftc$max_e, { exponent width }
    ftt$m = ftt$d,
    ftt$nh = 1 .. ftc$max_nh,
    ftt$long_w = 1 .. ftc$max_long_w,
    ftt$format_number = string (6),
    ftt$tab = ftt$count;
```

```
TYPE
  ftt$edit_type = (ftc$a, ftc$a_w, ftc$bn, ftc$bz, ftc$dw_d,
    ftc$ew_d, ftc$ew_dee, ftc$extd_e, ftc$fw_d, ftc$gw_d,
    ftc$gw_dee, ftc$iw, ftc$iw_m, ftc$lw, ftc$ow, ftc$ow_m, ftc$rw,
    ftc$tc, ftc$tlc, ftc$trc, ftc$zw, ftc$zw_m,
    ftc$nh, ftc$nh_apostrophe, ftc$nh_quote, ftc$plus_p,
    ftc$minus_p, ftc$nx, ftc$bog, ftc$eog, ftc$slash, ftc$colon,
    ftc$s, ftc$sp, ftc$ss, ftc$rc);


TYPE
  ftt$encoded_fmt_spec = record
    format_label: ftt$format_number,
    format: array [1 .. ftc$max_encoded_fmt_len] of
      ftt$encoded_edit_desc,
  recend;

TYPE
  ftt$encoded_edit_desc = record
    case (ftc$notag, ftc$tag) of
    = ftc$notag =
      more_string: string (3), {continuation of nH string}
    = ftc$tag =
      case edit_type: ftt$edit_type of
      = ftc$a_w =
        a_width: ftt$long_w,
      = ftc$dw_d, ftc$ew_d, ftc$ew_dee, ftc$fw_d, ftc$gw_d,
        ftc$gw_dee, ftc$iw, ftc$iw_m, ftc$lw, ftc$ow, ftc$ow_m,
          ftc$rw, ftc$zw, ftc$zw_m =
        width: ftt$w,
        d_part: ftt$d,
      = ftc$extd_e =
        extd_e: ftt$e,
      = ftc$tc, ftc$tlc, ftc$trc, ftc$nx, ftc$rc, ftc$bog,
        ftc$slash, ftc$nh, ftc$nh_apostrophe, ftc$nh_quote =
        count: ftt$count,
      = ftc$plus_p, ftc$minus_p =
        scale_factor: ftt$scale_factor,
      = ftc$eog = { termination of current flist }
        backward_index: ftt$backward_index,
      = ftc$a, ftc$bn, ftc$bz, ftc$colon, ftc$s, ftc$sp, ftc$ss =
        ,
      casend,
    casend,
  recend;
```

3.2.3 EXAMPLES OF ENCODED FORMAT SPECIFICATIONS

To be supplied only when encoding method firm.

3.3 SEQUENTIAL_FORMATTED_I/O_STATEMENTS

```
  ?? SKIP := 2 ??
  ?? PUSH (LIST := ON) ??
  { FLPFSIN -- Process sequential access formatted input statement }
  ?? POP ??

  PROCEDURE [XREF] flp$seq_acc_fmtd_in (VAR cilist: ftt$cilist,
    iolist: ftt$iolist);

  ?? SKIP := 2 ??
  ?? PUSH (LIST := ON) ??
  { FLPFSOU -- Process sequential access formatted output statement }
  ?? POP ??

  PROCEDURE [XREF] flp$seq_acc_fmtd_out (VAR cilist: ftt$cilist,
    iolist: ftt$iolist);
```

3.4 DIRECT_ACCESS_FORMATTED_I/O_STATEMENTS

```
  { FLPFDIN }

  PROCEDURE [XREF] flp$dir_acc_fmtd_in (VAR cilist: ftt$cilist,
    iolist: ftt$iolist);

  { FLPFDOU }


  PROCEDURE [XREF] flp$dir_acc_fmtd_out (VAR cilist: ftt$cilist,
    iolist: ftt$iolist);
```

3.5 SEQUENTIAL_UNFORMATTED_I/O_STATEMENTS

```
  { FLPBINP }

  PROCEDURE [XREF] flp$seq_acc_unfmtd_in (VAR cilist: ftt$cilist,
    iolist: ftt$iolist);

  { FLPBOUT }

  PROCEDURE [XREF] flp$seq_acc_unfmtd_out (VAR cilist: ftt$cilist,
    iolist: ftt$iolist);
```

3.6 DIRECT_ACCESS_UNFORMATTED_I/O_STATEMENTS

```
  { FLPUDIN }

  PROCEDURE [XREF] flp$dir_acc_unfmtd_in (VAR cilist: ftt$cilist,
    iolist: ftt$iolist);

  { FLPUDOU }
  PROCEDURE [XREF] flp$dir_acc_unfmtd_out (VAR cilist: ftt$cilist,
    iolist: ftt$iolist);
```

3.7 LIST_DIRECTED_I/O_STATEMENTS

```
  ?? SKIP := 2 ??
  ?? PUSH (LIST := ON) ??
  { FLPLDIN -- Process list directed input statement }
  ?? POP ??

  PROCEDURE [XREF] flp$seq_acc_lst_in (VAR cilist: ftt$cilist,
    iolist: ftt$iolist);

  ?? SKIP := 2 ??
  ?? PUSH (LIST := ON) ??
  { FLPLDOU -- Process list directed output statement }
  ?? POP ??

  PROCEDURE [XREF] flp$seq_acc_lst_out (VAR cilist: ftt$cilist,
    iolist: ftt$iolist);
```

3.8 INTERNAL_I/O_STATEMENTS

```
  { FLPIFIN }

  PROCEDURE [XREF] flp$int_file_fmtd_in (VAR cilist: ftt$cilist,
    iolist: ftt$iolist);

  { FLPIFOU }

  PROCEDURE [XREF] flp$int_file_fmtd_out (VAR cilist: ftt$cilist,
    iolist: ftt$iolist);
```

3.9 ENCODE/DECODE_STATEMENTS

These routines will be used to edit data in moving between extended internal files. The interface is shown following:

```
  { FLPENCO }

  PROCEDURE [XREF] flp$encode (VAR cilist: ftt$cilist,
    iolist: ftt$iolist);


  { FLPDECO }

  PROCEDURE [XREF] flp$decode (VAR cilist: ftt$cilist,
    iolist: ftt$iolist);
```

3.10 NAMELIST_I/O_STATEMENTS

If an item of a group is an array then the dimension table must be accessed to determine boundary conditions. Its format is discussed under array bounds checking. Each item in a NAMELIST group is represented as follows:

```
?? SKIP := 2 ??
?? PUSH (LIST := ON) ??
{ FTTNAMT - types for NAMELIST }
?? POP ??

TYPE
  ftt$namelist_group_desc = record
    name: ftt$symbolic_name,
    arg_list_ptr: ^ftt$arg_list,
    items: array [1 .. ftc$max_num_namelist_group_item] of
      ftt$group_item_desc,
      {items: array [1..osc$max_segment_offset DIV #SIZE
      {(ftt$group_item_desc)] of ftt$group_item_desc; CYBIL limitation
  recend,

  ftt$namelist_array_desc = record
    dim_info_table_ptr: ^ftt$dimension_info_table,
    case ftt$namelist_array_type of
      {= lowervalue (ftt$single_word_array_type) .. uppervalue
      {(ftt$single_word_array_type) =}
    = ftc$boolean_array .. ftc$real_array =
      s: ^ftt$max_single_word_vector,
      {= lowervalue (ftt$double_word_array_type) .. uppervalue
      {(ftt$double_word_array_type) =}
    = ftc$double_array .. ftc$complex_array =
      d: ^ftt$max_double_word_vector,
    = ftc$char_array =
      c: ^ftt$string_vector_ptr,
    = ftc$boolean_arg_array .. ftc$char_arg_array =
      arg_ordinal: ftt$arg_ordinal
    casend
  recend,

  ftt$group_item_desc = record
    name: ftt$symbolic_name,
    case type_: ftt$iolist_data_type of
      {= lowervalue (ftt$variable_type) .. uppervalue (ftt$variable_type= }
    = ftc$boolean_variable .. ftc$char_variable =
      variable_ptr: ^ftt$iolist_variable,
      {= lowervalue (ftt$array_type) .. uppervalue (ftt$array_type) =}
    = ftc$boolean_array .. ftc$char_array =
      array_desc_ptr: ^ftt$namelist_array_desc,
    = ftc$boolean_arg_array .. ftc$char_arg_array =
      arg_array_desc_ptr: ^ftt$namelist_array_desc,
    = ftc$boolean_arg_variable .. ftc$char_arg_variable =
      arg_variable: ftt$arg_ordinal,
    = ftc$end_iolist_final =
      ,
    casend,
  recend;
  { FLPNAMI }

  PROCEDURE [XREF] flp$seq_acc_name_in (VAR cilist: ftt$cilist);


  { FLPNAMO }

  PROCEDURE [XREF] flp$seq_acc_name_out (VAR cilist: ftt$cilist);
```

3.11 BUFFER_I/O_STATEMENTS

The compiler must add 8 bytes to the lwa for any double word type so that the library does not need to know the type. The interface for buffer input/output is as follows:

```
  { FLPBUFI }

  PROCEDURE [XREF] flp$bufferin (VAR cilist: ftt$cilist);

  { FLPBUFO }

  PROCEDURE [XREF] flp$bufferout (VAR cilist: ftt$cilist);
```

3.11.1 UNIT, EOF, LENGTH, LENGTHX, IOCHEC

```
  ?? SKIP := 2 ??
  ?? PUSH (LIST := ON) ??
  { UNIT }
  ?? POP ??

  FUNCTION [XREF] unit (VAR {IN} unit: ftt$unit_name_or_number): real;

  { FLPEOF }

  FUNCTION [XREF] eof (VAR {IN} unit: ftt$unit_name_or_number): real;

  { FLPLNG }

  FUNCTION [XREF] length (VAR {IN} unit: ftt$unit_name_or_number):
    integer;

  { FLPLNGX }

  PROCEDURE [XREF] lengthx (VAR {IN} unit: ftt$unit_name_or_number;
    VAR {OUT} number_of_words_read,
        num_unused_bits: integer);

  { FLPIOCK }

  FUNCTION [XREF] iochec (VAR {IN} unit: ftt$unit_name_or_number):
```

```
  integer;
```

3.12 OPEN

A number of auxiliary IO types and their associated ordinals are used in conjunction with the OPEN, CLOSE, and INQUIRE processing. The interface to the auxiliary routines is shown following:

```
{ FLPOPEN }

PROCEDURE [XREF] flp$open (VAR cilist: ftt$cilist);
```

3.13 CLOSE

```
{ FLPCLOS }

PROCEDURE [XREF] flp$close (VAR cilist: ftt$cilist);
```

3.14 INQUIRE

```
{ FLPINQU }

PROCEDURE [XREF] flp$inquire (VAR cilist: ftt$cilist);
```

3.15 REWIND

```
?? SKIP := 2 ??
?? PUSH (LIST := ON) ??
{ FLPREWD -- Process REWIND statement }
?? POP ??

PROCEDURE [XREF] flp$rewind (VAR cilist: ftt$cilist);
```

3.16 ENDFILE

```
{ FLPENDF }

PROCEDURE [XREF] flp$endfile (VAR cilist: ftt$cilist);
```

3.17 BACKSPACE

```
{ FLPBKSP }

PROCEDURE [XREF] flp$backspace (VAR cilist: ftt$cilist);
```

3.18 MASS STORAGE I/O SUBROUTINES

3.18.1 OPENMS

```
{ FLPOPMS }

PROCEDURE [XREF] openms (VAR {IN} unit: ftt$unit_name_or_number;
  VAR {IN} master_index_name: ftt$max_single_word_vector;
  VAR {IN} master_index_len: integer;
  VAR {IN} index_type: integer);
```

3.18.2 READMS

```
  { FLPRDMS }

  PROCEDURE [XREF] readms (VAR {IN} unit: ftt$unit_name_or_number;
    VAR {OUT} fwa: ftt$max_single_word_vector;
    VAR {IN} xfer_cnt,
        recd_key: integer);
```

3.18.3 WRITMS

Note: WRITMS has a variable number of arguments; so the following call is really not sufficient. It will require an assembler level intercept routine.

```
  { FLPWRMS }

  PROCEDURE [XREF] writms (VAR {IN} unit: ftt$unit_name_or_number;
    VAR {IN} fwa: ftt$max_single_word_vector;
    VAR {IN} xfer_cnt,
        recd_key,
        rewrite_cond,
        subindex_type: integer);
```

3.18.4 CLOSMS

```
  { FLPCLMS }

  PROCEDURE [XREF] closms (VAR {IN} unit: ftt$unit_name_or_number);
```

3.18.5 STINDX

```
  ?? SKIP := 2 ??
  ?? PUSH (LIST := ON) ??
  { FLPSTDX -- STINDX }
  ?? POP ??

  PROCEDURE [XREF] stindx (VAR {IN} unit: ftt$unit_name_or_number;
    VAR {IN} subindex: ftt$max_single_word_vector;
    VAR {IN} subindex_len,
        subindex_type: integer);
```

3.19 LABEL_SUBROUTINE

```
  { FLPLABL }

  PROCEDURE [XREF] label (VAR {IN} unit: ftt$unit_name_or_number,
        lab_info: string ( * ),
        num_elements: integer);
```

3.20 CONNEC/DISCON

```
  { FLPCONC }

  PROCEDURE [XREF] flp$connec (VAR {IN} unit: ftt$unit_name_or_number,
        char_set: integer);
  { FLPDISC }
  PROCEDURE [XREF] discon (VAR {IN} unit: ftt$unit_name_or_number);
```

3.21 LIMERR/NUMERR

```
  { FLPLMER }

  PROCEDURE [XREF] limerr (VAR {IN} errlim: integer);
  { FLPNMER }

  FUNCTION [XREF] numerr: integer;
```

3.22 EINBIN

This is a dummy routine that just returns.

4.0 RUNTIME_EXECUTIVE

4.1 INITIALIZATION

The parameters passed to the main program by the operating system, the compiler version, the default print limit, and the information on the PROGRAM statement are passed to FRTL by a call to flp$init.

Note that the library requires the order of unit names and equivalenced names to be the same as the order of appearance on the PROGRAM statement.

```
  ?? SKIP := 2 ??
  ?? PUSH (LIST := ON) ??
  { FTTINIT - types for initialization handling }
  ?? POP ??

  CONST
    ftc$max_program_stmt_units = 53;

  TYPE
    ftt$program_stmt_unit_desc = record
      left_unit_name: ftt$unit_name,
      case program_type: (ftc$no_alias_or_recl, ftc$alias_only,
        ftc$recl_only, ftc$units_term) of
      = ftc$alias_only =
        right_unit_name: ftt$unit_name,
      = ftc$recl_only =
        recl: ftt$record_length,
      = ftc$units_term, ftc$no_alias_or_recl =
        ,
      casend
    recend,
    ftt$program_stmt_units = array [1 .. ftc$max_program_stmt_units]
      of ftt$program_stmt_unit_desc;
```

The aplist initialization information is as follows :

```
 ??  SKIP := 2 ??
 ??  PUSH (LIST := ON) ??
 { FLPINIT - initialization procedure }
 ??  POP ??

 PROCEDURE [XREF] flp$init (VAR param_list: clt$parameter_list,
   status: ost$status,
   version: ftt$version,
   print_limit: ftt$print_limit,
   program_units: ftt$program_stmt_units,
   user_params: ^string( * ),
   program_name: ftt$symbolic_name,
   continuation_point: ^procedure);
```

4.2 PARAMETER_CRACKING

4.3 SYSTEM

```
  { FLPSYS }

  PROCEDURE [XREF] system (VAR {IN} errnum: integer,
        errmsg: string ( * ));
```

4.4 SYSTEMC

The calling sequence to SYSTEMC must be changed because LOCF does not exist on the 180. A third argument, which is the external of a recovery routine (if supplied), is added. The fifth word of the second argument array is still used to indicate whether the third argument actually is to be processed as the address of an external recovery routine (i.e., if negative the third argument is ignored.).

```
  ?? SKIP := 2 ??
  ?? PUSH (LIST := ON) ??
  { FLTSYSC -- definition of argument vector to SYSTEMC }
  ?? POP ??

  TYPE
    ftt$sysc_arg = record
      fatal_nonfatal: integer,
      frequency: integer,
      freq_increment: integer,
      print_limit: integer,
      recovery_routine_select: integer,
      max_traceback_limit: integer,
    recend;
  ?? SKIP := 2 ??
  ?? PUSH (LIST := ON) ??
  { FLPSYSC -- user callable SYSTEMC routine }
  ?? POP ??

  PROCEDURE [XREF] systemc (VAR {IN} errnum: integer,
        speclist: ftt$sysc_arg,
        recovery_routine: cell);
```

4.5 TERMINATION_PROCESSING

flp$stop and flp$end will not return to the caller. They will issue a log message of END prgm-name or STOP "text", respectively. EXIT is retained for compatibility with old programs. It is not the preferred means for terminating a program

```
  ?? PUSH (LIST := ON) ??
  { FLPSTOP - process STOP statement }
  ?? POP ??

  PROCEDURE [XREF] flp$stop (msg: string ( * ));
  ?? SKIP := 2 ??
  ?? PUSH (LIST := ON) ??
```

```
{ FLPEND - process end statement }
?? POP ??

PROCEDURE [XREF] flp$end (msg: string ( * ));
?? PUSH (LIST := ON) ??
{ FLPEXIT - CYBIL-callable EXIT subroutine }
?? POP ??

PROCEDURE [XREF] flp$exit;
```

5.0 COMPILER_SUPPORT_ROUTINES

5.1 MOVE_CHARACTERS

This routine provides a compiler service routine to move one or more character entities (including substring references) to a character destination with the appropriate truncation or blank filling. Substring references will be evaluated by the library routine. This version will do no substring bounds checking.

```
  ?? SKIP := 2 ??
  ?? PUSH (LIST := ON) ??
  { FLPMOVC -- Move (concatenated) character string }
  ?? POP ??

  FUNCTION [XREF] flp$move_characters (VAR {READ} destination:
    ftt$string_desc,
    source: array [1 .. * ] OF ftt$string_desc): integer;
```

This alternate version of the routine will do the same thing but also validate all substring references including the destination if it is a substring.

```
  ?? SKIP := 2 ??
  ?? PUSH (LIST := ON) ??
  { FLPMVCK -- Move (concatenated) character string and check validity
  {}
  ?? POP ??

  FUNCTION [XREF] flp$move_characters_check (VAR {READ} destination:
    ftt$string_desc,
    source: array [1 .. * ] OF ftt$string_desc): integer;
```

5.2 COPY_ARGUMENT_LIST

The compiler will call flp$copy_arg_list when it is necessary to generate a "union of argument lists" for the multiple entry point routine. flp$copy_arg_list will copy the old_arg_list parameters to the global_arg_list entry indicated by the control list entry parallel to the current old_arg_list entry. If the control list specifies that the item is fixed length character, then the character length part of the global_arg_list entry will be taken from the control list.

```
  { FTTCAL }

  TYPE
    ftt$cal_control_list = array [1 .. * ] of
      ftt$cal_control_list_entry,

    ftt$cal_control_list_entry = record
      arg_ordinal: ftt$arg_ordinal,
      case is_fixed_char: boolean of
    = TRUE =
      fixed_length: ftt$char_element_size,
    casend,
  recend;
```

```
{ FLPCAL }

PROCEDURE [XREF] flp$copy_arg_list (VAR old_arg_list: ftt$arg_list,
  global_arg_list: ftt$arg_list,
  control_list: ftt$cal_control_list);
```

5.3 CHARACTER_COMPARE_UNCOLLATED

This function will compare one or more left side character operands (due to concatenation) with one or more right side character operands (including logically extending the shorter with blanks to the length of the longer) and return the comparison result in the same form as the hardware character compare instructions result.

```
{ FLPCU }

FUNCTION [XREF] flp$compare_fixed (left_string: array [1 .. * ] OF
  ftt$string_desc,
  right_string: array [1 .. * ] OF ftt$string_desc):
    ftt$compare_result;
```

This alternate version does exactly the same thing as the compare uncollated version but with the addition of validation on any substring references occuring in the argument list.

```
{ FLPCUCK }

FUNCTION [XREF] flp$compare_fixed_check (left_string: array [1 .. *
  ] OF ftt$string_desc,
  right_string: array [1 .. * ] OF ftt$string_desc):
    ftt$compare_result;
```

5.4 CHARACTER_COMPARE_COLLATED

This routine does the same function with the same argument types as compare uncollated but uses a previously specified collation table to do the compare.

```
{ FLPCC }

FUNCTION [XREF] flp$compare_user (left_string: array [1 .. * ] OF
  ftt$string_desc,
  right_string: array [1 .. * ] OF ftt$string_desc):
    ftt$compare_result;
```

This alternate version does the same thing as compare collated but also validates any substring references that occur in the argument lists.

```
  { FLPCCCK }

  FUNCTION [XREF] flp$compare_user_check (left_string: array [1 .. * ]
    OF ftt$string_desc,
    right_string: array [1 .. * ] OF ftt$string_desc):
      ftt$compare_result;
```

5.5 FORM_SUBSTRING_PRIMARY

This function forms a substring arg_list pointer and returns it to the compiler to be inserted in a dummy argument list.

```
  ?? SKIP := 2 ??
  ?? PUSH (LIST := ON) ??
  { FLPSUBS -- Convert ftt$string_desc to ^string( * ) }
  ?? POP ??

  FUNCTION [XREF] flp$substring (VAR {IN} substring: ftt$string_desc):
    ^ string ( * );
```

This alternate version does the same thing but validates the substring as well.

```
  ?? SKIP := 2 ??
  ?? PUSH (LIST := ON) ??
  { FLPSUBC -- Convert ftt$string_desc to ^string( * ) and check
  {validity }
  ?? POP ??

  FUNCTION [XREF] flp$substring_check (VAR {IN} substring:
    ftt$string_desc): ^string ( * );
```

5.6 ARRAY_BOUNDS_CHECKING

Array bounds checking may be accomplished at run time by obtaining the appropriate information from the dimension_info_table. The function returns subscript value. The arg_list to the library routine to do the checking is as follows:

The format of the dimension table is as follows:

```
  ?? SKIP := 2 ??
  ?? PUSH (LIST := ON) ??
  { FTTDIMT - dimension info table types }
  ?? POP ??

  TYPE
    ftt$dimension_info_table = record
      number_dimensions: ftt$dimension,
    dim_desc: array [ftt$dimension] of ftt$dim_desc,
  recend,

  ftt$dim_desc = record
    lower_bound: ^integer,
    span: ^integer,
  recend;
```

```
?? RIGHT := 70 ??
  { FLPCKAB - Declare flp$check_array_bounds }

  FUNCTION [XREF] flp$check_array_bounds (array_name: string ( * <=
    7);
    VAR [IN} dim_table: ftt$dimension_info_table;
        subscript_exp: array [ftt$dimension] OF ^integer): 1 ..
      ftc$max_array_elements;
```

5.7 PAUSE

```
  { FLPPAUS }

  PROCEDURE [XREF] flp$pause (msg: string ( * ));
```

5.8 ASSIGNED_GO_TO_ERROR

The compiler will invoke this routine if an assigned GO TO attempts a transfer of control to a nonexecutable point.

```
  { FLPAGTO }

  PROCEDURE [XREF] flp$assigned_go_to_error;
```

6.0 INTRINSIC_FUNCTIONS

6.1 LEN

```
{ FLPLEN }

FUNCTION [XREF] flp$len (VAR {IN} char_arg: string ( * )):
  ftt$char_element_size;
```

6.2 INDEX

```
{ FLPNDX }

FUNCTION [XREF] flp$index (VAR {IN} arg_string,
      sub_string: string ( * )): 0 .. ftc$max_char_len;
```

6.3 SECOND

```
{ FLPSEC }

FUNCTION [XREF] flp$second: real;
```

6.4 SHIFT

```
{ FLPSHFT }

FUNCTION [XREF] flp$shift (VAR {IN} word: ftt$boolean,
      number_of_bits: integer): ftt$boolean;
```

6.5 MASK

```
{ FLPMASK }

FUNCTION [XREF] flp$mask (VAR {IN} number_of_bits: integer):
  ftt$boolean;
```

6.6 CHAR/ICHAR

```
{ FLPCHRU }

FUNCTION [XREF] flp$char_user (int_arg: integer): char;

{ FLPCHRF }

FUNCTION [XREF] flp$char_fixed (int_arg: integer): char;

{ FLPICHU }

FUNCTION [XREF] flp$ichar_user (VAR {IN} char_arg: string ( * )):
  ftt$char_size;

{ FLPICHF }
```

```
FUNCTION [XREF] flp$ichar_fixed (VAR {IN} char_arg: string ( * )):
  ftt$char_size;
```

6.7 LLT, LGT, LLE, LGE

The compiler should use the character compare fixed support routines.

6.8 BOOL

```
?? PUSH (LIST := ON) ??
{ FLPBOOL - BOOL intrinsic }
?? POP ??

FUNCTION [XREF] flp$bool (VAR {IN} char_arg: string ( * )):
  ftt$boolean;
```

7.0 LIBRARY_SUBROUTINES

7.1 COLLATION_ROUTINES

7.1.1 COLSEQ

```
  { FLPCOSQ }

  PROCEDURE [XREF] colseq (VAR {IN} colseq_name: string ( * ));
```

7.1.2 CSOWN

```
  { FLPCSWN }

  PROCEDURE [XREF] csown (VAR {IN} str: string ( * ));
```

7.1.3 WTSET

```
  { FLPWTST }

  PROCEDURE [XREF] wtset (VAR {IN} ind: string ( * ),
        wt_val: integer);
```

7.2 CLOCK

```
  { FLPCLCK }

  PROCEDURE [XREF] clock (VAR {OUT} clock_value: string ( * ));
```

7.3 TIME

```
  { FLPTIME }

  PROCEDURE [XREF] time (VAR {OUT} time_value: string ( * ));
```

7.4 DATE

```
  { FLPDATE }

  PROCEDURE [XREF] date (VAR {OUT} date_value: string ( * ));
```

7.5 JDATE

```
  { FLPJDAT }

  PROCEDURE [XREF] jdate (VAR {OUT} jdate_value: string ( * ));
```

7.6 REMARK

```
  { FLPRMK }

  PROCEDURE [XREF] remark (VAR {IN} msg: string ( * ));
```

7.7 DISPLA

```
  { FLTDPL }

  TYPE
  flt$displa_val = record
    case (displa_int, displa_real, parsed_real) of
    = displa_int =
      k_val: integer,
    = displa_real =
      r_val: real,
    = parsed_real =
      pr_val: ftt$parsed_real,
    casend,
  recend;


{ FLPDPL }

PROCEDURE [XREF] displa (VAR {IN} h: string ( * ),
      k: flt$displa_val);
```

7.8 LEGVAR

```
{ FLPLGVR }

FUNCTION [XREF] legvar (VAR {IN} var_: ftt$parsed_real): integer;
```

7.9 MOVLEV

```
{ FLPMOVL }

PROCEDURE [XREF] movlev (VAR {IN} source:
  ftt$max_single_word_vector;
  VAR {OUT} destination: ftt$max_single_word_vector;
  VAR {IN} num_words: integer);
```

7.10 MOVLCH

```
{ FLPMVCH }

PROCEDURE [XREF] movlch (VAR {IN} source: string ( * );
  VAR {OUT} destination: string ( * );
  VAR {IN} num_bytes: integer);
```

7.11 SSWTCH

```
{ FLPSWCH }

PROCEDJRE [XREF] sswtch (VAR {IN} sswtch_num: integer;
  VAR {OUT} sswtch_val: integer);
```

7.12 STRACE

```
{ FLPSTRC }

PROCEDURE [XREF] strace;
```

7.13 GOTOER

This routine will simply cause a fatal error message for " value outside of computed GO TO bounds" to be issued and execution terminated.

7.14 DUMP/PDUMP

Note: DUMP and PDUMP have a variable number of parameters; so they cannot be implemented in CYBIL. flp$dump and flp$pdump are called by the library to implement them.

```
  { FLTDUMP }

  TYPE
    flt$dump_entry = record
      dump_fwa: ^ftt$max_single_word_vector,
      dump_len: ftt$segment_offset,
      f: integer,
    recend;


  { FLPDUMP }

  PROCEDURE [XREF] flp$dump (dump_list: ^array [1 .. * ] OF
    flt$dump_entry);

  { FLPPDMP }

  PROCEDURE [XREF] flp$pdump (dump_list: ^array [1 .. * ] OF
    flt$dump_entry);
```

7.15 CMM_ROUTINES

7.16 DUMMY_ROUTINES

7.16.1 OVERLAY, CHEKPTX, RECOVR, LOVCAP, XOVCAP, UOVCAP

These routines will be provided but will be dummies that only return.

7.16.2 LTPLOAD, LTPDUMP, MANTRAP, DARRAY

These will not be supplied as part of FRTL. They were provided to FCL4 users for transition from the Leicester version of MANTRAP.

7.16.3 PMDLOAD, PMDDUMP, PMDSTOP, PMDARRY

These routines will be provided as dummy routines that only return.

7.17 SORT/MERGE_INTERFACE_ROUTINES

These will be provided by the SORT project and will be part of the SORT product. They will be FORTRAN callable but will not be supported as part of the FRTL.

Table of Contents