Published: 08/03/67
(Supercedes: BF.20.07, 05/12/67)

Identification

Generation and Usage of the Change List Structure using the I/O Command Translator (IOCT)
C. D. Olmsted

Purpose

The IOCT compiles from a source language input segment a GIOC interface module (GIM) data base containing change structures. The function of this data base is described briefly below and in more detail in BF.20.01, and 02 and 03. An understanding of these MSPM sections and also of the I/O table compiler (IOTC) (BF.20.06) is assumed.

Introduction

The change structures are the source data for the I/O control lists, i.e. the pseudo-lists described in BF.20.01. They specify what changes are to be made in the pseudo-lists. A device control module (DCM), in order to implement a device strategy, will issue a call to the GIM, requesting a change in a pseudo-list and passing a change structure as an argument. The GIM will effect the specified changes within the restrictions given by the class driving table (CDT) for the particular device (logical channel), resulting finally in a new pseudo-list item.

Use

The Multics command "ioct name" will cause the segment _name.ioct_ to be translated into a segment _name_ and a linkage segment _name.link_. The segment _name_ will contain the change structures and the segment _name.link_ will be a standard Multics linkage segment, providing symbolic access to each of the change structures as an entry in _name_.

The IOCT Language

Statements in the language are in free field form separated by a semicolon (;). Elements within statements are delimited by space or NL. There are three kinds of statements:

1) Comment statement. This begins with a right slash (/) as first nondelimiting character and is otherwise arbitrary.

```
      / This is a comment.;
```

2) End statement. This begins with an asterisk (*) as first nondelimiting character and is otherwise arbitrary.

```
      * [ anything ];
```

3) Changes statement. This is actually the only statement relevant to the compilation of the changes structure. It has the general form,

```
      label op_type
          field1 value1 [field2 value2...fieldn valuen];
```

where the elements are specified as follows:

<u>label</u> is a character string which will become the symbolic entry label in <u>name.link</u> for the changes structure. It must therefore have the form of a PL/I identifier.

The remaining elements are decimal or binary arguments. These have exactly the form described in the IOTC language. The facility for using mnemonics dictionary is used in both the IOTC and IOCT so that all of the standard mnemonics (listed in BF.20.06) are available. Thus in place of any decimal or binary argument a mnemonic may be used. Its defined value must conform in mode and size to the restrictions given below.

<u>op type</u> is the operation type. It is a decimal argument between 2 and 6 and specifies the type of I/O control.

<u>fieldi</u> corresponds to the field index of the CDT. It is a decimal argument between 1 and 50. Within a statement no <u>fieldi</u> may be repeated, thus limiting their number to 50.

<u>valuei</u> is a decimal or binary argument. The use to which its value is put depends upon the fld-action of the corresponding field entry in the CDT. If <u>valuei</u> is a binary argument, the number of bits specified must be $\leq$ 24. If more, only the left most 24 are used. If it is desired to leave a <u>valuei</u> "blank" (i.e. zero), the standard mnemonic "null" may be used. It is a decimal zero.

Error Returns

It is the responsibility of the creator of the input segment to see that the change structures are compatible with their corresponding CDTs. If not, error returns will be gotten from the GIM.

The IOCT extensively checks the syntax of the input segment. Errors are transmitted to the user's error file in the standard way with a copy of the ill-formed statement being included as extra information. The error signals are of five types, each of which may be raised for one of several reasons. The types and possible reasons are outlined below along with the action taken for each type. The numbers in parentheses are error codes.

1. Bad _label_. The statement is ignored.

    a. repeated (39)
    b. not in PL/I form (41)

2. Bad _optype_. The statement is ignored.

    a. out of bounds (42)
    b. in binary mode (43)
    c. undefined mnemonic (44)
    d. illegal character sequence (45)
    e. missing (56)
    f. end of statement follows immediately (57)

3. Bad _fieldi_. _fieldi_ and _valuei_ are ignored.

    a. out of bounds (46)
    b. in binary mode (47)
    c. undefined mnemonic (48)
    d. illegal character sequence (49)
    e. repeated (55)

4. Bad _label_ or _valuei_. Default value is taken.

    a. _label_ has > 31 characters. Leftmost 31 are used. (40)
    b. _valuei_ has > 24 bits. Leftmost 24 are used. (50)
    c. _valuei_ is undefined mnemonic. Zero used. (51)
    d. _valuei_ has illegal character sequence. Zero used. (52)
    e. _valuen_ (last one) is missing. Zero used. (59)

5. Boundary problems. An "end" statement is assumed.

   a. no "end" statement. (18)
   b. segment _name_ has become too big. (53)
   c. more than 50 _fieldi_. (58)

Example of Input Segment

The following changes structures are for use with a CDT designed to drive a dataphone.

```
/ Activate the list channel;

go ccw

  activate_list on;

/ Turn the receiver on;

rec_on cdcw

  mode receive

  dataset_ready on;

/ This provides for reception of up to 120 characters.
It searches for and exhausts on an EOT character (octal 004);

input_message ddcw

  data null

  isig sc4

  par on

  utag 1

  tally 120

  match isae

  flow read

  char 4;
```

```
/ Turn the transmitter on;

  tran_on cdcw

  mode transmit;

/ Send mark characters;

  marks ldcw

  exh sc2

  literal "1111111,24

  tally 50;

/ This specifies a transfer.  Where control goes is
determined by the user who fills in the values for list id
and indx;

transfer tdcw

list_id  null

indx     null;

*end;
```