Published: 08/04/67
(Supersedes: BF.20.06, 02/09/67)

Identification

Generation of the Class Driving Tables
using the I/O Table Compiler (IOTC)
C. D. Olmsted

Purpose

Given an appropriate input file (described below) the IOTC creates a segment which contains a Class Driving Table (CDT). The CDT and its use are described in BF.20.01.

Introduction

The IOTC is a program which can be called by issuing the Multics command

```
iotc name
```

which causes the file "name.iotc" to be compiled into a file "name.cdt". The file "name.iotc" is an ascii file which has been created by the user, using the Multics editor. It consists of statements in the IOTC language (described below). The segment created, "name.cdt", is the resulting Class Driving Table.

The IOTC is used only when a new CDT is required, i.e., when a peripheral device requiring new and different control is attached to a GIOC. The DCM for such a device will then be able to access this CDT through calls to the GIM which in turn will discover the CDT within the file system. These procedures are described in more detail in BF.20.01.

The IOTC Language

There are five kinds of statements in the language, each identified by its keyword which must occur as the first element of the statement. These keywords will be written in capitals here for clarity, but for actual input they will be typed in lower case. The statements are in free field form with elements separated by space or NL and with statements separated by a semicolon (;). A statement is made up of a key word followed by various argument elements. Square brackets "[...]" indicate that the presence of the argument is optional.

1) Comment keyword = "/". This has as arguments any character string, e.g.:

```
/ this is a comment for illustration;
```

2) Class Driver Table keyword = "CDT".

```
CDT op_type [type_value];
```

where op_type is a decimal number between 1 and 6 and type_value is a binary argument. The formation of binary arguments is described below.

3) Field keyword = "FLD".

```
FLD fld_no [fld_action [fld_end [fld_mask]]];
```

where fld_no is a decimal number,
fld_action is a decimal number between 0 and 4,
fld_end is a decimal number between 0 and 83,
and fld_mask is a binary argument.

4) Value keyword = "VAL".

```
VAL index [value(index) value(index+1)...value(n)];
```

where index is a decimal number between 0 and n, and the value(i) are decimal numbers or binary arguments.

5) Termination keyword = "*".

Each of keywords 2), 3), and 4) corresponds to the appropriately named level in the CDT structure (see the declaration in BF.20.01 or BF.20.03). Thus the "CDT" keyword will cause the value of the argument "type_value" to be substituted in

```
cdt_ptr->cdt(op_type).type_value.
```

Similarly, the "FLD" keyword causes values corresponding to its arguments to be substituted in the substructure array named "field" with its index having the value of "fld_no". "VAL" key words cause substitution in a like manner into the "value" substructure. The statement "END;" should be the last one in the file and signifies the end of input to the IOTC.

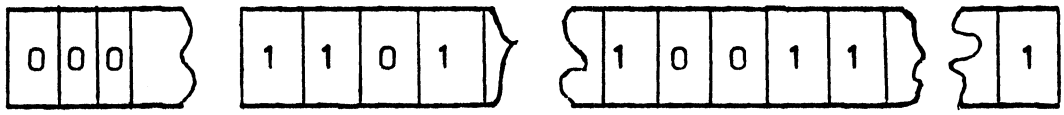
Binary arguments are a representation of bit strings of length 84. These arguments will provide the values to be substituted into the pseudo_DCW's and pseudo_CCW's which are described in BF.20.01. Binary arguments should be no more than 100 characters long. If longer, only the left-most 100 characters are considered. They have the form

"bit_string, position [-bit string, position] ...

where the punctuation marks double quote, comma, and minus sign occur literally, "bit_string" is a string of 0's and 1's representing a bit string, and "position" is a one or two digit decimal number between 0 and 83 which indicates the position of the rightmost bit of "bit_string" in the 84-bit bit-string being represented. The initial double quote identifies the argument as binary, the comma separates the bit string from its position indicator, and the minus sign separates the occurrance of (optional) subsequent bit strings and position indicators. No blanks are allowed within a binary argument. For example, the argument

"1101,12-10011,67-1,83;

will cause the construction of the 84 bit string



where zeros are inserted in the remaining unspecified positions. Binary arguments are processed from left to right so that if a subsequent string overlaps a previous one, the later bit configuration replaces the overlapped part of the earlier one.

Decimal arguments are strings of decimal digits of length ≤ 6. If more than 6, only the left most 6 are considered.

The order in which statements may occur is restricted as follows:

1) An "FLD" keyword may not occur unless a "CDT" keyword appears in some earlier statement. The field quantities entered by means of the "FLD" keyword will be associated with the "op_type" of the nearest preceeding "CDT" keyword.

2) Once a "CDT op_type..." statement has occurred, a second "CDT" statement may not occur with the same value for "op_type".

3) A "VAL" keyword may not occur unless an "FLD" keyword appears in some earlier statement. The value quantities entered by means of the "VAL" keyword will be associated with the "fld_no" of the nearest preceeding "FLD" keyword. The "FLD" statement must have a field action of 1 (masked value substitution).

4) Once a "FLD fld_no..." statement has occurred, a second "FLD" statement may not occur with the same value for "fld_no".

Mnemonics

For convenience in writing input files for the IOTC, the capability is provided for using mnemonics in place of either a decimal or binary argument. These mnemonics are defined by creating a segment named "mnem_dict" which is made available to the IOTC (and also to the IOCT--see BF.20.07). This file is a dictionary containing the mnemonics and their defined values. It is created by issuing the command

```
mnemonics input_file
```

where input_file is an ascii file written by the user in a very simple-minded language in which statements are in free field form, separated by semicolons, and with elements separated by one or more blanks, tabs, or carriage returns. Each statement is of the form

```
mnem value [comment];
```

where "mnem" is an alphanumeric character string of length ≤ 31 (if longer, only the leftmost 31 are considered), the first character of which is alphabetic, "value" is a decimal number or a binary argument, and "comment" is any character string at all (even empty). Such a statement will create an entry in "mnem" and "value". A distinction is made between binary and decimal values so that the IOTC can check the propriety of a mnemonic argument. The last statement in the file should start with a "*" and signifies the end of the mnemonics definitions. Comment statements of the same form as in the IOTC language (starting with "/") may also be included.

Error Returns

The IOTC extensively checks the syntax of the input segment. Errors are transmitted to the users error file in the standard way (BY.11.00) with a copy of the ill-formed statement being included as extra information. Also included is a code the meaning of which is given below.

code	meaning	action taken
1	bad key word	statement ignored
2	bad argument	zero inserted
3	undefined mnemonic	zero inserted
4	bad VAL index	statement ignored
5	excess VAL arguments	excess arguments ignored
6	repeated VAL index	previous values are written over
7	missing VAL index	statement ignored
8	argument in wrong mode	if op-type or fld_no, the statement is ignored. Otherwise zero inserted.
9	fld_action out of bounds	zero inserted
10	fld_end out of bounds	zero inserted
11	missing fld_no	statement ignored
12	invalid fld_no	statement ignored
13	repeated fld_no	statement ignored
14	missing op-type	statement ignored
15	invalid op-type	statement ignored
16	repeated op-type	statement ignored
17	keyword sequence error	statement ignored
18	no end statement	previous statements are lost.

Similarly, error codes are returned from the mnemonic dictionary maker. Their meanings are given below.

code	meaning	action taken
18	no end statement	size of dictionary lost
31	ill formed value	statement ignored
32	too many mnemonics	statement ignored
33	missing value	statement ignored
34	repeated mnemonic	statement ignored
35	ill formed mnemonic	statement ignored

Error 32 means that the size of the dictionary has been exceeded. This size is 40 binary mnemonic and 60 decimal ones. Running out of room for one mode does not prevent mnemonics of the other mode from being entered.

Summary of the Language

Argument types:

1. decimal integer
2. Binary argument of the form

   "bit_string, position [-bit_string, position]...

   where bit_string is zeros and ones and 0 ≤ position ≤ 83.
3. Mnemonics, which are from one to thirty-one characters long with the first character alphabetic.

Statements:

Let the superscripts identify the arguments by type as follows:

- * means binary or binary mnemonic (2. or 3.)
- ^ means decimal or decimal mnemonic (1. or 3.)
- ' means either * or ^ (1., 2. or 3.),

Then permissible statements are

1. / this is any comment;
2. CDT op_type^ [type_value*];
3. FLD fld_no^ [fld_action^ [fld_end^ [fld_mask*]]];
4. VAL index^ [value(index)' value(index+1)' ... value(n)'];
5. *

Argument Limits

1. Binary arguments are limited to 84 specified bits or 100 characters.
2. 1 ≤ op_type ≤ 6

3. $1 \leq$ fld_no ≤ 50
4. $0 \leq$ fld_action ≤ 3
5. $0 \leq$ fld_end ≤ 83
6. $0 \leq$ value(i) $\leq 2^{84} - 1$, if it is decimal
7. $0 \leq$ index ≤ 50

Standard Mnemonics

The mnemonic dictionary source file, "input-file" will include the following statements. These standard mnemonics should not, of course, be redefined.

```
status      1                   op_type mnemonics;
ccw         2;
cdcw        3;
tdcw        4;
ldcw        5;
ddcw        6;

mv          1                   field action mnemonics;
lit         2;
da          3;

off         0                   bit switches;
null        0;
on          1;

term        1                   status word fields;
adapt_err   2;
gioc_err    3;
tr_timing   4;
```

```
exh         2               field definitions;
esig        3;
isig        4;
xes         5;
par         6;
last        7;
utag        8;

term_mask   "111,5          field definition masks;
exh_mask    "111,8;
esig_mask   "111,11;
isig_mask   "111,14;
xes_mask    "1,15;
par_mask    "1,16;
last_mask   "1,17-1,81;
utag_mask   "111111,77;

nosc        0               status channel pointers;
sc1         1;
sc2         2;
sc3         3;
sc4         4;
sc5         5;
sc6         6;
sc7         7;
```

```
list_id          9          transfer DCW fields and masks;
list_id_mask                "111111111111,53;
indx             10;
indx_mask                   "111111111111,65;
literal          9          literal DCW fields;
literal_mask                "1111111111111111111,35;
tally            10;
tally_mask                  "111111111111,65;
data             9          data DCW fields;
pack             11;
pack_mask                   "11,67-111,71;
micro            12;
micro_mask                  "111,2-111,20;
match            13;
match_mask                  "111,2-111,20;
char             14;
count            15;
flow             16;
flow_mask                   "111,80;

pack6_v          "11,67-000,71 values for pack field;
pack9_v          "10,67-000,71;
clear            0          values for microcode DCW field;
clear_v          "0,83;
idle_v           "1,20;
internal                    2;
```

```
internal_v      "1,19;
ctl_char        3;
ctl_char_v      "1,21;
int_ctl         4;
int_ctl_v       "101,21;
nomatch         0               values for control character DCW
                                field;
nomatch_v       "1,2;
is              1;
is_v            "1,2-1,20;
isae            2;
isae_v          "1,2-1,19;
isedt           3;
isedt_v         "1,2-11,20;
misae           4;
misae_v         "1,2-1,18;
cis             5;
cis_v           "1,2-101,20;
cisae           6;
cisae_v         "1,2-110,20;
cisedt          7;
cisedt_v        "1,2-111,20;
read            0               direction of data flow values;
read_v          "1,78;
write           1;
write_v         "1,79;
```

Examples of IOTC Language Source Files

```
/    Example of transfer DCW IOTC definition;
     cdt  tdcw      "11,2;
     fld  utag      lit       77        utag_mask;
     fld  list_id   lit       53        list_id_mask;
     fld  indx      lit       65        indx_mask;
     *;

/    Example of literal DCW definition;
     cdt  ldcw      "101001,5;
     fld  term      lit        5        term_mask;
     fld  exh       lit        8        exh_mask;
     fld  esig      lit       77        esig_mask;
     fld  xes       mv        15        xes_mask;
     val  0         off       on;
     fld  last      mv        81        last_mask;
     val  0        "0,17-0,81 "1,17-1,81;
     fld  utag      lit       77        utag_mask;
     fld  literal   lit       35        literal_mask;
     fld  tally     lit       65        tally_mask;
     *;
```