Published: 08/03/67
(Supercedes: BF.20.08, 05/15/67)

## Identification

Implementation of the I/O Table Compiler (IOTC) and the I/O Command Translator (IOCT)
C. D. Olmsted

## Purpose

This implementation provides the compilers discussed in BF.20.06, and BF.20.07.

## Introduction

The IOTC and IOCT share 7 modules. These modules do the reading of the input files and the scanning and conversion of the language elements. Errors and mnemonics are also handled by the same routines. These modules are, therefore, described first and in close detail.

The remaining modules are more specific to the IOTC and IOCT and are very straightforward. By making standard calls, they create the appropriate data bases and then fill them in with the values gotten from the scanning and conversion routines.

## getfield (see flow chart)

The module which scans the input file and delivers "fields" or elements of the statements is named "getfield". Getfield is called as a function of one argument. It returns a character string. Thus:

```
chars = getfield (nofield);
```

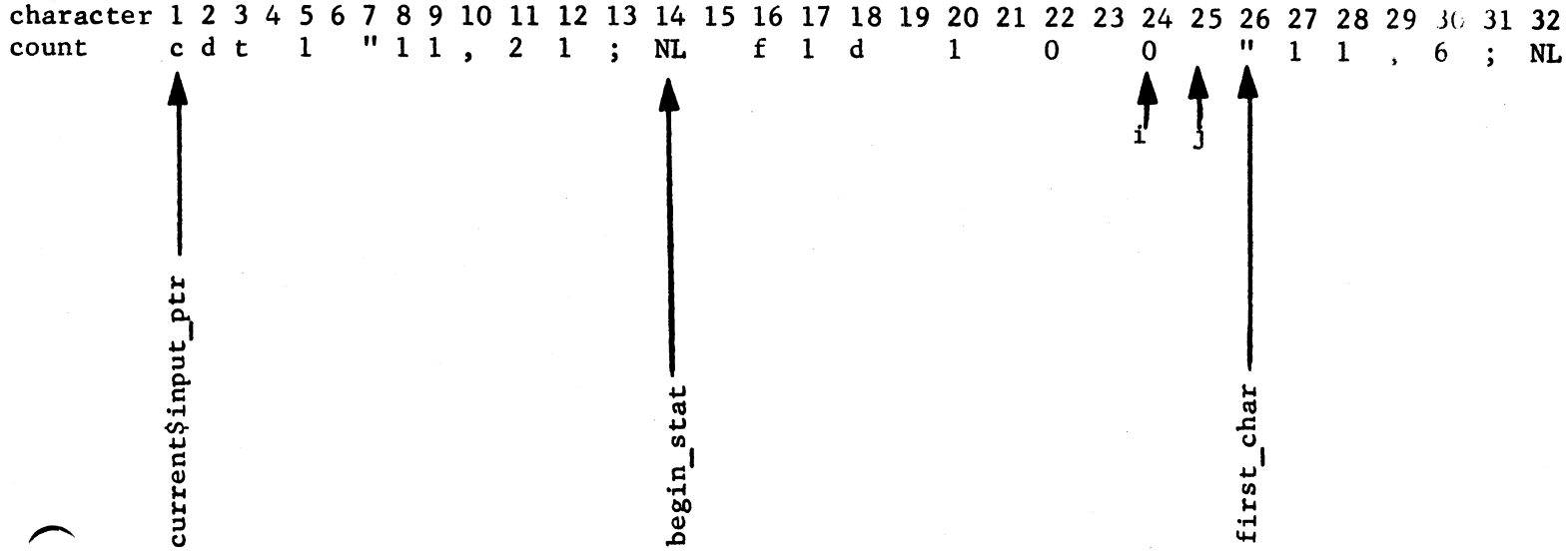
where "chars" is character (100) and "nofield" is a label. There is a return to nofield if there is no field, i.e., if the last call to getfield returned the last element of the statement. The string, chars, will be the next delimited field in the statement, left justified and filled with blanks on the right.

The input file is seen by getfield as a structure with the declaration:

```
dcl 1 input_file based(p),
      2 line char (4095);
```

The pointer to it is "current$input_ptr" where current is an external data base.

The actual picking out of the field is done by maintaining several indices or character pointers. Their functions are diagrammed and described below.

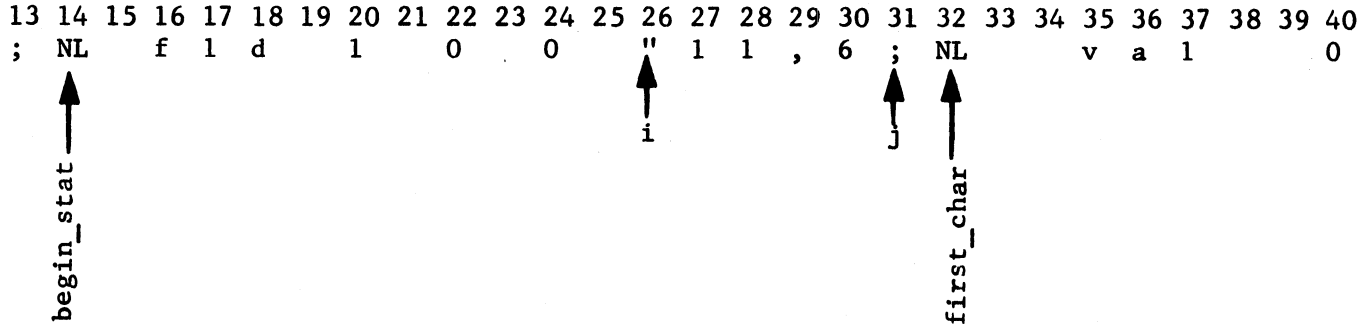



In the above diagram the first character of the statement being scanned is pointed at by begin_stat. The last call to getfield returned the field consisting of the 0 at character 24. The first nondelimiting character of this field is pointed at by i and the first delimiting character after it is pointed at by j. Just before returning, first_char is set to j+1 and points to the character where the scan will begin at the next call. Initially both begin_stat and first_char are 1. The character string that is returned is

substr(current$input_ptr->input_file.line,i,min(j-i,100)).

If there is not a return to the label nofield then begin_stat is set equal to begin_stat_init which is 1 initially. Whenever a semicolon is discovered, begin_stat_init is set to point to the following character and a signal is turned on which will cause the next call to return to nofield. Thus begin_stat is not updated until the next statement is actually being scanned.

After the next call to getfield the indices will be arranged as below.




The second call will result in a return to nofield, leaving the indices unchanged. Finally, the third call will set begin_stat to 32 and begin scanning the "val" statement, returning with begin_stat = 32, i = 35, j = 38, and first char = 39.

Getfield has a second entry named "washout". This entry is called as

```
dummy_chars = getfield$washout(dummy_nofield);
```

where dummy_chars, and dummy_nofield have the same attributes as their counterparts in the call to getfield. They are included, however, only because EPL requires that all entries have the same arguments and attributes. No character string will ever be returned nor will any transfer to the label be made. The purpose of this entry is to scan over the remainder of a statement. Thus a call to getfield$washout will return if the no field signal is on or else position first_char and begin_stat_init one character past the next semicolon and then return.

If either entry encounters an end of file, error 18 is raised and the program terminates.

get_value

This module is called by the modules which evaluate the various statement types. The calling secuence is:

```
call get_value(binval, decval, binde   error, nofield)
```

where the arguments have the following attributes:

```
binval    bit (84)
decval    fixed bin (17)
bindec    bit (1)
error     bit (2)
nofield   label.
```

Get_value makes a call to getfield to get the next field in the statement. If there is none, it returns to nofield. Otherwise it examines the first character of the field. If this is alphabetic, it calls lookup to evaluate the mnemonic. If the character is a double quote it calls binary_value. Otherwise it calls decimal_value.

The values thus obtained are returned as follows:

| | field is decimal arg | field is binary arg | field is ill-formed | field is undefined mnemonic |
|---|---|---|---|---|
| binval | "0"b | the value | "0"b | "0"b |
| decval | the value | 0 | 0 | 0 |
| bindec | "0"b | "1"b | "0"b | "0"b |
| error | "00"b | "00"b | "10"b | "01"b |

lookup

If the field is mnemonic get_value makes the call

```
call lookup(field, binval, decval, bindec, error);
```

where field is the string returned by getfield (presumably a mnemonic) and the remaining arguments are the same as the ones in get_value.

Lookup scans through the menmonics dictionary which it references by the pointer, current$mnem_ptr. If there is no match then the mnemonic is undefined. When there is a match, the corresponding value is picked out. The arguments are returned as follows:

| | field is decimal mnem | field is binary mnem | field is undefined mnem |
|---|---|---|---|
| binval | "0"b | the value | "0"b |
| decval | the value | 0 | 0 |
| bindec | "0"b | "1"b | "0"b |
| error | "00"b | "00"b | "01"b |

binary_value (see flow chart)

The call to binary_value has the form:

```
binval = binary_value (field, error_rtn);
```

where field and binval are the same as in get_value and error_rtn is a label to which control is transferred if the binary argument in field is ill-formed. Binary_value scans the argument and constructs the appropriate bit string. Its mechanism is best conveyed by flow chart (Figure 3). There are five reasons why binary_value may return to error_rtn. The terminology is defined below.

binary argument "101011011,61

binary subfield position subfield

1. a binary subfield has more than 84 bits
2. a binary subfield has a character ≠ 0,1.
3. a position subfield has a nondecimal character.
4. a position subfield has 3 or more digits.
5. a position subfield > 83 or <length (binary subfield) - 1

decimal_value

This module is called by

```
decval = decimal_value(field, error_trn);
```

where decval and field are the same as in get_value and error_rtn is a label to which control is transferred if field is ill-formed. The conversion from character to number is done very simple-mindedly. Each character from

left to right is compared successively with 0,1,...9. When a match is found, the counting index is added to 10 times the previously accumulated value (initially 0). If no match is found, then the error return is taken.

error_ms

This routine is called whenever an error condition is raised.

```
call error_ms(ercode)
```

The argument is a fixed bin (17) number which codes the type of error. Error_ms writes in the error file a mesaage of the form

```
error n in: statement
```

where n = ercode and statement is the character string starting at begin_stat (see getfield) and ending with a semicolon or the 105th character, whichever is sooner.

mnemonics

This is the mnemonic dictionary maker. It is implemented as a command and executed independently of the IOTC or IOCT. The dictionary itself is a structure declared as

```
dcl 1 mnem_dict based(p),
      2 dec_max fixed bin (17),
      2 dec_name(60) char(31),
      2 dec_val(60) fixed bin(17),
      2 bin_max fixed bin(17),
      2 bin_name(40) char(31),
      2 bin_value(40) bit(84);
```

Mnemonics calls getfield to get a name. This is checked to see that it is properly formed (error 35) (first character alphabetic) and that it is not included in the dictionary already (error 34). If these conditions are satisfied, get_value is called and its returned value is stored in the appropriate place. If the value is ill-formed (error 31)

or missing (error 33) the name is discarded. If the number of entries has been exceeded (error 32) processing continues because one mode only may have been filled. Since recursion is not permitted, get_value may not evaluate a mnemonic argument. Thus for the mnemonics process the segment, lookup, is replaced with a dummy routine which always returns an "ill-formed" error, i.e., binval = "0"b, decval = 0, bindec = "0"b, and error = "01"b.

When any name starting with "*" is encountered, then mnem_dict.dec_max and mnem_dict.bin_max are filled in and the program terminates.

iotc

The main program of the IOTC is named "iotc" and serves only as an initializer.

By calls to the supervisor iotc creates the cdt segment and gets pointers to it, to the input file, and to the mnemonics dictionary. The length in characters of the input file is also found.

The temporary cdt structures, temp_tp and temp_fld, are automatic variables in iotc and are initialized there. They have the same declarations as type and field respectively (BF.20.03 p 4) except that the arrays are of fixed size (50). These structures hold the cdt data until the number of fields and values has been found. All entries are set to zero except for temp_tp.nfld (= number of fields) which is set to one.

When it is done, iotc transfers control to tabmak, passing the temporary structures as arguments. When tabmak returns, the routine terminates.

```
call tabmak (temp_tp, temp_fld);
```

tabmak (see flow chart)

This routine picks up the keyword with a call to getfield. It compares this with the known keywords. If a match is found the appropriate statement processor is invoked. If no match, error 1 is raised. If no end of statement has been encountered, the remainder of the statement is discarded by a call to getfield$washout. If an end of statement has been found, control is transferred directly back to the beginning of tabmak to process the next statement.

When an end statement (begins with "*") is found, all the temporary structures are stored into the cdt (by a call to cdts$store,q.v.), the unused portion of the cdt is truncated by a supervisor call, and control is returned to iotc.

As explained in BF.20.06, there are restrictions on the reoccurance of cdt op types, field numbers, and values indices. To detect repetitions, lists are maintained for each of these three indices. They are named "cdtx_on", "fldx_on", and "valx_on", respectively. Each is an array of 1 bit switches which are set on when an index is encountered.

cdts

```
wash = cdts(temp_tp, temp_fld);
```

The value returned is a 1 bit switch which is 0 if an end of statement is encountered and 1 otherwise. The arguments are from tabmak.

First cdts calls get_value to pickup the op_type. This is checked for errors and, if it passes, it is saved, cdtx_on(op_type) is set on, and the entry cdts$store is called.

Cdts$store sets all fldx_on to zero. Then, except for the first call when it returns, cdts$store does 4 things:

1. call field$store and set the field entry switch off
2. allocate storage in cdt.free and update the cdt segment size
3. store temp_tp in cdt.free and reset temp_tp to zero
4. put the offset of the allocated storage in cdt.tpof.

After the return from cdts$store, cdts calls get_value for the type value, checks this for errors and stores the value in temp_tp. If there is an error in the formation of the type value, zero is used.

fields

Fields handles the "fld" statements. First it checks that a "cdt" statement has occurred. If not (error 17) control is returned to tabmak. Otherwise get_value is

called to get the field number (index) and this is checked for errors (errors 11, 3, 12, 13). If there is an error, control is returned to tabmak. Otherwise the index is saved, the fldx_on(index) is set on, the index is checked for largest yet encountered and saved if it is, and field$store is called.

The last entry is similar to cdts$store. First it resets all valx_on to zero. Then, unless this is the first "fld" statement after a "cdt" statement in which case it returns, it

1. allocates in cdt.free for temp_fld and updates segment size

2. stores temp_fld in cdt.free and resets temp_fld to zero,

3. puts the offset of the allocated storage in temp_tp.fldof.

After the call to fields$store, there are calls to get_value to pick up the field action, field and field mask. These are all checked for errors (2,3,8,9,10) and stored in the temp.fld stucture (zero is stored if an error is detected). A value for field mask or an end of statement causes a return.

The calling sequence is

```
wash = fields(temp_tp, temp_fld);
```

values

This routine handles the "val" statement. It first checks for proper sequence ("fld" statement must have occurred) and if there is a violation (error 17) returns. Error 17 is also returned if the field action of the preceding "FLD" statement (temp_fld.fldact) is not 1 (mask value substitution). Otherwise, it calls get_value to pick up the index and checks it for errors (4,5,6,7). If the index is invalid, values returns. Otherwise it sets valx_on(index) on and calls get_value for a value, checks it for errors (2,3,5,6) and stores it in temp_fld.val(index). The index is then incremented by 1 and the above procedure repeated until there are no more values in the statement or until the maximum index value (50) is exceeded (error 5). If a value is ill-formed or undefine zero is used in its place.

```
wash = values(temp_fld);
```

ioct

This is the main program for the IOCT. It serves as an initializer. By means of calls to the supervisor it creates a segment to hold the changes structures. Pointers to this and to the mnemonic dictionary and input file are also gotten. Another supervisor call gets the length of the input file in characters.

The temporary changes structure is initialized to zero except for the pointer which is set to null. After this there is a call to decode from which there will be no return.

decode

call decode;

A call to getfield picks up the first element of the statement and examines the first character. If it is "/" the statement is ignored. If it is "*" the program terminates. Otherwise the field is checked to see if it is a proper label. This means

| | | |
|---|---|---|
| 1. | first character alphabetic | error 41 |
| 2. | remaining alphanumeric or "_" | |
| 3. | not already used | error 39 |
| 4. | < 31 characters | error 40 |

If any of the first 3 are violated the statement is ignored. If the fourth, the first 31 characters are used.

Once a proper label is gotten there is a call to get_value to pick up the op type. This is also checked for errors (42,43,44,45,56) and, if any occur, the statement is ignored. Otherwise the op type is saved and the remainder of the statement is processed by a call to changes.

changes

call changes;

This routine processes the fieldi valuei pairs. They are pulled in sequentially by calls to get_value and checked for errors. Errors in a fieldi (error 46,47,48,49,55) cause that fieldi and its valuei to be ignored. Errors

in a valuei (Errors 51,52,59) cause it to be set to zero except error 50 which causes it to be set to the first 24 bits. These pairs are stored in the temporary structure until the end of the statement is encountered. Then a call to store is made. If, however, there are no (error free) fieldi valuei pairs, then this call is omitted and control is returned directly to decode.

store

call store;

Store adds the appropriate number of words to the changes structure segment and gets a pointer to the beginning of this new block of storage. This pointer is set up in the linkage section as an external with name = label. Using this pointer, the temporary structure is stored into the segment and the temporary structure and used fieldi list are reset to zero.

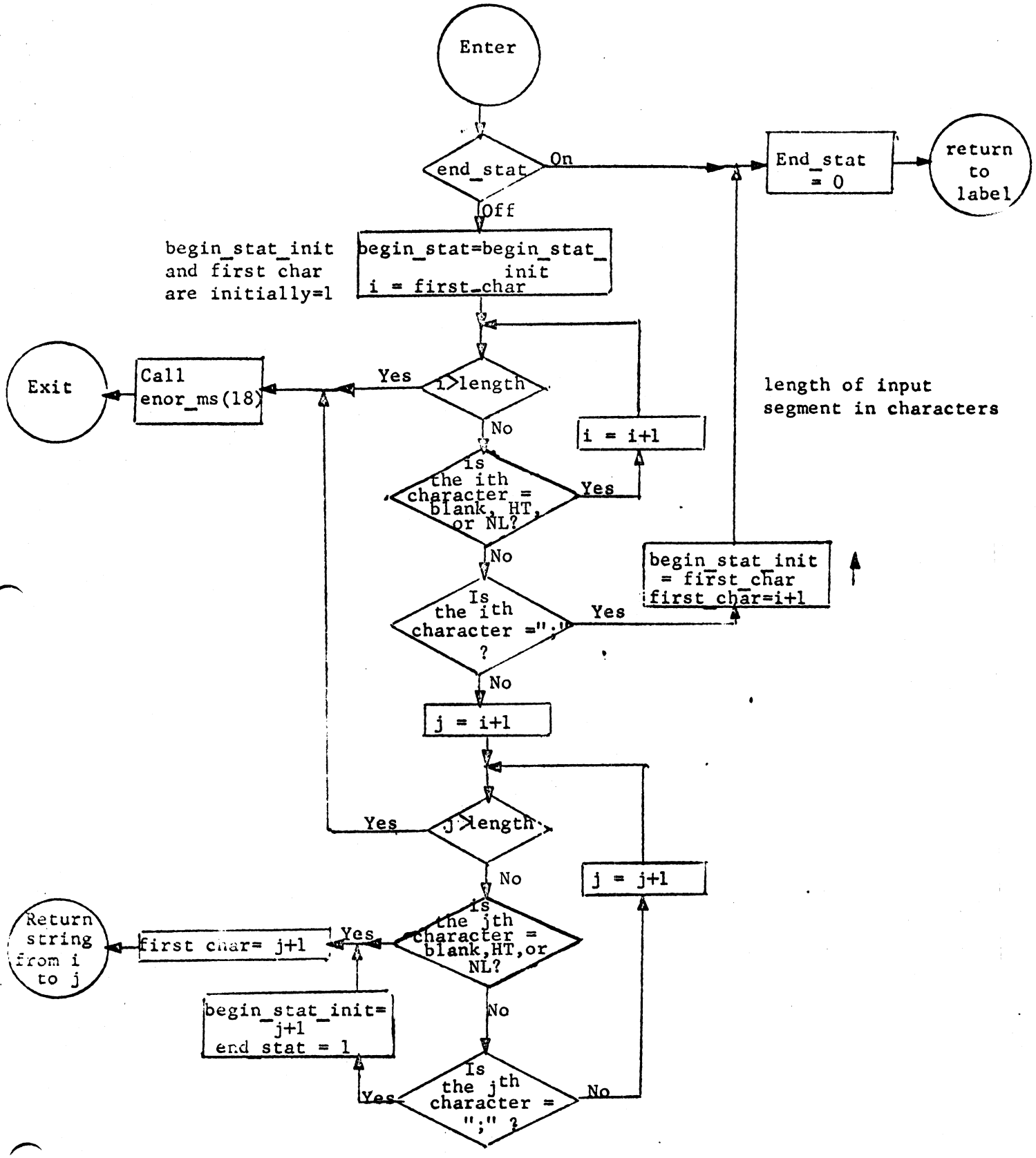



Figure 1. getfield

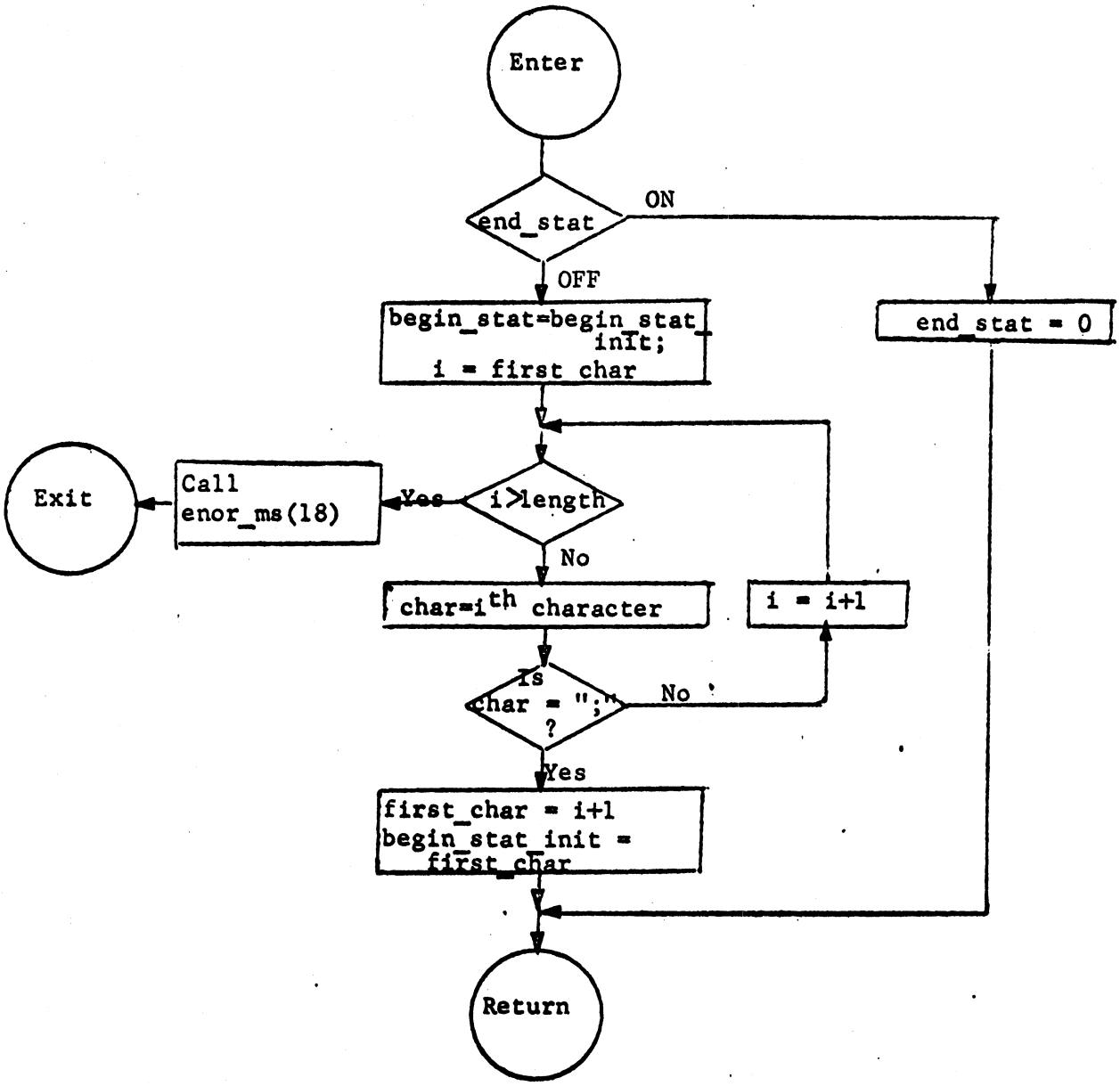



Figure 2. getfield$washout

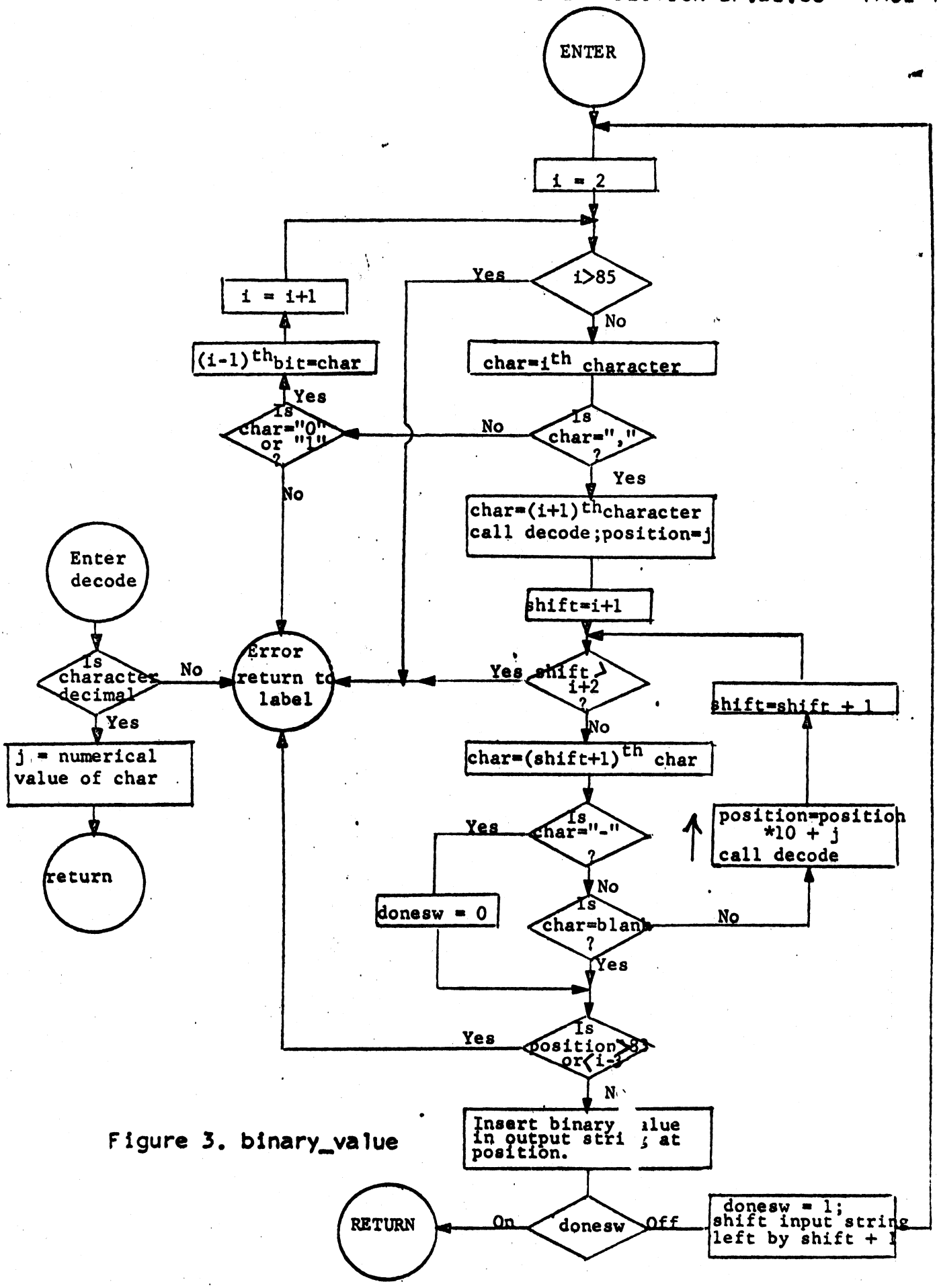



Figure 3. binary_value

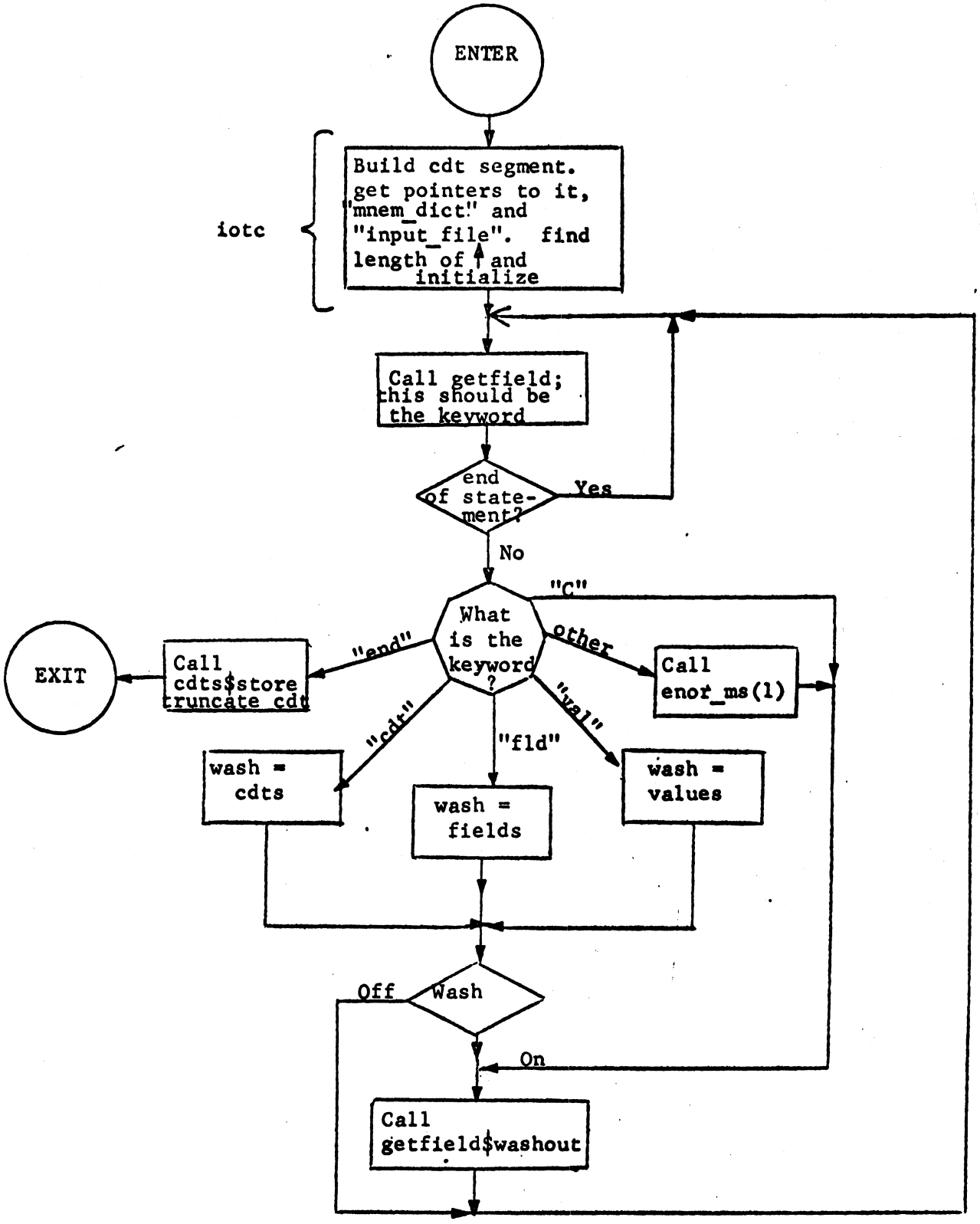



Figure 4. iotc and tabmak