FVN

RCSL : 44 - RT 740

Author : A.P. Ravn

Edited : September 1973

M U S I L

Keywords : RC 3600 MUS System Software, Programming Language.

Abstract : Syntax Rules for MUSIL language. Description of standard procedures. Explanation of I/O handling.

CONTENTS PAGE

CONTENTS PAGE

CONTENTS PAGE

PROGRAMS

A program has the form of a procedure declaration without heading

<program> ::= <constant definition part> <type definition part>
<variable declaration part>
<Procedure declaration part> <statement part>

The constant definition part contains all constant synonym definitions.

<constant definition part> ::= <empty> |
const <constant definition> {, <constant definition>}*;

The type definition part contains all type definitions

<type definition part> ::= <empty> |
type <type definition> {; <type definition>}*;

The variable declaration part contains all variable declarations

<variable declaration> ::= <empty>!
var <variable declaration> {; <variable declaration>}*;

The procedure declaration part contains all procedure declarations

<procedure declaration part> ::=
{<procedure declaration> ;}*

The statement part specifies the algorithmic actions to be executed upon an activation of the procedure by a procedure statement.

<statement part> ::= <compound statement>

(cf. p10).

IDENTIFIERS

Identifiers serve to denote constants, types, variables, and procedures.

<identifier> ::= <letter> { <letter>|<digit>}*

COMMENTS

The construct

!<any sequence of symbolts not containing "!">!

may be inserted at any place.

CONSTANT DEFINITIONS

A constant definition introduces an identifier as a synonym to a constant.

<unsigned constant> ::= <number> | '<character>$^{\oplus}$ ' |.# <number>$^{\oplus}$ #
<constant> ::= <unsigned constant> | <sign> <number>
<constant definition> ::= <identifier> = <constant>
<character> ::= <graphic> | <<number>>

A characterstring surrounded by '...'
is packed in ASCII alphabet. Char values less than 32_{10} are ignored.
Any string is terminated by a zero bytevalue.
The notation <<number>> denotes an 8 bit binary representation of a non-graphic or graphic character.

The construct # <number> ...# is used to pack a number of bytevalues which should be used for a table. E.g.

```
dtab =    #  1   8'377
             2   8'1777
             3   8'077
             4   8'007
             5   8'000    #
```

Examples:

```
'<10>error'      "name <25>"          #1 2 3 ! finis table ! #
```

The decimal notation is used for numbers, which are the constants of the data types integer.

In <number> ' <number> the first number is treated as radix for the second number.

<number> ::= <integer>|<integer> ' <integer>
<integer> ::= <digit>$^{\oplus}$
<sign> ::= + | −

Examples:

```
1     100      2'100     8'17777
```

DATA TYPE DEFINITIONS

A data type determines the set of values which variables of that type may assume and associates an identifier with the type.

In the case of structured types, it also defines their structuring method.

```
<type> ::= <scalar type> | <record type> |
           <file type> | <type identifier>
<type identifier> ::= <identifier>
<type definition> ::= <identifier> = <type>
```

Examples:

```
line = string (20)
pline = record
        l1:  line;
        l2:  line;
        l3:  line
        end
in =    file
        'MT0 , 14, 1, 600,FB
        of pline;
```

SCALAR TYPES

The scalar types may be one of two standard types.

integer — which denotes a 15 bit signed binary value type.

string (<number>) — which denotes a string of <number> 8 bit bytes.

RECORD TYPES

A record type is a structure consisting of a fixed number of components, possibly of different types. The record type definition specifies for each component, called field, its type and an identifier which denotes it. The scope of these so called field identifiers is the record definition itself, and they are also acessible within a field designator referring to a record variable of this type.

<record type> ::= record <field list> end
<field list> ::= <fixed part>
<fixed part> ::= <record section> { ; <record section> } *
<record section> ::= <field identifier> { , <field identifier> } *: <field type>
<field type> ::= <type> from <number>

<type> can only be of scalar type string(<number>).

The suffix 'from <number>' is used to force the field to start at byte <number> of the record. The bytes are numbered 1, 2, 3 .....

A field may not start beyond position 255.

A record section with a <field type> containing from, can only contain one fieldidentifier. (Otherwise the fieldidentifiers would be synonymous).

Examples:

```
record
ccw:      string (1);
line:     string (132)
end

record
totalrec: string (80);
col 10:   string (1) from 10
end;
```

FILE TYPES

Defines a file and the associated file records.

```
<file type> ::=      file
                     <device>, <kind>, <buffers>
                           , <blocklength> { , <recformat> }1 0
                     <file options>
                     of <filerecordtype>
<file options> ::=   { ;<giveup option> }1 0  { ;<conversion> }1 0
<giveup option> ::= giveup <action>, <mask>
<conversion> ::=     conv <table>
<filerecordtype> ::= <record> / <scalartype>      ; NOTE only string types allowed
```

The entries between file and of forms part of the file descriptor (zone).
See i/o handling.

<device> ::=	" { <character> }5 0 "	Device driver process name.
<kind> ::=	<number>	Kind for handling of device.
<buffers> ::=	<number>	The number of buffers used for the file.
<blocklength> ::=	<number>	The buffer size of each buffer.
<recformat> ::=	{ U / F / V } { B }1 0	The format of the records.
<action> ::=	<procedure identifier>	Exception handling procedure
<mask> ::=	<number>	User bits of status word.
<table> ::=	<string identifier>	Conversion table.

DECLARATIONS AND DENOTATIONS OF VARIABLES

Variable declarations consist of a list of identifiers denoting the new variables, followed by their type.

<variable declaration> ::= <identifier> {,<identifier>} *: <type>

Denotations of variables either denote an entire variable or a component of a variable.

<variable> ::= <entire variable> | <component variable>

Entire Variables

An entire variable is denoted by its identifier.

<entire variable> ::= <variable identifier>
<variable identifier> ::= <identifier>

Examples:

a xxx

Component Variables

A component of a variable is denoted by the denotation for the variable followed by a selector specifying the component. The form of the selector depends on the structuring type of the variable.

<component variable> ::= <current file component> | <field designator>

Field Designators

A component of a record variable is denoted by the denotation of the record variable followed by the field identifier of the component.

<field designator> ::= <record variable>. <field identifier>
<record variable> ::= <variable>
<field identifier> ::= <identifier>

Examples:

u.realpart in↑.ccw
v.realpart

Current File Components

At any time, only the one component determined by the current file position (or file pointer) is directly accessible.

If a file variable is used without ↑ (the file record indicator) it refers to the file descriptor record. See i/o-handling.

Example

```
in↑
```

STATEMENTS

Statements denote algorithmic actions, and are said to be executable.

<statement> ::= <simple statement> | <structured statement>

Simple Statements

A simple statement is a statement of which no part constitutes another statement.

<simple statement> ::= <assignment statement> |
<procedure statement> | <goto statement>

Assignment Statements

The assignment statement serves to replace the current value of a variable by a new value indicated by an expression. The assignment operator symbol is: := ,pronounced as "becomes".

<assignment statement> ::= <variable> := <expression>

The variable and the expression must be of identical type.

Procedure Statements

A procedure statement serves to execute the procedure denoted by the procedure identifier. The procedure statement may contain a list of actual parameters which are substituted in place of their corresponding formal parameters defined in the procedure declaration. The correspondence is established by the positions of the parameters in the lists of actual and formal parameters respectively. There exist two kinds of parameters: variable-, constantparameters.

In the case of variable parameters, the actual parameter must be a variable. If it is a variable denoting a component of a structured variable, the selector is evaluated when the substitution takes place, i.e. before the execution of the procedure. If the parameter is a constant parameter, then the corresponding actual parameter must be an expression.

<procedure statement> ::= <procedure identifier> |
<procedure identifier> (<actual parameter>
{,<actual parameter>} *)
<procedure identifier> ::= <identifier>
<actual parameter> ::= <expression> | <variable>

Goto Statements

A goto statement serves to indicate that further processing should continue at another part of the program text, namely at the place of the label. Labels can be placed in front of statements being part of a compound statement.

<goto statement> ::= goto <label>
<label> ::= <integer>

Structured Statements

Structured statements are constructs composed of other statements which have to be executed either in sequence (compound statement), conditionally (conditional statements), or repeatedly (repetitive statement).

Compound Statements

The compound statement specifies that its component statements are to be executed in the same sequence as they are written. Each statement may be preceded by a label which can be referenced by a goto statement

<compound statement> ::=
begin <component statement> {; <component statement>} * end
<component statement> ::=
<statement> | <label definition> <statement>
<label definition> ::= <label>:

Example:

begin z:= x; x:= y; y:= z end

Conditional Statements

A conditional statement selects for execution a single one of its component statements.

<conditional statement> ::= <if statement>

If Statements

The if statement specifies that a statemtnt be executed only if a certain condition is true . If it is false, then no statement is to be executed.

<if statement> ::= if <expression> then <statement>

The expression between the symbols if and then must be relational.

Repetitive Statements

Repetitive statements specify that certain statements are to be executed repeatedly.

<repetitive statement> ::= <while statement> |
<repeat statement>

While Statements

<while statement> ::= while <expression> do <statement>

The expression controlling repetition must be relational. The statement is repeatedly executed until the expression becomes false. If its value is false at the beginning, the statement is not executed at all.

Repeat Statements

<repeat statement> ::=
repeat <statement> { ; <statement> }* until <expression>

The expression controlling repetition must be relational. The sequence of statements between the symbols repeat and until is repeatedly (and at least once) executed until the expression becomes true .

EXPRESSIONS

Expression are constructs denoting rules of computation for obtaining values of variables and generating new values by the application of operators. Expressions consist of operands, i.e. variables and constants, operators, and functions.

The rules of composition specify operator precendences according to four classes of operators. The monadic have the highest precedence, followed by the so-called multiplying operators, then the so-called adding operators, and finally, with the lowest precedence, the relational operators. Sequences of operators of the same procedence are executed from left to right. These rules of precedence are reflected by the following syntax:

```
<factor> ::= <variable> | <unsigned constant> | <function designator> |
             <monadic operator> <variable> |
             (<simple expression>)
<term> ::= <factor> | <term> <multiplying operator> <factor>
<simple expression> ::= <term> |
                        <simple expression> <adding operator> <term> |
                        <adding operator> <term>
<expression> ::= <simple expression> |
                 <simple expression> <relational operator>
                 <simple expression>
```

Examples:

Factors: x byte in↑.ccw translate (x,ctab)
15
(x+y+z)

Terms x * y
i/(1 - i)

Simple expressions: x+y
-x
i * j+1

Expressions: $p \stackrel{\leq}{=} q$

OPERATORS

Monadic operators

<monadic operator> ::= byte | word

operator	operation	type of operands	type of result
byte	take first byte	string	integer
word	take first and second byte		

Adding Operators

<adding operator> ::= + | - | shift | extract | and

operator	operation	type of operands	type of result
+	addition	integer	integer
-	subtraction	integer	integer
shift	first operand logical shift left second operand positions	integer	integer
extract	first operand mask out second operand positions from right	integer	integer
and		integer	integer

When used as operators with one operand only, - denotes sign inversion, and + denotes the identity operation.

Multiplying Operators

<multiplying operator> ::= * | /

operator	operation	type of operands	type of result
*	multiplication	integer	integer
/	division	integer	integer

Relational Operators

<relational operator> ::= = | < > | < | <=|>=|>

operator	type of operands
= < >	
< >	any scalar type
<= >=	

Function Designators

A function designator specifies the activation of a function. It consists of the identifier designating the function and a list of actual parameters. The parameters are variables, expressions, procedures, and functions, and are substituted for the corresponding formal parameters

<function designator> ::=
<fuction identifier> (<actual parameter> { , <actual parameter>} *)
<function identifier> ::= <identifier>

PROCEDURE DECLARATIONS

Procedure declarations serve to define parts of programs and to associate identifiers with them so that they can be activated by procedure statements. A procedure declaration consists of the following parts:

<procedure declaration> ::=

<procedure heading> <statement part>

The procedure heading specifies the identifier naming the procedure and the formal parameter identifiers (if any). The parameters are either constant- or variable.

<procedure heading> ::= procedure <identifier> ;

Example:

```
procedure getnext
! iz is a file type, which describes a magnetic tape. The blocks have
  been organized:
  byte 1 - 5     skipped
  byte 6 - 7     variables block length
  remaining      records of variable length:
  byte 1         length of record -1
  byte 2         ccw
  remaining      printline

The procedure places, ccw of next record in global variable ccwx:
string (1) and makes the file record z point at the printline !

      begin
1:    if z.zrem = 0 then
        begin
        inblock (z);
        getrec (z,5);          ! skip 5 bytes !
        getrec (z,2);          ! get length !
        z.zrem := word z↑;     goto 1
        end;
```

```
getrec (z, 1);                  ! recsize !
z.zleng := byte z↑;
getrec (z, 1);                  ccwx := z↑;
getrec (z, z.zleng - 1)
end;
```

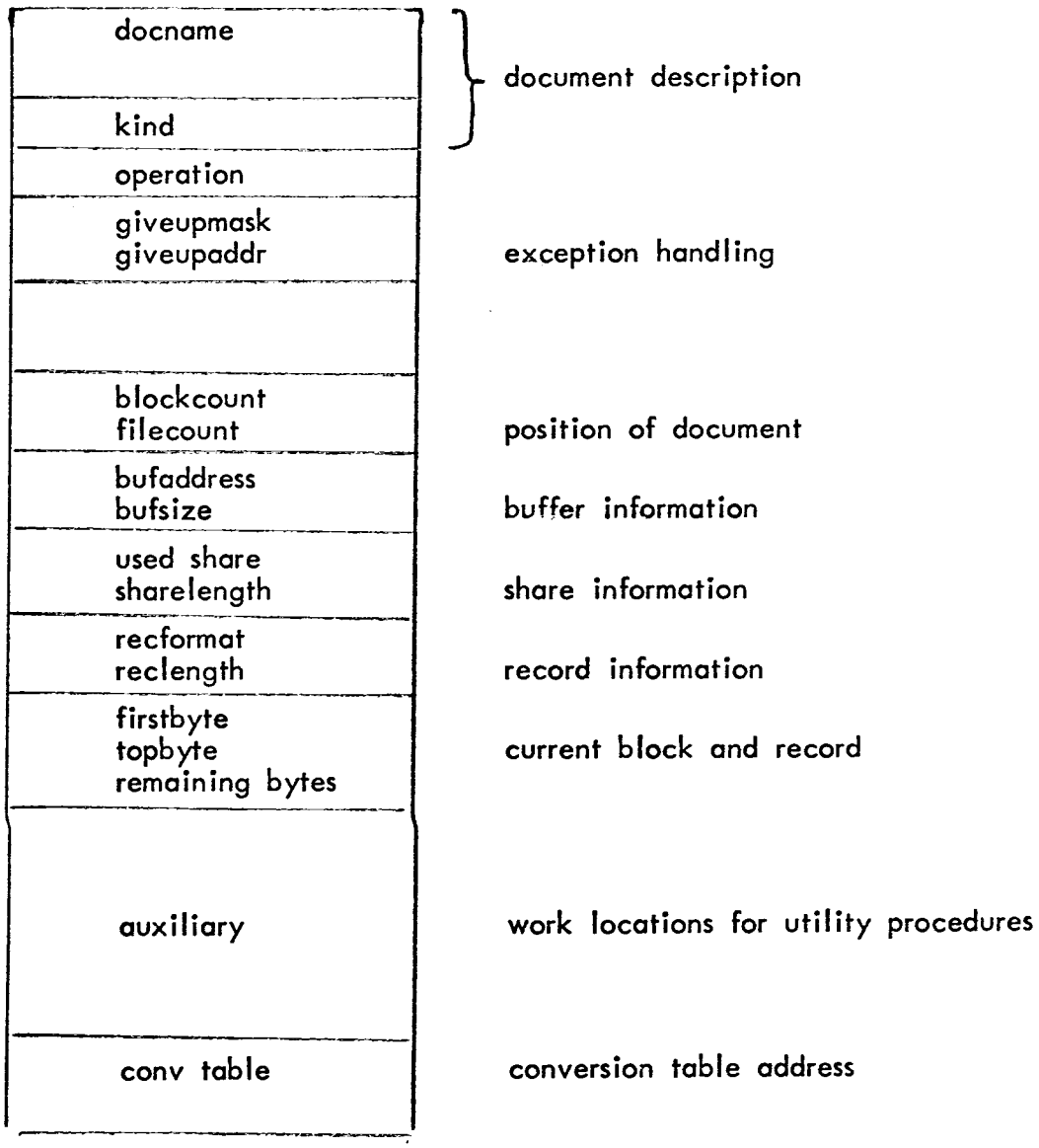
I/O-HANDLING

ZONES

All I/O-procedures work on files, and in order to understand the function in detail, some knowledge of the description of a file (a zone) is needed.

A zone contains 3 parts: Zone descriptor, which contains information about the document and the device, that holds it. Share descriptiors which holds information about the current activities in the buffers which they describe. A buffer area which physically contains the descriptors and associated buffers.

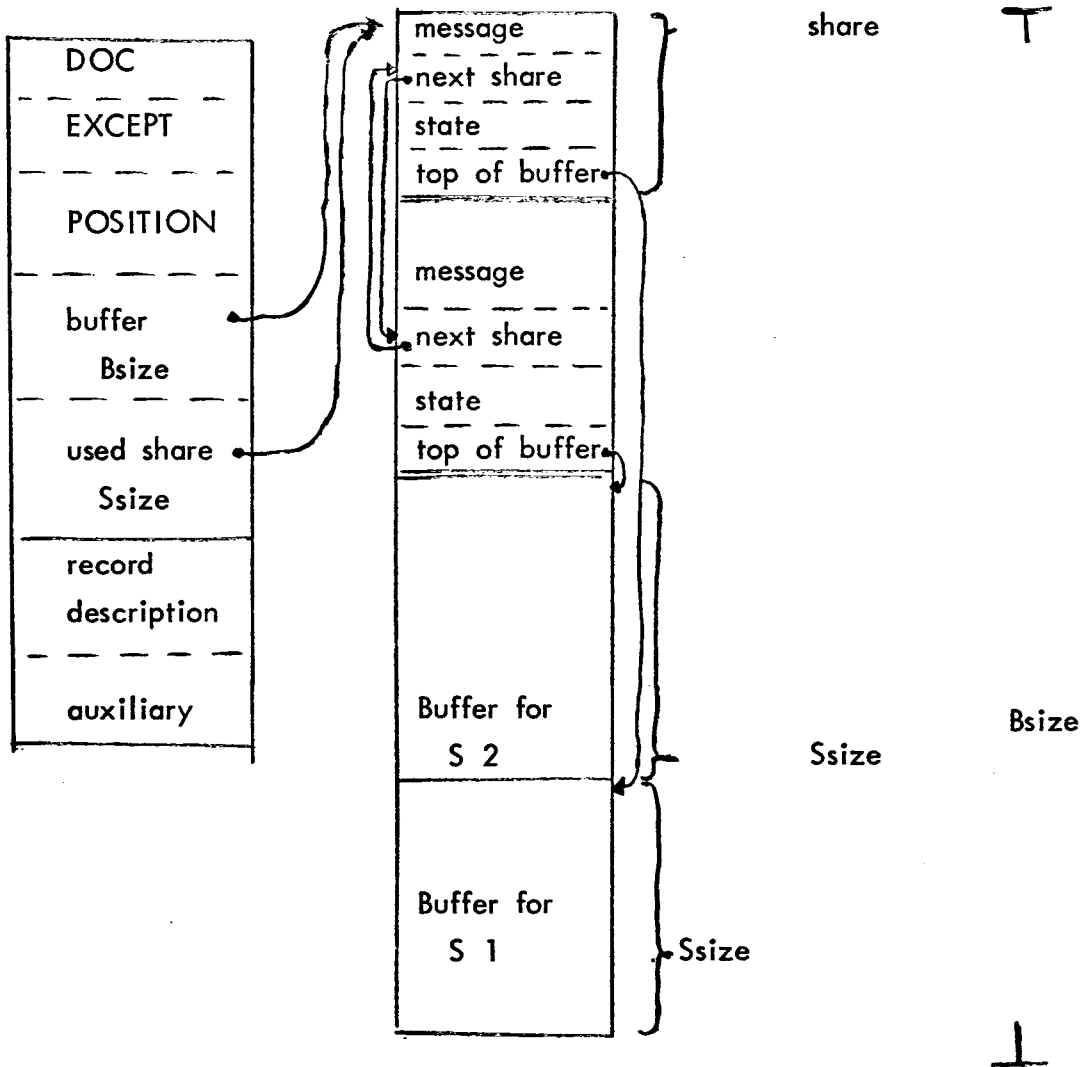
Zone descriptor:



Zone descriptor fields available in MUSIL:

z.zmode	:	mode of operation
z.zmask	:	giveup mask for device errors
z.zfile	:	file position of document
z.zblock	:	block position of document
z.zfirst	:	pointer (byteaddress) of first byte of current record
z.zstop	:	pointer to first byte after current record
z.zlength	:	length (in bytes) of current record
z.zrem	:	length (in bytes) of remaining part of current block
z.z0	:	user status of file in giveup procedure

Full organization of a file:



zone with 2 buffers

IDENTIFICATION OF A DOCUMENT

The term document is used to describe a medium, which is able to contain data, and which is mounted on a device.

A document is described inside a zone descriptor by:

document name, the process name of the driver, which controls the device.

operation, that is the operation code, which should be used in any transput operation sent as message to the driver process.

device kind, a word, which contains some bits, that describe how transfer errors may be handled.

At present, the following bits of kind are defined:

b15:	char	:	set if the device is character oriented, transfers information in terms of characters
b14:	blocked	:	set if a full block should be transferred as a unit
b13:	positionable	:	set if positioning has any effect.
b12:	repeatable	:	set if an operation may be repeated.

The remaining bits of the kind word should be zero.

Description of mode and kind applicable to standard driver processes, are found as part of their description.

Examples of kinds:

Magnetic Tape Station	1110
Line Printer	0001 or 0011
Card Reader	0010
Teletype	0001
Paper Tape Punch	0001
Paper Tape Reader	0001

HANDLING OF EXCEPTIONS

In the input/output procedures the user may select certain statusbits, which if set in the answer to a message to the driver, will transfer control to user code. These user facilities are described in the zone descriptor by:

giveupmask, giveupaddress.

When the basic procedure wait transfer receives an answer, the statusword is augmented with the following bits:

b15: repeaterror	is set if the standard repetition of operations has given negative results.
b13: rejected	is set if a control operation with command = 10_2 is checked.
b12: position error	is set, if kind (13) is one and filecount or blockcount of anseer does not match with the corresponding updated values of the zone descriptor.

This combined driver and standard procedure status is compared with the giveupmask. Common ones from the users_status.

Remaining status bits are given to the standard check actions, which executes the following recovery work:

b0:	disconnected	the error is hard.
b1:	off-line	the error is hard.
b2:	device busy	the operation is repeated.
b3:	- - 1	ignored.
b4:	- - 2	ignored.
b5:	- - 3	ignored.
b6:	illegal	the error is hard.
b7:	eof	the error is hard.
b8:	block_error	the error is hard.
b9:	data_late	if kind (12) is 1 then operation is repeated, otherwise the error is hard.
b10:	parity error	if kind (12) the operation is repeated else it is a hard error.

b11:	end medium	if bytecount of answer is nonzero and operation is input no action is taken, otherwise the error is hard.
b12:	position error	hard error.
b13:	rejected	hard error.
b14:	timer	hard error
b15:	repeaterror	hard error.

A hard error results in a breakprocess call, with errorcode = 5.

An operation is repeated a maximum of 5 times. If it is still erroneous, it is classified as having a repeat error. The cause of the unsuccessful repeats is included in user status.

When remaining bits have been treated by the standard actions, control is given to giveupprocedure if users_bits are different from zero. Otherwise a normal return from wait_transfer takes place.

RECORD STRUCTURE

There exist three formats for records. For each type, the records may be either blocked or unblocked.

Record type:	Format code:	Blocked:
Unformatted	U	
Fixed length	F	
Variable length	V	B

Unformatted

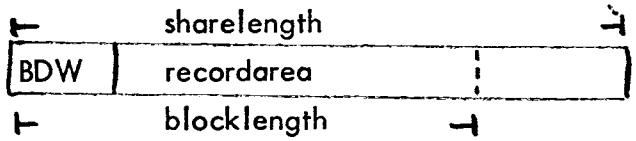
A block contains sharelength bytes or less. In output a full block is transferred to the device regardless of contents. By input as many bytes as requested are delivered from the block. If the records are blocked, change of blocks takes place, when the remaining bytes of the zone cover the demand insuffiently.

Fixed length

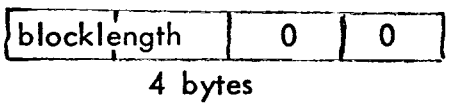
Every block containing one or more records (blocked) of fixed length.
The length is given by the zoneparameter reclength. If sharelength is not an integral multiple of recordlength, the last bytes of input are skipped.

Variable length

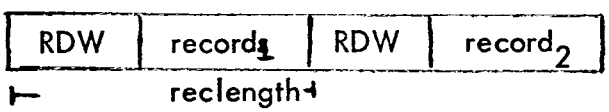
The block contains, in the block descriptor BDW, the length of the total block.



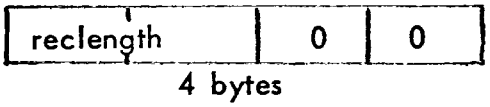
A BDW contains no further information:



The recordarea may contain one (unblocked) or more records. Each record is headed by a 4 byte record descriptor RDW.



A RDW contains the recordlength and a segmentcode, which always is zero.



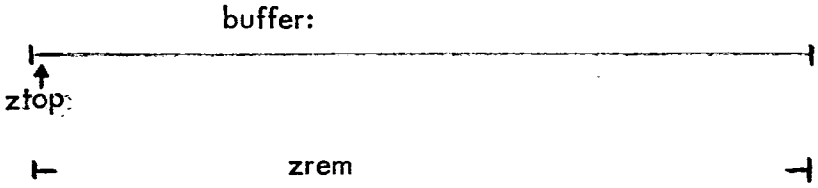
CURRENT RECORD

Changes in zone descriptor fields after activation of standard procedures:

Initialization procedures:

open, waitzone, setposition

1) output (zmode extract 2 = 3)



zfirst is undefined
zlength is undefined for U and V formats

2) input or sense (zmode extract 2 <> 3)

zrem = 0

zfirst is undefined
zlength is undefined for U and V formats.

close:

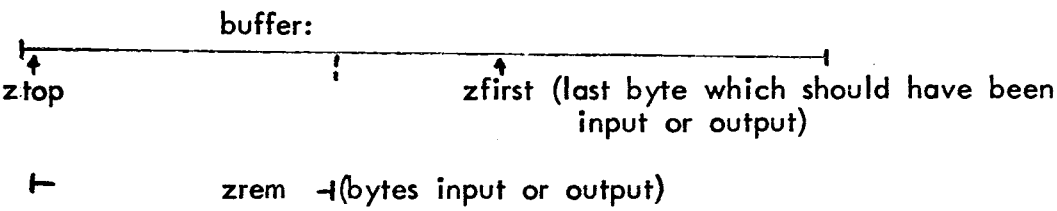
all pointers are undefined.
all values are undefined.

Basic I/O-procedures

transfer :

all pointers are undefined.
all values are undefined.

waittransfer, giveup procedure:



zlength undefined

inblock

zrem

zfirst undefined

zlength undefined

outblock

zrem

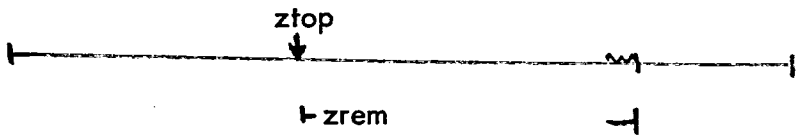
zfirst undefined

zlength undefined

Record I/O procedures.

getrec (z,length)

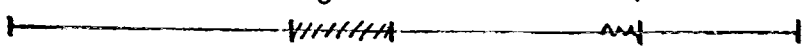
Before call



After call:

length =

zlength zrem



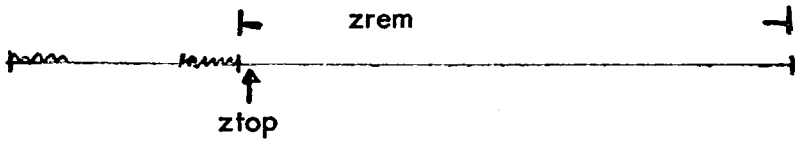
current rec

zfirst ztop

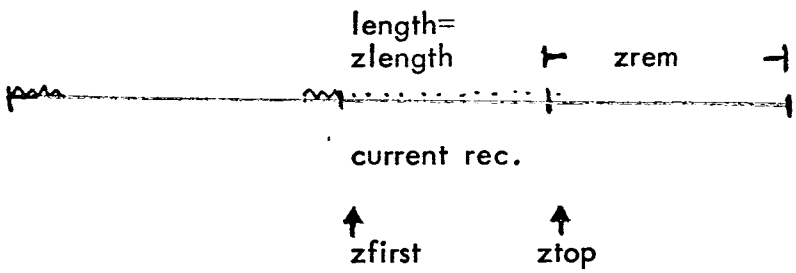
If length is greater than zrem different things may happen for the various formats.

putrec (z,length)

Before call:

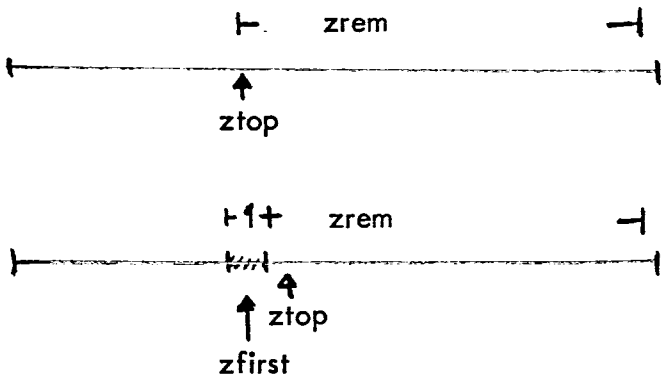


After call:

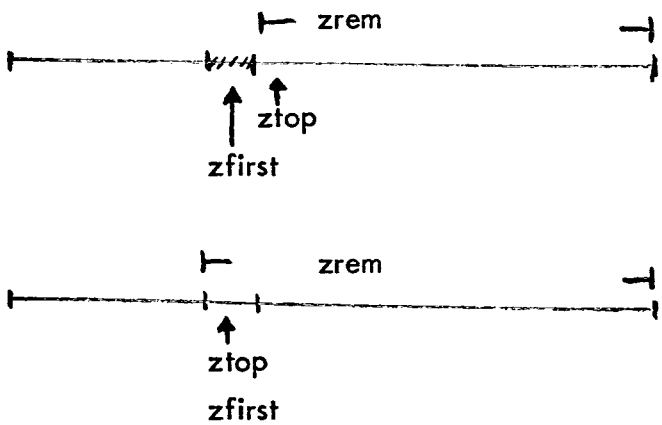


Character I/O

inchar , outchar



backspace:



I/O-PROCEDURES

Basic procedures

procedure TRANSFER (f: file ; const length, operation : integer) ;

Initiates a transfer operation described by 'operation' in used_share of zone. The bytecount of the message is put to length. Sharestate of used share points to the buffer used for the message. Used share is updated to next share.

NB: starttransfer does not check that state of used share is free (zero). If the state is not free, the buffer address saved in state is lost permanently.

procedure WAITTRANSFER (f : file) ;

Examines used_share of zone. If state is free (zero) the procedure returns immediately, otherwise it waits for answer to the message placed in buffer identical with state and sets state to free.

When the answer arrives the status is checked as described in HANDLING OF EXCEPTIONS f.ztop is adjusted to point at firstaddress of share. f.zrem is adjusted to bytecount of answer.

procedure REPEATSHARE (f : file) ;

NB: This procedure must not be called outside the giveup procedure of f.

It returns to waittransfer after having restarted the operation, which was rejected.

procedure INBLOCK (f : file) ;

Administrates the basic cyclic buffering strategy for input procedures inchar and getrec. The algorithm used is:

```
while f.zused.state = free do
  transfer (f, f.zsharelength, f.zmode) ;
waittransfer (f) ;
! n-1 shares are busy and one is ready with input!
```

procedure OUTBLOCK (f : file) ;

Administrates the basic cyclic buffering strategy for output procedures, outchar and putrec.

```
transfer (f, f.zsharelength_f.zrem, f.zmode) ;
waittransfer (f);
!n-1 shares may be busy with output, and one is ready with input!
```

Initialization procedures

procedure OPEN (f : file ; const operation : integer) ;

The operation is placed in the modeword of zonedescriptor.
Then remaining bytes of zone is initialized to zero if command = 1 or else to sharelength. Top of zone points to first of used share.

To initialize the transfers a control message is sent to the process specified by zname. This message includes reservation and set up of conversion table address.

procedure WAITZONE (f : file) ;

Terminates the current activities of the zone as follows:
If command is output, a last block is output.
Then all pending transfers are waited for. Either by means of wait transfer if command is output, or else by means of waitanswer (no checking takes place).

```
    if f.zmode extract 2 = 3 then outblock (f);
0:  stop:= f.zused;
1:  share:= f.zused:= f.zused.nextshare;
    if share.state <> free then
      begin
      if share.operation extract 2 = 3 then
      begin waittransfer (f); goto 0 end
      else waitanswer (share.state);
      end;
    if share <> stop then goto 1;
```

procedure CLOSE (f : file ; const release : integer);

First the zone is set neutral by waitzone. Then if command is output a termination, and if release is nonzero a release reservation, and disconnect message is sent to the process specified by the zonename.

Finally operation of zone is set to zero, and the file is set neutral by waitzone.

procedure SETPOSITION (f : file ; const file, block : integer);

Waits for all pending transfers to the zone as described in procedure waitzone. Then it sends a position message, which contains the new file- and blockcount.

Record Procedures:

procedure GETREC (f : file ; var bytes : integer);

Makes the next record available in inputbuffer. Depending on recordformat the actions are:

Unformatted unblocked:

```
inblock (f);
f.zlength:= bytes:= f.zrem;
goto 10;
```

Fixed length unblocked:

```
inblock (f);
bytes := f.zlength;
!recordlength is used!
goto 10;
```

Unformatted blocked:

```
f.zlength := bytes;
```

Fixed length blocked:

```
bytes := f.zlength;
if f.zrem < bytes then inblock (f);
goto 10;
```

Variable length blocked:

```
if f.zrem > 0 then goto 5;
```

Variable length unblocked:

```
inblock (f)
f.zrem:= word f↑ -4;
f.ztop:= f.ztop +4
5: f.zlength:= bytes:= word f  -4;
   f.zrem:= f.zrem -4;
   f.ztop:= f.ztop +4;
10: f.zfirst:= f.ztop;
   f.ztop:= f.ztop + bytes;
   f.zrem:= f.zrem -bytes;
```

procedure PUTREC (f : file ; const bytes : integer);

Makes space for a record in the output buffer. Depending on recordformat the actions are:

Fixed length unblocked:

```
bytes:= f.zlength;
```

Unformatted unblocked:

```
outblock (f);
if f.zrem < bytes then
  breakprocess (cur,4);
update_top_of_zone_and_rem_of_zone;
return;
```

Fixed length blocked:

```
bytes:= f.zlength
```

Unformatted blocked:

```
if f.zrem <bytes then outblock (f);
if f.zrem < bytes then
  breakprocess (cur,4);
update_top_of_zone_and_rem_of_zone;
return;
```

Variable length blocked:

```
if f.zrem < bytes + 4 then
  outblock (f);
goto 0;
```

Variable length unblocked:

```
      outblock (f);
0:    if f.zrem <bytes + 4 then
        breakprocess (cur,4);
      f↑. (0:1):= bytes + 4;
      f↑. (2:3):= 0;
      f.ztop:= f.ztop + 4;
      f.zrem:= f.zrem -4;
      update_top_of_zone_and_rem_of_zone;
      f.zused.first (0:1):= f.zsharelength - f.zrem;
      f.zused.first (2:3):= 0;
      ! block descriptor word inserted!
      return;

 procedure update_top_of_zone_and_rem_of_zone;
      begin
      f.zfirst:= f.ztop;
      f.zlength:= bytes;
      f.ztop:= f.ztop + bytes;
      f.zrem:= f.zrem - bytes;
      end;
```

Character I/O Procedures

procedure INCHAR (f : file ; var bytevalue : integer);
Fetches the next 8 bit bytevalue from f.

procedure OUTCHAR (f : file ; const bytevalue : integer);
Outputs the 8 bit of bytevalue on f.

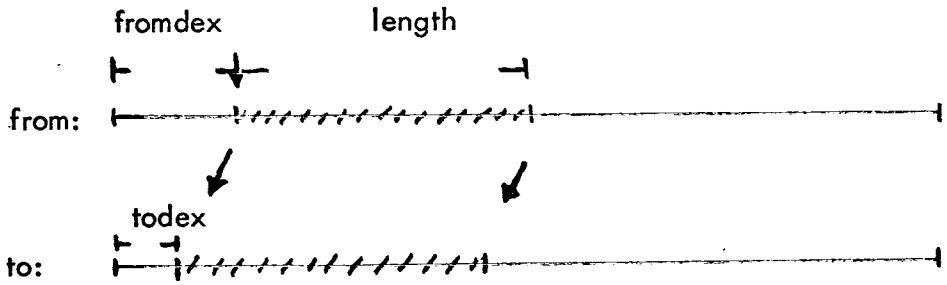
procedure OUTTEXT (f : file ; const text : string (*)) ;
Outputs the text, which should be terminated by a zero byte, on f.

STRING MANIPULATION

For all of the following procedures it should be noted that any explicity given lengths or indices are not checked for validity by the system. This may have unpredictable results in case of errors.

```
procedure MOVE (const from : string (*); fromdex : integer;
                var to : string (*); todec : integer;
                const bytes : integer);
```

'bytes' bytes are moved from string 'from' with an offset of 'fromdex' to string 'to' with offset 'todex'.



NB: MOVE always moves to a full word. This means, that if to + todex + length is an odd address, the following byte will be destroyed.

```
procedure CONVERT (const from : string (*); var to : string (*);
                   const table : string (*) ; length: integer);
```

The number of bytes indicated by 'length' is moved from string 'from' to string 'to'. Every byte moved, is converted by tablelockup in 'table'.

The table should contain as many entries, normally 256, as there is in the range of the possible bytevalues.

NB: In connection with files, the facility of conversion during in- or output should be used instead of a programmed call of CONVERT.

```
procedure TRANSLATE (const byte: string(*); var value: string (*);
                     const table : string (*));
```

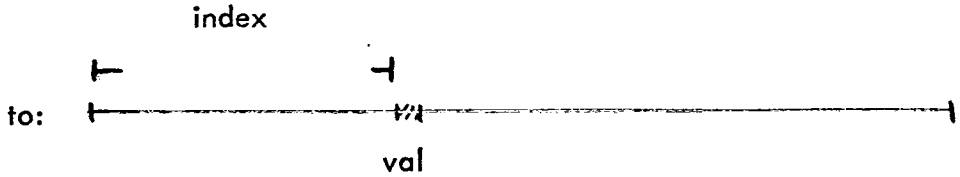
Converts a single byte from 'byte' through an associative search in 'table', and delivers the result in 'value'

The table should be organized as:

```
table = #   arg_1    value1
            arg_2    value2
            ..       ..
            ..       ..
             0        0
             0       default value #
```

```
procedure insert (const val : integer ; var to : string (*);
                  const index : integer);
```

The less significant 8 bit of val is regarded as a bytevalue, and inserted in string 'to', at position 1 + 'index'.



CONVERSION PROCEDURES

These procedures makes conversion between 16 bit binary integer format and ASCII-characters string format.

procedure BINDEC (const value : integer ; var text : string (*));

The binary value 'value' is converted to 5 decimal digits, which are placed in 'text' from the first byte position onwards.

NB: The string parameter should have a minimal length of 5 bytes.

procedure DECBIN (const text : string (*) ; var value : integer);

The text is supposed to contain a sequence of ASCII numeric characters, which represent an unsigned decimal number. This number is delivered as a 16 bit binary value in 'value'.

NB: The conversion will stop at the first non-digit. If no digits are present the value will be zero.

NB: A sign character + or - is regarded as a non digit.

NB: No check for overflow is made during conversion.

OPERATOR COMMUNICATION

When a program is compiled, a specific process is appointed, operator, for the program.

This process is able to output and input strings of ASCII characters. (No parity is provided in input).

Standard operator communication procedures, take care of the actual operations.

OUTPUT:

procedure OPMESS (const text : string (*));

This procedure will output the text in string 'text' on the operator device.

The string is terminated by a 0 (NULL) character or a maximal length of 53 bytes.

NB: If the string is less than 64 bytes, and it does not terminate with a NULL character, some irrelevant text may be output, but this does not destroy anything.

procedure OPSTATUS (const pattern : integer; text : string (*));

This procedure is designed to ease output of status messages, but may be used for other purposes. The 'text' should contain a number of messages separated by a single NULL-character. The 'pattern' is used to select the parts which should be output. A 1 in the corresponding bit-position indicates inclusion. A 0 indicates exclusion.

NB: Two adjecent NULL-characters will be interpreted as an empty string.

INPUT:

procedure OPIN (var text: string (*));

First time this procedure is called, a message is sent to the operator process, indicating that a line may be input to 'text'. The program does not stop to wait for the input. If the procedure is called in the following, without intervening calls of OPWAIT, this has no effects.

NB: The 'text' should be at least 64 bytes long, as this number of characters may be input.

procedure OPWAIT (var length : integer);

If OPIN has not been called, the program will continue. If OPIN has been called, it will suspend program execution until a text has been input. The 'length' parameter indicates how many bytes, that were delivered in the 'text' parameter of OPIN.

function OPTEST : integer;

If OPIN has been called, and a text has been input, this function will take a nonzero value. Otherwise the value will be zero.

Example of use:

A generalized operatorcommunication procedure, which display a given text, a given number and modifies it depending on the answer is:

```
const
questions =  "Q1<0>Q2<0> .... <0>Q16"
:..
...
var

opl,pno,paramvalue : integer;
opstring : string (64);
:
procedure opcom;
  begin
  opstatus (1 shift (16-pno), questions);
  bindec (paramvalue,opstring); opmess (opstring);
  opin(opstring); opwait(opl);
  decbin(opstring,opl);
  paramvalue:= opl
  end;
```