FVN

RCSL : 44 - RT 738
Author : A. P. Ravn
Date : September 1973

MUS SYSTEM SURVEY

Keywords : RC 3600 Software System

MUS SYSTEM OVERVIEW

The Multiprogramming Utility System for the RC 3600 Peripheral System attains the aims:

- to implement parallel processing including interprocess communication and interrupt processing.
- give a strong framework for i/o processing, both on character level and on record oriented level.
- support the user in the operation of the system, which includes easy operator communication, and a basic operating system that takes care of the creation, removal of processes and loading or deletion of programs to core.

These goals have been reached by development of the following software modules:

1.1 A multiprogramming monitor (system supervisor), the design of which rests heavily on the proven design of the RC 4000 multiprogramming system.

1.2 Driver programs for RC 3600 peripheral devices. These lay down the rules which are to be followed in coding drivers for new devices. These rules are purely a matter of overall cleanliness, as no real destination is made between driver programs and ordinary programs.

1.3 Reentrant i/o procedures designed around the zone concept of RC 4000, which has shown itself to be a clean and tidy way to handle device peculiarities, buffering, record formatting and - packing involved in any i/o activity.

1.4 An operating system, which caters for program load and deletion, process creation and removal and start or stop of existing processes. This operating system can receive commands from the human operator.

SYSTEM CONFIGURATION

The different modules have been fitted together in a manner, which gives as few logical dependencies as possible and eliminates non-hierarchical interfaces.

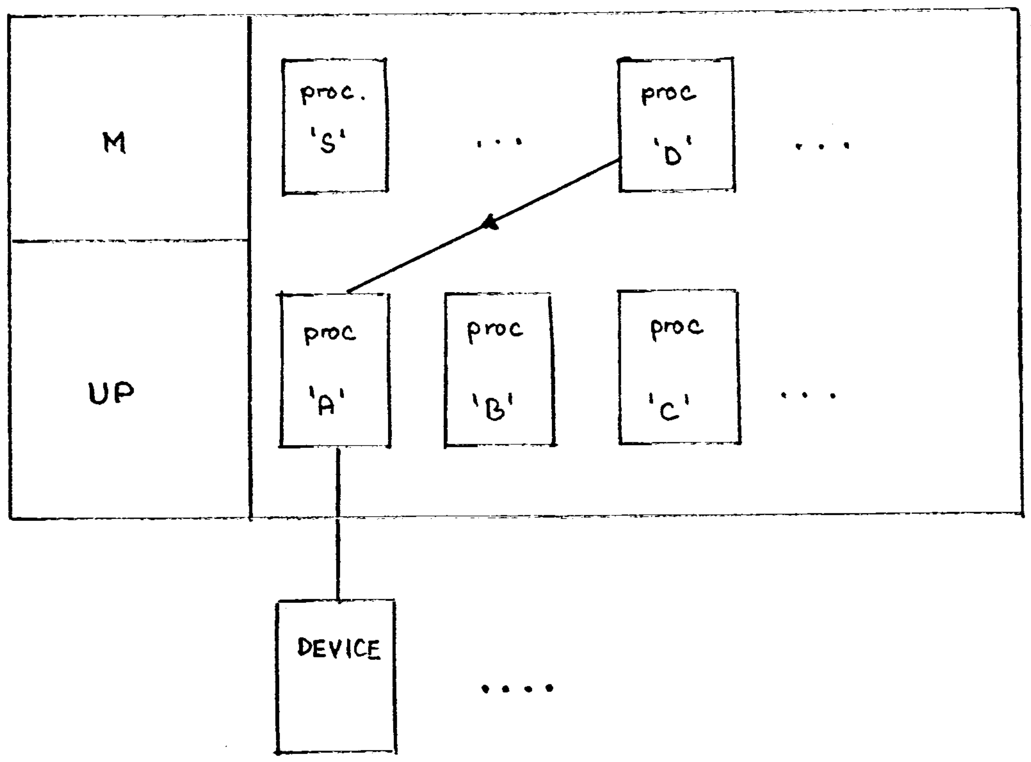


Fig. 1. Logical organization

The core store of the computer contains code for the monitor (M) and the utility procedures (UP). The remaining part of core contains a number of processes (independent programs), which are identified by a process name.

CPU-time is distributed among the processes by the monitor, in such a way that each process is executed independently of the other processes.

A process communicates with an external device through the hardware i/o-instructions and calls of the monitor function 'wait interrupt'. This function delays a process until an interrupt from a specified device arrives.

A process may synchronize its progress with one or more of the other processes through monitor functions, which implement a message, answer strategy. E.g.

The process 'D' calls the monitor function 'send message' indicating 'A' as receiver supplying the core address of the message. After the call the process is allowed to continue until it calls another monitor function e.g. 'wait answer'. In this case it is delayed until process 'A' has returned an answer. The process 'A' will at some point call the monitor function 'wait event', and if a message has been sent to it, it will be allowed to continue. When it has processed the information of the message it returns an answer by means of the monitor function 'send answer'. After the call the process is allowed to continue.

DRIVERS

To handle standard RC 3600 peripheral devices, a number of standard 'driver' processes are provided with the system. These processes may be included or excluded in a particular configuration. The reason for calling them drivers, are that they confirm to certain standards with regard to the messages they will accept and the answers they give.

At the moment drivers exist for the Teletype, Operator Panel, Magnetic Tape Transport, Line Printer, Paper Tape Reader, Paper Tape Punch, Card Reader Station, and some Communication Systems.

JOB CONFIGURATION

Within the framework mentioned above, the system may perform almost any number of activities, that may be wanted. A particular activity as off-line printing, output of cards to a data transmission line, conversion from paper tape media to magnetic tape etc. is called a job. Each job involves one or more processes. The system may be executing several jobs in parallel, only limited by the number of available devices and core storage size of the particular system.

The limitation stems from the fact that a device can only be used by one job at a time for reasons of data integrity. The only exception to this rule is the operator's teletype, which may be shared by several jobs as it displays the identification of the process which currently uses it.

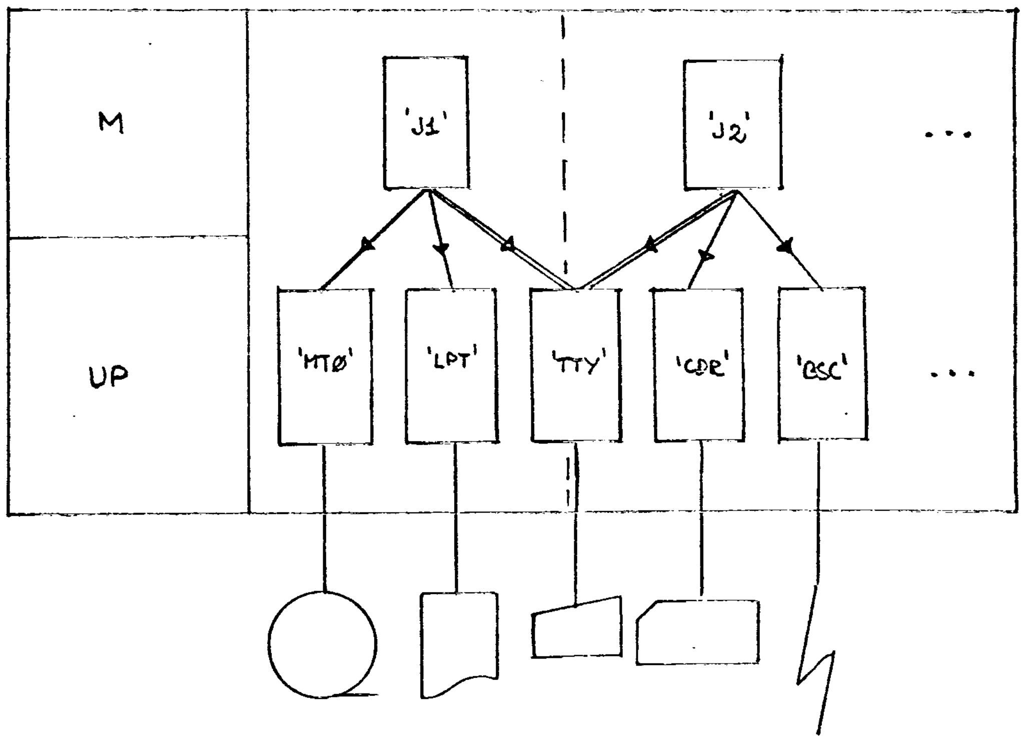


Fig. 2. Job organization (2 jobs)

The system of fig. 2 has two jobs executing in parallel, one is off-line print, the other is transmission of cards. The two jobs share the teletype for operator communication.

CREATION AND REMOVAL OF JOBS

One way of creating a particular configuration of processes would be to load an absolute core image, and then start at a certain address. (This was how a job was created in the old RC 3600 system). This solution has some defects as the core image has to be created manually on some medium, and as it excludes the possibility of creating a new job while another is running. The solution chosen for the new software system is to include a process which by means of operator commands may load processes at any time from an external device, and delete existing processes as needed. This process is called the operating system 's'.

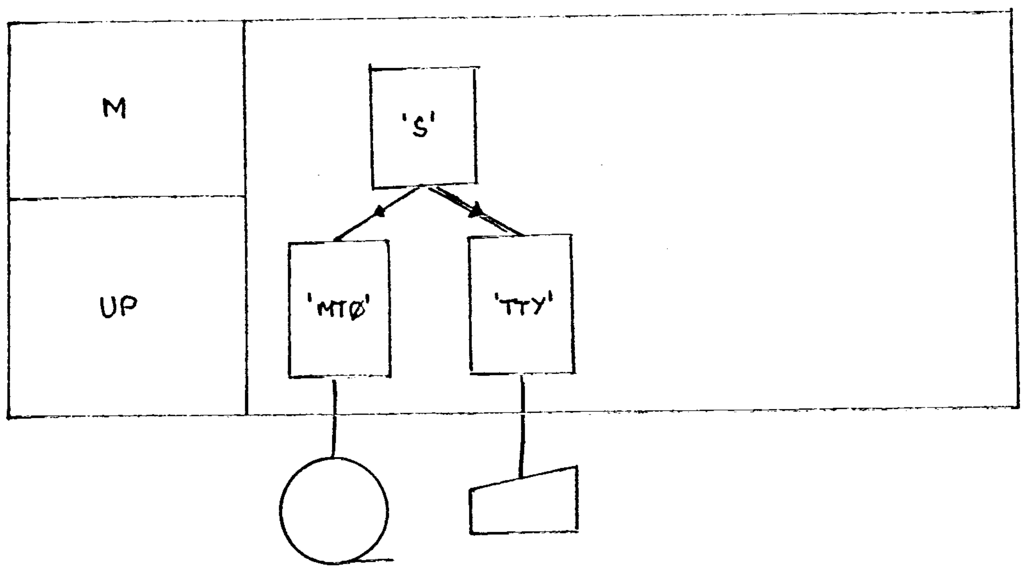


Fig. 3. Basic Conficuation

This solution means that we only have a limited number of basic configurations (absolute core images) which all include the operating system 's', an operator device driver ('OCP', or 'TTY') and a load unit driver ('MT0', 'PTR', or 'CDR').

Whenever the basic system is loaded by means of the RC 3600 AUTOLOAD feature, the operator may at any time instruct the operating system to load or delete the necessary processes to create a new job configuration. To ease this the operating system includes a facility to read and execute a predefined list of commands from the load unit, which includes the necessary job configuration commands.

The price which is paid for this possibility of dynamic job creation and removal is the core used for 's', it amounts to app. 3/4 K words.

When the necessary processes for a particular job configuration are in the system, there is no problem in activating a job, as it probably will be waiting for an operator command to start.

OPERATING SYSTEM

The operating system considers documents mounted on the load unit as consisting of files (for MTT normal files, for PTR and CDR whatever is in the read station). A file consists of identification and contents. A file may be divided into blocks with a maximum blocklength of 80 bytes.

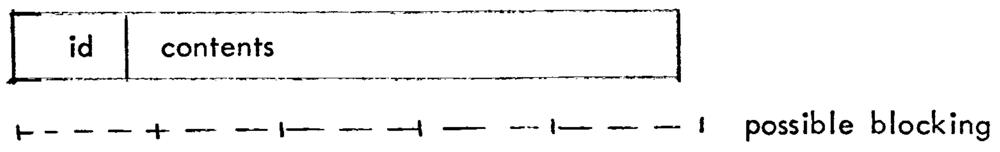


Fig. 4. File on load unit.

The identification may be empty or consist of an ASCII text which does not contain any control characters followed by some control character (CR, NL, SP). Only the first 5 characters are used in the identification, the remaining are skipped.

The contents is either a binary relocatable program in which case it may be preceeded by NULL characters, or it is a command file, which consists of a number of ASCII texts delimited by ASCII control characters, in which case the contents may be preceeded by SPACE characters.

The commands which are accepted by the operating system 's' includes:

INT	$\{\text{"ident"}\}_0^1$	search for file identified by "ident" on load unit, and interpret the contents as commands. The contents should end with the dummy command END, which activates execution of the contents.
KILL	"procident"	delete the identified process from the system.
START	"procident"	include the process as an active process.
STOP	"procident"	exclude the process as an active process.
LOAD	$\{\text{"ident"}\}_0^\infty$	1) if any idents are processes of the system delete the "idents" from the parameter list 2) search for files with the given idents on load unit and load any found as relocatable programs and include these as processes.

Example:

Suppose that a program tape contains the following files:

```
1)      0000
        LOAD LDT MT0 BSC CDR J1 J2
        START J1
        END
2)      0001
        START J2
        END
3)      LPT
        "relocatable LPT driver"
4)      CDR
        "relocatable CDR driver"
5)      BSC
        "transmission driver"
6)      J1
        "program to execute off-line list"
7)      J2
        "program to execute transmission"
```

Then the operator command:

```
        INT 0000
```

will load the system of figure 2 and start the off-line list job, and the command

```
        INT 0001
```

will start the transmission job.

Note that the operating system only uses a driver when it is executing commands, which involve input/output from it. Thus the driver can be utilized by the user jobs for the rest of the time.

REFERENCES

If a greater knowledge is wanted of the main principles of the system refer to:

MUS I manual RCSL: 44 - RT 614

If hard facts are wanted, refer to:

MUS II manual RCSL: 44 - RT 614

MUSIL

When the organization of the MUS system was discussed, the concept of standard processes -drivers and operating system, was defined. To complete the different job configurations a main program - running system - was left to be defined.

If every running system had to be coded in assembler code one would either have to restrict the number of configurations severely to only standard tasks or use - as the development of the old RC 3600 software system has shown - quite a lot of experienced and scarce manpower to code and modify systems. In order to solve or rather ease this problem it was decided to develop a high level language in which running system could easily be defined. The specifications for the design were:

1. The language should utilize the i/o utility procedures of the system.
2. It should contain adequate facilities for handling of structured and unstructured string type data.
3. The powers for numerical computation were allowed to be rather weak.
4. Speed outside the standard procedures were not of great importance, as the running systems were expected to contain only simple loops.

The outer appearance of the resultant language resembles the programming language PASCAL developed by N. Wirth and C.S.Hoare. This is largely a matter of taste, but one should realize that few other existing languages give a framework strong enough to cater for point 2 above. Point 1 and 4 lead to the conclusion, that the code generated should be interpretative instead of directly executable, as this is cheaper in development costs and core storage requirement.

MUSIL PROGRAMS

A program consists of two main parts data and algorithmic specification of how the data are to be treated i.e. statements.

Data are either constants, which do not change during the execution, or variables, which may change. While constants have values, variables have both values and types, which may be structured, i.e. a single variable can be considered a collection of variables.

The statements may either be in the body of the program or belong to a procedure, which is a collection of statements, which can be executed as a single statement simply by writing the name of the procedure.

Commands in programs are enclosed in exclamation marks. Whenever the word name or identifier is used in the following it indicates a string of alfanumeric character, the first always alfabetic.

This should explain the following layout of a program:

```
CONST
          declaration of constant values               this section
                                                       may be ommitted
                         .
                         .
                         .             ;
TYPE
          definition of types, which are to            this section
          be used in declaration of variables          may be omitted
                         .
                         .
                         .
VAR
          declaration of variables                     this section
                         .                             may be omitted
                         .                             (though there would
                         .                             be little sense in this)
PROCEDURE  "ident"
          BEGIN                                        this section
                                                       may be repeated
          procedural statements                        or omitted

          END ;
BEGIN
program statements                                     this section
                                                       should always
                     .                                 be present
                     .
                     .
END
```

Fig. 5. MUSIL PROGRAM SECTIONS

CONSTANTS

Constants are identified by a name which is associated with a value.

"ident" = "value"

The constant declarations are separated by commas.

The value may be:

1. A string, which is enclosed in quotes (" or '). Characters that are not ASCII graphics are denoted by their decimal values enclosed in "<" and ">". E.g.:

   ```
   "VALUE IS: <10><13>"
   ```

2. A numerical integer value, which may be signed or given in a radix different from 10. Value must be within the limits -32768 to 32767.
   E.g.:

   ```
   + 1
   32767
   - 1
   2'10000                 (radix 2)
   8'1777                  (radix 8)
   ```

3. A table, which is another way of giving a string type. It consists of numerical values ($0 \leq$ value ≤ 255) separated by spaces or new lines enclosed in number signs (#) E.g.:

   ```
   # 0   255
     1   254
     2   253 # ,

   # 64  64  78  104 #
   ```

TYPES

Type definitions may either occur in the TYPE part, if it is convenient to name a type separately, or they may appear with the declaration of the variables, which they describe. A type definition has the form

"ident" = "type" ;

The type might be:

1. Simple, that is <u>integer</u> or <u>string ("length")</u> where "length" is a numerical value which gives the length of the string in bytes.
2. Structured, where it may describe either a <u>record</u> or a <u>file</u> .

2.1 A <u>record</u> is a string, which contains named substrings, called the <u>fields</u> of the record.

```
RECORD
    .
    .
    .
"field ident", "field ident" ... : "field type" ;
    .
    .
    .
"field ident" : "field type" FROM "position" ;
    .
    .
    .
END
```

The field type must be simple. The first form of field specification denotes consecutive fields of identical type. The second form denotes a field of the given type starting at a specific position (byte) within the record.

```
RECORD
ccw    :  string (1) ;
line   :  string (132) ;
select:   string (1) from 2 ;
sline  :  string (132) from 3
END
```

References to field of a variable "v " of record type are denoted by:

```
"v ". "field ident"
```

2.2 A <u>file</u> is a type which describes a document on some external device.

```
FILE
  "file descriptor"
OF "filerecord type"
```

The file description part contains the name of the process which handles the device, a kind which specifies how it should be handled, a maximal blocksize, the number of buffers to be used and the standard record format of the document.

Furthermore it may describe a conversion table, which is a string identifier.

And it may also specify a procedure which should be activated in case of a specified set of i/o errors.

The filerecord type is a definition of the structure of the records of the file.

Example:

```
        FILE
        'LPT' ,     !device!
        2'10 ,      !kind!
           2 ,      !buffers!
        133 ,       !blocksize maximum!
           U ;      !rec format!
        CONV LPTTABLE ; !conversion table identifier!
        GIVEUP LPTERROR, 8'17666     !error proc and set of i/o errors!
        OF RECORD
              ccw : string (1) ;
              line : string (132)
              END
```

A variable of type file "f" may be used as parameter to standard i/o procedures. The file descriptor (which corresponds to a zone descriptor in the MUS system) is organized as a record with certain standard fields which may be accessed as:

"f". "fieldident"

When a input or output standard procedure has been activated, the file contains a current file component:

"f"↑

i.e. a pointer to the current file record.

Fields of the current file component are then accessible as normal records by:

"f"↑. "field ident"

VARIABLES

Declarations of variables have the format:

"varident" , "varident" , ... : "type" ;

Each declaration creates as many instances of each variable of the given type as there are identifiers.

STATEMENTS

The statements should be self explanatory to anybody with some knowledge of ALGOL. We will therefore only explain the available control statements:

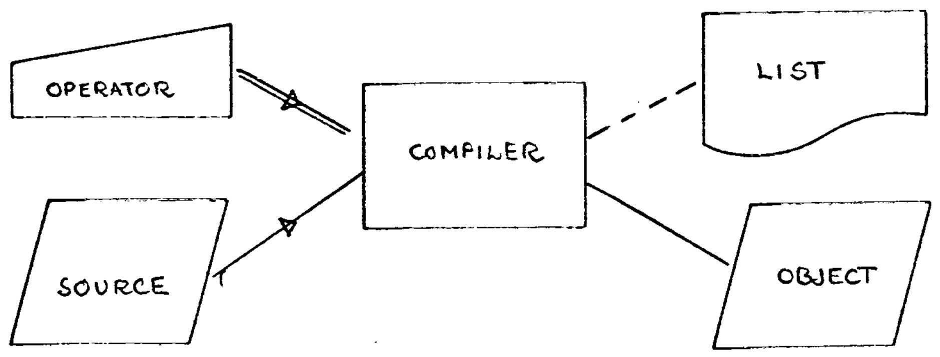
1. IF "relation" THEN "statement"
   indicates conditional execution of the statement.
2. WHILE "relation" DO "statement"
   indicates the repeated execution of the statement, while the relation is true. The statement is skipped totally if the relation is initially false.
3. REPEAT "statement" ; .... UNTIL "relation"
   indicates repetition of the statements until relation becomes true. The statements are executed at least once.
4. GOTO "number"
   indicates an unconditional transfer to a statement labelled by "number"
5. BEGIN "statement" ; .... END

STANDARD PROCEDURES

The compiler recognizes a number of standard procedures, which provides for i/o handling, operator communication, and special string handling.

COMPILER

The compiler runs as a job under the system. The compilation requires specification of a source document (in ASCII alphabet), a object document, an operator device, and possibly a list document.



The object code is only output if no errors are detected by the compiler.

EXAMPLE OF A PROGRAM LISTING

```
MUSIL COMPILER

!  MUS: PAPER TAPE TO PRINTER SYSTEM                          !

CONST
PROGRNAME= 'PTR TO LINEPRINTER<10>',
LPTTABLE= # 0 0 0 0 0 0 0 0 0 0 10 0 12 13 0 0
            0 0 0 0 0 0 0 0 0 32 0 0 0 0 0 0
            33 34 35 36 37 38 39 40 41 42 43 44
            45 46 47 48 49 50 51 52 53 54 55 56 57 58 59
            60 61 62 63 64 65 66 67 68 69 70 71 72 73 74
            75 76 77 78 79 80 81 82 83 84 85 86 87 88 89
            90 91 92 93 94 95 96 65 66 67 68 69 70 71 72 73
            74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89
            90 00 00 00 00 00 00 00
          # ,
STATUS= "DISCONNECTED<10><0>OFFLINE<10><0><0><0><0><0><0>
EOF<10><0>B8<10><0>B9<10><0>PARITY<10><0>EM<10><0><0><0><0><0>";

VAR
LPT: FILE
     'LPT',1,1,50,U;
     GIVEUP LPTERRORS, 2'1100001111111111;
     CONV LPTTABLE
     OF STRING(50);
PTR: FILE
     'PTR',1,1,50,U;
     GIVEUP PTRERRORS, 2'0000001111111111
     OF STRING(50);
D:   INTEGER;
DUMMY: STRING(10);

PROCEDURE LPTERRORS;
     BEGIN
       OPSTATUS(LPT,Z0,STATUS);
       OPIN(DUMMY); OPWAIT(D);
       REPEATSHARE(LPT)
     END;

PROCEDURE PTRERRORS;
     BEGIN
       OPSTATUS(PTR,Z0,STATUS);
       OPIN(DUMMY); OPWAIT(D)
     END;

BEGIN

0:   OPMESS(PROGRNAME);
     OPEN(LPT,3);
     OPEN(PTR,9);

   REPEAT
     GETREC(PTR,D);
     PUTREC(LPT,D);
     MOVE(PTR↑,0,LPT↑,0,D)
     UNTIL D<50;
     CLOSE(PTR,1); CLOSE(LPT,1);
   GOTO 0
END;000000  ERRORS
```

REFERENCES

A full description of the language, including the standard procedures are found in:

MUSIL REFERENCE MANUAL	RCSL	(in print)

Operation of the compiler, and explanation of errorcodes are found in:

MUSIL COMPILER OPERATION	RCSL	(in print)