A -

γ

BNL-35133

Contributed paper to the Europhysics Conference on
"Software Engineering, Methods and Tools in Computational Physics"
Brussels, Belgium, August 21-24, 1984

CONF-8408109--1

THE BNL MULTIPARTICLE SPECTROMETER SOFTWARE[*]

BNL--35133

DE85 001347

by

Alfred C. Saulys
Department of Physics
Brookhaven National Laboratory
Upton, New York, U.S.A. 11973

ABSTRACT

This paper discusses some solutions to problems common to the design, management and maintenance of a large high energy physics spectrometer software system. The experience of dealing with a large, complex program and the necessity of having the program controlled by various people at different levels of computer experience has led us to design a program control structure of mnemonic and self-explanatory nature. The use of this control language in both "on-line" and "off-line" operation of the program will be discussed. The solution of structuring a large program for modularity so that substantial changes to the program can be made easily for a wide variety of high energy physics experiments is discussed. Specialized tools for this type of large program management are also discussed.

1. INTRODUCTION

The trend in high energy physics has been toward larger and more complex experiments which require large and complex software systems for monitoring and data analysis from these experiments. The sheer size of

[*] This research was supported by the U.S. Department of Energy under Contract No. DE-AC02-76CH00016.

[illegible]

some of these programs can sometimes overwhelm the often limited staff of physicists needed to run these experiments. Even in the case where the manpower is not a basic limitation, the problems of putting together contributions from several people and making it a well-functioning, easily-maintained software system can overwhelm any experimental group, unless new tools and computer techniques are adopted by the physicists. In this paper I will attempt to describe some of the solutions to some of the problems encountered in the development of the MPS software. Our solutions may not all be original or unique, but they may be of interest to others. The discussion that follows is based on the contributions from several members of the MPS group [1]. The MPS program is coded in Fortran IV, except for a few computer-dependent machine-language routines.

2. PROGRAM STRUCTURE

The BNL Multiparticle Spectrometer (MPS) data-reduction program is well-suited for structuring as a set of tasks performed serially. This is probably true for most large high-energy physics experiments. Most spectrometers produce information from the various detectors in a form of encoded strings of bits as input and the task of the program is to produce a table of numbers summarizing the physics results as output, after going through a long chain of calculations. I don't mean by this that this type of program would not be well-suited for computers with parallel architecture when they come into existence.

This chain usually consists of a series of links of tasks to be performed on the data. An example of a task would be the unpacking of data from the various detectors and converting it to coordinates in the lab. Another task would be to recognize a pattern in the data from the detector which constitutes the sought-after signal which is selected from the detector background. As one continues through the analysis chain, other obvious well-defined tasks come up.

These types of analysis requirements lead to obvious program modules for each detector or group of detectors. But as experimental requirements change, so do the requirements for different types of detectors. In order to facilitate easy changes in the software from experiment to experiment, the modularization by detector is also desirable. These requirements have led us to modularization of our software in a matrix form. The tasks for each device form the matrix elements.

We have implemented this scheme in what we call the ROUTE structure. Each task is assigned a number code, and each detector (or device) is assigned a bit position in a word. This assignment occurs automatically at program initialization using a master list of devices. For convenience, we have extended the definition of a device in our scheme to some well-defined tasks as well, e.g. pattern recognition. The connection of each device to the analysis chain is by a

```
CALL ROUTE (I,DEV)
```

where I is the task number, and DEV is a string of bits, where each bit position signifies a particular device. Some of the task number definitions are shown in Table 1.

The subroutine ROUTE itself is nothing but a switchyard for "dispatching" routines located in each device-dependent module of code. Typically, two lines of code look like:

```
IF((DEV.AND.IRDRI).NE.0)CALL DISDRI(I)
IF((DEV.AND.IRPWC).NE.0)CALL DISPWC(I)
```

Subroutine DISDRI is the dispatching routine for drift chambers and DISPWC is the dispatching routine for proportional wire chambers. The dispatching routines respond to the value of the task number, I, to call the routines within the separate modules of code to perform the task itself. The connection of the device-dependent code is through the dispatching routines only. Therefore the addition or removal of device-dependent code is very simple. Also, on the dynamic scale, the selection of devices to perform a particular task is a matter of setting a bit mask. That is, the argument DEV in the call to ROUTE can select different devices on each call by turning the corresponding bits on or off.

3. CONTROL LANGUAGES

Another problem faced by most large high-energy physics programs is the ability to select and control the execution of the various tasks in the data analysis chain by a substantial number of different people who may have a limited knowledge of the internal structure of the program. This means the adoption of an easy-to-use, yet flexible, control language.

We have adopted a mnemonic character-oriented series of commands to specify the execution of a well-defined task in the program. Our acronym for this control language is COMAC. It recognizes a set of commands, each of which can be up to 5 characters long and may be abbreviated to the first 3 or 4 letters if there is no ambiguity. Each command branches to a well-defined task which may need an additional set or sets of commands to select a particular set of branches to subtasks.

A sample dialogue is shown in Figure 1. The operator replies to the computer prompts are shown in boxes (except for two blank responses). In Figure 1 on the first line the program prompts the operator for the next command. He replies with a request for his options by typing OPT. The command OPT or an unknown command at any level in the command string will list the options available to the operator. On the next two lines the computer answers with the list of recognized commands and comes back with a prompt for the next command. Note that in Figure 1 the list of commands contains only the characters against which the commands are tested by the program. Longer commands can be typed for mnemonic value, but only up to first 5 characters are tested. Unfortunately, at our current level of implementation, these lists are the actual mnemonic commands without any further explanation. This requires the operator to have some familiarity with functional meaning of these commands.

In this example, the operator wishes to assign the physical device LPP as his printer. He therefore requests for the ASSIGN task (with options) and continues with the dialogue until the device LPP is assigned. He then supplies two blank commands to return to the request for the next command. A blank response to a prompt for input signifies the preservation of the current value of a parameter and continuation to the next prompt. At the termination of any command level a blank response returns to the previous level. A blank response at the COMMAND level continues the execution of the analysis program.

A series of commands to subtasks can be strung out on one line separated by commas. This is shown in the last box in Figure 1. The semicolon termination (;) prompts the program to return for a request of a new command for a new task. A termination with a colon (:) ends the command string and initiates program execution. This feature of COMAC, allowing command input as a string of commands, leads to its easy implementation in the "off-line" batch mode of operation of the program. Each command string becomes an input data line which is read before the execution of the program is initiated.

The program is initialized with a default set of parameters. Therefore a dialogue with the program is necessary only if a change is needed. Very commonly used strings may be assigned a synonym equivalent of the string. The program compares all commands shorter than three characters to a table of synonym strings. For example, to suppress an error message, the command ES is equivalent to supplying the following string of commands:

```
ONOFF,FAC,ERRMESS,SUPR
```

4. SOFTWARE MANAGEMENT TOOLS

The complexity, size, and the necessary modularity of programs for high energy physics experiments produce management problems not unlike encountered by systems programmers. These large software systems require constant modification of some parts by any of a number of people in a group without producing chaos in the system as a whole. This requires adequate management tools to monitor the system as a whole. Considering the scope of this paper, I can only mention a few that our group has developed.

FORTRAN code leads to easy modularity if some self-discipline is adhered to, but it can lead to rather huge volumes of undocumented code if it is not. We have developed character manipulation code which will scan the source code for comments immediately after the subroutine statement and tabulate alphabetically all subroutine names with their corresponding comments. It will also indicate the code module name where the subroutine resides. We call this documentation program SUCOM. If the coder of the subroutine has briefly documented the purpose and function of his code, then we can obtain an up-to-date documentation of all our software libraries by running all the sources through SUCOM. In this way we reduce the amount of work necessary to obtain up-to-date documentation of our libraries.

FORTRAN code also allows the labeled common data storage structure. Again, this helps modularity, but it can produce chaos in allocating the variable positions and names within the common block. We have adopted a scheme based on a feature of the CDC UPDATE utility [2] for program library maintenance. This feature allows the separation of the code and the definition of the [illegible] of labeled commons into two different files. The common block [illegible] de is replaced by a statement

*CALL NAME

where NAME is the labeled common name. The contents of the common blocks are kept in a separate file with each one preceded by statement

*COMDECK NAME.

At the time of compilation, the two files are merged and the *CALL statements are replaced by the actual contents of the common block. The advantage of this scheme is that the common block is defined in only one place and yet it can be used by many subroutines. Of course, this imposes the rigidity of using specific variable names as defined in the common block when writing the code, but it is a small price to pay for the flexibility in making changes in the code. To those of you using CDC software this probably is a familiar scheme, but we felt that it has a great enough advantage that we have implemented this on two other computers: DEC PDP-10 and DEC VAX 11/780. Commercially developed code management systems [3] are also currently becoming available that have this COMDECK feature. They have been also implemented on several different computers.

5. CONCLUSIONS

In this brief paper I have discussed some of the solutions to problems raised in the design, management and operation of a large high-energy physics spectrometer software system. In particular, I have discussed the ROUTE matrix modularization scheme, the COMAC control language, the SUCOM documentation program and the COMDECK labeled common block management system as implemented in the MPS software system.

TABLE 1

Some examples of task definitions

I = 1	Read-in program parameters; called by PARFRW(CONTRO)
I = 2	Write-out same file read in under 1; called by PARFRW(CONTRO)
I = 3	Initialization; called by CONRUN(CONTRO)
I = 4	Unpack and store coordinates; called by CONVNT(CONTRO)
I = 5	Local pattern recognition; called by CONVNT(CONTRO)
I = 6	Print the coordinates for this event; called by EDUMP(FACLIB)
I = 7	Print pattern recog. results for this event; called by EDUMP(FACLIB)
I = 8	Print totals of error conditions; called by OUTPUT(CONTRO)
I = 9	Print other totals and statistics; called by OUTPUT
I = 10	Print parameters; called by OUTPUT

REFERENCES

1. A. Etkin, K.J. Foley, R.S. Longacre, W.A. Love, T.W. Morris, E.D. Platner, A.C. Saulys (BNL); S.J. Lindenbaum (BNL/CCNY); C.S. Chan, M.A. Kramer, J. Piekarz (CCNY).
2. UPDATE 1 Reference Manual, CDC Publication No. 60449900.
3. HISTORIAN PLUS by OPCODE, Inc. CMS and MMS by DEC.

FIGURE CAPTION

Fig. 1 COMAC dialogue with on-line program. Operator replies are shown in boxes.

```
COMMAND= OPT

COMMANDS ARE  ASSIG, DDT  , ERASE, LOOP , OPT  , PARAM. PATH ,
 PLOT , PROMP, ONOFF, ALTER, SPACE, OUTPU, STOP ,
COMMAND= ASSIGN

I/O DEVICE OPT

DEVICES ARE PRINT, PLOTR, EVENT, OUTAP,
I/O DEVICE PRINTER

PRINTER IS LPT   LPP

PRINTER IS LPP

I/O DEVICE

COMMAND= ASSI,PRINT,LPP;

COMMAND=
```