DESIGN PRINCIPLES OF THE HIGH-LEVEL CONTROLS STRUCTURE FOR THE CERN SPS

W.F. Herr, R.J. Lauckner and C.G. Saltmarsh*
SPS Division, CERN, CH-1211 GENEVA 23, Switzerland

*) Now at SSC, Central Design Group, LBL, Berkeley, CA, U.S.A.

ABSTRACT

A high level infrastructure is under development for the CERN SPS control system as demands on the accelerator become more complex. This paper discusses the principal elements driving the design and development work. These principles are derived in large part from experience with the SPS in its previous operational modes [1], and poses some challenging problems for computer science and operational implementation.

Application programs for the control system should be as simple and universal as possible and they should only deal with algorithms and machine physics. We will describe a software infrastructure where issues like control flow, data flow, screen management, system resources etc. are dealt with by a software manager. Such a framework will simplify the task of the operator and the programmer and allow the development of more flexible software to cope with the increasing complexity of the accelerator in the future.

INTRODUCTION

When the SPS was commissioned in the mid 70's little attempt was made to design the control software in a managed and integrated way. This was partly due to the lack of a basic infrastructure for the development and management of programs. Application programs were written in NODAL [2], a BASIC like interpreter with an built in interface to the network. Accelerator specific features such as the scheduling of processes, linked to the accelerator timing and an interface to the controls network are part of the language and NODAL was the only infrastructure for the program development.

In this environment most application programs were developed as standalone programs by the equipment specialists independently and serious limitations of the system became visible as the time went on. It became increasingly more difficult to improve and develop the control software to keep pace with the increasing demands on the machine for operational flexibility and efficiency. Moreover the maintenance of the existing software was complicated by the diverse nature of the programs, a typical code module contains graphics, machine physics algorithms and operator interaction portions.

The operation of the SPS has now become more complex (p$\bar{p}$ collider, pulsed collider, heavy ions) and since 1987 the SPS works in a multicycling mode with one hadron cycle and two lepton cycles. The different cycles have to be controlled independently, but many algorithms and programs for the control are the same for all these elementary cycles although they have to work in a different environment and with different data.

Another issue is the much heavier use of compiled languages (FORTRAN 77, C) in the future and the use of distributed personal workstations (APOLLO DN3000 and IBM PC/AT) as consoles and for software development. There is an urgent need for an infrastructure to organise the software development and its use in the control system of the SPS.

THE BASIC CONCEPT

The traditional approach to structure a program into functional units is the use of subroutines as illustrated in Fig.1. A main program (m) calls one or more subroutines (a, b) to perform a specific action or operation and the control is given back to the calling program after the program module has finished. This approach has some serious disadvantages. Although the program modules (subroutines) are logically independent they are nevertheless tightly coupled together because the control flow and the data flow are coded inside the modules.

![Diagram](attachment:image.png)

Fig.1: Classical Approach.)
In addition the modules interface directly to the operating system to establish the control and data flow and to access system resources such as files, terminals, screens, keyboard etc. This reduces the portability of the programs significantly. Finally, all those routines coupled together to perform an operation are usually written in the same programming language to simplify the development and maintenance.

Our approach is sketched in Fig.2. The program in the previous example is split into a set of programs which is collectively referred to as a "task" [3] in the following chapters. These modules are logically grouped together to perform a well defined action.

We provide a software frame where all modules perform their operations as independent processes. The supervision of these processes is carried out by another process which we will call the "Run Time Coordinator" (RTC). The control and data flow is now removed from the application modules and organised by the Run Time Coordinator. As illustrated in Fig.2 the vertical lines connecting the modules have been removed, processes are freed from the formal embedded interconnections leaving only the logical relationships between them and the interface to the operating system is now through the Run Time Coordinator. The application programs can now be written independently from the operating system. Another feature of this approach is that these modules can be written in any language or they can be command files and they can be run in different host computers if necessary. The Run Time Coordinator will maintain the relationship between the modules across the network. Such a standard data structure implemented in several programming languages [4].

It should be mentioned at this point that the Run Time Coordinator can handle more than one task at the time.

**THE RUN TIME COORDINATOR**

**The task specification**

Having written the necessary functional units (process modules) the programmer has to group these together to form a task and to pass instructions to the Run Time Coordinator on how the execution of these modules has to be performed. For this purpose the operator or programmer has to provide a task specification which is "compiled" [5] into a format which can be understood and executed by the Run Time Coordinator (Fig.3). Note in this figure that the operator i/o is through the RTC. An example for such a task specification is shown in the appendix. A complete task therefore consists of one or more process modules and a task specification.

The task specification is written in a "Task Control Language" (TCL) and it was one of our aims that this language can be easily understood and allows the manipulation of a task specification by non specialists. This increases the flexibility of the system. We have tried to use comprehensible English for this control language to achieve good readability and to serve as documentation.

**Services provided by the Run Time Coordinator**

Apart from providing a uniform and homogeneous environment to the task the services offered by the Run Time Coordinator are:
Implementation of control flow of a multi-process task in a single host or across the network.
Implementation of the data flow between the processes of a task in a single host or across the network.
Interfacing the processes to the operating system.
Implementation of communication between operator and process.
Logging of actions and faults.
Passing environment variables to the processes.

Control flow

The services provided by the Run Time Coordinator to establish the control flow between the processes of a task are:

- Execute a process immediately
- Execute processes in sequence
- Execute processes on condition (e.g. existence of data, process status)
- Repeat a process for a specified amount of time
- Repeat a process a specified number of times
- Repeat until a condition becomes true

Note that we have implemented services which are specific to the use of the Run Time Coordinator in the environment of an accelerator.

- Execute on accelerator timing
- Repeated execution

Data flow

A standard data interface has been developed [4] to exchange data between processes of a task or between tasks and for the transfer across the network. Data required and produced by a process has to be declared to the Run Time Coordinator in the task specification and the creation and the use of the data will be monitored by the RTC. This enables the RTC to establish the data flow between the different modules. The RTC can check the validity of existing data and can make sure that required data is available when the process is executed. If required the RTC itself can send data across the network if it is available in the standard data format.

The use of system resources

The use of system resources like screens, disk files, mouse, terminals etc. is organised by the RTC and possible contention is avoided.

Logging of actions and faults

All operator actions are logged by the Run Time Coordinator to allow problem tracing. Errors and faults are recognised and if possible interpreted by the system. A fault handler for a limited number of faults is part of the RTC. All errors and faults during the execution of a task are logged.

CONCLUSION

We have described a software concept which can be used as a framework for the development and the application of programs where we put some emphasis on its use in the control system of an accelerator like the SPS.

This framework consists of two principal components. The Run Time Coordinator (RTC), which provides the environment, the interface to the operating system and the run time organisation of the software and a Task Control Language (TCL) which has been developed to specify the connections between the process modules to the RTC.

The use of such a framework has several advantages such as:

- encourage people to write ubiquitous, exchangeable program modules rather than big monolithic and inflexible software.
- accelerator specific tools are provided and available outside the application programs.
- open software architecture using standard data structures.
- existing software is more adaptable to changes of the accelerator hardware and to new requirements for the accelerator.
- the portability of application programs is increased since control and data flow is treated externally and the interface to the operating system is through the RTC.
- it simplifies the development of control software by large groups of people and by non-specialists.
- control and data flow are described and documented outside the program in the task specification.

REFERENCES

APPENDIX

This task listed below will read the injection intensity every cycle and displays it on the screen. It consists of three independent modules: the process "init" initialises the data gathering from the accelerator, the process "fetch" starts the data acquisition and creates a standard data set [4] with the name "troll" every cycle when the accelerator event "injection" occurs. The process "plot" is our standard plotting facility which reads the data from "troll" and plots it. The module "init" is written in FORTRAN 77 while all other modules are written in "C". The executable files to be used are specified as a string to the keyword exec file and the keyword host specifies the computer on which the process is to be run. In case a module expects command line arguments they can be passed as comlineargs and if operator interaction with the module is desired, the option inter type (interactive type) has to be set to "full". These two declarations are optional and set to "no commandline arguments" and "not interactive" as a default if they are omitted in the task specification. The token exec init means that "init" should be executed immediately and without special conditions. The process fetch will be started after "init" has finished and every time the timing event "injection" occurs (repeat exec fetch when init finished and event injection) while "plot" has to be started when the data element "troll" has been produced (repeat exec plot on troll arrives).

These tokens and keywords are parsed by the task specification compiler [4] to produce a data set which drives the RTC to accomplish the synchronisation between the modules. It should be mentioned that the name of this data element "troll" appears only in this task specification.