Real-Time Scheduling of Software Tasks*

L.T. Hoff
Brookhaven National Laboratory
Upton, NY, 11973-5000, USA

Abstract

When designing real-time systems, it is often desirable to schedule execution of software tasks based on the occurrence of events. The events may be clock ticks, interrupts from a hardware device, or software signals from other software tasks. If the nature of the events is well understood, this scheduling is normally a static part of the system design. If the nature of the events is not completely understood, or is expected to change over time, it may be necessary to provide a mechanism for adjusting the scheduling of the software tasks.

At the Relativistic Heavy Ion Collider (RHIC) front-end computers (FECs) provide such a mechanism. The goals in designing this mechanism were to be as independent as possible of the underlying operating system, to allow for future expansion of the mechanism to handle new types of events and to allow easy configuration. Some things taken into consideration during the design were the programming paradigm (object oriented versus procedural), programming language and whether events are merely interesting moments in time or intrinsically have data associated with them. The design also needed to address performance and robustness tradeoffs involving shared-task contexts, task priorities and use of interrupt service routine (ISR) contexts versus task contexts. This paper will explore these considerations and tradeoffs.

1. Introduction

The RHIC control system adheres to the standard model for accelerator control systems[1]. In this model, "front end computers" (FECs) provide access to accelerator equipment and generate reports to be sent to operator consoles. In addition to these tasks, FECs are expected to perform certain routine functions, such as periodically checking readings against range limits, logging data and closing local control loops in discrete steps. These functions are typically scheduled-based on the occurrence of physical (real-world) events. The event may be an interrupt from the accelerator equipment indicating that new data is available, or it may be a clock tick indicating the completion of a time interval, or it may be a software event indicating the completion of the execution of a software task. RHIC FECs use a real-time O/S (VxWorks) specifically so that FEC functions can be scheduled deterministically with respect to these real-world events.

2. FEC software

RHIC FEC software has been designed to allow rapid reconfiguration using a concept called an Accelerator Device Object (ADO) [2]. The "core" FEC software is identical in all FECs. Software objects known as ADOs are created within each FEC, establishing each FEC's unique mission. The ADO contains the methods for access to specific accelerator equipment and for performing range checking, closing control loops, etc. on specific accelerator equipment. As new accelerator equipment is added to the system, or as existing equipment or needs change, new or modified ADOs can be created, without changing other FEC software, or otherwise affecting FEC running.

In keeping with the modular ADO design, a flexible, configurable and extensible system for scheduling FEC functions with respect to real-world events was considered to be necessary. The ADO provides methods which know "how" to perform a function, but the event system determines "when" to perform the function. A flexible scheduling system allows for the possibility of tuning the speed of control loops, or changing the point in the accelerator cycle when data is read from accelerator equipment.

---

* Work performed under the auspices of the U.S. Department of Energy

Using an object-oriented design approach similar to the approach used to design the ADO system, real-world events are represented by a software abstraction called an "fec event". The following requirements were established for the "fec event" software object; it:

- must be able to represent the following real-world events :
    - millisecond clock tick
    - interrupt from accelerator equipment
    - RHIC Event-Link (timing system) event
    - "software" event

- must be extensible to represent other real-world events

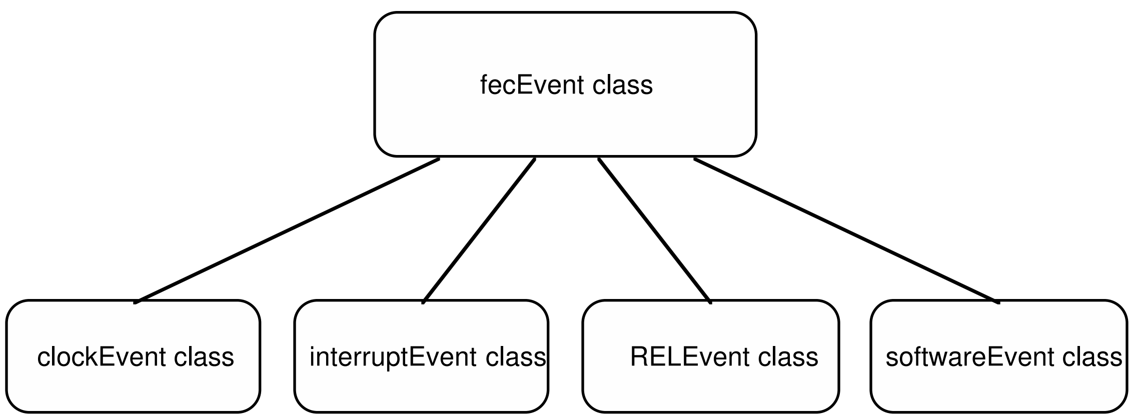
- must be able to schedule ADO and general FEC functions; an environment must provided for running any C function or C++ class member function

- the relationship between real-world events and the ADO or FEC function which is executed in response to that event must be reconfigurable without otherwise disrupting FEC operation

- must be able to abstractly represent real-world events in such a way that the idiosyncrasies of the source of the real-world event and the operating system are well hidden within the object

3. Design

The fundamental design was developed quickly. The "fec event" class would be implemented in C++ (the only supported object-oriented programming language). An inheritance tree would be developed, with a single root or base class. This base class would handle the establishment of the relationship between real-world events and C++ class member functions and be responsible for executing the functions. Derived from this base class would be specific classes representing various types of real-world events. These specific classes would be responsible for sensing the occurrence of their particular real-world event and establishing an environment in which to execute a C++ class member function when that event occurs. The idiosyncrasies as to how the event is sensed and how the environment is established would be encapsulated within these specialized objects.



4. Tradeoffs

Although the fundamental design was developed quickly, it soon became apparent that the details of sensing each real-world event, and especially the details of establishing an executing environment, involved many tradeoffs.

The first consideration was how to execute both C functions and C++ class member functions. The event system was to be used to schedule basic FEC functions (coded in C) as well as ADO functions (coded in C++). Executing a C function from a C++ object is straightforward, using the address of the function. Executing a C++

class member function from an object of a different class is another matter. Traditionally, this problem has been approached using one of two different paradigms.

The first approach is to define a special C++ class, known as a functor[3]. This class has a single pure virtual function, which derived classes are expected to override to perform the desired function. There are several drawbacks to this approach. Firstly, since C++ classes may be part of another inheritance hierarchy as well as the functor hierarchy, it typically requires language support for multiple inheritance. Secondly, it only allows for a single function to be executed per object. It was envisioned that ADOs might have a collection of functions to execute for different real-world events. Finally, this approach has no C language analog, precluding scheduling FEC functions coded in C.

The second approach is to execute only static C++ class member functions. Static C++ class member functions are essentially C functions, so there is a direct C language analog. Furthermore, there is no limit to the number of static C++ class member functions per object. However, static C++ class member functions cannot access non-static class member data, nor can a static C++ class member function be virtual[4]. The latter limitation was not viewed as terribly severe for ADO functions, but the former would prevent ADOs from performing any tasks whose operation depended on data specific to that ADO. A simple workaround for this problem is to pass an object pointer as a parameter to the static C++ class member function.

This last design was adopted for the event system. The base event class was designed to store a list of function pointers (either C functions, or static C++ class member functions) and a list of parameters (either a pointer to a C++ object, or any arbitrary value for a C function). Whenever called upon to execute a function, the corresponding parameter would be passed to the function.

The next debate revolved around whether events are merely interesting moments in time or whether they intrinsically have data associated with them. Some schools of thought maintain that events have data associated with them[5]. The data is related in some way to the root cause of the event. If it is desirable for the event system to hide the root cause of the events, so that one event type can be transparently substituted for another event type, then a dataless design is more appropriate. The drawback of a dataless approach is that there is no straightforward method to provide detailed information about the purpose of executing a function. The executed function must be coded to assume that it is appropriate at the appropriate point in time. The dataless design was chosen for the RHIC event system, though provisions were put in place to easily revise this decision if it proves to be short-sighted.

Each specific "fec event" class is responsible for sensing the occurrence of its particular real-world event. The mechanisms for doing so may depend on the underlying operating system as well as the nature of the real-world event. In traditional timeshare operating systems, such as UNIX, real-world events are typically sensed via device drivers, which execute in "kernel space". In typical real-time operating systems events may be sensed more directly, e.g. a "user space" task may attach interrupt service routines (ISR) directly into the CPU's interrupt vector table.

Each specific "fec event" class must establish an environment in which to execute C or C++ functions. If the operating system uses a heavy-weight process paradigm, where each process has its own protected data space, and all event objects exist within a single process, then a valid executing environment may be provided by the operating system.

RHIC FECs use a real-time operating system which uses a light-weight task paradigm. Data is shared between all tasks including ISRs. Each task is assigned a priority, and the highest priority "ready" task preempts any other ready tasks (priority preemptive multitasking). Using such an operating system, one has latitude in choosing how many tasks are used as executing environments, what the priorities of the tasks are and even whether or not ISRs constitute valid executing environments. In general, the tradeoffs involve performance, robustness, and the need for consistency between executing environments for different event types.

The advantage of using an ISR as an executing environment is a minimum overhead and therefore maximum response speed. The drawbacks, however, are numerous. The executed functions must be ISR-compatible, i.e. they must not use certain operating system functions, they must be fairly short in duration (since interrupts are typically disabled during ISR processing) and they must not contain any bugs which might cause the

ISR to crash, bringing down the entire CPU. For these reasons, the RHIC event system always uses a task context as an executing environment.

Each specific "fec event" class is responsible for setting up one or more task contexts in which to execute the C or C++ functions. The choice of the number of available task contexts trades off overhead versus robustness. C or C++ functions which share task contexts may affect each other's execution. This effect becomes most severe if one function has a bug which halts the execution of the task or merely takes an inordinate amount of time to complete. Using separate tasks for each function avoids these problems at the expense of additional overhead. Most specific "fec event" classes choose a middle ground with a small set of task contexts sharing the load.

Under a priority-preemptive operating system, priorities assigned to the task contexts must also be considered. Events which need more prompt attention, such as interrupt events, should use higher task priorities than events such as clocks with a 1 second tick rate. The RHIC interrupt event class uses the highest task priority. The event-link (timing system) event class uses tasks of two lower priorities than the interrupt event. The clock event class uses yet a lower priority. As future specific "fec event" classes are designed, the task priorities used by them must be appropriate as measured against task priorities used by existing classes.

5. Conclusion

The event system has been in use in RHIC FECs for several months now, though not extensively. The system will get its first rigorous test during the injection transfer line test currently underway. Lessons learned during this test will be used to steer the design of the specific "fec event" classes. It is expected that the object oriented design of the event system will allow fine tuning of the tradeoffs used for particular "fec event" classes without affecting other classes.

References

[1] B. Kuiper, Issues in Accelerator Controls, Proc. ICALEPCS 91, Tsukuba, 1991.
[2] L.T.Hoff, J.F. Skelly, Accelerator devices as persistent software objects, Proc. ICALEPCS 93, Berlin, Germany, 1993.
[3] James O. Coplien, Advanced C++ Programming Styles and Idioms, Addison Wesley, 1992.
[4] Bjarne Stroustrup, The Annotated C++ Reference Manual, Addison Wesley, 1990.
[5] V. Paxson, Glish: a software bus for high-level control Proc. ICALEPCS 93, Berlin, Germany, 1993.