The OPAL Event Builder; Practical Experience with C++ in Data Acquisition

Frans Meijers
CERN, 1211 Geneva 23, Switzerland

The OPAL experiment uses a VMEbus based data acquisition system with MC680x0 processors running OS-9. The control of the data flow through the event builder is implemented using C++. Experience with GNU C++ under OS-9 is discussed, including typical overheads associated with the use of object oriented primitives. A technique to map object member functions on external events is described.

Introduction

The programming language C++ has become popular and compilers are now available for many platforms, including the OS-9 operating system [1] often used at experiments for data acquisition. Object oriented (OO) programming techniques have been successfully applied in online systems in areas such as run control. However, the efficiency of the C++ language allows it to be used for more time critical applications also. This contribution describes a small application, the OPAL event builder, embedded in VMEbus.

The Data Acquisition System

The data acquisition (DAQ) system of the OPAL detector at LEP [2] is a modular and distributed system, based on a heterogeneous selection of computers. It consists of a large number of loosely coupled subsystems. The subsystems with real-time requirements are implemented in VME and run OS-9 on MC680x0 based single board computers. Each of the seventeen subdetectors uses a dedicated VMEbus based computer system for read-out and data reduction. The seventeen subdetector systems are connected to the event builder, also implemented in VME, which in turn is connected to a workstation with VMEbus interface. All these connections are provided by VME Inter-Crate (VIC) buses [3]. Events are about 100 kbyte long and trigger rates are 1-10 Hertz. An Ethernet local area network (LAN) connecting all computers is used for all communications except for trigger information and event data, which are both transmitted over dedicated buses. The DAQ control system, which runs on a VAX cluster supervises the activities of the subsystems during common data taking. It contains a model of each subsystem as an abstract object which can be in a few states. State transitions can be induced and completed transitions are reported back by commands sent over the LAN.

The Event Builder

The event builder can be viewed as a particular implementation of a generic event router, which collects subevent data from the individual data sources, merges these, and forwards the full events to one or more destinations. In addition to this basic functionality, the event builder fulfills the
following requirements. Subevents corresponding to the same trigger are not available at the same time in all subdetectors because of variable processing times in the subdetector systems. Furthermore, extensive data processing in parallel streams at the subdetector level may cause subevents to be delivered out of sequence. Therefore, subevents are read and buffered in the order they become available. Subevents are resynchronised using the trigger number as a key and full events are in trigger number sequence on output. Dynamic memory allocation is required since events span a large range in size. Extensive error detection and handling is provided. For example, an error condition is generated if a subevent does not arrive within a defined timeout. The DAQ control system is notified and corrective action can then be taken. The set of subdetectors to be included in the event building process can be changed whenever the run is paused intentionally or by an error condition.

Transmission of events is controlled by means of event descriptor messages (containing trigger number, event address and length) and event acknowledge messages. These are exchanged event-by-event between the event builder and all source and destination systems. The messages are transmitted over VICbus using VICbus interrupts. The event builder uses the information in the messages to setup the transfer of the event data by DMA devices (independent data movers). This uses memory mapped access over VICbus to transmit data between source or destination system and buffer memories in the event builder. Internally, the subevent data belonging to the same trigger number are scattered in the input buffer memories. They are copied to a concatenated full event in the output buffer by DMA. The hardware configuration can to a large extent be adapted to meet the required throughput by utilising separate VIC and VSB buses and multiple DMA devices performing concurrent transfers.

**Programming Model**

The control of the data flow through the event builder can be implemented in different ways. One approach for such an application involving multiple parallel operations is to map it onto a multitask design. Each task typically blocks until a particular event occurs (such as the increment of an event flag upon an interrupt). The different tasks communicate via shared memory and use semaphores to protect global variables and for synchronisation. However, an initial implementation along these lines showed that it is difficult to implement error handling as the overall system is not necessarily in a well defined state due to the different threads of execution.

Alternatively, the control can be implemented as a single task with handler routines to be executed on the occurrence of asynchronous events. Use of C++ allows to structure the design in terms of objects, instances of a class (a generalisation of the C struct), representing a real or abstract object. Each object is composed of data members and member functions to act on these data items. For this application a mechanism was developed, described below, to link a member function to the occurrence of a specific event. When the event occurs, the appropriate member function is called for the particular object. Hence, within the program object member functions are called from other objects or are invoked on the occurrence of specific events. All member functions typically implement short execution paths and do not invoke blocking system calls but rather use an asynchronous mechanism.
Dispatcher Mechanism

The mechanism to use object member functions as event handlers is implemented using OS-9 queued signals. At run-time a dispatcher table is maintained. A service routine allocates a signal and enters the pointer of the calling object and address of the member function in the table. The caller uses the allocated signal to setup an asynchronous event. After occurrence of the signal the intercept routine will call the appropriate handler.

Signals are well supported in OS-9. Standard devices can be set up such that OS-9 will send the specified signal when data is ready on the device. This is used to recieve commands through pipes. For user developed device drivers (eg to support DMA devices and for message exchange over VICbus) this functionality needs to be included. Timeouts are conveniently implemented with setup of alarm timers which cause OS-9 to send a signal after the specified time has elapsed.

Efficiency Issues

The attractiveness of C++ in general, as an improved C with in addition efficient support for OO features, are also valid for the OS-9 GNU [4] implementation\(^1\). Interfacing to other languages is easy. Object member function invocations are just function calls and the use of (single) inheritance does not introduce any extra overhead compared to C. Inline functions avoid function call overhead and are very useful for short functions (more powerful than C macros). The use of inheritance with virtual functions (where at run-time different functions are invoked depending on the actual object type) will introduce some penalty due to table lookup and due to the fact that short functions cannot be expanded inline. The typical function call overhead increased from 0.7 usec to 1.1 usec for a virtual function in the simplest case.\(^2\) Object creation and destruction with \texttt{new} and \texttt{delete} uses the underlying \texttt{malloc()} and \texttt{free()} system calls for memory allocation. A useful feature is that one can define class specific versions of \texttt{new} and \texttt{delete} to have full control over memory allocation.

The response time to external events (interrupts) using OS-9 queued signals was measured. The total time between the occurrence of the VME interrupt and the first statement in the intercept routine is 50 usec, using a minimal device driver with an interrupt service routine which clears the interrupt and sends a signal to the user task. This real-time response is similar to a scheme where OS-9 events are used to synchronise interrupt service routine and user task. However, there is no priority mechanism for the queued signals.

Implementation in C++ Objects

This application is composed of a dozen classes. A non-hierarchical approach, with a collection of free-standing classes that can be mixed was followed. No class libraries were used.

There are classes to provide services, such as the signal dispatcher and alarm timers which hide OS-9 specific aspects. Two instances of the memory storage class, with member functions

\(^1\) version 1.40 was used

\(^2\) The timing measurements were performed using a Motorola MVME167 VMEbus single board computer (25 MHz MC68040) running OS-9 2.4.
to allocate and free storage exist; one for input and one for output event buffer. There are objects for each source and destination system of a class representing a generic external system, with member functions to send and receive messages. Another class represents the event descriptors. Objects of this class are dynamically created and destroyed as events enter and leave the system. The model which the DAQ control has of the event builder is implemented in a class with methods corresponding to the commands to invoke state transitions.

Concluding Remarks

The event builder implementation separates the control of data flow and the transfer of the event data. The throughput rate for empty events is 50 Hz, with the control task running on a 16 MHz MC68020. The overall data throughput is several MBytes/s, depending on the configuration of DMA devices and intercrate busses.

The control of the data flow is a small, but relatively complex application. Use of C++, taking advantage of some of its OO features, allowed the implementation to be robust and well structured, without loss of efficiency. The C++ code is of the order of 1000 lines. Modifications to adapt to changing requirements were easily carried out.

References

[1] OS-9 from Microware Systems Corporation, Des Moines, Iowa, USA.

