A Flexible Real-Time Architecture

Gregory L. Wickstrom
Surety Electronics & Software Department
Sandia National Laboratories
glwicks@sandia.gov

Abstract:

Assuring hard real-time characteristics of I/O associated with embedded software is often a difficult task. Input-Output related statements are often intermixed with the computational code, resulting in I/O timing that is dependent on the execution path and computational load. One way to mitigate this problem is through the use of interrupts. However, the non-determinism that is introduced by interrupt driven I/O may be so difficult to analyze that it is prohibited in some high consequence systems. This paper describes a balanced hardware/software solution to obtain consistent, interrupt-free I/O timing, and results in software that is much more amenable to analysis.

Section I: Background

Much of the software being developed today is for embedded systems. It is used in numerous non-safety critical applications such as entertainment systems, kitchen appliances, and children's toys as well as safety critical systems such as automotive breaking systems, flight control systems, and medical instrumentation. In most cases the underlying computer based implementation is transparent. In order to maintain that transparency, and in most cases to be viable at all, these systems must react to real world events as they happen. That is they must perform in real-time. A computer can be said to be operating in real-time if it can take some input from an external system, do some processing, and output any resulting feedback to that system at or before the required time. A computer system that controls a chemical process that takes place over the course of hours may not have very demanding real-time requirements of it, but it must operate in real-time nonetheless. Conversely, an antilock breaking system must evaluate a number of input parameters and perform its output in tiny fractions of a second is also a real-time system, albeit with more stringent requirements. As the real-time requirements of a system approach the tens of microseconds range, the ability of software to meet those requirements becomes more difficult, even impossible.

Section II: I/O Timing Problems

The core requirement for a real-time system to meet its stringent I/O timing requirements is often not easy to meet. A variation of a few milliseconds or a few microseconds writing or reading some I/O value can mean the difference between success or failure. The following sections illustrate the source of difficulty in conforming to real-time requirements.

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000.

Output Timing

Any software beyond the most trivial of code will have multiple execution paths. For a real-time system to interact with the external world, input/output (I/O) statements are placed into the instruction stream at the logically appropriate place in the code. Figure 1 below illustrates part of an application where a pulse on a digital I/O line must be generated to some other device connected to its output port.

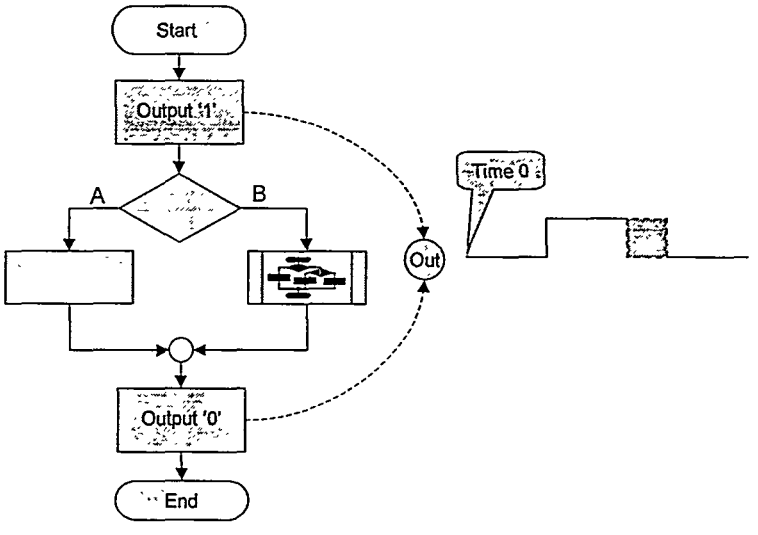

Figure 1

This simple application sets the output value to one and then continues with some other processing and eventually sets the output back to a zero. Depending on some internal or external state conditions, the code executed between those two I/O instructions will either follow path A or path B. When path A is followed only simple in-line processing is required with no branching. If path A is always taken, the duration of the output pulse will always be the same. If path B is taken, a function call is made, and within it a complicated series of branches and loops is performed. It would be a very rare occasion that the code execution time of path A and path B would be the same, as a result the duration of the output pulse would jitter with the software flow characteristics. The pulse duration jitter may not be a problem for some external systems using the pulse output, but for others it could be catastrophic. For simple code, analyzing and quantifying the jitter characteristics may be trivial, but the analysis grows very quickly with even mild increases in code complexity.

Note that the code fragment illustrated in Figure 1 is shown in isolation, but will most often be part of a larger software system where *Time 0* is not likely to coincide with the start of this code. In such cases the situation is even worse than is illustrated, because the pulse start time will jitter with execution path variations leading to this code fragment.

DISCLAIMER

Input Timing

Figure 2 illustrates that a similar problem exists with input data.

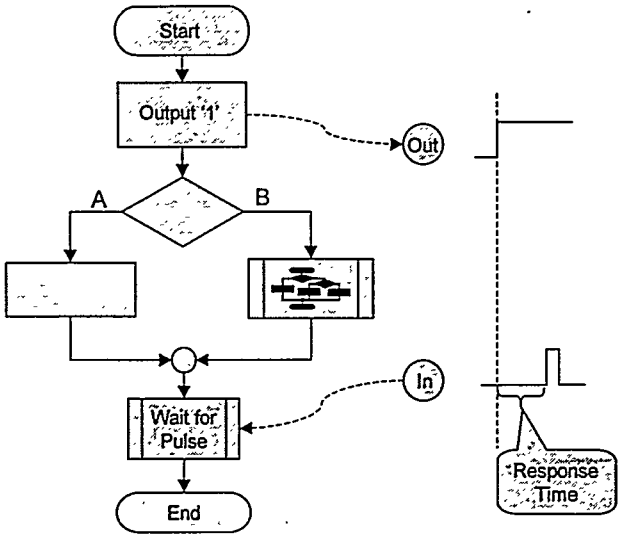

Figure 2

This system is similar to the previous one. However the output is only driven high, and the external system responds by sending a pulse back in return, where the software polls the external system until it senses the high going and then low going edges of the returned pulse arrives. We can only be guaranteed that this system will work if the longest execution path in the software can complete before the return pulse has begun, an effort that may not be trivial.

Note that a combination of the code fragment shown and the external device that generates a pulse in return will likely need to be analyzed as a system. Specifically, the response time of the external device will need to be analyzed. If that device is software based, it too is likely to have pulse jitter times that's dependent on its code flow. So an analysis on what appears to be a simple system, can quickly get very complicated.

Interrupts

As the processing requirements of a system increases between I/O events, the need for an asynchronous way of controlling I/O timing becomes necessary. Software system may make use of interrupts to enable a capability that would be difficult or impossible with a polling based system as previously illustrated. Figure 3 below illustrates a system that must accept and recognize an input pulse that may arrive at an arbitrary time. Although this is an elegant solution for relatively simple systems, it has its own share of problems.

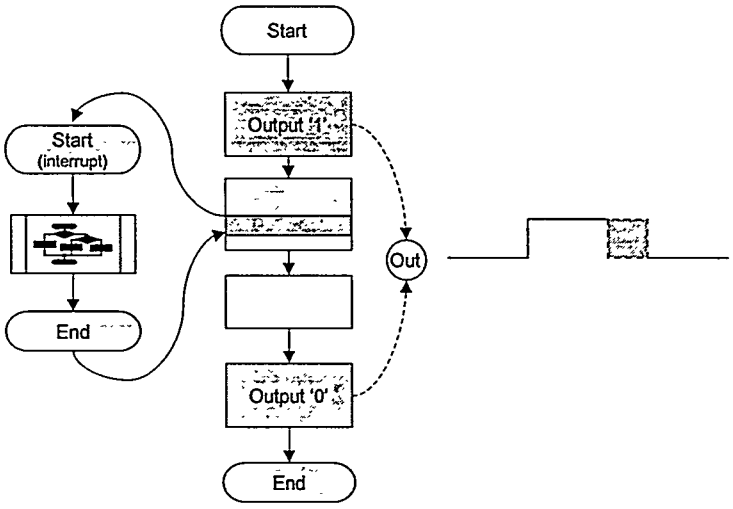

Figure 3

The main code fragment in Figure 3 was designed to meet a requirement for an absolutely stable output pulse duration with no branches or variable loops between output statements. However, since interrupts are asynchronous events, one may occur anytime between the output statements, violating the timing requirement for the output pulse.

A designer may choose to also control the output pulse via the interrupt system, but as the number of interrupt handlers increase so does the complexity of managing those interrupts. Only one interrupt can be executing at a given time, and a priority scheme may be identified to manage such problems. Such options become complex very quickly. In fact, for some safety critical systems, all interrupts are prohibited as they overly complicate correctness analysis.

Other Problems

The previous simple examples illustrate only a few of the many problems real-time systems can have meeting system level requirements. Some others include:

Clock Rate Changes: the timing at the interface will change with any processor clock rate changes, which may have been increased to provide additional processing power.

Processor Changes: a change in processor implementations may yield failure in code that once worked. As processor design capabilities advance, the clock cycles per instruction may change with a new version of the same processor, resulting in execution timing differences.

Deterministic Behavior: required for high consequence systems. We must be able to predict the behavior and timing for all inputs to assure that no sequence or variation in the timing of a sequence can lead to an undesirable state.

System Testing: meeting requirements is difficult when the timing of a system is tightly integrated into the architecture. Usually a custom designed tester is necessary to generate and validate I/O timing, these testers are often difficult themselves to design and build. As a result most of the core testing capability is often not available until late in the development cycle, increasing the risk of a design error.

Power Consumption: As embedded real-time systems are incorporated into many consumer devices, the need for many of them to operate on battery power is a must. As such, power consumption is a large factor that drives the processor selection, impacting the computing throughput and I/O capabilities.

Volume: A number consumer devices require embedded real-time systems to be small enough to fit in the palm of a hand. These requirements also drive processor selection, computing throughput and I/O capabilities.

Section III: A Solution

Certainly there are numerous products on the market that have overcome some of the problems described. Modems operate with approximately 20µs bit timing and networks operate at frequencies of 10MHz, 100MHz and higher. In order to meet these requirements, much of the required functionality has been put into hardware. Hardware based I/O offers two main advantages: it often provides some form of buffer memory, and it may handle some level of feedback to control the system. For example, a simple UART has a register that contains a byte of data to be transmitted using some hardware handshaking scheme. When system wishes to send information, the byte is loaded into the UART, the hardware waits until some input I/O reaches an '*okay to transmit state*', then data is shifted out another port until the entire byte has been sent. This scheme isolates the software from control issues and reduces the software timing requirements by a factor of eight, as eight bits are shifted out for every byte the software loads. Of course the timing requirements can be eased further with more internal UART memory with few changes to that UART's control hardware.

It is desired to devise a generic solution that may be reused and improved on with time without having to design from scratch for moderate requirements changes. For domain specific problems one such solution has already been demonstrated in the development of real-time software using port based objects [2]. Although not generally used for real-time systems, the domain of the PC based platform is another example of a generic architecture that has withstood the test of time.

The basic question becomes how the hardware/software functionality should be split to make most effective use of the strengths of each. Often designers overreact one way or the other by putting more functionality into the hardware or software than is required. To make matters worse, shifting responsibility between the two often requires a change in system architecture, heightening the risk of introducing design errors.

The Separation of Behavioral and Real-Time Components

To achieve a good hardware/software balance, lets first look at their relative strengths.

Hardware: extremely fast, deterministic timing, immutable, low power consumption, can easily handle simple feedback loops.

Software: straightforward algorithm implementations, flexible, can handle computationally intensive feedback loops.

Basically, software is best at behavioral functionality and hardware is best at deterministic real-time I/O. These observations imply a division point in the hardware/software architecture. We would like to develop an architecture where the basic behavioral/real-time components could be re-instantiated in either hardware or software with minimal changes to the overall system. We would also like to eliminate the need for interrupts, if possible.

A Flexible Architecture

Let's consider the architecture illustrated below:

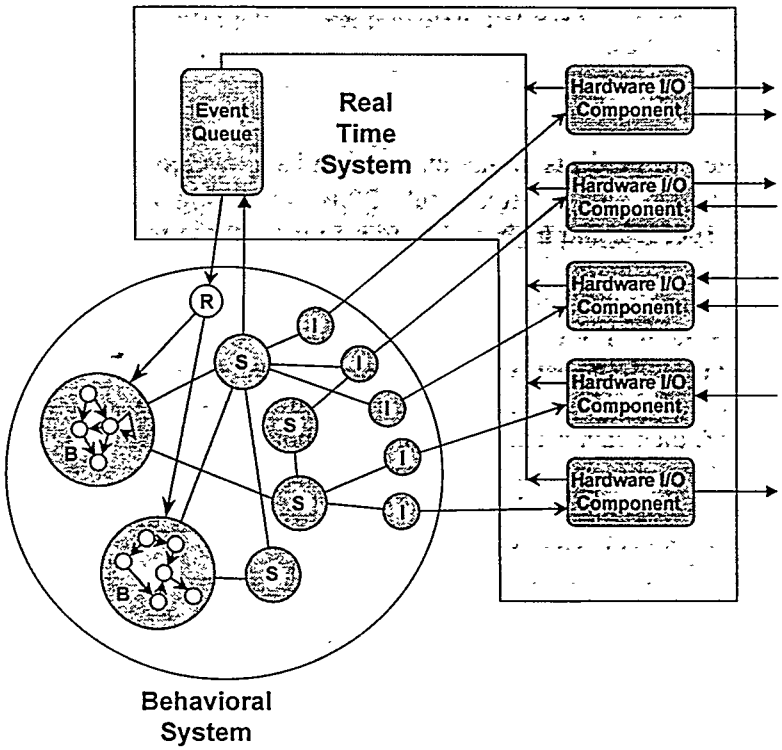

Figure 4

This illustration is basically a high level of abstraction for a generic embedded real-time system. This high level of abstraction provides a framework for mapping the requirements to potential implementations as well as numerous other benefits [1].

The round objects represent functionality implemented in software, and the rectangular objects represent functionality implemented in hardware. The real-time I/O is controlled by the hardware components. Each of these hardware components is memory mapped to the controlling processor and has internal configuration and status registers to allow the software to tailor its functionality and monitor its state. The functionality of these components alleviates much software real-time requirements by handling lower level I/O functions directly. Depending on the system

throughput characteristics, these requirements may be further reduced by incorporating embedded queuing memory into the hardware components. This internal memory may be used to smooth out peak throughput situations, and its effectiveness may be analyzed with standard queuing theory methodologies.

The event queue component is the key element that assists in eliminating a requirement for interrupts. All asynchronous input events are intercepted and handled by the hardware components. Any external event or hardware component condition that requires attention by the software will generate a unique event ID that will be placed in the event queue. Events are read out by the software system and the appropriate action is taken. The software system becomes event driven, and the only software polling allowed is to wait on an event to arrive in the event queue. A system may require that some external events be handled before others. This form of prioritization may be handled by an event queue implementation that has multiple internal queues, one for each priority.

As the system requirements increase to handle larger numbers of I/O ports, the number of hardware components grows. To minimize the design effort of these hardware modules, it is good practice to identify common functionality between the real-time hardware modules in order to maximize the opportunity for design reuse. In fact, a standard sub-component architecture is recommended. The following figure represents such an architecture.

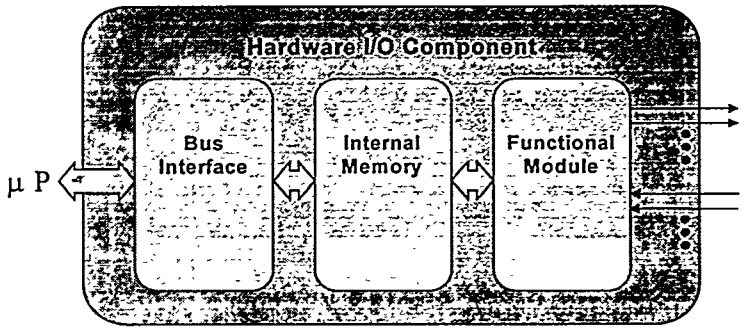

Figure 5

Hardware Components

The hardware sub-components are described as:

Bus Interface: is used to interface to the processor executing the behavioral software. If the processor is changed to one with a different bus protocol e.g. non-multiplexed to multiplexed, only this module will be affected.

Internal Memory: defines a parameterized internal configuration and status register design. Some of the status registers may be configured to generate events to the event queue as desired. This sub-component also contains a parameterized design of a queuing memory cell as needed.

Functional Module: is the sub-component that contains the circuitry to interface with the external I/O port and handles the high speed I/O protocol and timing.

This architecture simplifies the addition of new I/O components and isolates the process into three steps:

1. Reuse the existing bus interface verbatim.

2. Reuse the internal memory sub-component by instantiating the parameterized variables that define the internal queuing memory size and the number of configuration and status registers.
3. Design the functional component necessary to meet the low level I/O requirements.

Since much of the design complexity in I/O circuits is dealing with configuration and status registers, the design of the functional component is made much easier by having a standardized interface to the internal memory unit.

The event queue is the only component with a slight variation on the component architecture since it has numerous internal queues. Figure 6 illustrates its internal architecture.

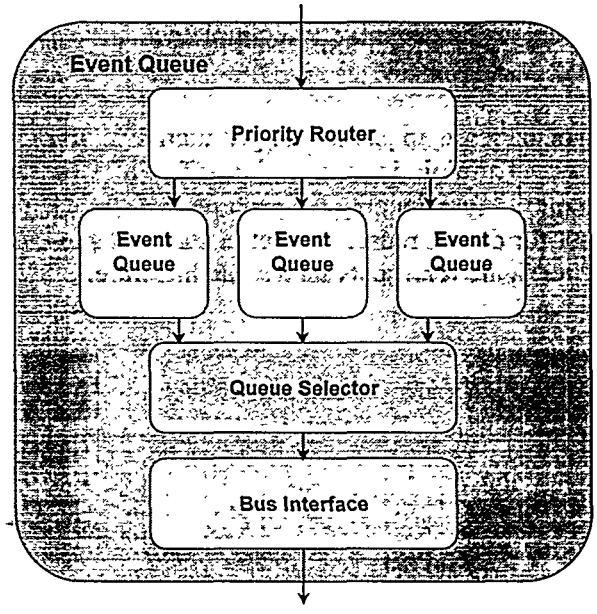

Figure 6

The bus interface is reused and the priority router may be thought of as the functional sub-component. The internal memory module is modified to handle more than one queuing memory unit, and another functional sub-component must be developed to read from the appropriated queue at the appropriate time.

Software Components

The software components illustrated in Figure 4 may be thought of as four basic types.

[R] Router: reads events from the event queue and calls appropriate functions. This is the only software component in the system that is allowed to poll an I/O device. Note however, that the event queue hardware could be implemented to provide a hardware '*wake up*' signal to bring the processor out of a sleep mode. Such an implementation would allow for a fairly transparent power conserve mode.

[B] Behavioral: is the high level system controller. Since the software is basically an event driven system this is likely to be a state machine model. Note that there may be one or more behavioral systems. Such an implementation effectively allows for cooperative multitasking.

[S] Service Object: services are those other functions/objects that are required so support the behavioral system.

[I] Interface Driver: are use to get information in and out of the hardware components.

An object oriented software implementation is desired as it allows for a plug and play like capability, and well defined and standardized interfaces will provide for effective testing techniques [3,4]. It will also prove to be important in supporting the capability to migrate functionality between hardware and software.

Hardware/Software Implementations

A key advantage of this architecture is that it will also support the notion of migrating component implementations between hardware and software without an architectural change. The architecture as described is an implementation of both hardware and software components. This same architecture, however, could be implemented entirely in hardware or entirely in software. In fact, our initial implementation of this real-time architecture was a software-only implementation.

A Hardware-Only Implementation

If a system were such that only simple computational requirements are necessary on the input data, a hardware implementation of the behavioral system would be straightforward. Since the behavioral system is event driven, a hardware based state machine could supply the control sequencing. Hardware implementations of state machines are well known, so the design and implementation should be relatively straightforward. In fact, the hardware I/O components could be used unchanged. Only a bus interface driver would need to be developed as an interface mechanism between the behavioral hardware and the I/O hardware.

A desire for such a system could be driven by power and volume constraints in the requirements. A hardware implementation of a simple behavioral system would likely be far fewer gates than a complete processor needed to execute the same behavior in software. This may result in much lower power consumption and smaller volume than its processor based cousin. Another benefit, would be a homogeneous development, testing, and maintenance environment. The drawbacks to such an implementation would be the risk of a requirement changing to include some processing that could result in a complex hardware design to support.

A Software-Only Implementation

Not all real-time systems are small embedded implementations. They may also be implemented in desktop PC's, workstations, and even mainframes. Generally in these cases, the real-time requirements are not quite as restrictive and often there is no need for custom hardware development to support fast and/or complicated bit I/O. However, the basic architecture can still be used. The only change is that what used to be hardware components must now be implemented as software components. In such an implementation, the asynchronous inputs must be handled via interrupts, or separate threads of execution that emulate the hardware's functionality. Of course, the internal architecture of these software I/O components probably should not reflect that of the hardware, but the basic configuration/status and behavioral elements would be the same.

Some of the benefits of this approach are a homogeneous development, testing, and maintenance environment. The drawbacks of a software-only solution: the inability to meet tight timing requirements, and it will run into interrupt related timing analysis problems as described earlier. It's most valuable contribution is scalability and the decoupling of behavioral and temporal design elements.

Component Migration

A major strength of this architecture is the ability for functional components to be easily implemented in either hardware or software with no change in the architecture. Let's take for example a system that must compute a complex but repetitious mathematical algorithm on some data. As a functional software object, the calculations required may work but require a large amount of computation time. If this function could be moved to hardware, the computation is likely to be much faster. Figure 7 illustrates how the component is moved to hardware.

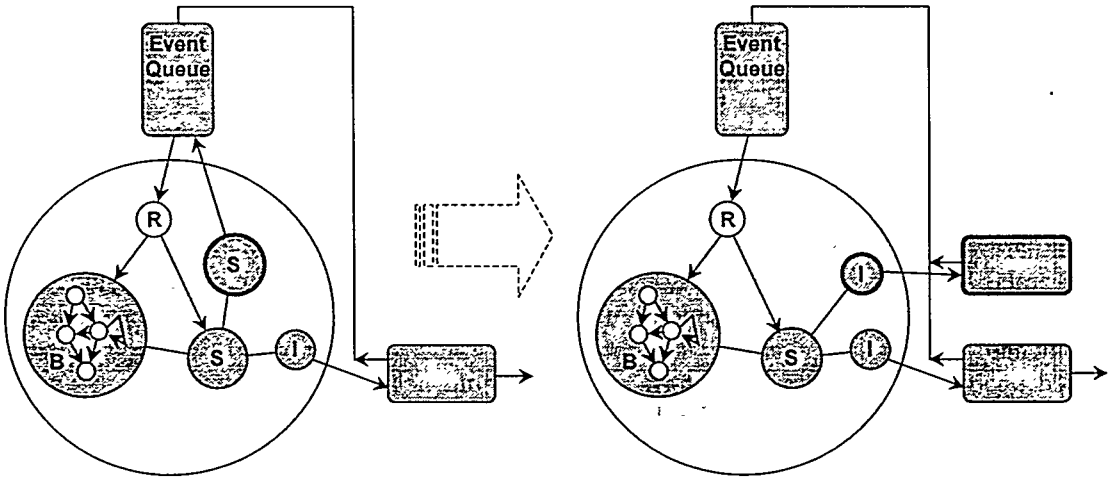

Figure 7

The component begins as a software service object with some interface functions to its behavior. Any requests made to it requiring a time consuming operation will receive a completion notice through the event queue. When the functionality is migrated to hardware, a software driver for the hardware must be developed with the same interface functions, and the hardware design will be such that it too places completion notices to the event queue. This allows for complete transparency of the computational functionality with respect to the rest of the system. The only change in the system is that its I/O handling capabilities have now been increased since the computation time has been decreased.

Solution as Applied to I/O Timing Problems

Figure 8 below illustrates the impact on the I/O timing code fragment discussed at the beginning of this paper that generates a simple output pulse.

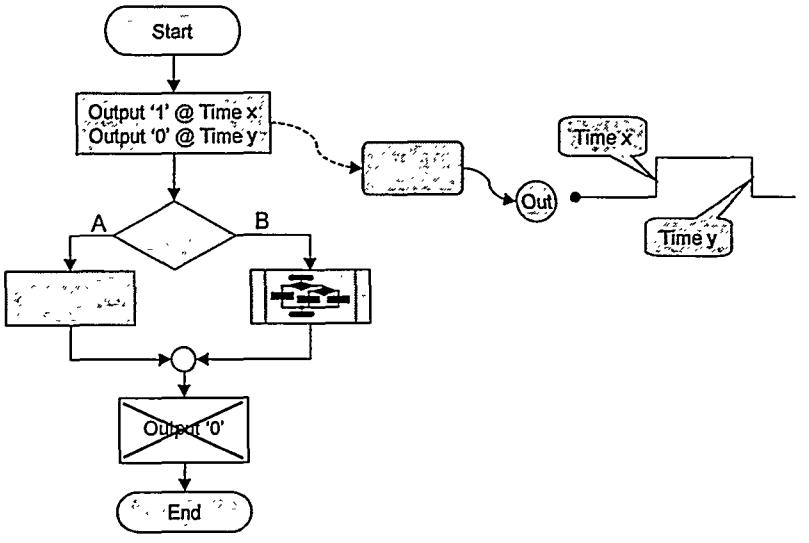

Figure 8

The primary difference is an I/O control element between the behavioral code and the I/O pin. On entry to the code fragment that element is configured with the starting and ending time of the pulse, and from that point on the system processing time has no effect on the pulse duration. The block of code setting the pin low is no longer needed.

Figure 9 below illustrates the effect this architecture has on the input signal code fragment.

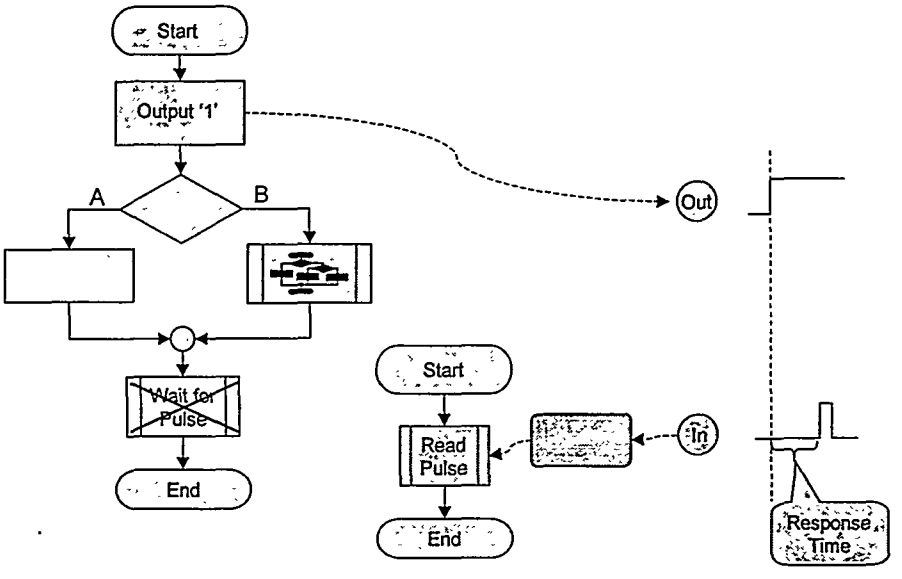

Figure 9

The first portion of the code fragment need not change (however to be pure, there would be an I/O component as in Figure 8). In this system the responding pulse may arrive at any time during the processing, because the event will be queued up by the input component which would then be processed by the '*Read Pulse*' code. Note that the pulse reader is now implemented as a completely separate processing element to handle the event independently. It would be called by the event queue reader when appropriate.

Section IV: Advantages and Limitations

There are several advantages to this architecture:

Abstraction: this approach provides a framework to describe a system at a high level of abstraction. As such, it may aid in identifying the elements required by a system and the best implementation.

Isolation: it is conductive to object oriented design approaches and leverages those techniques for data abstraction and well defined functional interfaces into the very architecture itself.

Implementation Independence: each of the components may be implemented in hardware or software, or be freely intermixed depending on the timing, power, and volume constraints.

Testing: the component nature of the system lends itself well to testing thoroughly at the component level to mitigate the risks of integrating the components at a later stage. Also, the behavior of a system can be fully tested entirely independent of the real-time I/O environment, since it has been decoupled from the timing of the system. Finally, this same architecture could be used in the design of a custom tester, as the tester itself is likely to have similar timing requirements.

No Interrupts: the system handles asynchronous input without interrupt processing, allowing it to be deterministic and more easily analyzed. As such, it may potentially be used in a number of high consequence systems.

Some of the disadvantages of this architecture are:

Event Processing Loop Time: although no interrupts are needed in the system, this does not mean that there is no latency to handling processing intensive asynchronous input events. The events must still be read and the appropriate function(s) must be called to handle it. As interrupt systems in microprocessors are hardware based, they can usually be executing interrupt relevant code in a few clock cycles. An implementation of this architecture with a software based event router will take longer to begin processing an input event that than an interrupt handler. However, if this latency proves to be a problem there is the possibility that some additional functionality could be handled in a hardware component.

Priority Inversion: the event queue contains a separate queue for events at each priority level. When an event comes into the event queue it will be placed into the appropriate internal queue. If the event reader reads a low priority event, it means no higher priority event existed at the time of the read. However during the processing of that low priority event, a high priority event may enter the queue but it will not be serviced until the current low priority event is completed being processed. This problem may be made more tolerable if any long duration process periodically invokes the event reader.

Section V: The Architecture in Practice

The general architecture was originally implemented entirely in software for a number of simulation systems requiring real-time performance in the millisecond range. The complexity of the implemented systems ranged from low to medium (from 5,000 to 20,000 lines of code) and performed as expected. Each of the systems was developed in a environment of changing behavioral and timing requirements, and all of the changes were made with only localized effects,

i.e. significant behavioral changes were made without affecting timing, and timing changes were made without affecting the behavior.

The hardware components described in Section III have been developed in the VHDL language, simulated, and fabricated in Field Programmable Gate Arrays (FPGA's) as well as an Application Specific Integrated Circuit (ASIC). With these implementations we have been able to target several applications with timing requirements in the microsecond range. Again we have been able to make changes with only local effects, and the complexity of these systems roughly the same as those for the software-only solution.

In addition to having the ability to easily make changes, we have targeted two vastly different processors with similarly encouraging results. Initially, we executed our behavioral simulation code on a 400MHz Power PC 604 communicating with the real-time FPGA based I/O hardware over a VMEbus backplane. Then recompiled the same source code for a 8031 microcontroller with a 2MHz clock. The microcontroller was part of a core based ASIC connected directly connected to IC logic implementing the same design as the FPGA hardware. Analysis with a logic analyzer revealed that the behavioral and timing differences between the two implementations are indistinguishable.

Section VI: Summary and Conclusion

In the design of embedded real-time systems, there is rarely an obvious solution to handle timing related problems. The available solutions diminish rapidly as the timing requirements become tighter, and quickly become impossible to meet with a software-only solution. When designing a system that has a hardware/software mix, the divisions of labor is also not always an obvious one. The architecture presented in this paper provides a design framework that aids in the definition of a system and provides a consistent methodology for handling a wide range of real-time applications.

References

[1] V.L. Winter. *Visualization and Animation as a Technique to Assist in the Construction of High Assurance Software*. CADE Workshop on Visual Reasoning, 13th International Conference on Automated Deduction, 1996.

[2] D.B Stewart, R.A. Volpe, and P.K. Khosla. *Design of Dynamically Reconfigurable Real-Time Software Using Port-Base Objects*. IEEE Transactions on Sofware Engineering, Vol. 23, No. 12, pp. 759-776, Dec. 1997.

[3] M.H. Chen, H.M. Kao, *Effect of class testing on the reliability of object-oriented programs*, The Eighth International Symposium on Sofware Reliability Engineering, Nov. 1997

[4] D. Graham, *Testing object oriented systems*, Object Expo Europe – Java Expo Conference Proceedings, pp. 309-318, 1996.