OBIOS: The Open Basic Input Output System Standardization Effort

Greg Buzzard and James Ready
Ready Systems
470 Potrero Ave.,
Sunnyvale, CA 94086

October 10, 1990

Abstract

This article describes the current state of the Open Basic Input Output System (OBIOS) standardization effort. This effort, which represents a collaboration of several key hardware and system software vendors in the real-time and embedded systems marketplace, is aimed at developing a call interface and associated client/device interaction model that is intended to provide a common abstract interface for a wide range of I/O devices.

1 Introduction

The OBIOS standardization effort began towards the end of the first quarter of 1990. The description provided in this article represents the current state of the design consensus that has been achieved by the OBIOS technical committee. The technical committee is currently working on the detailed specification of several key classes of devices including asynchronous serial lines, network interfaces, SCSI devices, counters and timers.

The primary goal of the OBIOS effort is to provide a high-performance physical I/O interface and underlying implementation that will provide *plug compatible* interoperability between hardware and software modules from a wide variety of different vendors. There are two major design criteria. One is that the OBIOS not require calls to any operating system services – this should make OBIOS portable across a wide range of software systems. The second criterion is that the OBIOS provide a powerful abstract interface that is common across a wide range of I/O devices. Throughout the remainder of this article OBIOS implementations will be generically referred to as the BIOS.

2 Key Features

The BIOS is viewed as a *passive server*. It is comprised of a collection of routines and data, the BIOS does not embody any active processing components such as tasks or interrupt service routines (ISRs). Callers of the BIOS routines are referred to as *clients*. The BIOS routines execute within the context of their calling clients. The following list describes the key features of the BIOS design.

- Instances of the BIOS exist to increase the portability of client software across a heterogeneous set of processors and I/O devices. The primary goal is to encourage elegantly concise implementations that provide portability without compromising performance.

- The BIOS defines *abstract* interfaces for devices such as timers, counters, serial lines, network controllers, disks, and SCSI controllers. The client sees only the abstract interface for each class of device. The abstract interface exposes all of the functionality that is commonly expected for a given class of device. For example, clients are presented with one interface to read, write, set baud rate, set parity, as so forth for all serial line devices – clients do not need to be aware of the details of the hardware interface (i.e., device registers and their usage semantics) that distinguish one particular serial device from another.

- The BIOS provides the mechanisms for controlling low-level physical I/O. Higher-level I/O operations such as buffer management, protocol handling and input line processing are to be provided by BIOS clients.

- The BIOS is operating system independent. It does not contain references to any operating system functions. For example, it does not assume access to semaphores, nor does it allocate or deallocate memory. Also, the handling of time-outs and the queuing of BIOS requests are the responsibility of the client. The BIOS provides information on physical I/O requests such as expected service times, but it does not make any operating system policy decisions.

- Interrupt service routines (ISRs) are handled outside of the BIOS. The BIOS is viewed as a passive server. The control and invocation of all BIOS activities, including those that occur at interrupt-time, is explicitly under control of the client. During an ISR a client will likely make a call to the BIOS interrupt-time routine.
- The individual BIOS calls are not reentrant. Serialization of BIOS calls and the queuing of requests are the responsibility of the client.
- The BIOS is expected be called from a privilege level that is capable of executing all machine instructions.

3 Overview

A BIOS is comprised of instances of modules. The physical manifestation of a module is not specified, it may be a single integrated circuit (IC), and entire board, or a portion of an IC. Module instances contain the data structures and code used to manipulate physical I/O devices. A module must be initialized before it can be used. The initialization process includes setting up data structures to associate the module with a given device address, determining if a device is present at the given address, associating interrupt vectors with the hardware and placing both hardware and software into a known quiescent state. Examples of modules range from MC68901 multiple function peripheral chips to Z8530 serial communications controller chips to EXOS202 ethernet controller boards. We can view each of the above modules as defining a class. A BIOS is comprised of instances of modules from one or more module classes.

Modules, in turn, are comprised of functional units. There may be one or more functional units per module. If there are multiple functional units within a module, they may be from the same or differing functional unit classes. It is the functional units that provide the services that are normally associated with the actions of physical devices – e.g., there is a functional unit class specification for timers and another for serial lines, another for network controllers, etc. The specific services that must be provided are determined by the class to which the functional unit belongs. There is a basic set of operations that must be provided by all functional units. Additionally, each class will specify operations that are unique to the functional units within that class.

As an example, consider two different serial line devices, the Zilog Z8530 and the Motorola MC68901. The Z8530 module is comprised of two instances of a Z8530 serial line function unit, one for each of the A and B channels. This chip is typically integrated directly onto a CPU board via memory or I/O space mapping. This is an example of a module comprised of two instances of a single functional unit from (obviously) a single class. Serial line operations are just a portion of the capabilities of the MC68901. This chip is typically integrated onto a CPU board in a manner similar to the Z8530. Thus, the MC68901 module is comprised of instances of functional units from differing classes. In particular, the MC68901 module includes one instance of an MC68901 serial line functional unit along with instances of MC68901 timer functional units – the MC68901 serial line functional unit is a (sub)class member of the serial line class, the MC68901 timer functional unit is a (sub)class member of the timer class.

The BIOS specification defines the operations and behavior of modules, functional units in general, and specific classes of functional units.

Each module provides four procedures: GET_MODULE_INFO, INITIALIZE, TERMINATE and GET_MODULE_STATUS. All functional units provide at least the following seven procedures: ENABLE_UNIT, DISABLE_UNIT, ENABLE_INTERRUPTS, DISABLE_INTERRUPTS, RESET, ABORT and POLL. Additionally, functional units will typically implement several distinct commands that initiate I/O or control activity with the hardware device. These commands are collectively referred to as START commands. For example, a serial line functional unit will have START commands for read, write, set baud rate, and others. Finally, functional units should provide an interrupt handling routine (INTR_SVC) for each type of interrupt that a given unit may receive.

The GET_MODULE_INFO call returns information that is statically known about a module.

The INITIALIZE call is used to bring a module and all of its units to a known quiescent state. It specifies the address of the associated hardware device and provides a memory buffer and other information specific to this instance of the specified module. This call returns information about the module including specific details of the functional units that it contains.

The TERMINATE call places the module in a quiescent state. This call should only be made when the functional units that comprise the specified module have ceased all activity. The module becomes unusable until a subsequent successful INITIALIZE call is made.

The GET_MODULE_STATUS call returns information about the

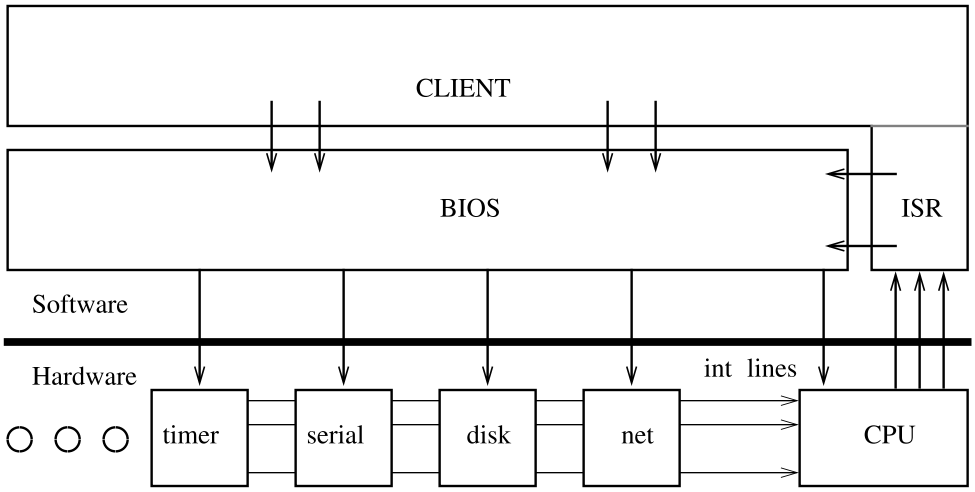

Figure 1: Module Interactions in a BIOS Environment.

dynamic state of a module. This information includes the number of active requests and the status for each functional unit.

The START commands request services from the particular functional unit to which they are made. A request may be accepted, or not accepted. Non-acceptance implies that the functional unit is busy and unable to concurrently execute another request. If accepted, the service will either be completed immediately, or initiated with completion expected at some time in the future. In either case, the client can assume that command will execute and return in an extremely short period of time. In the latter case, the client is expected to rendezvous with the BIOS in the future to formally complete the service.

The INTR_SVC call formally acknowledge completion of one stage of a service that has been initiated by a START command. This call is expected to be made from an interrupt service routine in response to a device interrupt. The INTR_SVC routine implements a rendezvous point between the client and device.

The ABORT call fatally terminates a service that has been initiated by a START command.

The POLL call is used to seek formal acknowledgement, along with any returned status and data, of completion of one stage of a service that has been initiated by a START command. The POLL routine implements a

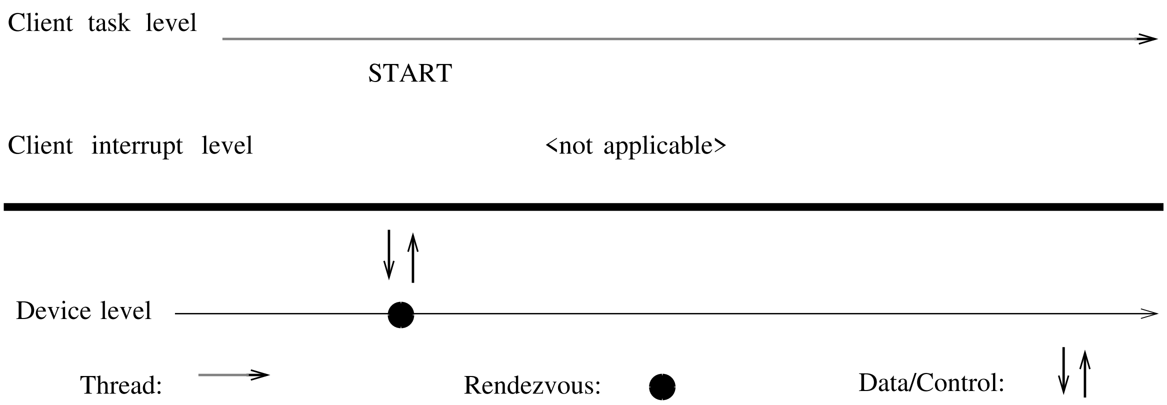

Figure 2: Single Stage Service Mode.

rendezvous point between the client and device.

The precise sequence and manner in which the client calls START commands, ABORT, INTR_SVC and POLL is directly related to the type of service requested. Services may be classified into one of three types: single-staged, multiple-staged, or continuous-staged.

3.1 Single-stage Services

A single-stage service is one that, once accepted, is immediately completed. Generally, such a service is expected to complete very quickly with the CPU waiting only a minimal amount of time for the device to respond. An example of a single-stage service is the changing of the baud rate of a serial port.

A single-stage request for service is accepted and completed via a single call to a START command (see Figure 2). Any data transfer that may occur between the client and device is specified in the request block that is passed to the START command. A status of *ok* is returned for a successful request.

3.2 Multiple-stage Services

A multiple-stage service is one that, once accepted, cannot be immediately completed. In general, a multiple-stage service takes considerable time to complete and, therefore, is performed in multiple stages to prevent the client from idly consuming CPU resources for an excessive amount of time (e.g.,

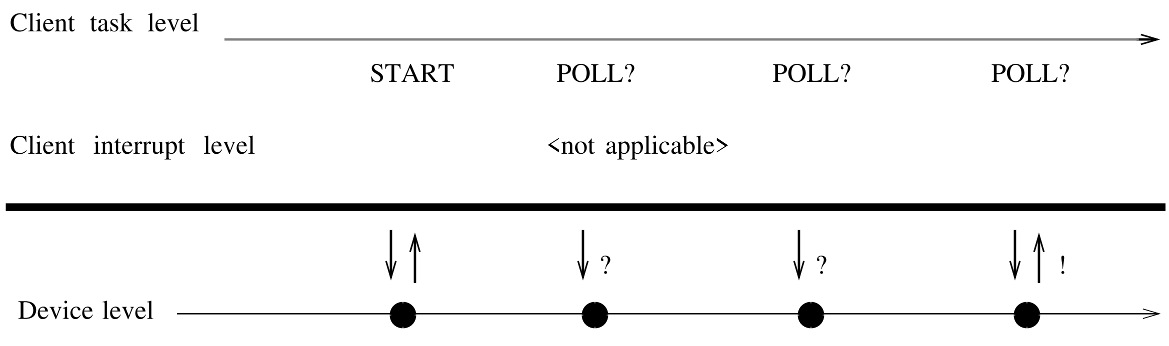

Figure 3: Multiple Stage, Polled Service Mode.

reading a sector from a disk, or waiting for a real-time alarm to expire). Such a request transitions from stage to stage either via interrupts (if the request was STARTed in interrupt mode), or via polling (if STARTed in polling mode).

Like a single-stage request, a multiple-stage request is accepted via a START command. However, unlike a single-stage request, a multiple-stage request returns a status of *ok_pending* from the START command. This indicates that one or more subsequent calls to INTR_SVC or POLL are required to eventually complete the request (typically from either an interrupt-level ISR, or from a task-level polling routine – see Figures 3 and 4). Specifically, if a subsequent call to INTR_SVC or POLL returns a status of *ok*, the request is said to have reached completion; if the call returns a status of *ok_pending*, another stage of the request must be processed in the future and a subsequent call to INTR_SVC or POLL must be made.

3.3 Continuous-stage Services

A continuous-stage service is one that, once accepted, is never explicitly completed. This "standing request for service" can be associated with, for example, a keyboard. A continuous read service request could specify to always raise an interrupt whenever keyboard input arrives. Similarly, a serial line unit could allow the client to enable a continuous read in parallel with a continuous write to provide the throughput necessary for a terminal emulation program. When the program exits, the client could ABORT the continuous read and write, returning the unit to a quiescent state.

Again, like a multiple-stage request, a continuous-stage request is

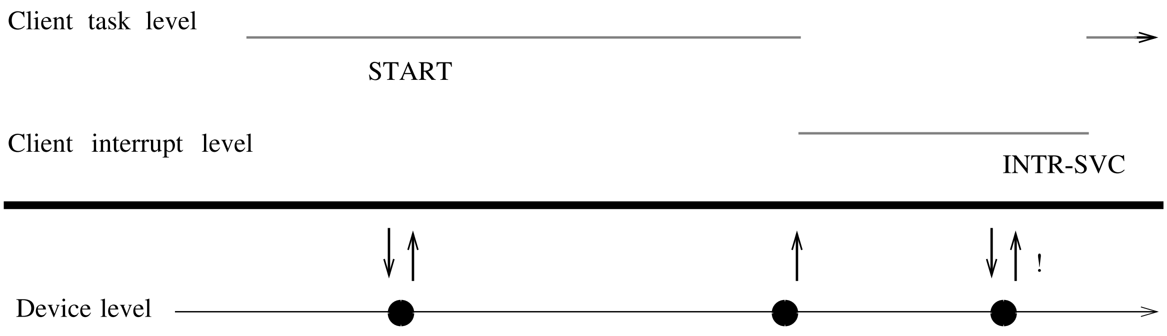

Figure 4: Multiple Stage, Interrupt Service Mode.

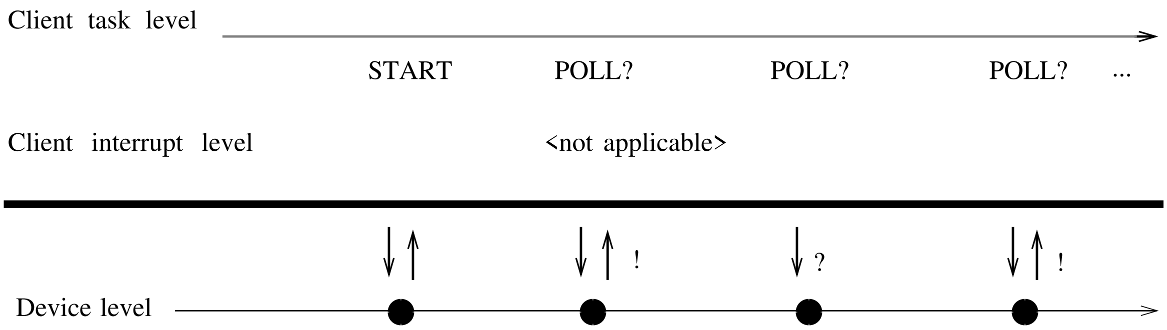

Figure 5: Continuous Stage, Polled Service Mode.

accepted via a START command, which yields a return status of *ok_pending*. Also, a continuous-stage request proceeds from stage to stage via calls to INTR_SVC or POLL, either at the interrupt-level or task level (depending upon whether interrupt mode or polling mode was specified to the START command – see Figures 5 and 6). Unlike a multiple-stage request though, a continuous-stage request will never return a status of *ok* from INTR_SVC or POLL. A continuous stage request will always return *ok_pending* and is said to never reach completion.

4 Client/BIOS Interaction

Interactions with modules and units are described by the following items:

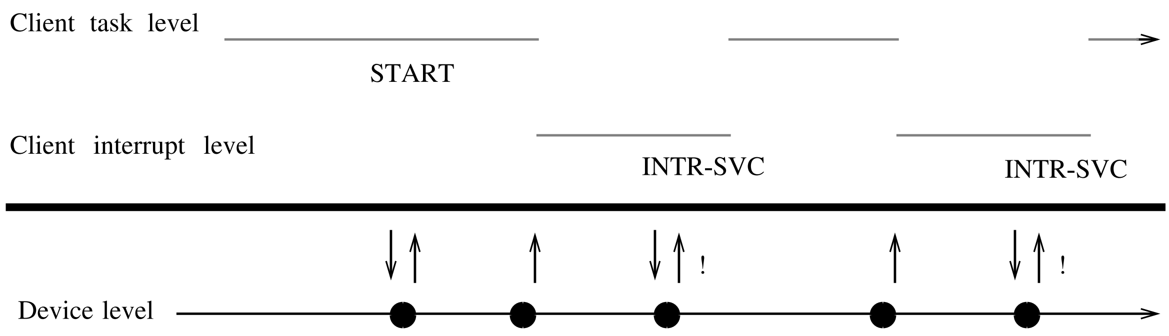

Figure 6: Continuous Stage, Interrupt Service Mode.

- A set of module call interfaces. Each module provides procedural interfaces: GET_MODULE_INFO, INITIALIZE, TERMINATE and GET_MODULE_STATUS. These call interfaces are common for all modules. The variable length parameters may vary in length, but the parameters are otherwise the same.

- A set of unit call interfaces. Each unit provides at least eight procedural interfaces, ABORT, POLL, INTR_SVC and at least the following five START commands: ENABLE_UNIT, DISABLE_UNIT, ENABLE_INTERRUPTS, DISABLE_INTERRUPTS, and RESET. These call interfaces are common for all units. The variable length parameters may vary in length, but the parameters are otherwise the same. Additionally, each unit class (e.g., serial, network, etc.) will define an additional set of pertinent commands which are specific for that class. Multiple and continuous stage START commands and all ABORT and POLL calls include a request block parameter. This parameter is described below.

- A description of valid request block formats. Each class of functional units supports distinct set of services. Each request block contains a generic portion that is common for all classes of functional units and a class specific portion that may vary from unit class to unit class. For the length of time that BIOS requests are active (i.e., from their initiation until their completion or abortion) the information recorded in the request block will be *shared* between the client and the BIOS. This shared information often includes I/O buffers that are to be read or written. The rules that govern this sharing are specified for each functional unit.

4.1 Request Blocks

Request blocks (RBs) contain the following information:

1. Staging mode (in).
2. Time-out value (out).
3. *Function-specific information* (in and/or out).

The first two items are common for all functional units and services. The last item is unit and, potentially, service specific.

Valid staging modes are interrupt and polled. The interpretation of the time-out value depends on the staging mode. For interrupt mode it is the maximum time duration the client should wait for the next stage of a multiple-stage or continuous-stage service to complete. In this case, the BIOS is stating that if notice of completion is not received in the specified amount of time, the client is advised to ABORT the transaction. In the case of the polled staging mode, the BIOS is suggesting a reasonable waiting interval for the client before it makes the next POLL call.

4.2 Procedural Interactions

Client interactions with a BIOS would typically proceed as follows. The client is assumed to have knowledge of device addresses and module types for all of the devices that may exist in the system. The client will make GET_MODULE_INFO calls for each module in the list. Upon return, the client will know the memory and vector requirements of the module, the number of units within the module, and the addresses of the remainder of the module routines. The client may then INITIALIZE the module and provide it with the requested (or larger) amount of memory and the requested vectors. Upon return from the INITIALIZE call, the client will know the class of each unit in the module, the addresses of the ABORT, POLL and START commands for each unit and the address of the instance data that defines each unit. The client can proceed to make a series of calls to the functional units that comprise the module until the module is TERMINATEd. The absence of *expected* devices may be implicitly indicated to the client at the time of the INITIALIZE call by a bus error fault.

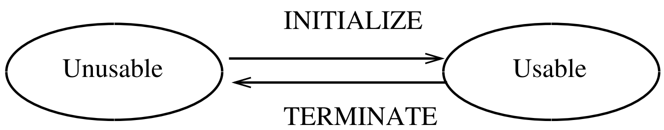

Figure 7: Device State Transitions.

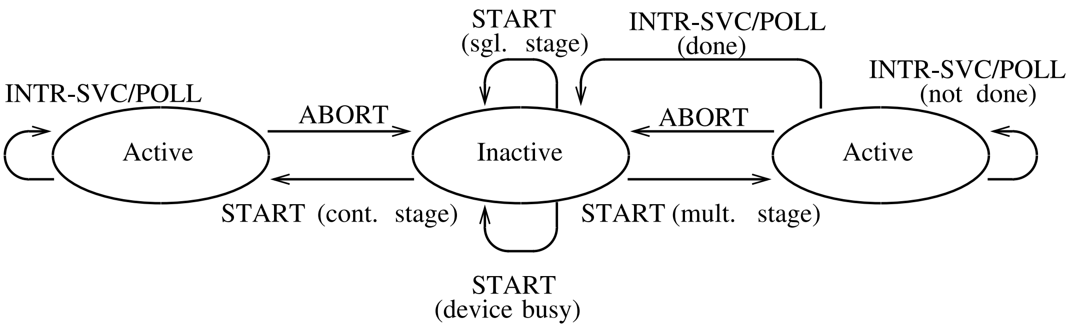

Figure 8: Request State Transitions.

Figures 7 and 8 show the state transitions that are allowed for modules and functional units. A module must be INITIALIZEd before any of its units are able to service a request. Requests for device interactions are then submitted via the START commands. If a request is accepted, the START command returns *ok* or *ok_pending*. If the request is not accepted because the device is busy with another request (and presumably incapable of processing another simultaneous request), the START command returns *device_busy*. If the request is not accepted because the device appears to be acting in an errant manner, *device_failure* is returned.

Note that START commands do not define how many active requests may be executing at any instance, each functional unit has its own particular semantics. For example, a simple timer may be capable of having only one active request at any instant, conversely, a serial port is usually capable of having two active requests at any instant – read and write. Even further, a given subclass of functional units, for example, a Ciprico 3550 SCSI disk controller, may allow a different number of active requests than some other subclass of disk controllers. This information is available to the client in the functional unit instance data structure

If a START command returns *ok*, as opposed to *ok_pending*, the service requested was single-stage. Thus, the request for service was completed immediately and any data transfer between the request block and device that was expected to occur has already occurred.

If a START command returns *ok_pending*, the request for service cannot be completed immediately. It will be completed sometime in the future, thus, the request is either multiple-stage or continuous-stage. In this case, in addition to a status of *ok_pending*, the START command will also return a time-out value. The exact interpretation of this timeout value depends on whether the requested staging mode is interrupt or polling. In either case, the timeout value can be used to determine how long a client should wait for a response from the device before it assumes the device has failed. When a failure is assumed the client is expected to call the ABORT interface to reset the device. The request block passed to ABORT should be the same one that was passed to the START command. Most ABORT calls will cancel the outstanding request and return immediately. When an ABORT request is acknowledged but cannot immediately be carried out, an *ok_pending* status is returned. ABORT must then be called again to see if the termination request has been completed.

If the staging mode specified to a START command is interrupt, stages will transition on device interrupts. If the staging mode specified to a START command is polled, stages will transition via client polling. In the polling case, the client is expected to call POLL (specifying the same request block that was passed to the START command) until a status other than *device_busy* is returned (or until a device failure is assumed and a call is made to ABORT). If a status of *ok* or *ok_pending* is received from POLL, one stage of the request has completed and any expected data transfer between the request block and device has already occurred. In the event of *ok_pending*, a new time-out value is returned.

In the interrupt case, upon receipt of the appropriate device interrupt, the client's ISR is expected to call the INTR_SVC procedure to formally acknowledge the interrupt. The INTR_SVC call is made to the routine associated with the interrupting functional unit vector. Two parameters are passed via the INTR_SVC routine, one is inbound, one is outbound. The inbound parameter is a pointer to an opaque data structure. The outbound parameter (from INTR_SVC to its caller) will point to the appropriate request block. The location of the INTR_SVC routine and the associated opaque data structure (likely to be the unit instance data) are provided to the client via the module INITIALIZE call. After calling INTR_SVC, any data transfer between the request block and device is completed. Again, in the event of *ok_pending*, a new time-out value has also been returned.

In summary, the return status from either the INTR_SVC or POLL procedure may indicate *ok*, *ok_pending*, *device_busy*, *device_failure*, or *no_pending_request*. *Ok* implies that the request for service has completely finished. *Ok_pending* implies that this stage of the request for service has completed and yet another stage (e.g., via a device interrupt) should be expected in the near future (within the specified time-out period). *Device_busy* indicates that the current stage is not completed yet, it can only be returned from a polling-mode request. *Device_failure* implies that a fatal error has occurred. *No_pending_request* implies that there is no request pending for the device and service specified in the request block. This may indicate that the request block that was passed to the START command to initiate a given operation was not passed to POLL.

4.2.1 Reentrancy

The procedural interfaces for all module and unit calls except INTR_SVC are mutually exclusive and non-reentrant. For example, at most one thread of execution is expected to be in any of these routines for a given serial functional unit at any given point in time. In order for a BIOS to be usable in a multiprocessor environment it must rely upon the client to enforce the exclusive access requirements of the mutually exclusive and non-reentrant routines. Thus, the client is expected to provide protection against the actions of other task level threads of execution. Disabling interrupts does not protect against the simultaneous access of data that may occur by threads running on separate CPUs in a multiprocessor. BIOS Implementations are allowed to disable interrupts (preferably by disabling the interrupt capability of a specific device) for brief periods of time to prevent interrupt level access to shared data structures. The combination of both client provided task level protection against reentrancy and simultaneous access, and the disabling of (device) interrupts can be used to ensure exclusive access to shared data.

BIOS clients are not required to prohibit reentrancy for a given INTR_SVC routine nor are they required to prohibit mutually exclusive access between INTR_SVC and any of the other five routines for a given device. BIOS clients are expected to call INTR_SVC only from interrupt level. INTR_SVC routines may disable further interrupts while the current interrupt is being serviced by the BIOS. For proper operation in multiprocessor systems, this should be done by disabling interrupts at the device level and not at the CPU level. This will protect against further data accesses that may occur from other interrupt service routines (ISRs) that invoke the same low-level INTR_SVC routine (i.e., the INTR_SVC routine for the same module). Implementations of INTR_SVC routines must be constructed so as to not require protection against simultaneous access of data by the task level routines (e.g., START commands, POLL, ABORT).

There are no reentrancy restrictions within the context of an active multiple or continuous stage service request. That is, successive calls may be made to a START command for a given functional unit while it is already servicing one or more active requests. These successive calls may either be accepted or rejected depending on the capabilities of the given device. The instance data for each functional unit indicates the maximum number of simultaneous requests that are allowed for that unit.

4.2.2 Task Level vs. Interrupt Level Calls

Satisfaction of the reentrancy behavior described in Section 4.2.1 strongly suggests that calls to INTR_SVC should be made only from interrupt level and that calls to all other routines should only be made from task level.

5 BIOS Data Structures

A BIOS implementation will comprise just three primary types of data: module information data, module instance data and functional unit instance data.

A pointer to the module information data area is provided to the client for each module when the GET_MODULE_INFO call is made. The data in this area is constant, it provides the following information:

1. Module class (i.e., type of hardware).
2. Module version number and revision level.
3. The number of units in the module.
4. The interrupt vector requirements of the module.
5. Pointers to the module routines.
6. The memory requirements of the module.

The client allocates the expected amount of memory, and then calls INITIALIZE, passing in the allocated memory. This data area is opaque to the client, it is to be manipulated solely by BIOS routines.

A functional unit instance data area is created for each unit contained in a when the module is INITIALIZEd. Pointers to these data areas are returned to the client by the INITIALIZE call. These data areas are created from a portion of the module instance data area. Portions of this data are intended to be visible to the client. This data should be considered read-only by the client, it is to be manipulated solely by BIOS routines. The non-visible, or *opaque* data is not required to have any specific format. The following information will be in the client visible portion of the functional unit instance data areas:

1. Unit class (i.e., type of hardware).
2. Unit version number and revision level.
3. Pointer to unit procedures.
4. Pointer to the subsuming module.

The specific format of the module information data should be the same for each class of module. The specific format of the client visible portion of unit instance data should, likewise, be similar for each class of unit, differing only in the length of the procedure pointer array.

6 BIOS Interfaces

A BIOS presents two groups of interfaces to its clients, module procedures and functional unit procedures.

Four module procedures are provided to clients. The interface and behavior of these routines should be the same for all classes of modules (e.g., Z8530, MC68901, EXOS202, etc). The location of at least the INITIALIZE routine must be known by the client – the client is free to link directly to all necessary module and unit routines. The addresses of all module routines are included as part of the client visible module instance data. The implementations for the routines in each module class are expected to be different. Distinct instances from the same class will share code, but operate on different module instance data areas.

The unit procedures, ABORT, POLL and the five required START commands, should each provide the same interface and behavior across all classes and subclasses of functional units (e.g., serial line class, MC68901 serial line subclass). The addresses of all unit routines are included as part of the client visible unit instance data. The implementations for the routines in each unit subclass are expected to be different, however the call interfaces for the respective routines are the same. Distinct instances from the same subclass will share code, but operate on different unit instance data areas.

Acknowledgements

I would like to thank the many people from several different companies that have contributed to the ideas presented in this article. As usual, space limitations preclude listing all of those involved, however I would like to give special thanks to Bruce Coorpenter of 88open, Wayne Hampton of Software Components Group, Glenn Kasten of Ready Systems, Shawn Morrissey of Force Computers, and David Wilner and Jason Cotton of Wind River Systems for their keen technical contributions.

Biography

Greg Buzzard, an engineering manager at Ready Systems, received his BS and MS degrees in Electrical Engineering and his PhD degree in Computer Science and Engineering from the University of Michigan. Greg is currently on leave from the faculty of the Computer Science Department of the Naval Postgraduate School in Monterey, CA. His research interests include architectures and operating systems for distributed and parallel computer systems. Greg is the technical chairperson for the OBIOS consortium and is a member of both the IEEE Computer Society and the Association for Computing Machinery.