Contents

1 Introduction

This document contains the interface specification for a basic input output system henceforth referred to as the Open Basic Input Output System or OBIOS. The choice of the word "open" indicates that this specification was developed in a manner that was open to input from a wide variety of sources. This standardization 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. Within this document, the term OBIOS refers solely to a collection of abstract device interfaces and related data structures that comprise the OBIOS standard. It is not meant to encompass boot strategies, board-level monitors or other items that may be found in a boot PROM, though these items will be likely clients of a OBIOS.

The remainder of this document is laid out as follows. Chapter 2 describes the operational concepts and functionality of this OBIOS specification. The detailed specification begins in Chapter 3. The specification for each type of device is given in an individual Appendix.

1.1 Key Features

The OBIOS is viewed as a *passive server*. It is comprised of a collection of functions and data, the OBIOS does not embody any active processing components such as tasks or interrupt service routines (ISRs). Callers of the OBIOS are viewed as *clients*. The term client is used generically throughout this document to refer to any code that may invoke or facilitate OBIOS functions. Typically there are two types of clients: one that configures, supports and initializes the OBIOS modules and a second type which uses the OBIOS to read, write and control individual I/O activities. The OBIOS functions execute within the context of their calling clients. The following list describes the key features of the OBIOS design.

— Instances of the OBIOS 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 OBIOS defines *abstract* interfaces for *abstract* 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, and 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 OBIOS 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 OBIOS clients.

— The OBIOS 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 timeouts and initiation of any related action is the responsibility of the client. The OBIOS 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 are handled outside of the OBIOS. The OBIOS is viewed as a passive server. The control and invocation of all OBIOS 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 OBIOS interrupt-time function.

— Mutual exclusion protection for the OBIOS data corresponding to a specific device unit shall be ensured by the client.

— The OBIOS is expected be called from a privilege level that is capable of executing all machine instructions.

— The source code of OBIOS implementations and this standard are intended to be publicly available and freely redistributable.

— ROM based implementations of the OBIOS shall conform to a standard ROM format that allows client code configure and initialize the OBIOS using only ROM resident information and that further allows OBIOS ROMs to be changed or upgraded without having to recompile or relink client software.

1.2 Definitions

For the purpose of this standard the following definitions shall apply:

Class. The term class is used to refer to a set of like objects. Module and unit instances (defined below) refer to specific elements of a class.

Magic cookie. The term magic cookie is used to refer to a specific opaque data item.

Module. The term module is used to describe a portion of the I/O hardware. The precise physical manifestation of a module is not specified, it may be a single integrated circuit (IC), an entire board, or a portion of an IC. In terms of the OBIOS, hardware that shares common initialization data or interrupt vectors is represented by a corresponding module instance data structure.

Module instance. A module instance is the data structure that uniquely describes a portion of the I/O device hardware that uses a single set of initialization data or interrupt vectors. A module instance has an associated set of module functions (code) that are used to operate on the module data and the corresponding physical device. The module code is shared by all module instances that correspond to the same type of device. The same set of functions is defined for all modules.

Opaque data. Data is referred to as opaque if it is meant to be used as a token (magic cookie) but not interpreted by the entity that possesses it.

Unit. The term unit, or functional unit, is used to describe a single hardware entity on which I/O operations are performed. Units are part of a module – there may be one or more units per module and these units may be elements of the same or differing 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, and so forth. In terms of the OBIOS, specific device units are represented by a corresponding unit instace data structure.

Unit instance. A unit instance is the data structure that uniquely a single hardware entity on which I/O operations are performed. A unit instance has an associated set of module functions (code) that are used to operate on the unit data and the corresponding physical device. The specific services that shall be provided are determined by the class to which the functional unit belongs. There is a basic set of operations that shall be provided by all functional units. Additionally, each class (e.g., serial or timer, and so forth) will specify operations that are unique to the functional units within that class.

1.3 Acronyms

The non-standard acronyms used in this specification are listed below.

ISR. Interrupt service routine.

PROM. Programmable read-only memory.

RB. Request block (an OBIOS data structure).

1.4 Notes on this Version of the Document

The changes from version 1.15 to 1.16 include:

a) Acknowlegements are moved to a preface which will be included in the final document.

b) Definitions section has been added.

c) Acronyms section has been added.

2 Operational Concepts

2.1 Overview

An OBIOS is comprised of instances of modules. A module instance is the data structure that uniquely corresponds to a given hardware device. A module instance has an associated set of module functions (code) that are used to operate on the module data and the corresponding physical device. The module code is shared by all module instances that correspond to the same type of device. The physical manifestation of a module is not specified, it may be a single integrated circuit (IC), an 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 shall be initialized before it is 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. An OBIOS 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 shall be provided are determined by the class to which the functional unit belongs. There is a basic set of operations that shall be provided by all functional units. Additionally, each class specifies operations that are unique to the functional units within that class. The same set of operations is defined for all modules.

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 OBIOS specification defines the operations and behavior of modules, functional units in general, and specific classes of functional units. In a typical systems module calls would be used by the client that is responsible for configuring (or reconfiguring) and initializing OBIOS modules. Unit calls would be used by the clients (high-level operating system device drivers or applications) that are interested in reading, writing or controlling I/O activities.

Each module provides the following functions: `get_module_info`, `initialize`, `terminate`, `get_module_status`, and `poll`. All functional units provide at least the following functions: `set_state`, `status`, `reset`, `abort` and `poll_any`. Functional units typically implement several additional commands to initiate or control I/O activity with the hardware device. For example, a serial line functional unit will include commands for read, write, set baud rate, and so forth. Finally, functional units shall provide an interrupt handling function (`intr_svc`) for each type of interrupt that a given unit may generate and a command to poll on the status of a specific previously started request (`poll`).

The `get_module_info` function returns information that is known about the characteristics and initialization requirements of a module. This is the only function that may be called at any time. Calls to all other functions of a module or any of its units (except `initialize`) prior to module initialization and after module termination are considered to be erroneous. The results of erroneous calls are not defined.

The `initialize` function 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 function returns information about the module including specific details of the functional units that it contains.

The `terminate` function places the module in a quiescent state. This function should only be made when the functional units that comprise the specified module have ceased all activity. It is considered erroneous to call `terminate` when there are active requests for any of the module's units. This module becomes unusable until a subsequent successful `initialize` call is made.

The `get_module_status` function returns the number of requests that are currently active for each functional unit.

The `poll` function provides a means by which a client can check on the status of a previous module call whose operation did not immediately complete. This is necessary in some systems that employ intelligent and, possibly, remote I/O devices.

All unit 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 function will return in an extremely short period of time. In the latter case, the client is expected to rendezvous with the OBIOS in the future to formally complete the service.

The `intr_svc` function formally acknowledges completion of one stage of a service that has been initiated by a unit command. This function is expected to be made from an interrupt service routine in response to a device interrupt. The `intr_svc` function implements a rendezvous point between the client and device.

The `abort` function fatally terminates a service that has been previously initiated by a unit command.

The `poll_any` and `poll` functions are 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 unit command. These functions implement a rendezvous point between the client and device. The `poll_any` and `poll` functions provide a *polling mode* alternative to `intr_svc`.

The sequence and manner in which the client calls the unit commands, 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. Of these three types, only multiple staged and continuous staged services may be requested. Single staged services may occur when a multiple stage service request completes immediately. The OBIOS interface is intended to work with a variety of hardware configurations. In order to accomodate remote or intelligent I/O devices that may experience a high command execution latency all unit commands that initiate an I/O data transfer or control activity shall be initiated as either a multiple or continuous stage service request.

The remaining commands that were listed perform the task indicated by their names. The detailed specification for all functions is given in Chapter 3.

2.1.1 Single-stage Services

A single-stage service results from a multiple stage request that, once accepted, is able to complete immediately. Generally, such a request executes very quickly with the CPU waiting only a minimal amount of time for the device to respond. An example of a request that frequently results in a single-stage service is the changing of the baud rate of a serial port.

A single-stage service is accepted and completed by a single (multiple stage) call to a unit command (see Figure 2). Any data transfer that may occur between the client and device is specified in the request block (explained in Section 2.2.1) that is passed to the unit command. A status of *ok* indicates a successful single-stage service.

2.1.2 Multiple-stage Services

A multiple-stage service request 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 spin-locking the CPU for an excessive amount of time (e.g., reading a sector from a disk, or interacting with a remote device). 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).

A multiple-stage request is accepted via a unit I/O data transfer or control command. Unlike a single-stage service, a multiple-stage service returns a status of *ok pending* from the service request command. This indicates that one or more subsequent calls to `intr_svc` (from interrupt level), `poll_any` or `poll` (from task level) are required to eventually complete the request (see Figures 3 and 4). Specifically, if a subsequent call to `intr_svc`, `poll_any` or `poll` returns a status of *ok*, the request is said to have reached completion; if the call returns a status that includes *ok pending* the request has reached a *stage completion*, another stage of the request shall be processed in the future and a subsequent call to `intr_svc`, `poll_any` or `poll` shall be made.

2.1.3 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.

Like a multiple-stage request, a continuous-stage request is accepted via a unit I/O data transfer or control command which produces a return status that includes *ok pending*. A continuous-stage request also proceeds from stage to stage via calls to `intr_svc`, `poll_any` or `poll`, either at the interrupt-level or task level (depending upon whether interrupt mode or polling mode was specified to the service request command – see Figures 5 and 6). Unlike a multiple-stage request though, a continuous-stage request shall never return a status of *ok* from `intr_svc`, `poll_any` or `poll`. A continuous stage request shall always return a status that includes *ok pending* (or, possibly, a fatal error) and is said to never reach normal (non-error) completion.

2.2 Client/OBIOS Interaction

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

— A set of module function interfaces. Each module provides the following functions: `get_module_info`, `initialize`, `terminate`, `get_module_status` and `poll`. These call interfaces are common for all modules. Their variable length parameters may vary in length, but the parameters are otherwise the same.

— A set of unit function interfaces. Each unit provides the following functions: `abort`, `poll_any`, `intr_svc` (one for each type of interrupt that the unit may generate), `set_state`, `status` and `reset`. These call interfaces are common for all units. Additionally, each unit class (e.g., serial, network, etc.) defines an additional set of pertinent commands which are specific for that class. All unit procedures that may require direct response from the hardware device shall include a request block parameter. This parameter is described below.

— A description of valid request block formats. Each functional unit class supports distinct set of services. Each

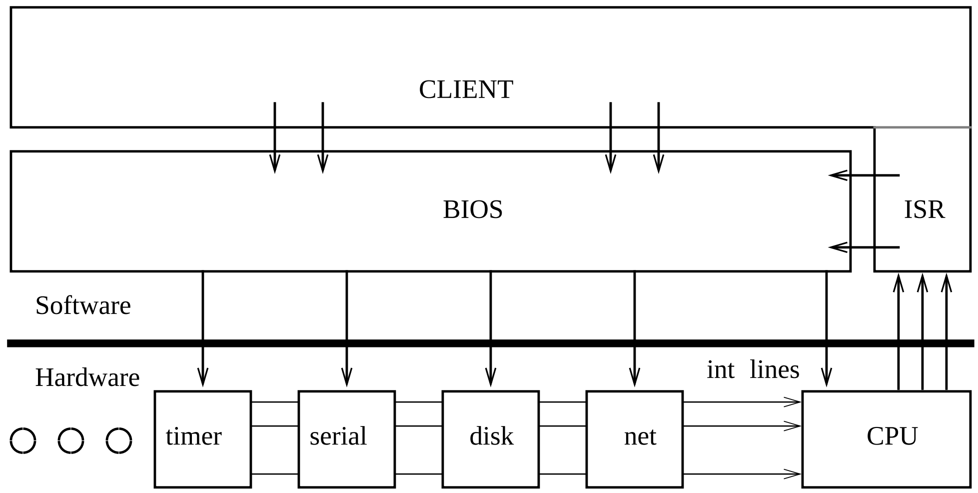

Figure 1 — Module Interactions in an OBIOS Environment.

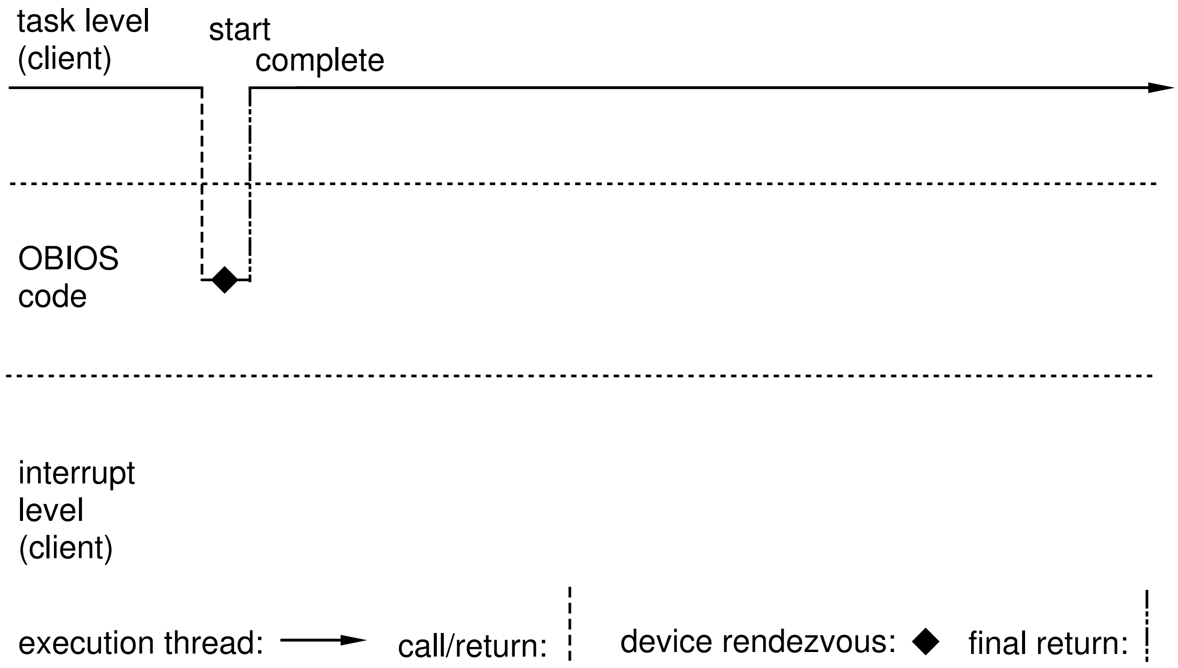

Figure 2 — Single Stage Service Mode.

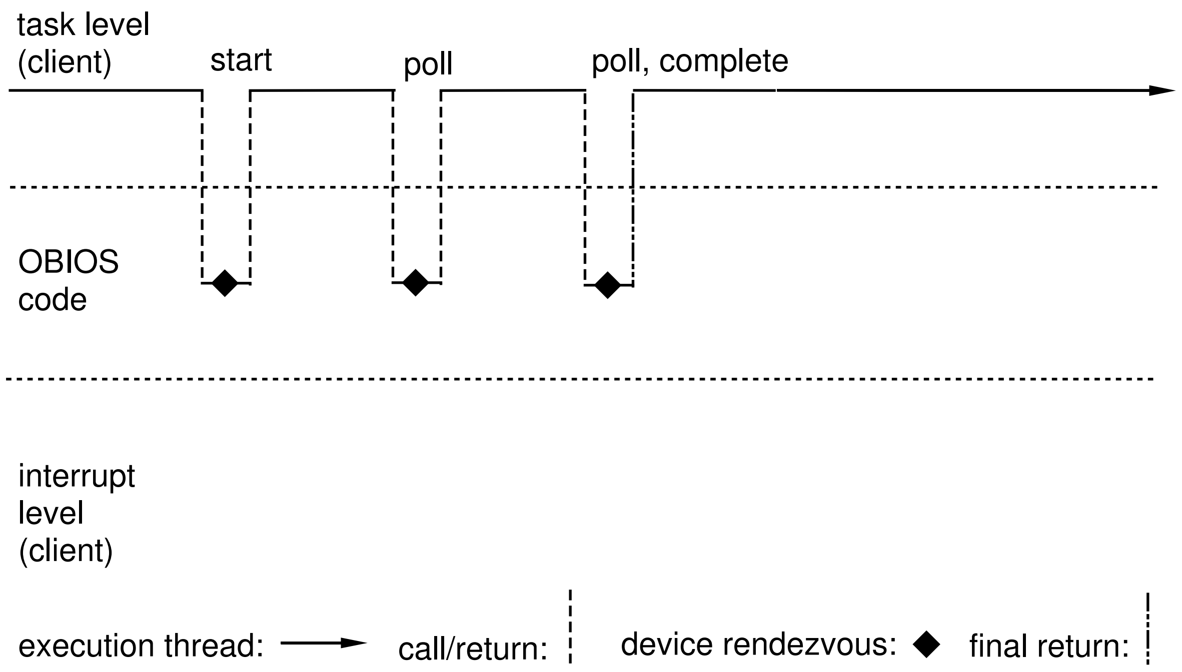

Figure 3 — Multiple Stage, Polled Service Mode.

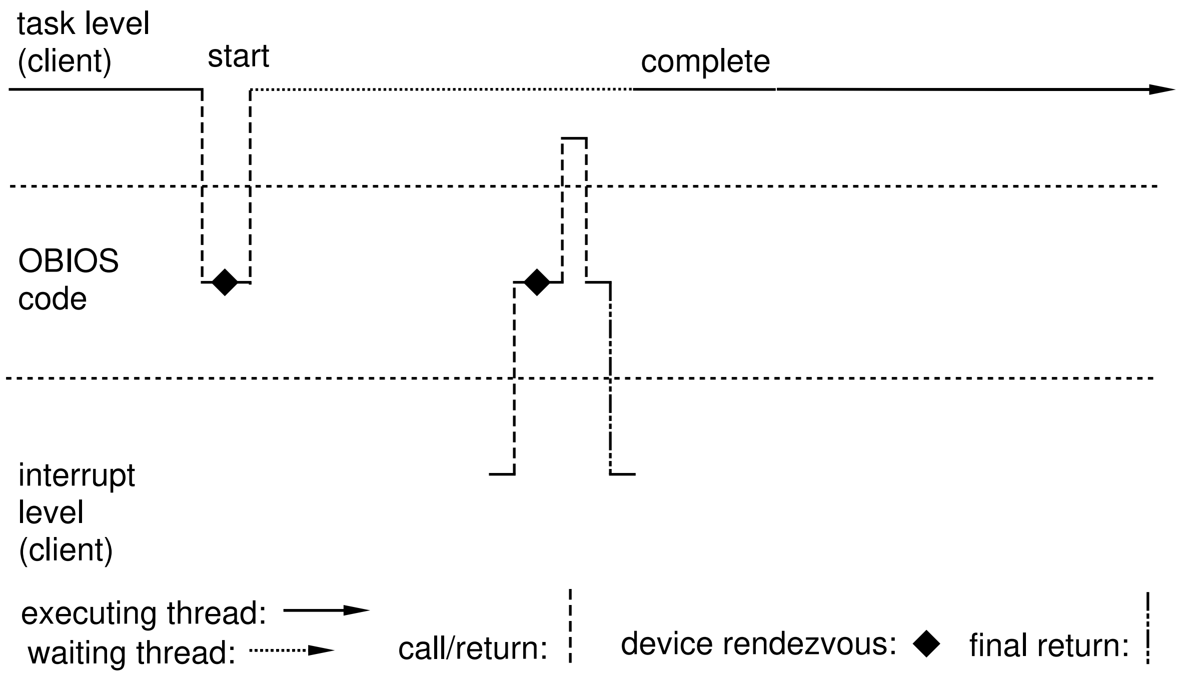

Figure 4 — Multiple Stage, Interrupt Service Mode.

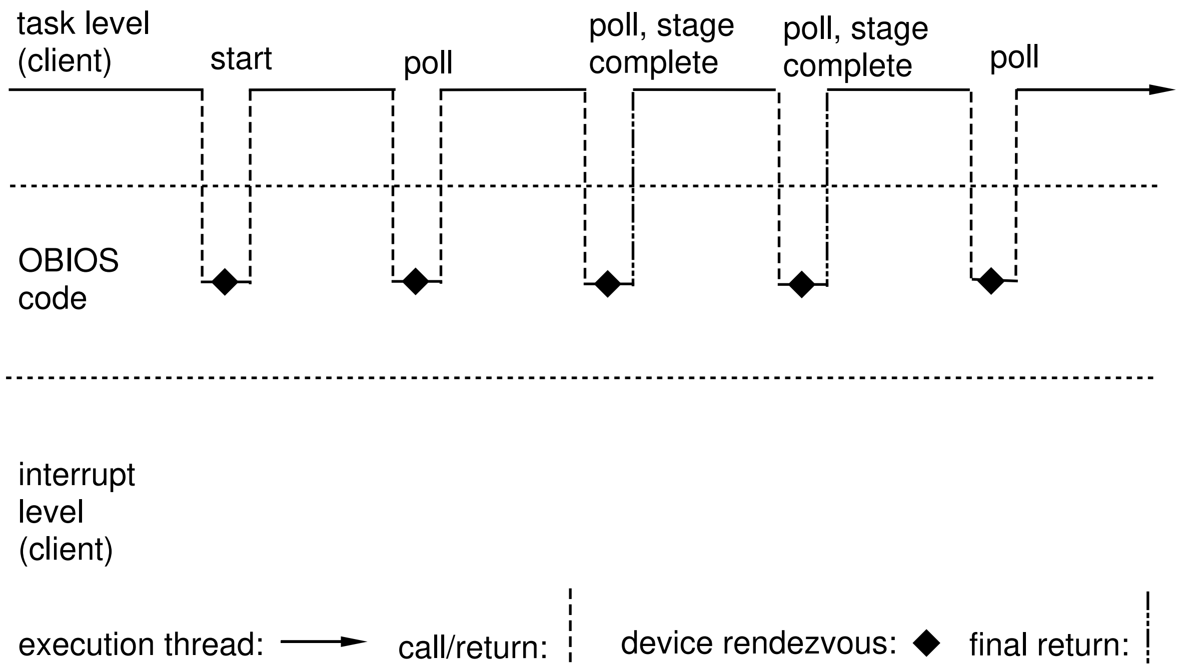

Figure 5 — Continuous Stage, Polled Service Mode.

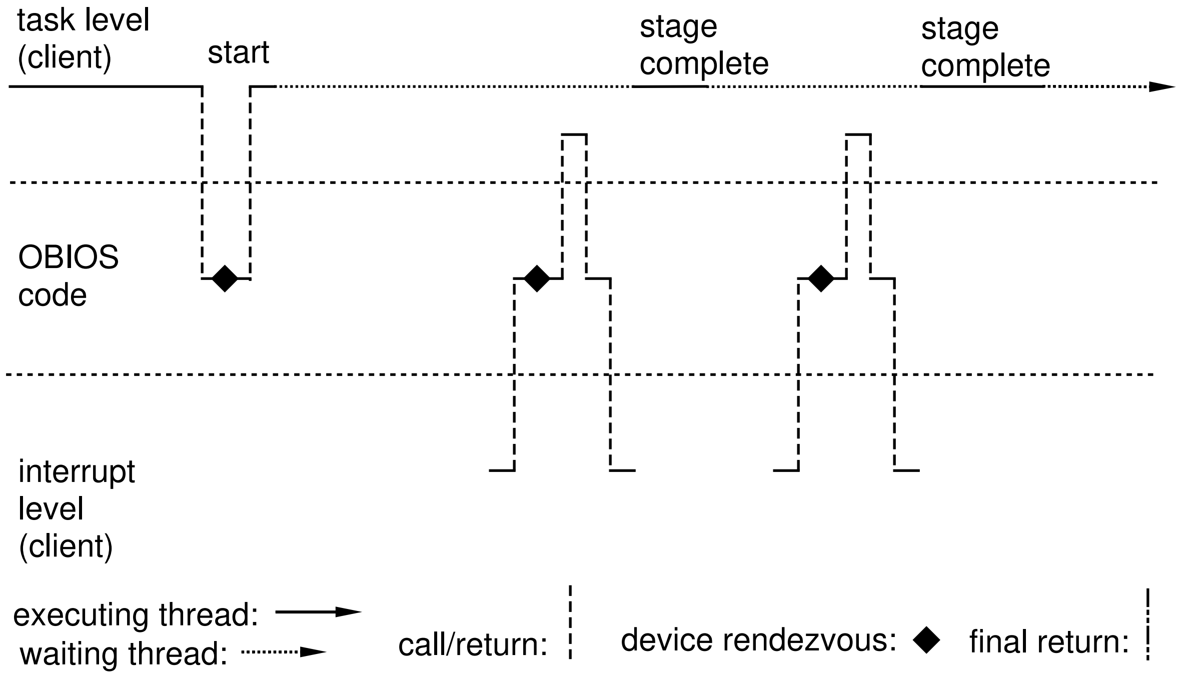

Figure 6 — Continuous Stage, Interrupt Service Mode.

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 OBIOS requests are active (i.e., from their initiation until their completion or abortion) the information recorded in the request block is *shared* between the client and the OBIOS. 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.

— Module instance data. This data structure maintains the internal OBIOS state of a module. Only a small initial portion of this structure is meant to be interpreted by the client. Module function code will be shared by instances of the same module class.

— Unit instance data. This data structure maintains the internal OBIOS state of a functional unit. Only a small initial portion of this structure is meant to be interpreted by the client. Unit function code will be shared by instances of the same unit class.

— A set of usage rules. These rules state when certain functions may or may not be invoked. Failure to comply with these rules defines any subsequent results to be erroneous. Compliance with these rules does not guarantee reasonable results – for example, the rules do not specifiy that a client shall set the baud rate before reading or writing a serial unit for the first time even though failure to do so may result in the inability to successfully read or write data. Compliance does, however, avoid erroneous conditions. OBIOS implementations are not required to monitor compliance.

2.2.1 Request Blocks

Request blocks (RBs) contain the following information:

a) Staging mode (in).

b) Timeout value (out).

c) Client call-back functions (in).

d) *Function-specific information* (in and/or out).

The first three items are defined in a common manner for all functional units and services. The last item is unit and, potentially, service specific. The exact structure and valid values for the fields of the request block for each class of units is specified in Chapter 3.

Valid staging modes are multiple stage interrupt, multiple stage polled, continuous stage interrupt and continuous stage polled. This value is used to indicate the desired type of service. As mentioned earlier, a multiple stage service request will result in a single stage service, if feasible. The staging mode value shall not be changed while the service transaction is in progress.

The client call-back functions are used by the OBIOS to invoke client defined functions as the result of either a partial or full completion of an I/O transaction or a fatal I/O error. The client call-back functions are invoked from interrupt level and provide the only substantial mechanism by which the OBIOS communicates with client code at interrupt time. Non-substantial communication is provided by the OBIOS to the client at interrupt time via the return value from the OBIOS interrupt service code, but this return value is limited to indicating whether the interrupt was spurious.

The timeout value indicates the maximum expected duration of time until the next stage of a multiple-stage or continuous-stage service completes. After the timeout duration, it may be appropriate for the client to `abort` the transaction. In the case of the polled staging mode, the timeout may be interpreted as an indication of the suggested maximum waiting interval for the client before it makes the next `poll` call – the client may choose to poll more frequently, for example, for units such as disks, the client may find it desirable to poll at intervals that are one-half of the timeout value. This value may be updated by the OBIOS while the transaction is in progress, though such behavior is not typically expected. Further details on the timeout value are specified with each unit. The constraints on updating the function-specific information are specified with each functional unit.

2.2.2 Procedural Interfaces

Client interactions with an OBIOS 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 functions. 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 unit commands for each unit, the address of the instance data that defines each unit and the amount of memory that was actually consumed. For some modules the OBIOS may be able to make use of more memory than the amount requested. This may occur for modules that allocate internal OBIOS I/O buffers. The OBIOS shall return any memory that exceeds the maximum amount that it can productively use. After initialization, the client can proceed to make a series of calls to the functional units that comprise the module until the module is `terminated`.

The OBIOS does not check the validity of device addresses (though in some implementations it may indicate if the expected module type was positively identified at the specified address). The reception of an invalid or incorrect device address by the `get_module_info` or `initialize` functions may result in a hardware fault or other processor error.

Figures 7 and 8 show the state transitions that are allowed for modules and functional units. A module shall be `initialized` before any of its units are able to service a request. Requests for device interactions are then submitted via the unit commands. If a request is accepted, the unit command will return a value such as *ok* or *ok pending* is returned. If the request is not accepted because the device is busy with another request (and presumably incapable of processing another simultaneous request), the command returns *device busy*. If the request is not accepted because the device appears to be acting in an errant manner, *device failure* is

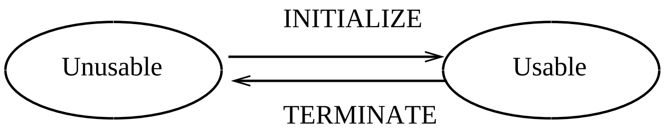

Figure 7 — Device State Transitions.

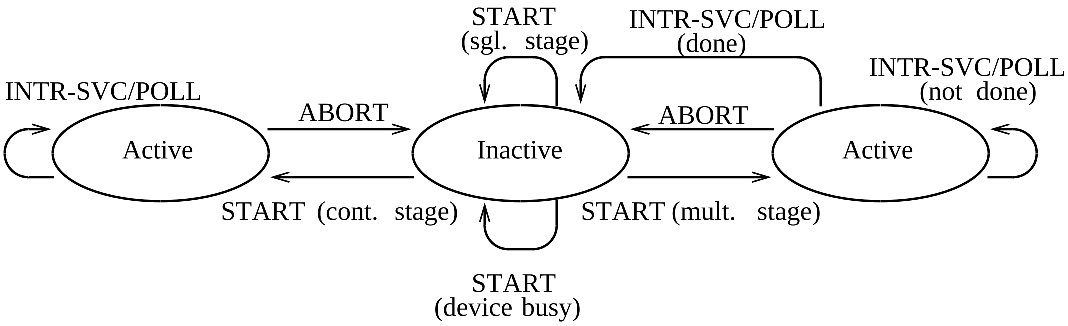

Figure 8 — Request State Transitions.

returned. Task level unit commands that check on service request status (`poll_any` and `poll`), as opposed to those that initiate services, shall return *device_busy* to indicate that a service is in progress but has not produced any new results since the last status check – that is, the current service stage is still busy and has not yet completed.

Each functional unit instance may define the maximum number of active requests that it can accommodate simultaneously. For example, a simple timer may be capable of having only one active request at any instant, conversely, a serial port will usually allow at least two active requests at any instant – a read and a write. Further, a given functional unit instance, such as one for a Ciprico 3550 SCSI disk controller, may allow a different number of active requests than some other instance of a similar or different disk controller. This information is available to the client in the functional unit instance data structure.

If a unit command returns *ok*, as opposed to *ok pending*, the request was completed as a single-stage service. Thus, any data transfer between the request block and device that was expected, occurred before return from the unit command.

If a unit command returns *ok pending*, the request for service cannot be completed immediately. It will complete sometime in the future, thus, the request is for either a multiple stage or continuous stage service. In this case, in addition to a status of *ok pending*, the command shall also return a timeout value. In general, this timeout value will indicate the maximum time that is expected to lapse for normal completion. The exact interpretation of this timeout value may be further specified on a unit by unit basis. Generally, the timeout value can be used to determine how long a client should wait for a response from the device before it assumes the request has failed. When a failure is assumed the client may call the `abort` function to terminate the transaction. The request block passed to `abort` should be the same one that was passed to the unit command that initiated the I/O activity. 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` shall then be called again to see if the termination request has been completed.

If the staging mode specified to a unit command indicates interrupt, stages will transition on device interrupts. If the staging mode specified to a unit 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 unit service request 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 pending* is received from `poll`, one stage of the request has completed and any expected (per stage) data transfer between the request block and device has already occurred also, a new timeout value is returned. If a status of *ok* is received from `poll`, the entire I/O transaction has completed and any expected data transfer between the request block and device has occurred.

In the interrupt case, upon receipt of the appropriate device interrupt, the client's ISR is expected to call the `intr_svc` function to formally acknowledge the interrupt. The `intr_svc` call is made to the function associated with the interrupting functional unit (or module) vector. A pointer to an opaque data structure is passed as an inbound argument to the `intr_svc` routine. If the interrupt is the result of an I/O stage completion, the appropriate client call-back routine is invoked with a pointer to the corresponding request block as an argument (there are two call-back routines, one for stage completion, the other to indicate that the entire I/O operation is complete). The location of the `intr_svc` function 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 stage completion, a new timeout value is also returned.

In summary, the return status from either the `poll` or `poll_any` functions may indicate *ok*, *ok pending*, *device busy*, or *device failure*. *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 timeout period). *Device busy* indicates that the current stage is not yet completed. *Device failure* implies something fatal went wrong. In interrupt mode, one of three specific client call-back functions will be invoked by the OBIOS to indicate either the completion of the entire I/O request, the completion of a single stage of the request, or the occurrence of a fatal I/O error.

2.2.2.1 State Transition Error Checking

In deference to performance, the OBIOS makes no attempt to prevent clients from invoking senseless sequences of commands. For example, an OBIOS need not attempt to prevent a client from using a unit before its subsuming module is initialized, nor need it detect if a client is attempting to send characters from a serial unit prior to setting the baud rate.

2.2.3 Reentrancy

The execution of all module and unit functions except `intr_svc` shall be mutually exclusive and non-reentrant. For example, at most one thread of execution is expected to be in any of these functions for a given serial functional unit at any given point in time. In order for an OBIOS to remain operating system independent and be usable in a multiprocessor environment it shall rely upon the client to enforce the exclusive access requirements of the mutually exclusive and non-reentrant functions. Furthermore, it is required that calls to `intr_svc` be made only from interrupt level and that calls to all other functions be made only from task level. Further details are given in Section 3.8.

2.3 OBIOS Data Structures

In addition to the request block structure (described in Section 2.2.1), an OBIOS implementation will comprise just three primary types of data: module information data, module instance data and functional unit instance data.

A pointer to a module information data area, which is allocated by the client, is provided to the OBIOS for each module when the `get_module_info` call is made. It provides the following information:

a) Module class (i.e., type of hardware).

b) Module version number and revision level.

c) The number of units in the module.

d) The interrupt vector requirements of the module.

e) Pointers to the module functions.

f) The memory requirements of the module.

The data in the above structure is often constant. However, for modules which can support an arbitrary number functional units (such as some types of disk controllers) the memory and vector requirements in the above structure may be based on a dynamic check of the number of units that are physically present. 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 OBIOS functions.

A functional unit instance data area is created for each unit contained in a module when the module is `initialize`d. 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 OBIOS functions. The non-visible, or *opaque* data is not required to have any specific format. The following information shall be in the client visible portion of the functional unit instance data areas:

a) Unit class (i.e., type of hardware).

b) Unit version number and revision level.

c) Pointer to unit functions.

d) Pointer to the module instance data of 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 function pointer array.

2.4 OBIOS Interfaces

An OBIOS presents two groups of interfaces to its clients, module functions and functional unit functions.

Five module functions are provided to clients. The interface and behavior of these functions should be the same for all classes of modules (e.g., Z8530, MC68901, EXOS202, etc). The location of at least the `get_module_info` function shall be known by the client for each module in the system. The addresses of all module functions shall be included as part of the information returned by the `get_module_info` function. Thus, the client may determine the locations of all of the remaining module and unit functions (see below) at run time. The client is also free to statically link to all necessary module and unit functions prior to run time. The implementations for the functions in each module class are expected to be different. Distinct instances from the same class shall share code, but operate on different module instance data areas.

The required unit functions 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 functions are included as part of the client visible unit instance data. The implementations for the functions in each unit subclass are expected to be different, however the call interfaces for the respective functions are the same. Distinct instances from the same subclass shall share code, but operate on different unit instance data areas.

2.5 Summary

An OBIOS can be summarily described as follows:

— it is a passive server,

— it provides I/O mechanisms and avoids making policy decisions,

— it always acts in the context of a client's thread of execution,

— it is called from supervisor mode,

— it assumes that mutual exclusion, where necessary, is ensured by the client.

The procedural interface is comprised of module and unit functions. The module functions are summarized below.

`get_module_info` — returns a pointer to static information about a module. `get_module_info` calls are made on modules.

`initialize` — initialize a module, put units into a quiescent state. `Initialize` calls are made on modules.

`terminate` — remove a module and its units from service. `Terminate` calls are made on modules. The client shall ensure that all activity has terminated for each unit within the module before calling `terminate`.

`get_module_status` — returns information about the dynamic state of a module. `Get_module_status` calls are made on modules.

`poll` — provides a subsequent rendezvous for a previously initiated module operation.

Unit service requests are made to initiate service transaction with a device unit. They provide the first (and sometimes only) rendezvous point between client and device unit. These functions are summarized below.

`abort` — fatally terminates an active functional unit service transaction.

`reset` — returns a functional unit to its initial quiescent state.

`poll` — subsequent rendezvous point, polled mode. `Poll` calls are made on functional units, they specify a particular service transaction.

`poll_any` — subsequent rendezvous point, polled mode. `Poll_Any` calls are made on functional units, they do not specify a particular service transaction.

`intr_svc` — subsequent rendezvous point, interrupt mode. `Intr_svc` calls are made to the OBIOS functions that are provided to the client during module initialization.

unit specific — each unit class will define additional functions that are appropriate for the functionality of the unit.

There are five basic modes of client/OBIOS service interaction. These five service modes are distinguished by the number of rendezvous required to reach completion and the method by which subsequent rendezvous are triggered:

Single-stage — rendezvous once with device.

Multiple-stage/polled mode — repeat rendezvous until first I/O.

Continuous-stage/polled mode — repeat rendezvous until explicitly aborted.

Multiple-stage/interrupt mode — repeat rendezvous until first I/O.

Continuous-stage/interrupt mode — repeat rendezvous until explicitly aborted. of times.

2.6 Notes on this Version of the Document

The from version 1.15 to 1.16 include

a) Discussion of the addition of I/O completion callbacks, see also the Section 3 changes.

b) Changed `info` function description to state that the OBIOS will always copy data to client allocated area.

3 General Specification

This Chapter provides an explicit specification of the portion of the OBIOS standard that is common for all module and unit classes. It does not cover implementation issues, these are discussed in in Appendix ??.

The exact specification is given by a combination of both the narrative text and the included C header file code. In general, the text will concentrate on the semantic aspects of the specification and the header file code will concentrate on the syntactic details.

3.1 Data Structure Formats

There are three data structures that are visible to the client. One contains the static module information, the second contains a portion of the unit instance data, the third is the request block which contains the data that is shared between the client and OBIOS for the duration of an active service transaction. All of these data structures refer to type definitions that are present in the file `bios.h`. The `bios.h` file includes two other files, `bios_module.h` and `bios_unit.h`. The `bios.h` file is shown on page 21

The structure of a request block consists of two parts, the first is a general part that is present in all request blocks, it is defined by the bios_req_blk structure in the file `bios_module.h`, which is shown on page 23. The second is defined by the class to which the unit belongs (e.g., serial, network). The unit class specific part of the request block formats are defined later with each unit class.

The general part of the request block includes the mode field, client call-back pointers and timeout field. Valid staging modes are multiple stage interrupt, multiple stage polled, continuous stage interrupt and continuous stage polled. This value is used to indicate the desired type of service. The staging mode value shall not be changed while the service transaction is in progress.

There are three client call-back functions which are used by the OBIOS to invoke client defined operations as the result of either a partial or full completion of an I/O transaction, or a fatal I/O error. The client call-back functions are invoked from interrupt level and provide the only substantial mechanism by which the OBIOS communicates with client code at interrupt time(see Section 3.3). Non-substantial communication is provided by the OBIOS to the client at interrupt time via the return value from the OBIOS interrupt service code, but this return value is limited to indicating whether the interrupt was spurious (see Section 3.6.7).

The timeout value indicates the maximum expected duration of time until the next stage of a multiple-stage or continuous-stage service completes. After the timeout duration, it may be appropriate for the client to `abort` the transaction. In the case of the polled staging mode, the timeout may be interpreted as an indication of the suggested maximum waiting interval for the client before it makes the next `poll` call – the client may choose to poll more frequently, for example, for units such as disks, the client may find it desirable to poll at intervals that are one-half of the timeout value. This value may be updated by the OBIOS while the transaction is in progress, though such behavior is not typically expected. Further details on the timeout value are specified with each unit. The constraints on updating the function-specific information are specified with each functional unit.

Finally, the general part of the request block structure includes two fields of type `void*`, one each for the client and OBIOS. The use and interpretation of the client field is at the sole control of the client, the OBIOS should not modify nor interpret its contents. Similarly, the use and interpretation of the OBIOS field is at the sole control of the OBIOS, the client should not modify nor interpret its contents.

The static module information is defined by the structure `bios_module_info` in the file `bios_module.h`, which is shown on page 23. The `memory_needed` field of the `bios_module_info` structure indicates the amount of memory that is required for all of the units that may be physically present. For modules in which units may be easily added or removed, this value may be calculated assuming the maximum number of units. However, it is preferable that the OBIOS code check at run time to see how many units are physically present before calculating the memory requirements. The `num_intr` field indicates the number of interrupt vectors that the module is capable of generating. The `start_vector` field indicates the number of the *preferred* interrupt base – that is, if multiple interrupt vectors are called for, they should be contiguous starting with `start_vector`. This value is meant to provide a suggestion to the client, many devices will work with arbitrary vector numbers, however, for those that don't this value will indicate at least one base vector number that should work. The OBIOS will indicate to the client the provision unacceptable interrupt vector values during module initialization. The `num_units` field indicates the total number units in the module. The `id_confirm` field indicates whether the identity of the corresponding hardware device has been positively confirmed, positive confirmation is not possible on all systems. The `ident` field provides a pointer to a character string that should textually describe the module. This last field is intended to be used for informational purposes when printing log or error messages, its content is unique for each class of module. See Section 3.5 for further details.

The unit instance data is specified by two types of structures, a generic unit data structure and a class specific unit data structure. It is possible for a client to receive a pointer to a class specific unit data structure prior to it knowing the specific class to which the unit instance belongs. In such cases, the client can access a subset of the unit data through the generic structure. The subset that is accessible includes the class of the unit. The generic structure is defined by `bios_unit` in the file `bios_unit.h` which is shown on page 25.

Once the class of the unit is known, the client can use the same unit instance data pointer along with a class specific structure to access the remainder of the client visible data. Unit class specific structures are defined later with their respective units.

An OBIOS implementation will typically include the client visible module and unit instance data structures inside of other structures. These outer structures are visible only by a given OBIOS module and its related functions, thus they are not defined by the OBIOS specification.

3.2 Return Values

The OBIOS return values can be divided into two groups: normal and abnormal. The normal group is used to describe the transitioning of OBIOS I/O transactions through their, possibly multiple, stages. This group is comprised of:

— BIOS_RET_OK

— BIOS_RET_OK_PENDING

— BIOS_RET_DEV_BUSY

The normal return values will typically be augmented with status information that is conveyed in the request block structure. This information may indicate successful I/O progress as well as non-fatal (recoverable) I/O errors – examples include dropped packets, receiver overruns and so forth.

The abnormal group is used to describe errors that are not recoverable with respect to either the I/O request or the whole I/O unit. As such, these return values may be rather precise since their decoding is not time critical. Depending on the return value, an abnormal error may be followed the following range of actions: (a) the retrying of a specific request (perhaps with corrected parameter data), (b) the abortion on ongoing I/O transactions, (c) the resetting of a device unit, and so forth.

It should be noted that once an I/O request has been accepted for service only errors that affect the unit as a whole should be possible. Individual I/O request errors should either manifest themselves when the I/O is initiated (e.g., bad initial parameter values, resource shortage and so forth), or they should be recoverable (e.g., read error and so forth).

3.3 I/O Completion Call-backs

There are three client call-back functions which are used by the OBIOS to invoke client defined operations at interrupt time. The three functions, listed below, will be invokes as the result of either a partial or full completion of an I/O transaction, or a fatal I/O error, respectively.

`io_complete` — is invoked with a pointer to the corresponding request block as a parameter. It indicates that the I/O request has completed. Additional information concerning the status of the I/O request may be found in the request block. This call-back serves the analogous purpose to the *ok* return value for the `poll` function.

`io_stage_complete` — is invoked with a pointer to the corresponding request block as a parameter. It indicates that a stage of the I/O request has completed. Additional information concerning the status of the I/O request may be found in the request block. This call-back serves the analogous purpose to the *ok pending* return value for the `poll` function.

`io_error` — is invoked with a `bios_ret_t` parameter. It indicates that a fatal I/O event has occurred. The exact cause is provided in the `bios_ret_t` parameter. It should be noted that once an I/O request has been accepted for service only errors that affect the unit as a whole should be possible. Individual I/O request errors should either manifest themselves when the I/O is initiated (e.g., bad initial parameter values, resource shortage and so forth), or they should be recoverable (e.g., read error and so forth). This call-back serves the analogous purpose to the *abnormal error* return values for the `poll` function.

3.4 Module Functions

There are five functions that shall be provided by each OBIOS module. The function prototypes for these functions as they appear in the OBIOS module function pointer table are given in the file `bios_module.h`. A module shall be initialized before any of its units can be used. Prior to initialization and after termination the only two module functions that may be used are `initialize` and `get_module_info`. In cases where device response latency is very high (e.g., remote intelligent devices) a module call may return with an *ok pending* value. When an *ok pending* value is returned, the client is required to poll the module with the `poll` call until a successful completion is achieved. During this time no other calls may be made to the module. The OBIOS is not required to perform any parameter error checking. Any module call that is made with invalid or incorrect parameters is considered erroneous and the results of such a call (as well as any side effects that it may cause) are undefined.

3.4.1 Info

Synopsis

```
bios_ret_t
(*info)
   (/* in  */ void* configP,
    /* out */ bios_module_info_pt infoP);
```

Description

The `configP` parameter is the same as the one documented below for the initialize command. The `infoP` parameter points to a client allocated data structure into which the OBIOS will copy data that describes the module. The structure is defined in the file `bios_module.h`, and should be treated as read-only by the client.

Return Values

`BIOS_RET_DEV_FAIL` — no action taken, device specific error (see implementation for details).

`BIOS_RET_OK_PENDING` — not yet complete, further rendezvous required.

`BIOS_RET_OK` — successful completion.

3.4.2 Initialize

Synopsis

```
bios_ret_t
(*initialize)
  (/* in    */ void* moduleP,
```

```
/* in/out */ INT32* module_sizeP,
/* in     */ void* configP,
/* in     */ UINT32 start_vector,
/* out    */ bios_unit_pt unit_tblP[],
/* out    */ bios_intr_svc_t intr_tblP[]);
```

Description

The moduleP parameter should point to a block of memory of module_sizeP. This memory is for the sole use of the OBIOS, its contents are not directly visible to the client. The module_sizeP parameter is used by the client to indicate the size (in bytes) of the moduleP block. The client may provide more memory than the bios_module_info structure indicates is needed. Certain types of units may be able to make productive use of the extra memory (e.g., for hardware accessible buffers). The module_sizeP parameter is also used by the OBIOS to indicate the actual number of bytes that are used. Any unused bytes, which will be contiguous and located at the end of the moduleP block, may be reclaimed by the client.

The configP parameter points to a configuration data area. The structure of this data area is individually defined for each type of module in the bios_mname.h file (where mname is a self descriptive name for the corresponding module). This structure holds information that is specific to a given instance of the module. Typically, this information includes the address or addresses at which the module and its units reside. Additionally, in the case of modules with serial units for example, it may contain input clock frequencies. The basic structure that is defined for each type of module may be augmented with additional OBIOS vendor defined information. See Section ?? for further details.

The addresses parameter is used to indicate the location of the device registers in the memory (or I/O) space of the CPU. The specific requirements for this parameter are described with each module. The start_vector parameter indicates the base interrupt vector number that the device should use. The start_vector parameter should be aligned on a boundary that corresponds to the value of the num_intr field of the bios_module_info structure. Such an alignment is not required, however, many drivers that use a block of vectors will return an error if the vectors are not aligned in this manner. The block of vectors that will be used by the OBIOS starts at start_vector and continues for a length of num_intr.

The unit_tblP parameter specifies a table of pointers to bios_unit structures that are to be filled in by the OBIOS. The OBIOS will typically allocate the bios_unit structures from the moduleP block of memory. Pointers to these structures are then returned to the client by the unit_tblP parameter. The length of the table is given by the num_units field of the bios_module_info structure.

The intr_tblP parameter specifies a table of pointers to bios_intr_svc structures that are to be filled in by the OBIOS. These structures contain two fields, one is a function pointer, the other is a *magic cookie* (i.e., a parameter that is opaque to the client). In reality, the magic cookie will likely be a pointer to either the unit or module instance data structure – however, the OBIOS specification does not specify the contents of this field. The length of the intr_tblP table is given by the num_intr field of the bios_module_info structure. The intr_tblP entries are in one-to-one correspondence with the interrupt vectors in the same module. When an interrupt occurs for a given vector, the function pointer in the corresponding entry of the intr_tblP should be called with its magic cookie as a parameter. At least one interrupt vector is required for all modules that are capable of interrupting. For systems that do not support vectored interrupts, it is the responsibility of the client to determine the source of the interrupt (if need be, by polling each unit) and call the corresponding function.

All of the units within a module shall be in a quiescent state after module initialization.

The expected physical hardware module may not be present at the address indicated by the initialize call. In such cases, the OBIOS may indicate this event by either an error return value or, less gracefully, by causing a hardware exception.

Return Values

BIOS_RET_INSUFFICIENT_MEM — no action taken, insufficient memory was provided.

BIOS_RET_DEV_FAIL — no action taken, device specific error (see implementation for details).

BIOS_RET_OK_PENDING — not yet complete, further rendezvous required.

BIOS_RET_OK — successful completion.

3.4.3 Status

Synopsis

```
bios_ret_t
(*status)
  (/* in     */ void* moduleP,
   /* out    */ INT32 num_req_blks[]);
```

Description

The moduleP parameter provides the module instance data to the OBIOS. The num_req_blks parameter specifies a table of integers that will be filled in by the OBIOS. Each entry corresponds to a unit within the module, the numerical value indicates number of requests that are active on each unit. The order of entries is the same as the order in the unit_tblP parameter of the module initialize function.

Return Values

BIOS_RET_OK_PENDING — not yet complete, further rendezvous required.

BIOS_RET_OK — successful completion.

3.4.4 Terminate

Synopsis

```
bios_ret_t
(*terminate)
  (/* in     */ void* moduleP);
```

Description

The moduleP parameter provides the module instance data to the OBIOS. Clients shall ensure that there is no module activity (i.e., that each unit is in a quiescent state) prior to issuing a terminate call. The terminate function will place the module into a quiescent state. All further module interactions are undefined until a subsequent initialize call is made.

Return Values

BIOS_RET_OK_PENDING — not yet complete, further rendezvous required.

BIOS_RET_OK — successful completion.

3.4.5 Poll

Synopsis

```
bios_ret_t
(*terminate)
  (/* in     */ void* moduleP);
```

Description

The moduleP parameter provides the module instance data to the OBIOS. The poll function will check for the completion of a previously initiated module request.

Return Values

BIOS_RET_INV_REQ_DATA — no outstanding request exists for the module.

BIOS_RET_OK_PENDING — not yet complete, further rendezvous required.

BIOS_RET_OK — successful completion.

3.5 Module Naming

The call interface to the functions of the currently defined modules are provided in the bios_mname.h files, where mname is a self descriptive name for the corresponding modules (e.g., bios_z8530.h). These files shall contain only function call prototypes for the benefit type checking client code that chooses to statically link to bios functions prior to run time.

— Zilog Z8530 Serial Communication Controller, by RTC Jan 1991

— Advanced Micro Devices AM7990 Local Area Network Controller, by RTC Jan 1991

Names for the defined modules (see Section 3.1, ident) shall follow the form of beginning with the name of the manufacturer of the module, followed by the modules part name, followed by a brief descriptive title and the name of the person or organization providing the implementation. The descriptive title and implementation providers name shall provide the means to distinguish multiple variations of an OBIOS implementation for a single device. The characters in this string should comprise a printable ANSI C string.

The OBIOS standard endeavors to facilitate unique module naming. Unique names are considered to be a useful convenience but are not strictly required. If desired, however, unique names may be obtained for one dollar (U.S.) by beginning the module name string with the serial number from a (U.S.) dollar bill – the dollar bill should then be removed from circulation (perhaps by burning). Any module names that do not include this unique identifier must not start with the two character sequence of upper case letter followed by numeral.

3.6 Unit Functions

The functions described in this section shall be provided by each OBIOS module. The function prototypes for these functions as they appear in the OBIOS module function pointer table are given in the file bios_module.h. Unit functions always return a value of type bios_ret_t. Full function names are of the form:

```
bios_ret_t
bios_mname_cname_function(...)
```

where mname is the name of the module and cname is the name of the unit class and function describes the purpose of the function. Full names appear in the source code to provide uniqueness in order to avoid name conflicts in a library containing several OBIOS drivers.

All unit functions that may interact with their corresponding hardware device shall contain a request block parameter. This allows the OBIOS interface to accommodate systems with high-latency I/O devices (see Section 3.7) for more details. The client provided mutual exclusion protection protects only against other task level accesses. Thus the OBIOS shall invoke and release protection around all critical regions within the unit functions to protect them from simultaneous access from interrupt level. The OBIOS is expected to invoke this protection, instead of the client, because it may do so across a much smaller span of code (see Section 3.8 for further details). It is erroneous to call reset while any transaction is active on the same unit, therefore, it will not contain any critical regions (see Section 3.8). The OBIOS is not required to perform any parameter error checking. Any module call that is made with invalid or incorrect parameters is considered erroneous and the results of such a call (as well as any side effects that it may cause) are undefined.

3.6.1 Abort

Synopsis

```
bios_ret_t
(*abort)
  (/* in     */ bios_unit_pt unitP,
   /* in     */ bios_req_blk_pt req_blkP);
```

Description

The `unitP` parameter provides the unit instance data to the BIOS. The `req_blkP` parameter points to the request block that represents the activity that is to be aborted.

Return Values

`BIOS_RET_INV_REQ_DATA` — request block pointer is not valid.

`BIOS_RET_OK_PENDING` — not yet complete, further rendezvous required.

`BIOS_RET_OK` — successful completion.

3.6.2 Poll

Synopsis

```
bios_ret_t
(*poll)
  (/* in     */ bios_unit_pt unitP,
   /* in     */ bios_req_blk_pt req_blkP);
```

Description

Poll on the status of a specific indicated transaction. The `unitP` parameter provides the unit instance data to the OBIOS. The `req_blkP` parameter points to the request block that represents the activity that is being polled for. In this function, the OBIOS will interact with the physical device and, possibly, update the state of the device and the OBIOS data structures. Upon return, any change in the state of the I/O operation will be reflected in the return value and/or the request block. The exact semantics of these functions are highly dependent on the unit class, for further details see the corresponding sections for each unit class. It is erroneous to pass to this function a request block for which a transaction has either not previously been started or has already completed.

Return Values

`BIOS_RET_DEV_BUSY` — No change in status since last poll (stage has not completed).

`BIOS_RET_DEV_FAIL` — Catastrophic device failure has been detected (device should be reset).

`BIOS_RET_OK_PENDING` — Stage has completed.

`BIOS_RET_OK` — Entire I/O request has completed.

3.6.3 Poll Any

Synopsis

```
bios_ret_t
(*poll_any)
  (/* in     */ bios_unit_pt unitP,
   /* out    */ bios_req_blk_pt* req_blkPP);
```

Description

Poll on the status of any transaction. The `unitP` parameter provides the unit instance data to the OBIOS. The `req_blkPP` parameter provides a location in which to return a pointer to the request block that represents any one completed activity. The semantic action of this function depends on whether the unit is enabled.

If the unit is completely enabled (for both input and output) the OBIOS searches for any completed transaction stage, if successful, a pointer to the corresponding request block is returned in the `req_blkPP` parameter. The function result value is the same as what would have been returned if a `poll` had been made specifically for the reqest block pointed to by `req_blkPP`. If no completed transaction stages exist but there are active transactions, a null value is returned in the `req_blkPP` parameter and the return value indicates `device busy`. If there are no active transactions, a null value is returned in the `req_blkPP` parameter and the return value indicates `ok`.

If the unit is completely disabled a pointer to the request block of any one transaction is returned in the `req_blkPP` parameter and *ok pending* is provided as the return value. If no active requests exist a null value is returned in the `req_blkPP` parameter and *ok* is provided as the return value.

If the unit is not either completely enabled or disabled the `req_blkPP` parameter is set to null and *undefined* is provided as the return value.

Return Values

`BIOS_RET_UNDEFINED` — if unit is not either completely enabled or completely disabled.

`BIOS_RET_OK_PENDING` — if unit is disabled and more transactions remain.

`BIOS_RET_DEV_BUSY` — if unit is enabled and more transactions remain, but none have a completed transaction stage.

Same as for poll function — if unit is enabled and a completed transaction stage exists.

`BIOS_RET_OK` — successful completion, if no other transacitons remain.

3.6.4 Status

The declaration of the OBIOS unit status function pointer is given below.

```
bios_ret_t
(*status)
  (/* in     */ bios_unit_pt unitP,
   /* in/out */ bios_req_blk_pt req_blkP,
   /* out    */ bios_req_blk_pt req_blkPP[],
   /* out    */ bios_unit_status_pt statusP);
```

Description

The unitP parameter provides the unit instance data to the OBIOS. The req_blkP parameter provides the request block to the OBIOS. The req_blkPP parameter points to an array of request block pointers that the OBIOS will fill in. The length of the req_blkP array should be the value indicated in the max_requests field of the bios_unit structure for the given unit. The OBIOS will fill this array from the beginning with pointers to active request block the first of any unused array slots will contain a NULL pointer. The statusP parameter will be filled in by the OBIOS to indicate whether the unit is enabled and whether unit interrupts are enabled.

Return Values

BIOS_RET_OK_PENDING — not yet complete, further rendezvous required.

BIOS_RET_OK — successful completion.

3.6.5 Set State

Synopsis

```
bios_ret_t
(*set_state)
  (/* in     */ bios_unit_pt unitP,
   /* in/out */ bios_req_blk_pt req_blkP,
   /* in     */ bios_unit_status_t state);
```

Description

The unitP parameter provides the unit instance data to the OBIOS. The req_blkP parameter provides the request block to the OBIOS. The state parameter indicates desired setting for unit enable/disable and unit interrupt enable/disable. The exact representation of this parameter is specified individually for each unit class (see the bios_unit.h header file).

Return Values

BIOS_RET_OK_PENDING — not yet complete, further rendezvous required.

BIOS_RET_OK — successful completion.

3.6.6 Reset

Synopsis

```
bios_ret_t
(*reset)
  (/* in     */ bios_unit_pt unitP,
   /* in/out */ bios_req_blk_pt req_blkP);
```

Description

The unitP parameter provides the unit instance data to the OBIOS. The req_blkP parameter provides the request block to the OBIOS. The client shall ensure that there are no active requests for the unit before calling this function. Upon return the unit should be in the same state that it was in immediately after module initialization.

Return Values

BIOS_RET_OK_PENDING — not yet complete, further rendezvous required.

BIOS_RET_OK — successful completion.

3.6.7 Interrupt Service

For architectures with vectored interrupts a OBIOS unit will typically provide one interrupt service function for each separate type of interrupt (i.e., vector) that a unit can generate. OBIOS implementations for architectures with a single interrupt service routine per device will likely provide one interrupt service function per unit. It is also possible that an implementation will have one interrupt function per module. In any case, the interface to this function is described in the bios_module.h header file (it is also shown below). Pointers to OBIOS interrupt handling functions and their corresponding *opaque* input parameter are passed to the client by the module initialization function. The synopsis below shows the structure that is passed to the client.

```
struct bios_intr_svc
{
  bios_intr_ret_t (*intr_func)
    (void* magic_cookie);
  void* magic_cookie;
};
```

Description

The OBIOS interrupt handler code should be invoked from the system's interrupt service routine shortly after the occurrence of a hardware interrupt. When invoked, the *magic cookie* will be passed as a parameter. The contents of the magic cookie is left to the discretion of the OBIOS implementor, it will typically be a pointer to the unit instance data. As a result of the interrupt level processing done by the OBIOS, one of three client provided call-back functions may be invoked. The io_complete function is invoked if there has been a non-final I/O stage transition. The io_stage_complete function is invoked if the I/O transaction is fully completed. The io_error function is invoked if a fatal I/O error has occurred. It is also possible that none of these three call-back functions are invoked. This may occur because the interrupting event is not significant to the client (see the appropriate unit-specific intr_svc function for details, if applicable), or because the interrupt was spurious with respect to the intr_svc function that was called.

Return Values

`BIOS_INTR_RET_OK` — interrupt was handled.

`BIOS_INTR_RET_SPURIOUS` — spurious interrupt.

3.6.8 Interrupt and Polled Operations

The OBIOS expects clients to use the interrupt and polled modes of operations in a consist manner. The client shall never initiate service requests in one mode when there are active service transactions of the other mode in progress on the same unit. Similarly, when changing from the use of one mode to the other on a unit, the client shall ensure that there are no active service transactions in progress on that unit. A client may poll an interrupt mode service transaction that is in progress. However, the OBIOS requires that clients not invoke an **intr_svc** function when polled mode service transactions are active on the same unit. As a result, OBIOS implementations need only protect the interrupt mode code of unit functions from interrupt level access.

Several OBIOS functions may change the interrupt enabled/disabled state of a unit. These include **set_state**, any call that starts I/O, **poll**, **poll_any** and **intr_svc**. The OBIOS does not check the current interrupt state before these commands modify the state. The changes that may be made by the latter three functions will depend on the mode (interrupt or polled) of the transaction that they are servicing.

3.7 High Latency I/O Support

All OBIOS unit functions require a request block parameter, even though in many implementations there will be several calls that do not make any use of this parameter. This parameter is required in order to provide a single consistent interface that will continue to work in systems with high latency I/O devices. Examples of such devices may be found in systems which make use of intelligent I/O devices that are accessed remotely via a bus or other system interconnect.

As an example, consider a simple operation like changing the baud rate on a serial line. This would often be a single stage operation that completes immediately – either successfully, if the request is for a supportable rate, or unsuccessfully if it is an unsupportable request. However, if the serial line device is controlled by an intelligent I/O controller (which may be located on some auxillary I/O bus) the confirmation of success or failure could take considerable time. In such cases, it is clearly desirable to use a multiple stage mode of service.

The OBIOS accommodates high latency devices with the same interface as low latency devices, clients of the OBIOS should not have to know the difference. This is accomplished by requiring all OBIOS unit functions to include a request block parameter. Requests for service from the OBIOS may specifiy one of four modes: multiple stage interrupt, multiple stage polled, continuous stage interrupt and continuous stage polled. Multiple stage requests may effectively degenerate to single stage services by completing immediately. Module functions support this same requirement (see Section 3.4).

3.8 Reentrancy and Multiprocessor Support

The execution of all module and unit functions except **intr_svc** on a given module or unit instance shall be mutually exclusive and non-reentrant. For example, at most one thread of execution shall be in any of these functions for a given serial functional unit at any given point in time. This level of protection is explicitly required by this specification and shall be provided by the client. Additionally, in order to achieve correct implementations, OBIOS functions that are called from task level may need to protect portions of their code (critical regions) from reentrancy by interrupt service routines – for example, when they are accessing a sequence of device registers. On a uniprocessor, this protection is typically provided by briefly disabling interrupts during. However, disabling interrupts does not protect against the simultaneous access of data that may occur by interrupt routines running on separate CPUs in a multiprocessor.

Much of this protection can be handled by the client. For instance, in a multiprocessor system the client ISR code can protect interrupt level OBIOS code from reentrancy by task level code running on a separate CPU by surrounding the OBIOS call with operating system calls to enter and exit a critical region. However, in order to minimize the amount of time that an interrupt service routine might have to spend waiting for a thread on another CPU to exit a critical region in a task level OBIOS function on a multiprocessor system or, in order to minimize the amount of time that interrupts are disabled on a uniprocessor system, the OBIOS task level functions must be able to access mutual exclusion primitives. In most systems these primitives are privided directly by the hardware architecture and may be used by OBIOS implementation code. In rare cases, this may not suffice. It is strongly recommended that OBIOS implementations follow the suggestions in Section **??** concerning the use of a standard set of macros for implementing and using mutual exclusion functions. This will help improve the portability of OBIOS implementation code and, if necessary, will aid in the interfacing to operating systems mutual exclusion support.

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.

3.9 Notes on this Version of the Document

The changes from version 1.15 to 1.16 include:

a) Changes to the **bios_req_|blk** structure to accommodate I/O completion and I/O stage completion call-backs. The primary justification for this addition is that it allows multiple requests (possibly from different units on the same module) to be handled on a single interrupt in a clean fashion.

b) Added a small section on call-backs.

c) Added a small section to explain return values. The bios_ret_t enumeration was simplified, to account for values needed only by task-level calls. All class specific non-fatal status info should be passed back in the request block. Return values for intr_svc functions are now a new type (bios_intr_ret_t). poll function return values also changed as a result of this.

d) Changes to the intr_svc function. It was simplified as a result of the addition of I/O completion call-backs.

e) A further attempt was made to clarify the reentrancy and multiprocessor support section, again.

f) Return values were defined for the module info function.

g) Changed infoP parameter of info function to be pointer to client allocated data area (instead of pointer to pointer), OBIOS will always copy data to client area.

h) Changed the define module section to be clearer and, hopefully, easier to use.

i) Changed poll function semantics (slightly) for the response to being called with a non-active request block pointer.

The manner in which the OBIOS performs critical region locking and unlocking needs to be looked at more closely.

```
/*********************************************************************
 *
 * Copyright 1990 Real Time Consortium
 *
 *   This program is free software; you can redistribute it and/or modify
 *   it under the terms of the RTC General Public License as published by
 *   the Real Time Consortium; either version 1, or (at your option)
 *   any later version.
 *
 *   This program is distributed in the hope that it will be useful,
 *   but WITHOUT ANY WARRANTY; without even the implied warranty of
 *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *   RTC General Public License for more details.
 *
 *   You should have received a copy of the RTC General Public License
 *   along with this program; if not, write to the Real Time Consortium,
 *   P.O. Box 61315, Sunnyvale, CA  94088, USA, or send e-mail to
 *   rtc@gate.ready.com.
 *   <end copyright notice>
 *
 * Module Name:          %M%
 *
 * Identification:       %Z% %I% %M%
 *
 *
 * Date:                 %G%  %U%
 *
 * Description:  This file and the file for each device class that
 *               is to be used comprise all of the header files that
 *               are needed to use the BIOS.
 *
 *********************************************************************/

#ifndef BIOS_H
#define BIOS_H

/* The following macros and type defs, which define basic types, */
/* should be modified for the target machine. */

#ifndef INT32
#define INT32   int
#endif
#ifndef UINT32
#define UINT32  unsigned int
#endif
#ifndef INT16
#define INT16   short
#endif
#ifndef UINT16
#define UINT16  unsigned short
#endif
#ifndef INT8
#define INT8    char
#endif
#ifndef UINT8
#define UINT8   unsigned char
#endif
#ifndef NULL
#define NULL    0
#endif
#ifndef TRUE
#define TRUE    1
#endif
#ifndef FALSE
#define FALSE   0
#endif


typedef int bios_boolean_t;

typedef UINT32 bios_addr_t;
typedef UINT32* bios_addr_pt;
```

```
/* useful type definitions */

/* BIOS return values.  The BIOS return values are divided into two */
/* classes, normal and abnormal.  Abnormal errors indicate a */
/* condition that is fatal with respect to the I/O request or the */
/* entire I/O unit. Abnormal error return values may be rather precise */
/* since their decoding is not time critical. */
typedef enum
{
  /* Normal return values. */
  BIOS_RET_OK = 0,
  BIOS_RET_OK_PENDING = 1,
  BIOS_RET_DEV_BUSY = 2,
  /* Abnormal (fatal) return values must be 4 or greater. */
  BIOS_RET_INSUFFICIENT_MEM = 4,
  BIOS_RET_UNSP_SVC,
  BIOS_RET_INV_MODE,
  BIOS_RET_INV_REQ_DATA,
  BIOS_RET_DEV_FAIL,
  BIOS_RET_IO_ERROR,
  BIOS_RET_DATA_SIZE,
  BIOS_RET_BAD_ADDRESS,
  BIOS_RET_UNDEFINED,
  BIOS_RET_BUFFER_EXCESS,
  BIOS_RET_BUFFER_DROUGHT,
  BIOS_RET_RESOURCE_ERROR,
  BIOS_RET_LOST_CARRIER
} bios_ret_t;

/* BIOS interrupt level return values.  Limited info is passed back to */
/* the client ISR, most communication is via the request block and */
/* client call-back mechanism. */
typedef enum
{
  BIOS_INTR_RET_OK,
  BIOS_INTR_RET_SPURIOUS,
} bios_intr_ret_t;



typedef bios_ret_t (*bios_func_pt) ();


/* Modes of service */
typedef enum
{
  BIOS_MULT_STAGE_POLL,
  BIOS_MULT_STAGE_INTR,
  BIOS_CONT_STAGE_POLL,
  BIOS_CONT_STAGE_INTR
} bios_mode_t;

/* Unit classes */

typedef enum
{
  /* To aid debugging, start enum with 1. */
  BIOS_SERIAL_UNIT = 1,
  BIOS_TIMER_UNIT  = 2,
  BIOS_NETWORK_UNIT = 3,
  BIOS_CLOCK_UNIT = 4
} bios_unit_class_t;

#include "bios/bios_module.h"

#endif   /* not BIOS_H */
```

```c
/*******************************************************************
 *
 * Copyright 1990 Real Time Consortium
 *
 *   This program is free software; you can redistribute it and/or modify
 *   it under the terms of the RTC General Public License as published by
 *   the Real Time Consortium; either version 1, or (at your option)
 *   any later version.
 *
 *   This program is distributed in the hope that it will be useful,
 *   but WITHOUT ANY WARRANTY; without even the implied warranty of
 *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *   RTC General Public License for more details.
 *
 *   You should have received a copy of the RTC General Public License
 *   along with this program; if not, write to the Real Time Consortium,
 *   P.O. Box 61315, Sunnyvale, CA  94088, USA, or send e-mail to
 *   rtc@gate.ready.com.
 *   <end copyright notice>
 *
 * Module Name:          %M%
 *
 * Identification:       %Z% %I% %M%
 *
 * Date:                 %G%  %U%
 *
 * Description:  Module interface.
 *
 * History:      07/25/90          Added request block definition
 *                                 King Chou
 *
 *******************************************************************/

#ifndef BIOS_MODULE_H
#define BIOS_MODULE_H

struct bios_module_info;
typedef struct bios_module_info bios_module_info_t;
typedef struct bios_module_info* bios_module_info_pt;

/* General Request Block format */

struct bios_req_blk
{
  bios_mode_t mode;
  UINT32 timeout; /* in microseconds */
  void  (*io_complete)
    (bios_req_blk_pt req_blkP); /* I/O (final) completion call back routine */
  void  (*io_stage_complete)
    (bios_req_blk_pt req_blkP); /* I/O stage completion call back routine */
  void  (*io_error)
    (bios_req_blk_pt req_blkP); /* fatal I/O error call back routine */
  void* client; /* unrestricted client use */
  void* bios; /* unrestricted bios implementation use */
};

typedef struct bios_req_blk bios_req_blk_t;
typedef struct bios_req_blk* bios_req_blk_pt;

struct bios_intr_svc
{
  bios_ret_t (*intr_funcP)(void* magic_cookie)
  void* magic_cookie;
};

typedef struct bios_intr_svc bios_intr_svc_t;
typedef struct bios_intr_svc* bios_intr_svc_pt;


#include "bios/bios_unit.h"
```

```
struct bios_module_functions
{
  bios_ret_t (*info)(/* in  */ void* configP,
                     /* out */ bios_module_info_pt infoP);

  bios_ret_t (*initialize)(/* in     */ void* moduleP,
                           /* in/out */ UINT32* module_sizeP,
                           /* in     */ void* configP,
                           /* in     */ UINT32 start_vector,
                           /* out    */ bios_unit_pt unit_tblP[],
                           /* out    */ bios_intr_svc_t intr_tblP[]);

  bios_ret_t (*terminate)( /* in     */ void* moduleP);

  bios_ret_t (*status)(/* in  */ void* moduleP,
       /* out */ UINT32 num_req_blks[]);

  bios_ret_t (*poll)(/* in  */ void* moduleP);
};

typedef struct bios_module_functions bios_module_functions_t;
typedef struct bios_module_functions* bios_module_functions_pt;


struct bios_module_info
{
  bios_module_functions_pt funcP;        /* module functions */
  UINT32 memory_needed;                  /* memory needed */
  UINT32 num_intr;                       /* number of interrupt vectors */
  UINT32 start_vector;                   /* preferred base intrpt vector number */
  UINT32 num_units;                      /* number of units in module */
  UINT16 major_rev;                      /* major revision number */
  UINT16 minor_rev;                      /* minor revision number */
  bios_boolean_t id_confirm;             /* hardware device ID confirmed */
  char* ident;                           /* module identifier text */
};

#endif /* !BIOS_MODULE_H */
```

```
/*******************************************************************
 *
 * Copyright 1990 Real Time Consortium
 *
 *   This program is free software; you can redistribute it and/or modify
 *   it under the terms of the RTC General Public License as published by
 *   the Real Time Consortium; either version 1, or (at your option)
 *   any later version.
 *
 *   This program is distributed in the hope that it will be useful,
 *   but WITHOUT ANY WARRANTY; without even the implied warranty of
 *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *   RTC General Public License for more details.
 *
 *   You should have received a copy of the RTC General Public License
 *   along with this program; if not, write to the Real Time Consortium,
 *   P.O. Box 61315, Sunnyvale, CA  94088, USA, or send e-mail to
 *   rtc@gate.ready.com.
 *   <end copyright notice>
 *
 * Module Name:          bios_unit.h
 *
 * Identification:       @(#) 3.3 bios_unit.h
 *
 * Date:                 3/13/91  12:45:08
 *
 * Description:          Generic unit interface.
 *
 ******************************************************************/

#ifndef BIOS_UNIT_H
#define BIOS_UNIT_H

union bios_unit_status
{
  #define BIOS_UNIT_STATUS_QUIESCENT 0
  UINT32 all;
  struct
    {
      unsigned int input_enabled : 1;
      unsigned int output_enabled : 1;
      unsigned int input_intr_enabled : 1;
      unsigned int output_intr_enabled : 1;
    } serial;
  struct
    {
      unsigned int input_enabled : 1;
      unsigned int output_enabled : 1;
      unsigned int input_intr_enabled : 1;
      unsigned int output_intr_enabled : 1;
    } network;
  struct
    {
      unsigned int unit_enabled : 1;
      unsigned int unit_intr_enabled : 1;
    } timer;
};
typedef union bios_unit_status bios_unit_status_t;
typedef union bios_unit_status* bios_unit_status_pt;

struct bios_unit_functions;
typedef struct bios_unit_functions bios_unit_functions_t;
typedef struct bios_unit_functions* bios_unit_functions_pt;

struct bios_unit
{
  bios_unit_functions_pt funcP;   /* unit function table pointer */
  void* moduleP;                   /* subsuming module */
  bios_unit_class_t class;         /* unit class (i.e., serial) */
  INT16 max_requests;              /* max simultaneously active requests */
};

```

```
typedef struct bios_unit bios_unit_t;
typedef struct bios_unit* bios_unit_pt;

struct bios_unit_functions
{
  bios_ret_t (*abort)(/* in     */ bios_unit_pt unitP,
                      /* in     */ bios_req_blk_pt req_blkP);
  bios_ret_t (*poll)(/* in     */ bios_unit_pt unitP,
                     /* in     */ bios_req_blk_pt req_blkP);
  bios_ret_t (*poll_any)(/* in     */ bios_unit_pt unitP,
                         /* out    */ bios_req_blk_pt* req_blkPP);
  bios_ret_t (*status)(/* in     */ bios_unit_pt unitP,
                       /* in/out */ bios_req_blk_pt req_blkP,
                       /* out    */ bios_req_blk_pt req_blkPP[],
                       /* out    */ bios_unit_status_pt statusP);
  bios_ret_t (*set_state)(/* in     */ bios_unit_pt unitP,
                          /* in     */ bios_unit_status_t state,
                          /* in/out */ bios_req_blk_pt req_blkP);
  bios_ret_t (*reset)(/* in     */ bios_unit_pt unitP,
                      /* in/out */ bios_req_blk_pt req_blkP);
};

#endif /* !BIOS_UNIT_H */
```