OBIOS: Open Basic Input Output System DRAFT Specification Version 1.14

January 24, 1991

Contents

Chapter 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 BIOS 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 BIOS.

The remainder of this document is laid out as follows. Chapter 2 describes the operational concepts and functionality of this BIOS 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 BIOS is viewed as a *passive server*. It is comprised of a collection of functions and data, the BIOS does not embody any active processing components such as tasks or interrupt service routines (ISRs). Callers of the BIOS are viewed as *clients*. The term client is used generically throughout this document to refer to any code that may invoke or facilitate BIOS functions. Typically there are two types of clients: one that configures, supports and initializes the BIOS modules and a second type which uses the BIOS to read, write and control individual I/O activities. The BIOS functions 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 *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 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 timeouts and initiation of any related action is 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 function.

- The individual BIOS calls are not reentrant. Serialization of BIOS calls is responsibility of the client.

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

- The source code of BIOS implementations are intended to be publicly available and freely redistributable.

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

1.2 Acknowledgements

Significant contributions towards making BIOS an industry-wide standard have been made by several people. We would particularly like to acknowledge the efforts of:

Greg Buzzard	Ready Systems
Dalibor Vrsalovic	Ready Systems
Glenn Kasten	Ready Systems
Shawn Morrissey	Force Computers
Michael McCrary	Force Computers
Wayne Fischer	Force Computers
Fred Rehhauser	Force Computers
Michael McCrary	Force Computers
Cousy Maher	Motorola
Chuck Corley	Motorola
Wayne Hampton	Software Components
David Wilner	Wind River Systems
David Bond	Wind River Systems
Jason Cotton	Wind River Systems
Andy Crane	Microware
Michael Burgher	Microware
Todd Wynia	Heurikon
Richard Peterson	Heurikon
Bruce Coorpender	88Open Consortium
Reed Cardoza	Eyring
Dave Steck	Eyring
Richard Stallman	Free Software Foundation (Project GNU)

Chapter 2

Operational Concepts

2.1 Overview

A BIOS 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), 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. All modules provide the same set of operations.

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. In a typical systems module calls would be used by the client that is responsible for configuring (or reconfiguring) and initializing BIOS 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 will 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 must 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 BIOS 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

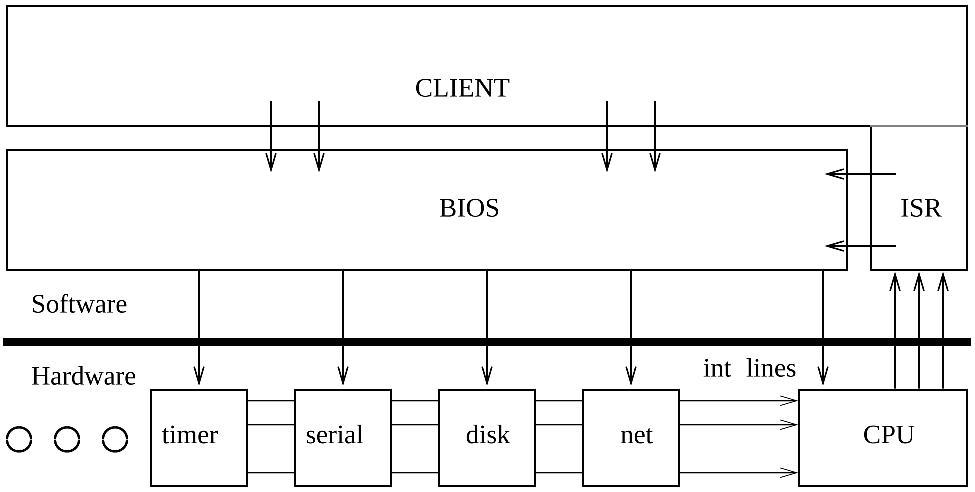

Figure 2.1: Module Interactions in a BIOS Environment.

`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 or continuous stage service request completes immediately. The BIOS 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 must 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 2.2). 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

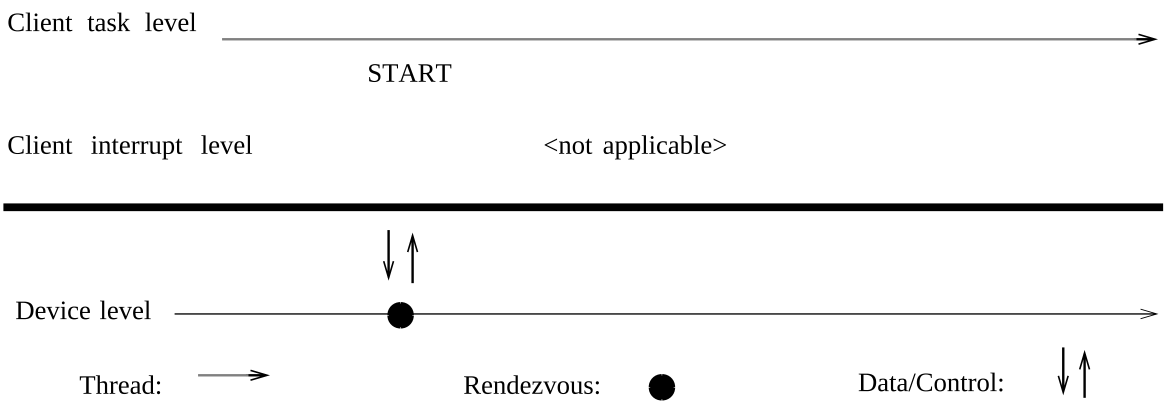

Figure 2.2: Single Stage Service Mode.

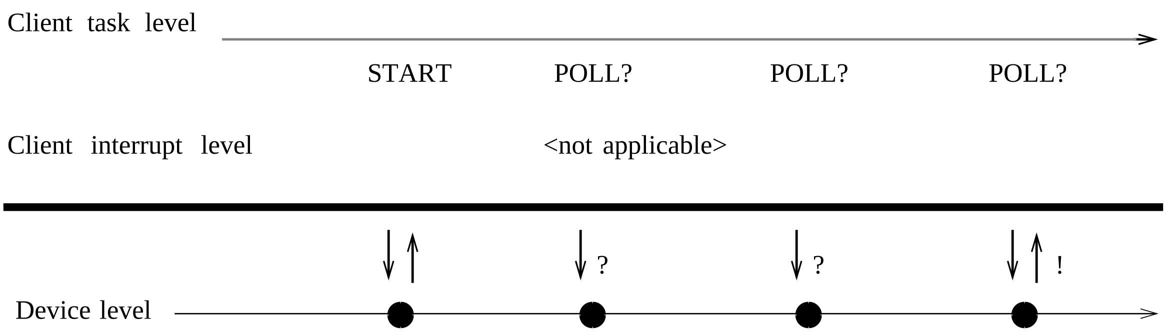

Figure 2.3: Multiple Stage, Polled Service Mode.

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`, `poll_any` or `poll` are required to eventually complete the request (typically from either an interrupt-level ISR, or from a task-level polling function – see Figures 2.3 and 2.4). 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*, another stage of the request must be processed in the future and a subsequent call to `intr_svc`, `poll_any` or `poll` must 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

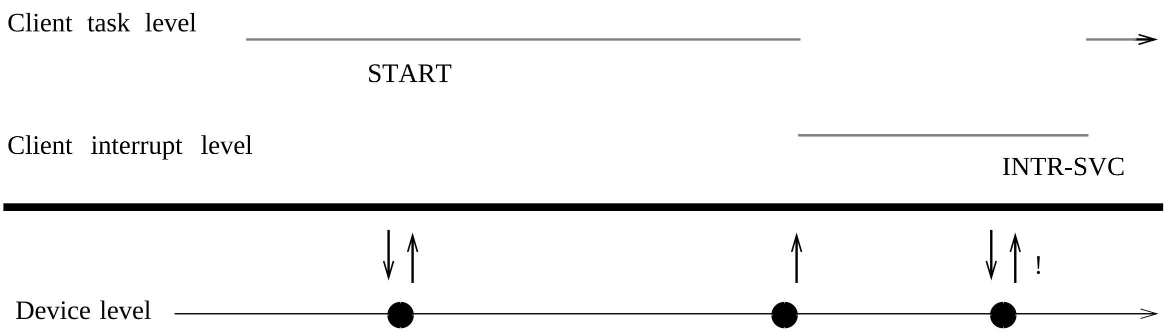

Figure 2.4: Multiple Stage, Interrupt Service Mode.

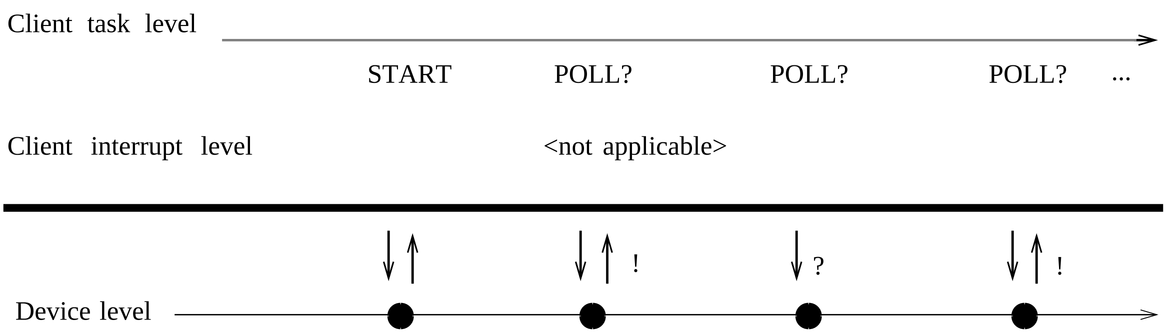

Figure 2.5: Continuous Stage, Polled Service Mode.

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 2.5 and 2.6). Unlike a multiple-stage request though, a continuous-stage request will never return a status of *ok* from `intr_svc`, `poll_any` or `poll`. A continuous stage request will always return a status that includes *ok_pending* (or, possibly, a fatal error) and is said to never reach normal (non-error) completion.

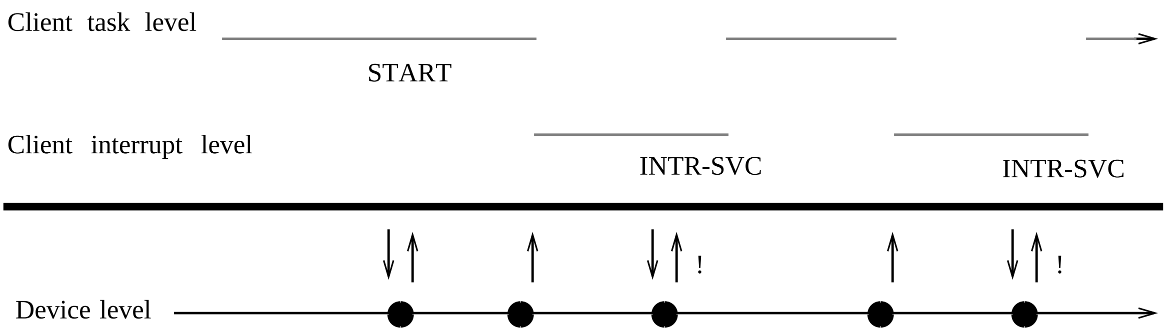

Figure 2.6: Continuous Stage, Interrupt Service Mode.

2.2 Client/BIOS 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 tinerrupt 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.) will define an additional set of pertinent commands which are specific for that class. All unit procedures that may require direct response from the hardware device must 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 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.

Module instance data. This data structure maintains the internal BIOS state of a module. Only a small initial portion of this structure is meant to be interpreted by the client.

Unit instance data. This data structure maintains the internal BIOS state of a functional unit. Only a small initial portion of this structure is meant to be interpreted by the client.

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 must 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. BIOS implementations are not required to monitor compliance.

2.2.1 Request Blocks

Request blocks (RBs) contain the following information:

1. Staging mode (in).
2. Timeout 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. The exact structure and valid values for the fields of the request block for each class of units is specified in Chapter 3.

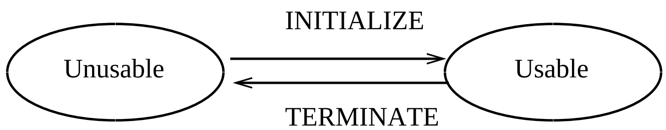

Figure 2.7: Device State Transitions.

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. This value must not be changed while the service transaction is in progress. 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 BIOS 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 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 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 BIOS may be able to make use of more memory than the amount requested. This may occur for modules that allocate internal BIOS I/O buffers. The BIOS will 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 `terminate`d.

The BIOS 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 2.7 and 2.8 show the state transitions that are allowed for modules and functional units. A module must be `initialize`d 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 a status 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 returned. Unit commands that check on

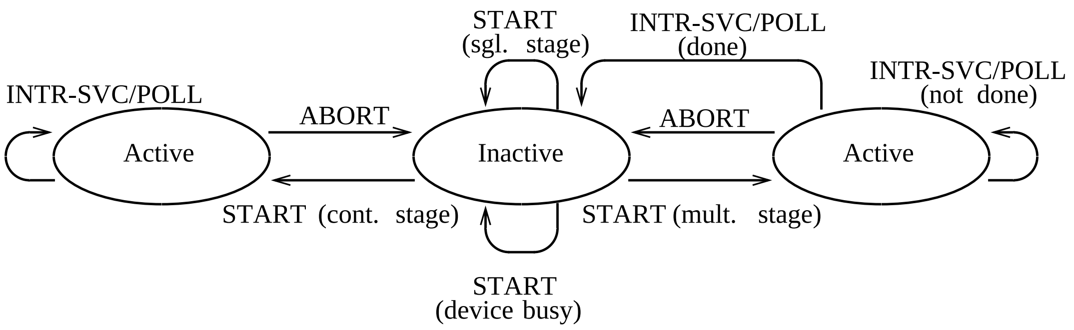

Figure 2.8: Request State Transitions.

service request status (`intr_svc`, `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 will 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 be completed 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 will 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` must 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* 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 timeout 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` function to formally acknowledge the interrupt. The `intr_svc` call is made to the function associated with the interrupting functional unit vector. Typically, 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 – this parameter is not included for all units. 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 *ok_pending*, a new timeout value has also been returned.

In summary, the return status from either the `intr_svc`, `poll` or `poll_any` functions 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 timeout period). *Device_busy* indicates that the current stage is not yet completed. *Device_failure* implies something fatal went wrong. *No_pending_request* implies that there is no request pending for the unit.

State Transition Error Checking

In deference to performance, the BIOS makes no attempt to prevent clients from invoking senseless sequences of commands. For example, a BIOS 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` must 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 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 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.

2.3 BIOS Data Structures

In addition to the request block structure (described in Section 2.2.1), 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 often 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 functions.

6. The memory requirements of the module.

The data in the above structure is often constant. However, for modules which can support an arbitrary 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 BIOS functions.

A functional unit instance data area is created for each unit contained in a module 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 functions. 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 functions.

4. 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 BIOS Interfaces

A BIOS 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 must be known by the client for each module in the system. The addresses of all module functions must 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 will 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 will share code, but operate on different unit instance data areas.

2.5 Summary

A BIOS 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 protection, where necessary, is made available by the client.

The procedural interface is comprised of:

`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 must 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 request commands – are used to initiate service transaction with a device unit. They provide the first (and sometimes only) rendezvous point between client and device unit. These command calls are made on functional units.

`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 BIOS functions that are provided to the client during module initialization.

There are five basic modes of client/BIOS 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.

Chapter 3

General Specification

This Chapter provides an explicit specification of the portion of the BIOS 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 BIOS 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 below.

```
/*****************************************************************
 *
 * 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, 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



/* useful type definitions */

/* BIOS return values */
typedef enum
{
  BIOS_RET_OK,
  BIOS_RET_OK_PENDING,
  BIOS_RET_INSUFFICIENT_MEM,
  BIOS_RET_DEV_BUSY,
  BIOS_RET_INV_SVC,
  BIOS_RET_UNSP_SVC,
  BIOS_RET_INV_MODE,
  BIOS_RET_INV_REQ_DATA,
  BIOS_RET_DEV_FAIL,
```

```
  BIOS_RET_INPUT_OVERRUN,
  BIOS_RET_INPUT_OVERRUN_HW,
  BIOS_RET_OK_PENDING_BUFFER_DONE,
  BIOS_RET_OK_PENDING_OUT_OF_DATA,
  BIOS_RET_NO_NEW_INPUT_DATA,
  BIOS_RET_IO_ERROR,
  BIOS_RET_SPURIOUS_INTR,
  BIOS_RET_DATA_SIZE,
  BIOS_RET_BAD_ADDRESS,
  BIOS_RET_UNDEFINED
} bios_ret_t;


typedef bios_ret_t (*bios_func_pt) ();

typedef int bios_boolean_t;

typedef UINT32 bios_addr_t;
typedef UINT32* bios_addr_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 */
```

The static module information is defined by the structure `bios_module_info` in the file `bios_module.h`, which is shown below.

```
/**********************************************************************
 *
 * 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* 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, bios_req_blk_pt* req_blkPP);
  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 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 */
```

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 BIOS 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 `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. The currently defined module identifiers are listed in Section 3.3.

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`:

```
/*******************************************************************
 *
 * 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:          Generic unit interface.
 *
 *******************************************************************/

#ifndef BIOS_UNIT_H
#define BIOS_UNIT_H

union bios_unit_status
{
  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;
};
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/out */ bios_req_blk_pt req_blkP,
                          /* in     */ bios_unit_status_t state);
  bios_ret_t (*reset)(/* in     */ bios_unit_pt unitP,
                      /* in/out */ bios_req_blk_pt req_blkP);
};

#endif /* !BIOS_UNIT_H */
```

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.

A BIOS 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 BIOS module and its related functions, thus they are not defined by the BIOS specification.

The structure of a request block consists of two parts, the first is a general part that is present in all request blocks, 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 two fields of type `void*`, one each for the client and BIOS. The use and interpretation of the client field is at the sole control of the client, the BIOS should not modify nor interpret its contents. Similarly, the use and interpretation of the BIOS field is at the sole control of the BIOS, the client should not modify nor interpret its contents.

3.2 Module Functions

There are five functions that must be provided by each BIOS module. The function prototypes for these functions as they appear in the BIOS module function pointer table are given in the file `bios_module.h`. A module must 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 BIOS 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.

3.2.1 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 returns a pointer to a data structure that describes the module. The structure is defined in the file `bios_module.h`, and should be treated as read-only by the client.

3.2.2 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 BIOS, its contents is 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 BIOS 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 BIOS vendor defined information. See Section G.1 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 BIOS 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 BIOS. The BIOS will typically allocate the `bios_unit` structures from the `moduleP` block of memory. Pointers to these structure 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 BIOS. 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 BIOS 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 must 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 BIOS 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.

3.2.3 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 BIOS. The `num_req_blks` parameter specifies a table of integers that will be filled in by the BIOS. 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.

3.2.4 Terminate

Synopsis

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

Description

The `moduleP` parameter provides the module instance data to the BIOS. Clients must 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.

3.2.5 Poll

Synopsis

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

Description

The `moduleP` parameter provides the module instance data to the BIOS. 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.

3.3 Defined Modules

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. The currently defined modules are:

- 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 shall follow the form of beginning with the full 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.

3.4 Unit Functions

The functions described in this section must be provided by each BIOS module. The function prototypes for these functions as they appear in the BIOS 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 BIOS drivers.

All unit functions that may interact with their corresponding hardware device must contain a request block parameter. This allows the BIOS interface to accommodate systems with high-latency I/O devices (see Section 3.5) for more details. The BIOS must invoke and release protection around all critical regions within the unit functions to protect them from simultaneous access from interrupt level. The BIOS is expected to invoke this protection, instead of the client, because it may do so across a much smaller span of code. It is erroneous to call `reset` while any transaction is active on the same unit, therefore, it will not contain any critical regions. The BIOS 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 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.

3.4.2 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 BIOS. The `req_blkP` parameter points to the request block that represents the activity that is being polled for. In this function, the BIOS will interact with the physical device and, possibly, update the state of the device and the BIOS 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 not previously been started. When called after a transaction has completed but prior to any reuse or other use of the request block, this function must return `BIOS_RET_OK`.

Return Values

These are highly dependent on the unit class, for further details see the corresponding sections for each unit class.

3.4.3 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 BIOS. 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 BIOS 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_DEVICE_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.

3.4.4 Status

The declaration of the BIOS 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 BIOS. The `req_blkP` parameter provides the request block to the BIOS. The `req_blkPP` parameter points to an array of request block pointers that the BIOS 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 BIOS 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 BIOS 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.

3.4.5 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 BIOS. The req_blkP parameter provides the request block to the BIOS. 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.

3.4.6 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 BIOS. The req_blkP parameter provides the request block to the BIOS. The client must 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.

3.4.7 Interrupt Service

For architectures with vectored interrupts a BIOS unit will typically provide one interrupt service function for each separate type of interrupt (i.e., vector) that a unit can generate. BIOS implementations for architectures with a single interrupt service routine per device will likely provide one interrupt service function per unit. The interface to this function is described in Section 3.2.2. The semantic actions are described separately for each unit class.

If a device unit does not support the interrupt mode of operation this function need not exist.

3.4.8 Interrupt and Polled Operations

The BIOS expects it clients to use the interrupt and polled modes of operations consistently. The client must 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 must 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 BIOS requires that clients not invoke an intr_svc function when polled mode service transactions are active on the same unit. As a result, BIOS implementations need only protect the interrupt mode code of unit functions from interrupt level access.

Several BIOS 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 BIOS 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.

3.5 High Latency I/O Support

All BIOS 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 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 BIOS accommodates high latency devices with the same interface as low latency devices, clients of the BIOS should not have to know the difference. This is accomplished by requiring all BIOS unit functions to include a request block parameter. Requests for service from the BIOS 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 3.2).

3.6 Reentrancy and Multiprocessor Support

The execution of all module and unit functions except `intr_svc` must 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. This level of protection is explicitly required by this specfication. Additionally, in order to achieve correct implementations, 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 threads or interrupt routines running on separate CPUs in a multiprocessor.

In a multiprocessor environment, one must rely on operating system and architectural support to achieve mutual exclusion. Furthermore, interrupt level functions (i.e., `intr_svc`) must be protected from reentrancy by task-level calls that may occur on other processors.

Much of this protection can be handled by the client. For instance, in a multiprocessor system the client ISR code can protect interrupt level BIOS code from reentrancy 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 BIOS function on a multiprocessor system or, in order to minimize the amount of time that interrupts are disabled on a uniprocessor system, the BIOS task level functions must be able to access operating system provided mutual exclusion primitives. To meet this need, BIOS implementations must surround task level critical regions with the macros `BIOS_CR_ENTER(lock)` and `BIOS_CR_EXIT(lock)`, where `lock` is an argument of type `void*`. Furthermore, BIOS implementations must use the macro `BIOS_CR_CREATE`, which is defined to return a value of type `void*`, to acquire the `lock` parameter value from the client system for each unit that requires critical region protection. `BIOS_CR_FREE` must be used to free the critical region lock when the module is terminated.

To explicitly summarize, BIOS implementations must:

1. Surround task level critical regions with the macros BIOS_CR_ENTER(lock) and BIOS_CR_EXIT(lock), where `lock` is an argument of type `void*`.

2. Use the macro `BIOS_CR_CREATE`, which is defined to return a value of type `void*`, to acquire the `lock` parameter value from the client system for each unit that requires critical region protection. Use the macro `BIOS_CR_FREE` to free the critical region lock when the module is terminated.

The required client support is specified in Appendix G. In many architectures, the required mutual exclusion support is provided directly by the hardware. In such cases, the BIOS vendor may supply the support that is specified in Appendix G.

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.

3.7 Notes on this Version of the Document

Changes from the v1.12 document are indicated with changebars.

1. The `get_info` function has been modified to be able to actually contact the device before returning memory requirements.

2. The `poll_any` function return values have been fixed to correct an error in the previous version.

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

Appendix A

Serial Unit Specification

The serial unit provides an abstraction for buffered serial I/O operations. The buffering abstraction is careful to provide a powerful, flexible and efficient interface that simultaneously avoids any high-level buffer management functions such as allocation or queuing. This unit supports the full range of service modes and makes use of the timeout output parameter – timeouts are specified in microseconds.

Support is provided for the detection of special events such as break reception, reception errors, and status changes in communication circuits. The detection of a break and any reception errors will be reflected in the status field of the request block structure. The BIOS will accumulate values into the status field. The client may read and clear this field as desired. The status function will check for the occurrence of a break and any reception errors. To accommodate the wide variety of serial communications circuit standards (e.g., RS-232-C, RS-449, and so forth), the BIOS provides general functions to get, set, and notice changes in the various DTE circuits. The generality is provided by specifying the DTE connector contact number. In deference to performance considerations, support for getting, setting and noting changes in the communications circuirts is not required. There is a function which will return information to indicate the circuits are supported by the driver.

A.1 Data Structures

The notion of a buffer that is defined by the BIOS is that of a contiguous linear array of bytes that is described by a maximum extent and an active region within the extent. The extent is defined by two pointers *bob* (beginning of buffer) and *eob* (end of buffer). The active region is defined by a head (H – the current point of insertion into the buffer) and a tail (T – the current point of removal from the buffer).

For receive operations the BIOS writes incoming data into the buffer area between H and *eob*, moving H as the data is written and taking a service specific action as *eob* is approached (see Figure A.1). The client has the option of processing blocks of data by waiting for the buffer full indication (see Figure A.2), replacing the full buffer with a new empty buffer and proceeding to process the full buffer while characters are arriving in the new buffer. Alternatively, the client may process a single character at a time by continually removing characters from the buffer (see Figure A.3) until the buffer full indication, then wrapping the H around back to the *bob* of the same buffer – in this case it is the responsibility of the BIOS to ensure that H does not overtake T. With the exception of wrapping H around back to the *bob* when the buffer becomes full, the client should treat H as read only and T as read/write. The BIOS will ignore T and treat H as read/write.

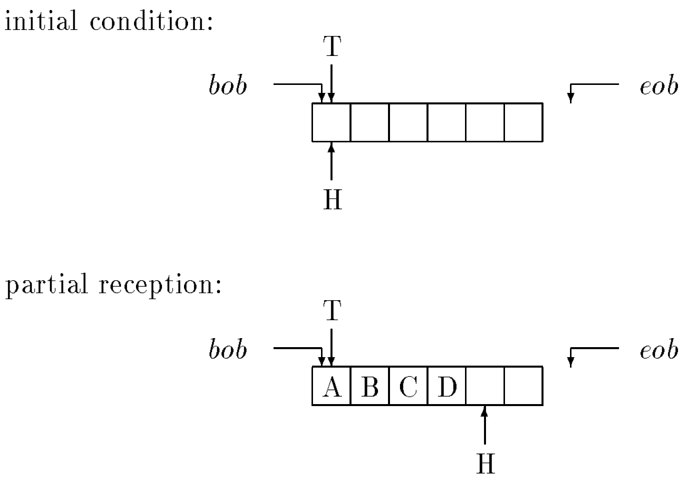

Figure A.1: Serial Read, Start Up.

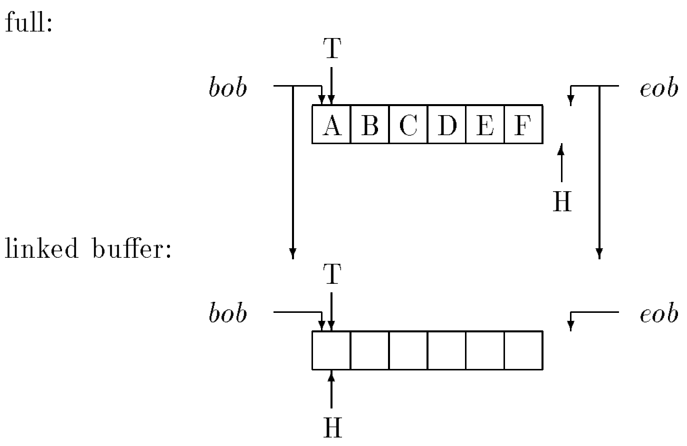

Figure A.2: Serial Read, Linked Linear Buffers.

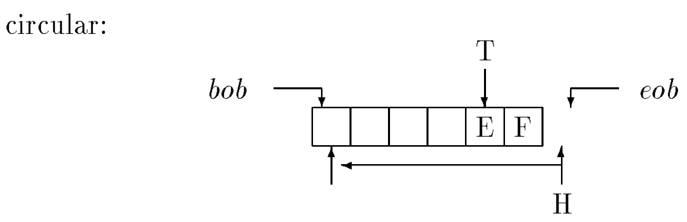

Figure A.3: Serial Read, Single Circular Buffer.

For transmit operations the BIOS reads and outputs data from the buffer area ahead of T, moving T as the data is output and taking a service specific action as either H or *eob* are approached. This is the converse of the recieve operation. The client may send blocks of data by providing an initially full buffer, collecting another block of output data while waiting for the buffer empty indication, then replacing the empty buffer with the new block of output data. Alternatively, the client may send a character at a time by filling the output buffer, then wrapping the H around back to the *bob* – in this case it is the responsibility of the BIOS to ensure that H does not overtake T. It should be noted that a continuous mode transmit operation (in either interrupt or polled mode) that stalls because T runs into H, may be restarted after adding characters to the buffer (and moving H) by calling the `poll` function. The client should treat T as read only and H as read/write. The BIOS will treat H as read only and T as read/write.

The data structure, return type and function prototypes that comprise the serial unit interface are specified in the file `bios_serial.h` as follows.

```
/*****************************************************************
 *
 * 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:  Generic serial device header file
 *
 * History:      07/25/90          King Chou
 *
 *****************************************************************/

#ifndef BIOS_SERIAL_H
#define BIOS_SERIAL_H

/* Enumerate the values for the parameters */

typedef enum
{
  BIOS_SERIAL_ONE,
  BIOS_SERIAL_ONE_DOT_FIVE,
  BIOS_SERIAL_TWO,
```

```
} bios_serial_stop_t;

typedef enum
{
  BIOS_SERIAL_PARITY_NONE,
  BIOS_SERIAL_PARITY_EVEN,
  BIOS_SERIAL_PARITY_ODD,
} bios_serial_parity_t;

/* Modem control inputs and special input conditions.  These values */
/* are ORed into the return value. */
typedef enum
{
  BIOS_SERIAL_OVE = 0x00000001,         /* receiver overrun */
  BIOS_SERIAL_FRE = 0x00000002,         /* framing error */
  BIOS_SERIAL_PE  = 0x00000004,         /* parity error */
  BIOS_SERIAL_BRK = 0x00000008,         /* break received */
} bios_serial_special_t;

typedef struct bios_serial_dir bios_serial_dir_t;
typedef struct bios_serial_dir* bios_serial_dir_pt;

/* read and write service request block structure */
struct bios_serial_rw_block
{
  bios_serial_special_t spec; /* special event status info */
  char *tailP;                  /* for read: Client moves this (in). */
                                /* for write: BIOS moves this (in/out). */
  char *headP;                  /* for read: BIOS moves this (in/out). */
                                /* for write: Client moves this (in). */
  char *bobP;                   /* Beginning of block (in). */
  char *eobP;                   /* End of block (in). */
};

typedef struct bios_serial_rw_block bios_serial_rw_block_t;
typedef struct bios_serial_rw_block* bios_serial_rw_block_pt;

struct bios_serial_ccts
{
  UINT32  in_lo;     /* pins 1-32 (1 = LSB) */
  UINT32  in_hi;     /* pins 33-64, of above */
  UINT32  out_lo;    /* pins 1-32 (1 = LSB) */
  UINT32  out_hi;    /* pins 33-64, of above */
  UINT32  intr_lo;   /* pins 1-32 (1 = LSB) */
  UINT32  intr_hi;   /* pins 33-64, of above */
};

typedef struct bios_serial_ccts bios_serial_ccts_t;
typedef struct bios_serial_ccts* bios_serial_ccts_pt;


/* serial device Request Block formats */

struct bios_serial_req_blk
{
  bios_req_blk_t gen;
  union
    {
      bios_serial_rw_block_t rw_block;   /* read/write buffers */
```

```
     bios_serial_ccts_t     ccts;       /* supported data comm */
 /* circuits */
    } u0;
};

typedef struct bios_serial_req_blk bios_serial_req_blk_t;
typedef struct bios_serial_req_blk* bios_serial_req_blk_pt;


struct bios_serial_unit_functions
{
  bios_unit_functions_t gen;
  bios_ret_t (*set_baud)(/* in     */ bios_unit_pt unitP,
                         /* in     */ UINT32 in_baud,
                         /* in     */ UINT32 out_baud,
                         /* in/out */ bios_serial_req_blk_pt req_blkP);
  bios_ret_t (*set_bits)(/* in     */ bios_unit_pt unitP,
                         /* in     */ bios_serial_parity_t parity,
                         /* in     */ bios_serial_stop_t stop,
                         /* in     */ UINT32 bits,
                         /* in/out */ bios_serial_req_blk_pt req_blkP);
  bios_ret_t (*brk)(/* in     */ bios_unit_pt unitP,
                    /* in     */ bios_boolean_t val,
                    /* in/out */ bios_serial_req_blk_pt req_blkP);
  bios_ret_t (*get_cct_usage)(/* in     */ bios_unit_pt unitP,
                              /* in/out */ bios_serial_req_blk_pt req_blkP);
  bios_ret_t (*get_ccts)(/* in     */ bios_unit_pt unitP,
                         /* in/out */ bios_serial_req_blk_pt req_blkP);
  bios_ret_t (*set_ccts)(/* in     */ bios_unit_pt unitP,
                         /* in/out */ bios_serial_req_blk_pt req_blkP);
  bios_ret_t (*intr_ccts)(/* in     */ bios_unit_pt unitP,
                          /* in/out */ bios_serial_req_blk_pt req_blkP);
  bios_ret_t (*loop)(/* in     */ bios_unit_pt unitP,
                     /* in     */ bios_boolean_t val,
                     /* in/out */ bios_serial_req_blk_pt req_blkP);
  bios_ret_t (*write)(bios_unit_pt unitP,
                      bios_serial_req_blk_pt req_blkP);
  bios_ret_t (*read)(bios_unit_pt unitP,
                     bios_serial_req_blk_pt req_blkP);
};

typedef struct bios_serial_unit_functions bios_serial_unit_functions_t;
typedef struct bios_serial_unit_functions* bios_serial_unit_functions_pt;

#endif /* !BIOS_SERIAL_H */
```

A.2 Functions

The must BIOS invoke and release protection around all critical regions within the unit functions to protect them from simultaneous access from interrupt level. The BIOS is expected to invoke this protection, instead of the client, because it may do so across a much smaller span of code. It is erroneous to call the `reset`, `set_bits`, `set_baud` or `set_loopback` while any transaction is active on the same unit, therefore, these functions will not contain any critical regions.

The following functions are described by the general description given in Section 3.4.

```
bios_ret_t
bios_mname_serial_abort(bios_unit_pt unitP,
                        bios_serial_req_blk_pt req_blkP)

bios_ret_t
bios_mname_serial_poll_any(bios_unit_pt unitP,
                           bios_req_blk_pt* req_blkPP)

bios_ret_t
bios_mname_serial_status(bios_unit_pt unitP,
                         bios_serial_req_blk_pt req_blkP,
                         bios_serial_req_blk_pt req_blkPP[],
                         bios_unit_status_pt statusP)

bios_ret_t
bios_mname_serial_set_state(bios_unit_pt unitP,
                            bios_unit_status_t state,
                            bios_serial_req_blk_pt req_blkP)

bios_ret_t
bios_mname_serial_reset(bios_unit_pt unitP,
                        bios_serial_req_blk_pt req_blkP)
```

All of the serial unit functions that initiate an I/O data transfer or control operation specify a `req_blkP` parameter. Unless otherwise specified, this parameter is used only to serve as a transaction identifier to accommodate the possibility of needed to check for later completion if the hardware device provides high-latency responses. When used in this manner, none of the fields in the request block are interpreted.

A.2.1 Set Baud

Synopsis

```
bios_ret_t
bios_mname_serial_set_baud(/* in     */ bios_unit_pt unitP,
                           /* in     */ UINT32 in_baud,
                           /* in     */ UINT32 out_baud,
                           /* in/out */ bios_serial_req_blk_pt req_blkP)
```

Description

Set the baud rate for both the transmitter and receiver to `in_baud` and `out_baud`, respectively. A zero value for either of the baud rate parameters indicates that no change is desired for the corresponding parameter. A set baud call may fail for a variety of reasons including the inability of the hardware to support different baud rates for transmit and receive – providing a zero value for exactly one of the baud rates may cause a failure. Note, the term *baud* is used for historical purposes and is not technically correct, for the purposes of this document it is defined baud to mean bits per second. The `req_blkP` parameter is used

Return Values

BIOS_RET_DEV_FAIL	– baud rate request cannot be satisfied.
BIOS_RET_OK_PENDING	– not yet complete, further rendezvous required.
BIOS_RET_OK	– successful completion.

A.2.2 Set Bits

Synopsis

```
bios_ret_t
bios_mname_serial_set_bits(/* in     */ bios_unit_pt unitP,
                           /* in     */ bios_serial_parity_t parity,
                           /* in     */ bios_serial_stop_t stop,
                           /* in     */ UINT32 bits,
                           /* in/out */ bios_serial_req_blk_pt req_blkP)
```

Description

Set the parity, stop bits, and bits per character to the indicated values.

Return Values

BIOS_RET_INV_REQ_DATA	– bad input value.
BIOS_RET_OK_PENDING	– not yet complete, further rendezvous required.
BIOS_RET_OK	– successful completion.

A.2.3 Send Break

Synopsis

```
bios_ret_t
bios_mname_serial_brk(/* in     */ bios_unit_pt unitP,
                      /* in     */ bios_boolean_t val,
                      /* in/out */ bios_serial_req_blk_pt req_blkP)
```

Description

Send (or end) a break sequence.

Return Values

BIOS_RET_OK_PENDING	– not yet complete, further rendezvous required.
BIOS_RET_OK	– successful completion.

A.2.4 Get Circuit Usage

Synopsis

```
bios_ret_t
bios_mname_serial_get_cct_usage(/* in     */ bios_unit_pt unitP,
                                /* in/out */ bios_serial_req_blk_pt req_blkP);
```

Description

The req_blkP parameter contains a ccts structure. This function fills in the fields of this structure as follows.

in_lo	circuit pins 1-32 (1 = LSB), bit is asserted if BIOS implementation can get value of this pin.
in_hi	circuit pins 33-64, same as above.
out_lo	pins 1-32 (1 = LSB), bit is asserted if BIOS implementation can set value of this pin.
out_hi	circuit pins 33-64, same as above.
intr_lo	pins 1-32 (1 = LSB), bit is asserted if BIOS implementation can cause interrupt on change in value of this pin.
intr_hi	circuit pins 33-64, same as above.

Return Values

BIOS_RET_OK_PENDING	– not yet complete, further rendezvous required.
BIOS_RET_OK	– successful completion.

A.2.5 Get Asserted Circuits

Synopsis

```
bios_ret_t
bios_mname_serial_get_ccts(/* in     */ bios_unit_pt unitP,
                           /* in/out */ bios_serial_req_blk_pt req_blkP);
```

Description

The req_blkP parameter contains a ccts structure. This function fills in the in_lo and in_hi fields of this structure as follows.

in_lo	circuit pins 1-32 (1 = LSB), bit is asserted if BIOS implementation can get value of this pin and the circuit is asserted.
in_hi	circuit pins 33-64, same as above.

The other ccts structure fields are ignored.

Return Values

BIOS_RET_OK_PENDING	– not yet complete, further rendezvous required.
BIOS_RET_OK	– successful completion.

A.2.6 Set (Assert) Circuits

Synopsis

```
bios_ret_t
bios_mname_serial_set_ccts(/* in     */ bios_unit_pt unitP,
                           /* in/out */ bios_serial_req_blk_pt req_blkP);
```

Description

The req_blkP parameter contains a ccts structure. This function asserts supported circuits that are indicated by the out_lo and out_hi fields of this structure as follows.

out_lo	circuit pins 1-32 (1 = LSB), if bit is asserted and the BIOS implementation can assert this pin, the circuit is asserted.
out_hi	circuit pins 33-64, same as above.

The other ccts structure fields are ignored.

Return Values

BIOS_RET_INV_REQ_DATA	– no action taken, requested circuits cannot be asserted.
BIOS_RET_OK_PENDING	– not yet complete, further rendezvous required.
BIOS_RET_OK	– successful completion.

A.2.7 Interrupt on Circuit State Change

Synopsis

```
bios_ret_t
bios_mname_serial_intr_ccts(/* in     */ bios_unit_pt unitP,
                            /* in/out */ bios_serial_req_blk_pt req_blkP);
```

Description

The req_blkP parameter contains a ccts structure. This function causes an interrupt to be raised when the input circuits that correspond to the bits that are set (by the client) in the intr_lo and intr_hi fields this structure as follows.

intr_lo	circuit pins 1-32 (1 = LSB), if bit is asserted and the BIOS implementation can raise an interrupt on a change in this pin, it will do so.
intr_hi	circuit pins 33-64, same as above.

Upon a change in value, an interrupt is raised. In the BIOS interrupt routine a bit is set in the in_lo and in_hi field of the structure above for all input circuits which asserted (as long as they are also *interrupt capable* by the implementation). The client will need to perform an exclusive-or between the present in_lo and in_hi fields and their previous values to determine which circuits changed value. The other structure fields (out_lo and out_hi) are ignored.

When invoked in multiple stage mode, the BIOS will complete the transaction after the first *watched* circuit changes status. When invoked in continuous stage mode, the BIOS will continue to raise interrupts whenever a *watched* circuit changes status (client must use abort to end a continuous mode transaction). The circuits to be *watched* cannot be changed during the life of an intr_ccts transaction.

Return Values

BIOS_RET_INV_REQ_DATA	– no action taken, interrupt cannot be raised for change on requested circuits.
BIOS_RET_OK_PENDING	– not yet complete, further rendezvous required.
BIOS_RET_OK	– successful completion.

A.2.8 Set Loopback Mode

Synopsis

```
bios_ret_t
bios_mname_serial_loop(/* in     */ bios_unit_pt unitP,
                       /* in     */ bios_boolean_t val,
                       /* in/out */ bios_serial_req_blk_pt req_blkP)
```

Description

Set (or reset) hardware device loopback mode, based on the `val` parameter.

Return Values

BIOS_RET_INV_SVC	– requested service is not supported.
BIOS_RET_OK_PENDING	– not yet complete, further rendezvous required.
BIOS_RET_OK	– successful completion.

A.2.9 Write

Synopsis

```
bios_ret_t
bios_mname_serial_write(bios_unit_pt unitP,
                        bios_serial_req_blk_pt req_blkP)
```

Description

Initiate a write operation. The service mode and buffer pointers should be provided in the request block structure, the timeout value will be returned in this structure. If data may be transferred to the device by this function, it is. In this function and the subsequent polling and interrupt handling functions, the BIOS will write only the head buffer pointer and will only read the tail, beginning of buffer and end of buffer pointers. The client is normally expected to write the tail pointer and read the head pointer. However, the client may manipulate any of the pointers to implement the buffer strategy of its choice – it need only watch out for the side effect of the head buffer pointer moving as a result of a write action completed interrupt. Transmit interrupts will be enabled or disabled as specified by the service mode. This function supports all service modes. Multiple stage requests terminate when the head and tail pointers are equal. Continuous stage requests must be explicitly terminated via the abort function.

Return Values

BIOS_RET_DEV_BUSY	– no action taken, write unit is already active.
BIOS_RET_INV_MODE	– no action taken, non-existent mode was requested.
BIOS_RET_OK_PENDING	– request active, end of buffer has not been reached.
BIOS_RET_OK_PENDING_BUFFER_DONE	– request active, end of buffer has been reached.
BIOS_RET_OK	– successful completion.

A.2.10 Read

Synopsis

```
bios_ret_t
bios_mname_serial_read(bios_unit_pt unitP,
                       bios_serial_req_blk_pt req_blkP)
```

Description

Initiate a read operation. The service mode and buffer pointers should be provided in the request block structure, the timeout value will be returned in this structure. The interrupting capability is set appropriately based on the specified mode, the timeout value is set, and the request block is attached to the unit instance data structure. In the subsequent polling and interrupt handling functions, the BIOS will write only the tail buffer pointer and will only read the head, beginning of buffer and end of buffer pointers. The client is normally expected to write the head pointer and read the tail pointer. However, the client may manipulate any of the pointers to implement the buffer strategy of its choice – it need only watch out for the side effect of the tail buffer pointer moving as a result of a write action completed interrupt. This function supports all service modes. Multiple stage requests terminate when the head and tail pointers are equal. Continuous stage requests must be explicitly terminated via the abort function.

Return Values

BIOS_RET_DEV_BUSY	– no action taken, read unit is already active.
BIOS_RET_NO_NEW_INPUT_DATA	– no action taken, no new data has arrived.
BIOS_RET_INV_MODE	– no action taken, non-existent mode was requested.
BIOS_RET_OK_PENDING	– request active, end of buffer has not been reached.
BIOS_RET_OK	– successful completion.

A.2.11 Poll

Synopsis

```
bios_ret_t
bios_mname_serial_poll(bios_unit_pt unitP,
                       bios_serial_req_blk_pt req_blkP)
```

Description

Perform a poll on a previously initiated operation.

The the operation is a write request the service mode and buffer pointers should be provided in the request block structure, the timeout value in the request block structure is preserved. If data may be transferred to the device by this function, it is. The BIOS will write only the head buffer pointer and will only read the tail, beginning of buffer and end of buffer pointers. The client is normally expected to write the tail pointer and read the head pointer. However, the client may manipulate any of the pointers to implement the buffer strategy of its choice. Multiple stage requests terminate when the head and tail pointers are equal. Continuous stage requests must be explicitly terminated via the abort function.

Return Values

BIOS_RET_INV_REQ_DATA	– no action taken, invalid request block pointer was provided (the inclusion of this check is optional).
BIOS_RET_OK_PENDING	– request active, data sent, end of buffer has not been reached, head not equal to tail.
BIOS_RET_DEV_BUSY	– no action taken, no additional characters have been transmitted since last poll.
BIOS_RET_OK_PENDING_BUFFER_DONE	– request active, end of buffer has been reached.
BIOS_RET_OK_PENDING_OUT_OF_DATA	– request active, end of buffer not reached, but head equals tail, continuous mode only.
BIOS_RET_OK	– successful completion.

If the operation is a read request the service mode and buffer pointers should be provided in the request block structure, the timeout value in the request block structure is preserved. If data may be transferred from the device by this function, it is. The BIOS will write only the tail buffer pointer and will only read the head, beginning of buffer and end of buffer pointers. The client is normally expected to write the head pointer and read the tail pointer. However, the client may manipulate any of the pointers to implement the buffer strategy of its choice. Multiple stage requests terminate when the head and end of buffer pointers are equal. Continuous stage requests must be explicitly terminated via the abort function.

Return Values

BIOS_RET_INV_REQ_DATA	– no action taken, invalid request block pointer was provided (the inclusion of this check is optional).
BIOS_RET_NO_NEW_INPUT_DATA	– no action taken, no new data has arrived.
BIOS_RET_INPUT_OVERRUN	– no action taken, data has arrived, but we are at the end of the buffer.
BIOS_RET_OK_PENDING	– request active, data has been received, end of buffer has not been reached.
BIOS_RET_DEV_BUSY	– no action taken, no additional characters have been received since last poll.
BIOS_RET_OK_PENDING_BUFFER_DONE	– request active, data has been received, and end of buffer has been reached, continuous mode only.
BIOS_RET_OK	– successful completion.

If the operation is other than a read or write request, the service must be multiple stage (continuous stage service is supported for only the read and write operations) and the return values are set the same as they would be if the transaction completed in a single stage with the followoing exception. In the case that the transaction in not yet complete and a further rendezvous is required, a value of *device_busy* is returned instead of *ok_pending*.

A.2.12 Interrupt Handlers

Pointers to BIOS 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, the declaration of the functions is given in the module specific header file.

Synopsis

```
struct bios_intr_svc
{
  bios_ret_t (*intr_func)(void* magic_cookie, bios_req_blk_pt* req_blkPP);
  void* magic_cookie;
};
```

Description

The BIOS 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* and a pointer to a request block pointer will be passed as parameters. The content of the magic cookie is left to the discretion of the BIOS implementor, it will typically be a pointer to the unit instance data. A pointer to the request block associated with the interrupt or a null value will be returned in the second parameter. The semantic activities and return values are identical to those of the polling functions with the addition of return values to indicate spurious interrupts and errors.

Return Values

same as poll functions plus...

`BIOS_RET_IO_ERROR`	– I/O error.
`BIOS_RET_SPURIOUS_INTR`	– spurious interrupt.

A.3 Notes on this Version of this Appendix

Changes from the v1.12 document are indicated with changebars.

1. The read and write poll calls have been recombined. And the specification of the return values now includes the use of *device_busy* to indicate no activity since last poll.

2. The request block parameter has been added to all calls (to accommodate high-latency devices).

3. The status return value kludge has been fixed per Glenn's comments of 6 December.

Sample code, based on the version 1.12 document and header files, is available. The modifications to support the changes in this version of the document should be pretty straightforward. The sample code implements a driver for the Z8530 serial communications controller. The sample code does not include the *special condition* checking or support for setting modem lines, loopback, and sending breaks, but should be complete in its basic functionality. This code should basically work, I've done some testing (reading and writing character strings), but there are zillions of combinations of things that I have not tried. The code has been used, without modification, on both mvme147 and mvme133 boards.

Appendix B

Network Unit Specification

The network unit provides an abstract interface for high-speed multiple access peer-to-peer communication devices that is suitable as a basis on which to build a logical data-link level service such as IEEE 802.2. The network unit interface expects the client to have filled in all of the data link frame fields. This implies that the network device has no knowledge of logical protocols such as ARP (address resolution protocol) or trailer encapsulation. The network interface is designed to be applicable to a variety of data-link formats (e.g., ethernet, token ring, etc.). The setting of many of the data-link specific parameters (e.g., toke hold times, token rotation times, and so forth) can be accommodated via the configuration block (`configP` parameter) that is provided to the module initialize function. This unit supports the multiple stage service modes and makes use of the timeout output parameter – timeouts are specified in microseconds.

Network read and write functions handle, at most, one data link frame at a time. Network write transactions are defined to complete as soon as transmission of the data link frame has finished. Network read transactions are defined to complete as soon as a received data link frame is successfully returned to the client.

There are two common schemes for handling pools of transmit and receive buffers: (1) the BIOS always uses buffers that are explicitly supplied by the client (via read and write calls) or (2) the BIOS maintains its own pool of buffers and copies data to/from the client during read/write transactions.

Network read transactions differ from read transactions for many other units because the arrival of network data is often not correlated to any action on the receiving machine. The network data arrival times, amount of data, and other characteristics of the arriving message are typically unknown. Thus, in the former case, network read transactions have a dual mission: (1) to provide the network unit with buffers for subsequently arriving data and (2) to provide a means by which to return to the client a, typically different, set of buffers that correspond to an arrived data link frame.

In the latter case, the buffers provided with a network read request remain associated with that request. If the BIOS currently has a fully arrived data link frame, it is copied to the buffers supplied with the read call and the call completes. If not, the read call does not complete at this time. At a future rendezvous (`intr_svc` or `poll_any`) a fully arrived data link frame will be copied to the buffers supplied with the pending read call and the transaction will then complete.

Some data-link interfaces define a message priority. Support for message priorities is optional and is provided via a request block field that is interpreted by the read and write functions. The range of acceptable priority values is defined by the specification of the data-link that is being used. For modules that do not support message priorities, the priority field in the request block is ignored.

B.1 Data Structures

The data structure that is used to represent the data buffers that are passed between the BIOS and client, along with the return type and function prototypes, are shown in the code listing below. This structure, the network buffer list, consists of at least one entry. This first required entry specifies the maximum length of the list (in unit of numbers of entries), the actual length of the list (note the minimum length is one since the first entry is always required), and a pointer field that is used when it is neccessary for the BIOS to communicate to the client that it has processed a portion of the supplied buffer list. The fields of the *actual* entries indicate (1) the maximum length of their corresponding buffer, (2) the actual length of their corresponding buffer, and (3) a pointer to the buffer. Any remaining entries are not interpreted, i.e., their values do not matter.

The syntactic interface to the network unit is specified in the file `bios_network.h` as follows:

```
/**********************************************************************
 *
 * 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:  Generic network device header file
 *
 * History:
 *   21-Nov-90  Greg Buzzard (buz@ready.com) Ready Systems.
 *     Created.
 *
 **********************************************************************/

#ifndef BIOS_NETWORK_H
#define BIOS_NETWORK_H

/*
 * Network buffer lists are constructed from list elements
 * (bios_network_list_el_t) as follows:
 *
```

```
 *                  <len> <cnt> <bufP>
 *                 +--------------------+
 *                 |  5  |  3  | null   |
 *                 |--------------------|                 +----------------+
 *                 |  20 |  20 | bufP --+----------------->| 20 byte buffer |
 *                 |--------------------|                 | 20 bytes used  |
 *                 |  30 |  8  | bufP --+----+            +----------------+
 *                 |--------------------|    |   +--------------------+
 *                 |  -  |  -  |  -     |    +-->|  30 byte buffer    |
 *                 |--------------------|        |   8 bytes used     |
 *                 |  -  |  -  |  -     |        +--------------------+
 *                 +--------------------+
 *
 * Where: the first entry indicates the maximum length of the list (5
 * elements), the actual length of the list (3 elements -- note the
 * minimum length is one since the first entry is always required),
 * and a null pointer.  The remaining actual entries indicate the
 * maximum length of their corresponding buffer, the actual length of
 * their corresponding buffer, and a pointer to the buffer.  Any
 * remaining entries are not interpreted (i.e., the values do not
 * matter).
 */

struct bios_network_list_el
{
  INT32         len;            /* maximum number of elements in buffer */
  INT32         cnt;            /* actual number of elements in buffer */
  void*         bufP;           /* pointer to buffer */
};

typedef struct bios_network_list_el bios_network_list_el_t;
typedef struct bios_network_list_el* bios_network_list_el_pt;

/* network device Request Block formats */

struct bios_network_req_blk
{
  bios_req_blk_t                gen;
  int                           prio; /* message priority, data-link defined */
  bios_network_list_el_pt       lP;   /* message data frame buffers */
};

typedef struct bios_network_req_blk bios_network_req_blk_t;
typedef struct bios_network_req_blk* bios_network_req_blk_pt;

struct bios_network_errors
{
  union
    {
      UINT32 crc;
      UINT32 alignment;
      UINT32 resource;
      UINT32 overrun;
      UINT32 recv_collision;
      UINT32 short_frame;
    } enet;
};

typedef struct bios_network_errors bios_network_errors_t;
```

```
typedef struct bios_network_errors* bios_network_errors_pt;

struct bios_network_info
{
  bios_network_data_link_t
    data_link; /* type of data link */
  UINT32 min_tu_size; /* min transmission unit (in bytes) */
  UINT32 max_tu_size; /* max transmission unit (in bytes) */
  bios_boolean_t hw_bcast; /* does HW support broadcast */
  bios_boolean_t xmit_echo; /* does HW receive its own transmits */
};

typedef struct bios_network_info bios_network_info_t;
typedef struct bios_network_info* bios_network_info_pt;


struct bios_network_unit_functions
{
  bios_unit_functions_t gen;
  bios_ret_t (*get_address)(/* in     */ bios_unit_pt unitP,
                            /* in     */ void* addrP,
                            /* in/out */ bios_network_req_blk_pt req_blkP);
  bios_ret_t (*add_address)(/* in     */ bios_unit_pt unitP,
                            /* in     */ void* addrP,
                            /* in/out */ bios_network_req_blk_pt req_blkP);
  bios_ret_t (*bcast_enable)(/* in     */ bios_unit_pt unitP,
                             /* in     */ bios_boolean_t value,
     /* in/out */ bios_network_req_blk_pt req_blkP);
  bios_ret_t (*errors)(/* in     */ bios_unit_pt unitP,
               /* in/out */ bios_network_req_blk_pt req_blkP,
                       /* out    */ bios_network_errors_pt errorsP);
  bios_ret_t (*info)(/* in     */ bios_unit_pt unitP,
                     /* in/out */ bios_network_req_blk_pt req_blkP,
                     /* out    */ bios_network_info_pt);
  bios_ret_t (*set_promiscuous)(/* in     */ bios_unit_pt unitP,
                                /* in     */ bios_boolean_t value,
                                /* in/out */ bios_network_req_blk_pt req_blkP);
  bios_ret_t (*set_loop)(/* in     */ bios_unit_pt unitP,
                         /* in     */ bios_boolean_t value,
                         /* in/out */ bios_network_req_blk_pt req_blkP);
  bios_ret_t (*write)(/* in     */ bios_unit_pt unitP,
                      /* in/out */ bios_network_req_blk_pt req_blkP);
  bios_ret_t (*read)(/* in     */ bios_unit_pt unitP,
                     /* in/out */ bios_network_req_blk_pt req_blkP);
};

typedef struct bios_network_unit_functions bios_network_unit_functions_t;
typedef struct bios_network_unit_functions* bios_network_unit_functions_pt;

#endif /* !BIOS_NETWORK_H */
```

B.2 Functions

The must BIOS invoke and release protection around all critical regions within the unit functions to protect them from simultaneous access from interrupt level. The BIOS is expected to invoke this protection, instead of the client, because it may do so across a much smaller span of code. It is erroneous to call the `reset`, `set_address`, `set_promiscuous` or `set_loopback` while any transaction is active on the same unit, therefore, these functions will not contain any critical regions.

The following functions are described by the general description given in Section 3.4.

```
bios_ret_t
bios_mname_network_abort(bios_unit_pt unitP,
                         bios_serial_req_blk_pt req_blkP)

bios_ret_t
bios_mname_poll_any(bios_unit_pt unitP,
                    bios_req_blk_pt* req_blkPP)

bios_ret_t
bios_mname_network_status(bios_unit_pt unitP,
                          bios_network_req_blk_pt req_blkP,
                          bios_network_req_blk_pt req_blkPP[],
                          bios_unit_status_pt statusP)

bios_ret_t
bios_mname_network_set_state(bios_unit_pt unitP,
                             bios_unit_status_t state,
                             bios_network_req_blk_pt req_blkP)

bios_ret_t
bios_mname_network_reset(bios_unit_pt unitP,
                         bios_network_req_blk_pt req_blkP)
```

All of the network unit functions that initiate an I/O data transfer or control operation specify a `req_blkP` parameter. Unless otherwise specified, this parameter is used only to serve as a transaction identifier to accommodate the possibility of needed to check for later completion if the hardware device provides high-latency responses. When used in this manner, none of the fields in the request block are interpreted.

B.2.1 Get Address

Synopsis

```
bios_ret_t
bios_mname_get_address(/* in     */ bios_unit_pt unitP,
                       /* in     */ void* addrP,
                       /* in/out */ bios_network_req_blk_pt req_blkP);
```

Description

Get address will always return the local hardware address (either the module hardware knows this address or it was provided by the module initialization call).

Return Values

BIOS_RET_IO_ERROR	– requested service is not successful.
BIOS_RET_OK_PENDING	– request is still pending.
BIOS_RET_OK	– successful completion.

B.2.2 Add Address

Synopsis

```
bios_ret_t
bios_mname_add_address(/* in     */ bios_unit_pt unitP,
                       /* in     */ void* addrP,
                       /* in/out */ bios_network_req_blk_pt req_blkP);
```

Description

Add an address to list of network addresses for which to receive data link frames. After the capacity for additional receive addresses is reached, subsequent calls will fail.

Return Values

BIOS_RET_IO_ERROR	– requested service is not successful.
BIOS_RET_OK_PENDING	– request is still pending.
BIOS_RET_OK	– successful completion.

B.2.3 Broadcast Enable

Synopsis

```
bios_ret_t
bios_mname_bcast_enable(/* in     */ bios_unit_pt unitP,
                        /* in     */ bios_boolean_t value,
                        /* in/out */ bios_network_req_blk_pt req_blkP);
```

Description

Set (or reset) the ability to receive network frames with a broadcast destination address.

Return Values

BIOS_RET_INV_SVC	– requested service is not supported.
BIOS_RET_DEV_FAIL	– requested service is not successful.
BIOS_RET_OK_PENDING	– request is still pending.
BIOS_RET_OK	– successful completion.

B.2.4 Errors

Synopsis

```
bios_ret_t
bios_mname_errors(/* in     */ bios_unit_pt unitP,
  /* in/out */ bios_network_req_blk_pt req_blkP,
```

```
                  /* out    */ bios_network_errors_pt errorsP);
```

Description

The accumulated error counts are returned by this function in the `errorsP` parameter. The error counts are set to zero on a unit reset and at module initialization time. The errors that are counted are defined by the media access type (e.g., IEEE 802.3, IEEE 802.4, etc.). The media access type is provided by the network information function.

Return Values

BIOS_RET_DEV_FAIL	– requested service is not successful.
BIOS_RET_OK_PENDING	– request is still pending.
BIOS_RET_OK	– successful completion.

B.2.5 Network Information

Synopsis

```
bios_ret_t
bios_mname_info(/* in     */ bios_unit_pt unitP,
                /* in/out */ bios_network_req_blk_pt req_blkP,
                /* out    */ bios_network_info_pt);
```

Description

This function returns static information about the following network media access characteristics: the type (e.g., IEEE 802.3, FDDI, etc.), the minimum transmission unit size, the maximum transmission unit size, whether the hardware provides a broadcast capability, and whether the hardware receives its own transmissions. This information is returned in the infoP parameter.

Return Values

BIOS_RET_DEV_FAIL	– requested service is not successful.
BIOS_RET_OK_PENDING	– request is still pending.
BIOS_RET_OK	– successful completion.

B.2.6 Set Promiscuous

Synopsis

```
bios_ret_t
bios_mname_set_promiscuous(/* in     */ bios_unit_pt unitP,
                           /* in     */ bios_boolean_t value,
                           /* in/out */ bios_network_req_blk_pt req_blkP)
```

Description

Set (or reset) the ability to receive all network frames, regardless of destination address. Implementations are not required to receive frames that they send.

Return Values

BIOS_RET_INV_SVC	– requested service is not supported.
BIOS_RET_DEV_FAIL	– requested service is not successful.
BIOS_RET_OK_PENDING	– request is still pending.
BIOS_RET_OK	– successful completion.

B.2.7 Set Loopback

Synopsis

```
bios_ret_t
bios_mname_network_loop(/* in     */ bios_unit_pt unitP,
                        /* in     */ bios_boolean_t value,
                        /* in/out */ bios_network_req_blk_pt req_blkP)
```

Description

Set (or reset) hardware device loopback mode, based on the `val` parameter.

Return Values

BIOS_RET_INV_SVC	– requested service is not supported.
BIOS_RET_DEV_FAIL	– requested service is not successful.
BIOS_RET_OK_PENDING	– request is still pending.
BIOS_RET_OK	– successful completion.

B.2.8 Write

Synopsis

```
bios_ret_t
bios_mname_network_write(/* in     */ bios_unit_pt unitP,
                         /* in/out */ bios_network_req_blk_pt req_blkP)
```

Description

Initiate a write operation. The service mode parameter, a pointer to an allocated buffer list and, optionally, a message priority should be provided in the request block structure, a timeout value will be returned in this structure. This function moves or copies a data link frame from the request block buffer list to a device controlled output queue. The `bufP` field of the first buffer list element is set to `NULL` by the BIOS if the data has been copied to a transmit buffer that is internal to the BIOS – this indicates to the client that the supplied buffers are free to be reused upon return from this call (prior to transaction completion).

Return Values

BIOS_RET_DEV_BUSY	– device is presently unable to accommodate the requested service (usually because the output buffer is full).
BIOS_RET_DEV_FAIL	– requested service is not successful.
BIOS_RET_OK_PENDING	– request is still pending.
BIOS_RET_OK	– successful completion.

B.2.9 Read

Synopsis

```
bios_ret_t
bios_mname_network_read(/* in     */ bios_unit_pt unitP,
                        /* in/out */ bios_network_req_blk_pt req_blkP)
```

Description

Initiate a read operation. The service mode, a pointer to an allocated buffer list and, optionally, the minimum message priority that is desired should be provided in the request block structure, the timeout value and, if applicable, actual message priority will be returned in this structure. Consecutive network read requests should complete in the order in which they are initiated or, if applicable, in an order that is modified by the priority value of the available messages. The action of the read function depends on whether the BIOS has preallocated its own pool of buffer.

If the BIOS is using client provided buffers the actions are: (1) move as many (empty) buffers as the implementation can accept from the provided buffer list (which may be empty) to a device controlled list, (2) updates the `bufP` field of the (head) first buffer list entry to point to the first location past the last accepted buffer, (3) if all provided buffers have not been accepted, return with `BIOS_RET_BUFFER_EXCESS`, if more buffers are needed, return with `BIOS_RET_BUFFER_DROUGHT`, otherwise (4) move any set of buffers that contain one completely received data link frame to the provided buffer list.

If the BIOS is using its own buffers the action is to copy any set of buffers that contain one completely received data link frame to the provided buffer list. In contrast to the preceding case, the buffers supplied by the provided buffer list stay associated with their request block for the duration of the transaction.

In either of the above cases, if a complete data link frame of an appropriate priority is not immediately available, the read call will return with a *pending* status indicating that a further client/BIOS rendezvous will be necessary (i.e., this is the first stage of a multiple stage transaction).

Return Values

BIOS_RET_INV_SVC	– requested service is not supported.
BIOS_RET_DEV_FAIL	– requested service is not successful.
BIOS_RET_OK_PENDING	– request is successfully initiated.
BIOS_RET_OK	– successful completion.

B.2.10 Poll

Synopsis

```
bios_ret_t
bios_mname_network_poll(/* in     */ bios_unit_pt unitP,
                        /* in/out */ bios_network_req_blk_pt req_blkP)
```

Description

Poll on the status of a specific active transaction. See also the poll any function that is described in Section 3.4.

Return Values

BIOS_RET_INV_REQ_DATA	– request block not found.
BIOS_RET_DEV_BUSY	– request is still pending (no action has been taken).
BIOS_RET_OK	– successful completion.

B.2.11 Interrupt Handlers

Pointers to BIOS 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, the declaration of the functions is given in the module specific header file.

Synopsis

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

Description

The BIOS 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* must be passed as a parameter. The content of the magic cookie is left to the discretion of the BIOS implementor, it will typically be a pointer to the unit instance data.

The processing at the completion of a network transaction may take a significant amount of time. In order to minimize the amount of time spent at interrupt level, the network interrupt function does a minimal amount of work and expects the client to make a subsequent call to one of the polling functions to complete the transaction. For spurious interrupts, a subsequent call to a polling function need not be made.

Return Values

BIOS_RET_IO_ERROR	– I/O error.
BIOS_RET_SPURIOUS_INTR	– spurious interrupt.
BIOS_RET_OK__PENDING	– network data transfer completed, make subsequent polling function call to complete transaction.

B.3 Notes on this Version of this Appendix

1. Read and write poll have been coalesced.
2. Message priorities have been added to request block.
3. Network info type has been added, needs review.

Appendix C

Clock Unit Specification

The clock unit provides an abstraction for a non-volatile clock, which maintains the current time-of-day and calendar, even while the client is not running. The clock is maintained in units of seconds and days relative to Coordinated Universal Time. Support is provided for reading and writing the clock.

C.1 Data Structures

The primary data structure is `struct tm`, which is excerpted from `<time.h>` of ANSI C. See ANSI C for the full description of `struct tm`.

```
struct tm {
  int tm_sec;   /* seconds after the minute (0..61) */
  int tm_min;   /* minutes after the hour (0..59) */
  int tm_hour;  /* hours since midnight (0..23) */
  int tm_mday;  /* day of month (1..31) */
  int tm_mon;   /* month since January (0..11) */
  int tm_year;  /* years since 1900 (>=0) */
  int tm_wday;  /* day since Sunday (0..6) */
  int tm_yday;  /* day since January 1 (0..365) */
  int tm_isdst; /* daylight savings time flag */
};
```

The data structure, return type and function prototypes that comprise the clock unit interface are specified in the file `bios_clock.h` as follows.

bios_clock.h

```
/****************************************************************************
*
* Module Name:  bios_clock.h
*
* Identification:  @(#) 3.3 bios_clock.h
*
* Date:  1/24/91  16:44:11
*
* Description:  Generic clock device header file
*
```

```
****************************************************************************
*/

#ifndef BIOS_CLOCK_H
#define BIOS_CLOCK_H

#include <time.h>

/* serial device Request Block formats */

struct bios_clock_req_blk
{
  bios_req_blk_t gen;
  struct tm      tm;
  long           cut_offset;
};

typedef struct bios_clock_req_blk bios_clock_req_blk_t;
typedef struct bios_clock_req_blk* bios_clock_req_blk_pt;


struct bios_clock_unit_functions
{
  bios_unit_functions_t gen;
  bios_ret_t (*read)(/* in     */ bios_unit_pt unitP,
     /* in/out */ bios_clock_req_blk_pt req_blkP);

  bios_ret_t (*write)(/* in     */ bios_unit_pt unitP,
      /* in/out */ bios_clock_req_blk_pt req_blkP);
};

typedef struct bios_clock_unit_functions bios_clock_unit_functions_t;
typedef struct bios_clock_unit_functions* bios_clock_unit_functions_pt;

#endif /* !BIOS_CLOCK_H */
```

C.2 Functions

The following functions are described by the general description given in Section 3.4.

```
bios_ret_t
bios_mname_clock_abort(bios_unit_pt unitP,
                       bios_clock_req_blk_pt req_blkP)

bios_ret_t
bios_mname_clock_poll_any(bios_unit_pt unitP,
                          bios_req_blk_pt* req_blkPP)

bios_ret_t
bios_mname_clock_status(bios_unit_pt unitP,
                        bios_clock_req_blk_pt req_blkP,
                        bios_clock_req_blk_pt req_blkPP[],
                        bios_unit_status_pt statusP)
```

```
bios_ret_t
bios_mname_clock_set_state(bios_unit_pt unitP,
                           bios_unit_status_t state,
                           bios_clock_req_blk_pt req_blkP)

bios_ret_t
bios_mname_clock_reset(bios_unit_pt unitP,
                       bios_clock_req_blk_pt req_blkP)
```

C.2.1 Read

Synopsis

```
bios_ret_t
bios_mname_clock_read(/* in     */ bios_unit_pt unitP,
                      /* in/out */ bios_clock_req_blk_pt req_blkP)
```

Description

The client specifies a pointer to a request block, which receives the current time (field `tm`) and the offset from Coordinated Universal Time in seconds (field `cut_offset`). The offset from Coordinated Universal Time will be set to -1 if unknown.

Return Values

`BIOS_RET_IO_ERROR`	– requested service is not successful.
`BIOS_RET_OK_PENDING`	– not yet complete, further rendezvous required.
`BIOS_RET_OK`	– successful completion.

C.2.2 Write

Synopsis

```
bios_ret_t
bios_mname_clock_write(/* in     */ bios_unit_pt unitP,
                       /* in/out */ bios_clock_req_blk_pt req_blkP)
```

Description

The client specifies a pointer to a request block, containing the new time (field `tm`) and offset from Coordinated Universal Time in seconds (field `cut_offset`). The offset from Coordinated Universal Time should be -1 if unknown.

Return Values

`BIOS_RET_IO_ERROR`	– requested service is not successful.
`BIOS_RET_OK_PENDING`	– not yet complete, further rendezvous required.
`BIOS_RET_INV_SVC`	– requested service is not supported.
`BIOS_RET_OK`	– successful completion.

C.2.3 Poll

Synopsis

```
bios_ret_t
bios_mname_clock_poll(/* in     */ bios_unit_pt unitP,
                      /* in/out */ bios_clock_req_blk_pt req_blkP)
```

Description

Poll on the status of a specific active transaction. See also the poll any function that is described in Section 3.4.

Return Values

BIOS_RET_INV_REQ_DATA	– request block not found.
BIOS_RET_OK_PENDING	– request is still pending.
BIOS_RET_OK	– successful completion.

C.2.4 Interrupt Handlers

Pointers to BIOS 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, the declaration of the functions is given in the module specific header file.

Synopsis

```
struct bios_intr_svc
{
  bios_ret_t (*intr_func)(void* magic_cookie,
                          bios_clock_req_blk_pt* req_blkPP);
  void* magic_cookie;
};
```

Description

The BIOS 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* must be passed as a parameter. The content of the magic cookie is left to the discretion of the BIOS implementor, it will typically be a pointer to the unit instance data.

Return Values

BIOS_RET_IO_ERROR	– I/O error.
BIOS_RET_SPURIOUS_INTR	– spurious interrupt.
BIOS_RET_OK	– clock request successfully completed.

C.3 Usage Notes

Some implementations of the clock device may take a *very* long time to read or write the clock (e.g. on the order of seconds), or may disable interrupts for a long time while accessing the device. Thus it is recommended that the client only consult the clock during initialization, and use an alernative software clock (based on an interrupting BIOS timer device) to keep the time thereafter.

In some implementations (e.g. those based on a high-precision standard accessed via radio), the clock is read-only to the client. In this case, an attempt to write to the clock will return an error code.

The `tm_sec` field of `struct tm` has range 0 to 61, to allow for up to 2 leap seconds. Implementations based on a battery-backed-up crystal oscillator will probably be incapable of returning leap seconds, however those based on a radio may very well do so. For example, a leap second was inserted on December 31, 1990. Therefore, the client software should be written to deal with this possibility.

C.4 Rationale

The time is returned in a `struct tm` rather than as seconds relative to an *epoch*, in order to be more operating system independent. This approach also supports leap seconds, which an approach based on an *epoch* would not support.

Some UNIX implementations have added the field `tm_gmtoff` to `struct tm`, which holds the offset from Coordinated Universal Time in seconds. However, this field is not specified by ANSI C, thus relying on it would be non-portable. Therefore the offset from Coordinated Universal Time is a separate field in the request block.

C.5 Open Issues

How can the client determine whether Daylight Savings Time is in effect? Should the `tm_isdst` field of `struct tm` be used for this purpose? How can the client determine the time zone? Should the (non-portable) `tm_zone` field of `struct tm` be used for this purpose?

Appendix D

Timer Counter Unit Specification

Not yet complete.

Appendix E

Disk Unit Specification

Not yet complete.

The Open Basic Input Output System standard provides abstract interfaces for abstract devices to handle common I/O requirements. One of the major objectives of OBIOS is to increase the portability for client software across a heterogeneous set of processors and I/O devices. One such class of I/O devices is the generic disk unit.

E.1 Position Statement for A Generic Disk Unit

The Disk Unit abstraction will be provided through the SCSI Unit as an implementation of the well-defined SCSI commands for disks. BIOS modules will be required to provide the identical interface required for the SCSI *pass-through* unit and must also provide the control software and data structures necessary to emulate SCSI disk functionality.

This abstraction is illustrated below:

```
          SCSI Unit Class Device                  Disk Unit Class Device

      The client access generic SCSI devices      The client accesses disk
      through the SCSI unit abstraction           devices through the SCSI unit
                                                  abstraction using SCSI commands


          +----------+                                     +----------+
          |  Client  |                                     | Client   |
          +----------+                                     +----------+
               |                                                |
-------------+------------------------------------------------+---------------
               |                                                |
               V                                                V
          +----------+                                     +----------+
          |SCSI UNIT |                                     |DISK UNIT |
          |  BIOS    |                                     | BIOS     |
          +----------+                                     +----------+
               |                                                |
-------------+------------------------------------------------+---------------
               |                                                |
               V                                                V
```

```
    +------------+                                     +----------+
    |generic SCSI|                                     |   DISK   |
    |  Device    |                                     | Device   |
    +------------+                                     +----------+


The SCSI unit BIOS implements the            The Disk unit BIOS not only
"command pass through" functionality.        accepts client SCSI commands
                                             but also emulates the execution
                                             of these commands as if the
                                             Disk Device were a SCSI disk.
```

In cases where the SCSI device is a disk, the Disk Unit BIOS can be viewed as being provided by the *intelligent disk*. In cases where the Disk device is not *intelligent* (e.g. ST-506, SA412, ESDI, SMD) the Disk Unit BIOS provides emulation functionality in the form of code and associated data structures to provide the SCSI *intelligence*. In either case, the client accesses a disk device in the same manner.

E.2 Justification

The concept of a generic disk device interface for the OBIOS was discussed at length. Several discussions (summarized below) justify the OBIOS position relating to disk devices.

The choice of SCSI – The Small Computer Systems Interface (SCSI) standard has addressed disk interfaces very well. This massive standardization effort has been evolving for the last 6-8 years and investments from peripheral manufacturers as well as computer manufacturers have resulted into a well-accepted mass storage standard. The functionality specified by the SCSI standard covers all client application disk interface requirements.

Emulation of a SCSI disk – The disk interface functionality provided by the BIOS module implementer is tightly coupled to the requirements of the SCSI standard for SCSI Random Access Devices. As with a SCSI disk controller implementer, the BIOS SCSI emulation implementer must meet the minimum requirements specified by the SCSI specification, and is free to implement any optional extensions of the SCSI specification as well as any *vendor unique* extensions of the specification.

The OBIOS SCSI Device offers an abstraction for interfacing to any class of SCSI device (Random Access, Sequential Access, Read-Only, Write Only, etc.) through a *SCSI command pass through* interface. For an OBIOS client, the OBIOS Disk Device abstraction consists of a class of these SCSI commands. For the Disk Unit BIOS module implementer, there are two levels of implementation: (1) if the disk device is a SCSI disk, then the implementation only requires communication with the SCSI bus; (2) if the disk device is not SCSI, then the implementation not only requires the client interface as an OBIOS SCSI device but also the emulation of the SCSI commands for the disk. This emulation will at minimum require physical block to SCSI block conversions and optionally require defect management and other *intelligent disk* functionality. The implementation is left open to the implementer as long as it meets the minimum SCSI requirements as specified in (DISCUSSION NEEEDED HERE) revision of the SCSI specification.

Code, memory, and performance issues – In the infant days of SCSI, SCSI controllers were developed to provide *SCSI intelligence* to available disk interfaces (ST-506, SA412, SMD, etc.). The technology today allows the integration of SCSI functionality directly on the disk drives. This technology trend influenced the choice for SCSI disk emulation for the OBIOS disk device.

Concerns regarding the extra complexity required to emulate the SCSI interface for a non-SCSI disk came up during the committee's discussions. Such concerns usually were reconciled with the statements that the work done in the emulation module (SCSI logical to physical block conversion, for example) would most likely have to be performed anyway in a wide majority of client applications; the trade-off is whether the work is done by the client or by the BIOS module. Memory requirements for code and data structures are about the same whether the work is done by the client or by the BIOS module. Performance would be approximately the same with a slight overhead in providing some of the SCSI command interpretations for non-SCSI disks but this overhead would exist in any non-specific generic disk interface.

E.3 Summary

A direct disk device interface could be more elegantly and concisely implemented through other means than through SCSI emulation. But providing a consistent standard for disk interfaces across a set of heterogeneous disk controllers probably would lead to a complex implementation approaching that which is already defined in SCSI.

Why Re-invent The Wheel? The SCSI standardization has already put forth the effort; OBIOS will build upon that work.

The model of the OBIOS Disk Device is summarized below:

```
                                 +--------------------+
                                 |     DISK DEVICE    |
                                 |       Client       |
                                 +--------------------+
                                          |
                                          |
                                          V
                                 +--------------------+
                                 | SCSI Pass-through  |
                                 |      Client        |
                                 |   Code and Data    |
                                 |     Structures     |
                                 +--------------------+
                                          |
------------------------------------------+------------------------------------
                                          |
                                          V
                                 +--------------------+
                                 | SCSI Pass-through  |
                                 |       BIOS         |
```

```
                                +---------+----------+
                                     |    |SCSI disk |
                                     |    |Emulation |
                                     |    +----------+
                                     |          |
-------------------------------------+----   ---+---------------------------
                                     |          |
                                     V          V
                        +------------+         +----------------+
                        | SCSI  DISK |         |  Non-SCSI DISK |
                        | Device     |         |  Device        |
                        +------------+         +----------------+
```

Appendix F

SCSI Unit Specification

Not yet complete.

Appendix G

Client Configuration and Support Requirements

While this specification attempts to achieve total independence from the client code and architecture of the system on which it will execute, this goal was compromised in a few limited areas in deference to performance and ease of use. Independence from the application and I/O subsystem code that initiates and controls I/O operations is still maintained. This Appendix details the support that is required or expected from the client code that is responsible configuring the BIOS into the overall system.

G.1 Configuration and Initialization

It behooves a client to know the addresses of the modules that are present in the system. The BIOS may not be able to detect if the expected physical module exists at the address specified in a `get_module_info` or `initialize` call. The capability of positively identifying a device is not supported on all systems. Furthermore, the BIOS will not detect if a specified address is valid.

The client must know the location of the `get_module_info` function for each module it plans to call. Once this location is known, it may dynamically determine the addresses of the remaining module and unit functions.

G.1.1 Configuration Block Data Structure

[Not yet complete.] BIOS vendors may add vendor specific information to the basic configuration block structure that is specified with each module.

G.1.2 ROM Based Implementations

[Not yet complete.]

Need to indentify list of module configuration block pointers and corresponding module `get_module_info` function pointers for each module instance in supported by the ROM.

G.2 Reentrancy and Multiprocessor Support

[Need to review.] The client code which links with the BIOS must:

1. Ensure that execution of all module and unit functions within the same unit except `intr_svc` are mutually exclusive and non-reentrant.

2. Ensure that interrupt level calls to the BIOS are protected from reentrancy by task-level calls that may occur on other processors in a multiprocessor system.

3. In some special cases the client may need to provide BIOS compile-time definitions for the macros listed in Section 3.6. These definitions must implement the necessary mutual exclusion operations. For uniprocessors, the entry and exit of critical regions may correspond to enabling and disabling interrupts at the CPU, device or interrupt controller levels and, hence, may be provided by the BIOS implementation. For multiprocessors, the entry and exit of critical regions may correspond to enabling and disabling of interrupts at the device or interrupt controller levels and, again, may be provided by the BIOS implementation. If this is not possible or desirable, the client will have to provide the support described above.

4. Ensure that calls to `intr_svc` be made only from interrupt level and that calls to all other BIOS functions be made only from task level.

G.3 Time Delay Support

[Not yet complete.] Need to make available a microsecond spin delay function calibrated for the target system.

G.4 Data Size and Alignment

[Not yet complete.] Need to indicate stack requirement and any alignment requirements for module `initialize`.

G.5 Address Translations

[Not yet complete.] For systems with MMUs or with address mappings that result in I/O device DMA units using different addresses than the CPU, support for performing the CPU to other address translations will need to be provided by the client. The manner in which this support will be provided is not yet specified.