OBIOS the Open Basic Input Output System

Greg Buzzard

December 12, 1990

1 OBIOS: What and Why?

The Open Basic Input Output System is a uniform and consistent abstraction that comprises a collection of raw device drivers that conform to a well specified interface standard. There are two distinct components to this system, the program code that comprises the collection of raw device drivers and the companion standard that precisely specifies the interface that the program code must provide. The program code provides, and the standard describes, a common abstract interface and client/device interaction model that will span a wide range of devices, systems and usage requirements. Throughout this article the Open Basic Input Output System will be referred to as the OBIOS.

The OBIOS is viewed as a *passive server*. Callers of the OBIOS are viewed as clients. The OBIOS standard specifies the interface between the client and the OBIOS in a very careful manner. It must strike an optimal balance between specifying enough details so that the client is provided with a powerful abstraction, but it must also specify sufficiently little so that vendors of a OBIOS may exercise their talents to provide sensible and efficient implementations that work for a variety of devices across a wide range of boards.

Before discussing the OBIOS in more detail, let's digress for a moment to clear up one of the myths about the strategic implications of standards. While there is generally widespread agreement in the user community that standards are a good idea, this viewpoint is not always shared by producers of products. In particular, there is a large consensus that the growth of the importance of standards in the market place is one of the primary causes of the recent decline in profitability of much of the computer industry. However, as pointed out by Marc Schulman in an address delivered at the recent (1990) Uniforum conference, the fallacy of this myth can be revealed by exploring the differences between standards and standardization (an example of the latter is the production of standardized, or *cloned*, products).

Briefly, users would like to use software tools and components from a variety of vendors on a wide range of different host and target combinations. It is rather easy to satisfy this desire for *interoperability* by rapidly generating clones of popular software for a variety of hardware systems. But this is damaging in two ways. First, this practice produces tremendous profit pressures. It removes most elements of product differentiation and added value. This tends to move the product towards a commodity status that rewards the lowest cost producers – often marginally, at best. The decline in profit margins then often produces the second damaging effect which is the stifling or slowing of innovation and further product development.

The goal of providing users with interoperability can be achieved through another means, the adoption of sensible industry standards. A well chosen and comprehensive set of standards can meet all of the user requirements for interoperability, while at the same time providing wide latitude for different vendors to distinguish their products with special features, or performance or reliability characteristics, that appeal to a specific market niche. There are two key issues that must be evaluated in any set of standards. The first, is that each standard must specify an interface at a level that is appropriate for providing interoperability amongst software tools and components. The second, is that the standards themselves must strike that careful balance in design that allows for a powerful abstract interface while permitting a range of implementation options. The OBIOS standard has been carefully designed to provide both the appropriate level of interface and the proper amount of specification.

2 Goals

The OBIOS is a source level standard, along with its related implementation it has eight major goals.

Operating system independence and portability. The OBIOS has been very carefully designed as a *passive server* that does not rely on any operating system support in order to provide its functionality. This fact becomes apparent upon inspection of the client/device interaction model and call interface. The OBIOS design is also very careful to provide abstract device interfaces that allow the OBIOS to hide (or *abstract*) the specific device that lies beneath the generalized interface. As an example, there is a single client interface that describes serial devices. The OBIOS implementation provides this same interface regardless of whether the actual device is a MC68901, Z8530, SCN68562 or other. This allows client code, such as an operating system or application to be more easily ported to different target systems.

Concise interface and consistent user model. The OBIOS provides a concise, object oriented call interface to all devices. This interface, along with the semantic model of client and device interactions, is common across all devices. Of course some devices, due to their simple functionality, may only provide a subset of the interaction model and call interface.

Good performance characteristics. While blazing performance is not the only concern, the OBIOS specification is careful to allow for very efficient modes of operation for performance critical device interactions. Also, the OBIOS code may reside in either ROM or RAM. In many systems RAM accesses are significantly faster than ROM accesses.

Allowing field upgrades with ready to use PROMs. The OBIOS is designed to be used in a manner that allows OBIOS PROMs to be replaced without having to modify or relink any other system or application code. The specific method by which this is done is not specificed by the OBIOS. Rather, the OBIOS provides the mechanisms which allow a client to easily configure devices and link to their controlling functions at run time.

User configurability. The OBIOS allows users to dynamically dynamically replace existing device driver routines.

Easy to port. OBIOS implementations may be designed in an easy to construct manner. In particular, conforming device drivers for specific hardware devices may be collected into a library. As new devices come into existence, a conforming driver may be written and archived into the library. The builder of the OBIOS then *plugs* an appropriate selection of library routines into their I/O system. All device dependencies are isolated within the driver library routines. This *easy to construct* organization of OBIOS modules is not dictated by the OBIOS specification, rather it is a feature of our sample implementation. All OBIOS implementations are available in source form.

Support for multiprocessor systems. The OBIOS directly supports both multiprocessor and uniprocessor systems.

Support for high-latency devices. The OBIOS supports both local devices that respond immediately to simple requests and also intelligent (possibly remote) devices that queue multiple requests and may respond with high latency.

An overview of the general structure of OBIOS and its client interaction model is given in the following section. Specific details are provided in the draft specification.

3 Overview

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

Modules, in turn, are comprised of functional units. There may be one or more functional units per module. If there are multiple functional units within a module, they may be from the same or differing functional unit classes. It is the functional units that provide the services that are normally associated with the actions of physical devices – e.g., there is a functional unit class specification for timers and another for serial lines, another for network controllers, etc. The specific services that 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 OBIOS specification defines the operations and behavior of modules, functional units in general, and specific classes of functional units.

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 distinct commands that initiate I/O or control activity with the hardware device. These commands are collectively referred to as START commands. For example, a serial line functional unit will have START commands for read, write, set baud rate, and others. Finally, functional units must provide an interrupt handling function (intr_svc) for each type of interrupt that a given unit may generate.

The get_module_info function returns information that is statically known about a module.

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 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. This module becomes unusable until a subsequent successful initialize call is made.

The get_module_status function returns information about the dynamic state of a module. This information includes the number of active requests and the status 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.

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

The intr_svc function formally acknowledges completion of one stage of a service that has been initiated by a START 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 initiated by a START command.

The poll_any function is used to seek formal acknowledgement, along with any returned status and data, of completion of one stage of a service that has been initiated by a START command. The poll_any function implements a rendezvous point between the client and device. the poll_any function provides a *polling mode* alternative to intr_svc.

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

The remaining commands that were listed perform the task indicated by their names.

3.1 Single-stage Services

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

A single-stage request for service is accepted and completed via a single call to a START command. Any data transfer that may occur between the client and device is specified in the request block (explained in Section 4.1) that is passed to the START command. A status of *ok* is returned for a successful request.

3.2 Multiple-stage Services

A multiple-stage service is one that, once accepted, cannot be immediately completed. In general, a multiple-stage service takes considerable time to complete and, therefore, is performed in multiple stages to prevent the client from spin-locking the CPU for an excessive amount of time (e.g., reading a sector from a disk, or waiting for an alarm to expire). Such a request transitions from stage to stage either via interrupts (if the request was STARTed in interrupt mode), or via polling (if STARTed in polling mode).

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

3.3 Continuous-stage Services

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

Again, like a multiple-stage request, a continuous-stage request is accepted via a START command, which yields a return status of *ok_pending*. Also, a continuous-stage request proceeds from stage to stage via calls to `intr_svc` or `poll`, either at the interrupt-level or task level (depending upon whether interrupt mode or polling mode was specified to the START command. Unlike a multiple-stage request though, a continuous-stage request will never return a status of *ok* from `intr_svc` or `poll`. A continuous stage request will always return *ok_pending* and is said to never reach completion.

4 Client/OBIOS Interaction

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

1. A set of module function interfaces.
2. A set of unit function interfaces.
3. A description of valid request block formats.

4.1 Request Blocks

All unit procedures that may require direct response from the hardware device must include a request block parameter. For the length of time that OBIOS requests are active (i.e., from their initiation until their completion or abortion) the information recorded in the request block will be *shared* between the client and the OBIOS. This shared information often includes I/O buffers that are to be read or written.

4.2 Procedural Interaction

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

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 START commands. If a request is accepted, the START command returns *ok* or *ok_pending*. If the request is not accepted because the device is busy with another request (and presumably incapable of processing another simultaneous request), the START command returns *device_busy*. If the request is not accepted because the device appears to be acting in an errant manner, *device_failure* is returned.

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

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

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

If the staging mode specified to a START command is interrupt, stages will transition on device interrupts. If the staging mode specified to a START command is polled, stages will transition via client polling. In the polling case, the client is expected to call `poll_any` (specifying the same request block that was passed to the START command) until a status other than *device_busy* is returned (or until a device failure is assumed and a call is made to `abort`). If a status of *ok* or *ok_pending* is received from `poll_any`, 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` or `poll_any` function 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 completed yet, it can only be returned from a polling-mode request. *Device_failure* implies something fatal went wrong. *No_pending_request* implies that there is no request pending for the device and service specified in the request block. This may indicate that the request block that was passed to the START command to initiate a given operation was not passed to `poll_any`.