SMBus Architecture and Implementation Overview

Robert A. Dunstan

Mobile and Handheld Products Group - Mobile Technology Lab
Intel Corporation
robert_dunstan@ccm.jf.intel.com

Abstract

The SMBus was originally designed to add functionality, with a minimum impact on cost (pin count). A single SMBus in a system was envisioned to have sufficient bandwidth and flexibility to communicate with a variety of devices including batteries, LCD contrast and backlight controllers, power plane switches etc. As the SMBus and systems have matured, a different architecture is emerging. On the hardware front, new devices and uses for the bus are appearing, while multiple SMBus segments within a system are becoming common. SMBus host controllers are being adapted to meet these needs. A single SMBus host controller in the chipset has given way to multiple hosts ranging from a dedicated SMBus connecting a pair of devices to embedded controllers that support one or more SMBuses. At the system level, the Advanced Configuration and Power Interface (ACPI) defines standard methods that can be used to identify the SMBus(s) and devices on a given platform extending the operating system's the ability to access and control these devices. ACPI will enable the operating system to coordinate its efforts to control the all system components, including those on the SMBus, resulting in better power management, longer battery life and reduced heat generation.

History

SMBus (System Management Bus) was created to extend chipset functionality without adding more pins. Intel's Mobile and Handheld Products Group (MHPG) found that while more functionality could be added to the chipset, the packaging only had a limited the number of pins available for these new functions. MHPG engineers looking for a solution, identified low-speed serial buses as a way to extend the functionality of their chipsets. At the same time, another group within MHPG was investigating intelligent batteries and needed a simple, inexpensive means to attach the battery to the system. These efforts were combined to define SMBus, based on the existing I^2C serial bus. I^2C's addressing, start/stop and collision detection/correction were adopted unchanged, but the electrical characteristics were tailored to meet the special needs of batteries and a well defined set of higher level protocols was added. The drafts of the SMBus and Smart Battery Data specifications were first published early in 1994, with a Version 1.0 of each released early in 1995. But was SMBus a done deal?

As with many emerging technologies, the SMBus was incomplete. Not the specification itself, but rather the environment. How was data going to get from the SMBus to the system? To meet the need, the SMBus BIOS spec was published. Use of the BIOS allowed system designers to hide the details of the actual SMBus implementation from the system software. At a higher level, there was a need to access the SMBus devices from the operating system. Intel created and made available a set of reference drivers that worked in both Windows 3.1 and Windows 95. These drivers allowed applications to access SMBus devices, and in particular provided access to the Smart Battery.

Implementations

The first commercially available SMBus devices were Smart Batteries targeted at the notebook market. They appeared in commercial quantities early in 1995 with notebooks using them and SMBus BIOSes following later in

1995. However, there were challenges facing the early adopters of SMBus, notably the absence of an SMBus host controller chip or a component with an SMBus UART. These first implementations used the keyboard controller to emulate the SMBus host.

The early systems using the keyboard controller to emulate the SMBus host typically supported a single SMBus device, the Smart Battery. The SMBus host often ignored the fact that the Smart Battery mastered the SMBus to issue alarms and simply depended on the battery's ability to survive until the SMBus host discovered that the battery was in an alarm condition. Ignoring the state of the bus when the host initiated a bus transaction could cause the battery to suffer a temporary communications failure. It also required the host to poll on the SMBus to determine if any device was in an alarm condition. These systems could not take advantage of the SMBus Alert mechanism, which allows a device to assert an alert line that notifies the host of an alarm condition because the Smart Battery does not use this feature.

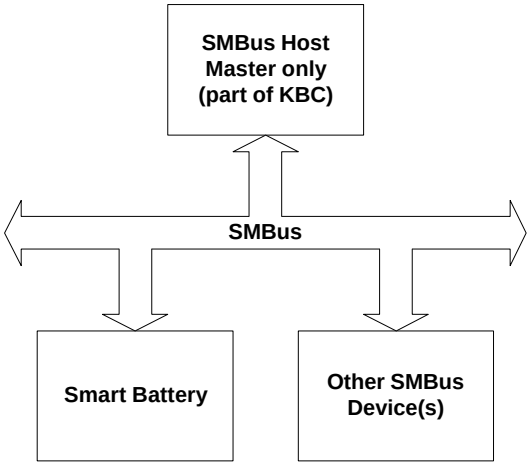

Figure 1.

Many of the early SMBus host emulations failed to observe good SMBus etiquette and simply started writing on the SMBus without checking to see if it was busy. This could cause other SMBus devices to become confused. As these were incomplete implementations of the SMBus specification, they often exhibited other unpredictable behaviors. For example, many had difficulty when the Smart Battery was inserted or removed, particularly during a transaction or when the Smart Battery mastered the bus to send messages to the charger or to the host.

Master only hosts today, take more care in their handling of the bus. They observe bus timing requirements so as not to cause SMBus devices on the bus to time out. They do not try to master the bus when a transaction is in progress. Many periodically reset the bus and/or whenever they detect an unexpected bus state. These improvements have resulted in more reliable SMBus hosts.

Systems today usually support at least two batteries and a Smart Charger. This adds additional requirements to the SMBus host. Hosts that are master only must be able to deal with multiple masters on the bus.

Along with the trend towards multiple batteries is the addition of multiple SMBuses in a system. Systems with multiple Smart Batteries require independent SMBus segments to allow independent access and charger coherency. To ensure safe operation, the SMBus between the Smart Battery Charger and the Smart Battery it is charging must be maintained at all times. Failure to do this can result in spectacular battery failures. But while maintaining a tight connection between the battery and the charger, the host needs to have the capability to read information from all Smart Batteries in the system. To do this, an additional battery system component, the Smart Battery Selector, has been developed.

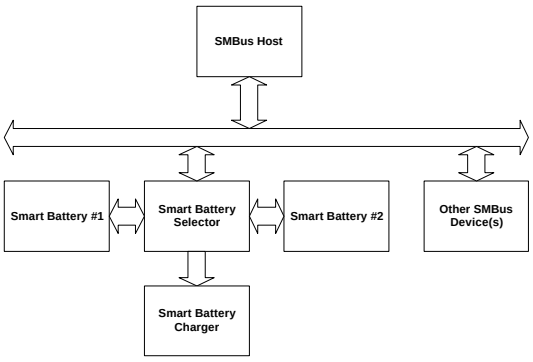

Figure 2

Multiple SMBus segments have appeared in another context. Private SMBus segments between pairs of tightly coupled components are being used. One example is a CardBus controller and a switch that controls the voltage supplied to the CardBus slot. The system

communicates to the CardBus controller in the typical fashion using IO ports. The CardBus controller accepts commands to control the voltage at its slot and then translates them into SMBus commands that go to the voltage selection switch via a private SMBus segment. If the system was required to communicate directly with the voltage selection switch, it would require at least a read/write line, eight data lines and a chip select instead of the SMBus's clock and data line. Using a private SMBus segment saves a relatively large number of pins that can result in a considerable cost saving.

Systems in the future will probably have multiple SMBuses, multiple SMBus hosts and private SMBus segments. Figure 3 below is a block diagram of such a system.

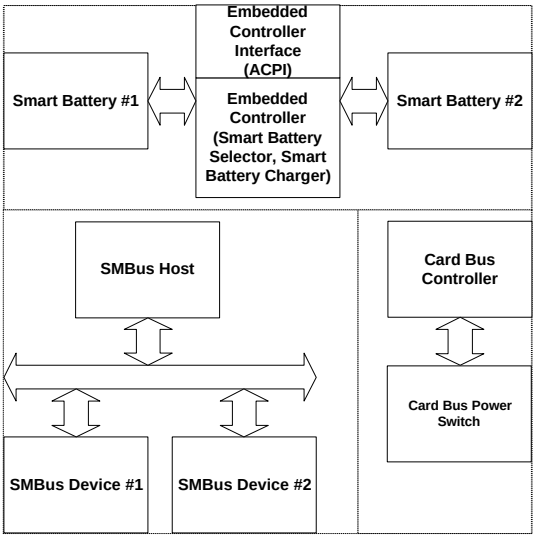

Figure 3

This system shows three different collections of SMBus components. The upper is of particular interest as an example of an embedded controller whose interface is reported to the operating system via ACPI. The key features are:

- ACPI reports the location of the embedded controller register set.
- ACPI defines a set of commands to communicate with devices on the SMBus.

This set of commands allows a standard device driver for the embedded controller to included with the operating system. Since ACPI also supports the Smart Battery systems specifications, a standard battery device driver can be included with the operating system as well.

This example embedded controller also replaces the functionality of two Smart Battery system components, the charger and the selector. At the interface, it looks like all the components are on the SMBus, but the actual implementation is quite different, taking advantage of embedded controller to replace functional blocks reducing component count and real estate. The example embedded controller emulates two SMBus segments to individually communicate with two Smart Batteries. It either operates in a master/slave mode or master only where it must poll the batteries for alarm conditions. When an alarm is detected or there is a status change in the Smart Battery system, the embedded controller issues an SCI (ACPI style event notification) which causes the operating system to identify issuer and service the interrupt.

In this example, the embedded controller may serve another purpose as well. There are some devices that should not be fully exposed. For example, a rogue piece of code could continually force the Smart Charger on the SMBus to charge the battery, potentially causing a battery failure or could command the SMBus power plane controller to turn off the main power plane. These SMBus devices can be "hidden" behind the embedded controller interface, available to the system, but totally inaccessible at the SMBus interface. The embedded controller can reject or ignore requests in order to maintain system safety and integrity. It should be noted that the embedded controller may perform many other system activities as well.

The lower left of Figure 3 represents a chipset with a SMBus UART. In this case the SMBus host is a simple UART and is limited to communicating with devices on the SMBus.

The lower right of Figure 3 represents a pair of devices that use an SMBus segment to pass proprietary control, command and data between the devices. SMBus is attractive in this application because the implementation can be as simple as a pair of shift registers and a clock. Most if not all implementations of this type are expected to have a master only in one device and a slave only in the other. The system has no direct access to the SMBus.

SMBus Host Emulation

A major problem seen in many systems in use today is the quality of their SMBus host emulation. The problems range from the failure to check for bus activity before attempting to master the bus to timing violations. There are generally simple solutions to these problems. In the following paragraphs, several of these problems will be examined and solutions posed.

Failure to detect bus activity before mastering the bus can cause problems for the devices on the bus that are already communicating. It is rarely a problem for the emulated host. Since indiscriminate mastering of the bus can occur at random times and intervals, clock and data transitions outside of the normal timing specs are likely to occur. The SMBus specification recognized that micro-controllers would be used to emulate the SMBus host function and as a result specified a relatively short clock high time during any bus transaction. This allows the SMBus emulation to ensure that the bus is inactive by observing the that clock has remained high for greater than 100µs before attempting to master the bus.

Micro-controllers operating as bus slaves are required to respond to their slave address at a 100KHz rate or else follow the clock stretching rules. If the micro simply depends on an interrupt to respond to the start condition (clock high, data transitions from high to low), the interrupt latency needs to be less than 4.7µs so the micro can meet the timing requirements to read the data. This requirement can be difficult to meet with many micro-controllers taking 20µs or more to respond to the interrupt. The solution here can be helper hardware, that is hardware that holds the clock low when a start condition is detected and is reset by the micro-controller when it is ready to respond. The micro-controller is then free to stretch the remaining clocks in order to control the effective clock rate to one it can easily sustain. This helper hardware is present on some chipsets in use today.

Another area where some SMBus host emulation as well as some SMBus components have difficulty is dealing with transients associated with hot insertion/removal of Smart Batteries. The exact failure mechanism is less clear in this case, but a host should detect a bus time-out and assert a start followed by a stop. This action will resynchronize the SMBus and cause all components on the SMBus to be in a ready state.

Areas for Improvement

The SMBus specification has proven to be workable and many reliable components use it now to communicate. However, as more devices are designed and new uses are found for the SMBus, there are areas in the specification that could be improved. Time-outs, while sensible for batteries, are still relatively long. These long time-outs, in the order of 30ms for SMBus devices, may render the bus unusable for more time critical devices and/or may limit the number of devices that can share the same host.

SMBus uses fixed addressing. This leads to two problems: someone needs to coordinate and assign addresses and what happens when the 100 or so available addresses are all used. The first problem is now being addressed by the System Management Bus/Smart Battery Implementers Forum (SBS-IF). The SMBus specification reserved the addresses used for I^2C's 10-bit addressing which should allow for expansion to 10-bit addressing sometime in the future.

One other troublesome area is semantic. It comes from differing meanings of the word address. An SMBus address is defined as 7-bits. However, a 0 (write bit) is usually appended to that address and the resulting in an 8-bit value is often used as a device's address. The SMBus specification uses the 7-bit convention, while the Smart Battery Specifications use the 8-bit notation. A cautionary note here is that ACPI consistently uses only the 7-bit address.

Where do we go from here?

The SBS Implementers Forum was announced late in 1996. It consists of a core group of silicon vendors and battery vendors. In addition, there are other members from various industries. This group owns the SMBus Specification as well as the Smart Battery Data Specification, the Smart Charger Specification and the Smart Battery Selector specification.

This SBS-IF was an outgrowth of the Smart Battery Plugfests. The Plugfests allowed the

participating companies to get together and verify the interoperability of their components., Changes and improvements to the specifications are discussed at the Plugfests. They are open to all SMBus component manufacturers who are invited and encouraged to test their component's interoperability with those from other manufacturers.

The inclusion of SMBus in the Advanced Configuration and Power Interface (ACPI) is very important to the continuing success of SMBus. The ACPI specification was authored by Microsoft, Intel and Toshiba and endorsed by many other OEMs. It is a publicly available specification that defines interfaces and methods that the operating system can use to access, configure and control devices on the system board. ACPI is based on open, publicly available specifications and its inclusion of the SMBus, means that SMBus devices are part of the mainstream. An SMBus device driver will be part of the operating system. Well defined interfaces will allow standard device drivers for common SMBus devices, such as Smart Batteries to be included with the operating system.

ACPI is the latest in a series of initiatives from Intel, Microsoft and computer OEMs to simplify system configuration and reduce device/system power consumption.

Call to action

- Get a copy of the specification (www.sbs-forum.org).
- Use the SMBus where appropriate.
- Participate in the SBS-IF.