VLB ARRAY MEMO No. 682

NOTES ON USE OF THE VLBA MONITOR AND CONTROL BUS

Larry R. D'Addario
September 26, 1991

SOME PRELIMINARIES

The VLBA Monitor and Control Bus (MCB) is a multi-drop, bi-directional digital communication channel for connecting a single "controller" to many "devices" that can accept or supply information. It was invented to serve the needs of the VLBA stations, where numerous receivers, signal processors, recorders and a large antenna must be controlled and monitored in real time.

The MCB is defined by NRAO Specification A55001N001. Another specification, A55001N002A, describes a specific implementation of an interface from the MCB to a device; this implementation is known as the "VLBA Standard Interface" (SI) and consists of a printed circuit board that has been produced in large quantities for the VLBA project. For anyone having an interest in using the MCB, the first specification in required reading. To date, most users have adopted the Standard Interface rather than taking the trouble to design their own. For them, the second specification is also required reading. But it should be clearly understood that the two specifications are separate, and that a device designer need not use the SI; in fact, as we shall see, it imposes restrictions beyond those of the bus specification, and these are undesirable in some applications. Even for those adopting the Standard Interface, there is no excuse for not reading and being familiar with the basic bus specification.

Unfortunately, the cited Specifications, while logically complete, do not provide much insight into the design philosophy of the MCB, nor do they offer much advice about its proper and efficient use. It is my hope to fill this gap with the present document.

Various people outside the VLBA project, both inside and outside the NRAO, have made use of the MCB. In many cases, this has been because they are using electronic hardware designed for the VLBA and already containing an MCB interface (e.g., the VLBA recorder); their job, then, is to provide a connection to a computer and appropriate software to effect communication with the VLBA module. Other projects, not involving VLBA hardware, have used the Standard Interface because of its ready availability (at least within the NRAO) as a quick way of establishing a computer connection to some new device, even though another approach would be more efficient if they were starting from scratch (e.g., the 230 GHz tipping radiometers used for site surveys). Finally, new projects are underway or may come up in the future for which the MCB is very appropriate (e.g., the Green Bank OVLBI earth station and the GBT) whether or not VLBA hardware is involved. For the benefit of all of these users, especially those who do not have close connections with the VLBA project, a tutorial discussion of the MCB would seem to be very useful.

The VLBA MCB is based on a preliminary specification that I wrote in 1984. Important revisions were later made by Barry Clark and therefore his name appears as the author of the final, published specification. I also originated the idea that there should be a standard interface design, so as to avoid duplication of design effort by many users, and I suggested some features that it should have. But I did not design the SI nor write the specification for it; this was done mainly by Dave Weber and Wayne Koski. Yet, strangely enough, my name appears as author of the SI specification because I did the final revisions just before it was issued.

Recently, a detailed manual on the SI was issued [W. Koski and D. Weber, VLBA Technical Report No. 12, July 1991]. This is valuable for those who must maintain the SI boards or who might wish to modify them or copy portions of their design (including firmware), and for these purposes I recommend it highly.

The rest of this document assumes that the reader is familiar with at least the main MCB specification, A55001N001. Various details of the MCB that are already well covered there will be omitted here.

MCB DESIGN PRINCIPLES

I have often been asked why the VLBA project did not adopt an existing standard for data communication within a station, rather than inventing something new. The IEEE-488 bus is frequently suggested, since it too is a multi-drop bus. In fact, we considered IEEE-488 as well as several other commercially-available systems. All had serious problems for our application. We needed to be able to address a fairly large number of devices (IEEE-488 is limited to 31), some of which would be 100-200 m from the controller (IEEE-488 is limited to a 15 meters). We also were very concerned about reliability, given the unattended and remote nature of the VLBA stations. For this reason, we wanted something with very few wires and hence few pins per connector, dictating serial data transfer (IEEE-488 uses 8-bit parallel data plus 8 control wires, leading to thick cables and large connectors). Finally, we wanted something with a very simple protocol, so that interfaces could be implemented easily and cheaply, in view of the expectation that they would be needed in large numbers. Fortunately, the VLBA control problem requires only modest data rates, and nearly all functions can be implemented with very short data messages (1 or 2 bytes); this is consistent with using a simple serial protocol.

These considerations led to a design with only two twisted pairs of wires, one for transmission in each direction. Asynchronous transmission was chosen, based on the standard start-bit/stop-bit framing of each byte; the overhead bits were judged a good tradeoff against the need for a separate clock line, especially since the framing is implemented in commonly available UART chips and still allows data rates that are more than fast enough. The bit rate was fixed at 57,600/sec. In applications less demanding than a VLBA station, a slower rate would often be more convenient; this will be discussed in more detail later.

Bus Protocol

The bus protocol was made as simple as possible. Only one controller (computer) is allowed, and it must originate every bus transaction. That is, devices must be polled for information and they cannot interrupt the controller at arbitrary times. Although this makes programming of the controller less convenient in some environments, it eliminates the need for any mechanism to deal with bus contention by multiple devices. To further simplify the interface design, signals from the controller are broken into messages which always have the same format: they are exactly five bytes long, consisting of a beginning-of-message ("sync") byte, a two-byte address, and a two-byte value.

Addressing

A major concept of the MCB design that seems to have been poorly understood by many users is the addressing scheme. It is intended to appear very much like memory-mapped I/O on a microprocessor. The address field of each controller message consists of a 15-bit number giving the destination of the message in a space of 32768 possible addresses, plus a 1-bit read/write flag. The latter specifies whether the following 16-bit value is to be sent to the destination, or a 16-bit value is to be requested from it. In this way, any device connected to the bus can be assigned to any subset of the address space; complicated devices can uses many addresses, and simple ones very few (perhaps only one). Now here is the critical point (and the one that tends to be missed): the address specifies the *final* destination in the controlled/monitored device; it does not specify the *interface* by which the device is connected to the bus. This is rather unlike other schemes (including IEEE-488), where each collection of hardware that might be thought of as a separate "device" typically has its own interface, and it is the device/interface that has an "address" (typically only one). In such schemes, internal workings of the device are distinguished by sub-addressing or by the syntax of the value portion of the message; flexibility then demands variable-length messages with device-dependent syntax. Our scheme is drastically simpler, but still allows great flexibility in the logical organization of hardware into devices. We can easily have an arbitrary mixture of simple and complicated devices. Furthermore, the interfaces connecting the bus to the devices are logically transparent. The programmer need not be concerned about which interface is transmitting his message to the destination; he need only have the right address. In fact, if several logically and physically separate devices are located near each other, it might be economical to service them through a single interface; conversely, a single, large device might be conveniently connected through two or more interfaces -- yet all of this is transparent to the programmer.

Some confusion about the addressing scheme may have arisen because of the way the Standard Interface works. Each SI handles a single block of contiguous addresses. The SI connects to the bus by the two serial data lines, but it connects to the device with 16 bits in parallel for the value and 8 bits in parallel for the "relative address"; the latter is the difference between the bus address and the beginning of the SI's assigned block. This allows fewer wires to be used for the address information, but limits each SI to a block of at most 256 addresses (note that this limit does not apply to a general, non-standard interface). This may make it seem that some knowledge of the interface's operation is needed, namely its base address and block size. But this is not true; one can always communicate with any desired function in a device by knowing its 15-bit address, without knowing through which interface it connects, indeed without knowing anything about the structure of the address assignments to the interfaces on the bus.

Bus Protocol, Continued

To complete the discussion of bus protocol, we should consider the responses required to a controller-initiated message. If it is a "control message" (read/write bit set for "write"), then the destination device need only return an acknowledgement that the message was received. The protocol requires two separate responses, each a specified 1-byte code: the first indicates that some device recognized the address as belonging to it; the second indicates that that device received the value portion of the message correctly. The only data integrity check provided is byte parity. If either address byte has incorrect parity, then there should be no response (otherwise two devices might respond to the same message). If either value byte has incorrect parity, then the value of the second acknowledgement byte is changed to indicate the error; this is possible since the address has already been correctly decoded, so only one device can respond. In the case of a "monitor request message" (read/write set for "read"), the response consists of three bytes: an acknowledge code to indicate a recognized address, followed by the 2-byte value being read.

Allowance must be made for failures. An addressed device might be powered down, or otherwise not working; the controller software may contain bugs that cause unassigned addresses to be polled; etc. All such cases are handled by timeouts. There is a maximum allowed time for each kind of response, and if none occurs within this time, the controller is free to send the next message. The specification attempts to set these time limits in a clever way. If every device responds as fast as possible, then the controller can send messages continuously with no gaps, because the bus is full-duplex and some of the returned bytes from one message can be overlapped with the outgoing bytes of the next. On the other hand, devices need not be this fast. At least for monitor requests, some time must be allowed for the device to acquire the requested data. The maximum allowed time was intended to be sufficient for an analog to digital conversion using readily available chips. The choice of timeout limits is thus a compromise between making difficult demands on the device speeds and delaying other, perhaps critical, bus transactions because of a slow or failed device.

Bus Electrical Characteristics

The two twisted pairs that constitute the physical bus, and the signals on them, are required to conform to RS485. The latter describes differential, serial data on a multi-drop line. It is similar to RS422, which also describes differential, serial data but allows for only point-to-point connections. In selecting hardware for connecting a computer to the MCB as the bus controller (and not as one of the device interfaces), it is possible to use cards that support either RS422 or RS485 (or converters from RS232 to either of these) because the controller's transmit line is always driven. The device interfaces, on the other hand, need to have three-state drivers which become active only when the particular interface is addressed.

The RS485 specification allows for only 32 devices on one pair of wires, and most manufacturers of driver and receiver chips conforming to this standard will guarantee performance only up to this number of connections. Users should be aware, however, that this does not limit the number of devices logically connected to the MCB. To have more than 32, one can operate several sets of lines in parallel at a cost of two chips per additional line-set at the controller (star configuration); or one can include repeaters along a single daisy chain (linear configuration). One need not use any additional serial ports (UARTs) at the computer. (At the VLBA stations, separate serial ports are used to drive separate branches of the bus, and these are made logically the same through software; but this was not necessary, and was done because the VME board chosen happened to have six UARTs that were not needed for anything else).

FLAWS IN THE PRESENT MCB SPECIFICATION

Fixed baud rate

The fixed rate of 57.6 kbaud has proved inconvenient for some applications. For example, it is difficult to support on some laptop computers which would otherwise be valuable as portable test generators for MCB devices. There are several ways in which variable rates might have been allowed. One is to add a separate clock wire-pair to the bus, with the controller required to send a square wave on this line at some multiple of the desired rate. Another is to require each interface to support automatic rate adjustment based on the rate observed on the outgoing data line. Both of these were thought to add too much complexity.

New systems can use a different baud rate, provided that it is done consistently throughout, but this will make them incompatible with existing devices. Note that all timing specifications are changed proportionally. If the SI is used, the baud rate can be lowered by a simple change to the firmware, which requires burning a new ROM and plugging it in. Such ROMs have been made at the NRAO and used for various small-scale tests. However, the biggest advantage of going to a lower rate would be a simplified interface, based on a different microprocessor that might allow lower power consumption and perhaps lower chip count; this would require a new design. In some applications a higher speed might be desirable. The bus electrical specifications, based on RS485, would support this, but the SI would not; the microprocessor being used is not fast enough to keep up with higher rates and perform all the functions required by the specification.

Wrong parity for certain bytes

As mentioned earlier, the only data integrity check is byte parity; a ninth parity bit is included with each 8-bit byte (so that, with start and stop bits, 11 bit times are needed for each byte, or 55 bit times for each 5-byte outgoing message). The two address bytes of outgoing messages and the two value bytes of both outgoing and return messages are sent with odd parity. In the final revision of the bus specification (but not in my original draft), the special-code bytes (including the sync byte in outgoing messages and all acknowledge bytes in returned data) were required to have even parity. Although there was a good reason for this, it turns out to have been a major mistake.

The idea was to ensure that the special bytes could not occur in random data, and this was designed to protect against the following type of failure: Each interface on the bus must use the occurrence of the sync byte to mark the beginning of a message, and thereafter check the two address bytes for messages assigned to it. Suppose that a message addressed to one interface contains a value byte that happens to be the same as the sync byte; then, were it not for the parity inversion, all other interfaces would take this to be the beginning of a message. They would then take the true sync byte of the next message to be an address byte, so that the next message would be missed. There are other ways to avoid this problem, but the inverted-parity method seemed simple and straightforward.

Unfortunately, the parity reversal has serious negative consequences. First of all, no commonly available hardware allows for efficient implementation of frequent changes of parity during serial transmission. In practice, serial data interfaces to nearly all microcomputers are implemented with UART chips of the same family. These are all bi-directional, full duplex, byte-buffered devices with parity generation and checking implemented in the chip hardware. The parity sense and various other parameters are set by programming certain registers in the chip. Transmission and reception can proceed simultaneously and independently, but there is only one bit for setting the parity sense of both. Since this bit must be toggled in order to cause the specified parity change during transmission of a message, the parity checks on the received bytes cannot be meaningfully interpreted. Two solutions to this have been in use: (1) use two separate UARTs, one for transmit and one for receive (this is usually two separate serial ports of the computer); and (2) ignore all parity errors detected on the receive side, foregoing this check of the data integrity. The first approach is not too unreasonable, since adding serial ports to a computer (especially a PC) is usually cheap; but all PC implementations of which I am aware use the second approach.

However, there is another problem with the parity switching, and it is more serious. In transmit, obtaining full throughput from a serial port without tying up the CPU requires proper buffering. The common UARTs allow the CPU to write to a buffer register while the preceding byte is being transmitted; transmission of the new byte then begins automatically immediately after the present one finishes. The newer UARTs allow the CPU to write many bytes to a FIFO, and these are then transmitted in rapid succession without bothering the CPU. But these features cannot be used with the parity-switching protocol; for two bytes out of each five, the CPU must wait until the present byte is completely transmitted before loading the next one, because it must first switch the parity control bit. It can be arranged that an interrupt is generated at this time, but this causes many more separate interrupts than would otherwise be required, costing CPU overhead; and all of the interrupt latency time becomes unnecessary dead time on the outgoing serial line. This was handled for the VLBA stations in a very expensive way: a dedicated, powerful CPU is used to handle just the transmission and reception of bytes from the (two separate) serial ports used to connect to the MCB.

A better design would have left the parity sense always the same. To avoid the problem described above, one could begin by requiring each interface to expect sync bytes only every five bytes; the four bytes following a valid sync byte would not be treated as beginning-of-message bytes, even if their values happen to equal that of a sync byte. To acquire syncronization in the first place, and to recover from any accidental loss of sync, one can require that when a sync byte was expected but not found then all succeeding bytes are treated as potential sync bytes until the next one is found.

A revised bus specification along the lines of the last paragraph could be used for new applications. Unfortunately, I see no way to correct this deficiency such that a revised system remains compatible with existing hardware.

Soft addressing

Another major problem with the MCB specification is its requirement for soft address assignments for each interface. It should be obvious that each outgoing message may elicit a response from only one device; hence each device must be assigned a set of addresses distinct from those of all other devices. The bus specification requires that this be ensured by assigning only one block of contiguous addresses to each *interface* that is connected to the bus, and furthermore it requires adherence to a scheme by which the starting address and length of this block is set by the controller using the bus itself. Exactly how this is accomplished will be discussed shortly. The idea seems to be that in a large system there may be a need for frequent upgrades, including the addition of new devices; so it might be necessary to re-organize the address space fairly often. In that case, the ability to re-assign all the address blocks via software alone, without having to make any hardware changes, is a valuable feature.

The obvious alternative would have been to have each device (or interface) "hard wired" to a particular set of addresses. In order to allow a standard interface to be mass produced, the address information might have been stored in a ROM or in a set of jumper wires on a plug. Changes in the address assignments would then require replacing the ROM or plug in each affected interface. This scheme was certainly considered during development of the bus specification. One of the strongest arguments against it was the need to stock spares of all the ROMs or plugs, and to ensure that the correct one is installed whenever an interface or a module containing an interface needs replacement. This was judged to be too big of a burden on the maintenance staff.

However, the experience of all users of the MCB, both inside and outside of the VLBA project, has been that the considerable disadvantages of the soft-addressing scheme strongly outweigh these supposed advantages. We will discuss these disadvantages shortly. I say "all users" advisedly; I mean that I have not yet met anyone who has used the MCB from either the hardware or software side and who disagrees with this statement. All would prefer the hard wired addressing scheme to the one we have.

Before discussing the problems with the soft addressing, we will review for reference how it works. To begin with, addresses 0 through 255 are reserved for the special purpose of initializing the interfaces. Upon power-up, each interface must respond to exactly two of these addresses, so that a maximum of 128 interfaces can be accommodated. Each interface must have a fixed, unique "ID" value N, where 0<=N<=127, and the special addresses assigned to it are then 2N and 2N+1. The value in 2N+1 is the starting address of the block currently assigned to that interface, and the value in 2N is the length of the block. Both values can be read as well as written.

The difficulties introduced by this scheme are these:

1. The controller cannot communicate with any device until its interface has been initialized, and it cannot be completely sure that everything remains initialized unless it makes explicit checks. Interfaces can lose their initialization (forgetting who they are, so to speak) without the software knowing about it. This has often caused simple software to hang up, requiring manual intervention for recovery. More robust software is needed for operational systems, and to produce it is complicated and messy. If an interface is powered down and then back up, no device connected to it is accessible until it is re-initialized. Furthermore, the initialization can be disrupted by unintentionally sending commands to the special addresses, as can happen during software debugging.

2. The transparency of the interfaces is destroyed. Recall that in the section on Design Principles I made a big deal about the fact that the addressing scheme makes the interfaces transparent to the programmer. Now I must take that back, at least to some extent. Everything I said remains true provided that the initialization has been done. At first this does not seem too bad, since a software system can include the bus initialization code along with that for various other initializations that it must do at the beginning of its execution; thereafter it can ignore the interfaces and concentrate on the final devices to be controlled -- set it and forget it, if you like. This is fine if nothing ever goes wrong. But suppose that later it finds that there is no response from some address. In order to attempt a recovery, it must remember not only what device the address goes with, but also what interface. It then must poll the special addresses associated with that interface to see whether it is correctly initialized.

3. The method chosen to allow the soft addressing, namely assigning an "ID byte" to each interface, negates one of the main purposes of the scheme. Each interface must still have a ROM or plug with the ID value set into it! True, only 8 bits are needed compared with a potential 2x15 bits to allow specification of all possible starting addresses and lengths. But I will show below that by imposing much milder restrictions than those of the current specification one can get away with 15 hard-wired bits, and this would be negligibly more of a hardware burden than the 8 bits now required.

4. The full flexibility of the addressing scheme is limited. To accommodate the ID byte concept, the number of interfaces was limited to 128. While this has so far been adequate for the VLBA stations and for many other applications, it is a severe cut from the 32768 separate devices (at one address each) that would otherwise be allowed, and this limits the potential applications of the bus.

5. The implementation of the ID byte in the Standard Interface is awkward. This will be discussed in the next section.

6. Generally, the soft addressing scheme has made the bus

much more complicated to program than it should be, introduced a host of pitfalls for new users, and created undesirable potential for software errors.

Having said all that, I am glad to provide some good news: there is a way around this. For new systems, the soft addressing portions of the MCB specification can be ignored, and the address assignments of devices can be hard wired. This point does not seem to have been appreciated by any potential users, but an important thing to notice is that doing so does not introduce any incompatibility. Old devices that have interfaces needing soft addressing can still be connected to the new system; they must be initialized, of course, even though their newer busmates need not be.

One of the objections to hard addressing is that the storage of the fairly large number of bits needed is too much trouble; it's too few to justify a dedicated ROM (or even re-burning some bits of an already-included ROM), and too many to handle with a plug. But here's how I would handle it using only 15 bits. Suppose we restrict the starting addresses of blocks to multiples of 32; then only 10 bits are needed to specify one of them. Block sizes then might as well also be multiples of 32, and with our remaining 5 bits we can specify any such block up to a size of 1024. While this addressing scheme is still somewhat restricted, it is far less so than that of the present specification. Furthermore, with hard addressing, such restrictions are not part of the basic bus specification at all, but rather part of a specific implementation. If you are willing to have more than one type of interface on your bus, then some can have more general addressing capabilities than others. I choose 15 bits because they can be programmed with jumpers on a common 16-bit DIP header (the remaining pin being ground).

DEFICIENCIES IN THE STANDARD INTERFACE IMPLEMENTATION

I have the following difficulties with the present SI design:

1. There is too much emphasis on analog signal acquisition. Good support for the monitoring of analog voltages was considered an important requirement when the SI was being designed, and indeed this has been useful in many VLBA station modules. However, there are also many applications where no A-to-D conversion is needed. A better design might have been to build a digital-only board and to provide a second add-on board for analog support. But even accepting the decision to include both on the same board, it seems to me that the designers went too far: the analog circuitry accounts for roughly 40% of the board area, 21 of 50 pins on the device connector, about 6 out of 14 text pages in the specification, and 2 out of the 3 power supplies needed. It also imposes a lot of complexity in terms of further restrictions on the addressing scheme and external logic that the user must supply, in most cases. Most of the complexity comes from multiplexing: there is just one ADC on board, but with an 8-way multiplexer and provisions for external sub-multiplexing. Admittedly, the connections and addressing are very simple for the user who wants to monitor 8 or fewer voltages and who has no control or digital monitoring to do; then he merely connects his voltages to the appropriate pins. The other extreme, no analogs at all, is also fairly simple; the user can remove all the analog chips, provide only

the +5 V supply, and there is no extra burden except the wasted board space. But for all other cases, which account for nearly all uses of the SI so far, it is much more complicated.

2. The implementation of soft addressing (the ID byte) is messy. The design of the SI board was mostly complete by the time that the idea of soft addressing was introduced into the bus specification, and there was then no room on the board to add a socket that might take a jumper-programmed plug. Besides, it was realized that in some applications the ID should be determined fairly far from the interface and hence off the board (this is the case when a system includes several identical modules, each of which must be addressed according to where it is plugged in). Therefore the ID byte needed to be read through the device connector. But all the pins here were already assigned, so some of the 16-bit parallel I/O lines were multiplexed to read the ID byte. This means that the user must provide a tri-state buffer for the ID in addition to some non-volatile storage (typically a set of jumper wires or DIP switches). Furthermore, there was not even one spare pin on the device connector for the signal that enables this buffer ("ID Request"), so that signal comes from the other connector, which would otherwise be dedicated to the serial data to/from the bus. All of this is, as I said, messy.

There is one redeeming point, though. If we want to use the SI with hard-wired addressing as suggested in the preceding section, it can be done with a firmware change only. Instead of just using the bottom 8 I/O bits for the ID byte, we can use all 16 to encode the full specification of the address block.

3. The power consumption is rather high. About 700 mA is needed at 5 V (or 650 if all analog chips are removed). This is because a fairly fast microprocessor is needed to handle the 57.6 kbaud bus rate and meet all the timing constraints. However, I don't know of any way to do better.