DIGITAL'S TRI/ADD PROGRAM

A FULL SPECTRUM OF HARDWARE SOLUTIONS

ACCESS.BUS™

TECHNICAL OVERVIEW

Specifications for this open interconnect are offered
by Digital Equipment Corporation and Philips/Signetics

Version 1.1

digital™ Digital Equipment Corporation

100 Hamilton Avenue Palo Alto, CA U.S.A. 94301 [1] 415.853.6531 FAX [1] 415.853.0155 triadd@decwrl.dec.com

Australia	France	Germany	Italy	Japan	U.K.	U.S.A./Canada
0014.800.125.388	05.90.2874	0130.81.1974	1678.19087	0031.12.2363	0800.89.2610	1.800.678.OPEN

Notice to prospective developers:

This document is provided for developer's who are interested in learning about a new technology available to the open computing marketplace. The information is subject to change because it is based on specifications that are being completed for availability in the Fall of 1991. In particular, this document contains preliminary protocol information that may not be implemented as described in this version of the Overview. Vendors developing devices based on the ACCESS.bus or integrating it into host systems should check their version of documentation with the Program (see Section 7 for telephone numbers) before beginning any design work.

Part number for ordering: TA-0021-2

1. Introduction

ACCESS.bus™ (a BUS for connecting ACCESSory devices to a host system) is a peripheral interconnect system defined and developed by Digital Equipment Corporation in partnership with Philips/Signetics and offered to the computer industry as an open standard. This overview aims to introduce the prospective developer of ACCESS.bus systems or peripherals to the essential technical features of this interconnect.

1.1 What is ACCESS.bus?

ACCESS.bus is a system for connecting a number of relatively low-speed peripheral devices to a host computer, typically a desktop system, such as a workstation, personal computer, or terminal. The appropriate devices include both interactive peripherals - keyboards, locators, hand held scanners, bar code readers, and magnetic card readers - and non interactive peripherals - printers, plotters, and signal transducers in real-time control applications. Further, the ACCESS.bus protocol is general enough to accommodate a wide range of unusual peripheral types [Figure 1].

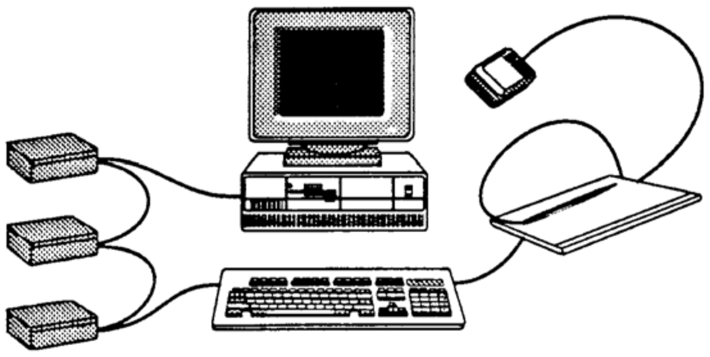


Figure 1. ACCESS. bus connects keyboards, locators, and text-type devices to a system

ACCESS.bus has the topology of a bus. A single ACCESS.bus on the host can accommodate up to 14 peripheral devices (or more when devices can share a common controller). The total length of the cable connecting the devices on a common ACCESS.bus may be up to eight meters. ACCESS.bus supports a maximum aggregate data throughput of approximately 80 Kbits/sec.

Digital and Philips/Signetics has made the ACCESS.bus an open specification, enabling any vendor to implement it on host systems or in peripheral devices, without fee or royalty. The two companies are working together to promote ACCESS.bus as an industry standard. The Advanced Computing Environment (ACE) initiative has designated ACCESS.bus as an option in the Advanced RISC Computer (ARC) specification. Digital plans to organize a committee of vendors to endorse the ACCESS.bus as a standard and to guide its future development.

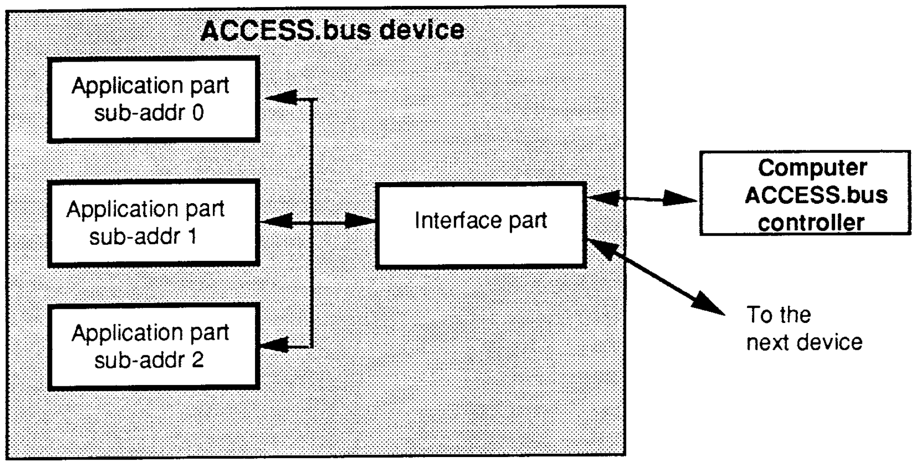
Forthcoming Digital workstation products, both host systems and appropriate peripheral devices, will include ACCESS.bus as a built-in feature. ACCESS.BUS offers a number of advantages both to end users and to the developers of systems and peripheral devices. It is an advantage that a host computer needs only one hardware port to connect to a number of devices. The commonality in communication methods for a number of device types leads to economy in software and hardware development. As an open industry standard, ACCESS.bus will stimulate development of diverse peripheral devices, each usable with a number of different types of host system. Further advantages of low cost, ease of implementation, and ease of use are consequences of particular features of the ACCESS.bus design presented in this Overview.

The idea of a bus-topology interconnect for desktop interactive peripherals is fairly new in the computer industry. ACCESS.bus incorporates more sophisticated technology and offers higher performance than any other such systems. Moreover, it is the first system of this kind to be offered as an open standard.

1.2 ACCESS.bus Hardware

At the hardware level, ACCESS.bus is based on the well-established InterIntegrated Circuit (I^2C™) serial bus developed and patented by Philips. The serial bus architecture, in which a single data line carries one bit of information at a time, entails lower costs for cabling, connectors and controller circuitry than parallel bus architectures. As its name indicates, the I^2C serial bus was developed primarily as a means of connecting integrated circuits.

Standard low-cost IC components available from Philips handle the logical complications of the bit-level handshaking. For the ACCESS.bus, the relevant components are the microcontrollers of the 80C51 family, which provide the intelligence for executing the ACCESS.bus protocol in peripheral devices and in host systems. More detail on these components are given in Section 5.


ACCESS.bus device
Application part sub-addr 0
Application part sub-addr 1
Application part sub-addr 2
Interface part
Computer ACCESS.bus controller
To the next device

Devices are usually identified with their microcontrollers. However, the ACCESS.bus system includes provisions for controlling up to four subdevices with a single microcontroller. The subdevices may be of diverse functional types. The term "Interface Part" refers to the unique controller of a device, and the term Application Part refers to the application-specific functionality. A device has one Interface Part and up to four Application Parts [Figure 2].

The physical medium for ACCESS.bus is a shielded cable containing four wires: serial data (SDA), serial clock (SCL), power (+12v), and ground. It uses standard low-cost shielded modular connectors available from AMP and Molex [Figure 3]. Shielding of the cables and connectors facilitates making ACCESS.bus-based systems conform to FCC radiation requirements. A typical ACCESS.bus device will have two connectors so that devices may be chained on the single The bus; hand-held devices may have a captive cable joined to the bus trunk with a "T" connector.

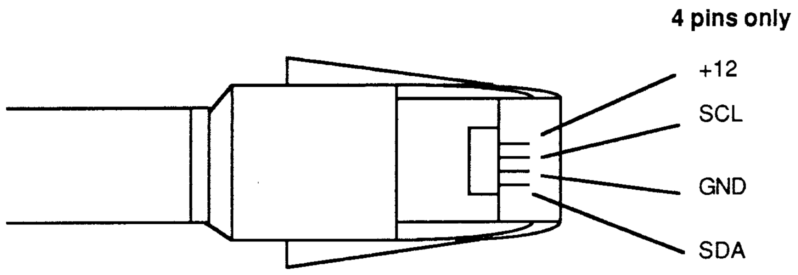


Figure 3. A shielded modular connector with only four pins for connecting the ACCESS. bus cable to a system

The way that the serial data and serial clock lines work together to define the information carried on the bus is described in Section 2.3. The +12 V power line provides up to 500 ma to supply the peripheral devices.

The I^2C technology can support clock rates up to 100 KHz. The maximum ACCESS.bus data transfer rate of approximately 80 Kbits/sec is derived from the top clock rate by subtracting the overheads imposed by the ACCESS.bus communication protocols for handshaking, addressing, and error control.

1.3 ACCESS.bus Protocols

The ACCESS.bus communication protocol is composed of three levels: 1^{2}C protocol, Base Protocol, and Application Protocol.

At the lowest level, nearest the hardware, the basic discipline of the ACCESS.bus is defined as a subset of the Philips Inter-Integrated Circuit (I^{2}C) bus protocol. The simple and efficient I^{2}C protocol defines a symmetric multimaster bus on which arbitration among contending masters is effected without losing data. I^{2}C provides for cooperative synchronization of the serial clock for exchange of data between bus partners with different maximum clock rates. The 1^{2}C protocol defines a bus transaction scheme with addressing, framing of bits into bytes, and byte - acknowlegement by the receiver. More detail on the 1^{2}C protocol level is given in Section 2.

The next ACCESS.bus protocol level is the Base Protocol. This level, common to all types of ACCESS.bus devices, establishes the nature of ACCESS.bus as an asymmetrical interconnect between a host computer and a number of peripheral devices. The host plays a special role as a manager of the ACCESS.bus. Data communication is always between host and peripheral device and never between two peripherals. While the I^{2}C protocol provides for mastership by either the sender or the receiver of a bus transaction, in the ACCESS.bus protocol masters are exclusively senders and slaves are exclusively receivers. Of course, the host and all the devices are both master/senders and slave/receivers at different times.

The ACCESS.bus Base Protocol defines the format of an ACCESS.bus message envelope, which is an I^{2}C bus transaction with additional semantics, including checksum reliability control. Further, the Base Protocol defines a set of seven control and status message types which are used in the configuration process.

Two of the unique features of this configuration process are auto-addressing and hot plugging. Auto-addressing refers to the way that devices are assigned unique bus addresses in the configuration process, without the need for setting jumpers or switches on the devices. Hot plugging refers to the ability for attaching or detaching devices while the system is running, without the need for rebooting the host. The means by which the ACCESS.bus protocol provides these features is discussed in Section 2.

The highest level of the ACCESS.bus protocol, the Application Protocol, defines message semantics that are specific to particular functional types of devices. Different device types require different Application Protocols. Application Protocols have been defined so far for three device classes: keyboards, locators, and general text devices. Each of these predefined classes is fairly broad. Thus, the keyboard device protocol provides for communicating essentially all conceivably relevant state information about a keyboard with up to 255 keys. The locator device is defined broadly enough to include not only the most common pointer devices - mice, tablets, trackballs - but also valuator sets (such as dial boxes) with up to 15 valuators and function button boxes with up to 16 buttons. The text device represents essentially any character stream device, such as card readers, printers, bar code readers, or modems. The predefined Application Protocols are discussed in greater detail in Section 3.

A major advantage in designing devices that conform to these general devicetype semantics is that they may share device-specific software levels in common, both in the device-resident firmware and in the driver software needed in the

host operating system to allow application programs to access the devices. Digital anticipates that further device-specific Application Protocols will be defined in the future, as industry standards under the aegis of the planned ACCESS.bus committee. Further, any device vendor may implement a special device protocol within the general message envelope defined by the Base Protocol.

Participation in all three of the protocol levels requires intelligence at the device. The same microcontroller supplies this intelligence at all levels through appropriate firmware. The lower levels of this firmware are likely to be common to many devices. Higher levels of the firmware are expected to be more specific to the device and the application [Figure 4].

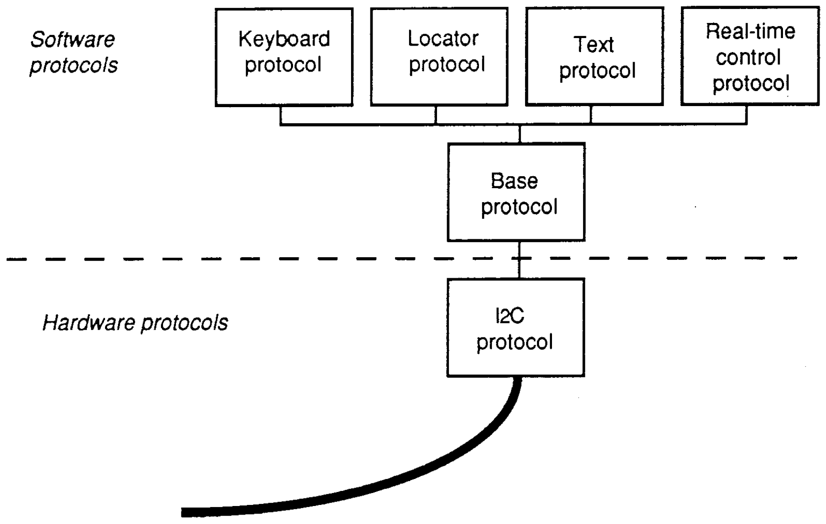


Figure 4. ACCESS. bus protocol hierarchy

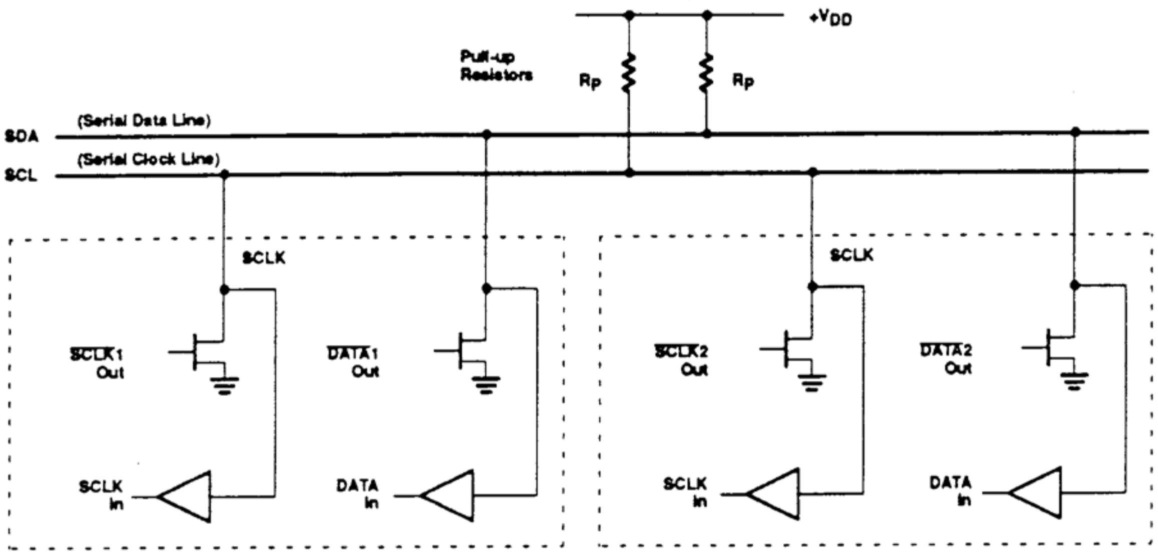
2. How ACCESS.bus Works

2.1 Electrical

The host and devices are connected to both the serial data (SDA) and serial clock (SCL) lines in a "wired-AND" logic configuration. The wired-AND may be implemented by connecting the data and clock output stages of each bus partner to the SDA and SCL lines respectively through open-collector or opendrain transistors. The standard 1^2C components include such output stages on-chip. The significance of the wired-AND logic is that any attached bus may force either of these lines to low (the ground level). When there is no output from any bus partner, the lines are held high by pull-up current sources in the host. And of course, every bus partner can sense the level on both of these lines [Figure 5].

Figure 5. Connection of devices to the I²C Bus.

2.2 Bus Transactions



During a bus transaction, there is one clock pulse on SCL for each bit transferred on SDA. The SDA information is valid when SCL is high. During a transaction, the SDA must be stable between the rising and falling edges of the SCL pulse; SDA may change state only when SCL is low [Figure 6]. SDA transitions when the SCL is high are signals that delimit the

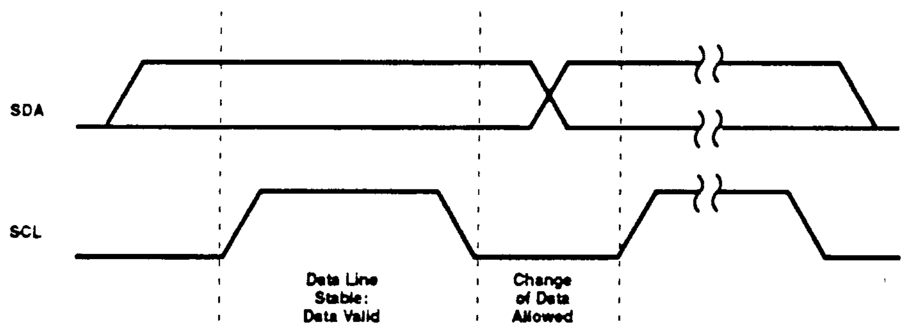


Figure 6. Bit transfer on the I²C Bus

Bus transaction. When the ACCESS.bus is free, both SCL and SDA are high. A high-to-low SDA transition when SCL is high is a start condition; it signals the beginning of a bus transaction. A bus partner asserts mastership by pulling SDA low when the bus is free. A low-to-high SDA transition when SCL is high is a stop condition; it signals the end of a bus transaction. A master generates a stop condition when it relinquishes mastership [Figures 7].

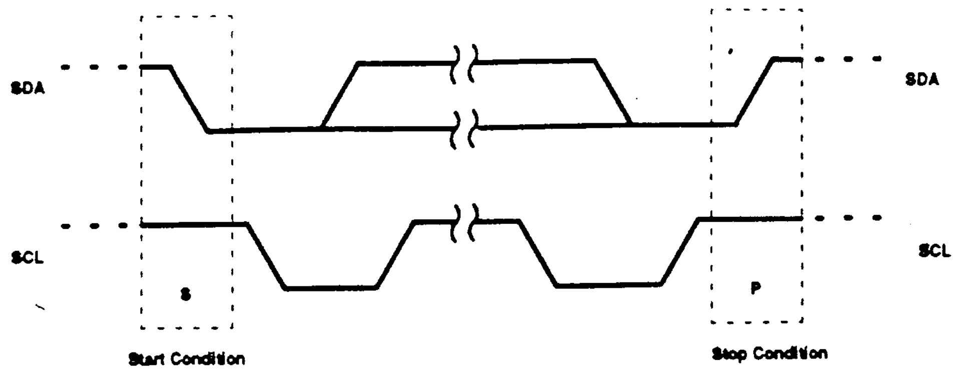


Figure 7. Start and stop conditions

2.3 Synchronization

When a bus partner wishes to assert mastership of a free bus, it generates a start condition by pulling the data SDA low. When SDA is low the new master begins the clock cycle, pulling the SCL low. All bus partners must be able to sense these events, and they respond by pulling all their SCL outputs low and beginning to count off their low periods. When each bus partner has reached the end of its low period, it lets its SCL output go high. Thus the SCL line will remain low for the duration of the longest low clock period among the bus partners.

When all the bus partners have reached the end of their low periods and let their SCL outputs go high, then the SCL goes high. All bus partners must be able to sense this event, and they begin counting their high period. The first bus partner to reach the end of its high period pulls the SCL low again. In this way, all the bus partners simultaneously communicating on the bus are synchronized by a clock pulse whose low is as long as the longest of the low periods and whose high is as long as the shortest of the high periods. This synchronization persists until the master relinquishes the bus by generating a stop condition. The cooperative synchronization is a mechanism by which devices with slower clocks can regulate the operating rate of the bus. However, this mechanism, called "clock stretching", is not the normal means of data stream flow control. The ACCESS.bus protocol provides another mechanism for this purpose. (See Section 3.3.)

2.4 Byte Framing and Acknowledgement

During this synchronized exchange the master/transmitter puts data on the SDA, one bit for each clock pulse. Eight successive bits comprise a byte, the most significant bit going first.

As the new master puts the the first byte on the bus, all the other bus partners participate in the synchronization. The first byte of the transaction contains the address of the intended slave/receiver of the transaction. Each non-master can check the address bits as they appear and cease participating in the synchronization as soon as the address bit on SDA fails to match the corresponding bit of its own address.

At the end of the first byte, the master/transmitter lets its data output go high for the next clock pulse and the slave/receiver whose address matches the transmitted address is obliged to acknowledge receipt of the byte by pulling the SDA low for this pulse. This 1-bit Ack continues after each byte of the bus transaction; the master lets its SDA output go high and the receiver must pull the SCL low. Failure of the receiver to acknowledge a byte is an exception condition, which requires the master to terminate the transaction.

2.5 Addressing

The slave/receiver of the bus transaction is determined by the address contained in the first byte. I^2C uses the seven high-order bits of the first byte for addressing, and bit 0 to indicate whether the master is transmitting or receiving data. In ACCESS.bus, the master is always the transmitter, so bit 0 of the first byte of a transaction is always 0. Of the 128 7-bit addresses, the I^2C protocol reserves various ranges for specific kinds of integrated circuit devices. The ACCESS.bus uses only 16 addresses for general micro controllers. Thus, considered as 8-bit bytes, the valid ACCESS.bus addresses are the even numbers between 50 hexidecimal and 6E hexidecimal, inclusive (80 and 110 decimal, inclusive).

The host computer address is always 50h. In the configuration process to be described below, each peripheral device is assigned a unique address from the set of even numbers between 52h and 6Ch. 6Eh is used as a default address for devices before they have been assigned a unique address.

A device address designates the device controller, or Interface Part, of a peripheral device. When several Application Part subdevices share a controller, they are distinguished by a subaddress field in the third byte of the message, discussed in Section 2.7.

2.6 Arbitration

What happens when two devices simultaneously assert mastership? While putting data on the SDA, each transmitting master is, of course, independently sensing the state of SDA. Whenever a contending master detects that the state of the SDA is different from the data value it is putting out during a clock high, the contending master backs off, and waits for the stop condition before trying again. Thus, two contending masters will both put data on the bus as long as they are putting out the same data. The first bit where they differ will cause the contender that put out a 1 to back off.

Thus, contending masters trying to send to different bus addresses will resolve the contention by the end of the first byte of the bus transaction. In the ACCESS.bus Base Protocol, the second byte of a transaction is the address of the transmitting master. Thus, as long as bus addresses are unique, the mastership of the bus will be resolved by the end of the second byte of the transaction. However, if two devices have the same address and are trying to send identical messages to a common address, then they will both send the entire message in unison. This situation can happen only during the configuration process before devices have all been assigned unique addresses; it is discussed further in Section 2.9 below.

Note that this arbitration mechanism never causes lost data or wasted transmissions, since the addresses of the receiver and transmitter are necessary overhead, in any case, for any sensible bus protocol. Note that bus priorities during arbitration are fixed by the device addresses, first by the address of the receiver, and then, for messages addressed to a common receiver, by the address of the transmitter. Lower addresses have priority over higher addresses. Lockouts of devices with high addresses are prevented by a rule of the Base Protocol that requires partners to wait a minimum time after relinquishing mastership before asserting it again.

2.7 Message Format (Preliminary)

An ACCESS.bus message comprises one 1^2C bus transaction. It consists of a string of bytes sent by a master/transmitter, each byte acknowledged by a one bit SCL-low Ack from the slave/receiver. The entire transaction is delimited by start and stop conditions generated by the master.

The first byte in the message is the receiver's unique address, as described above in Section 2.5. The second byte contains the transmitter's unique address. The third byte of an ACCESS.bus message comprises three fields. The low order 5 bits provide a byte count for the body of the message, which follows the third byte. A value of zero in this field specifies 32 bytes. Thus, a message body can have 1 to 32 bytes. The message body is followed by a checksum byte, for error control. The checksum is the bitwise XOR of all the preceding bytes of the message.

Bits 5 and 6 of the third byte of the message comprise a subaddress, which can distinguish among up to four Application Part subdevices sharing a common microcontroller, and so a common bus address.

The high order bit of the third byte is a Protocol Flag P to distinguish between data stream messages (P=0) and control/status messages (P=1). The data stream messages carry the application information being exchanged between the device and the host. The control/status messages are used to manage the ACCESS.bus protocol [Figure 8].

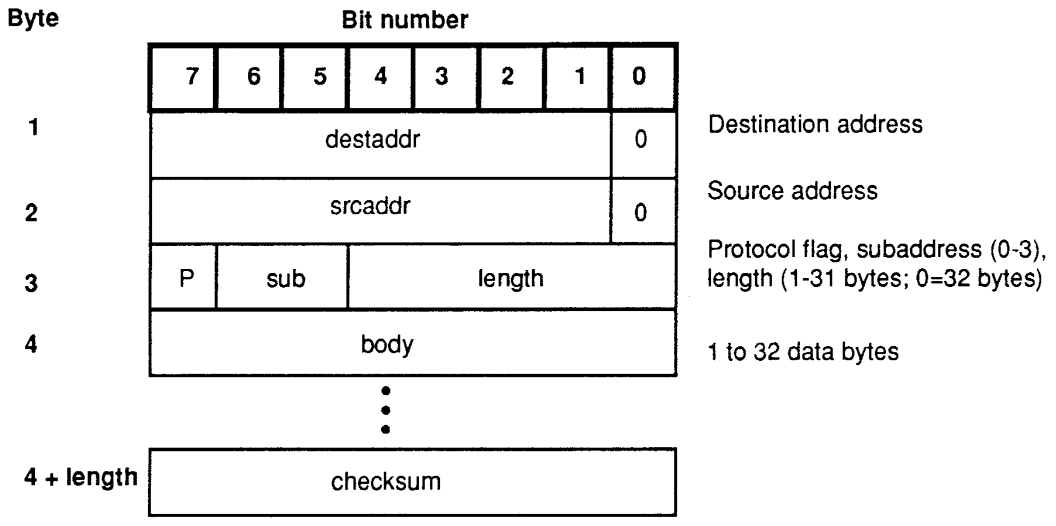


Figure 8. ACCESS. bus little Endian bit ordering

2.8 Control/Status Messages (Preliminary)

The ACCESS.bus Base Protocol defines a number of control/status messages that pertain to the Interface Parts of devices. These control/status messages are used for the configuration process, in which devices are assigned unique bus addresses and connected with the appropriate drivers in the host. The configuration process is described in Section 2.9 to 2.11. In a control/status message, the message type is indicated by an operation code contained in the first byte of the message body. The various Interface Part control/status messages are as follows:

Computer -> Device	
Reset()	Force device to power-up state, default address
IdRequest()	Ask device for its identification string
AssignAddress (ID,addr)	Tell device with matching ID its unique bus address
CapRequest(offset)	Ask device to send part of its capabilities
Device -> Computer	
Attention (status)	Inform computer of device presence and result of power-up reset test
IdReply(ID)	Send device ID string to computer
CapReply(offset,str)	Send part of capability information to computer

The Base Protocol also defines three Application Part control/status messages, applicable to all device types:

AppHardSig	a Provision for a subdevice to generate an interrupt of the host system
AppTest	A message from the host to a peripheral device commanding it to test the Application Part subdevice
AppTestReply	A message by which a device replies to an AppTest command.

In addition, the Application Protocols define further control/status messages that are specific to particular device types. Some of these are discussed in Section 3 on the predefined device types.

2.9 Configuration

The ACCESS.bus features auto-address and hot-plugging. These features are supported by the ACCESS.bus configuration and process, which uses the seven types of control/status messages. Configuration consists of assigning unique bus addresses to the attached devices and connecting them with the appropriate drivers to provide host-resident application programs with access to the devices. Configuration occurs when the device is powered-up or when it receives a Reset message. When the system is powered up, so are devices attached to the ACCESS.bus. Otherwise, devices are powered-up when they are hot-plugged into the bus. A device may have a power source other than the +12 V power line of the ACCESS.bus, but it must also be able to sense this line voltage and enter the power-up state when attached to the ACCESS.bus power source. When the system completes its boot up, the host sends a Reset message to all 14 legal device addresses to put all devices in the power-up state**.**

Usually, when a device is powered up, it performs its specific self-testing. At the conclusion of self-test the device must assume the default address 6Eh and send an Attention message to announce its presence to the host. This message contains a single status byte to inform the host of the results of the power-up self-testing; zero indicates normal results and non-zero values indicate exception conditions that are specific to the device type.

On receiving an Attention message, the host sends an IDRequest message to the default address. Each device at this address replies with an IDReply message containing a unique 28-byte ID string described in the next section. The host is then able to assign unique ACCESS.bus addresses to each of the devices at the default address, by sending AssignAddress messages to the default address. Each AssignAddress message contains in its body the assigned address and the unique ID string of the corresponding device.

2.10 Device Identifiers (Preliminary)

During configuration, before address assignment, each device can be identified by a 28-byte unique ID string, which may be partly or entirely encoded in the device ROM. The first 24 bytes of the ID string are understood as ASCII-encoded information characterizing the device type:

- Model revision designation (8 bytes) e.g., "REV X0.2"
- Vendor name (8 bytes) e.g., "DEC"
- Module name (8 bytes) e.g., "LK401".

The first 24 bytes characterize the device's firmware and are encoded in the device ROM.

The remaining 4 bytes of the device ID string are understood as a 32-bit two's complement integer that uniquely identifies the device among devices of the same type. This integer may be provided as a unique serial number contained in the device ROM. Or, in the absence of such a serial number, interactive devices may use a random or arbitrarily determined number for this part of the ID string. As an aid to the host software, the Base Protocol specifies that, in the IDReply message, unique serial numbers be sent as positive integers and randomly generated numbers be sent as negative integers, in the two's complement sense.

In the random number case, it is possible that different devices of the same type may come up with the same 32-bit discriminator. In this case, the different devices will be assigned the same bus address, an undesirable situation. The ACCESS.bus specification suggests a guideline to help avoid identical random identifiers: use the number of cycles of the device's own clock between power up and the time the IDRequest message is received. Then natural dispersion of the frequencies of these oscillators is likely to provide unique numbers.

The Base Protocol includes a provision to ensure against the unlikely circumstance that different interactive devices of the same type and without unique serial numbers will generate the same random number. Namely, each such device must send a Reset message to its own assigned address. This selfaddressed Reset is sent only once between power-ups or external Resets, just before the device sends the first message instigated by a user action. Of course, the transmitting device will ignore the self-addressed Reset, but other devices possibly at the same address will be reset and will go through configuration again. The Base Protocol specifies that a device using a random number in its ID string shall change that random number after receiving a Reset message. In this way, all interactive devices are guaranteed assignment of unique addresses before sending their first data-carrying messages.

If several noninteractive devices of the same type are to be attached to a single ACCESS.bus, then they must have hardwired unique serial numbers.

During normal operation, the host periodically checks the configuration by sending IDRequest messages to inactive devices. The host also sends IDRequest messages to all assigned addresses whenever it receives an Attention message from a device seeking configuration. The purpose of these IDRequest messages is to verify the current state of the ACCESS.bus - what devices are still connected and which devices are no longer present.

2.11 Device Capabilities Information (Preliminary)

In order that a device be accessible to application programs running on the host, it must be connected to an appropriate software driver. Establishing this association is the last phase of configuration.

The appropriate driver will depend on the device type. There may be further parameters that characterize the device and which affect the choice of driver, or which at least must be furnished as arguments to the selected driver. Moreover, the application program may also need to be informed of these device parameters. The device Capabilities Information feature of the ACCESS.bus protocol allows a measure of device independence in the selection of drivers and provides for informing the host software of the device characteristics.

Device Capabilities Information is an explicit statement of a device's functional characteristics that are only implicit in the device type designation contained in the ID string, or that may even vary among individual devices of a given type. For example, the Capabilities Information about a keyboard might include the national alphabet used, or the capabilities information about a locator might include its resolution or units.

The Capabilities Information for each device is contained in a single human-readable ASCII-encoded text string stored on the device ROM. The ACCESS.bus Base Protocol defines a simple and compact grammar for building the capabilities string. This grammar provides for hierarchical organization of the Capabilities Information and for the inclusion of subdevice-specific information.

The semantics of the Capabilities Information is carried by keywords. The Base Protocol defines some keywords that can apply to all sorts of devices. Then each Application Protocol will define further keywords that are meaningful only for certain types of devices. To date, the ACCESS.bus Application Protocols define semantics for the Capabilities Information for generic keyboards, locators, and general text devices. The grammar allows for easy extension of the Capabilities Information specification.

An example of a simple mouse Capabilities string might be as follows:

```
(
 protocol (locator)
type (mouse)
usage (main)
buttons ( 1(L) 2 (M) 4 (R) )
dim(2)
rel
res(200 inch)
range(-127 127)
d0(dname(X))
d 1(dname (Y)
)
```

This would specify that the device is a mouse using the standard locator protocol; that it is a primary device to be used during system startup [usage(main)]; that it has three buttons designated left, middle and right coded with the respective mask values 1, 2, and 4; that it has two degrees of freedom, designated "X" and "Y", using inches as units with 200 counts per inch; that it reports relative values (displacement since last report) in the range -127 to 127 counts.

After assigning a unique address to a device, the host sends it a CapRequest message to command it to send its Capabilities Information in a CapReply message. Of course, the device Capabilities string may well exceed the 31-byte capacity of the message body of a control/status message, so several exchanges of CapRequest/CapReply messages are needed. The two-byte 'offset' argument of each CapRequest specifies where in the Capabilities Information string the fragment should start. This 'offset' value is repeated in the CapReply message, to be used as a check by the host in reassembling the Capabilities string. Thus each CapReply message can contain up to 29 characters of Capabilities Information.

3. Application Device Types

The initial ACCESS.bus specification defines Application Protocols for three kinds of devices: keyboards, locators, and text devices. An important advantage of developing devices that conform to these defined protocols is the availability of pre-existing software to implement them, in particular, the drivers in the operating software of the host system through which application programs gain access to the devices.

3.1 Keyboard Devices

The keyboard device protocol is designed to implement the full functionality of a user interface style like that of the standard Digital LK501 keyboard, while minimizing the amount of state information that must be modeled in both the host and the keyboard, and minimizing also the memory required in the keyboard.

The protocol allows for up to 255 keys. The Capabilities Information semantics provide for specifying the existence of special keypads - numeric, arrows, editing, function keys - the national alphabet, and the parameters of feedback features such as key clicks, bells, and LED's.

There is just one device data stream message, a keyboard state report, which the keyboard sends on every key transition (key up or key down) and which reports the full state of the keyboard, as a list of all the keys that are down (assumed to be no more than ten).

Device-specific control/status messages from the host include an AppPoll poll command asking the keyboard to report its present state, and commands to produce clicks, bells, and to control the LEDs.

More detailed information can be found in the *ACCESS. bus Keyboard Device* Protocol *Specification.*

3.2 Locator Devices

The *ACCESS. bus Locator Device Specification* provides for a device that has up to 15 degrees of freedom (with 16-bit precision) and up to 16 binary keys or buttons. Thus, in addition to such conventional pointer/locator devices as mouse, tablet, trackball, the ACCESS.bus locator protocol is suitable for valuator sets, such as dial boxes, and function key boxes with up to 16 function keys.

The locator capabilities information provides for specifying the number of switches and their designations (for example, "left", "right", "middle", etc.), whether the locator values are relative (like a mouse) or absolute (like a tablet or dial box), the resolution (counts per unit, and units), the dynamic range, and the names of the locator axes (for example, "x", "y", etc.).

The first 2-byte word of the event report message body contains a mask giving the state of the switches; the remaining words contain the value of each of the locator axes, either the absolute values or the change since the previous report in the case of relative devices. The locator report message is sent either on a regular sampling interval or on receipt of an AppPoll message from the host.

The locator-specific Application Protocol control/status messages are from the host to the device:

AppPoll	Requests the device to report its state
AppSamplingInterval	Sets the device sampling interval or instructs the device to report only when polled

More detailed information can be found in the *ACCESS. bus Locator Device Protocol Specification.*

3.3 Text Devices

The ACCESS.bus text device protocol is a simple model for handling devices that transmit and/or receive messages consisting of strings of characters in some fixed character set. The text device data stream message body simply contains 1 to 32 bytes of text, and has no other assumed structure. The text device protocol also includes a provision for high-level flow control.

The defined capabilities information includes the device type (for example, "barcode", "magcode", "smartcard", "port", "printer"), whether the device is for input, output, or both, character set ("ASCII" or "ISO Latin-1"), and the flow control ("input" or "output").

Flow control provides for the case that a data stream receiver cannot handle data as fast as the transmitter can send it, as when a printer with limited buffer capacity receives voluminous data from the host. Flow control uses two textdevice-specific Application Protocol control/status messages from the data stream receiver to the data stream sender:

AppHold	Tells the sender to hold transmission of data stream messages until requested to resume
AppResume	Tells the sender it mayu resume transmitting up to ‘count’ characters; if ‘count’ is zero, then the sender may continue transmitting until receiving an AppHold.

More detailed information can be found in the *ACCESS.bus Text Device Protocol Specification*.

4. Timing Rules

To ensure good interactive response and to ensure that all devices will have access to the bus, the ACCESS.bus specifies some rules on transaction timing. Further, specific timeouts are needed to avoid hanging up the interconnect when devices fail or are removed.

4.1 Transaction Timing Rules

The ACCESS.bus is designed primarily for interactive devices. A basic objective of the definition of the ACCESS.bus specification is that every interactive device should be able to update the host on its state at least once in every display video frame time. To help meet this criterion, the Base Protocol imposes some rules on the timing of a device's interaction with the bus.

4.2 Response Timeouts

In order that a dead or unplugged device will not hang up the system indefinitely, there must be time limits for responding to commands that require a response. The ACCESS.bus protocol specifies that devices shall complete the Reset command within 250 ms. Further, a device shall respond to any command requiring a response within 40 ms, or, in the case of commands that can be answered by several devices, within 40 ms after the last device to respond.

5. Micro controllers

Participation in the several levels of the ACCESS.bus protocol requires intelligence at the devices as well as at the host. Philips produces a number of 8-bit CMOS microcontrollers, of the 80C51 architecture family, featuring some level of 1^2C interface support on-chip. The use of these low-cost, low-pin count, low power-draw components greatly facilitates the design of ACCESS.bus devices. The following are Philips part numbers for the components discussed in this section:

8XCL410
8XC528
8XC552
8XC652
8XC654
8XC751
|8XC752

where X is 0, 3, or 7.

Each of these microcontrollers includes an 8-bit CPU with a clock oscillator, interrupt structure, and a number of I/O lines. All come in versions with some ROM program memory and a small amount of RAM data memory. In most cases the ROM may be either mask programmable (X=3) or EPROM (X=7). Most of these parts also support up to 64K each of external program and data memories, but accessing external memory requires using some I/O lines as address/data lines. Some of these parts have ROMless versions (X=0), and so would have to make use of external data memory.

Each of these parts contains at least one 16-bit timer/counter, needed for participation in the 1^2C bus synchronization; some contain as many as three timer/counters, plus a watchdog timer.

Some of these parts have analog input pins and on-board A/D convertors as well as pulse-width-modulation (PWM) outputs, both usable for communication with the application-specific circuitry of an ACCESS.bus peripheral device.

All of these components offer open-drain output pins that can be connected to the SDA and SCL lines of the 1^2C bus. In addition they may contain special registers for controlling the 1^2C interface functions. Some of these components offer only bit-level I^2C interface support in hardware. For example, in the 8XC528 the hardware performs the following functions:

- Generates an interrupt on receiving an 1^2C start condition
- Recognizes a stop condition and indicates bus-busy or bus-free status
- Latches a received serial bit
- Generates serial clock pulses
- Detects bit-level arbitration loss

But some of the components, such as the 8XCL410, offer byte-level support for the 1^2C protocol, including byte framing with Ack generation and address comparison and detection. The choice of any of these components as the peripheral component considerably simplifies the development of device firmware and also enables better run-time performance.

The following table summarizes some of the relevant parameters and features of each of these microcontrollers, by part number. For detailed descriptions of these components, as well as the complete definition of the 8051 architecture, see the *Data Handbook for 80C51 and Derivative Microcontrollers,* available from Philips. This manual also contains a complete specification of the I^2C protocol, and provides application notes on 1^2C implementations, including sample

	ROM	RAM	16-bit timers	I/O	Other features/limitations
8XC751	2K	64	1	19	No ROM
8XC752	2K	64	1	21	5 x 8-bit A/D, 1 PWM, no ROM
8XCL410	4K	128	2	32	Byte-level 1^2C, no EPROM
8XC552	8K	256	3+watchdog	48	8 x 10-bit A/D, 2 PWM
8XC652	8K	256	2	32	
8XC654	16K	256	2	32	No ROM
8XC528	32K	512	3+watchdog	32	

source code for some of the necessary firmware.

I^2C Bus Microcontroller Parameters and Features:

In addition to these intelligent microcontrollers, Philips offers the PCD8584 I^2C bus controller. This is a non-intelligent component that serves to convert between byte-parallel data and the I^2C serial data. It may be used to provide a general microprocessor with an I^2C interface.

6. Software Architecture and Development

An ACCESS.bus peripheral requires software at both ends of the bus transaction for managing all levels of the peripheral interaction: the I^2C interface, the ACCESS.bus Base Protocol, and the ACCESS.bus Application Protocol. Further, the peripheral device requires software to support communication between the device microcontroller and the application-specific I/O transducer circuitry. Finally, the host system operating software must provide interfaces by which application programs can access both the ACCESS.bus devices and the ACCESS.bus itself. An important advantage of the ACCESS.bus approach is that the lower levels of the interaction are common to diverse device types, so they can be supported by the same or similar software modules.

6.1 Device Firmware Development

The microcontroller in the device provides the intelligence for managing the device's participation in all the levels of the ACCESS.bus protocol. Use of the components with hardware 1^2C interface functionality, described in Section 5, can simplify the development of the lowest level of this software. Moreover, because they concern only the bus communication methods that are common to all sorts of peripheral devices, the I^2C interface and the ACCESS.bus Base Protocol may well be implemented by reusing software previously developed for some of these components. And devices conforming to the semantics of the predefined standard Application Protocols may also benefit from the availability of some off-the-shelf code at the top level.

Of course, each device vendor will have to develop substantial firmware specific to his or her device. Philips and several third party vendors offer a range of tools to support firmware development for the standard components of the 80C51 family. These tools include cross assemblers and cross compilers for C and PL/M, in-circuit emulators with symbolic debugging and real-time trace support, and EPROM programming equipment. Generally, these software and hardware tools are for use with IBM PC-compatibles as the development platforms

6.2 Host Software Architecture

Vendors of host systems supporting ACCESS.bus will have to supply drivers and other kernel modules to provide access to the ACCESS.bus port, both for application program clients and for other system software, such as the interactive 1/0 handlers of the window system. In this context, an X Window System server is itself a client of the ACCESS.bus services [Figure 9].

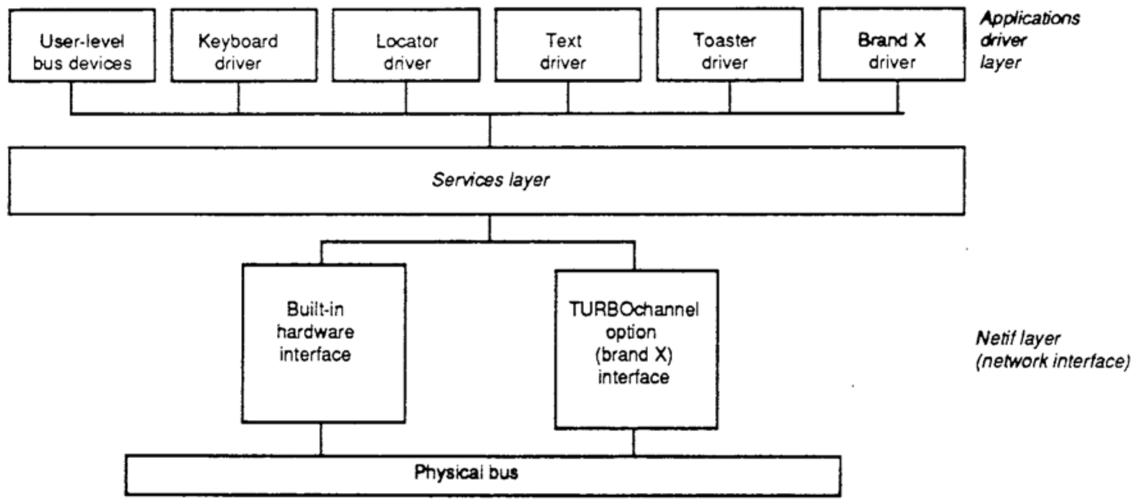

User-level bus devices
Keyboard driver
Locator driver
Text driver
Toaster driver
Brand X driver
Applications driver layer
Services layer
Built-in hardware interface
TURBOchannel option (brand X) interface
Netif layer (network interface)
Physical bus

Figure 9. Host software architecture

At the lower levels, the I^2C level and the ACCESS.bus Base Protocol level, the host is much like a peripheral device; its ACCESS.bus port is based on one of the microcontrollers discussed in Section 5, and its low-level software requirements are similar. The low-level host software is responsible for framing and deframing the ACCESS.bus messages coming from and going to the higher levels.

The heart of the ACCESS.bus access will be in an ACCESS.bus Services layer, which manages the multiplexing of the ACCESS.bus interface resource among the various clients making use of it. This level is responsible for the semantics of the ACCESS.bus messages exchanged with the interface level. The software at this level takes care of the host's responsibilities in the ACCESS.bus configuration process. It performs the mapping between bus addresses and the device handles by which devices are known to applications. It caches device Capabilities Information and makes it available to clients.

At the next level, there are Application Protocol drivers for the predefined device types, providing the bridge between the ACCESS.bus Services layer and the window system services by which applications normally access the common interactive devices. This level should also provide a general-purpose interface by which user-process software may access the bus. This interface may be used for debugging purpose, for access to interactive peripherals independent of the window system, and to accommodate unusual device types.

Of course, all these software levels will generally be specific to the host operating system or window system. Digital will be providing timely ACCESS.bus support in each of its operating systems as hardware interfaces and ACCESS.bus devices are introduced as Digital products. It is expected that standard support for ACCESS.bus will be available for the standard ACE operating systems. Therefore, developers of ACCESS.bus peripherals that conform to the defined standard device type specifications can expect that their products will automatically be compatible with any of the target systems, even in the hot-pluggable sense. Developers of unusual ACCESS.bus peripheral devices will have also to develop the appropriate driver software and to deal with the problems of having it installed on customers' computers.

7. Development Support

Both Digital and Philips/Signetics offer technical support and assistance to developers of ACCESS.bus devices and host systems.

7.1 Digital's TRI/ADD Program

Digital's TRI/ADD Program provides support to third party vendors of hardware products compatible with all of Digital's open buses and interconnects, including ACCESS.bus. Technical services available to TRI/ADD members include free consultation on hardware and software issues, necessary documentation and updates, technical development seminars, and newsletters. The products of TRI/ADD members are listed in Digital's hardware catalogs and can enjoy further marketing support. Further, TRI/ADD products can be supported by Digital's world-wide field service organization. The TRI/ADD Program will also be a source of information on the activities of the planned ACCESS.bus standards committee.

For further information on the TRI/ADD Program and its support of the ACCESS.bus, contact

TRI/ADD Program
Digital Equipment Corporation
100 Hamilton Avenue
Palo Alto, CA 94301

Using the numbers below, you can contact the program office directly:

Australia	0014.800.125.388
France	05.90.2874
Germany	0130.81.1974
Italy	1678.19087
Japan	0031.12.2363
U.K.	0800.89.2610
U.S./Canada 1.800.678.OPEN	
Other countries	[1] 415.853.6531
FAX	(415)-853-0155
Email	triadd@decwrl.dec.com

7.2 Philips/Signeties Support

An ACCESS.bus Developer's Kit is available from Philips/Signetics. This kit includes the complete specifications for the ACCESS.bus Base Protocol and predefined Application Protocols, as well as the Philips Data Handbook containing the detailed specification of the I^2C bus, characteristics of the available integrated circuits which support it, application notes, and sample firmware code. The Data Handbooks also contain listings of development systems and third-party products supporting microcontroller firmware development, mentioned in Section 6.1.

Beside the 80C51-family microcontrollers, Philips and other manufacturers offer over 100 different components with built-in I^{2}C support: memories, display controllers, data converters, clock/calendars, voice synthesizers, video processors, and others.

For technical questions on I^{2}C call your local Phihps/Signetics field application engineer, or contact the Headquarters Application Group:

Greg Goodhue	[1] 408.991.5410
Bill Houghton	[11408.991.3560
Carol Jacobson	[11408.991.3682