BUSMASTER -- attaching IBM-PC or Multibus peripherals to an 1108 with Extended Processor Option (CPE)

Initially released: 25 Sept 1984
Michael Herring
Edited: 22 Oct 1984

IBM PC- or Multibus- compatible peripheral devices (not both at once!) can be attached to an 1108 which has the Extended Processor Option (CPE), using Xerox's BusMaster interface option.

This document describes the functionality of the BusMaster hardware, and the BUSMASTER software package for controlling it. It is divided into sections:

this introductory section
single-byte transfers - the BUS functions
microcoded block transfers - the BUSBLT functions
dma - overview of direct memory access
dma - detailed discussion of the dma process
dma - register model of the BusMaster dma controller
dma - the BUSDMA functions
dma - summary of simple use
dma - technical notes

(This document and the accompanying BUSMASTER software package are somewhat preliminary: While they have been used with IBM PC-compatible peripherals, the Multibus alternative has not been fully integrated at this writing. Thus some changes can be expected before the BusMaster interface option is released. In particular, more obscure details may change, the more technical documentation may become more exact, the documentation will become less PC oriented, and (perhaps most importantly) well-packaged diagnostic software will be included. Also hopefully global initialization will be better understood and better packaged, and the problem of accessing nonexistent memory on the external bus will be properly addressed.

Note in particular that much of this documentation is written as though only IBM PC peripherals were supported, which is false.

IBM PC, IPB PC-XT, Intel 8237-5A, and Data Translation DT2801 are proprietary names.)

Additional hardware is required between the BusMaster and the peripheral devices: an IBM PC or PC-XT expansion chassis to mount the peripherals' controller cards in, and in some cases a PC memory card, mounted in the expansion chassis. The BusMaster connects the PC expansion chassis (the "external bus") to the BusExtender high-speed parallel port of the 1108.

The BUSMASTER library package makes use of the BUSEXTENDER library package.

In terms of the hardware environment, PC peripherals fit into this augmented 1108 system in just the same way they would into an IBM PC. Most programming of the peripherals is done in just the same way it would be done in BASIC on an IBM PC: "peeks and pokes" -- that is, explicitly programmed transfers of individual bytes from and to individual i/o and memory addresses. There are two restrictions affecting more advanced i/o programming techniques:

1) Interrupts: Interrupts are not as yet supported. More precisely, while the hardware does support interrupts, exactly as on an IBM PC, the 1108 microcode does not as yet support interrupts.

2) Direct memory access: PC peripherals cannot perform direct memory access (dma) to the internal memory of the 1108. However, the peripheral can perform dma to/from PC memory mounted in the PC expansion chassis, and the 1108 can *simultaneously* access that memory, either with peeks & pokes or with microcoded high-speed block transfer instructions.

Our experience is that example i/o programs in BASIC, not involving interrupts or dma, can be translated immediately to Interlisp-D without subtlety.

We have also implemented several applications that use dma techniques, with two different peripheral devices, the Data Translation DT2801 Analog & Digital I/O System and a Tecmar-like 640x400x4-bit color board. One of our applications has the data acquisition board sampling acoustic data at

10KHz, while the 1108 processes the data (including FFTs) and graphs the results, continuously in real time. This application uses quite sophisticated dma techniques: the data acquisition board dmas continuously into a circular buffer in memory in the expansion chassis, while the program in the 1108 continuously reads & processes data from the circular buffer. This is possible because (1) the PC's dma controller (which is functionally duplicated in the BusMaster) can be programmed to wrap dma around a circular buffer continuously without software intervention, (2) the dma pointers can be read by the software program, and (3) data transfer from the PC memory to the 1108 does not interfere with dma from the peripheral device. This dma technique took only a couple of days to design & implement using the BUSMASTER software.

Our interface functions for the Data Translation board are available as the LISP Library Package PCDAC; it is one example of how to use the BUSMASTER package.

Single-byte transfers to/from the external bus -- the BUS functions

Note that whenever you power-up or BUS.RESET, you then have to call BUSDMA.INIT at least once, to initialize the memory refresh apparatus, before the memory on the external bus can be expected to hold data.

(BUS.RESET) -- acts like a power-up: resets the BusMaster and asserts the Reset signal on the external bus, thereby causing every device there to reset itself as at power-up. BUSDMA.INIT will then have to be called again if the dma controller or memory refresh apparatus is needed.

(BUS.INPUT *i/oaddress*) => *8bitvalue* -- input a byte from an external-bus i/o address, returning it as a small nonnegative integer.

(BUS.OUTPUT *i/oaddress 8bitvalue*) -- output a byte (the least significant 8 bits of the integer argument) to an external-bus i/o address

(BUS.READ *memoryaddress*) => *8bitvalue* -- read a byte from external-bus memory, returning it as a small nonnegative integer.

(BUS.READHL *memaddrhi memaddrlo*) => *8bitvalue* -- read a byte from external-bus memory, returning it as a small nonnegative integer. *memaddrlo* is the less significant 16 bits of the memory address; *memaddrhi* is the more significant bits of the address.

(BUS.WRITE *memoryaddress 8bitvalue*) -- write a byte (the least significant 8 bits of the integer argument) to external-bus memory

(BUS.WRITEHL *memaddrhi memaddrlo 8bitvalue*) -- write a byte (the least significant 8 bits of the integer argument) to external-bus memory. *memaddrlo* is the less significant 16 bits of the memory address; *memaddrhi* is the more significant bits of the address.

Microcoded block tranfers to/from the external bus -- the BUSBLT functions

These functions transfer data between an array in Interlisp-D virtual memory and a consecutive region in the memory on the external bus.

Note that BUSDMA.INIT has to be called at least once, to initialize the memory refresh apparatus, before the memory on the external bus can be expected to hold data.

(BUSBLT.BYTES *array startingindex busaddress nelements toexternalmemory?*) -- transfer every byte of *nelements* elements of the *array* in Interlisp-D memory, starting with the *startingindex*'th element, to or from consecutive byte addresses on the external bus, starting at *busaddress*. *array* must be an array of either BYTEs, WORDs (= SMALLPOSPs), or FIXPs. If the array is of WORDs or FIXPs then the more significant byte of each Interlisp-D word is transferred to the lower address on the external bus.

(BUSBLT.RIGHTBYTES *array startingindex busaddress nelements toexternalmemory?*) -- transfer the less significant byte of each word of *nelements* elements of the *array* in Interlisp-D memory, starting with the *startingindex*'th element, to or from consecutive byte addresses on the external bus, starting at *busaddress*. *array* must be an array of either WORDs (= SMALLPOSPs) or FIXPs. (Note that *array* cannot be of BYTEs.) On transfers to Interlisp-D virtual memory, the more significant byte of each word is zeroed.

(BUSBLT.NYBBLES *array startingindex busaddress nelements*) -- transfer every 4-bit nybble of *nelements* elements of the *array* in Interlisp-D memory, starting with the *startingindex*'th element, to consecutive byte addresses on the external bus, starting at *busaddress*. Each 4-bit nybble of Interlisp-D virtual memory corresponds to a byte in the external bus memory: the Interlisp-D nybble is right-aligned in the external-bus byte, with the left 4 bits of the external-bus byte unspecified. The byte resulting from the more significant nybble of each Interlisp-D word is transferred to the lower byte address on the external bus. *array* must be an array of either BYTEs, WORDs (= SMALLPOSPs) or FIXPs. NOTE that only the transfer from Interlisp-D memory to external-bus memory is implemented at this writing.

A read form of BUSBLT.NYBBLES is expected to be added later. Also BUSBLT.SWAPBYTES, which will differ from BUSBLT.BYTES only in that the more significant byte of the 1108 word will correspond to the higher address on the external bus.

Dma -- overview of direct memory access on the external bus

The BusMaster includes, in addition to peek and poke apparatus, a 3-channel DMA controller and memory refresh circuitry.

Memory refresh circuitry is necessary for dynamic RAM (such as that used in the PC) to keep its data. As in the IBM PC, the BusMaster's memory refresh circuitry is integrated with its dma controller and timer, and uses the fourth dma channel #0. Memory refresh is initiated during the global initialization of the dma controller. These operations would be done automatically on an IBM PC.

Direct memory access (dma) refers to a process whereby an i/o device can transfer data to or from main memory without direct intervention by the central processor. That is, although the central processor must help in setting up the dma operation and in tidying up after it, many bytes of data can be transferred in a single dma operation without the central processor having to be involved as each byte is transferred. This involves having dedicated hardware (in this case the BusMaster's dma controller) with memory-address and transfer-count registers and control circuitry.

Typically a single dma operation transfers a predetermined number of bytes of data from the i/o device to sequential addresses in main memory, or similarly with the data going from memory to the i/o device. The dma circuitry has to allow for the operation to terminate prematurely; the BusMaster allows the operation to be terminated prematurely either by the program or by the i/o device (though not all i/o devices are smart enough).

The BusMaster will also allow dma to take place to or from a "circular buffer" in memory. This is referred to in this document by the not-obviously-appropriate name "Autoinitialization Mode". The name comes from the fact that, in Autoinitialization Mode, the dma controller does not terminate the dma operation when the transfer count runs out, but rather re-initializes the address and counter registers to their initial values, thus "wrapping around" the "circular buffer". In this case the BusMaster will let the dma go on forever: the i/o device or the program has to terminate it.

The direct memory access discussed here takes place entirely on the external bus. There is no direct memory access possible between i/o devices on the external bus and the 1108's memory. A similar effect can be got by using dma between the i/o device and the memory on the external bus, together with block transfers between the external bus memory and the 1108's memory (discussed in their own section above).

As on the IBM PC, the dma controller and its support hardware are actually controlled via peeks and pokes. However, the BUSMASTER package includes a set of BUSDMA functions that implement a

higher-level view of the dma controller. This view of the dma controller is basically a simplified version of that in the specifications for the Intel 8237-5A, on which the system is based.

Dma -- detailed discussion of the dma process

Typical dma operations using the BusMaster are quite easy to program -- you are only dealing with one i/o device and in only one way. Even planning the use of one particular i/o device is not too bad if you're not going to be exotic. Unfortunately, i/o devices differ in major & minor ways, the BusMaster is designed to fit well with most of them, and this discussion has to enable you to plan the use of (almost) any i/o device in (almost) any way reasonable. So please be tolerant. My hope is that after you read through to the Summary section below, you should have at most a few questions, which can likely be answered by skimming around. (If you are planning exotic things, you may need to read the specifications for the the Intel 8237-5A dma controller chip, on which the Busmaster is based, and the Technical Notes section below.)

The dma controller and memory refresh circuitry have to be initialized before any dma can take place and before the memory in the PC expansion chassis can hold data reliably. This is done with the BUSDMA.INIT function (though under some circumstances you may also need the BUS.RESET function discussed above). BUSDMA.INIT can be called again at any time, with the side effect of disabling (masking) all three dma channels.

The dma controller has three separate dma channels, numbered 1 to 3. (There is actually also a fourth channel, numbered 0, dedicated to the memory refresh apparatus.) Each i/o device that does dma has to know or be told its dma channel number, as does the software controlling the device. Usually a device's interface card has switches or jumpers which determine which dma channel it will use.

Only one device can be using any one dma channel at a time. Thus there can be at most three i/o devices doing dma at any time. The three dma channels are controlled separately.

We think in terms of dma "operations". A "single dma operation" is: the program gets the i/o device and its channel set up for the operation; many bytes are transferred one at a time at the instigation of the i/o device but under the control of the channel; the transfer is terminated somehow by either the channel, the i/o device, and/or the program; finally the program tidies up the i/o device & the channel. Also, during the time when individual bytes are being transferred by the i/o device and the channel, the program might intervene to temporarily "suspend" and then "resume" the dma operation. It is important to notice that the dma controller itself does not actually distinguish between "suspended" and "terminated": if the program resumes the operation then it was "suspended"; if it sets up a new operation then the old one was "terminated"!

The setup of the i/o device depends on the device. Setting up the dma channel basically involves setting up address, transfer-count, and mode registers. It would be premature to discuss the setup further here, as we have not yet motivated the issues. Rather we will discuss everything else, including the function of the channel's registers, then discuss setup again in the Summary section.

During the extended "dma operation", individual byte transfers are requested by the i/o device involved. If more than one device requests dma at the same time, the dma controller services the lower-numbered channel first. The direction of the transfer (to or from memory) has to have been set up the same in both the i/o device and the dma channel.

The memory address to/from which the transfer will take place is determined entirely by the dma channel. Normally it uses successively increasing byte addresses in memory, but it is possible to use successively decreasing addresses, and in "Autoinitialization Mode", the channel will wrap the addresses around a circular buffer (wrapping in either direction). For exactly how addresses are generated, see the discussion of the page, current-address, and base-address registers below in the Register Model section.

Dma on a particular channel can be suspended and resumed by "masking" and "unmasking" the channel. A masked channel refuses to honor any dma transfer requests. If there is a dma transfer request still pending when the channel becomes unmasked, the channel will service the request then. Obviously, data can be lost by keeping the dma channel masked too long while its i/o device is requesting transfers.

It remains to discuss how a dma operation is terminated. This is potentially rather complicated.

Dma transfers will cease under either of three conditions: the i/o device ceases to request dma transfers; the dma channel masks itself because its transfer counter runs out; or the dma channel is masked by the program.

In either case, the dma channel itself does not distinguish between "suspended" and "terminated". I will try to say what it does do:

At any given point, a dma channel is asking itself "Am I masked?". If so, that's all.

But if it is not masked, it then asks "Am I receiving a request for a dma transfer?". If not, that's all.

But if it is receiving a request for a dma transfer, then it gets the transfer done, and then increments or decrements its current-address register, and decrements its current-transfer-count register. If its current-transfer-count register does not go to zero, then that's all.

But if its current-transfer-count register does go to zero, then the channel asserts the external-bus signal named TC, and also sets its "TC bit" in the dma controller's status register. (The TC signal is also referred to as EOP, and is available for the i/o device to sense if it wants.) The channel then asks itself "Am I in Autoinitialization Mode?". If it is not in Autoinitialization mode, then it masks itself; if it is in Autoinitialization mode, then it reinitializes its current-address and current-transfer-count registers from the corresponding base registers, but does not mask itself.

In any case, the dma channel does not "terminate the dma operation" really: it may mask itself, and it may emit the TC signal and set its TC bit, and the program or the i/o device may act on these things. But all the channel cares about this, until it is acted on by the program, is that if it masked itself, then it stays masked until the program unmasks it (possibly after changing its other registers!).

The i/o device might cease to request dma transfers for any of several reasons, including: it has its own transfer count register, which has run out; it has detected an error condition and chosen to stop transfer; it has detected the TC signal from its dma channel on the external bus; it has detected some other termination condition; or it has been disabled by the program. Any particular i/o device may well not support all these choices. The dma channel cannot tell directly that the device has ceased requesting dma transfers; it simply responds to them if & as they arise. Typically the program determines that the i/o device considers the dma operation done.

The program might mask the channel temporarily, for some kind of housekeeping reason, or it may do so as part of terminating the dma operation. It can have any sort of reason for doing this. The dma channel does not care: if the program chooses to reset some of the channel's parameters while it is masked, that's fine; otherwise when the channel becomes unmasked, it will keep on from where it was. Changing a dma channel's parameters while it is unmasked is risky and to be avoided.

Direct memory access -- register model of the BusMaster dma controller

In our model of the dma controller, there are no global registers, and each channel has a set of seven registers:

Page register--

Supplies the more significant bits of the external-bus memory addresses generated by the channel. The contents of this register are simply concatenated onto the left of the contents of the current-address register when an address is generated.

Unfortunately, incrementing or decrementing a channel's current-address register does not affect its page registers. (This implies that any dma operation must take place entirely within one (64KB-aligned) 64KB page, since the current-address registers are 16 bits wide. Further, a dma buffer that seems to overlap a 64KB-page boundary will really wrap around within the 64KB-page it starts in.

The page registers are set up by the program, but cannot be read by it.

Current-address register--

Holds the less significant bits of the external-bus memory addresses generated by the channel, that is, the 16-bit address-within-page. The channel generates addresses by concatenating its page register onto the left of its current-address register.

After each byte transfer, the channel increments or decrements the current-address register by one. The direction depends on a bit in the channel's mode register. Unfortunately, this increment or decrement does not affect the page register (as discussed above).

The current-address register is set up by the program, incremented or decremented by the channel, and, in "Autoinitialization Mode", reloaded by the channel from its base-address register when its current-transfer-count register runs down. The current-address register can be read by the program, so that the program can keep its access to the external-bus memory synchronizeed with dma on the external bus.

Base-address register--

In Autoinitialization Mode, holds the starting address of the circular buffer. That is, in Autoinitialization Mode, when a channel's current-transfer-count register runs down, the channel reinitializes its current-address register from its base-address register.

The base-address register is loaded automatically whenever the program sets up the current-address register. There is no BUSDMA function for reading the base-address register.

Current-transfer-count register--

Controls the length of the dma operation (or, in Autoinitialization Mode, the length of the circular buffer). That is, after each data transfer, the channel decrements its current-transfer-count register, and *if* it goes to zero, *then* a TC signal is asserted to the external bus *and* the channel is masked (if not Autoinitialization Mode) or the current address & transfer-count registers are reinitialized from the base ones (if Autoinitialization Mode).

Note that the current-transfer-count register contains the number of byte transfers remaining to be done, as an unsigned 16-bit number. Note also that while a current-transfer-count register value of zero, viewed as *after* a dma transfer, means "done", the same zero value viewed as *before* a dma transfer, means 64K. Thus the maximum transfer (or circular buffer size) is 64K bytes.

(You may also need to know that the hardware register keeps its values one less that what we've described here, modulo 64K. The BUSDMA functions maintain the translation.)

The current-transfer-count register is set up by the program, decremented by the channel, and, in Autoinitialization Mode, reloaded by the channel from its base-transfer-count register when it runs down. The current-transfer-count register can be read by the program, so that the program can synchronize its external-bus memory accesses with the external-bus dma.

Base-transfer-count register--

In Autoinitialization Mode, holds the initial value of the current-transfer-count register. That is, in Autoinitialization Mode, when a channel's current-transfer-count register runs down, the channel reinitializes it from its base-transfer-count register.

The base-transfer-count register is loaded automatically whenever the program sets up the current-transfer-count register. There is no BUSDMA function for reading the base-transfer-count register.

Mode register--

Contains some control bits. The mode register is set up by the program, but cannot be read by it.

writememory? -- determines whether dma transfers are from the i/o device to memory, or vice versa.

autoinit? -- governs Autoinitialization Mode. If false, the current-transfer-count register should be set up with the length of the transfer, and when it runs down the channel masks itself. If true, the current-transfer-count register should be set up with the length of the circular buffer in external-bus memory, and, when it runs down, the channel reloads the current-address and current-transfer-count registers from the base-address and base-transfer-count registers, thus wrapping around the circular buffer.

decaddr? -- determines whether the current-address register is to be incremented or decremented after each byte transferred.

Mask bit--

While a channel's mask bit is set (the channel is said to be "masked"), dma on the channel is suspended in that the channel will ignore requests for dma transfers from the i/o device.

Note that if an i/o device requests a dma transfer while its channel is masked, and is still asserting that request when the channel becomes unmasked, the channel will service the request at that time. Thus data can be lost while a channel is masked only if the channel is kept masked for a significant time relative to the speed of the i/o device.

All channels' mask bits are set when the dma controller is (re)initialized, including at power-up. They can be set or cleared by the program. A channel sets its mask bit when the current-transfer-counter runs down, except in Autoinitialization Mode. The program cannot read the mask bits.

TC bit--

Whether the channel's current-transfer-count register has run down since the last time *either* the dma controller was (re)initialized, including at power-up *or* BUSDMA.READTCBIT was explicitly used to reset this bit.

Direct memory access -- the BUSDMA functions

See the Register Model section above for explanations of the arguments of these functions and of the registers referred to by them.

These functions do not check their arguments except as specifically noted.

(BUSDMA.INIT) -- (re)initialize the dma controller and memory refresh circuitry. Masks channels 1, 2 and 3, and clears their TC bits.

(BUSDMA.SETMODE *channel writememory? autoinit*? *decaddr*?) -- set the mode register for the channel.

(BUSDMA.SETPAGE *channel highbitsofaddress*) -- write to the page register for the channel. Checks that *channel* is in the range 1-3.

(BUSDMA.SETADDRESS *channel low16bitsofaddress*) -- write to both the base & current address registers for the channel. The channel must be masked when this function is called -- this is the caller's responsibility.

(BUSDMA.READADDRESS *channel*) => low16bitsofaddress -- read the current address register for the channel. The channel must be masked when this function is called -- this is the caller's responsibility.

(BUSDMA.SETCOUNTER *channel nbytes*) -- write to both the base & current transfer-count registers for the channel. Checks that *nbytes* is in the range 1-65536. The channel must be masked when this function is called -- this is the caller's responsibility.

(BUSDMA.READCOUNTER *channel*) => *nbytes* -- read the current transfer-count register for the channel. Note that the value 65536 is returned as (the functionally indistinguishable value) zero. The channel must be masked when this function is called -- this is the caller's responsibility.

(BUSDMA.MASK *channel*) -- set the mask bit for the channel, disabling dma on the channel.

(BUSDMA.UNMASK *channel*) -- clear the mask bit for the channel, enabling dma on the channel.

(BUSDMA.READTCBIT *channel clearthebit*?) => the channel's TC bit, as T or NIL. Also clears the bit if requested.

Direct memory access -- summary of simple use

Generally each channel can be dealt with separately.

Planning--

Determine what channel the device will use. Determine what conditions will terminate the dma operations, and how the device, the program, and the dma controller will find out about them. Determine where the buffer(s) will be, to the extent that they are not dynamically allocated.

If you are using circular buffering, or other exotic stuff, you'll know it by this point.

Global initialization--

Call BUS.RESET whenever you want to simulate a power-down, power-up cycle for the BusMaster, the expansion chassis, and the devices on the expansion chassis bus.

Call BUSDMA.INIT. This has to be done at least once before doing dma on the external bus or expecting the memory on the external bus to keep data. It can be called redundantly, but has the side effect of masking channels 1-3. Since there is no way to read the mask bits, if you have multiple, logically independent i/o devices using dma on the system, you will have to coordinate BUSDMA initialization in your own software. This may be fixed later.

A dma operation--

If the channel is not masked (which it is after BUSDMA.INIT), call BUSDMA.MASK. Then call BUSDMA.SETMODE, BUSDMA.SETPAGE, BUSDMA.SETADDRESS, and BUSDMA.SETCOUNTER in any order. (If your mode settings or page number for the channel are constant, you don't have to set them each time, though it's cheap.) While the channel is masked is a safe time to set up the i/o device.

Now call BUSDMA.UNMASK, then, when ready, start the i/o device. (It could be started while the channel is masked if data wouldn't be lost, but this way always works.)

The dma operation is now hopefully "in progress". You will need to test things during this time, especially since we don't have interrupts. If you are going to use BUSDMA.READADDRESS or BUSDMA.READCOUNTER, or otherwise need to suspend dma on the channel temporarily for some reason, call BUSDMA.MASK to suspend, then call BUSDMA.UNMASK to resume.

To stop the dma channel call BUSDMA.MASK.

Direct memory access -- technical notes

(1) The current- and base- transfer-count registers are kept in the hardware as the number of transfers remaining *less one*, and the channel checks after each byte transfer for the current-transfer-count having been decremented *from* zero to -1. BUSDMA.SETCOUNTER and BUSDMA.READCOUNTER perform the translation to the model described above.

(2) The i/o addresses of the devices described herein are the same as on the IBM PC. They can be discovered by inspecting the Interlisp-D source code in this BUSMASTER library package.

(3) The main point of this section is to tell technically advanced readers how the BUSDMA model of the dma process relates to the Truth about the BusMaster and the Intel 8237-5A dma controller chip. This for those who have read the specifications for the Intel 8237-5A dma controller chip or looked at the BusMaster or IBM PC schematics. A side benefit, perhaps, is to give hints for those who are thinking about using an exotic device through the BusMaster.

This section is not intended to be intelligible to a casual reader.

BusMaster functionality that is hidden by the BUSDMA functions-- the Interrupt and Interrupt Mask bits, Parity Error, and in general all the BusMaster status and control register bits (except Reset via BUS.RESET).

Note that interrupts from the external-bus i/o devices to the 1108 are supported by BusMaster, but the 1108 microcode will at present blow up if presented with such an interrupt.

See also Technical Note (1) above.

Intel 8237-5A functionality that does not apply in the hardware context of the BusMaster-- Channel 0 is dedicated to memory refresh. And therefore block memory-to-memory transfer and block memory initialization are impossible. I/o devices cannot assert the TC signal to the dma controller. Command register: the timing variations may or may not be physically possible, I don't know; DREQ & DACK sense are of course fixed. Mode register: "cascade" mode is of course not possible.

Note that the page registers are not on the 8237-5A.

Intel 8237-5A functionality that is hidden because it is either impossible in the BusMaster hardware context or only useful with very exotic hardware (I don't know which)-- Mode register: "block" mode, "demand" mode. Command register: rotating priority.

Intel 8237-5A functionality that is hidden in BUSDMA because it was judged not useful in context-- Read & write the request bits, read temporary register, clear mask register, read base-address register, read base-transfer-count register. Command register bits: disable controller. Mode register bits: the "verify" transfer type.

Intel 8237-5A functionality that is partially hidden in BUSDMA-- Master clear is imbedded in BUSDMA.INIT. Read & clear status register TC bits is presented, somewhat modified, in BUSDMA.READTCBIT. Clear byte select flipflop is hidden in the four functions BUSDMA. READ/SET ADDRESS/COUNTER.