"**IEC** disected"

This document describes the IEC-bus information gathered during the development of the 1541-III.

Contents

1 General info

The serial (IEC) bus uses a synchronous protocol. There are three lines: DATA, CLK, and -ATN. The lines are driven by a 6526 CIA on the C-64, through open-collector inverters (7406's); 1 kohm pull-up resistors are used. The 1541 and 1571 drives do the same, but they don't drive the -ATN line, and they use 6522 VIA's. The lines are also read, directly by the C-64. Address $DD00 (PA of CIA2) is used by the C-64. Bit 3 is ATN OUT, bit 4 is -CLK OUT, bit 5 is -DATA OUT, bit 6 is CLK IN, and bit 7 is DATA IN.

A device's identity is usually stored in RAM after boot up, so a drive POKE will let it change to anything in the range 0-30. The C64 kernal (unless patched) prevents 0-3 from actually going down the serial bus, that’s why these are marked with the colour “grey” in the table below.

CBM Device ID’s	
Device ID	**Device type**
0	Keyboard
1	Cassette port
2	RS232 on user port or second cassette on older units
3	Screen
4	Printer
5	Printer
6	Typically plotter device
7	Second plotter? Also used by a freeze cartridges named TurboTape, but that's only an internal definition
8	Primary Disk Drive
9	Disk Drive
10	Disk Drive Also used by some serial-to-parallel printer interface cartridges
11	Disk Drive
12	Disk Drive
13	Disk Drive
14	Disk Drive Sometimes used by some serial-to-parallel printer interface cartridges
15	Disk Drive
16-30	Unknown
31	Device #31 is reserved as a command to all devices Devices will not ignore a command to #31, since they cannot do this. Sending a TALK command to #31 means UNTALK, sending a LISTEN to #31 means UNLISTEN. This however is just a statement found on the web and has not been verified by other sources.

Regarding the signals on the bus:
0V = TRUE or PULLED DOWN
5V = FALSE or RELEASED

Regarding the data on the bus:
0V = logical 0
5V = logical 1

Bytes are sent with low bit first.
Data is valid on rising edge of clock

2 A brief history of the IEC-bus

By Jim Butterfield

As you know, the first Commodore computers used the IEEE bus to connect to peripherals such as disk and printer. I understand that these were available only from one source: Belden cables. A couple of years into Commodore's computer career, Belden went out of stock on such cables (military contract? who knows?). In any case, Commodore were in quite a fix: they made computers and disk drives, but couldn't hook 'em together! So Tramiel issued the order: "On our next computer, get off that bus. Make it a cable anyone can manufacture". And so, starting with the VIC-20 the serial bus was born. It was intended to be just as fast as the IEEE-488 it replaced.

"Technically, the idea was sound: the 6522 VIA chip has a "shift register" circuit that, if tickled with the right signals (data and clock) will cheerfully collect 8 bits of data without any help from the CPU. At that time, it would signal that it had a byte to be collected, and the processor would do so, using an automatic handshake built into the 6522 to trigger the next incoming byte. Things worked in a similar way outgoing from the computer, too. We early PET/CBM freaks knew, from playing music, that there was something wrong with the 6522's shift register: it interfered with other functions. The rule was: turn off the music before you start the tape! (The shift register was a popular sound generator). But the Commodore engineers, who only made the chip, didn't know this. Until they got into final checkout of the VIC-20.

By this time, the VIC-20 board was in manufacture. A new chip could be designed in a few months (yes, the silicon guys had application notes about the problem, long since), but it was TOO LATE!

A major software rewrite had to take place that changed the VIC-20 into a "bit-catcher" rather than a "character-catcher". It called for eight times as much work on the part of the CPU; and unlike the shift register plan, there was no timing/handshake slack time. The whole thing slowed down by a factor of approximately 5 to 6.

When the 64 came out, the problem VIA 6522 chip had been replaced by the CIA 6526. This did not have the shift registerproblem which had caused trouble on the VIC-20, and at that time it would have been possible to restore plan 1, a fast serial bus. Note that this would have called for a redesign of the 1540 disk drive, which also used a VIA. As best I can estimate - and an article in the IEEE Spectrum magazine supports this - the matter was discussed within Commodore, and it was decided that VIC-20 compatibility was more important than disk speed. Perhaps the prospect of a 1541 redesign was an important part of the decision, since current inventories needed to be taken into account. But to keep the Commodore 64 as a "bit-banger", a new problem arose.

The higher-resolution screen of the 64 (as compared to the VIC-20) could not be supported without stopping the CPU every once in a while. To be exact: Every 8 screen raster lines (each line of text), the CPU had to be put into a WAIT condition for 42 microseconds, so as to allow the next line of screen text and color nybbles to be swept into the chip.(More time would be needed if sprites were being used). But the bits were coming in on the serial bus faster than that: a bit would come in about every 20uSec! So the poor CPU, frozen for longer than that, would miss some serial bits completely! Commodore's solution was to slow down the serial bus even more. That's why the VIC-20 has a faster serial bus than the 64, even though the 64 was capable, technically, of running many times faster. Fast disk finally came into its own with the Commodore 128

3 HOW THE VIC/64 SERIAL BUS WORKS

By Jim Butterfield (Compute! July 1983, Copyrighted 1983 by J.Butterfield)

The Serial bus connects VIC or Commodore 64 to its major peripherals, especially disk and tape. The workings of this interface have been a source of bafflement to most of us. We know that it's somehow related to the IEEE-488 bus which is used on PET and CBM computers. But it has fewer wires, and it's slower. For anyone interested in interfacing details, this article will clear up the mystery.

GROUND RULES

To understand the workings of this bus, you must work through a few concepts. Later, we'll get technical for this who want it. The bus, like the IEEE, has two modes of operation: Select mode, in which the computer calls all devices and asks for a specific device to remain connected after the call ("Jones, would you stay in my office after the meeting?"); and Data mode, in which actual information is transmitted ("Jones, I've decided to give you a raise"). Select mode is invoked by the use of a special control line called "Attention," or ATN. By using Select mode, you can call in any device you choose, but you may need to do more before you transmit data. You might have several disk files in progress - writing some and reading others – and when you select the disk, device 8, you'll still need to specify which "part" of the disk you want to reach: subchannel 3, subchannel 15, or whatever. To do this, we use a "secondary address" which usually signals a subsystem within a specific device. That goes in as part of the command during Select mode. Finally, we may need to send other control information: the name of the file we wish to open, for example. That's not data; it's device setup information, so we also send it in Select mode. But the main part is: you select a device, and then you send to it or receive from it. Finally, you shut it off. All devices are connected, but only the one you have selected will listen or talk.

If you're not into volts and signals and things, the rest of this article may not do much for you. I want to talk about technical aspects of the bus. First, all the data flows over two wires; They are called the Clock line and the Data line. There are other wires used for control purposes, but the data uses only the two main ones. All wires connect to all devices. The wires don't go "one way"; any device can put a ground on a signal line, and all other devices will see it. Indeed, that's the secret of how it works: each wire serves as a common signal bus.

When no device puts a ground on a signal line, the voltage rises to almost five volts. We call this the "false" logic condition of the wire. If any device rounds the line, the voltage drops to zero; we call this the "true" condition of the line. Note that if two devices signal "true" on a line (by grounding it), the effect is exactly the same as if only one has done so: the voltage is zero and that's that. We can summarize this as an important set of logic rules:

-A line will become "true" (PULLED DOWN, or 0V) if one or more devices signal true;
-A line will become "false" (RELEASED, or 5V) only if all devices signal false.

Remember that we have several lines, but the important ones for information transmission are the Clock line and the Data line. Let's watch them work.

TRANSMISSION: STEP ZERO

Let's look at the sequence when a character is about to be transmitted. At this time, both the Clock line and the Data line are being held down to the true tate. With a test instrument, you can't tell who's doing it, but I'll tell you: the talker is holding the Clock line true, and the listener is holding the Data line true. There could be more than one listener, in which case all of the listeners are holding the Data line true. Each of the signals might be viewed as saying, "I'm here!".

STEP 1: READY TO SEND

Sooner or later, the talker will want to talk, and send a character. When it's ready to go, it releases the Clock line to false. This signal change might be translated as "I'm ready to send a character." The listener must detect this and respond, but it doesn't have to do so immediately. The listener will respond to the talker's "ready to send" signal whenever it likes; it can wait a long time. If it's a printer chugging out a line of print, or a disk drive with a formatting job in progress, it might hold back for quite a while; there's no time limit.

STEP 2: READY FOR DATA

When the listener is ready to listen, it releases the Data line to false. Suppose there is more than one listener. The Data line will go false only when all listeners have released it - in other words, when all listeners are ready to accept data. What happens next is variable. Either the talker will pull the Clock line back to true in less than 200 microseconds - usually within 60 microseconds - or it will do nothing. The listener should be watching, and if 200 microseconds pass without the Clock line going to true, it has a special task to perform: note EOI.

INTERMISSION: EOI

If the Ready for Data signal isn't acknowledged by the talker within 200 microseconds, the listener knows that the talker is trying to signal EOI. EOI, which formally stands for "End of Indicator," means "this character will be the last one." If it's a sequential disk file, don't ask for more: there will be no more. If it's a relative record, that's the end of the record. The character itself will still be coming, but the listener should note: here comes the last character. So if the listener sees the 200 microsecond time-out, it must signal "OK, I noticed the EOI" back to the talker, I does this by pulling the Data line true for at least 60 microseconds, and then releasing it. The talker will then revert to transmitting the character in the usual way; within 60 microseconds it will pull the Clock line true, and transmission will continue. At this point, the Clock line is true whether or not we have gone through the EOI sequence; we're back to a common transmission sequence.

STEP 3: SENDING THE BITS

The talker has eight bits to send. They will go out without handshake; in other words, the listener had better be there to catch them, since the talker won't wait to hear from the listener. At this point, the talker controls both lines, Clock and Data. At the beginning of the sequence, it is holding the Clock true, while the Data line is released to false. the Data line will change soon, since we'll send the data over it. The eights bits will go out from the character one at a time, with

the least significant bit going first. For example, if the character is the ASCII question mark, which is written in binary as 00011111, the ones will go out first, followed by the zeros. Now, for each bit, we set the Data line true or false according to whether the bit is one or zero. As soon as that's set, the Clock line is released to false, signalling "data ready." The talker will typically have a bit in place and be signalling ready in 70 microseconds or less. Once the talker has signalled "data ready," it will hold the two lines steady for at least 20 microseconds timing needs to be increased to

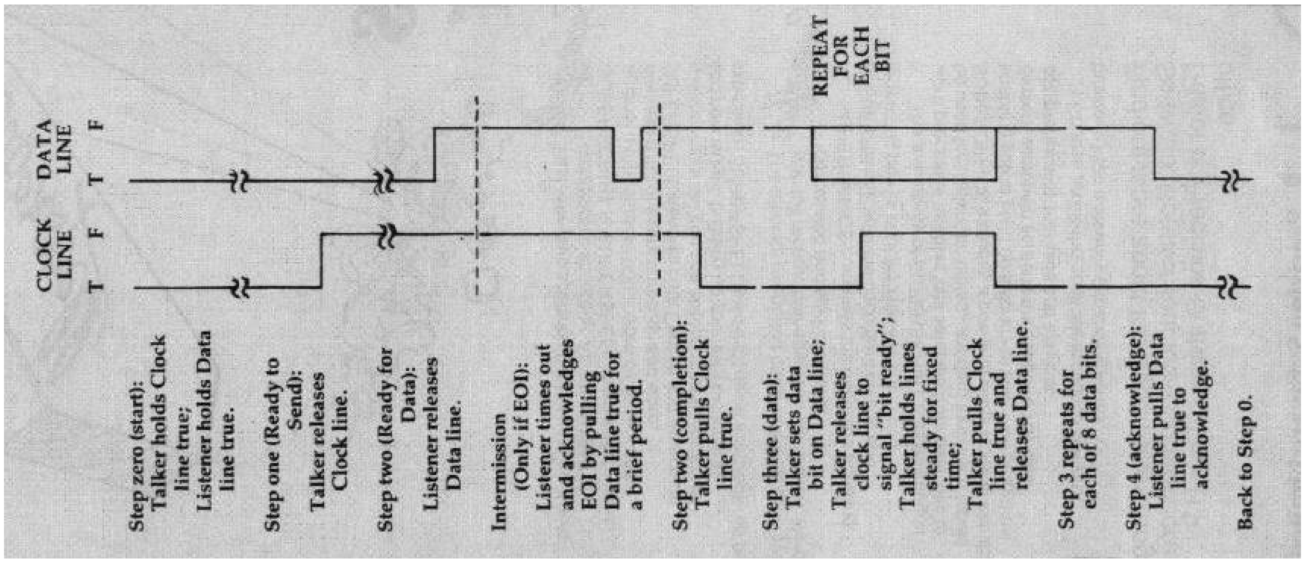
60 microseconds if the Commodore 64 is listening, since the 64's video chip may interrupt the processor for 42 microseconds at a time, and without the extra wait the 64 might completely miss a bit. The listener plays a passive role here; it sends nothing, and just watches. As soon as it sees the Clock line false, it grabs the bit from the Data line and puts it away. It then waits for the clock line to go true, in order to prepare for the next bit. When the talker figures the data has been held for a sufficient length of time, it pulls the Clock line true and releases the Data line to false. Then it starts to prepare the next bit.

STEP 4: FRAME HANDSHAKE

After the eighth bit has been sent, it's the listener's turn to acknowledge. At this moment, the Clock line is true and the Data line is false. The listener must acknowledge receiving the byte OK by pulling the Data line to true. The talker is now watching the Data line. If the listener doesn't pull the Data line true within one millisecond - one thousand microseconds - it will know that something's wrong and may alarm appropriately.

STEP 5: START OVER

We're finished, and back where we started. The talker is holding the Clock line true, and the listener is holding the Data line true. We're ready for step 1; we may send another character - unless EOI has happened. If EOI was sent or received in this last transmission, both talker and listener "let go." After a suitable pause, the Clock and Data lines are released to false and transmission stops.


DATA LINE
T F
CLOCK LINE
T F
REPEAT FOR EACH BIT
Step zero (start): Talker holds Clock line true; Listener holds Data line true.
Step one (Ready to Send): Talker releases Clock line.
Step two (Ready for Data): Listener releases Data line.
Intermission (Only if EOI): Listener times out and acknowledges EOI by pulling Data line true for a brief period.
Step two (completion): Talker pulls Clock line true.
Step three (data): Talker sets data bit on Data line; Talker releases clock line to signal "bit ready"; Talker holds lines steady for fixed time; Talker pulls Clock line true and releases Data line.
Step 3 repeats for each of 8 data bits.
Step 4 (acknowledge): Listener pulls Data line true to acknowledge.
Back to Step 0.

ATTENTION!

This is all very well for a transmission that's under way, but how do we set up talker and listener? We use an extra line that overrides everything else, called the ATN, or Attention line. Normally, the computer is the only device that will pull ATN true. When it does so, all other devices drop what they are doing and become listeners. Signals sent by the computer during an ATN period look like ordinary characters - eight bits with the usual handshake - but they are not data. They are "Talk," "Listen," "Untalk," and "Unlisten" commands telling a specific device that it will become (or cease to be) a talker or listener. The commands go to all devices, and all devices acknowledge them, but only the ones with the suitable device numbers will switch into talk and listen mode. These commands are sometimes followed by a secondary address, and after ATN is released, perhaps by a file name. An example might help give an idea of the nature of the communications

that take place. To open for writing a sequential disk file called "XX," the following sequence would be sent with ATN on:DEVICE-8-LISTEN;SECONDARY-ADDRESS-2-OPEN. When ATN switches off, the computer will be waiting as a talker, holding the Clock line true; and the disk will be the listener, holding the Data line true. That's good, because the computer has more to send, and it will transmit: X;X;comma;s;comma;W - the W will be accompanied with an EOI signal. Shortly thereafter, the computer will switch ATN back on and send DEVICE-8-UNLISTEN. The file is now open; later, the computer will want to send data there. It will transmit, with ATN on, DEVICE-8-LISTEN;SECONDARY- ADDRESS-2-DATA. Then the computer releases the ATN line and sends
its data; only the disk will receive the data, and the disk will know to put it onto the file called XX. The last character sent by the computer will also signal EOI. After the computer has sent enough data for the moment, it will pull ATN on again and send DEVICE-8-UNLISTEN. Many bursts of data may goto the file; eventually, the computer will close the file by sending (with ATN on, of course) DEVICE-8-LISTEN;SECONDARY-ADDRESS-2-CLOSE. ATN overrides everything in progress, A disk file might have lots of characters to give to the computer, but the computer wants only a
little data. It accepts the characters it wants, then switches on ATN and commands the disk to Untalk. The disk has not sent EOI, but it will disconnect as commanded. Later, when it's asked to Talk again, it will send more characters.

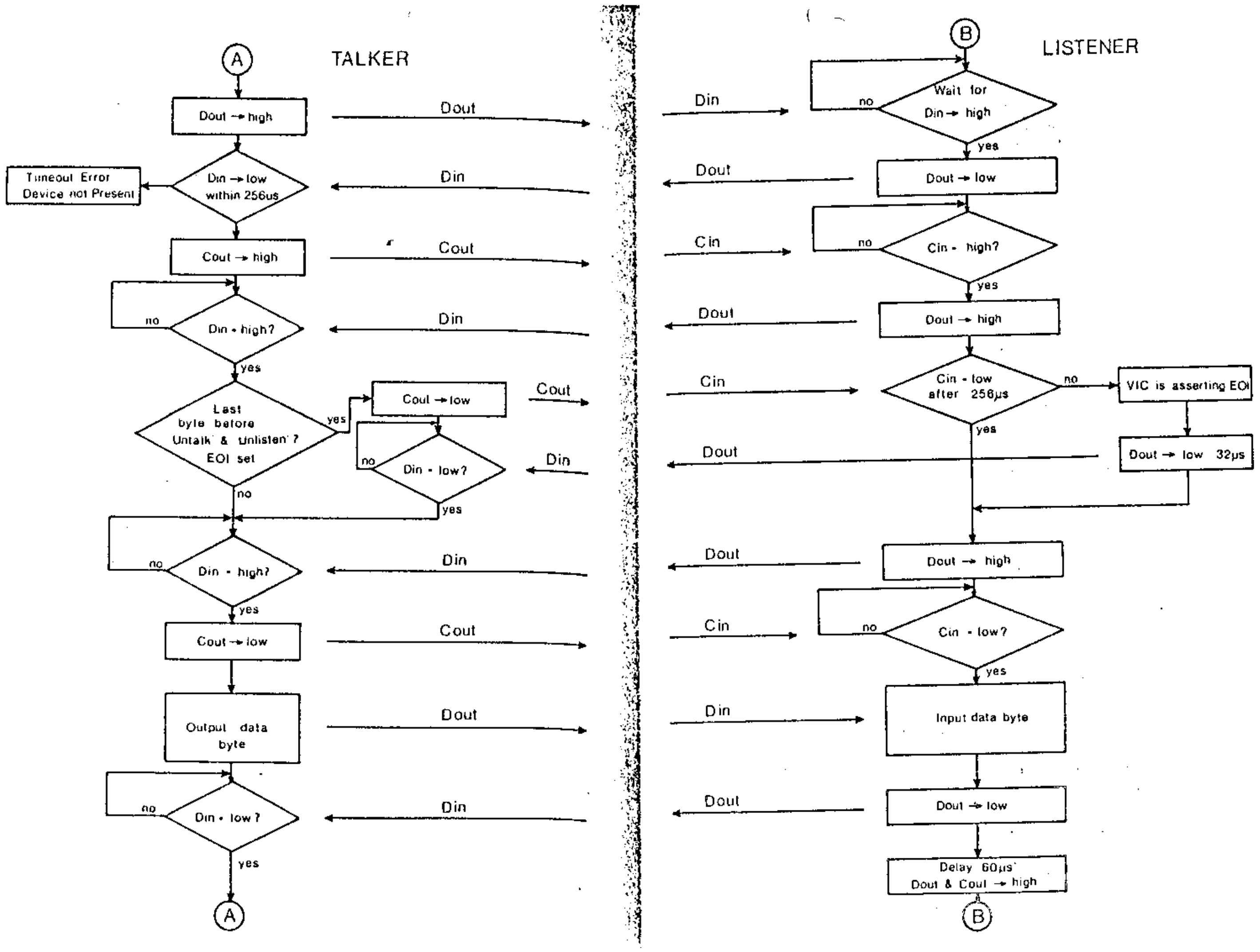
ATN SEQUENCES

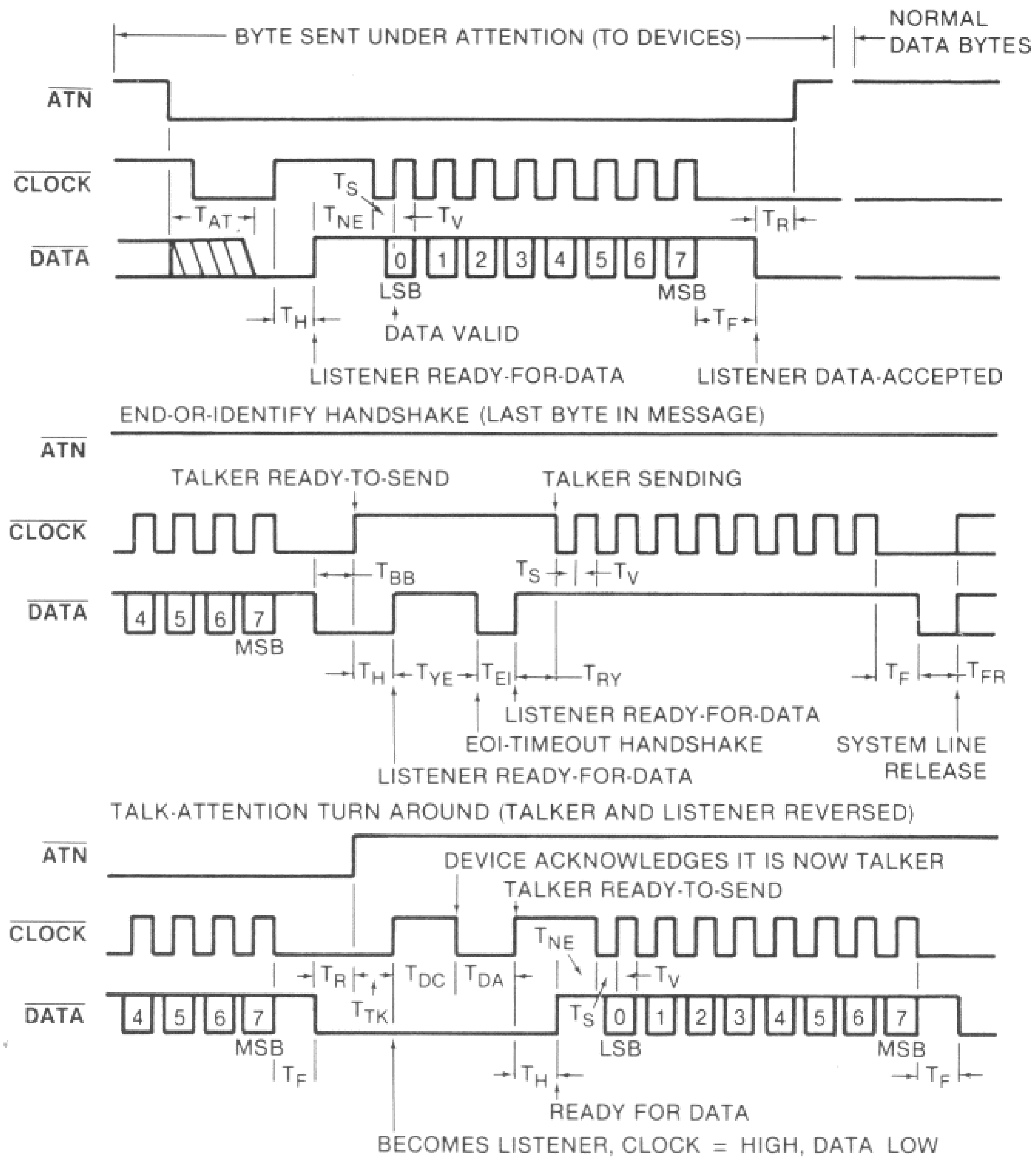
When ATN is pulled true, everybody stops what they are doing. The processor will quickly pull the Clock line true (it's going to send soon), so it may be hard to notice that all other devices release the Clock line. At the same time, the processor releases the Data line to false, but all other devices are getting ready to listen and will each pull Data to true. They had better do this within one millisecond (1000 microseconds), since the processor is watching and may sound an alarm ("device not available") if it doesn't see this take place. Under normal circumstances, transmission now takes place as previously described. The computer is sending commands rather than data, but the characters are exchanged with exactly the same timing and handshakes as before. All devices receive the commands, but only the specified device acts upon it. This results in a curious situation: you can send a command to a nonexistent device (try "OPEN 6,6") - and the computer will not know that there is a problem, since it receives valid handshakes from the other devices. The computer will notice a problem when you try to send or receive data from the nonexistent device, since the unselected devices will have dropped off when ATN ceased, leaving you with nobody to talk to.

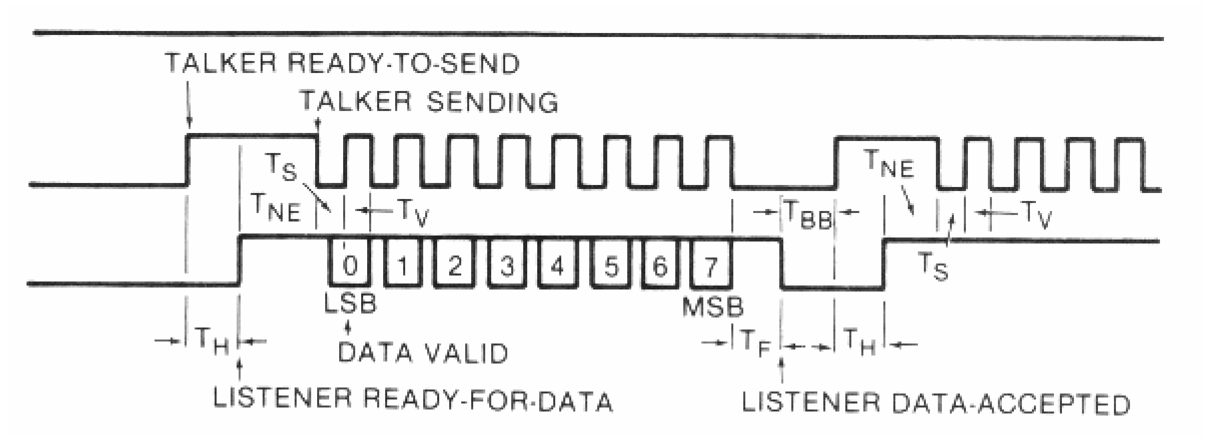
TURNAROUND

An unusual sequence takes place following ATN if the computer wishes the remote device to become a talker. This will usually take place only after a Talk command has been sent. Immediately after ATN is released, the selected device will be behaving like a listener. After all, it's been listening during the ATN cycle, and the computer
has been a talker. At this instant, we have "wrong way" logic; the device is holding down the Data line, and the computer is holding the Clock line. We must turn this around. Here's the sequence: the computer quickly realizes what's going on, and pulls the Data line to true (it's already there), as well as releasing the Clock line to false. The device waits for this: when it sees the Clock line go true, it releases the Data line (which stays true anyway since the computer is now holding it down) and then pulls down the Clock line. We're now in our starting position, with the talker (that's the device) holding the Clock true, and the listener (the computer) holding the Data line true. The

computer watches for this state; only when it has gone through the cycle correctly will it be ready to receive data. And data will be signalled, of course, with the usual sequence: the talker releases the Clock line to signal that it's ready to send. The logic sequences make sense. They are hard to watch with a voltmeter or oscilloscope since you can't tell which device is pulling the line down to true. The principles involved are very similar to those on the PET/CBM IEEE-488 bus - the same Talk and Listen commands go out, with secondary addresses and similar features. There are fewer "handshake" lines than on IEEE, and the speed is slower; but the principle is the same.




BYTE SENT UNDER ATTENTION (TO DEVICES)
NORMAL DATA BYTES
ATN
CLOCK
DATA
T_S
T_{AT}
T_{NE}
T_V
T_R
0 1 2 3 4 5 6 7
LSB
MSB
T_H
T_F
DATA VALID
LISTENER READY-FOR-DATA
LISTENER DATA-ACCEPTED
END-OR-IDENTIFY HANDSHAKE (LAST BYTE IN MESSAGE)
ATN
TALKER READY-TO-SEND
TALKER SENDING
CLOCK
T_{BB}
T_S
T_V
DATA
4 5 6 7
MSB
T_H
T_{YE}
T_{EI}
T_{RY}
T_F
T_{FR}
LISTENER READY-FOR-DATA
EOI-TIMEOUT HANDSHAKE
LISTENER READY-FOR-DATA
SYSTEM LINE RELEASE
TALK-ATTENTION TURN AROUND (TALKER AND LISTENER REVERSED)
ATN
DEVICE ACKNOWLEDGES IT IS NOW TALKER
TALKER READY-TO-SEND
CLOCK
T_{NE}
T_R
T_{DC}
T_{DA}
T_V
T_{TK}
T_S
DATA
4 5 6 7
0 1 2 3 4 5 6 7
MSB
LSB
MSB
T_F
T_H
T_F
READY FOR DATA
BECOMES LISTENER, CLOCK = HIGH, DATA LOW



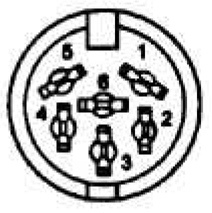
CBM Serial Bus Control Codes				
Description	**Symbol**	**Min.**	**Typ.**	**Max.**
ATN RESPONSE (REQUIRED)[1]	T_{AT}	-	-	1000µs
LISTENER HOLD-OFF	T_H	0	-	infinite
NON-EOI RESPONSE TO RFD[2]	T_{NE}	-	40µs	200µs
BIT SET-UP TALKER[4]	T_S	20µs	70µs	-
DATA VALID	T_V	20µs	20µs	-
FRAME HANDSHAKE[3]	T_F	0	20	1000µs
FRAME TO RELEASE OF ATN	T_R	20µs	-	-
BETWEEN BYTES TIME	T_{BB}	100µs	-	-
EOI RESPONSE TIME	T_{YE}	200µs	250µs	-
EOI RESPONSE HOLD TIME[5]	T_{EI}	60µs	-	-
TALKER RESPONSE LIMIT	T_{RY}	0	30µs	60µs
BYTE-ACKNOWLEDGE[4]	T_{PR}	20µs	30µs	-
TALK-ATTENTION RELEASE	T_{TK}	20µs	30µs	100µs
TALK-ATTENTION ACKNOWLEDGE	T_{DC}	0	-	-
TALK-ATTENTION ACK. HOLD	T_{DA}	80µs	-	-
EOI ACKNOWLEDGE	T_{FR}	60µs	-	-

Notes:

1: If maximum time exceeded, device not present error.
2: If maximum time exceeded, EOI response required.
3: If maximum time exceeded, frame error.
4: T_V and T_{PR} minimum must be 60µs for external device to be a talker.
5: T_{EI} minimum must be 80µs for external device to be a listener.

4 Serial bus pinout

Serial Bus Pinouts			
Pin	**Pin name and function**		
1	SRQ	:	Serial Service Request In
2	GND	:	Ground
3	ATN	:	Serial Attention In/Out
4	CLK	:	Serial Clock In/Out
5	DATA	:	Serial Data In/Out
6	RESET	:	Serial Reset



IEC connector
(on rear of CBM computer)

SRQ: Serial Service Request In:
This signal is not used on the C64. On C128 it is replaced with Fast Serial Clock for the 1571 disk drive.

ATN: Serial Attention In/Out:
Sending any byte with the ATN line low (sending under attention) causes it to be interpreted as a Bus Command for peripherals on the serial bus. When the C64 brings this signal LOW, all other devices start listening for it to transmit an address. The device addressed must respond in a preset period of time; otherwise, the C64 will assume that the device addressed is not on the bus, and will return an error in the STATUS word.

CLK: Serial Clock In/Out:
This signal is for timing the data sent on the serial bus. This signal is always generated by the active TALKER. RISING EDGE OF THE CLOCK means data bit is valid.

DATA: Serial Data In/Out:
Data on the serial bus is transmitted bit by bit at a time on this line.

RESET: Serial Reset:
Some say that you may disconnect this line to save your disk drive, but in fact resetting the drive when you reset your computer is very logical and is the only way to reset your drive whithout switching it OFF and ON.

5 Bytes sequences of specific operations

CBM Serial Bus Control Codes			
Base address	**Command name and details**		
20	LISTEN	+	device number (0-30)
3F	UNLISTEN		
40	TALK	+	device number (0-30)
5F	UNTALK		
60	OPEN CHANNEL / DATA	+	Secondary Address / channel (0-15)
70	-Undefined-		
80	-Undefined-		
90	-Undefined-		
A0	-Undefined-		
B0	-Undefined-		
C0	-Undefined-		
D0	-Undefined-		
E0	CLOSE	+	Secondary Address / channel (0-15)
F0	OPEN	+	Secondary Address / channel (0-15)

/	=	byte is send under attention (ATN low)
↑	=	bus turn around (computer is set to slave)
↓	=	bus turn around (computer is set to master (normal situation)

Note: the device (1541) is called by the computer (C64)

5.1 Normal operation:

5.1.1 Loading a program

LOAD"filename",8,1

/28 /F0 filename /3F
/48 /60 ↑read data↓ /5F
/28 /E0 /3F

The IEC-bus traffic is shown below:

```
28 F0 filename 3F
.. .. filename  ..

48 60


01 08
start address of this program 0801

0F 08 0A 00 9F 20 31 35 2C 38 2C 31 35 00
                                       eol

20 08 14 00 84 31 35 2C 41 2C 42 24 2C 43 2C 44 00
                                                eol

30 08 1E 00 99 20 41 2C 20 42 24 2C 43 2C 44 00
                                             eol

39 08 28 00 A0 20 31 35 00 00 00
                        end of file

5F
..

28 E0 3F
.. .. ..


When listed this appears to be the following program
```

10 OPEN 15,8,15
20 INPUT#15,A,B$,C,D
30 PRINT A,B$,C,D
40 CLOSE 15

5.1.2 Saving a program

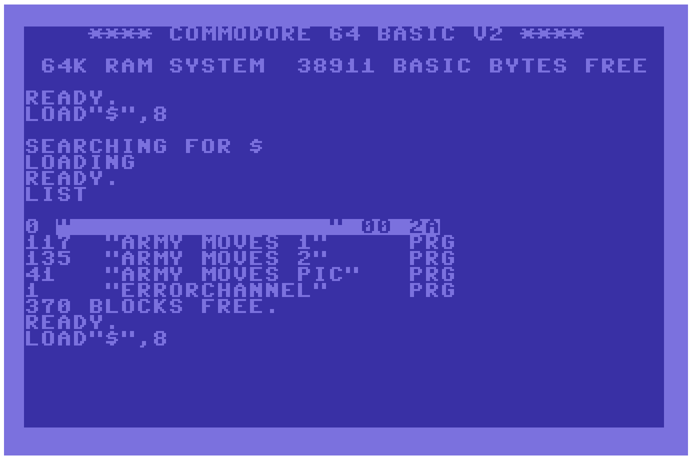
SAVE"filename",8,1

/28 /F0 filename /3F
/28 /60 send data /3F
/28 /E0 /3F

5.1.3 Loading of a directory

LOAD”$”,8
LIST

The output on the screen of the C64 :



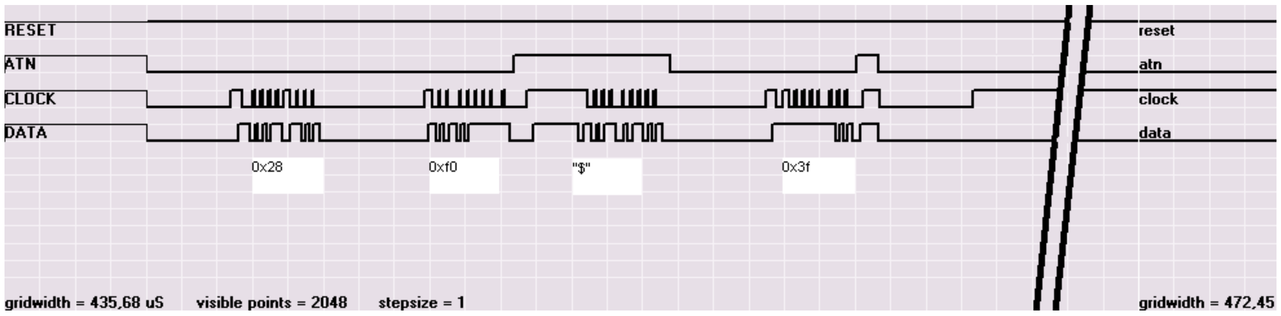
```
    **** COMMODORE 64 BASIC V2 ****

 64K RAM SYSTEM  38911 BASIC BYTES FREE

READY.
LOAD"$",8

SEARCHING FOR $
LOADING
READY.
LIST

0 "                " 00 2A
117   "ARMY MOVES 1"      PRG
135   "ARMY MOVES 2"      PRG
41    "ARMY MOVES PIC"    PRG
1     "ERRORCHANNEL"      PRG
370 BLOCKS FREE.
READY.
LOAD"$",8
```

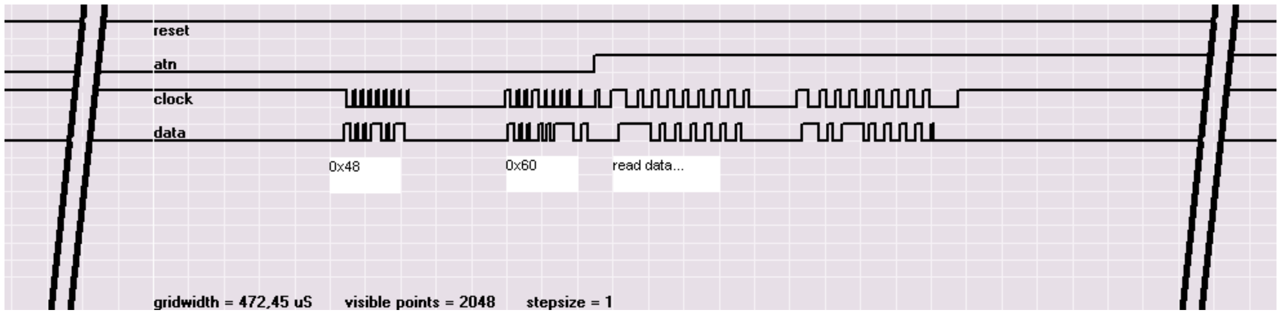
The IEC-bus traffic is shown below:

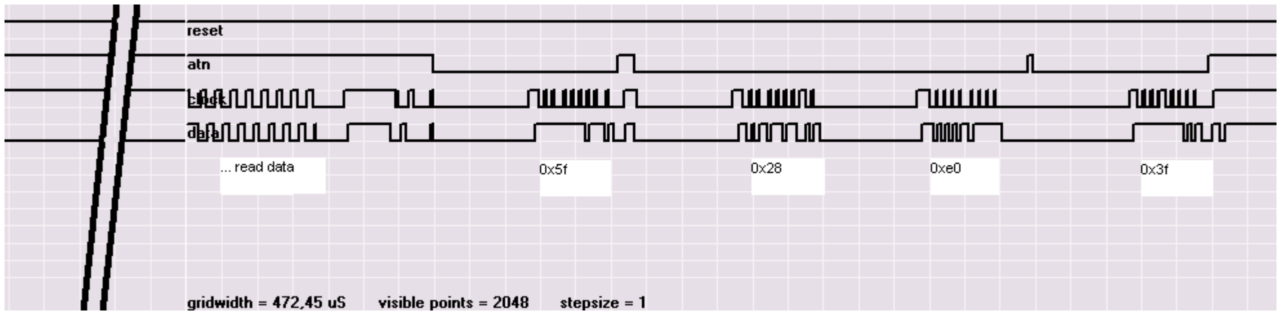
```
/28 /F0 24 /3F
... ... $  ...

/48 /60 ↑
... ...

01 04   01 01   00 00   12 22 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 22 20 30 30 20 32 41 00
loadadr start   drive0  RVS"                                                  "     0  0     2  A  EOL

01 01   75 00   20 22 41 52 4D 59 20 4D 4F 56 45 53 20 31 22 20 20 20 20 20 50 52 47 20 20 20 20 00
starte blocks   "  A  R  M  Y     M  O  V  E  S     1  "                 P  R  G                 EOL

01 01   87 00   20 22 41 52 4D 59 20 4D 4F 56 45 53 20 32 22 20 20 20 20 20 50 52 47 20 20 20 20 00
start  blocks   "  A  R  M  Y     M  O  V  E  S     2  "                 P  R  G                 EOL

01 01   29 00   20 20 22 41 52 4D 59 20 4D 4F 56 45 53 20 50 49 43 22 20 20 20 50 52 47 20 20 20 00
start  blocks   "  A  R  M  Y     M  O  V  E  S     P  I  C  "           P  R  G                 EOL

01 01   01 00   20 20 20 22 45 52 52 4F 52 43 48 41 4E 4E 45 4C 22 20 20 20 20 20 50 52 47 20 20 00
start  blocks   "  E  R  R  O  R  C  H  A  N  N  E  L  "                 P  R  G                 EOL

01 01   72 01   42 4C 4F 43 4B 53 20 46 52 45 45 2E 20 20 20 20 20 20 20 20 20 20 20 20 20 00 00 00
start  blocks   B  L  O  C  K  S     F  R  E  E  .                                         End Of File

↓ /5F
  ...

/28 /E0 /3F
... ... ...
```

<u>Note:</u> you may have noticed that every line is exactly 32bytes long. In order for the Final Cartrdige III (and other programs) to be supported correctly this behaviour should be copied exactly. Although the kernal routines doe not require this. When you leave out the spaces (20) before the EOL (00) you will notice no difference when loading the directory with the LOAD”$”,8 command. However a program that loads the directory and interprets it’s data immediately might generate strange behaviour due to the missing bytes. So in short, keep every line 32 bytes and all will be fine.

If you would have a logic analyzer connected to the IEC-bus, you would have seen the following signals. These images were taken using a PC with a hacked parallel port logic analyzer, the samplerate is barely suffiecient so these aren't the best IEC-bus captures in the world. However the give a good indication of the signals in this situation.







5.2 "FILE NOT FOUND" situation:

<u>Read file from device#8</u>

LOAD"filename",8,1

/28 /F0 filename /3F
/48 /60 ↑↓ release data and clock *This situation is recognized by the computer as an error*
/28 /E0 /3F

5.3 Abort current action (for example: pressing run/stop during LOAD):

<u>Read file from device#8</u>

LOAD"filename",8,1

When run/stop is pressed, the computer makes ATN low (0 Volt). The device that is talking stops sending bytes and the device that was talking immediatly becomes a listener (the bus does another turnaround resulting in the computer as talker and the device as listener). The device is now waiting for further commands.

5.4 Reading the error channel:

When the following commands are given the IEC-bus traffic is as follows:
OPEN 15,8,15 *This line does not start IEC traffic, it prepares the computer for communication*
INPUT#15,A,B$,C,D
PRINT A,B$,C,D
CLOSE 15

The response on the screen of the C64:
00, OK,00,00

The bus traffic:

/48	**/6F**	↑	**30**	**30**	**2C**	**20**	**…**
TALK #8	DATA ch15	Turn	‘0’	‘0’	‘,’	‘SPACE’	…

?!?!?! Reading the error channel only works when no file has been opened on channel 15 ?!?!?!

But when executed as a program it is slightly different
10 OPEN 15,8,15
20 INPUT#15,A,B$,C,D
30 PRINT A,B$,C,D
40 CLOSE 15

The programs response on the screen of the C64:
00, OK,00,00

The IEC-bus traffic:
90 6F 30 30 2C 20 4F 4B 2C 30 30 2C 30 30 0D BE 50 EF 3F

6 The difference between a 2-line and a single line command to the 1541 drive :

Scratch a file name "T" on a media inside device#8

OPEN 15,8,15 ◄this line does not start IEC traffic, it prepares the computer for communication
PRINT#15,"S:T"

/28	**/6F**	**53**	**3A**	**54**	**0D**	**/3F**
LISTEN #8	DATA ch15	'S'	':'	'T'	'CR'	UNLISTEN

OPEN 15,8,15,"S:T"

/28	**/FF**	**53**	**3A**	**54**	**/3F**
LISTEN #8	OPEN ch15	'S'	':'	'T'	UNLISTEN

Note: The second method can only be used for the errorchannel (15) and is not use for data transfer. For data transfer the first method is used. However the opening of files can be done with the single line commandline. Example: **OPEN 8,9,10,"FILENAME,S,W"**

To send a data byte to a drive, that device must first be "listened". If the Secondary address (from here referred to as: SA or channel) is 15, the drive will interpret the data as a DOS command. A DOS command is executed when the drive is UNLISTENed ($3F).

If the channel is not 15, DOS will ignore it unless you first sent an OPEN.
An OPEN is sent to tell DOS where you want your data to go.
That is done by LISTENing the device. With -ATN high, a character string must be sent. That will be either a filename (to open a write file), or something like "#", "#2", etc. That tells DOS to write your data to one of the five DOS buffers. Then, with -ATN low, send an UNLISTEN.
channel = 0 is reserved for a reading a PRG file.
channel = 1 is reserved for a writing a PRG file.
channel = 2-14 need the filetype and the read/write flag in the filename as ",P,W" for example.
channel = 15 for DOS commands or device status info.

After the OPEN is sent, you can send a LISTEN using the channel used in the OPEN.
DOS has a table of opened files, and will use the channel to write your data to the corresponding file.

The purpose of a CLOSE, for a file named "#", is to free up that DOS buffer. For a file whose name appears in the disk directory, a CLOSE will set bit 7 of the filetype byte in the directory descriptor, update the block count, and save BAM back to disk. DOS will even turn off the drive-LED.

You can keep files open in several devices, and write data first to one and then another. Just UNLISTEN a device before LISTENing another. You can have a drive with a "LISTEN" device address of 8 and a "TALK" device address of 9. Another drive can use 8 for both "LISTEN" and "TALK" device addresses. That is how you can download a file to both devices at the same time, and still be able to read the individual error channels.

The KERNAL routines, when the computer is sending data, always buffer one byte. The KERNAL UNLISTEN, UNTALK, LISTEN, and TALK routines check for a buffered byte. If one is present, it is sent delayed. Then the routine does its regular job.

To open a file in a disk drive, there is no need also to open a file in the computer. However, that is usually done for the sake of convenience. But the KERNAL LOAD and SAVE routines open read and write files in the IEC (serial) device, without also opening files in the computer. To open a file in an IEC device, the -ATN line must be taken low (by setting bit 3 of $DD00). Then, the CLK line is taken low (bit 4 of $DD00 is set), and the computer waits for one millisecond. After which the device number ORed with $20 is sent via slow serial. With -ATN still low, the channel ORed with $F0 is then sent; -ATN is then taken high. Next, the filename string is sent to the drive, still using slow serial. The string is terminated by "=", by ",", or by a count of 16 characters, whichever comes first. If the drive DOS encounters ",a" or ",a,b" (where "a" and "b" may be any characters), it uses that information to determine file type and data direction. If the character is not valid it is tossed; there is no error. First direction is checked; "w" is "write" and "r" is "read". That may be overridden by the channel; an channel of 0 defines "read" and an channel of 1 defines "write". For any other channel (2 to 14), if there is no direction character, "read" is set. DOS then looks for a file type characater; "u" is "usr", "p" is "prg", and "s" is "seq". If the direction is "read", and there is no file type character, ANY file type is acceptable. An exception occurs for no file type character and an channel of 0; that defines "prg". If the direction is "write", and there is no file type character, "seq" is set. An exception occurs for no file type character and an channel of 1; that defines "prg". In the special case of the string's being ",w", the infamous "comma" filename will appear in the directory of a 1541 disk. This C-64 BASIC program will write 144 files named "," to a freshly formatted disk. The files cannot be directly deleted.

```
10 fori=1to144:open8,8,8,",w":close8
20 next
```

After the string is sent, the device is UNLISTENed. -ATN is taken low, CLK is taken low, the computer waits for one ms, and $3F is sent to the device. The file is now opened in the device; the file may be closed by doing a modified OPEN. No string is sent, and the channel is ORed with $E0 rather than with $F0. Between the OPEN and the CLOSE, data may be read from a "read" file by TALKing the drive, or written to a "write" file by LISTENing the drive. Also, a device may be UNTALKed or UNLISTENed. For LISTEN, the device number ORed with $20 is sent as with OPEN or CLOSE. Then, with -ATN still low, the channel is sent ORed with $60. -ATN is then taken high. For TALK, it's the same, except that the device number is ORed with $40. Also for TALK, after the channel is sent, the bus must be "turned around". That is accomplished by taking the DATA line low (setting bit 5 of $DD00), taking the -ATN line high (clearing bit 3 of $DD00), taking the CLK line high (clearing bit 4 of $DD00), and looping until the CLK line is low (until bit 6 of $DD00 is clear). For UNLISTEN and UNTALK, just one byte is sent (in the same manner as the device number is sent for LISTEN, TALK, etc.). That byte is $3F for UNLISTEN and $5F for UNTALK; ALL devices on the bus are commanded either to close their ears or to shut up. After the byte is sent, -ATN is taken high.

7 How to change the device number without changing the jumpers

This actually works on all the drives. It works on all the variations of the 1541, plus the 1571, 1581, and all the CMD drives including the RAMLink and the RAMDrive.
If you want to disable a drive, modify line 20 so it will accept a value of 1. Since no software will ever attempt to access device 1 the drive will never respond and will sit there quietly even though it is still turned on. If you still want to be able to bring it back to life, change the device number to 6 or 7. Very few programs ever look for anything below device 8.

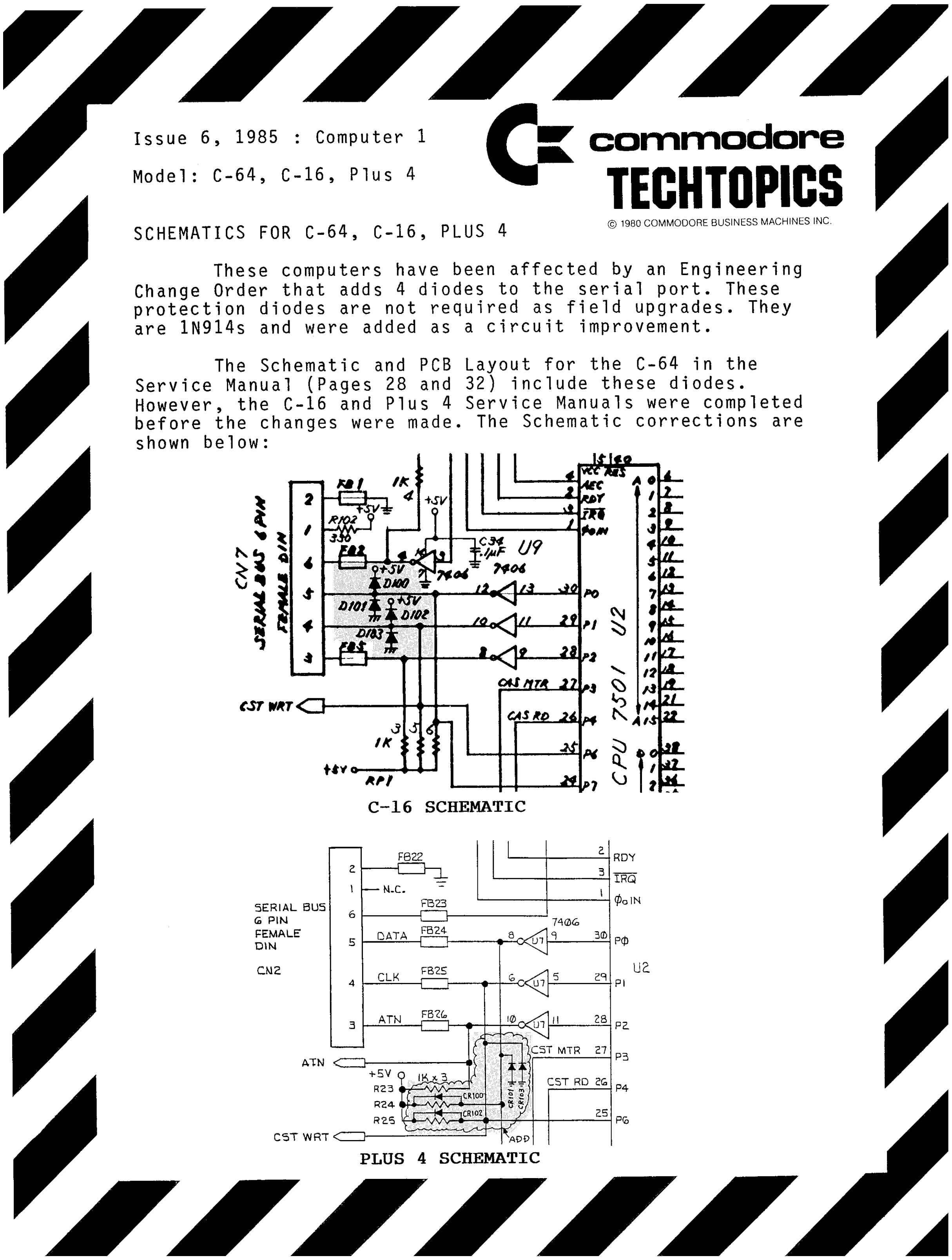
```
5 INPUT "OLD DEVICE NUMBER";ODV
10 INPUT "NEW DEVICE NUMBER";DV
20 IF DV<8 OR DV>11 THEN 10
30 OPEN 15,ODV,15
40 PRINT#15,"M-W"CHR$(119)CHR$(0)CHR$(2)CHR$(DV+32)CHR$(DV+64)
50 CLOSE 15
```

8 Commodore 64, serial bus related KERNAL functions

Address	Function
$ED09	Send TALK command to serial bus. Input: A = Device number. Output: – Used registers: A.
$ED0C	Send LISTEN command to serial bus. Input: A = Device number. Output: – Used registers: A.
$ED40	Flush serial bus output cache, at memory address $0095, to serial bus. Input: – Output: – Used registers: A.
$EDB9	Send LISTEN secondary address to serial bus. Input: A = Secondary address. Output: – Used registers: A.
$EDC7	Send TALK secondary address to serial bus. Input: A = Secondary address. Output: – Used registers: A.
$EDDD	Write byte to serial bus. Input: A = Byte to write. Output: – Used registers: –
$EDEF	Send UNTALK command to serial bus. Input: – Output: – Used registers: A.
$EDFE	Send UNLISTEN command to serial bus. Input: – Output: – Used registers: A.
$EE13	Read byte from serial bus. Input: – Output: A = Byte read. Used registers: A.
$EE85	Set CLOCK OUT to high. Input: – Output: – Used registers: A.
$EE8E	Set CLOCK OUT to low. Input: – Output: – Used registers: A.
$EE97	Set DATA OUT to high. Input: – Output: – Used registers: A.
$EEA0	Set DATA OUT to low. Input: – Output: – Used registers: A.
$EEA9	Read CLOCK IN and DATA IN. Input: – Output: Carry = DATA IN; Negative = CLOCK IN; A = CLOCK IN (in bit #7). Used registers: A.

9 Commodore additional IEC-bus info

Some additional info spread by commodore…



Issue 6, 1985 : Computer 1

Model: C-64, C-16, Plus 4

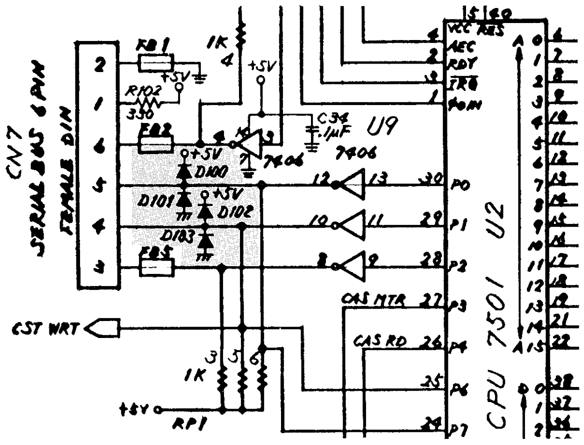
commodore TECHTOPICS

© 1980 COMMODORE BUSINESS MACHINES INC.

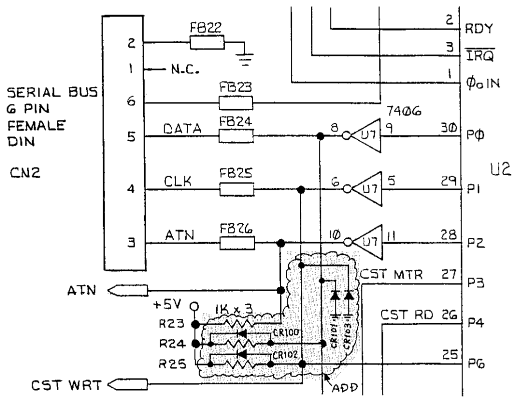
SCHEMATICS FOR C-64, C-16, PLUS 4

These computers have been affected by an Engineering Change Order that adds 4 diodes to the serial port. These protection diodes are not required as field upgrades. They are 1N914s and were added as a circuit improvement.

The Schematic and PCB Layout for the C-64 in the Service Manual (Pages 28 and 32) include these diodes. However, the C-16 and Plus 4 Service Manuals were completed before the changes were made. The Schematic corrections are shown below:



C-16 SCHEMATIC

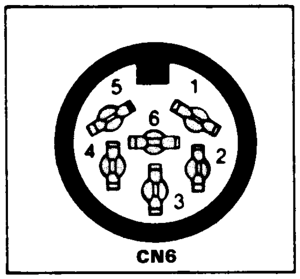


PLUS 4 SCHEMATIC

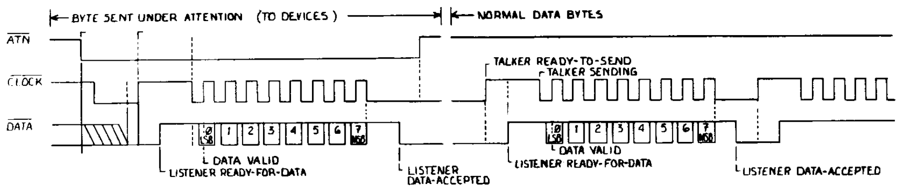
THE SERIAL BUS

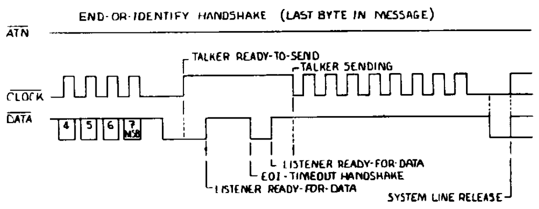
The C128 Serial Bus is an improved version of the C64/VIC 20 serial bus. The C128 improves this bus by allowing communication at much greater speeds with specially designed peripherals, the most important being the disk drive, while still maintaining capability with older, slower peripherals used by the VIC 20 and the C64.

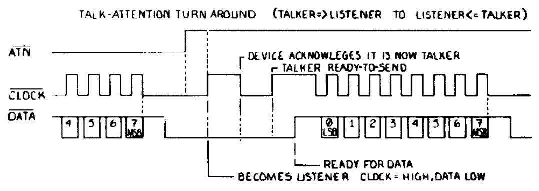
Serial Interface Connector



Pin No.	Signal	Description
1	SERIAL SRQ	The slow serial bus does not use the SERVICE REQUEST line. The fast serial bus uses it as a fast bidirectional clock line.
2	GND	Chassis ground.
3	SERIAL ATN	The ATTENTION line is a low active handshake used to address a device on the bus.
4	SERIAL CLK	This is the slow serial CLOCK. It is used by slow serial devices to clock data transmitted on the serial bus.
5	SERIAL DATA	The bidirectional serial DATA line is used by both slow and fast devices to transmit data in sync with a clock signal.
6	$\overline{\text{RES}}$	The RESET line is used to reset all peripherals when the host resets.






TALK-ATTENTION TURN AROUND (TALKER=>LISTENER TO LISTENER<=TALKER)
ATN
CLOCK
DATA
DEVICE ACKNOWLEGES IT IS NOW TALKER
TALKER READY-TO-SEND
READY FOR DATA
BECOMES LISTENER CLOCK=HIGH, DATA LOW

Bus Operations

There are three basic bus operations that take place on the serial bus, in both fast and slow modes. The first of these is called **Control.** The C128 is the **controller** in most circumstances. The **controller** of the bus is always the device that initiates protocol on the bus, requesting peripheral devices to do one of the two other serial operations, either **Talk** or **Listen.**

All serial bus devices can listen. A **Listener** is a device that has been ordered by the **Controller** to receive data. Some devices, such as disk drives, can talk. A **Talker** can send data to the **Controller.** Both hardware and software drive this bus protocol.

The Standard (Slow) Serial Bus

The slow serial bus uses the port lines of the 6526 at U4, C1A 2, to drive ATN, CLK and DATA. The operation is the same as that of the C64 and when in C64 mode, slow to fast interference is automatically removed.

The Fast Serial Bus

In order to talk as a fast talker, the Controller must be addressing a fast device. When addressing any device, the C128 sends a fast byte, toggling the SRQ line eight times, with the ATN line low. If the device is a fast device, it will record the fact that a fast Controller accessed it and respond with a fast acknowledge. If the device is a slow device, no response is delivered and the C128 then assumes it is talking with a slow device. The status of the fast or slow is retained until the device is requested to untalk, unlisten, or if an error or system reset occurs.

The fast serial bus, in order to achieve its speed increase, uses different hardware than that of the slow serial bus. The fast serial method is to use the serial port lines of the 6526 U1, CIA 1, pin 39, to actually transfer the serial data. This increases the transfer rate dramatically.

The $\overline{\text{FSDIR}}$ bidirectional control line signals the MMU at U7, pin 44, that a fast device is present. The MMU then outputs control signals to the data direction buffer hardware for fast serial operation.

10 How fastloaders work...

10.1 The Final Cartridge III (FC3) load/save protocol

Credits go to: Thomas Giesel, who analysed and documented the load/save protocol
Ingo Korb, notes about the Freezer-variation

This document describes the fast loader and saver protocol of the FC3. The description may not be accurate in any point.

10.1.1 Loading directories

The loading of directories using the FC3 is…

10.1.2 Transferring the load/save routines to the drive

Before the device is capable of understanding the fastloader protocol it requires some additional routines to be uploaded into the device. These routines are the load and save routines and are uploaded using the "M-W" commands. Once these routines are uploaded they will be executed using the "M-E" command. Now the device is capable of fastloading/saving.

The uploaded code is shown below:

calculating the required CRC is done using:

```
datacrc = _crc16_update(datacrc, command_buffer[i]);

void crc16_update(crc, new_byte)
{
        crc = (unsigned char)(crc >> 8) | (crc << 8);
        crc ^= new_byte;
        crc ^= (unsigned char)(crc & 0xff) >> 4;
        crc ^= (crc << 8) << 4;
        crc ^= ((crc & 0xff) << 4) << 1;
}
```

The detection of the loadertype is done using the calculated CRC:

```
  if (datacrc == 0xf1bd) {
    detected_loader = FL_FC3_LOAD;
  }
  else if (datacrc == 0xbe56) {
    detected_loader = FL_FC3_SAVE;
  }
```

For this reason the sd2iec implements these two commands (check doscmd.c). When a C64 uses the M-W command the sd2iec calculates a chacksum over all bytes written. Often fastloaders use several M-Ws to write the while code, so the checksum is updated on each command. Of course the data being written using M-W is discarded but not kept. When the C64 uses M-E <start addr> the pair (<checksum>, <start addr) to identify the fast loader. Then a native (Atmel) implementation of the fast loader is called instead of the 6502 implementation. These implementations can be found in fastloader*.* Maybe it's best you check these sources and maybe you want to port them to PIC. Remember that the license is GPL... Possibly you are also interested in DreamLoad. This is also implemented in sd2iec (no binary released yet, but you can get the latest sources). With dreamload you can start some Demos using this IRQ fast loader.

10.1.3 Loading

First of all the FC3 opens the file and reads the first two bytes to check the load address. It does not close the file before starting the fast loader, so that one just starts with the last sector loaded.

When the loader starts it sets both data and clock to high, to signal that it is currently busy.

The FC3 does always transfer whole blocks, even if not all bytes are used. Each block is transferred in 65 tuples with 4 bytes each.

Before sending a block the drive pulls CLOCK low and waits for the host to respond by pulling DATA low. Then the drive and host release these CLOCK and DATA.

The first tuple is sent about 180 us after this handshake. As each tuple has its own synchronization this timing doesn't need to be accurate. Between two tuples there are about 190 us.

1st tuple:	0: not used (always 7, data marker on disk)
	1: block counter, starting at 0
	2: 0 if all bytes must be used, number of bytes otherwise
	3: First byte of block
2nd..64th tuple:	0: byte from block
	1: byte from block
	2: byte from block
	3: byte from block
65th tuple:	0: last byte of block
	1: not used
	2: not used
	3: not used

Each tuple is syncronized by the drive pulling CLOCK low, then 2 bits are transfered at once until the 4 bytes are done. Bit 0 is on CLOCK, Bit 1 on DATA and so on. Electrical 0 means binary 0. Finally CLOCK and DATA are released to high level.

EOF is marked by pulling DATA low instead of starting a new handshake.
I/O errors are marked by pulling DATA and CLOCK low.

The "|" marks are the point of time when the bus is written or read.

```
C64 read (PAL)
 1   (+12)   13  (+12)   25  (+12)    37  (+12)   49   (+14)    63 (+12)   75  (+12)   87  (+12)   99   (+14)     113  (+12)   125   (+12)      cycle

-|||||||-----|||||||-----|||||||------|||||||-----|||||||-------|||||||----|||||||-----|||||||-----|||||||--------|||||||------|||||||---------
 1..7        13.2..19.2  25.4..31.4   37.6..43.6  49.7..55.7    63.9..69.9 76.1..82.1  88.3..94.3  00.5..06.5     14.7..20.7   26.9..32.9       us

 (sync)

1541 write
|-----------|-----------|------------|-----------|------------|-----------|------------|-----------|------------|-----------|-------------------

0   (+12)   12  (+12)   24   (+13)   37  (+12)   49   (+13)   62  (+12)   74   (+13)   87   (+12)  99   (+13)   112  (+12)  124  (+13)          us

CLK low     1st pair    2nd pair     3rd pair    4th pair     5th pair    6th pair     7th pair    8th pair     9th pair    10th pair


sd2iec write
|-----------|-----------|-----------|-----------|-------------|-----------|-----------|-----------|-------------|-----------|-------------------

0  (+12)    12  (+12)   24  (+12)   36  (+12)   48   (+14)    62  (+12)   74  (+12)   86  (+12)   98   (+14)    112  (+12)  124  (+12)          us




===============================================================================


C64 read (PAL) cont'd
        137  (+12)  149   (+14)   163  (+12)   175  (+12)  187  (+12)              199                                                        cycle
...-----|||||||-----|||||||-------|||||||------|||||||-----|||||||---------------|||||||-------------------------------------------------------
        39.1..45.1  51.2..57.2    65.4..71.4   77.6..83.6  89.8..95.8           02.0..08.0                                                     us


1541 write
...---|-----------|------------|-----------|------------|---------------------|----------------------------------------------------------------
      137  (+12)  149  (+13)   162  (+12)  174  (+13)   187  (+12)            199  (+16)                                                        us
      11th pair   12th pair    13th pair   14th pair    15th pair             16th pair

sd2iec write
...--|-----------|-------------|-----------|-----------|--------------------|----------------------------------------------------------------
     136  (+12)  148  (+14)    162  (+12)  174  (+12)  186  (+12)          198   (+14)

                                                           us
```

Fastloader in freezed files

Programs freezed using the "F.DISK" command of the FC3 use a fast loader similiar to the normal FC3 fastloader. Like the normal fastloader it opens the target file for reading with the standard protocol and it also uses the same protocol and timing to transmit a four-byte tuple to the C64. However, the handshaking in between and the contents of the first tuple are different. The freezer-variant transmits the data from disk directly which means that the first tuple contains a 0x07, the track and sector of the next sector in a file and the first data byte. Everything but the data byte is ignored by the C64-side code.

Both end-of-file and error conditions are signalled by pulling DATA low. A full clock/data handshake is done before the transmission of every tuple which means that the fixed delays between tuples can be dropped.

The freeze-loader closes and reopens the data file using the standard protocol after the fast protocol has finished transmitting the file.

10.1.4 SAVING

The FC3 first opens the file for writing, if this works well, it starts its drive code. This code receives and saves the data.

The first byte marks the size of the block. If this byte is 0, a full block of 254 bytes follows which is not the last one. If other numbers are received, n - 1 bytes will follow and this will be the last block. Note that transferred blocks do not correspond to disk sectors, as each full block has 254 bytes, also the first one which contains the start address.

The drive releases DATA to show that it's ready to receive data. Then the C64 releases CLOCK to synchronize the file transfer.

The following diagram shows the position of each bit being transfered. The bits are send low-active.

```
C64 write (PAL)
0   (+12)   12  (+10) 22    (+16)    38  (+10) 48  (+10) 58               cycle
|-----------|---------|--------------|---------|---------|---------------
0           11.8      21.7           37.4      47.3      57.1                us
CLOCK high  D=5       D=4            D=1       D=0       CLOCK low
(sync)      C=7       C=6            C=3       C=2       (busy)

sd2iec read
|----------------|------------|-----------|---------|--------------------
0     (+17)      17   (+13)   30  (+12)   42  (+10) 52                       us
```

After reading the last pair of bits the drive pulls DATA down to show it's busy. Apparently there's no mechanism to signal situations like "disk full".