CASSIOPEI v2.0

Programmers reference

this manual is intended for
firmware version: V20191006 or higher

Introduction

This reference is intended for anyone who wants to write software on a CBM computer for use with the Cassiopei. This manual consists of technical information varying from the way a datasette works on a CBM computer to how the protocol for data transfer and the command definitions required for transferring data between Cassiopei and CBM computer.

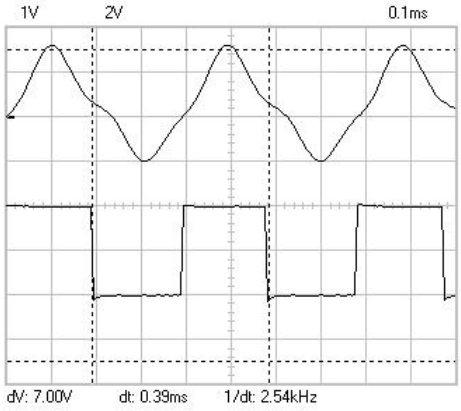
This manual is mainly intended for advanced programmers.

Table of Contents

1 Tape protocol of the CBM kernal loader

1.1 Signal encoding

Shown here is the pilot wave (a.k.a. trailing tone). It is the long beep at the start of each program. As you can see, the sinewave it not a very clean one. It looks more like a triangle. Fortunately this is not a real problem as the signal is digitized by a schmitt trigger circuit into a digital high or low value.



The upper signal is the signal as read from tape. The lower signal is the digitized value that is send to the computer. The signal shown is the signal defined as a series of S's.The S stands for Short pulse (more about this at the bottom of this page)

The system uses three different frequencies that and are always used in pairs. The only exception is the pilot tone because this is just a series of S's for several seconds. The pilot tone is used for the C64 to lock into the tape signal. This because the tape could have been recorded on a different recorder with a different speed. Since the Pilot tone is of a defined frequency, the C64 can measure the freq. of the pilot tone and compensate for the difference in tape speed. This is why the pilot tone is heard for several seconds. Also, when a tape is started it requires time to get to the proper and stable speed. This also add to the length of the pilot tone. You can imagine that a digital system does not have these tape speed problems and for those systems (like the Cassiopei) the pilot tone can be many times shorter, but only if the generated signals are perfectly within the range of the specifications.

Pulse duration specifications

Theoretical values for S, M and L as described in a "Data Becker" book (C64, VIC20, PET, C128)
Short : a sine of 2840Hz (=>352uS), this means that a high-pulse takes 176uS and that the low -pulse takes 176uS
Medium : a sine of 1953Hz (=>512uS), this means that a high-pulse takes 256uS and that the low -pulse takes 256uS
Long : a sine of 1488Hz (=>672uS), this means that a high-pulse takes 336uS and that the low -pulse takes 336uS

The C16 and Plus4 use slightly different timings
Short : a sine of 2045Hz (=>489uS)
Medium : a sine of 1036Hz (=>965uS)
Long : a sine of 520Hz (=>1923uS)

The following combinations are defined:

Pilot wave		: S
Byte marker waves	(more data follows)	: LM
Byte marker waves	(end-of-data)	: LS
Data waves	bit = 0	: SM
Data waves	bit = 1	: MS

Note:

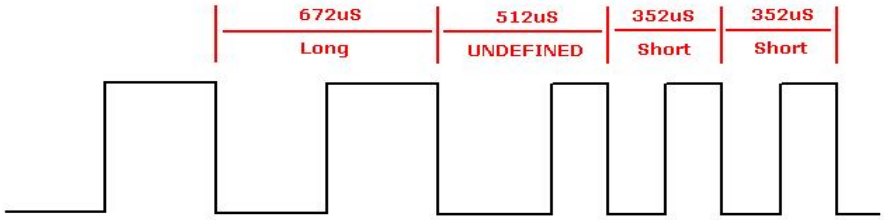
Somewhere on the internet there is a rumor that the C16/Plus4 were originally designed to operate on 2MHz instead of 1MHz. And that for some reason on the very last minute in the design it was decided to lower the freq. to 1MHz. This information was derived from the fact that the tape frequencies were exactly half the freq. of the other CBM models. But if you look at the info above this isn't the really case at all. Sure the "Medium" pulse of 1036Hz looks like the halve of 1953Hz. But if this were true then why is the "Long" pulse 520Hz and not 744Hz and why the "Short" pulse 2045Hz and not 1420Hz.

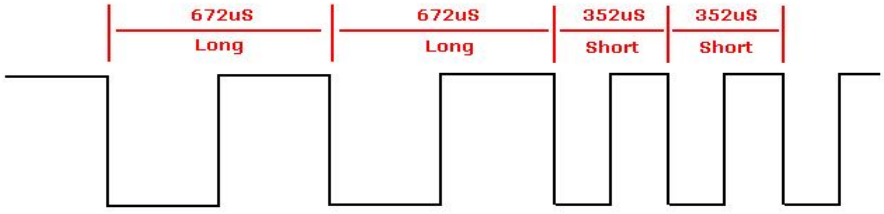
One thing is certain, it doesn't matter if the C16 or Plus 4 run on 1MHz of 2MHz the tape signals are not fully compatible. They simply deviate too much to be fully accepted by the other models. So the real question is why did Commodore choose these "strange" pulse times, why didn't they want the tape to be compatible?

1.2 Polarity of the signal

Now you might have wondered why is this important. Very simple. The C64 outputs a write signal to the tape that is the inverse of the signal read back from the tape. If you do not know this then this is a pitfall you can easily step into. So you can imagine that during the development of the Cassiopei I fell into this pitfall and did not understand why my signal could not be read back into the C64. The reason was simple in the end. I needed to invert it.

When you consider that **the C64 only can detect the falling edge of the read tape signal** (digitized tape output is connected to the CIA's FLAG input) it becomes clear that the C64 measures the time of a complete (digitized) sine. It does not detect the individual high or low pulse. It detects the falling edges of the pulse, which is very smart as it can be done using interrupts. Because the falling edges are triggered (and not the width of the high or low pulse) it is important that the signal is presented in the proper way. In other words in the correct polarity. When (for whatever reason) it is inverted then the C64 can no longer measure the duration of the digitized sine. Below are 2 signals. As you can see in the picture below, the first signal should be inverted. Because when it is not inverted the digitized sine is measured incorrectly and undefined timing values are detected which causes the C64 to generate a LOAD ERROR





1.3 How a byte is recorded

Data is transferred with odd parity (can be calculated by XOR of all Bits with starting value is 1). Example: 1 XOR bit0 XOR bit1 XOR bit2 XOR bit3 XOR bit4 XOR bit5 XOR bit6 XOR bit7.

Pay attention to the fact that the LSB is send first. The byte marker indicates whether or not this is the last byte.

Byte marker	Bit-0	Bit-1	Bit-2	Bit-3	Bit-4	Bit-5	Bit-6	Bit-7	Odd parity

The byte marker does not need to produce and end-of-data after the checksum byte of the header and repeated header because the header is a fixed length. It does produce an end-of-data marker for the data and repeated data blocks. Below is shown what happens when an end-of-data marker is transmitted (LS) there is simply nothing more send. No more bits follow. When the end-of-data byte marker is used it indicates that the block has finished.

Byte marker = LM	Bit-0	Bit-1	Bit-2	Bit-3	Bit-4	Bit-5	Bit-6	Bit-7	parity

Byte marker = LS	-	-	-	-	-	-	-	-	parity

Leader:
A 10 second leader is written on the tape before recording of the data or program commences. This leader has two functions; first it allows the tape motor to reach the correct speed, and secondly the sequence of short pulses written on the leader is used to synchronize the read routine timing to the timing on the tape. The operating system can thus produce a correction factor which allows a very wide variation in tape speed without affecting reading.

Interrecorded gaps:
Interrecord gaps (2 sec.) are primarily used in ASCII files and their function is to allow the tape motor time to decelerate after being turned off and accelerate to the correct speed when turned on prior to a block read or write. Each inter-record gap is approximately two seconds long and is recorded as a sequence of short pulses in the same manner as the 10 second leader.

Trailer:
The protocol allows the use of trailers. This is just like the leader but then at the end of the file. This is optional and almost never used. The Cassiopei, does not make use of trailers as it does not add any benefits.

1.4 Structure on the tape

When C64 Saves data to tape it normally creates 4 tape blocks. First two are HEADER (always the same length: 202 bytes) and last two are DATA. The header and program are repeated to create the simplest form of error checking, when both versions are identical, then loading must have been successful.

LEADER	: 0x6A00 S's (approx 10 Sec) *
ID	: 0x89 0x88 0x87 0x86 0x86 0x84 0x83 0x82 0x81
HEADER	: 192 bytes (see next chapter)
CHECK	: 1 byte that represents the XOR of the data in this block

LEADER	: 79 waves (consists only of S)
ID	: 0x09 0x08 0x07 0x06 0x05 0x04 0x03 0x02 0x01
HEADER Rep.	: (repeated) 192 bytes (see next chapter)
CHECK	: 1 byte that represents the XOR of the data in this block
TRAILER	: this is not required, but doesn't hurt either

silence (roughly 0.4 seconds)
tape stops, computer waits for user to press space... within 5 seconds...

LEADER	: 0x1A00 S's (approx 2 Sec) *
ID	: 0x89 0x88 0x87 0x86 0x86 0x84 0x83 0x82 0x81
PROGRAM	: the actual program data
CHECK	: 1 byte that represents the XOR of the data in this block

LEADER	: 79 waves
ID	: 0x09 0x08 0x07 0x06 0x05 0x04 0x03 0x02 0x01
PROGRAM Rep.	: the actual program data (repeated)
CHECK	: 1 byte that represents the XOR of the data in this block

*: the Cassiopei uses 1000 pulses, this works fine. This is possible because the Cassiopei is a digital system that does not suffer from motors/tapes that require time to get up to speed or slowing down (slightly oscillating on start or finish). This is great as it greatly reduces the time required for loading our programs. Since the leader sometimes takes up more time then the actual program.

Note:
The above is for the situation that a program is saved using: **SAVE "FILENAME",1**
But when a program is saved using: **SAVE “FILENAME”,1,2** then an "End-of-tape marker" is requested to be written AFTER the program that is saved. In that case the CBM computer writes an additional empty HEADER. This header will only be found if the user tries to read directly after the saved program. So actually it doesn't change the saved program in any way. It justs adds the beginning of a second empty file.

Header						
offset	Bytes used	Cassette buffer memory location				Information
		VIC20, C64	C16, Plus4	C128	PET/CBM series Cass #1	
0	1	0x033C	0x0332	0x0B00	0x027A	File type
1	1	0x033D	0x0333	0x0B01	0x027B	Load address Low
2	1	0x033E	0x0334	0x0B02	0x027C	Load address High
3	1	0x033F	0x0335	0x0B03	0x027D	End address+1 Low
4	1	0x0340	0x0336	0x0B04	0x027E	End address+1 high
5	16	0x0341 0x0350	0x0337 0x0346	0x0B05 0x0B14	0x027F 0x028E	File name
21	171	0x0351 0x03FB	0x0347 0x03F1	0x0B15 0x0BBF	0x028F 0x0339	Unused *(note 1)*

File type	
ID	Information
1	Basic file *(note 2)*
2	Data block (for sequential file)
3	Fixed address file *(note 3)*
4	Sequential file
5	End of tape maker

Note 1:
The “unused” area of the cassette buffer would normally contain all spaces (0x20), but there could also be data. For example, many turbo loaders put part of their code in here. Also the Cassiopei uses this area to load it's loader routines. The C16 and Plus 4 are using filenames of 17 characters instead of 16, this means that the 17th character is located in first byte of the unused area, but the PET's even allow 128bytes?!?!? The Cassiopei simply uses 16 chars, no fuss.

Note 2:
For the VIC20, C64, etc. files are relocatable: the start address is moved to the location pointed to by the zero page locations 0x2B and 0x2C (C64), and the end address is moved by the same amount. This happens when loading a BASIC program with **`LOAD"NAME",1`**
However, this can be overridden by typing: **`LOAD"NAME",1,1`**
In those cases, the start and end addresses provided by the header will always be used. HOWEVER for a PET computer this isn't quite the case. As files AREN'T relocatable. Files will always be loaded to the address as specified in the file.

Note 3:
Fixed address files: the start and end addresses provided by the header will always be used, no matter what the flag ,1,1 (as used in **`LOAD”NAME”,1,1`**) is set or not. This is very useful for auto starting programs as it prevents the loading to any other address then the address specified in the

header itself. HOWEVER this functionality isn't the same on a PET, there it is best to load files with the filetype 1. If you would attempt to load a file with filetype 3 on a PET, then it simply ignores the file (took me half a day to figure that one out...)

1.5 Data on the tape

Now we know what signals there are on a tape and how we should interpret them you may be wondering how the actual data would look like if you would capture all the individual bytes. This paragraph gives an example of a small program and the bytes that are stored onto the tape.

10 PRINT”HOERA HET WERKT”;
20 GOTO 10

The above program results in the following data when saved to tape on a C64:

Header:

byte 000 = 01
byte 001 = 01
byte 002 = 08
byte 003 = 24
byte 004 = 08
byte 005 = 20
byte 006 = 20
byte 007 = 20
byte 008 = 20
byte 009 = 20
byte 010 = 20
byte 011 = 20
byte 012 = 20
byte 013 = 20
byte 014 = 20
byte 015 = 20
byte 016 = 20
byte 017 = 20
byte 018 = 20
byte 019 = 20
byte 020 = 20
byte 021 = 20
byte 022 = 20
byte 023 = 20
byte 024 = 20
byte 025 = 20
byte 026 = 20
byte 027 = 20
byte 028 = 20
byte 029 = 20
byte 030 = 20
byte 031 = 20
byte 032 = 20
byte 033 = 20
byte 034 = 20
byte 035 = 20
byte 036 = 20
byte 037 = 20
byte 038 = 20
byte 039 = 20
byte 040 = 20
byte 041 = 20
byte 042 = 20
byte 043 = 20
byte 044 = 20
byte 045 = 20
byte 046 = 20
byte 047 = 20
byte 048 = 20
byte 049 = 20
byte 050 = 20
byte 051 = 20
byte 052 = 20
byte 053 = 20
byte 054 = 20
byte 055 = 20
byte 056 = 20
byte 057 = 20
byte 058 = 20
byte 059 = 20
byte 060 = 20
byte 061 = 20
byte 062 = 20
byte 063 = 20
byte 064 = 20
byte 065 = 20
byte 066 = 20
byte 067 = 20
byte 068 = 20
byte 069 = 20
byte 070 = 20
byte 071 = 20
byte 072 = 20
byte 073 = 20
byte 074 = 20
byte 075 = 20
byte 076 = 20
byte 077 = 20
byte 078 = 20
byte 079 = 20
byte 080 = 20
byte 081 = 20
byte 082 = 20
byte 083 = 20
byte 084 = 20
byte 085 = 20
byte 086 = 20
byte 087 = 20
byte 088 = 20
byte 089 = 20
byte 090 = 20
byte 091 = 20
byte 092 = 20
byte 093 = 20
byte 094 = 20
byte 095 = 20
byte 096 = 20
byte 097 = 20
byte 098 = 20
byte 099 = 20
byte 100 = 20
byte 101 = 20
byte 102 = 20
byte 103 = 20
byte 104 = 20
byte 105 = 20
byte 106 = 20
byte 107 = 20
byte 108 = 20
byte 109 = 20
byte 110 = 20
byte 111 = 20
byte 112 = 20
byte 113 = 20
byte 114 = 20
byte 115 = 20
byte 116 = 20
byte 117 = 20
byte 118 = 20
byte 119 = 20
byte 120 = 20
byte 121 = 20
byte 122 = 20
byte 123 = 20
byte 124 = 20
byte 125 = 20
byte 126 = 20
byte 127 = 20
byte 128 = 20
byte 129 = 20
byte 130 = 20
byte 131 = 20
byte 132 = 20
byte 133 = 20
byte 134 = 20
byte 135 = 20
byte 136 = 20
byte 137 = 20
byte 138 = 20
byte 139 = 20
byte 140 = 20
byte 141 = 20
byte 142 = 20
byte 143 = 20
byte 144 = 20
byte 145 = 20
byte 146 = 20
byte 147 = 20
byte 148 = 20
byte 149 = 20
byte 150 = 20
byte 151 = 20
byte 152 = 20
byte 153 = 20
byte 154 = 20
byte 155 = 20
byte 156 = 20
byte 157 = 20
byte 158 = 20
byte 159 = 20
byte 160 = 20
byte 161 = 20
byte 162 = 20
byte 163 = 20
byte 164 = 20
byte 165 = 20
byte 166 = 20
byte 167 = 20
byte 168 = 20
byte 169 = 20
byte 170 = 20
byte 171 = 20
byte 172 = 20
byte 173 = 20
byte 174 = 20
byte 175 = 20
byte 176 = 20
byte 177 = 20
byte 178 = 20
byte 179 = 20
byte 180 = 20
byte 181 = 20
byte 182 = 20
byte 183 = 20
byte 184 = 20
byte 185 = 20
byte 186 = 20
byte 187 = 20
byte 188 = 20
byte 189 = 20
byte 190 = 20
byte 191 = 20
byte 192 = 04

Data:

byte 000 = 19
byte 001 = 08
byte 002 = 0A
byte 003 = 00
byte 004 = 99
byte 005 = 22
byte 006 = 48
byte 007 = 4F
byte 008 = 45
byte 009 = 52
byte 010 = 41
byte 011 = 20
byte 012 = 48
byte 013 = 45
byte 014 = 54
byte 015 = 20
byte 016 = 57
byte 017 = 45
byte 018 = 52
byte 019 = 4B
byte 020 = 54
byte 021 = 22
byte 022 = 3B
byte 023 = 00
byte 024 = 22
byte 025 = 08
byte 026 = 14
byte 027 = 00
byte 028 = 89
byte 029 = 20
byte 030 = 31
byte 031 = 30
byte 032 = 00
byte 033 = 00
byte 034 = 00
byte 035 = 78

2 Digital flywheel technology

During the development of the original Cassiopei there was no reason to implement this, but then a user of TAP-files reported a strange issue with a C64 game called "Turbo Charge". When this game was loaded, a strange bar appeared in the middle of the screen and loading did not continue. After some investigation it appeared that the game's fastloader routine was switching the motor-signal off for a very small time. Normally this has no significant impact on the speed of a real tape, because of the flywheel present in the tape transport mechanism, therefore loading would not be disturbed. But the Cassiopei could stop .TAP playback instantly which caused the game's loader to fail.

Now normally you would say that the game has a badly written loader. But saying so does not solve the problem. So the only way to fix it was to make the Cassiopei behave more like a real tape. In other words when the motor signal goes low, the Cassiopei no longer stops the playback, it will wait for a few extra milliseconds and if the signal is still low then stops the tape. Making it impossible to stop the digital TAP file playback instantly. The small routine that realizes this “delay” can be compared to a flywheel. When a flywheel is up to speed, it takes time to slow down. Due to it's mass it holds so much energy that it simply can't stop instantly. Hence the name “digital flywheel”.

This sollution was required in the original Cassiopei, but in the Cassiopei v2.0 this now longer aplies, the stopping and starting of the tape-signal from the Cassiopei is much more buffered and cannot stop instantly, therefore the need for an additional digital flywheel is no longer there.

Now these kind of “problems” are very rare and I have not heard or read about another game with the same problem. But these kind of problems are interesting in their own way, first there is the troubleshooting stage. Determining what the problem is and reproducing it. Second is solving it. But third is the mystery of the origin of the problem. Why would the programmer of this tape fastloader have chosen to program it like this or was he/she even aware of this problem?

3 How the fastloader is loaded

As the principle of operation is for most non-PET computers is the same, this will only be about the C64. The C64 has in it's memory a few locations that are not really used. The first is the unused section in the header of the standard tape protocol. This is 171 bytes that is unused but required for loading/saving. It is stored into memory locations 0x0351-0x03FB. The area 0x02A7-0x02FF (89 bytes) is unused and conveniently close to the Basic warm start vector, located at 0x0302-0x0303. So we write a program that is stored in memory in 0x02A7-0x02FF and further in 0x0351-0x03FB. However when we make the small part of our program 4 bytes bigger then 89 bytes, we will write into the “Print BASIC error message vector” and the “BASIC warm start vector”, we'll fill the “Print BASIC error message vector” with the default value of 0x8B and 0xE3 and we fill the “BASIC warm start vector” with the start address of our program. Notice that 0x0302 holds the low byte of that address and that 0x0303 holds the high byte.

When the C64 has finished loading our program stored in partially free memory and partially in the cassette buffer will be called automatically, because the “BASIC warm start vector” holds the start address of our program. And then the fun begins, using the CPIO protocol the Cassiopei is requested to transfer the final program. This will load the program byte by byte until finished. The the program can be executed. But first we must make sure that the following registers are set:

0x2D, 0x2F, 0x31, 0xAE must contain the low byte of the end address of our just loaded program
0x2E, 0x30, 0x32, 0xAF must contain the high byte of the end address of our just loaded program

Also memory area 0x02A7 – 0x02FF must be cleaned up. Because the C64 does not use this area and fills this area with 0x00 on reset. Some programs expect these values. And if there are other values sotred then 0x00 in this area, the executed program might behave unexpected. For instance the Final Cartridge III uses 0x02AA for a flag that keep track of the basic TRACE function. And I've seen a demo flicker because of the values in this memory area. So clean using 0's and everything would be OK.

Although currently it seems to work without the suggestions below (source: Gideon Z.):
to set store the value 0x40 to zero page address 0x90 (status byte, 0x40 means end of file)
to set store the value 0x01 to zero page address 0xBA (current device number, 1=datasette)
to set store the value 0x00 to zero page address 0x35
to set store the value 0xA0 to zero page address 0x36
to set store the value 0x00 to zero page address 0x02
set stack pointer to 0xFB (LDX #$FB followed by TXS)

Then we can finally start executing our program. And we do that by typing RUN... well not really. We poke “R”,“shift+U”,“<return>” (the short notation for RUN<return>) into the keyboard buffer. Then we set the zero page address 0xC6 to 3 because that is the number of characters we've just put into the keyboard buffer. Then we release the interrupts again (using CLI command) and we jump to the basic input loop vector (0xA480). That will notice the run command in the keyboard buffer and execution begins just as if we manually typed RUN<return>.

We use the return trick simply because this proved to be the most effective, typing run appears to set some basic flags we otherwise needed to set in our loader program at the cost of many bytes of programming space. Which is something that we simply cannot afford to do in the limited area 0x02A7-0x02FF and 0x0351-0x03FB.

The PET/CBM 2000, 3000, 4000, 8000 models work differently then the C64 and therefore they cannot use the same method of auto-starting, in order to auto-start, the Cassiopei needs to modify the stack of the PET/CBM computer, this is a very tricky exercise and is therefore not implemented. Modifying the stack requires absolute knowledge of the system the program is running on and because there are so many types and variations regarding the contents of the stack of these different models this functionality will never be developed. Sorry, you must load with shift+RUNSTOP (BASIC 2) or type RUN (BASIC 4)

4 CPIO protocol

The CPIO (Cassette Port Input Output) protocol is a protocol designed to interface a slave device to a CBM computer (it started with a C64) using the cassette port. The intention of the CPIO is to have a fast synchronous and bi directional (half duplex) protocol to transfer data between the CBM computer and it's slave.

CPIO is defined as a synchronous protocol to make sure that the VIC (the C64's video processor) can continue to operate without causing problems of disturbing the timing of the data transfer protocol. In other words during the fast data transfer of the Cassiopei there is no need to disable the screen (to prevent the VIC from interrupting the timing). Disabling of the screen is known to be used for LOADING and SAVING from the datasette during normal use. The CPIO protocol does not require disabling the screen. So You can have a high data transfer rate without errors and with all the fancy things on your screen visible.

The slave device can be all sort of devices. Using the CPIO in combination with a fast micro-controller connected to the cassette port of a CBM computer, the number of possible applications becomes unlimited. For example an AD-converter, simple digital IO, fast serial port UART, counter, etc. But also file loading becomes possible. Important to know is that the CPIO device must be capable of transferring data over the cassette port as if the device is a datasette. This makes it possible to have the application as well as the software for that application into one device. Because the CPIO is a fast protocol, the application software can be loaded according a 2 step procedure. The user simply types LOAD and the CBM computer initiates a load sequence as if it would load from a real datasette. The device loads a small piece of software into the CBM which incorporates the CPIO protocol. Because this code will auto start the user does not have to do anything anymore until the application becomes fully active.

The CBM computer is the slowest device and therefore determines the max. speed of the CPIO protocol.

4.1 The CPIO protocol's signals (synchronous 8-bit)

Master : CBM computer
Slave : Cassiopei or any other device using the CPIO protocol

ATTENTION:
This signal indicates to the slave the desired state of communication. When this signal goes high, the slave is expected to respond with READY = low as soon as possible. When this signal goes low, the slave must finish it's current byte read/write action and then stop communication.

This signal is actually the motor power line. It is driven by a power transistor, which is relatively slow. Also this line should be loaded with a few hundred ohm in order to make the high to low transition faster (820 Ohm is sufficient, this will dissipate approx. 1/8 Watt when the ATTENTION is high, it is not much compared to the current of the motor, but it is enough to drain the voltage quickly). Because this signal is so slow, we must clear this signal before we send or read the last byte. Deactivating attention takes approx 150uSec on a C64. By deactivating the attention signal just before the last byte is send/received the attention signal will become deactivated in the middle of the byte (because it is so slow). What means that it is stable when the byte has finished. Which is exactly the moment that the attention signal is being sampled for it's state by the connected slave device. This is the best way of working around this slow signal without compromising the data transfer speed.

Although the Cassiopei can respond (or acknowledge) a rising attention signal, it cannot respond to a falling attention signal. Therefore the CBM cannot detect if the Cassiopei has detected the dropping of the attention signal. To make sure this happens we must wait for a very long time before we make attention rise again. This time is called the back-off time. It should be at least 50ms.

low = no communication
high = communication with slave
default = low

CLOCK:
This signal makes the data transfer synchronous required for high speed communication. The CLOCK line is controlled by the master and the CLOCK line will go low as soon as the slave responds with a READY = low.

low = data setup (slave/master are expected to write directly after the falling edge)
high =data stable (slave/master are expected to read directly after the rising edge)
default = high

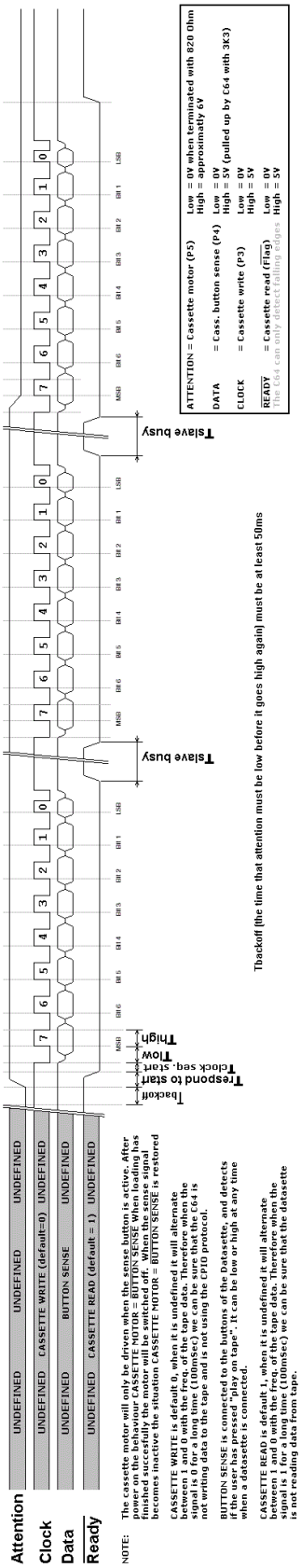
DATA:
This line is a bi-directional line. It is used to transfer the data between the master and slave or vice versa. The master determines the direction of communication. When the data direction is changed is described in the command definition (see the chapter CPIO message definitions).

low = logical '0'
high = logical '1'
default = high

READY (active low):
This signal indicates (or acknowledges) that the slave is ready to communicate, as long as this signal is high, the master will not start with lowering the CLOCK line. Please note that the C64 can only detect the falling edge of this signal (READY is connected to the Flag input of the CIA, which is an edge negative triggered input pin).

low = slave is ready for communication
high = slave is busy
default = high

This page shows the signaling of the CPIO (the pinout described are for the C64).



NOTE: The cassette motor will only be driven when the sense button is active. After power on the behaviour CASSETTE MOTOR = BUTTON SENSE When loading has finished succesfully the motor will be switched off. When the sense signal becomes inactive the situation CASSETTE MOTOR = BUTTON SENSE is restored

CASSETTE WRITE is default 0, when it is undefined it will alternate between 1 and 0 with the freq. of the tape data. Therefore when the signal is low about one of (0msec) we can be sure that the C64 is not writing data to the tape and is not using the CPIO protocol.

BUTTON SENSE is connected to the buttons of the Datasette, and detects if the user has pressed 'play on tape'. It can be low or high at any time when a datasette is connected.

CASSETTE READ is default 1, when it is undefined it will alternate between 1 and 0 with the freq. of the tape data. Therefore when the signal is low about one of (0msec) we can be sure that the datasette is not reading data from tape.

5 CPIO messages definitions

The Cassiopei is a device that has many functions. Therefore the master (the Commodore computer also referred to as CBM) needs to tell the Cassiopei in which mode it should operate or what task it should perform. Because it can do so many, the CBM (or actually the software of the CBM) must be aware of the exact flow of data.

Therefore the CPIO protocol demands that the first byte in each session is the mode-byte. This byte describes the slaves mode of operation. All bytes that follow can be read or write. The exact sequence of events required for each command is defined in the command description of the following chapters.

5.1 Command: PRG Load

This function is intended for loading PRG files and nothing else. It is used when loading PRG files (games and programs) and the Cassiopei menu program.

Load PRG file CPIO_LOAD (0x00) read PROGRAM from slave, this mode is useful as it is faster then the standard tape protocol										
Byte	**Data direction**	**Byte type**	**Information**							
1	Master write	mode byte	0	0	0	0	0	0	0	0
2	Slave write	data	Low byte of file's last byte address in C64 RAM							
3	Slave write	data	High byte of file's last byte address in C64 RAM							
4	Slave write	data	Low byte of LOAD address of file's first byte's address in C64 RAM							
5	Slave write	data	High byte of LOAD address of file's first byte's address in C64 RAM							
6-...	Slave write	data	The file to be loaded							
End	Slave write	data	This byte may be discarded, it is just a dummy in order to keep our 6502 loops simple, since this last byte is a dummy, we do not need to process it and the loading loop does not need to check for it.							

For the VIC-20, it is sometimes required to load to a different address other then specified in the file. In this case the function below make that partially possible. It still requires the software in the VIC-20 (menu program) to have a decicated loader that overrules the file's load address.

5.2 Command: DataFile related commands

This function is intended for the reading/writing of raw data from/to a file, for example: PETSCII video files or disk images. In order for this command to work, it requires some filename and path details. These can be send trough the CPIO_PARAMETER command OR by using the browsing menu using the command CPIO_BROWSE. The first requires the filename to be known and in the same path as the current file (practical if you write a game that requires external data from the cassiopei). The latter allows the user to select a file that could be anywhere on the filesystem.

Datafile path information CPIO_PARAMETER (0xFF) optional parameters for the CPIO_DATALOAD command										
Byte	**Data direction**	**Byte type**	**Information**							
1	Master write	mode byte	1	1	1	1	1	1	1	1
2	Master write	data	1st character of filename							
3	Master write	data	2nd character of filename							
4	Master write	data	3rd character of filename							
...	Master write	data	Etc.							

The CPIO_DATAFILE_OPEN command is used to open the file as indicated by browsing OR as defined by the absolute filenamepath in the CPIO_PARAMETER command.

Open file for reading/writing CPIO_DATAFILE_OPEN (0x80) open data file										
Byte	**Data direction**	**Byte type**	**Information**							
1	Master write	mode byte	1	0	0	0	0	0	0	0
2	Master write	mode	0=read, 1=write, 2=append							
3	Slave write	data	0 = error, for instance file not found 1 = ready for data transfer 2 = file already exists							
4	Slave write	data	MSB of filesize (a 4 byte value allows for a max filesize of 4GByte)							
5	Slave write	data	.SB of filesize							
6	Slave write	data	.SB of filesize							
7	Slave write	data	LSB of filesize							

The Cassiopei keeps sending data when the end of the file is reached, the Cassiopei will send 0's. Therefore the program that processes the file must be aware of the size of the file and therefore be able to know when to stop reading. This is done to keep the transfer simple, this way there is no reserved character/byte that indicates the end of the file. Therefore keeping the overhead as small as possible for the datatransfer.

The CPIO_DATAFILE_SEEKPOS command is used to set the datapointer to a required position, therefore allowing the user to skip data or to go back, this may be useful for playing sample loops through CPIO.

Set datapointer to any position in the file CPIO_DATAFILE_SEEKPOS (0x81) modify datapointer										
Byte	**Data direction**	**Byte type**	**Information**							
1	Master write	mode byte	1	0	0	0	0	0	0	1
2	Master write	data	MSB of filesize (a 4 byte value allows for a max filesize of 4GByte)							
3	Master write	data	.SB of filesize							
4	Master write	data	.SB of filesize							
5	Master write	data	LSB of filesize							
6	Slave write	data	0 = error, requested data position exceeds filesize 1 = ready for data transfer							

The CPIO_DATAFILE_READ command is used to read a byte from the file. The user may read as many bytes as the file allows. When attention is dropped, simply re-send the CPIO_DATAFILE_READ command and continue reading.

Read data from the file CPIO_DATAFILE_READ (0x82) read data										
Byte	**Data direction**	**Byte type**	**Information**							
1	Master write	mode byte	1	0	0	0	0	0	1	0
2	Slave write	data	Data from the file							
3	Slave write	data	Data from the file							
...	Slave write	data	Data from the file...							

The CPIO_DATAFILE_WRITE command is used to write data to the file. The user may write as many bytes as the file allows, the max filesize is limitted by the filesystem (FAT32) which is 4GByte. This size is virtually infinite for a 1MHz 8-bit computer. When attention is dropped, simply re-send the CPIO_DATAFILE_WRITE command and continue writing.

Write data to the file CPIO_DATAFILE_SEEKPOS (0x83) write data										
Byte	**Data direction**	**Byte type**	**Information**							
1	Master write	mode byte	1	0	0	0	0	0	1	1
2	Master write	data	Data to the file							
3	Master write	data	Data to the file							
...	Master write	data	Data to the file...							

Close the file CPIO_DATAFILE_CLOSE (0x84) close file (a file that is not closed after writing corrupts the filesystem, therefore always close when writing has finished.)										
Byte	**Data direction**	**Byte type**	**Information**							
1	Master write	mode byte	1	0	0	0	0	1	0	0
2	Master write	data	Dummy (value is not used in any way, but allows for closing the CPIO communication in a conventional manor)							

5.3 Command: menu

This routine is for navigating through the menu of the Cassiopei. By handling all the menu related action inside the Cassiopei itself there is no longer a need to define registers or methods for modifying them. Just a menu that is controlled using a simple set of commands. The menu itself is build in the menu-screen buffer inside the Cassiopei and the menu-command returns this buffer. The caller only needs to draw this information directly to the screen saving him/her the effort of analyzing/processing the data.

CPIO Menu (request+response) CPIO_MENU (0x04) Request a Cassiopei menu action										
Byte	**Data direction**	**Byte type**	**Information**							
1	Master write	mode byte	0	0	0	0	0	1	0	0
2	Master write	mode byte	Menu action: *(main menu related actions)* 0x00 = refresh (only request the current menu screen data) 0x01 = previous (perform a previous action inside the main menu) 0x02 = select (perform a select action inside the main menu) 0x03 = next(perform a next action inside the main menu) 0x04 = previous page(perform multiple previous actions) 0x05 = next page(perform multiple next actions) 0xFF = reset the menu (force the menu to the begin state)							
3	Master write	mode byte	Width of the screen in characters (max=80)							
4	Master write	mode byte	Height of the screen in characters (max=25)							
5	Slave write	data	Status: 0x00 = menu no longer active (the exit function has been selected) 0x01 = menu active							
6	Slave write	data	First byte read from menu-screen buffer							
..	...	...	...							
n	Slave write	data	n-th byte read from the menu-screen buffer							

5.4 Command: file select

This routine is for navigating (browsing) through a list of files and allows the user to select a file. The beauty of this command is that it (after a successful file selection) it will do a CPIO_PARAMETER command automatically. Meaning that the file name will be put into the memory of the Cassiopei. Therefore there is no need to do a CPIO_PARAMETER command before opening the selected file. This means that the filename itself does not need to be transported to the CBM and back again, preventing all sorts of problems that could arise due to the otherwise required conversion from ASCII to PETSCII and back to ASCII again. The file offset value is always set to 0 (the beginning of the file).

CPIO file select (request+response) CPIO_BROWSE (0x05) Request Cassiopei file browsing and prepare for file opening										
Byte	**Data direction**	**Byte type**	**Information**							
1	Master write	mode byte	0	0	0	0	0	1	0	1
2	Master write	mode byte	Browser action: *(browse related actions)* 0x00 = refresh (only request the current browser screen data) 0x01 = previous (perform a previous action inside the browser) 0x02 = select (perform a select action inside the browser) 0x03 = next(perform a next action inside the browser) 0xFF = reset the browser (force browser to the begin state)							
3	Master write	mode byte	Width of the screen in characters (max=80)							
4	Master write	mode byte	Height of the screen in characters (max=25)							
5	Slave write	data	Status: 0x00 = browser no longer active (the exit function has been selected) 0x01 = browser active							
6	Slave write	data	First byte read from browser/menu-screen buffer							
..	...	...	...							
n	Slave write	data	n-th byte read from the browser/menu-screen buffer							

5.5 Command: Read / Write settings

The Cassiopei has all sorts of settings that need to be read/stored by the Cassiopei itself and/or the programs using it. The settings are stored on settings file on the SD-card. Although there is really no need for this command, it does exist. Please keep writing the settings to a minimum, because of wear of the SD-card.

Read/write SETTINGS CPIO_RW_SETTINGS (0x0F) read/write setting from/to SD-card settings file										
Byte	**Data direction**	**Byte type**	**Information**							
1	Master write	mode byte	0	0	0	0	1	1	1	1
2	Master write	R/W	0x00 = read setting 0x01 = write setting							
3	Master write	address	0x00 = mode 0x01 = reserved 0x02 = computermodel 0xFF = write all settings to SD-card file							
4	Slave write (read setting)	data	Data read from settings							
	Master write (write setting)	data	Data written to setting Note: in case of “current file” and “write all settings to SD-card file”current file there is no data required, but a value should be written in order to complete the command. Therefor in those situations this byte should have the value 0)							

If a register is read that cannot be read then 0 is returned.
It is not possible to read the current filename using this command. In order to do so use directory reading command instead, the first entry returned will be the current file.

5.6 Command: Simulate button

This function is intended for the CPIO menu program (sometimes referred to as configuration program) and it not recommended for other purposes.

Simulate button CPIO_BUTTON (0x07) read PROGRAM from slave, this mode is useful as it is faster then the standard tape protocol										
Byte	**Data direction**	**Byte type**	**Information**							
1	Master write	mode byte	0	0	0	0	0	1	1	1
2	Master write	function	Indicate what button function needs to be simulated							

Function byte		
Value	**Simulated button**	**Information**
0x01	PLAY	This value simulates that the PLAY button is pressed
0x02	MENU	This value simulates that the MENU button is pressed
0x11	PLAY-D	This value simulates that the PLAY button, but with a fixed delay of 3sec.

5.7 Command: NTP clock

This function is intended to transfer the time (inside the Cassiopei as gathered by the NTP service) to the CBM computer. The first part is to set the time of the jiffy clock (the software clock of the CBM computer), the second part is for transferring the time to the CIA and is therefore in BCD format.

CPIO NTPCLOCK (request+response) CPIO_NTPCLOCK (0x10) Request the NTP clock values (time and date)										
Byte	**Data direction**	**Byte type**	**Information**							
1	Master write	mode byte	0	0	0	1	0	0	0	0
2	Slave write	data	MSB of jiffy clock (3-Byte value, ready for copying into the Jiffy-clock)							
3	Slave write	data	.SB of jiffy clock							
4	Slave write	data	LSB of jiffy clock							
5	Slave write	data	Hour (BCD value between 00-12) (bit 7 indicates AM/PM)							
6	Slave write	data	Minute (BCD value between 00-59)							
7	Slave write	data	Second (BCD value between 00-59)							
8	Slave write	data	Year							
9	Slave write	data	Month							
10	Slave write	data	Day							
11	Slave write	data	Hour							
12	Slave write	data	Minute							
13	Slave write	data	Seconds							

5.8 Command: FMV

This function is intended to transfer large amount of data through the userport. This functionality is intended to offer “full motion video” to the commodores with a userport (*Video for the masses, not the classes*).
The only reason this could work is because that the data is transferred almost directly to the required locations using a highly optimized unrolled loop. If the loop wasn't unrolled, the CPU would be spending more time on checking the loop itself and incrementing the pointers. Audio and video are interleaved and the definition of the file itself is not described in this document.

CPIO FMV (request+response) CPIO_FMV (0xF0) Request full motion video data stream										
Byte	**Data direction**	**Byte type**	**Information**							
1	Master write	mode byte	1	1	1	1	0	0	0	0
2	Master write	function	Details about how to play							
3	Master write	data	Optional details depending on the given function (it will hold the index value when the function Play FMV indexed is given).							

Function byte		
Value	**Simulated button**	**Information**
0x00	Play FMV file	Start playback of the FMV file
0x01	Restart FMV file	Start the last played file (this will fail if no file has been played before)
0x02	Play FMV indexed	Start playback of other FMV file in same folder, filename is only an index

FMV game files are setup with the following name convention:
All files that belong to the same game need to be stored in the same folder, do not store anything else in this folder. As it would only be confusing. All files have a unique number (index) ranging from 0 to 255, with the exception of the first file to be loaded, that one has the full name of the game. So there are 257 different possible FMV videofiles possible in a single FMV game.

Now technically, the game programmer can choose any number in any sequence, but it would be a good idea to use the same system for all of your games. A good system is that of 1 filename (game.dat) and a set of file ranging from 000.dat to 255.dat. This would mean that you have the following structure on the SD-card:

All game data is stored in a folder named “data” which is located in the same path as the game.prg.

SD-card / computermodel / … /game/game.prg		← the .PRG file to play the game with
SD-card / computermodel / … /game/data/	game.dat	← short intro/splash screen
	000.dat	← the game menu
	001.dat	← game level
	. . .	
	255.dat	← short exit splash screen

The idea is that the game is loaded by starting the splash screen (the only file with a real name).

This can be a short video, explaining the game or just some credits to show your company logo (like: the famous “EA sports… it's in the game” as could be heard on the nintendo64 games of the 90's). This video could be made (un)skip-able depending on your wishes as a game programmer. However when this video is stopped or ended, it should go to the file 000.dat, which is the games menu.

From the menu the player can choose and this could results in going to another video, for instance 001.dat (the first level of the game). From here we can jump to another file (depending on the game) or when the file ends, we can go to the menu again.

From the menu the player should also be able to select “exit” this should result in loading of the file 255.dat. So this way the user get a nice exit screen to shutdown the game in a nice way.

5.9 Command: Telnet client

This function is intended to let the CBM computer communicate with a telnet based BBS.

Before a telnet connection can be made the server URL and PORT must be specified. This is done using the CPIO_PARAMETER command. URL and PORT are send in one large string in the following format: <URL>:<port>
for example: borderlinebbs.dyndns.org:6400

Datafile path information CPIO_PARAMETER (0xFF) optional parameters for the CPIO_DATALOAD command										
Byte	**Data direction**	**Byte type**	**Information**							
1	Master write	mode byte	1	1	1	1	1	1	1	1
2	Master write	data	1st character of URLand PORT							
3	Master write	data	2nd character of URLand PORT							
4	Master write	data	3rd character of URLand PORT							
...	Master write	data	Etc.							

Directly after sending the URL and PORT string, the CPIO_TELNET_CLIENT command with a connect request must be given in order to use the string as specified by CPIO_PARAMETER.

The CPIO_DATAFILE_OPEN command is used to open the file as indicated by browsing OR as

CPIO TELNETCLIENT (request+response) CPIO_TELNET_CLIENT (0xF1) Request telnet communication											
Byte	**Data direction**	**Byte type**	**Information**								
1	Master write	mode byte	1	1	1	1	0	0	0	1	
2	Master write	function	function								
3	Master r/w	data	Optional details depending on the given function								

Function byte		
Value	**Simulated button**	**Information**
0x00	Initialize	<undefined>
0x01	Status	This function will check the connection status **Master will respond with:** write: 0=connection lost, 1=connected
0x02	Connect	This function try to connect to a specified telnet server. **Master will respond with:** write: 0=failed, 1=connected
0x03	Disconnect	This function will disconnect from the connected telnet server. **Master will respond with:** write: 0 (dummy value)
0x04	Send data	This function will send data to the telnet server. **Master will respond with:** read: number of bytes to be send (1-250) read: data read: data (if available) etc.
0x05	Receive byte	This function will get the data as send by the telnet server. **Master will respond with:** write: data available (0=no data, 1-255=data) write: data byte (if available) write: data byte (if available) etc. write: dummy byte (mainly used to end CPIO communication)