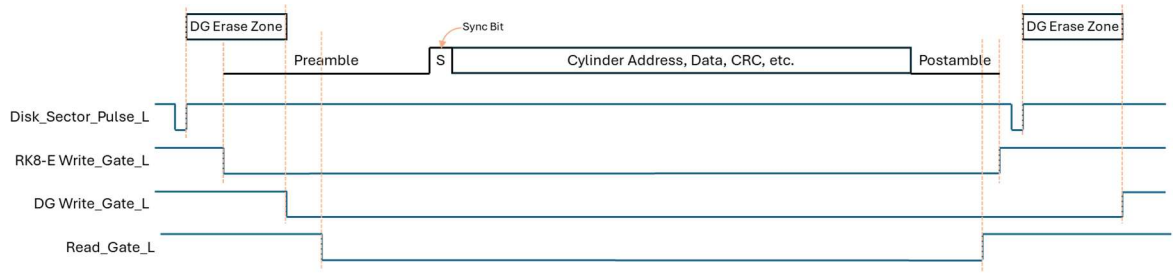
Controller-independent drive emulator data format

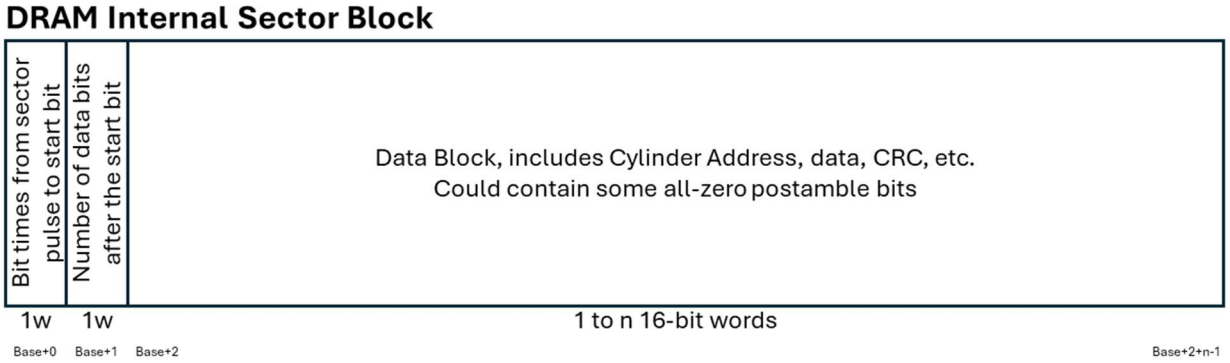
Challenges for the previous emulator architecture:

- Having the drive use controller-specific fields made it difficult to accommodate different controllers. It was necessary to know details of the timing of the disk bus signals for different controllers.
- The timing of some control signals, such as the Write Gate signal on Data General controllers, extend past the sector being written or read which was a complication for the emulator but solvable.

All disk formats in this family of drives and controllers have an all-zero preamble, a single one bit for the sync bit, some number of data bits stored in each sector which includes various fields in addition to data, and an all-zero postamble. Data General controllers keep the Write Gate signal asserted past the sector mark of the next sector.



The controller-independent format has the following data structure for each sector in DRAM and also each sector on the microSD card.



By referencing time from the sector pulse, the emulator will always output the start bit and data block at the correct time. It's just like bits on the surface of the disk platter.

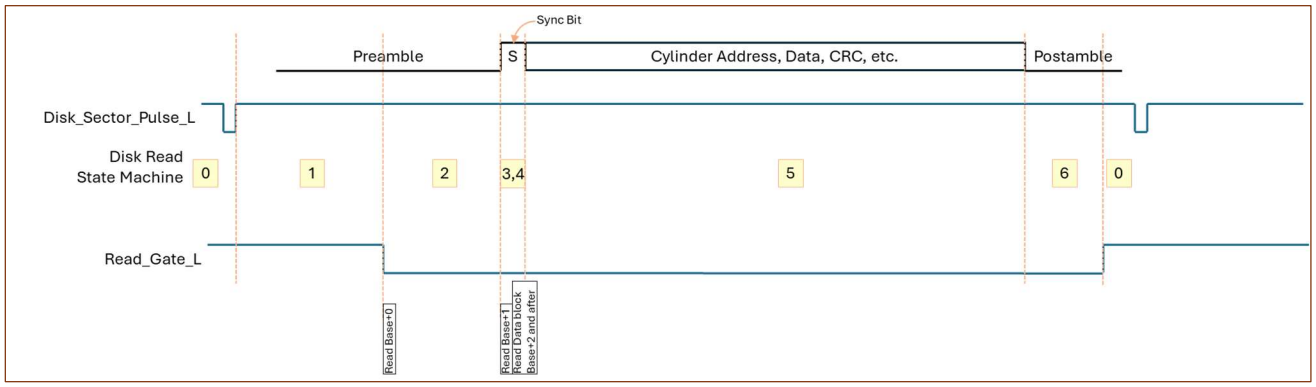
The fields are as follows:

- **Bit times from sector pulse to start bit** – address Base+0, the number of bit times from the leading edge of the sector pulse to the start bit that follows the preamble. Since the preamble is an all-zero field, the emulator needs only to count the number of zero bits prior to the start bit and record the count in this field.
- **Number of data bits after the start bit** – address Base+1, a 16-bit count of the number of bits in the following Data Block. The emulator knows that any bits following the Data Block are all-zero which is either all of the postamble of the last bits of the postamble.

- **Data block** – from addresses Base+2 up to Base+2+n-1, a block of 1 to n words which is the data for a sector on the disk which includes the Cylinder Address, sector data, and CRC. This field could also contain some of the postamble bits when sectors are written by the computer's disk controller. This is the same data that is presently stored in the DRAM and on the microSD card disk image file.

It is helpful to describe the events that occur during disk read and write operations to be able to implement these functions in FPGA logic.

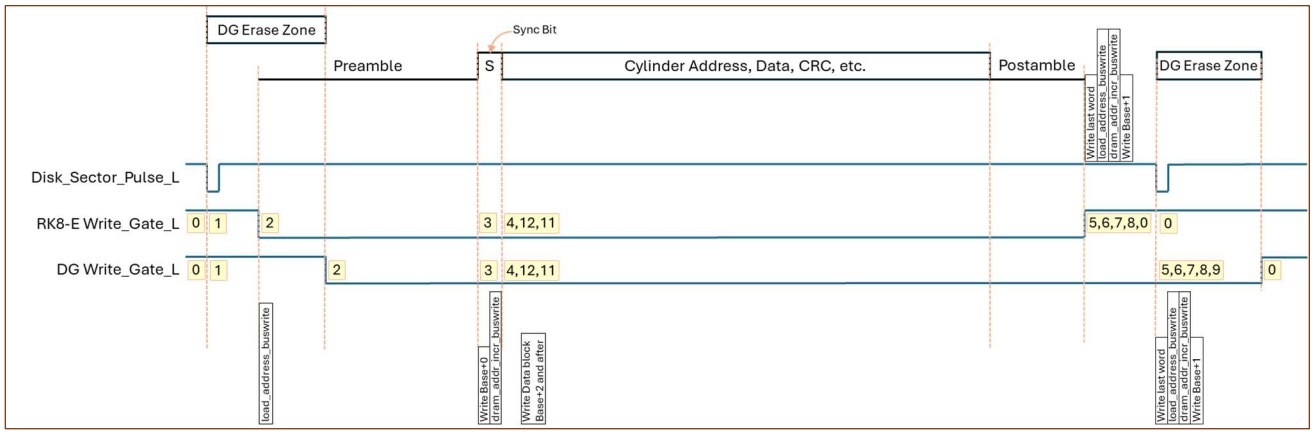
Disk Read Operation:



The following sequence of events occurs when the disk controller reads a sector from the disk:

1. The Emulator generates a periodic sector pulse.
2. At the leading edge of the sector pulse, the Emulator internally loads the fieldbitcount counter to 1, which tracks the number of bit times since the sector pulse. The Emulator begins to increment the fieldbitcount counter once per bit time.
3. If the controller asserts the Read Gate signal, then the Emulator reads the first word of the **DRAM Internal Sector Block** from DRAM at address Base+0 which is the value: "Bit times from sector pulse to start bit" and saves it in cthreshold. The fieldbitcount counter continues to increment once per bit time. If there is no read operation (Read Gate doesn't go active after the sector pulse) and the next sector pulse arrives without having Read Gate asserted, then the fieldbitcount counter is loaded with 1 again at the leading edge of the sector pulse. This happens often.
4. The Emulator detects when the fieldbitcount counter is equal to cthreshold (which was read from Base+0 in step #3) and outputs the Start Bit at the proper clock/data phase time on the serial read data signal, BUS_RD_DATA_L.
5. The Emulator reads the second word of the **DRAM Internal Sector Block** from DRAM at address Base+1 which is "Number of data bits after the start bit" and saves it in cthreshold. This is to know when to stop the process of reading from DRAM and outputting serial data to the Disk Bus. The fieldbitcount counter is reset to 1 because it now keeps track of how many bits are transferred following the Sync Bit.
6. The Emulator now outputs serialized data read from the Data Block in the DRAM starting at address Base+2. The fieldbitcount counter is incremented once per bit time. If the fieldbitcount counter is equal to cthreshold (which was read from Base + 1 in step #5) then the emulator advances to step #7. If Read Gate goes inactive before all bits of the Data Block have been read from DRAM and serialized, then reset the data read state machine to its starting state.
7. While Read Gate is still in the active state, the Emulator outputs all-zero data to the Disk Bus until Read Gate goes inactive.
8. When Read Gate goes inactive, then reset the data read state machine to its starting state.

Disk Write Operation:



The following sequence of events occurs when the disk controller writes a sector to the disk:

1. The Emulator generates a periodic sector pulse.
2. At the leading edge of the sector pulse, the Emulator internally loads the fieldbitcount counter to 1, which tracks the number of bit times since the sector pulse. The Emulator begins to increment the fieldbitcount counter once per bit time. Prior to receiving the composite write data-clock signal from the controller, the emulator uses the internal bit time reference signal, clkenbl_read_bit, as a bit time reference.
3. The controller asserts the Write Gate signal.
4. The Emulator decodes serial data on BUS_WT_DATA_CLK_L to recover clock and data.
5. The emulator continues to increment the fieldbitcount counter but now uses the bit enable signal from the decoded BUS_WT_DATA_CLK_L signal as a bit-time reference.
6. When the Sync Bit is detected on the decoded BUS_WT_DATA_CLK_L, the Emulator writes the value in the fieldbitcount counter (which is the bit count since the leading edge of the index pulse) into the first word of the **DRAM Internal Sector Block** at Base+0. The Emulator sets the fieldbitcount counter to 1 which will now be used to count the number of bits that follow the Sync Bit.
7. Emulator decodes serial data on BUS_WT_DATA_CLK_L and writes data words to the **DRAM Internal Sector Block** area called Data Block which begins at Base+2.
8. When the Controller de-asserts Write Gate, the Emulator writes the last word or partial word of decoded data to the Data Block and then writes the value in the fieldbitcount counter to the second word of **DRAM Internal Sector Block** at Base+1 and the internal state machine returns to its starting state. If the Emulator asserts the periodic sector pulse prior to the inactive edge of Write Gate, then the Emulator interprets this event as the end of the sector data and the Emulator writes the last word or partial word of decoded data to the Data Block and then writes the field_bit_count counter to the second word of **DRAM Internal Sector Block** at Base+1 and the internal state machine goes to a state where no more data is written to DRAM and it waits for Write Gate to go inactive. In the event where Write Gate goes inactive following the assertion of the sector pulse, the internal state machine returns to its starting state when Write Gate goes inactive.

The file size on microSD is only two words per sector larger than the previous rk05 file format size, so the load and unload times are about the same.

Essential fields needed in the microSD file header to support the functions described above are:

- Number of Cylinders
- Number of Sectors per Track
- Number of Heads
- Microseconds per sector
- Bit rate

The Bit Rate field is the only controller-specific information needed by the Emulator.

Need a format-specific converter utility for each disk controller type to convert from SIMH to the emulator file format. Maybe this is only needed for PDP-8 and PDP-11. For less-commonly-used formats it is probably sufficient to have a utility to create an empty (all-zero data) file with header parameters in the list above, and the lengths of data blocks need to only be approximate given the data rate and number of sectors per track. The emulator disks can be formatted on each machine or users can develop their own SIMH converter for other formats using the source code for the PDP-8 utilities.

rke File Format Description:

The file on the microSD card has a file extension of “rke”. The emulator RUN/LOAD switch is toggled to the RUN position, the emulator searches for the first rke file that it finds and loads that file into the emulator’s DRAM. The rke file consists of a group of header fields followed by a group of sector data blocks. The sector data blocks can vary in size slightly. This is determined by the original lengths of each block when the microSD file was created and the width of the write gate pulse produced by the disk controller that may have written new data to the disk while it was loaded in the drive.

The file consists of Header + Data Blocks. For 16-sector PDP-8 there are 203 * 16 * 2 = 6496 blocks. For 12-sector PDP-11 there are 203 * 12 * 2 = 4872 blocks.

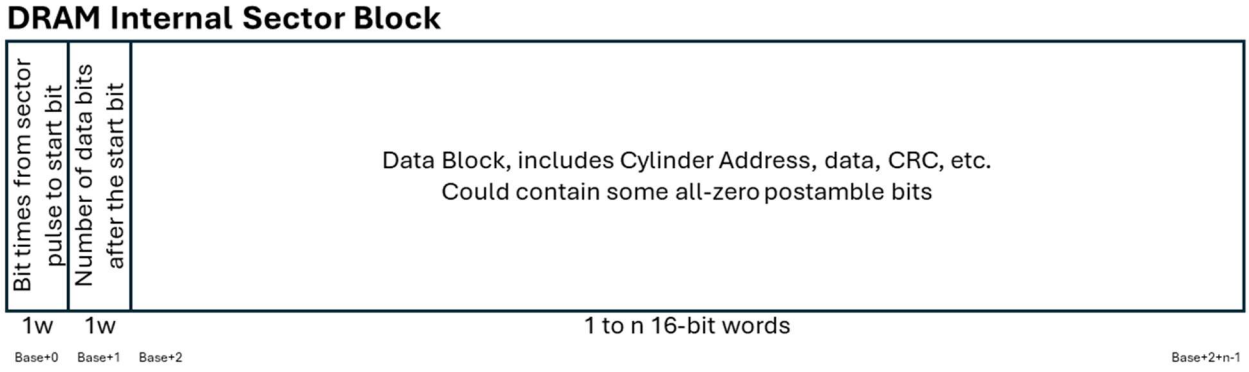
The rke file header consists of the following fields:

- char imageName[11];
- char imageDescription[200];
- char imageDate[20];
- char controller[100];
- int bitRate;
- int numberOfCylinders;
- int numberOfSectorsPerTrack;
- int numberOfHeads;
- int microsecondsPerSector;

Following the header there is a group of Sector Blocks. The number of Sector Blocks is:

$$\text{numberOfCylinders} \times \text{numberOfSectorsPerTrack} \times \text{numberOfHeads}$$

The format of each Sector Block is as follows:



- uint16 bit_times_sector_to_start_bit;
- uint16 data_bits_after_start_bit;
- uint16 data_in_sector[data_word_count]

The length of the Data Block in 16-bit words, data_word_count, can vary in each sector because it is determined by the emulator measuring the length of the write gate pulse from the controller. A file might be initially created by converting a simH file to an rke file, in which case all Sector Blocks will be the same length. Then if that rke file is loaded into the emulator and some sectors are written by the disk

controller, the sectors written by the disk controller will likely have a slightly different length than the sectors created by the simH conversion software. The number of 16-bit words in the Data Block is based on the data_bits_after_start_bit field value.

If (data_bits_after_start_bit % 16) == 0 then the Data Block length is (data_bits_after_start_bit / 16).
If (data_bits_after_start_bit % 16) != 0 then the Data Block length is (data_bits_after_start_bit / 16) + 1.